A Black-Box Fork-Join Latency Prediction Model for Data-Intensive Applications

Minh Nguyen, Sami Alesawi, Ning Li, Hao Che, *Senior Member, IEEE*, and Hong Jiang, *Fellow, IEEE*

Abstract—The workflows of the predominant datacenter services are underlaid by various Fork-Join structures. Due to the lack of good understanding of the performance of Fork-Join structures in general, today's datacenters often operate under low resource utilization to meet stringent service level objectives (SLOs), e.g., in terms of tail and/or mean latency, for such services. Hence, to achieve high resource utilization, while meeting stringent SLOs, it is of paramount importance to be able to accurately predict the tail and/or mean latency for a broad range of Fork-Join structures of practical interests. In this article, we propose a black-box Fork-Join model that covers a wide range of Fork-Join structures for the prediction of tail and mean latency, called ForkTail and ForkMean, respectively. We derive highly computational effective, empirical expressions for tail and mean latency as functions of means and variances of task response times. Our extensive testing results based on model-based and trace-driven simulations, as well as a real-world case study in a cloud environment demonstrate that the models can consistently predict the tail and mean latency within 20 and 15 percent prediction errors at 80 and 90 percent load levels, respectively, for heavy-tailed workloads, and at any load levels for light-tailed workloads. Moreover, our sensitivity analysis demonstrates that such errors can be well compensated for with no more than 7 percent resource overprovisioning. Consequently, the proposed prediction model can be used as a powerful tool to aid the design of tail-and-mean-latency guaranteed job scheduling and resource provisioning, especially at high load, for datacenter applications.

Index Terms—Tail latency, mean response time, Fork Join queuing networks, datacenter resource provisioning

1 Introduction

Fork-Join structures underlay many datacenter services, including web searching, social networking, and big data analytics. A Fork-Join structure is a critical building block in the job processing workflow that constitutes a major part of job processing time and hardware cost, e.g., more than two-third of the total processing time and 90 percent hardware cost for a Web search engine [1]. In a Fork-Join structure (see Fig. 1), each job in an incoming flow spawns multiple tasks, which are forked to, queued and processed at different nodes, called Fork nodes in this paper, in parallel and its task results are then merged at a Join node to yield the final result. Due to barrier synchronization, the job response time is determined by the slowest task, i.e., the tail probability, which is hard to capture, from both modeling and measurement points of view, making it extremely challenging to predict the job performance, e.g., the job tail latency.

Tail latency is considered to be the most important performance measure for user-facing datacenter applications [2], such as web searching and social networking, and normally expressed as a high percentile job response time, e.g., the 99th percentile response time of 200 ms. Mean latency is also an important performance measure for big data analytics workloads which are generally scale-out by design, involving one or multiple rounds of parallel processing of a (massive) number of tasks and task result merging phases with barrier synchronization, based on, e.g., MapReduce [3] or Spark [4] frameworks. In addition, it is harder but more important[1] to predict the tail and mean latency under heavy load conditions than light ones. This is because as the load becomes heavier, so does the tail distribution, e.g., the 99th percentile of memcached request latencies on a server jumps from less than 1 ms at the load of 75 percent to 1 s at the load of 89 percent [5].

Due to the lack of good understanding of the job-vs-task performance of such workloads, i.e., how distributed task-level performance determines the job-level performance, especially in the high load[2] region, to provide high assurance of meeting tail-latency and/or mean-latency SLOs for such workloads, the current practice is to overprovision resources, which however, results in low resource utilization in datacenters [6], [7]. For example, aggregate CPU and memory utilizations in a 12,000-server Google cluster are mostly less than 50 percent, leaving 50 and 40 percent allocated CPU

• M. Nguyen, N. Li, H. Che, and H. Jiang are with the Department of Computer Science and Engineering, The University of Texas at Arlington, Arlington, TX 76019. E-mail: mqnguyen@mavs.uta.edu, {ning.li, hong.jiang}@uta.edu, hche@cse.uta.edu.
• S. Alesawi is with the Department of Computer Science and Engineering, The University of Texas at Arlington, Arlington, TX 76019, and also with the Faculty of Computing and Information Technology in Rabigh, King Abdulaziz University, Jeddah 21589, Saudi Arabia. E-mail: salesawi@kau.edu.sa.

Manuscript received 29 July 2019; revised 27 Feb. 2020; accepted 3 Mar. 2020. Date of publication 20 Mar. 2020; date of current version 16 Apr. 2020. (Corresponding author: Minh Nguyen.) Recommended for acceptance by jianfeng Zhan. Digital Object Identifier no. 10.1109/TPDS.2020.2982137

1. In the low load region, tail and/or mean latency requirements can be easily satisfied as the available resources are abundant. In contrast, in the heavy load region in which the leftover resource is scarce, resource allocation with high precision must be exercised to meet user requirements.

2. The term "load" can be generally defined as the offered workload per unit time divided by processing capacity per unit time. In the context of Fork-Join structure, it is the maximum of the loads among all the Fork nodes.

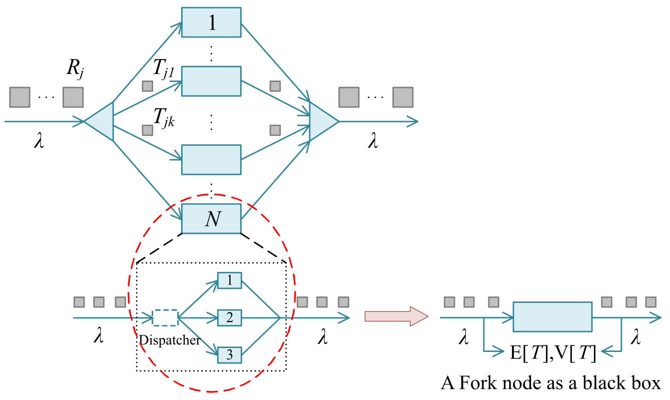


Fig. 1. Black-box Fork-Join model. Each job in the incoming flow spawns k tasks mapped to k out of N Fork nodes. Each Fork node is treated as a black box, completely determined by the mean and variance of the task response time, i.e., $\mathbb{E}[T]$ and $\mathbb{V}[T]$.

and memory resources, respectively, idle almost at all time [6]. Similarly, in a large production cluster at Twitter, aggregate CPU usage is within 20–30 percent even thought CPU reservations are up to 80 percent and aggregate memory usage is mostly within 40–50 percent while memory allocation consistently exceeds 75 percent [7]. Hence, how to improve resource utilization or the load from currently less than 50 percent to, say, 80-90 percent, while meeting stringent SLOs has been a challenging issue for datacenter service providers [7]. To this end, *a key challenge to be tackled is how to accurately capture the tail and mean latency with respect to various Fork-Join structures at high load*.

Fork-Join structures are traditionally modeled by a class of queuing networks, known as Fork-Join queuing network (FJQN) [8], as depicted in Fig. 1. FJQNs are white-box models in the sense that all the Fork nodes are explicitly modeled as queuing systems with given arrival process, queuing discipline, and service time distribution. In this paper, we argue that attempting to use FJQNs to cover a sufficiently wide range of Fork-Join structures of practical interests is not a viable solution. Instead, a black-box solution that can cover a broad range of Fork-Join structures must be sought.

On one hand, FJQNs are notoriously difficult to solve in general. Despite the great effort made for more than half a century, to date, no exact solution is available even for the simplest FJQN where all the nodes are M/M/1 queues [9], i.e., Poisson arrival process and one server with exponential service time distribution. Although empirical solutions for some FJQNs are available, e.g., [10], [11], [12], [13], [14], they can only be applied to a very limited number of Fork-Join structures, e.g., homogeneous case, the case of First-In-First-Out (FIFO) queuing discipline, and a limited number of service time distributions.

On the other hand, *the design space of Fork-Join structures of practical interests* is vast. It encompasses (a) a wide range of queuing disciplines and service time distributions (e.g., both light-tailed and heavy-tailed) [8]; (b) the case with multiple replicated servers per Fork node for failure recovery, task load balancing, and/or redundant task issues for tail cutting [15], [16] or fast recovery from straggling tasks [17]; (c) the case where the number of spawned tasks per job may vary from one job to another [18]; and (d) the case of consolidated services, where different types of services and applications may share the same datacenter cluster resources [19]. Clearly, the existing FJQNs can hardly cover such a design space in practice.

To tackle the above challenges, in this paper, we propose to study a *black-box* Fork-Join model for the prediction of job tail and mean latency, called ForkTail and ForkMean, respectively, to cover a broad range of Fork-Join structures of practical interests. By "black-box", we mean that each Fork node is treated as a black box, regardless of how many replicated servers there are and how tasks are distributed, queued, and processed inside the box. In other words, for a black-box Fork-Join model, one can only use the task statistics measurable from outside of Fork nodes, e.g., the mean and variance of the task response time (see Fig. 1). This is in stark contrast to a white-box Fork-Join model where the exact task queuing discipline and the service model for a Fork node must be known. It also allows the number of spawned tasks per job, k, to be a random integer taking values in $[1, N]$, where N is the maximum number of Fork nodes. As we shall see, our black-box model can indeed adequately covers the above design space.

However, general solutions to this model are unlikely to exist, given the limited success in solving the white-box FJQNs. Nevertheless, we found that for the black-box model, empirical solutions under heavy load conditions do exist, known as the central limit theorem for G/G/m queuing systems, where the arrival process is general with independent interarrival times, the queuing discipline is FIFO, and there are m servers with general service time distributions, under heavy load [20], [21]. Inspired by this theorem, we were able to demonstrate [22] that in a load region of 80 percent or higher, where resource provisioning with precision is most desirable and necessary, an empirical expression of the tail-latency for a special case of the black-box model, i.e., $k = N$ for all the requests, exists, which can predict the tail latencies within 15 percent error at any load levels for light-tailed service time distribution and the load level of 90 percent for heavy-tailed one in the cases (a) and (b) in the design space mentioned above. As our sensitivity analysis in Section 4 shows, such prediction errors can be well compensated for with no more than 7 percent resource overprovisioning.

The work in this paper makes the following contributions. First, it generalizes the solution in [22] to also cover cases (c) and (d) in the design space, hence, making it applicable to most Fork-Join structures of practical interests. Second, it gives the first empirical, universal solutions to tail and mean job latencies for both black-and-white-box FJQNs at high load and hence, it makes a contribution to the queuing network theory as well. In fact, for any white-box FJQN with G/G/1 Fork queuing servers, our approach leads to closed-form approximate solutions, which are on par with the most elaborate white-box solutions in terms of accuracy across the entire load range at much lower computational complexity. Third, comprehensive testing and verification of the proposed approximations for tail and mean latency are performed for all (a)–(d) Fork-Join structures, based on model-based and trace-driven simulation, as well as a real-world case study. Fourth, sensitivity analysis indicates that our proposed solutions can lead to accurate resource provisioning for data-intensive services and applications in a consolidated datacenter environment at high load. Finally, preliminary ideas are provided as to how to use this solution to facilitate SLO-guaranteed job scheduling and resource provisioning.

The rest of the paper is organized as follows. Section 2 introduces our black-box model and ForkTail and ForkMean, the empirical approximations for the tail and mean latency, respectively. Section 3 performs extensive testing of the accuracy of these approximations. Section 4 presents the sensitivity analysis for the proposed approximations. Section 5 explores the range of applicability of the proposed solutions. Section 6 discusses how the proposed approximations may be used to facilitate effective job scheduling and resource provisioning with tail-latency-SLO guarantee. Section 7 reviews the related work. Finally, Section 8 concludes the paper and discusses future work.

2 Model and Solutions

2.1 Black-Box Model

The black-box model described in this section greatly extends the scope of the black-box model introduced in [22] to address the entire design space mentioned in Section 1.

Consider a black-box Fork-Join model with each job in the incoming flow spawning k tasks mapped to k out of N Fork nodes, as depicted in Fig. 1. The results from all k tasks are finally merged at a Join node (i.e., the triangle on the right). Jobs arrive following a random arrival process with average arrival rate λ. Each Fork node may be composed of more than one replicated servers for task-level fault tolerance, load balancing, tail-cutting, and/or straggler recovery. An example Fork node with three server replicas is depicted in Fig. 1.

The above model deals with a general case where $k \leq N$. Note that the traditional FJQNs cover only a small fraction of this design space, i.e., $k = N$, homogeneous Fork nodes with a single server per node, which is modeled as a FIFO queuing system.

General solutions to this model are unlikely to exists. Fortunately, *we are most interested in finding solutions in high load regions where precise resource provisioning is highly desirable and necessary.* There is a large body of research results in the context of queuing performance in high load regions (e.g., see [23] and the references therein). In particular, a classic result, known as the central limit theorem for heavy traffic queuing systems [20], [21], states that for a G/G/m queue under heavy load, the waiting time distribution can be approximated by an exponential distribution. Clearly, this theorem applies to the response time distribution as well, since the response time distribution converges to the waiting time distribution as the traffic load increases. Inspired by this result, we postulate that for tasks mapped to a black-box Fork node and in a high load region, the task response time distribution $F_T(x)$ for any arrival process and service time distribution can be approximated as a generalized exponential distribution function [24], as follows,

$$F_T(x) = (1 - e^{-x/\beta})^{\alpha}, \quad x > 0, \alpha > 0, \beta > 0, \tag{1}$$

where α and β are shape and scale parameters, respectively.

The mean and variance of the task response time are given by [24]

$$\mathbb{E}[T] = \beta[\psi(\alpha + 1) - \psi(1)], \tag{2}$$

$$\mathbb{V}[T] = \beta^2[\psi'(1) - \psi'(\alpha + 1)], \tag{3}$$

where $\psi(.)$ and its derivative are the digamma and polygamma functions.

From Eqs. (2) and (3), it is clear that the distribution in Eq. (1) is completely determined by the mean and variance of the task response time. In other words, the task response time distribution can be measured by treating each Fork node as a black box as shown in Fig. 1. The rationale behind the use of this distribution, instead of the exponential distribution, is that it can capture both heavy-tailed and light-tailed task behaviors depending on the parameter settings and meanwhile, it degenerates to the exponential distribution at $\alpha = 1$ and $\mathbb{E}[T] = \beta$. In [22], we showed that this distribution significantly outperforms the exponential distribution in terms of tail latency predictive accuracy.

Now, with all the Fork nodes in Fig. 1 being viewed as black boxes, the response time distribution for any job with k tasks can be approximated using the order statistics [9] as follows,

$$F_X^{(k)}(x) = \prod_{i=1}^{k} F_{T_i}(x) = \prod_{i=1}^{k} (1 - e^{-x/\beta_i})^{\alpha_i}. \tag{4}$$

Note that the above expression is exact if the response times for tasks mapped to different Fork nodes are independent random variables. This, however, does not hold true for any Fork-Join structures, simply because the sample paths of the task arrivals at different Fork nodes are exactly the same, not independent of one another. This is the root cause that renders the Fork-Join models extremely difficult to solve in general. In what follows, we introduce ForkTail and ForkMean, separately, based on this approximation.

2.2 ForkTail

ForkTail was originally presented in [25]. Our postulation is that as load reaches 80 percent or higher where precise resource provisioning is desirable and necessary, the tail-latency prediction errors introduced by the above assumption will become small enough for resource provisioning purpose. Our extensive testing results in this paper provide strong support of the postulation, making our modeling approach the only practically viable one for tail latency prediction.

Tail latency x_p, defined as the pth percentile job response time, can be written as,

$$x_p = F_X^{(k)^{-1}}(p/100). \tag{5}$$

Eq. (5) simply states that in a high load region, the tail latency can be approximated as a function of the means and variances of task response times for all k tasks at their corresponding Fork nodes, irrespective of what workloads cause the heavy load. The implication of this is significant. It means that this expression is applicable to a consolidated datacenter cluster where more than one service/application share the same cluster resources. Moreover, this expression allows tail latency to be predicted using a limited number of job samples thanks to its dependence on the first two moments of task response times only, i.e., the means and variances.

The results so far is general, applying to the heterogeneous case, where task response time distributions may be different from one task to another, due to, e.g., the use of heterogeneous Fork nodes and/or uneven background workloads. As a result, the tail latency predicted by Eq. (5) may be different from one job to another or even for two identical jobs, as long as their respective Fork nodes do not completely coincide with one another, or they are issued at different times. In other words, Eq. (5) is a fine-grained tail latency expression. For certain applications, such as offline resource provisioning (see Section 6 for explanations) and coarse-grained, per-service-based tail-latency prediction, one may be more interested in the homogeneous case only. In this case, the response time distribution can be further simplified as,

$$F_X^{(k)}(x) = (1 - e^{-x/\beta})^{k\alpha}. \tag{6}$$

This is because the means and variances given in Eqs. (2) and (3) are the same for the homogeneous case. A coarser-grained cumulative distribution function (CDF) of the job response time can then be written as,

$$F_X(x) = \sum_{k_i} F_{X|K}(x|k_i)P(K = k_i), \tag{7}$$

where $F_{X|K}(x|k_i)$ is the conditional CDF of the job response time for jobs with k_i tasks, given by Eq. (6), i.e., $F_{X|K}(x|k_i) = F_X^{(k_i)}(x)$, and $P(K = k_i) = P_i$ is the probability that a job spawns k_i tasks.

Further assume that there are m job groups with distinct numbers of tasks k_i's, $i = 1, \ldots, m$, and corresponding probabilities P_i's. We then have,

$$F_X(x) = \sum_{i=1}^{m} P_i \cdot F_X^{(k_i)}(x). \tag{8}$$

Correspondingly, the tail latency for the m groups of jobs as a whole can then be readily obtained, similar to Eq. (5), as follows,

$$x_p = F_X^{-1}(p/100). \tag{9}$$

For example, the tail latency for a given service can be predicted by collecting statistics for k_i's and P_i's, as well as mean and variance of task response time and applying them to the tail latency expression in Eq. (9).

2.2.1 Application to White-Box FJQNs

Clearly, the above black-box approach leads to closed-form solutions for any white-box models whose analytical expressions for the means and variances of task response times are available, whether it is homogeneous or not. In fact, our solution works for the case where different Fork nodes may have different service time distributions and queuing disciplines. For instance, our approach can be applied to a large class of FJQNs, where each Fork node is an M/G/1 queue or a more general G/G/1 queue, whose mean and variance of the task response time can be computed from Takács recurrence theorem [26] or the queuing network analyzer [27], respectively.

2.3 ForkMean

While the approximations in Eqs. (5) and (9) work well for the job tail latency even for the $k < N$ cases, it fails to accurately predict the job mean response time,[3] yielding more than three times larger errors for the same cases studied, especially for the case of light-tailed service time distributions. We find that the reason for this to happen is due to the fact that to accurately predict the job mean response time, the entire job response time distribution including the tail portion must be accurately captured, as the barrier synchronization tends to push the job mean response time towards the tail part of the task response time distribution, as the workload scales out.

On the basis of the above modeling, this section aims at finding solutions to reduce the prediction errors for the job mean latency. To this end, we make the following two key observations.

Observation 1. For a wide range of Fork-Join models, the difference between the exact tail-mean ratio and the model-based tail-mean ratio, derived from the CDF in Eq. (4), hereafter called the *gap* and denoted as Δ, converges to a constant as the number of Fork nodes becomes large enough. Mathematically, we have,

$$\frac{x_p}{x_m} - \frac{x_p^{\text{ge}}}{x_m^{\text{ge}}} = \Delta, \tag{10}$$

where x_p and x_m are the exact pth percentile and mean of job latency, respectively, which can be estimated by experiments, while x_p^{ge} and x_m^{ge} are derived from the prediction model, i.e., Eq. (4). Hence, the mean latency can be approximated as follows,

$$x_m = \frac{x_p}{R^{\text{ge}} + \Delta} \approx \frac{x_p^{\text{ge}}}{R^{\text{ge}} + \Delta}\ , \tag{11}$$

where $x_p \approx x_p^{\text{ge}}$ at high loads, since ForkTail give accurate predictions for the pth percentile at high loads, as indicated in the testing results, and $R^{\text{ge}} = x_p^{\text{ge}}/x_m^{\text{ge}}$.

Fig. 2 illustrates the gaps for systems with different task service time distributions, including light-tailed and heavy-tailed ones, where each Fork node is a single server, i.e., without replication. As one can see, the gap converges to a constant as N becomes sufficiently large, say, $N \geq 100$, for all the cases. Similar trends are also observed for the systems with 3-replica Fork nodes with Round-Robin and redundant-task-issue dispatching policies as well as the systems with variable numbers of forked tasks (not shown here).

Observation 2. There is a strong correlation between the tail heaviness of service time distribution and the gap Δ, i.e., the heavier the tail, the smaller the gap. It is evident from Fig. 2 that the light-tailed distributions, including Exponential and Weibull, have larger gaps than the heavy-tailed ones, including the truncated Pareto and empirical (defined in Section 3.1). With this observation, we make the following postulation: The gap is much more of a function of the tail heaviness of a service time distribution than the service time distribution itself.

From the above observations, we propose two empirical solutions, one is white-box and the other black-box, for the approximation of the gap, Δ, and hence, the job mean response time.

3. We use the terms 'latency' and 'response time' interchangely in this paper.

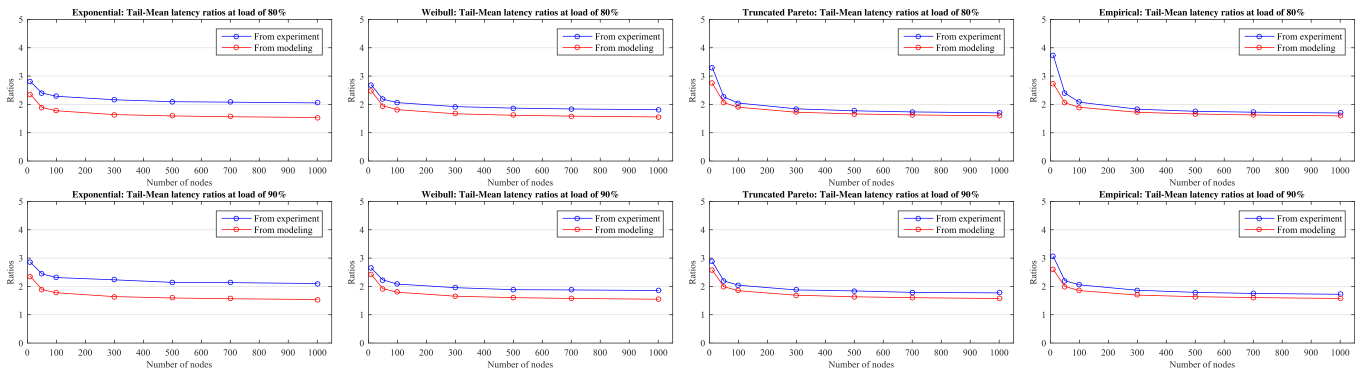


Fig. 2. The gaps for Fork-Join systems with different service time distributions at load levels of 80 percent (upper row) and 90 percent (lower row).

2.3.1 White-Box Approach

This approach is based on the above postulation. Here we consider a homogeneous white-box Fork-Join queuing model where each Fork node can be modeled as a G/G/1 queue. With known interarrival and service time distributions, one can find the job response time distribution and the corresponding tail and mean latencies, and so their ratio R^{ge}, from ForkTail. So to find the job mean latency, x_m, all that is left to be done is to find Δ.

To this end, we first define tail heaviness, $w(F_T)$. We use Right Quantile Weight [28] which measures the tail heaviness on the right side of a distribution, the region of interest in all of our experiments. This tail weight measure is defined as,

$$w(F_T) = \frac{F_T^{-1}\left(\frac{1+q}{2}\right) + F_T^{-1}\left(1-\frac{q}{2}\right) - 2F_T^{-1}(0.75)}{F_T^{-1}\left(\frac{1+q}{2}\right) - F_T^{-1}\left(1-\frac{q}{2}\right)}, \tag{12}$$

where $0.5 < q < 1$ and $F_T^{-1}(q)$ is quantile q of task service time distribution F_T. To capture the tail effect but still retain a reasonable robustness, we set $q = 0.99$.

Based on our postulation, $\Delta = \Delta(\rho, w)$, independent of $F_T(x)$. Here ρ is the load. In other words, as long as $w(F_T^{(1)}) = w(F_T^{(2)})$, the two homogeneous Fork-Join models with different service time distributions, $F_T^{(1)}$ and $F_T^{(2)}$, respectively, will have the same gap. In other words, if one can find the function, $\Delta(\rho, w)$, using one distribution function with different tail weights, this $\Delta(\rho, w)$ can then be used by any Fork-Join models with other distribution functions to find the gap. In this paper, we use the generalized exponential distribution in Eq. (1) at different coefficients of variance to generate different tail weights from Eq. (12) and the corresponding gaps and then use nonlinear regression to find $\Delta(\rho, w)$. Table 1 shows the gaps for different tail weights, averaged over $N = 100$ to 1,000 at three different load levels.

From experimental data with different distribution parameters, we found that the power function, i.e., $\Delta = aw^b + c$, yields a very good fit to these gap-tail weight points. Fig. 3 illustrates the fitted curve at load level of 80 percent from Table 1 with respect to the fitted points from the generalized exponential distribution (the black points). It also shows the actual points from other distributions, which are used for testing in the experiments (the green points), relative to the fitted curve. As one can see, the green points stay reasonably close to the curve itself, meaning that our postulation indeed holds true. Table 2 presents the fitted functions for the cases in Table 1.

In summary, this white-box approach results in a closed-form solution for the approximation of job mean latency, which is composed of the following computation steps,

- With given $\mathbb{E}[T]$ and $\mathbb{V}[T]$, compute the tail and mean latencies, i.e., x_p^{ge} and x_m^{ge} from the predicted CDF in Eq. (4) and their corresponding ratio, i.e., R^{ge};
- With a given service time distribution F_T, calculate the tail weight w from Eq. (12), which is then mapped to a Δ at a given load, e.g., using one of the functions in Table 2;
- Approximate the mean latency using Eq. (11).

TABLE 1
The Gaps for Different Tail Heavinesses and Load Levels

Load	Tail weight						
	0.703	0.772	0.851	0.918	0.962	0.986	0.999
75%	0.486	0.271	0.160	0.097	0.063	0.029	0.009
80%	0.511	0.283	0.169	0.106	0.069	0.044	0.013
90%	0.573	0.319	0.190	0.129	0.070	0.055	0.023

2.3.2 Black-Box Approach

The white-box approach above leads to closed-form solutions for homogeneous white-box Fork-Join models with known

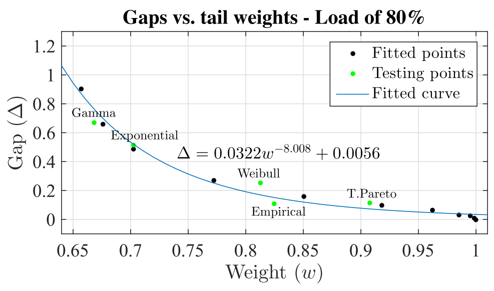


Fig. 3. An example of the gap-vs-tail-weight fitted curve.

TABLE 2
Examples of Fitted $\Delta(\rho, w)$ Curves

Load	Function
75%	$\Delta = 0.0371w^{-7.517} - 0.0052$
80%	$\Delta = 0.0322w^{-8.008} + 0.0056$
90%	$\Delta = 0.0274w^{-8.654} + 0.0284$

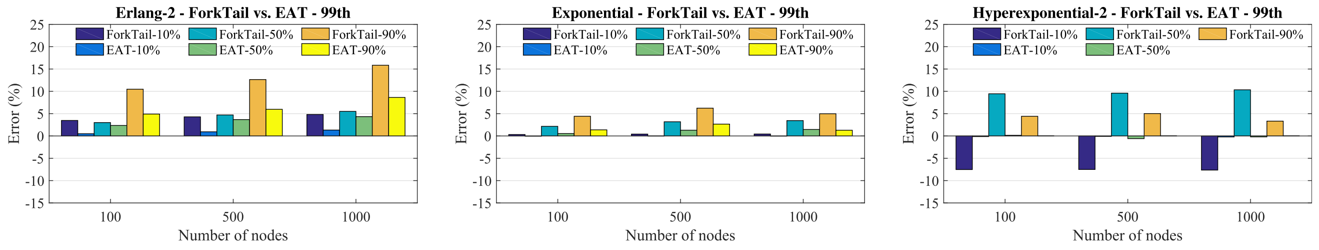


Fig. 4. Prediction errors for the 99th percentile response times for ForkTail and EAT.

service time distributions for Fork nodes. However, in practice, determining those distributions is nontrivial, e.g., for systems with multi-replica Fork nodes. Hence, it is necessary to seek a black-box solution applicable to a wide range of Fork-Join structures of practical interests.

Based on the Observation 1, i.e., Δ converges to a constant as the number of Fork nodes becomes large enough, i.e., around 100, based on all the testing cases. This suggests that, if for a target application, Δ can be measured on a small testbed or by simulation, with 100 virtual machines/nodes, or equivalently, a few commodity servers, e.g., 5, then the mean latency can be predicted when the application is deployed on a much larger number of nodes. This approach requires only the means and variances of task response times as inputs, and hence is a hybrid, black-box solution.

The steps taken to find the job mean latency are similar to those for the white-box approach above except for step 2 where Δ is predicted by running experiments for the target application on a system with a given number of Fork nodes, e.g., 100, and measure the ratio gap between the results from the experiments and the prediction model.

Compared to the white-box solution, the black-box one is simpler and can be applied to a much wider range of Fork-Join structures. However, as a hybrid approach, it requires to run experiments, either via simulation or on a real testbed, with an adequate number of Fork nodes, e.g., 100. Consequently, it should be applied to large-scale systems where a job is forked to at least hundreds of nodes, much larger than the one used for testing. Note that the hybrid approach, which combines analysis and simulation, is not unusual in analyzing performance of the Fork-Join model. Indeed, it has been used in several previous works in the literature [10], [13], [29].

3 Validation

3.1 Tail Latency Prediction Validation

In this section, ForkTail is extensively validated against the results from model-based simulation, trace-driven simulation, and a case study in Amazon EC2 cloud. The validation is performed for the systems with $k = N$, $k \leq N$, and consolidated services, separately. The accuracy of the prediction is measured by the relative error between the value predicted from ForkTail, t_p, and the one measured from simulation or real-system testing, t_m, i.e.,

$$error = \frac{100(t_p - t_m)}{t_m}.$$

3.1.1 *Case 1:* $k = N$

A notable example for this case is Web search engine [30] where a search request looks up keywords in a large inverted index distributed on all the servers in the cluster. We validate ForkTail with three testing approaches, i.e., white-box and black-box model-based testing as well as a real-world case study in Amazon EC2 cloud.

White-Box Model-Based Validation. Here we study the accuracy of ForkTail when applied to homogeneous, single-queuing-server-Fork-node Fork-Join systems with the assumption that the service time distribution is known in advance, the approach taken in all the existing works on performance analysis of FJQNs [9]. The tail latency prediction involves the following steps:

- Find the mean and variance of task response times with the given task service time distribution;
- Substitute the above mean and variance into Eqs. (2) and (3), respectively, and solve that system of equations to find the scale and shape parameters of the generalized exponential distribution in Eq. (1), which is then used to approximate the task response time distribution;
- Calculate the pth percentile of request response times from Eq. (9).

First, we compare ForkTail against the state-of-the-art tail latency approximation for *homogeneous* FJQNs [14], known as *EAT*, which is derived from analytical results for single-node and two-node systems. Fig. 4 shows the comparative results for three service time distributions studied in [14], i.e., Erlang-2, Exponential, and Hyperexponential-2, at the loads of 10, 50, and 90 percent[4] and numbers of nodes of 100, 500, and 1,000.

EAT provides more accurate (from a few to several percentage points) approximations for the 99th percentiles of response times across all the cases studied. Much to our surprise, our approach yields most of the errors within 10 percent, across the entire load range. Although outperforming our approach, EAT has its limitations. First, it can be applied only to homogeneous FJQNs where each node can be generally modeled as a MAP/PH/1 queuing system, i.e., Markovian arrival processes and phase-type service time distribution with one service center. Second, the method requires the service time distribution to be known in advance and converted into a phase-type distribution, which is nontrivial, especially for heavy-tailed distributions [31]. Third, the method may incur high computational complexity, depending on the selection of a constant C, whose value determines the computational runtime and prediction accuracy. It takes

4. For EAT, the case for Hyperexponential-2 at the load of 90 percent is not available, due to a numerical error running the code provided in [14].

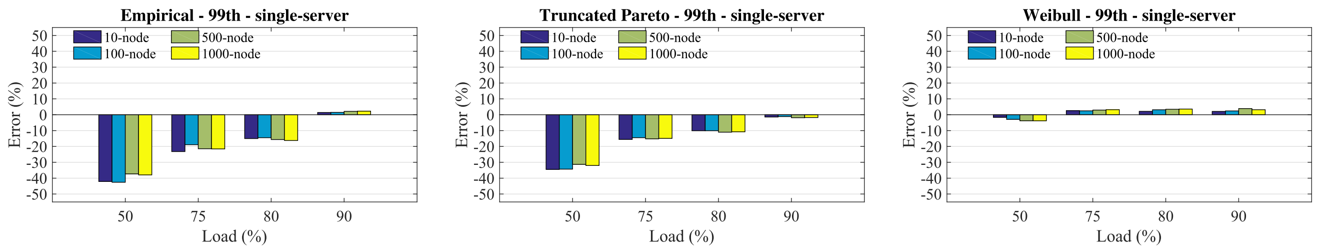


Fig. 5. Prediction errors of the 99th percentile response times for white-box systems with single-server Fork nodes.

at least 2 seconds on our testing PC (Core i7-4940MX Quad-core, 32GB RAM) to get the resulting percentiles even at the lesser degree of accuracy with $C = 100$ (more than 300 seconds at $C = 500$). In contrast, our method takes less than 5 milliseconds to compute the required percentiles. As a result, similar to other existing white-box solutions, EAT has limited applicability for datacenter job scheduling and resource provisioning in practice.

To cover a sufficiently large workload space, we further consider service time distributions with heavy tails, which are common in practice [32] and cannot be easily dealt with by EAT, including the following,

- Empirical distribution measured from a Google search test leaf node provided in [32], which has a mean service time of 4.22 ms, a coefficient of variance (CV) of 1.12, and the largest tail value of 276.6 ms;
- Truncated Pareto distribution [31] with the same mean service time and a CV of 1.2, whose CDF is given by,

$$F_S(x) = \frac{1-(L/x)^{\alpha}}{1-(L/H)^{\alpha}} \qquad 0 \le L \le x \le H, \tag{13}$$

  where α is the shape parameter; L is the lower bound; and H is the upper bound, which is set at the maximum value of the empirical distribution above, i.e., $H = 276.6$ ms, resulting in $\alpha = 2.0119$ and $L = 2.14$ ms.
- Weibull distribution [8], also with the same mean service time and a CV of 1.5, whose CDF is defined as,

$$F_S(x) = 1 - \exp[-(x/\beta)^{\alpha}] \qquad x \ge 0, \tag{14}$$

  where $\alpha = 0.6848$ and $\beta = 3.2630$ are shape and scale parameters, respectively.

Fig. 5 presents the prediction errors for the 99th percentile response times for the above cases. The Weibull distribution, which is less heavy-tailed, consistently yields smaller errors, well within 5 percent, for the entire load range studied, similar to the light-tailed distribution cases studied earlier. The empirical and truncated Pareto distributions, which are more heavy-tailed, provide good approximations for the 99th percentiles at the load of 80 percent or higher, which is well within 17 and 5 percent at the load of 80 and 90 percent, respectively, agreeing with our postulation.

We also consider the cases with general arrival process and general service time distribution, i.e., G/G/1 Fork nodes. Fig. 6 shows the prediction errors for example cases with Erlang-2 (CV = 0.5) and Hyperexponential-2 (CV = 1.2) arrival processes and Truncated Pareto service time distribution (CV = 3.0). Again, ForkTail yields quite accurate approximations for tail latency at high load regions, i.e., above 75 percent. The prediction results also show the same trend for Weibull and Exponential service time distributions, which are not shown here.

Black-Box Model-Based Validation. We now validate ForkTail without making assumption on the service time distribution at each Fork node. We treat each Fork node as a black-box and empirically measure the mean and variance of task response times at each given arrival rate λ or load. These measures are then substituted into Eqs. (2) and (3), respectively, to find the shape and scale parameters, which are in turn used to predict the tail latency based on Eq. (9).

For all the three heavy-tailed FJQNs studied above, we consider two types of Fork nodes, i.e., one with single server and the other with three replicated servers. For the one with three servers, we explore two task dispatching policies. The first policy is the Round-Robin (RR) policy, in which the dispatcher will send tasks to different server replicas in an RR fashion. The second policy is still RR, but it also allows redundant task issues, a well-known tail-cutting technique [15], [16]. This policy allows one or more replications of a task to be sent to different server replicas in the Fork node. The replications may be sent in predetermined intervals to avoid overloading the server replicas. In our experiments, at most one task replication can be issued, provided that the original one does not finish within 10 ms, which is around the 95th percentile of the empirical distribution above.

Figs. 7, 8, and 9 present the prediction errors at different load levels and N's for the 99th percentile response times for all three FJQNs with single server and three servers per Fork node, respectively. First, we note that the prediction errors for the cases in Fig. 7 are very close to those in Fig. 5. This is expected as the white-box and black-box results, ideally, should be identical. The differences are introduced due to simulation and measurement errors. Second, the prediction performances of the cases with three replicas and the RR policy in Fig. 8 are also very close to those of the cases in Fig. 7, with errors being well within 20 and 10 percent at the

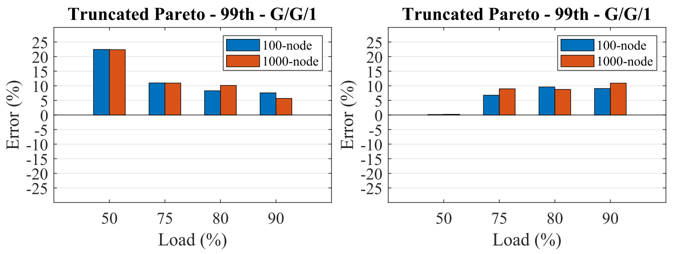


Fig. 6. Prediction errors of the 99th percentile response times for white-box systems with Erlang-2 (left) and Hyperexponential-2 (right) arrival distributions and Truncated Pareto service time distribution.

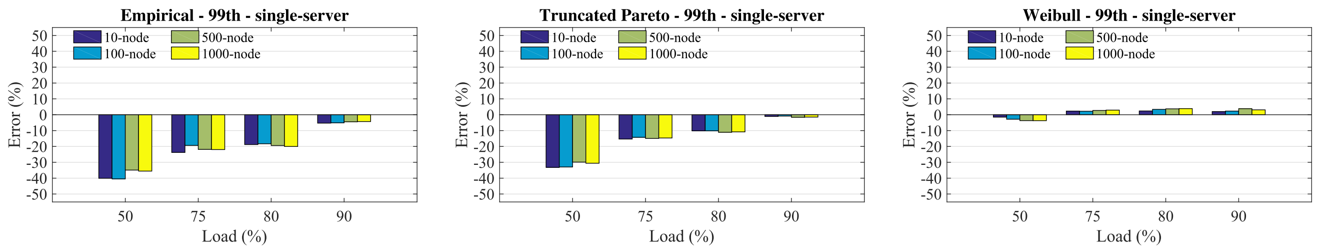


Fig. 7. Prediction errors of the 99th percentile response times for black-box systems with single-server Fork nodes.

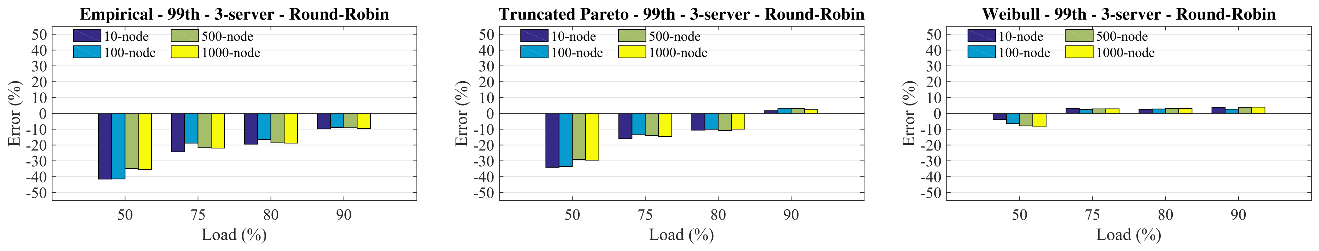


Fig. 8. Prediction errors of the 99th percentile response times for black-box systems with 3-server Fork nodes and Round-Robin policy.

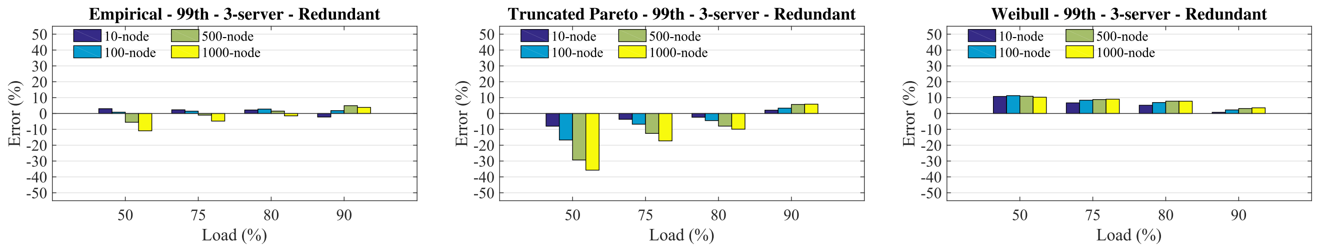


Fig. 9. Prediction errors of the 99th percentile response times for black-box systems with 3-server Fork nodes and redundant-task-issue policy.

loads of 80 and 90 percent, respectively, for all the case studies, further affirming our postulation. The two scenarios have similar performance because they are compared at the same load levels, where the RR policy in the second scenario simply balances the load among three replicas, making each virtually identical to the single-server scenario. In contrast to these two scenarios, Fig. 9 shows that with the application of the tail-cutting technique, the prediction errors are substantially reduced, with less than 10 percent at the load of 80 percent or higher. This is consistent with the earlier observation, i.e., the lighter the tail, the smaller the prediction errors. This suggests that the tail-cutting techniques, often utilized in datacenters to curb the tail effects, can help expand the load ranges in which ForkTail can be applied.

A Case Study in Cloud. We also assess the accuracy of ForkTail for a real case study in Amazon EC2 cloud. We implement a simple Unix grep-like program on the Apache Spark framework (version 2.1.0) [4]. It looks up a keyword in a set of documents and returns the total number of lines containing that keyword, as depicted in Fig. 10. The cluster for the testing includes one master node using an EC2 c4.4xlarge instance and 32 or 64 worker nodes using EC2 c4. large instances. We use a subset of the English version of Wikipedia as the document for lookup. Each worker node holds a shard of the document whose size is 128 MB, corresponding to the default block size on Hadoop Distributed File System (HDFS) [33]. A client, which runs a driver program, sends a flow of keywords, each randomly sampled from a pool of 50K keywords, to the testing cluster for lookup. Each worker searches through its corresponding data block to find the requested keyword and counts the number of lines containing the keyword. The line count is then sent back to the client program to sum up. Clearly, this testing setup matches the black-box model.

We measure the request response time, i.e., the time it takes to finish processing each keyword at the client. We also collect the task response times, composed of the task waiting time and task service time. The task waiting time is the one between the time the request the task belongs to is sent to the cluster and the time the task is sent to a given worker for processing. This is because in the Spark framework, all the tasks spawned by a request are kept in their respective

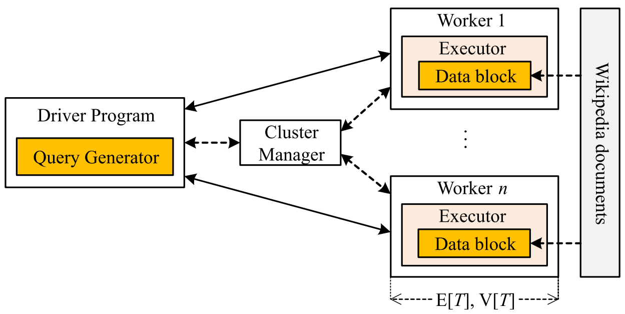


Fig. 10. Experiment setup in Amazon EC2 cloud. Each worker should be viewed as a blackbox as in Fig. 1.

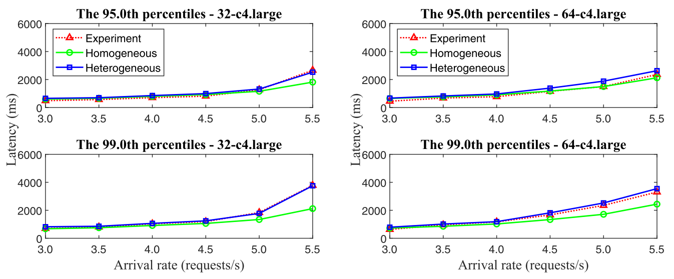


Fig. 11. Predicted tail latencies for keyword occurrence counts in Amazon cloud with 32 (left) and 64 (right) nodes.

virtual queues corresponding to their target workers centrally. A task at the head of a virtual queue cannot be sent to its target worker until the worker becomes idle. Hence, to match our black-box model, the task response time must include the task waiting time, i.e., the task queuing time plus the task dispatching time, and the task service time, which is the actual processing time at the worker the task is mapped to. From the collected samples, we compute the means and variances of task response times, which are in turn used to derive the task response time distribution as in Eq. (1).

Ideally, the task response time distributions for all the tasks are the same, given that the workers are identical. In other words, one would expect that this case study is homogeneous. However, our measurement indicates otherwise. A careful analysis reveals that this is mainly due to the task scheduling mechanism in the Spark framework. Each data block has three replicas distributed across different workers. By default, the placement preference is to send a task to an available worker where the data block resides. Unfortunately, as the request arrival rate or load increases, more tasks are mapped to workers that do not hold the required data blocks for the tasks, causing long task response time due to the need to fetch the required data blocks from the distributed file system. This results in higher variability in the task response time distributions among different workers. Therefore, the heterogeneous model given in Eq. (4) is found to be more appropriate in high load regions.

The above observation is confirmed by the experimental results, presented in Fig. 11. As one can see, the heterogeneous model (the blue lines) gives quite accurate prediction for both 95th and 99th percentiles at both $N = 32$ and 64 cases, while the prediction from the homogeneous model (the green lines) gets worse as the load becomes higher. Based on the heterogeneous prediction, the prediction errors at both $N = 32$ and 64 and the 99th percentile are well within 10 percent in a high load region, i.e., 60 percent or higher. Note that the load here is measured in terms of request arrival rate. Since the system is heterogeneous, we estimated the equivalent loads corresponding to different arrival rates

TABLE 3
Estimated Loads (%) for the Testbed Based on Request Arrival Rates

#workers	Request arrival rates (requests/s)					
	3.0	3.5	4.0	4.5	5.0	5.5
32	48.33	56.39	64.44	72.50	80.56	88.61
64	50.04	58.38	66.72	75.06	83.40	91.74

based on the maximum value of means of task service times across all the workers, as given in Table 3.

Finally, we note that to achieve a reasonably good confidence of measurement accuracy for the 99th percentile tail latency, we collected 80K samples in our experiments at the maximum possible sampling rate equal to the average request arrival rate of 5.8 per second, which translates into a measurement time of 13,793 seconds or about 4 hours. It takes even more time to run the experiments at lower arrival rates. The average runtime across all the request arrival rates in the experiments is about 6 hours. Due to the costly cloud services, we have to limit our experiments to 64 worker nodes.

This example clearly demonstrates that it can be expensive and time consuming, if practical at all, to estimate tail latency based on direct measurement. In contrast, ForkTail is able to do so with far fewer number of samples at much lower cost. For example, with 800 samples collectable in less than three minutes, we can estimate the response-time means and variances for all the tasks and hence the tail latency with reasonably good accuracy. This means that our prediction model can reduce the needed samples or prediction time by two orders of magnitude than the direct measurement.

3.1.2 Case 2: Variable Number of Tasks $k \leq N$

Notable examples for this case are key-value store systems in which a key lookup may touch only a partial number of servers and web rendering which requires to receive web objects or data from a group of servers in a cluster.

In this case study, we assess the accuracy of our prediction model (i.e., Eqs. (8) and (9)) for applications whose jobs may spawn different numbers of tasks with distribution $P(K = k_i)$. Specifically, we study two scenarios where $P(K = k_i)$ is nonzero for a specific value of K and uniformly distributed, respectively. We further consider three different service time distributions: two heavy-tailed ones, the empirical and truncated Pareto as in Section. 3.1.1, and a light-tailed exponential distribution, with the same mean service time, i.e., 4.22 ms.

Scenario 1: Fixed Number of Tasks per Job. In this scenario, we consider the cases when the number of forked tasks per job is a fixed number k ($k \leq N$), i.e., every incoming job is

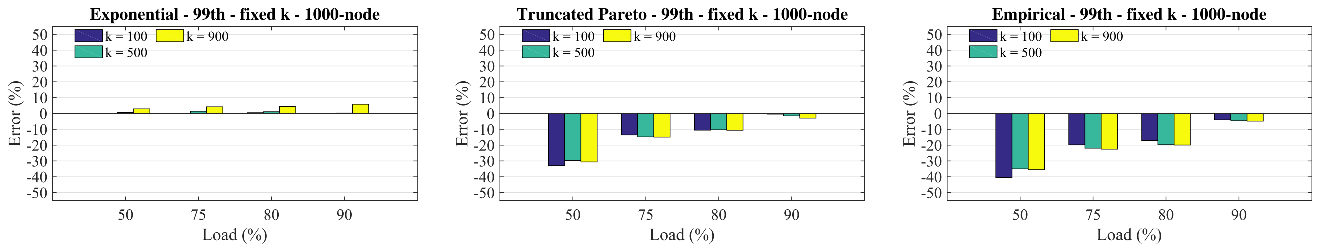


Fig. 12. Prediction errors of the 99th percentile response times for an 1000-node cluster when the number of tasks per job is fixed ($k = 100, 500, 900$).

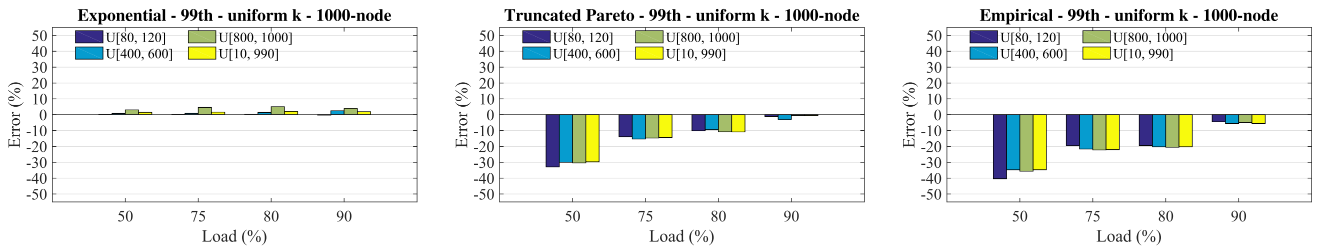


Fig. 13. Prediction errors of the 99th percentile response times for an 1000-node cluster when the number of tasks per job is uniformly distributed.

split into exactly k tasks which are dispatched to k randomly selected Fork nodes in an N-node cluster.

Fig. 12 shows prediction errors for the 99th percentile response times for an 1,000-node cluster with $k = 100$, 500, and 900 tasks. ForkTail provides good prediction in high load regions, with all the errors within 10 percent at the load of 90 and 20 percent at the load of 80 percent for all the cases studied. The case with the light-tailed exponential distribution gives quite accurate prediction for the entire range under study, i.e., all within 6 percent.

Scenario 2: Uniform Distribution. Here we deal with cases when an incoming job is forked to k random nodes in the cluster where k is randomly sampled from an integer range $[a, b]$, i.e., $k_i \in \{a, a+1, \ldots, b-1, b\}$ with probability $P_i = P = 1/m \forall i$, where $m = b - a + 1$. Therefore, the mean number of tasks is $(a+b)/2$.

Fig. 13 presents prediction errors for an 1,000-node cluster with k in four different ranges, i.e., [80, 120], [400, 600], [800, 1000], and [10, 990]. The results again show that ForkTail yields good approximations for the 99th percentile job response times when the system is under heavy load, i.e., 80 percent or higher. Furthermore, again for all the cases with the exponential distribution, ForkTail gives accurate predictions across the entire load range studied.

The above prediction model applies to the case where a single tail-latency SLO is imposed on a service or application as a whole, a practice widely adopted in industry. However, this practice can be too coarse grained. To see why this is true, Table 4 provides the predicted tail latencies for some given jobs with distinct k values in a cluster of size 1,000 and at the load of 90 percent. As one can see, the 99th percentile tail latencies for jobs at different k's can be drastically different, e.g., the 10-task and 900-task cases. This suggests that even for a single application, finer grained tail latency SLOs may need to be enforced to be effective, e.g., enforcing tail-latency SLOs for job groups with each having k's in a small range. Table 5 shows that ForkTail can indeed provide accurate, finest-grained prediction at given k's, i.e., all well within 10 percent at load of 90 percent.

TABLE 4
The Predicted 99th Percentile of Latencies (ms)

Distribution	Number of forked tasks				
	10	400	500	600	900
Exponential	291.32	446.97	456.38	464.08	481.19
Truncated Pareto	448.83	705.45	720.97	733.66	761.87
Empirical	391.27	616.22	629.83	640.95	665.68

TABLE 5
Errors in the 99th Percentile Prediction When Tracking Jobs With a Given Number of Tasks at Load of 90 percent

Distribution	Number of nodes				
	10	400	500	600	900
Exponential	−0.861	0.052	0.433	0.647	2.791
Truncated Pareto	−0.571	−0.403	1.763	−0.489	−1.433
Empirical	−2.814	−6.929	−6.239	−5.322	−6.541

3.1.3 Case 3: Consolidated Services

In this case study, we evaluate the accuracy of ForkTail when applied to the consolidated datacenter where multiple applications, including latency-sensitive user-facing and background batch ones, share cluster resources as illustrated in Fig. 14. We conduct a trace-driven simulation based on a trace file derived from the Facebook 2010 trace, a widely adopted approach in the literature to explore datacenter workloads [19], [34], [35]. We test the accuracy of ForkTail in capturing the tail latency for a given *target application*.

Workload. The trace file is generated based on the description of the Facebook trace in some previously published works [19], [34], [35]. Specifically, we first generate the number of tasks for job arrivals based on the distribution of the job size in terms of the number of tasks per job, as suggested in [35]. It includes nine bins of given ranges of the number of tasks and corresponding probabilities, assuming that the number of tasks is uniformly distributed in the range of each bin. We then generate the mean task service time based on the Forked task processing time information in [34]. Individual task times are drawn from a Normal distribution with the generated mean and a standard deviation that doubles the mean as in [19]. The resulting trace file contains a total of two million requests, each including the following information: request arrival time, number of forked tasks, mean task service time, and the service times of individual forked tasks.

In the experiments, the jobs in the trace file serve as the background workloads, which are highly diverse, involving

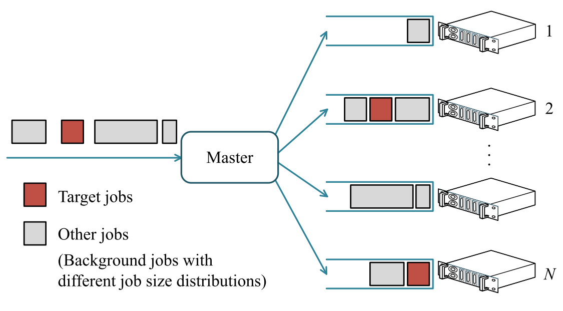
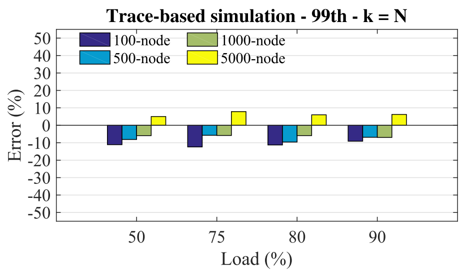


Fig. 14. Consolidated applications running on a cluster.



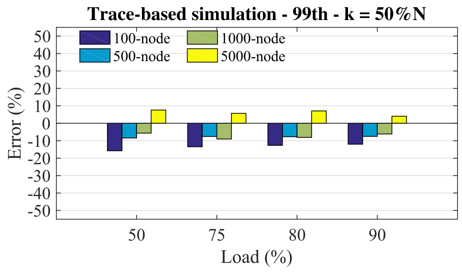


Fig. 15. Prediction errors of the 99th percentile target response times in a consolidated workload environment when the tasks of each target job reach all the nodes (top) and randomly reach 50 percent number of nodes (bottom) in the cluster.

a wide range of applications with mean service times ranging from a few milliseconds to thousands of seconds. The target jobs are generated at runtime using the same approach the trace file is generated. The only difference is that the target jobs are statistically similar with the same mean service time, to mimic a given application or simply a group of jobs with similar statistic behaviors. For each simulation run, a predetermined percentage, e.g., 10 percent, of target jobs are created and fed into the cluster at random.

Simulation Settings and Results. In the simulation, the target and background jobs are set at 10 and 90 percent of the total number of jobs, respectively. We evaluate two cases, one with the number of tasks per target job set at one half of the cluster size and the other the same as the cluster size. The tests cover multiple cluster sizes, i.e., 100, 500, 1,000, and 5,000 nodes with each having three replicated servers. All the cases are homogeneous.

The prediction errors for the 99th percentiles of target response times for the two case studies at loads of 50, 75, 80, and 90 percent are shown in Fig. 15. As one can see, the prediction errors are within 15 percent for all the cases studied.

Finally, we note that although the validations for tail latency prediction are exclusively focused on the 99th-percentile tail latency, ForkTail offers similar and consistent performance at higher percentiles, which are not shown here due to the lack of space.

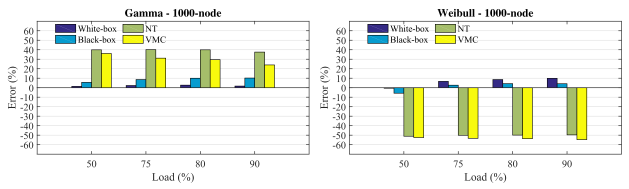


Fig. 17. Comparison of percentage errors in mean latency approximations with M/G/1 queues for Gamma and Weibull service time distributions.

3.2 Mean Latency Prediction Validation

In this section, we extensively validate the predicted mean latencies from ForkMean, for both white-box and black-box approaches, against the results from the existing white-box solutions, the event-driven simulation experiments, and a case study on Amazon EC2 as in Section 3.1.

3.2.1 *Scenario 1: Single-Server Queues*

In this scenario, we compare ForkMean with some well-known closed-form approximations, including NT [10], VMC [36], and VM [37].

Fig. 16 shows the comparison for the systems with 50, 1,000, and 5,000 nodes, each modeled as an M/M/1 queue, at load levels of 50, 75, 80, and 90 percent. Overall, the NT approximation is the most accurate one. The white-box ForkMean yields errors within 5 percent for all the cases studied, which are close to those of the NT approximation. The black-box one that is based on the measured Δ's at 100 node also gives good approximations to mean latency even for the case of 50 nodes, with errors within 10 percent for all the cases. Note that, due to its high computational complexity, the VM approximation is not included in the cases of 1,000 and 5,000 nodes. With small n's, e.g., 50, it is a little better than the VMC approximation but not as good as the NT one.

The NT and VMC approximations above, which are tailored to M/M/1 queues, could not be applied to general service time distributions as the prediction errors are too large to be useful. Indeed, Fig. 17 shows that while both black-box and white-box ForkMean solutions continue to perform well, with errors within 10 percent, VMC and NT offer extremely poor performance with up to 40 and 50 percent errors for Gamma and Weibull task service time distributions, respectively.

The existing methods for the approximation of the mean response time in the case of M/G/1 Fork-Join models are heuristic-based [37] or hybrid-based [13], [29], i.e., combining simulation and analysis. Moreover, these works mainly focus on light-tailed distributions, e.g., Exponential (Exp), Erlang-2 (E2), and Hyperexponential-2 (H2). In contrast, in addition to these distributions, ForkMean solutions are also validated for a wide range of service time distributions.

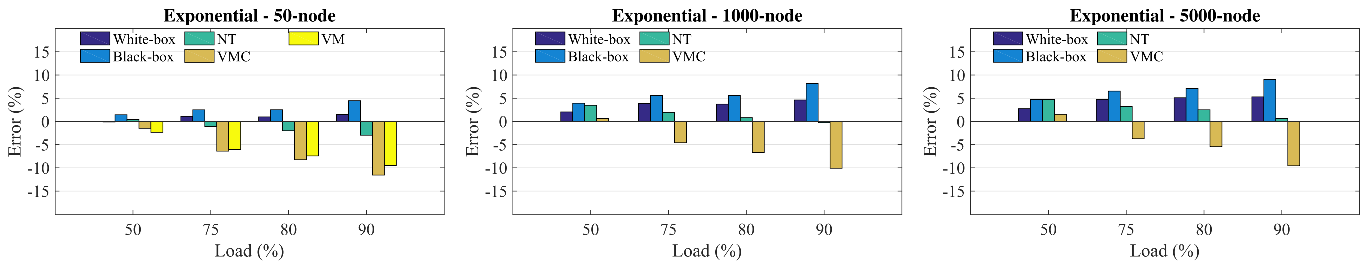


Fig. 16. Comparison of percentage errors in mean latency approximations where each Fork node is modeled as an M/M/1 queuing system.

TABLE 6
Errors for Mean Latency Prediction With M/E2/1 Queues

Load	Method	Number of nodes			
		5	10	15	20
50%	VM	−0.806	−1.486	−1.985	−1.827
	White-box	−7.947	−6.312	−5.483	−4.934
75%	VM	−2.989	−4.587	−5.748	−5.637
	White-box	−9.827	−7.360	−6.316	−5.104
80%	VM	−3.440	−5.336	−6.886	−7.400
	White-box	−10.101	−7.524	−6.666	−5.922
90%	VM	−5.414	−7.885	−9.039	−9.538
	White-box	−11.001	−8.110	−6.398	−5.251

TABLE 7
Errors for Mean Latency Prediction With M/H2/1 Queues

Load	Method	Number of nodes			
		5	10	15	20
50%	VM	−1.007	6.446	13.389	17.937
	White-box	0.869	0.945	1.881	2.118
75%	VM	−1.682	6.556	12.601	16.678
	White-box	−1.255	0.975	2.091	2.574
80%	VM	−0.402	6.361	11.687	14.975
	White-box	−0.106	1.503	2.563	2.753
90%	VM	0.111	4.030	6.366	8.697
	White-box	−0.081	1.183	1.242	1.825

To test the effectiveness of ForkMean, we first compare our white-box solution with the heuristic approximations in [37] for the cases of Erlang-2 (E2) and Hyperexponential-2 (H2) service time distributions with Poisson arrivals, i.e., M/G/1 queues.

Tables 6 and 7 present the comparative results for Erlang-2 and Hyperexponential-2, respectively. Again due to the computational complexity concerning the VM approximation, we perform comparison only for small n's, i.e., up to 20, the maximum problem size studied by the authors of the VM approximation [37], although our solution offers consistent performance at large n's as well. For the Erlang-2 distribution, the VM approach gives better predictions at load level of 50 percent and lower numbers of nodes, i.e., 5 and 10 nodes, while our solution yields comparable or better predictions for the other settings. The accuracy of our approach outperforms that of the VM for the Hyperexponential distribution. Although yielding good prediction performance for systems with small numbers of Fork nodes, the VM approximation faces the issue of numerical instatibility and computational complexity due to big binomial coefficients, resulting in higher prediction errors for higher numbers of nodes, as observed from the reported results. In additon, while the VM approximation can in theory be applied to G/G/1 queues, finding light and heavy traffic limits for an arbitrary service time distribution, e.g., Weibull or truncated Pareto, is nontrivial.

Fig. 18 shows the prediction accuracy of ForkMean for the above heavy-tailed service time distributions. Both white-box and black-box solutions yield quite accurate predictions for less heavy-tailed distributions, i.e., Weibull, for all the cases studied, with errors within 12 percent for all the cases. For heavier tailed distributions, i.e., truncated Pareto and empirical, the solutions give good approximations at high load levels, i.e., 80 percent or higher, a region of interest for resource provisioning. Overall, the black-box solution gives comparably close prediction performance to that of the white-box one. The errors are mostly within 20 and 10 percent at the load levels of 80 and 90 percent, respectively.

The predictions for G/G/1 cases as in Section 3.1 also show similar performance, i.e., within 20 percent errors at the load levels of 80 percent or higher, which are not shown here due to the lack of space.

3.2.2 Scenario 2: Systems With Replicated Servers

We now validate ForkMean for systems with 3-replica Fork nodes. We consider two dispatching policies, i.e., Round-Robin and redundant-task-issue, and heavy-tailed service time distributions as in Section 3.1. The validation is run only for the black-box solution since the exact service time distributions for the Fork nodes are simply unknown for such cases.

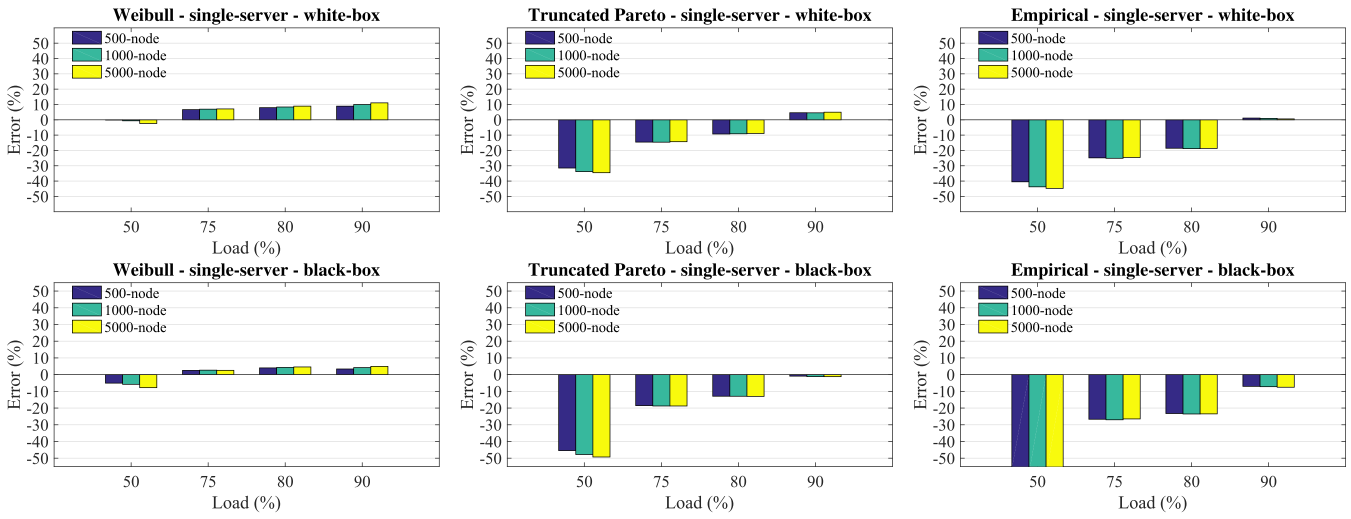


Fig. 18. Errors for mean response time approximations using the white-box (upper row) and black-box (lower row) solutions.

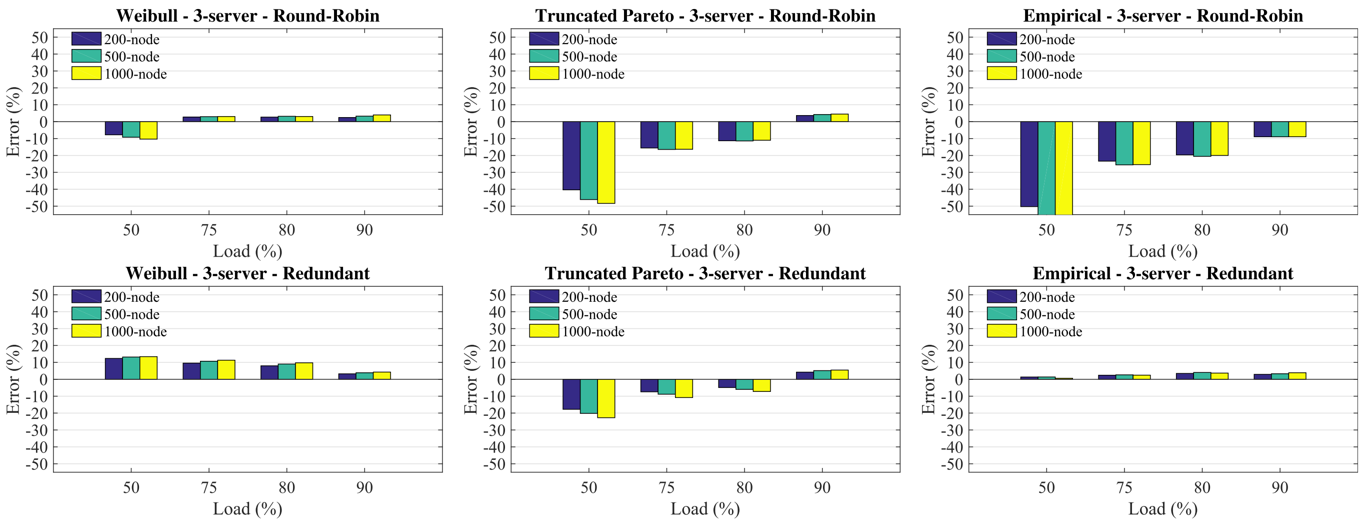


Fig. 19. Errors in mean response time approximation for systems with replicated servers applying Round-Robin (upper row) and redundant-task-issue (lower row) policies.

Fig. 19 presents the results for these cases using the black-box approach, applying the Δ values measured from the respective systems at $n = 100$ to the ones with 200, 500, and 1,000 nodes. One can see that the results for the Round-Robin cases are close to those in the previous scenario. This is due to the fact that the Round-Robin policy mainly performs load balancing between replica and thus the effective service time distributions on the Fork nodes are almost unchanged. In contrast, the model yields good predictions for the redundant-task-issue policy for the entire load range under study. This is largely because this policy curbs the tail effects and makes the effective service time distributions less heavy-tailed. These results agree with those from the previous scenarios for less heavy-tailed distributions, i.e., Gamma and Weibull.

3.2.3 Scenario 3: Systems With Variable Numbers of Tasks

For illustrative purposes, we validate the results on Fork-Join models with homogeneous, single-server Fork nodes with the above service time distributions using the black-box solution, assuming that the tasks for each incoming job is randomly dispatched to 40–60 percent total number of Fork nodes. As a result, the effective load on each Fork node is half of that on the single-server systems in Scenario 3.2.1. Therefore, we double the arrival rate, λ, to keep the same arrival rate on each node as in the previous cases. The results of this scenario are shown in Fig. 20. Similar to the previous scenarios, the black-box solution gives accurate predictions across the entire load range for light-tailed distributions, e.g., Exponential, Gamma (which is not shown here), while yielding good approximations for the heavy-tailed distributions, i.e., truncated Pareto and empirical, at high load regions, e.g., 80 percent or above.

3.2.4 Scenario 4: A Case Study on Amazon EC2

We also evaluate the accuracy of the black-box solution for the case study on AWS EC2 as in Section 3.1.1. To illustrate the effectiveness of the black-box solution for this case study, we compute the gap for the 32-worker cluster and apply it to the approximation of request mean response time for the case of the 64-worker cluster. Table 8 presents the prediction errors for this case study. Again, the black-box method predicts mean response time quite accurately when the system at the effective load of 60 percent or higher, corresponding to arrival rates greater than 3.5 requests/s.

Finally, we note that the tail effect is a recognized issue in datacenter applications and tail-cutting techniques are often exploited in datacenters to reduce the tail effects [1], [15], [16], [38]. As a result, the effective service time distributions tend to be less heavy-tailed. Therefore, ForkTail and Fork-Mean show a great potential to be able to accurately predict the tail and mean latencies in a wide load range in practice, not limited to a high load region.

4 Sensitivity Analysis

From all the experiments above, we can see that the proposed approximations can be applied to a wide range of

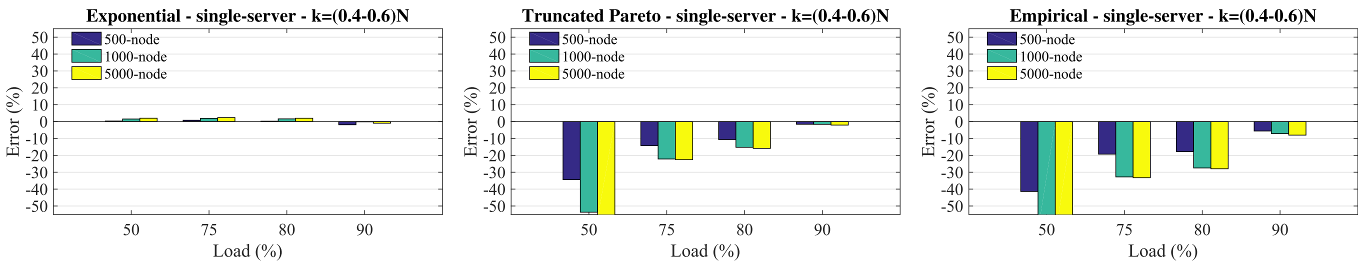


Fig. 20. Errors in mean response time approximation for systems with variable numbers of tasks.

TABLE 8
Errors in Mean Response Time Approximation Using the Black-Box Solution for the Test Case on AWS

	Effective load (Arrival rate (requests/s))					
	50.0%	58.4%	66.7%	75.1%	83.4%	91.7%
#workers	**(3.0)**	**(3.5)**	**(4.0)**	**(4.5)**	**(5.0)**	**(5.5)**
64	31.678	10.489	7.817	8.874	15.274	13.991

systems with reasonable prediction errors for the 99th percentile and mean job latency, consistently within 20 and 15 percent at the loads of 80 and 90 percent, respectively. Now, the question yet to be answered is how much impact these errors will have on the accuracy for resource provisioning at high loads. To this end, we conduct a sensitivity analysis of tail and mean latencies as functions of load.

We perform experiments with different load levels in the high load region, i.e., 78 to 95 percent, for FJQNs with different service time distributions, i.e., exponential, Weibull, truncated Pareto, and empirical ones. Figs. 21 and 22 shows results from both simulation and the proposed approximations for 1,000-node systems. First, we note that the proposed models consistently overestimates the tail and mean latencies for the exponential and Weibull cases, while mostly underestimates them for the truncated Pareto and empirical cases. In other words, the former causes resource overprovisioning, whereas the latter leads to resource underprovisioning. Then the question is how much. Take the exponential case as an example, the predicted mean latency at 90 percent load is roughly equal to the simulated one at 91 percent load. This means that the model may lead to 1 percent resource over provisioning for the exponential cases. Following the same logic, it is easy to find that for both exponential and Weibull cases, the prediction models for both tail and mean latency may result in no more than 1 percent resource overprovisioning in the entire 78–95 percent load range. By the same token, we find that for the truncated Pareto and empirical cases, the models may cause up to 4 and 6 percent resource underprovisioning at 80 percent load and 2 and 1 at 90 percent load for tail and mean latency, respectively. This can be well compensated for by leaving a 6 percent resource margin in practice. This implies that in the worst-case when the actual service time distribution is light-tailed, our approximations may cause up to 7 percent resource overprovisioning at the loads of 80 percent or higher, given that we don't have the knowledge about the tail-heaviness of the workload. With the prediction and the small overprovisioning to compensate the prediction error proposed in this paper, one can expect to run the system at up to 90 percent instead of 50 percent resource utilization with tail and mean latency guarantee.

Our sensitivity analyses for the other Fork-Join structures, which are not shown here, have led to similar conclusions. This demonstrates the effectiveness of our prediction models as a powerful means to facilitate multi-SLO-guaranteed, e.g., tail and mean latency guaranteed job scheduling and resource provisioning for datacenter applications.

5 Applicability Range

In this section, we want to answer the following question: In what parameter range can our models predict the request latency within 20 percent errors at high load? To this end, we note that we need to focus on identifying the applicability range on the heavy tail end, rather than the light tail end for two reasons. First, from the extensive experiments above, we found that our methods give quite accurate approximations for tail and mean latency for a wide range of loads for light-tailed distributions, e.g., Exponential, Gamma, and Erlang-2. Second, in practice, server wokloads in datacenters exhibit heavy-tailed distributions [15], [32]. Also, the heavy-tailed truncated Pareto distribution given in Eq. (13) was found to be a good fit for empirical data from server workloads [31]. Hence, in what follows, we test the applicability range of our approximations based on this distribution.

From extensive experiments with the truncated Pareto distribution, we found that our approximations predict the tail and mean latencies within 20 percent errors at the loads of 80 percent or higher, when the tail index α in Eq. (13) is less than 2, i.e., $0 < \alpha < 2$. This range of α was found to be large enough to cover the server workloads in [31].

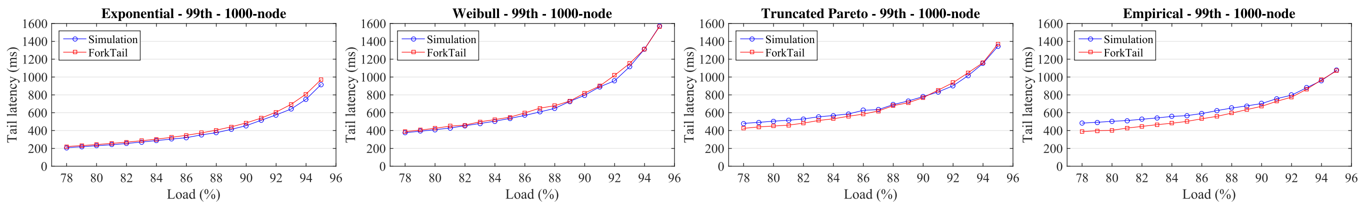


Fig. 21. Differences in the 99th percentile response times from simulation and ForkTail for 1000-node systems with different service time distributions and fixed number of Fork tasks.

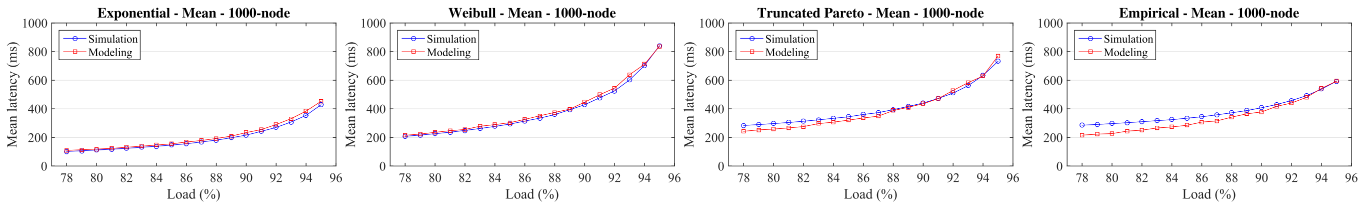


Fig. 22. Differences in mean response times from the simulation and black-box ForkMean for 1000-node systems with different service time distributions and fixed number of Fork tasks.

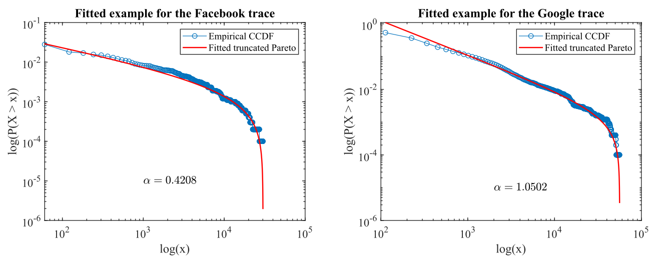


Fig. 23. Examples of fitting the truncated Pareto distribution to sampled data from Facebook and Google traces. The plots show the complementary CDF (CCDF), which is on a log scale, to focus on the tail portion of the distribution.

To further test if today's datacenter workloads indeed fall into the above range, we test the fitting of the truncated Pareto distribution to the workload traces from Facebook and Google provided in [19]. These traces include a mixture of different types of workloads placed on datacenter servers. To simulate the workload on one server, we draw 10,000 random samples from each trace and fit them to the truncated Pareto distribution based on the procedure suggested in [39], which uses the $(r+1)$ largest-order statistics and visual check. We found that the fitted values of α for Google and Facebook samples are mostly within the applicability range of $(0, 2)$. Fig. 23 illustrates two examples of the fitted curves.

The above results strongly suggest that our proposed methods can indeed serve as a useful tool for the approximation of tail and mean latency for datacenter workloads.

6 Facilitating Job Scheduling and Resource Provisioning

We now discuss how our proposed approximations may be used to facilitate both SLO-guaranteed job scheduling and resource provisioning. We present here only the procedures for tail latency approximation, i.e., ForkTail. The procedures for mean latency follow similar steps since the approximation of mean latency is based on ForkTail. The proposed ideas are preliminary and somewhat sketchy, but yet, they do help reveal the promising prospects of our proposed model and point directions for future studies on this topic.

Job Scheduling. We describe the ideas of how a tail-latency-SLO-guaranteed hybrid centralized-and-distributed job scheduler can be developed, based on ForkTail. The main idea is to rely on distributed measurement of the means and variances of the task response times and centralized decision making as to how and whether the request tail-latency SLO can be met, as depicted in Fig. 24. In the master server on the left resides the central job scheduler to which users submit their requests with given tail-latency SLOs. All the servers in the cluster measures the means and variances of task response times for tasks of different sizes or in different bins on a continuous basis. All the servers periodically convey their measurements to the central scheduler. Upon the arrival of a request with a given tail-latency SLO and given k tasks to spawn, based on Eq. (5), the central scheduler will run a Fork-node selection algorithm to determine which k Fork nodes should be used such that the tail-latency SLO can be met. If such k Fork nodes are found, the request will be admitted, otherwise, either the tail-latency SLO will be renegotiated

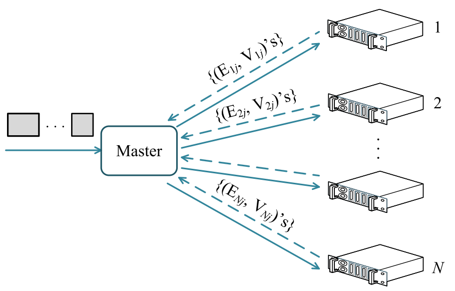


Fig. 24. A hybrid, centralized-and-distributed job scheduler.

or the request will be rejected. At runtime, the central scheduler periodically run the prediction model using the up-to-date means and variances as input to ensure that the tail-latency SLOs for the on-going requests continue to be met.

Resource Provisioning. ForkTail for the homogeneous case (i.e., Eqs. (8) and (9)) naturally enables a resource provisioning solution involving two steps: (a) the evaluation of the task-level performance requirements to achieve a given tail-latency SLO; and (b) the selection of an underlying platform to meet the requirements. Here, step (a) is platform independent and hence is portable to any datacenter platforms.

For example, consider a service deployment scenario with a given tail-latency SLO and a minimum throughput requirement, R. Assuming that N, k_i, and $P(K = k_i)$ for the given service are known, Eq. (9) can be used to first translate the tail-latency SLO into a pair, i.e., the mean and variance of the task response time. This pair then serves as the task performance budgets or the task-level performance requirements, which are platform independent and portable. This completes step (a).

In step (b), a Fork node is set up, e.g., using three virtual machine instances purchased from Amazon EC2 to form a 3-replica Fork node, loaded with a data shard in the memory. Then run tasks at increasing task arrival rate λ until the measured task mean and/or variance are about to exceed the corresponding budget(s). At this arrival rate λ, the tail-latency SLO is met without resource over-provisioning. In other words, the λ value at this point would be the maximum sustainable task throughput, or equivalently, the request throughput, in order to meet the tail-latency SLO. If this throughput is greater than R, the minimum throughput requirement is also met. This means that the resource provisioning is successful and a cluster with $3N$ VM instances can be deployed. Otherwise, repeat step (b) by using a more powerful VM instance or with a re-negotiated tail-latency SLO and/or minimum throughput requirement.

7 Related Work

Fork-Join structures are traditionally modeled by FJQNs, which have been studied extensively in the literature. To date, the exact solution exists for a two-Fork-node FJQN only [10], [40]. Most of the previous works primarily focus on the approximation of mean response time [10], [11], [41] and its bounds [42], [43]. For networks with general service time distribution, several works have introduced hybrid approaches that combine analysis and simulation to derive the empirical approximation for mean response time [10], [13].

Some analytic results are available on redundant task issues [44], [45], [46]. They either address only a single replicated server subsystem with exponential task service time distribution [45] or parallel request load balancing without task spawning [44], [46].

Tail Latency Approximation. In terms of tail-latency related research, several works dealt with the approximation of response time distribution assuming a simple queuing model for each Fork node, e.g., M/M/1 [47] or M/M/k [12]. Computable stochastic bounds on request waiting and response time distributions for some FJQNs are provided in a recent work [48]. The most interesting and relevant work is given in [14]. The authors of this work proposed a method, called EAT, for the approximation of tail latency for homogeneous FJQNs based on the analytical results from single-node and two-node cases. The approximation applies to FJQNs with any service time distribution that can be transformed into a phase-type distribution. Although outperforming our solutions by a few percentage points in terms of tail prediction, its computational complexity renders it infeasible to facilitate online resource provisioning. Moreover, this work can only cover a small fraction of the aforementioned design space and hence, cannot be used to facilitate resource provisioning in practice.

Mean Latency Approximation. Various works have been proposed for the approximation of mean response time of FJQNs using model-based or hybrid approaches. The work in [10] introduces a hybrid approach for the approximation of mean response time, R_n, for a Fork-Join model with n M/M/1 Fork nodes ($2 \leq n \leq 32$) based on the exact solution for the 2-way network [40] and simulation. In [36], the authors proposed an approximation for mean response time based on the optimistic and pessimistic bounds. Another approximation for mean response time of Fork-Join models with general inter-arrival and service time distributions is proposed in [37] based on light traffic interpolation and heavy traffic limit. The light traffic interpolation is computed from the mean response time of the Fork-Join network when there is only a tagged job in the network, which is equivalent to the maximum of task service time random variables. The heavy traffic limit is postulated based on the observation of the relationship between expressions for light and heavy traffic for 1-way and 2-way networks. In [29], the authors proposed a hybrid procedure for the approximation of mean response time for Fork-Join models with M/G/1 queues. Indeed, this work proposed a methodology rather than specific expressions for finding mean response time. In a recent work [49], a simulation study assessed the accuracy of the approximation based on order statistic.

The existing approaches above are white-box solutions targeting at individual Fork-Join models with specific queuing server models. In contrast, in this paper, we propose both white-box and black-box solutions, applicable to Fork-Join networks with arbitrary server models.

SLO-Aware Resource Provisioning. Due to the lack of theoretical underpinning, the existing SLO-aware resource provisioning proposals cannot provide tail and/or mean latencies SLO guarantee by design. Instead, various techniques such as tail-cutting techniques [15], [16], a combination of job priority and rate limiting based on network calculus [50] are employed to indirectly provide high assurance of meeting tail-latency SLOs. As indirect solutions, however, they cannot ensure precise resource allocation to meet tail-latency SLOs, while allowing high resource utilization, and hence may result in resource overprovisioning. Yet, another alternative solution is to track the target tail-latency SLO through online, direct tail-latency measurement and dynamic resource provisioning [51], [52]. This approach, however, may not be effective, especially in enforcing stringent tail latency SLOs. To see why this is true, consider the 99.9th percentile request response time of 200 ms, i.e., probabilistically, only one out of 1,000 requests should experience a response time greater than 200 ms. Assume that the average request arrival rate is 50 per second. To track, through direct tail-latency measurement, whether this tail latency SLO is violated or not with reasonably high confidence, one needs to collect, e.g., 100K samples to see if there are more than 100 requests whose response times exceed 200 ms. This, however, takes about 100K/50 = 2000 seconds or about 33 minutes of measurement time! Given possibly high volatility of datacenter workloads, the tail latency SLO may have been violated multiple times during this measurement period, even though the total number of requests whose response times exceeding 200 ms may be well within 100. In constrast, using our proposed models, with only 20 seconds of measurement time, one can collect $20 \times 50 = 1000$ task samples at individual Fork nodes to allow a reasonably accurate estimation of the means and variances of task response times. With moving average for a given time window, e.g., 20 seconds, these means and variances and hence, the 99.9th percentile, can be updated every tens of milliseconds, making it possible to enable fast online tail-latency-guaranteed job scheduling and resource provisioning.

In summary, a solution that can predict the tail and/or mean latency using a small number of samples collected in a short period of time as input and that applies to a large design space of Fork-Join structures must be sought, the primary motivation of the current work.

8 Conclusion and Future Work

A key challenge in enabling tail-latency and/or mean-latency SLOs for data-intensive services and applications in datacenters is how to predict the latencies for a broad range of Fork-Join structures underlying those services and applications. In this paper, we proposed to study a generic black-box Fork-Join model for the approximations of tail and mean latency that covers most Fork-Join structures of practical interests. On the basis of a central limit theorem for queuing servers under heavy load, we were able to arrive at approximate solutions to this model for both tail and mean latencies, called ForkTail and ForkMean, respectively. These approximations were found to be able to predict the tail and mean latencies for most practical scenarios consistently within 20 percent in a load region of 80 percent or higher, resulting in at most 7 percent resource overprovisioning, making it a powerful tool for resource provisioning at high load. Finally, we discussed some preliminary ideas of how to make use of the proposed prediction model to facilitate tail-latency-SLO-guaranteed job scheduling and resource provisioning.

In our future work, based on ForkTail and ForkMean, we shall develop both job scheduling and online/offline resource provisioning solutions with tail-latency and/or mean-latency SLO guarantee.

ACKNOWLEDGMENTS

This work is supported by the NSF under awards CCF XPS 1629625 and CCF 1704504.

REFERENCES

[1] M. Jeon *et al.*, "Predictive parallelization: Taming tail latencies in web search," in *Proc. 37th Int. ACM SIGIR Conf. Res. Develop. Inf. Retrieval*, 2014, pp. 253–262.

[2] J. Brutlag, "Speed matters for Google web search," 2009. [Online]. Available: https://services.google.com/fh/files/blogs/google_delayexp.pdf

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th Conf. Symp. Operating Syst. Des. Implementation*, 2004, pp. 137–150.

[4] Apache spark, Accessed: Feb. 26, 2020. [Online]. Available: https://spark.apache.org

[5] G. Blake and A. G. Saidi, "Where does the time go? Characterizing tail latency in memcached," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2015, pp. 21–31.

[6] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale," in *Proc. 3rd ACM Symp. Cloud Comput.*, 2012, pp. 7:1–7:13.

[7] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware cluster management," in *Proc. 19th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2014, pp. 127–144.

[8] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi, *Queueing Networks and Markov Chains: Modeling and Performance Evaluation With Computer Science Applications*. Hoboken, NJ, USA: Wiley-Interscience, 2006.

[9] A. Thomasian, "Analysis of fork/join and related queueing systems," *ACM Comput. Surv.*, vol. 47, no. 2, pp. 1–71, 2014.

[10] R. Nelson and A. N. Tantawi, "Approximate analysis of fork/join synchronization in parallel queues," *IEEE Trans. Comput.*, vol. 37, no. 6, pp. 739–743, Jun. 1988.

[11] E. Varki, "Response time analysis of parallel computer and storage systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 11, pp. 1146–1161, Nov. 2001.

[12] S. S. Ko and R. F. Serfozo, "Response times in M/M/s fork-join networks," *Advances Appl. Probability*, vol. 36, no. 3, pp. 854–871, 2004.

[13] R. J. Chen, "A hybrid solution of fork/join synchronization in parallel queues," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 8, pp. 829–845, Aug. 2001.

[14] Z. Qiu, J. F. Pérez, and P. G. Harrison, "Beyond the mean in fork-join queues: Efficient approximation for response-time tails," *Perform. Eval.*, vol. 91, pp. 99–116, 2015.

[15] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[16] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker, "Low latency via redundancy," in *Proc. 9th ACM Conf. Emerg. Netw. Experiments Technol.*, 2013, pp. 283–294.

[17] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz, "Wrangler: Predictable and faster jobs using fewer resources," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 26:1–26:14.

[18] R. Nishtala *et al.*, "Scaling memcache at Facebook," in *Proc. 10th USENIX Conf. Netw. Syst. Des. Implementation*, 2013, pp. 385–398.

[19] P. Delgado, F. Dinu, A. M. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid datacenter scheduling," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2015, pp. 499–510.

[20] J. F. C. Kingman and M. F. Atiyah, "The single server queue in heavy traffic," *Proc. Cambridge Philosophical Soc.*, vol. 57, pp. 902–904, 1961.

[21] J. Köllerström, "Heavy traffic theory for queues with several servers. I," *J. Appl. Probability*, vol. 11, no. 3, pp. 544–552, 1974.

[22] M. Nguyen, Z. Li, F. Duan, H. Che, Y. Lei, and H. Jiang, "The Tail at Scale: How to Predict It?" in *Proc. 8th USENIX Workshop Hot Topics Cloud Comput.*, 2016.

[23] S. Sani and O. A. Daman, "Mathematical modeling in heavy traffic queuing systems," *Amer. J. Operations Res.*, vol. 4, pp. 340–350, 2014.

[24] R. D. Gupta and D. Kundu, "Generalized exponential distributions," *Australian New Zealand J. Statist.*, vol. 41, no. 2, pp. 173–188, 1999.

[25] M. Nguyen, S. Alesawi, N. Li, H. Che, and H. Jiang, "ForkTail: A black-box fork-join tail latency prediction model for user-facing datacenter workloads," in *Proc. 27th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2018, pp. 206–217.

[26] L. Kleinrock, *Queueing Systems*, Vol. 1: Theory. Hoboken, New Jersey, USA: Wiley, 1975.

[27] W. Whitt, "The queueing network analyzer," *The Bell Syst. Tech. J.*, vol. 62, no. 9, pp. 2779–2815, Nov. 1983.

[28] G. Brys, M. Hubert, and A. Struyf, "Robust measures of tail weight," *Comput. Statist. Data Anal.*, vol. 50, no. 3, pp. 733–759, 2006.

[29] A. Thomasian and A. N. Tantawi, "Approximate solutions for M/G/1 fork/join synchronization," in *Proc. 26th Conf. Winter Simul.*, 1994, pp. 361–368.

[30] L. A. Barroso, J. Dean, and U. Hölzle, "Web search for a planet: The Google cluster architecture," *IEEE Micro*, vol. 23, no. 2, pp. 22–28, 2003.

[31] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*, 1st ed. Cambridge, U.K.: Cambridge Univ. Press, 2013.

[32] D. Meisner, W. Junjie, and T. F. Wenisch, "BigHouse: A simulation infrastructure for data center systems," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2012, pp. 35–45.

[33] Apache hadoop, Accessed: Feb. 26, 2020. [Online]. Available: https://hadoop.apache.org

[34] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of MapReduce workloads," *Proc. VLDB Endowment*, vol. 5, no. 12, pp. 1802–1813, Aug. 2012.

[35] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. 5th Eur. Conf. Comput. Syst.*, 2010, pp. 265–278.

[36] E. Varki, A. Merchant, and H. Chen, "The M/M/1 fork-join queue with variable sub-tasks," 2002. [Online]. Available: http://www.cs.unh.edu/ varki/publication/2002-nov-open.pdf

[37] S. Varma and A. M. Makowski, "Interpolation approximations for symmetric fork-join queues," *Perform. Eval.*, vol. 20, no. 1/3, pp. 245–265, 1994.

[38] L. Suresh, M. Canini, S. Schmid, and A. Feldmann, "C3: Cutting tail latency in cloud data stores via adaptive replica selection," in *Proc. 12th USENIX Conf. Netw. Syst. Des. Implementation*, 2015, pp. 513–527.

[39] I. Aban, M. Meerschaert, and A. Panorska, "Parameter estimation for the truncated pareto distribution," *J. Amer. Statist. Assoc.*, vol. 101, no. 473, pp. 270–277, 2006.

[40] L. Flatto and S. Hahn, "Two parallel queues created by arrivals with two demands I," *SIAM J. Appl. Math.*, vol. 44, no. 5, pp. 1041–1053, 1984.

[41] F. Alomari and D. A. Menasce, "Efficient response time approximations for multiclass fork and join queues in open and closed queuing networks," *IEEE Trans. Parallel Distributed Syst.*, vol. 25, no. 6, pp. 1437–1446, Jun. 2014.

[42] S. Balsamo, L. Donatiello, and N. M. Van Dijk, "Bound performance models of heterogeneous parallel processing systems," *IEEE Trans. Parallel Distributed Syst.*, vol. 9, no. 10, pp. 1041–1056, Oct. 1998.

[43] R. J. Chen, "An upper bound solution for homogeneous fork/join queuing systems," *IEEE Trans. Parallel Distributed Syst.*, vol. 22, no. 5, pp. 874–878, May 2011.

[44] D. Wang, G. Joshi, and G. Wornell, "Efficient task replication for fast response times in parallel computation," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 42, no. 1, pp. 599–600, Jun. 2014.

[45] K. Gardner, S. Zbarsky, S. Doroudi, M. Harchol-Balter, E. Hyytiä, and A. Scheller-Wolf, "Reducing latency via redundant requests: Exact analysis," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst.*, 2015, pp. 347–360.

[46] Z. Qiu and J. F. Perez, "Evaluating the effectiveness of replication for tail-tolerance," in *Proc. 15th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2015, pp. 443–452.

[47] S. Balsamo and I. Mura, "Approximate response time distribution in Fork and Join systems," in *Proc. ACM SIGMETRICS Joint Int. Conf. Meas. Model. Comput. Syst.*, 1995, pp. 305–306.

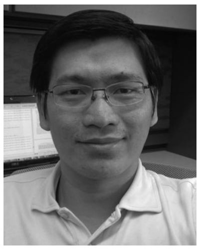
[48] A. Rizk, F. Poloczek, and F. Ciucu, "Computable bounds in fork-join queueing systems," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst.*, 2015, pp. 335–346.

[49] A. Lebrecht and W. J. Knottenbelt, "Response time approximations in fork-join queues," in *Proc. 23rd Annu. UK Perform. Eng. Workshop*, 2007.

[50] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "PriorityMeister: Tail latency QoS for shared networked storage," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 29:1–29:14.

[51] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica, "Cake: Enabling high-level SLOs on shared storage systems," in *Proc. 3rd ACM Symp. Cloud Comput.*, 2012, pp. 14:1–14:14.

[52] A. D. Ferguson, P. Bodik, E. Boutin, and R. Fonseca, "Jockey: Guaranteed job latency in data parallel clusters," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 99–112.



Minh Nguyen received the BS and MS degrees in electrical engineering from the Ho Chi Minh City University of Technology, Vietnam; and the PhD degree in computer engineering from the University of Texas at Arlington, Arlington, Texas. He is currently a lead hardware integration engineer at Ikon Technologies. His current research interests include datacenter resource management and job scheduling, edge computing, IoT, and smart cities.

Sami Alesawi received the BS degree in computer engineering and the MS degree in computer science from King Abdulaziz University, Jeddah, Saudi Arabia, and the PhD degree from The University of Texas at Arlington, Arlington, Texas. He is currently working as an assistant professor at the Faculty of Computing and Information Technology in Rabigh, King Abdulaziz University, Saudi Arabia. His current research interests include datacenter resource management and job scheduling.

Ning Li received the BSc degree in computer science from Jiangsu University, China; the MSc degree in computer engineering from the Nanjing University of Science and Technology, China; and the PhD degree in computer system architecture from the Huazhong University of Science and Technology, China. He is currently working as a post-doc research associate with the University of Texas at Arlington, Arlington, Texas. His research interests include virtualization, quality of service, cloud computing and storage systems.

Hao Che (Senior Member, IEEE) received the BS degree from Nanjing University, Nanjing, China; the MS degree in physics from the University of Texas at Arlington, Arlington, Texas; and the PhD degree in electrical engineering from the University of Texas at Austin, Austin, Texas. He is currently a full professor in the Department of Computer Science and Engineering, University of Texas at Arlington, Texas. Prior to joining UTA, he was a system architect with Santera Systems, Inc. in Plano (2000–2002) and an assistant professor of electrical engineering at the Pennsylvania State University (1998 to 2000). His current research interests include network architecture and Internet traffic control, datacenter resource management and job scheduling, edge computing and IoT.

Hong Jiang (Fellow, IEEE) received the BSc degree in computer engineering from the Huazhong University of Science and Technology, Wuhan, China; the MASc degree in computer engineering from the University of Toronto, Toronto, Canada; and the PhD degree in computer science from the Texas A&M University, College Station, Texas. He is currently chair and Wendell H. Nedderman Endowed professor of Computer Science and Engineering Department, University of Texas at Arlington, Arlington, Texas. Prior to joining UTA, he served as a program director at National Science Foundation (2013–2015) and he was at University of Nebraska-Lincoln since 1991, where he was Willa Cather professor of Computer Science and Engineering. He has graduated 17 PhD students and supervised 20 post-doctoral fellows and visiting scholars. He is currently supervising/co-supervising more than 10 PhD students and post-doc fellows. His present research interests include computer architecture, computer storage systems and parallel I/O, high-performance computing, big data computing, and cloud and edge computing. He is an associate editor of the *IEEE Transactions on Computers* and recently served as an associate editor of the *IEEE Transactions on Parallel and Distributed Systems*. He has more than 300 publications in major journals and international Conferences in these areas, including the *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *Proceedings of IEEE*, *ACM Transactions on Architecture and Code Optimization*, the *ACM Transactions on Storage*, USENIX ATC, FAST, EUROSYS, ISCA, MICRO, SOCC, LISA, SIGMETRICS, ICDE, DATE, ICDCS, IPDPS, MIDDLEWARE, OOPLAS, ECOOP, SC, ICS, HPDC, INFOCOM, ICPP, etc., and his research has been supported by NSF and industry. He is a member of the ACM.

ESetStore: An Erasure-Coded Storage System With Fast Data Recovery

Chengjian Liu, Qiang Wang, Xiaowen Chu, *Senior Member, IEEE*,
Yiu-Wing Leung, and Hai Liu, *Member, IEEE*

Abstract—Erasure codes have been used extensively in large-scale storage systems to reduce the storage overhead of triplication-based storage systems. One key performance issue introduced by erasure codes is the long time needed to recover from a single failure, which occurs constantly in large-scale storage systems. We present ESetStore, a prototype erasure-coded storage system that aims to achieve fast recovery from failures. ESetStore is novel in the following aspects. We proposed a data placement algorithm named ESet for our ESetStore that can aggregate adequate I/O resources from available storage servers to recover from each single failure. We designed and implemented efficient read and write operations on our erasure-coded storage system via effective use of available I/O and computation resources. We evaluated the performance of ESetStore with extensive experiments on a cluster with 50 storage servers. The evaluation results demonstrate that our recovery performance can obtain linear performance growth by harvesting available I/O resources. With our defined parameter recovery I/O parallelism under some mild conditions, we can achieve optimal recovery performance, in which ESet enables minimal recovery time. Rather than being an alternative to improve recovery performance, our work can be an enhancement for existing solutions, such as Partial-parallel-repair (PPR), to further improve recovery performance.

Index Terms—ESetStore, ESet, Erasure coded storage systems, Fast data recovery

1 Introduction

Recent years have witnessed rapid growth in the amount of data in large-scale distributed storage systems. In 2015, the European Centre for Medium-Range Weather Forecasts revealed that its data had reached 100 PB and had an annual growth rate of 45 percent [1]. A recent study illustrated that genomic big data have reached full storage of a data center with a 100-PB storage capacity [2]. Triplication [3], which is a reliability mechanism used in traditional storage systems, introduces unaffordable storage costs with 3x storage overhead. This makes the reduction of storage overhead an unavoidable task in large-scale storage systems. Many storage systems have begun to use erasure codes as their reliability mechanism [4]. Microsoft's cloud service Azure [5], Facebooks warehouse [6], and Web service storage system f4 [7] have already adopted erasure codes to reduce their storage costs. Famous distributed file systems such as HDFS [8] and Ceph [9] also support erasure coding, yielding greater reliability and lower storage overhead.

An erasure-coded storage system is defined by two integer parameters, n and k. A file stored in the system is divided into k equally sized data blocks. These data blocks generate $n-k$ equally sized parity blocks. The blocks are stored in n storage components to protect data against up to $n-k$ failures. The storage overhead is n/k. For example, Facebook f4 [7] sets k as 10 and n as 14, where its storage overhead is 1.4x. The QFS sets k as 6 and n as 9 with 1.5x storage overhead [10]. However, an erasure-coded storage system may suffer from many performance penalties, one of which is the long time needed to recover a failed storage component, which can be a disk device or a storage server. To recover a missing block, k blocks are retrieved from k storage components. Because recovering a failed component is a constantly performed task [6], the long recovery time may introduce degraded service quality to the whole system.

Studies have shown that single failures accounted for more than 99 percent of recoveries [5], [11], [12]. *Recovery from a single failure* is a performance critical operation in erasure-coded storage systems [11], [13]. This motivated us to focus on recovery from a single failure to evaluate recovery performance of erasure-coded storage systems. A single failure can refer to the failure of a single disk and to the failure of a single storage server. Here we regard a single failure as the failure of a single storage server. The recovery of both kinds of failures requires disk I/O and network bandwidth, both of which can be regarded as I/O resources. We aim to gain adequate I/O resources for each recovery.

The key reason behind the long time needed to recover from a single failure is the heavy I/O operations, which is k times the replication-based storage. To improve recovery performance, researchers have proposed solutions from the following aspects. Some work reduces the I/O operations required to recover a failed component. The Microsoft cloud

- *C. Liu is with the College of Big Data and Internet, Shenzhen Technology University, Shenzhen, Guangdong 518055, China. E-mail: liuchengjian@sztu.edu.cn.*
- *Q. Wang, X. Chu, and Y.-W. Leung are with the Department of Computer Science, Hong Kong Baptist University, Kowloon Tong, Hong Kong. E-mail: {qiangwang, chxw, ywleung}@comp.hkbu.edu.hk.*
- *H. Liu is with the Department of Computing, Hang Seng University of Hong Kong, Siu Lek Yuen, Hong Kong. E-mail: hliu@hsu.edu.hk.*

*Manuscript received 12 Mar. 2019; revised 14 Feb. 2020; accepted 21 Mar. 2020. Date of publication 31 Mar. 2020; date of current version 17 Apr. 2020. (Corresponding author: Xiaowen Chu.)
Recommended for acceptance by K. Mohror.
Digital Object Identifier no. 10.1109/TPDS.2020.2983411*

service Azure adopted local repair codes (LRC) that require around $k/2$ blocks to recover a missing block, but at the cost of increasing its storage overhead from n/k to $(n+2)/k$ [5]. The *Hitchhiker* reduces I/O operations by 25 to 45 percent for recovery of single failures [14]. Some other works improve I/O utilization of the recovery. Partial-parallel-repair pipelined I/O operations to introduce k times of improvement of recovery performance [15]. A recent work studied the pipelined recovery for heterogeneous environments [16].

We can conclude that the long recovery time is caused mainly by limited I/O resources for the recovery of a single failure. Data placement algorithms, a promising means to aggregate the desired I/O resources to recover from a single failure, are ignored by studies of erasure-coded storage systems. This motivated us to propose ESetStore, an erasure-coded storage system with fast data recovery via a novel data placement algorithm ESet. To this end, we designed a data placement algorithm named ESet to harvest adequate I/O resources for recovery of a single failure [17]. Our solution can be an enhancement to existing solutions to improve the recovery performance of erasure-coded storage systems. Our major contributions are summarized as follows:

1) We present a placement algorithm named ESet to improve recovery from failures.
2) We conduct a theoretical analysis to illustrate how ESet can achieve desired I/O aggregation for recovery of a single failure with proper configuration.
3) Rather than being an alternative to existing solutions for improving recovery performance, the placement algorithm ESet can be an enhancement to existing solutions to bring better recovery performance.
4) We achieve efficient I/O utilization for the read and write operations in ESetStore by making an efficient utilization of both I/O and computation resources.
5) We conduct extensive experiments to validate the effectiveness of our proposed ESetStore, and compare it with two storage systems: HDFS [8] and Ceph [9].
6) ESetStore is open-source and available to the public.[1]

The remainder of this paper is organized as follows. Section 2 introduces the background and some related work. Section 3 formulates the problem of long recovery time caused by improper data placement. We present the design of ESet in Section 4. The analysis of ESet is presented in Section 5. The design and implementation of ESetStore are presented in Section 6. Section 7 presents the experimental results of ESetStore. We conclude the paper in Section 8.

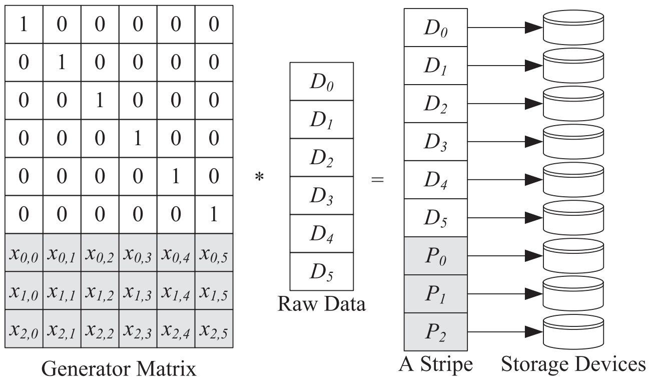
2 Background and Related Work

In this section, we first briefly introduce the background of erasure-coded storage systems. Then, we present some related work about the optimization of the recovery.

2.1 Erasure-Coded Storage Systems

Fig. 1 presents a concrete example with $n = 9$ and $k = 6$. In this example, we assume that the system uses the Reed-Solomon code [18] as the erasure code, which satisfies the *Maximum Distance Separable (MDS)* property [19]. When writing data to the storage system, the raw data are divided

1. Available at https://github.com/stevenlcj/ESetStore



1 0 0 0 0 0
0 1 0 0 0 0
0 0 1 0 0 0
0 0 0 1 0 0
0 0 0 0 1 0
0 0 0 0 0 1
$x_{0,0}$ $x_{0,1}$ $x_{0,2}$ $x_{0,3}$ $x_{0,4}$ $x_{0,5}$
$x_{1,0}$ $x_{1,1}$ $x_{1,2}$ $x_{1,3}$ $x_{1,4}$ $x_{1,5}$
$x_{2,0}$ $x_{2,1}$ $x_{2,2}$ $x_{2,3}$ $x_{2,4}$ $x_{2,5}$
Generator Matrix
*
D_0 D_1 D_2 D_3 D_4 D_5
Raw Data
=
D_0 D_1 D_2 D_3 D_4 D_5 P_0 P_1 P_2
A Stripe
Storage Devices

Fig. 1. Erasure-Coded storage with $n = 9$ and $k = 6$ using RS code.

into k equally sized data blocks. A generator matrix serves as another input. The matrix consists of a $k \times k$ identity matrix and a $(n-k) \times k$ matrix, where each element in the $(n-k) \times k$ matrix is an integer. The matrix multiplies with the set of data blocks to generate n blocks. The set of n blocks with k data blocks and $n-k$ parity blocks together is called a *stripe*. The set of blocks is then distributed into n storage components to tolerate up to $n-k$ failures.

The selected n devices for storing a stripe are typically from n distinct *failure domains* to minimize the probability of data loss. A failure domain is a physical container that includes a set of storage servers. When the container encounters a failure, all servers in the container become unavailable. The failure domain is a frequently used term in discussions of system failures [7], [20], [21], [22]. In this paper, we consider each rack as a failure domain. Stripes are distributed across failure domains to protect data loss against failures from the same failure domain.

When any storage device becomes unavailable, the storage system must recover the data within it. For each missing block in a stripe, k blocks are gathered together to reproduce it. The process of recovering a missing block is performed repeatedly until all blocks in the failed storage device become available. This recovery process has become a routine job in large-scale storage systems. For example, at Facebook, which has a production cluster of 3,000 storage servers, around 20 storage servers will encounter failures and require recovery each day [23]. It was also revealed that the recovery of a failed server storing about 150 blocks (each block size is 64 MB and k = 10) could take more than 50 minutes, which indicates that recovery performance plays a critical role in storage systems.

Recovery occupies heavy I/O resources in erasure-coded storage systems. Thus, recovering a single failure takes a long time if I/O resources are limited. Reducing the required I/O operations is a major approach to improving the recovery performance of erasure-coded storage systems [5], [14]. Some studies have attempted to reduce recovery time by making better use of available I/O resources [15], [16]. However, the performance is still limited by available I/O resources. This motivated us to design and implement a data placement algorithm to improve the available I/O resources for recovery of single failures. Our work can be used with existing solutions such as Partial-parallel-repair (PPR) [15] to further improve the recovery performance of erasure-coded storage systems by better I/O utilization.

TABLE 1
Properties of Some State-of-the-Art Data Placement Algorithms

Algorithm Name	Examples	Reliability Mechanism	Scenario	Scalability	Reliability	Efficient Reconstruction
Random Placement [21]	HDFS	Replication & Erasure Coding	Centralized	Yes	No	Replication
Copyset [21]	HDFS	Replication	Centralized	Yes	Yes	Replication
CRUSH [22]	Ceph	Replication & Erasure Coding	Decentralized	Yes	Yes	N/A
ESet	ESetStore	Erasure Coding	Centralized	Yes	Yes	Erasure Coding

2.2 Related Work

Recovery of missing data comprises two operations: data gathering and decoding. The performance of decoding was improved by some recent studies like those by Gibraltar [24], PErasure [25] and G-CRS [26]. This leaves I/O operations as the major bottleneck of recovery. We classify the existing studies into two categories: one attempts to reduce I/O operations, and the other attempts to improve I/O utilization. We also investigate why some state-of-the-art data placement algorithms are not suited for fast data recovery in erasure-coded storage systems.

Reducing I/O Operations. XOR-based erasure codes access less data when recovering a single failure. Some researchers proposed optimization techniques for specific codes like RDP Code [27] and X-Code [28]. The *rotated Reed-Solomon codes* hold the reliability and performance properties of standard Reed-Solomon codes and can reduce the data required for recovery by up to 30 percent [11]. Algorithms have also been proposed to find solutions for reading less data for XOR-based erasure codes [13], [29].

Some studies sought solutions to reduce the required I/O operations by half but at the cost of increased storage overhead. The family of LRCs [30] is representative: it encodes each $k/2$ of data out of a stripe and stores the parity in a new storage device. When performing recovery, only $k/2$ of data blocks are required to recover a missing block. The solution is used in Microsoft Azure [5]. The XORing Elephants studied the LRCs and noted that the cost of faster recovery is a 14 percent increase in storage overhead [23].

The family of regenerating codes [31], which hold the properties of *MDS* codes [19], reduced the amount of data needed to recover a failed storage component with special constraints on k and n-k. Recent studies proposed some solutions without special constraints. A new storage code developed from the *Piggybacking* framework [32] can reduce network and disk use during recovery by 30 percent and is used in Facebook's warehouse cluster [6]. The *Hitchhiker* can reduce I/O operations by around 25 to 45 percent for recovery of a single failure [14], where it also holds the property of *MDS* codes and supports arbitrary n and k.

Improving I/O Utilization. When recovering a block from a stripe, k blocks from the same stripe are retrieved from k storage servers concurrently. Because each server has the same network throughput, the network bandwidth is underutilized in the recovery. The Partial-parallel-repair (PPR) makes a thorough investigation of this issue and proposed a pipelined mechanism to make full use of the available bandwidth when performing recovery [15].

The repair pipelining technique was proposed in [16] to make better use of bandwidth for small-size units. An ECPipe was developed to allow the pipelined mechanism to work in heterogeneous environments [16].

Data Placement Algorithm. Data placement algorithms play a key role in large-scale storage systems. Using a data placement algorithm to achieve good recovery performance has been proposed in [33], [34] to help each recovery gain more disk I/O resources. However, these algorithms are designed for scenarios in which disks are in a single server and are not proper for recovery of storage servers in large-scale distributed storage systems. Nowadays, many solutions compromise the property of efficient recovery for other important properties in large-scale storage systems such as balanced storage, reliability, and scalability.

Table 1 lists the main properties of some state-of-the-art placement algorithms. A random placement algorithm can work with both replication and erasure-coding based storage systems [35]. It can select an arbitrary number of storage servers to store data. However, this algorithm is not reliable in large-scale storage systems, as revealed in [21].

Copysets addresses the issue of random placement for replication-based storage systems [21]. The probability of data loss is reduced greatly when random placement is replaced with with Copysets. The parameter "scatter width" can help each storage server in a replication-based storage system achieve efficient data reconstruction. However, this algorithm was designed for replication-based storage systems, and may not work for erasure-coded storage systems to improve recovery performance when the number of storage servers in each copyset is increased.

CRUSH [22], from the family of RUSH algorithms [36], [37], is a well-known algorithm adopted in the storage system Ceph [9]. It uses a pseudo-random algorithm to select a set of storage servers when storing data. However, it is not designed to satisfy the property of efficient recovery for erasure-coded storage systems.

For erasure-coded storage systems, researchers have considered placement algorithms to reduce cross-rack traffic when recovering a single failure [38]. The study case is the situation in which a rack contains more than one block of a given stripe. However, when we need to simultaneously tolerate disk-level, host-level, and rack-level failures [7], cross-rack traffic may not be reduced by them.

In summary, the state-of-the-art data placement algorithms do not allow efficient reconstruction in erasure-coded storage systems. This motivated us to design and implement the placement algorithm ESet, which allows efficient recovery in erasure-coded storage systems.

3 Problem Formulation

In this section, we formulate the problem of efficient reconstruction for erasure-coded storage systems. Our study focuses on the recovery of the single failure. We first explain some terms, and then illustrate our motivation by presenting a simple example. We finish this section by making a formal formulation of the problem.

3.1 Terminologies

Block. A block is a sequence of bytes with a fixed-length. Our storage system includes two kinds of blocks: data blocks and parity blocks. Each data block contains raw data stored by users. The parity blocks are generated from data blocks to protect raw data. The erasure-coded storage system record the location of each file by memorizing the location of its blocks in a metadata server.

Erasure Code. We use Reed-Solomon (RS) code as an example of erasure code in this paper. In an (n, k) RS code, where n is greater than k and k is greater than 1, k data blocks are put together to generate $n-k$ parity blocks (or code blocks). When no more than $n-k$ blocks fail in these blocks, any k remaining blocks can be used to restore the missing blocks. Although the encoding and decoding operations involve huge computation, many existing studies have resolved this challenge with multicore CPU and many-core GPU accelerations [24]. Disk and network I/O overheads are currently the major challenge for such storage systems.

Stripe. A stripe contains n blocks in our system, where k blocks are data blocks and the other $n-k$ blocks are parity blocks. Any k blocks in the same stripe can be gathered together to reproduce other $n-k$ blocks in the system.

Erasure-Coded Set. Each stripe can be distributed in n disks from n distinct storage servers in the storage system. We define an *erasure-coded set* as a set of n storage servers to carry a complete stripe. It can serve as a unit of failure in erasure-coded storage systems. When a storage server fails, all *erasure-coded set*s that contain the failed storage server will begin to recover the missing blocks in the server.

Recovery of Single Failures. A single failure can be a disk failure or a storage server failure. The recovery of a disk or a storage server consumes both disk I/O and network bandwidth from k storage servers, which can both be regarded as I/O resources. We aim to gain adequate I/O resources for the single failure recovery. We regard the recovery of a single failure as the recovery of a storage server.

Recovery I/O Parallelism. Here we define a concept of *recovery I/O parallelism*. We use the symbol I to denote the degree of *recovery I/O parallelism* in the system, which indicates the number of *erasure-coded set*s to which each storage server belongs. A larger value for I indicates that more erasure-coded sets will be available to recover a failed storage server. This may leave more I/O resources for recovery of a failed server. Thus, we can achieve efficient data reconstruction from a single failure by setting a proper value of I.

3.2 Motivation Illustration by an Example

A placement algorithm can decide the number of erasure-coded sets for each storage server. The number of erasure-coded sets for a server indicates the amount of I/O resources that can be involved for recovery from a single failure. More I/O resources may result in better recovery performance.

We begin the demonstration of our motivation with an example in Fig. 2. Here we set n as 3 and k as 2. Six storage servers are indexed from 1 to 6 and organized in three columns. We assume that each column belongs to one rack. Blocks in the same stripe are distributed across racks to tolerate disk-level, host-level, and rack-level failures. We have four stripes indexed from 1 to 4. We describe two data placement algorithms to distribute the stripes. Suppose the network bandwidth is ϕ and the block size is ω. The time needed to recover a block can be calculated as $(k \times \omega)/\phi$, which is the time needed to gather k blocks into a single server by reading blocks one by one.

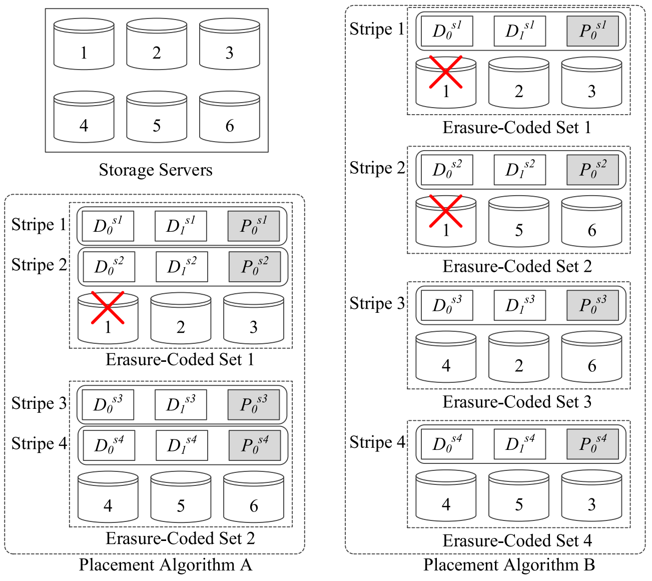


Fig. 2. A naive example of data distribution.

Placement algorithm A, which is shown at the bottom-left of Fig. 2, divides storage servers into two erasure-coded sets. Each erasure-coded set carries two stripes. Placement algorithm B (on the right side of Fig. 2) partitions the six storage servers into four erasure-coded sets. In this way, each erasure-coded set carries one stripe.

Both placement algorithms meet the requirement to distribute blocks across racks to tolerate rack-level failures. The data are evenly distributed among six storage servers, which means that each server stores the same amount of data. However, the recovery performance will differ for placement algorithms A and B when any failure occurs. Note that the placement algorithm A and B differ from standard parity declustering [33]. The standard parity declustering will distribute parity across all disks. As a result, it is not able to tolerate rack-level failures. We will clarify the difference between ESet and parity decluster in Section 5.

For algorithm A, each storage server belongs to one erasure-coded set, so the value of I is 1. For algorithm B, each storage server belongs to two erasure-coded sets, so the value of I is 2. If storage server 1 fails as illustrated in the figure, the time for recovery is $(2 \times k \times \omega)/\phi$ with placement algorithm A, and the time for recovery is $(k \times \omega)/\phi$ with placement algorithm B. This is because in algorithm B, two erasure-coded sets can proceed in parallel to recover missing blocks.

The recovery performance of placement algorithm B is twice that of placement algorithm A, which means that a higher value for I can allow better recovery performance. For fast recovery of these systems, a placement algorithm must be able to set a proper I for each storage server.

Given the above description, we formulate the problem as follows. Given a data center with α racks and β storage servers, we build an erasure-coded storage system to store *s* stripes. The problem is how to design a data placement algorithm that makes each storage server stay in *I* erasure-coded sets, where *I* is a parameter configured by the system administrator when initializing a storage system to indicate the desired recovery performance; meanwhile, the algorithm can evenly distribute data in each storage server, i.e., each storage server stores ns/β blocks, and the algorithm must also distribute each stripe across racks to tolerate rack-level failure. In this way, the recovery time of each storage server can approximate $(ns/\beta) \times ((k \times \omega)/\phi)/I$, where ns/β is the number of blocks in each storage server and $(k \times \omega)/\phi$ is the time needed to recover a single block. In a word, the solution to the problem should have the recovery time for each storage server approximate $(ns/\beta) \times ((k \times \omega)/\phi)/I$.

4 Design of ESet

In this section, we present the design of the data placement algorithm ESet. To minimize the probability of data loss, we partition the storage servers into many *ESetGroup*s, where each *ESetGroup* contains $n \times I$ storage servers organized into *n* rows and *I* columns. We form I^2 erasure-coded sets in each *ESetGroup*. Each storage server in an *ESetGroup* is contained by *I* erasure-coded sets.

4.1 Construction of ESetGroups

To minimize the probability of data loss, we must distribute *n* blocks from each stripe across failure domains. We first transform the whole data center into a set of *ESetGroup*s. Each *ESetGroup* is a set with $n \times I$ storage servers organized into *n* rows and *I* columns. The storage servers in each *ESetGroup* are indexed from $N_{0,0}$ to $N_{n-1,I-1}$. The first subscript represents storage server's row, and the second subscript represents its column within its *ESetGroup*. In each *ESetGroup*, the storage servers from any two different rows share no common failure domain. We can construct each erasure-coded set by selecting *n* storage servers from *n* rows of an *ESetGroup*. In this fashion, we can achieve the minimum data loss probability when storing stripes that distribute blocks across failure domains.

Algorithm 1. Construction of *ESetGroup*s

```
1:  Input: Data Center with $\beta$ storage servers , n, I
2:  Output: g ESetGroups indexed from $G_0$ to $G_{g-1}$
3:  Set the value of g as $\beta/(n \times I)$
4:  Initialize g ESetGroups indexed from $G_0$ to $G_{g-1}$
5:  for each $i \in [0, g-1]$ do
6:     for each $j \in [0, n-1]$ do
7:        Find and select I servers from the Data Center
          that share no common failure domain with
          the storage servers in $G_i$
8:        Add selected servers to the jth row of $G_i$
9:        Remove selected servers from the Data Center
10:    end for
11: end for
12: Return g ESetGroups
```

Here, we present Algorithm 1 to divide β storage servers in the data center into *g* *ESetGroup*s, where *g* is equal to $\beta/(n \times I)$. The *ESetGroup*s are indexed from G_0 to G_{g-1}. Each storage server can only remain in one *ESetGroup*. The for-loop (from line 6 to line 10 in the Algorithm 1) iterates *n* times to generate an *ESetGroup*. When selecting *I* storage servers as one row of an *ESetGroup*, the selected storage servers and other storage servers in the *ESetGroup* must share no common failure domain (at line 7).

4.2 Construction of Each Erasure-Coded Set

We generate an erasure-coded set by selecting *n* storage servers from *n* rows of an *ESetGroup*. Because each storage server in an *ESetGroup* remains in *I* erasure-coded sets, we must generate I^2 erasure-coded sets for each *ESetGroup*.

Here we use the symbol *V* to denote a storage server in an erasure-coded set. The storage servers in an erasure-coded set are indexed from V_0 to V_{n-1}. The subscript refers to the index of the storage server in the erasure-coded set.

The process of generating each erasure-coded set is shown in Algorithm 2 with a given *ESetGroup* G_i. The first for-loop (from line 4 to line 12) iterates I^2 to generate I^2 *erasure-coded set*s. Each erasure-coded set selects *n* storage servers from *n* rows of the G_i. The first storage server V_0 in an erasure-coded set comes from the storage server in the first row of G_i. The *r*th storage server V_r in an erasure-coded set is selected from the *r*th row of an *ESetGroup*.

Algorithm 2. Generate Erasure-Coded Sets

```
1:  Input: $G_i$, n, I
2:  Output: Erasure-coded sets indexed from $E_{iI^2}$ to $E_{(i+1)I^2-1}$
3:  set startIdx as $iI^2$
4:  for each $j \in [0, I^2-1]$ do
5:     Initialize $E_{startIdx+j}$
6:     Set columnIdx as the value of j/I
7:     Select the storage server $N_{0,columnIdx}$ in $G_i$ as the $V_0$ in
       $E_{startIdx+j}$
8:     for each $r \in [1, n-1]$ do
9:        Set columnIdx as the value of ((j/I)(r-1)+j) mod I
10:       Select the storage server $N_{r,columnIdx}$ in $G_i$ as the $V_r$ in
          $E_{startIdx+j}$
11:    end for
12: end for
13: Return $I^2$ Erasure-coded sets
```

The procedure of translating an entire data center with α racks and β hosts into many erasure-coded sets are presented in Fig. 3. We partition the data center into *gESetGroups* with Algorithm 1. Each *ESetGroup* contains $n \times I$ storage servers organized in *n* rows and *I* columns. The Algorithm 2 then iterates *g* times to generate gI^2 erasure-coded sets. We set ε as the value of gI^2. We have ε erasure-coded sets to carry stripes, where each stripe contains *n* blocks. So far, we have illustrated how to generate all erasure-coded sets. Now we must assure each storage server appears in *I* erasure-coded sets to obtain the required *recovery I/O parallelism*. In the next section, we analyze the property of ESet algorithm.

5 Analysis of ESet

To ensure that each storage server has the same *recovery I/O parallelism*, each storage server is expected to remain in *I* erasure-coded sets. In this section, we first validate this property,

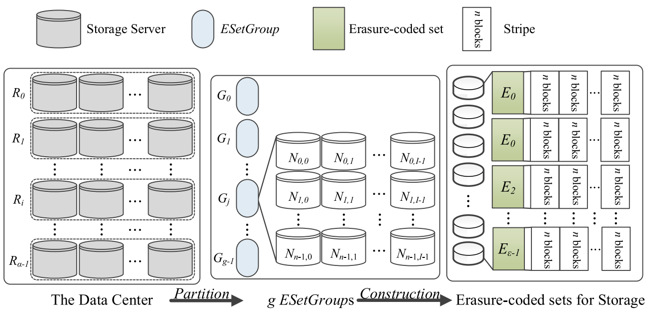


Fig. 3. Partition a data center with α racks and β hosts into ε erasure-coded sets to store stripes.

and then analyze the condition in which each storage server has the optimal *recovery I/O parallelism*.

5.1 Answer When $N_{a,b} \in E_j$

We first prove that our proposed ESet algorithm for constructing erasure-coded sets can make each storage server from any *ESetGroup* belong to *I* erasure-coded sets.

Theorem 5.1. *Given a storage server $N_{a,b} \in G_0$, there are* I *and only* I *erasure-coded sets that for each erasure-coded set E_j, $N_{a,b} \in E_j$, that is V_a from E_j is selected from $N_{a,b}$, where $0 \leq a < n, 0 \leq b < I, 0 \leq i < g$, and $0 \leq j < I^2$.*

Proof of Theorem 5.1. Based on Algorithm 2, the erasure-coded sets that contain the storage server $N_{a,b} \in G_0$ are indexed from E_0 to E_{I^2-1}. We can have $0 \leq j < I^2$ in Theorem 5.1 as $N_{a,b} \in E_j$. We divide the rest of the proof into two parts. First, we prove that *I* and only *I* erasure-coded sets contain $N_{a,b}$ when *a* is equal to 0. We then discuss the case when $a \neq 0$. □

Case 1. $a = 0$. The value of *b* is equal to *j/I* if and only if $N_{a,b} \in E_j$ according to line 6 in Algorithm 2. We can have that for any E_j that can satisfy *b = j/I*, $N_{a,b} \in E_j$. As $0 \leq j < I^2$ and $0 \leq b < I$, *I* and only *I* values of *j* satisfy *b = j/I*; that is, *j* is any value from *bI* to *(bI + I - 1)*. To this end, we can conclude that *I* erasure-coded sets and only *I* erasure-coded sets contain the storage server $N_{a,b}$ when $a = 0$.

Case 2. $a \neq 0$. When $N_{a,b} \in E_j$, *b* must be equal to *((j/I)(a-1) +j)mod I* according to line 9 in Algorithm 2. If $N_{0,x} \in G_0$ satisfies the conditions that both $N_{0,x} \in E_j$ and $N_{a,b} \in E_j$, then it must satisfy the condition that *b* is equal to *(x(a-1)+j) mod I*, where $0 \leq x < I$. As $xI \leq j < xI + I$ according to the above paragraph, there will be one and only one *ESet* in which both $N_{0,x}$ and $N_{a,b} \in E_j$. Because $0 \leq x < I$, we can deduce that *I* and only *I* erasure-coded sets contain $N_{a,b}$ for the case that when the value of *a* is not equal to 0.

In a word, when any storage server fails, we have *I* erasure-coded sets to recover the failed storage server concurrently according to Theorem 5.1.

5.2 Optimal Recovery I/O Parallelism

When a storage server fails, *I* erasure-coded sets that contain the failed server will begin to recover missing blocks in the server. Each erasure-coded set will select *k* storage servers to perform recovery. A storage server that participates in recovery may be selected by more than one erasure-coded set. As a result, I/O contention may occur for the selected storage servers during recovery, which will further decrease the recovery performance. To obtain optimal *recovery I/O parallelism* when recovering a single failure, there should be no I/O contention for the recovery. To this end, we give the theorem of optimal *recovery I/O parallelism* for a storage server as follows:

Theorem 5.2. *Given a storage server $N_{x_1,y_1} \in G_0$, where $0 \leq x_1 < n, 0 \leq y_1 < I$ and $0 \leq i < g$, for any other storage server $N_{x_2,y_2} \in G_0$, where $x_1 \neq x_2, 0 \leq x_2 < n, 0 \leq y_2 < I$, if there is only one erasure-coded set that includes both N_{x_1,y_1} and N_{x_2,y_2}, then the storage server N_{x_1,y_1} can have optimal* recovery I/O parallelism.

Proof of Theorem 5.2. Suppose $N_{x_1,y_1} \in E_a$ and $N_{x_1,y_1} \in E_b$, where $0 \leq x_1 < n, 0 \leq y_1 < I$ and $a \neq b$. The I/O contention only exists when a storage server $N_{x_2,y_2} \in E_a$ *and* $N_{x_2,y_2} \in E_b$ will participate in the recovery, where $x_1 \neq x_2, 0 \leq x_2 < n, 0 \leq y_2 < I$. Because this violates the condition that $N_{x_1,y_1} \in E_a, N_{x_1,y_1} \in E_b$ and $N_{x_2,y_2} \in E_a, N_{x_2,y_2} \in E_b$ only exists when *a* = *b*. Thus, there is no I/O contention for recovering N_{x_1,y_1}, so it can have optimal *recovery I/O parallelism*. Now we give the condition in which a storage server can have optimal *recovery I/O parallelism* of our proposed ESet: □

Theorem 5.3. *For a given storage server $N_{x_1,y_1} \in G_0$, where $0 \leq y_1 < I$, it can have optimal* recovery I/O parallelism *when it satisfies one of the following conditions:*

1) *$x_1 = 0$. Namely, a storage server in the first row can always have optimal* recovery I/O parallelism
2) *For $x_1 \neq 0$. When* I *is a prime number and is equal to or greater than* n-1, *N_{x_1,y_1} can have optimal* recovery I/O parallelism

Proof of Theorem 5.3. We first prove that any storage server $N_{0,y_1} \in G_0$, where $0 \leq y_1 < I$, can always have optimal *recovery I/O parallelism*. Given a storage server $N_{x_2,y_2} \in G_0$, where $0 < x_2 < n, 0 \leq y_2 < I$. If $N_{0,y_1} \in E_a$, $N_{0,y_1} \in E_b$ and $N_{x_2,y_2} \in E_a, N_{x_2,y_2} \in E_b$, we can have $y_2 = (y_1(x_2 - 1) + a) mod I$ and $y_2 = (y_1(x_2 - 1) + b) mod I$ based on line 9 in Algorithm 2. Then we can have $(a - b) mod\ I = 0$. As *a* and *b* are values ranging from $y_1 I$ to $(y_1 + 1)I - 1$, *a=b* when $(a - b) mod\ I = 0$. Thus, only one erasure-coded set E_a satisfies $N_{0,y_1} \in E_a$ and $N_{x_2,y_2} \in E_a$. Based on Theorem 5.2, N_{0,y_1} can always have optimal *recovery I/O parallelism*. □

For $x_1 \neq 0, x_2 \neq 0$ and $x_1 \neq x_2$, if $N_{x_1,y_1} \in E_a, N_{x_1,y_1} \in E_b$ and $N_{x_2,y_2} \in E_a, N_{x_2,y_2} \in E_b$, we have the following equations:

$$y_1 = \left(\frac{a}{I}(x_1 - 1) + a\right) mod\ I \tag{1}$$

$$y_2 = \left(\frac{a}{I}(x_2 - 1) + a\right) mod\ I \tag{2}$$

$$y_1 = \left(\frac{b}{I}(x_1 - 1) + b\right) mod\ I \tag{3}$$

$$y_2 = \left(\frac{b}{I}(x_2 - 1) + b\right) mod\ I. \tag{4}$$

We have the following equation based on Equations (1), (2), (3), and (4)

$$0 = \frac{(a-b)(x_1 - x_2)}{I} mod\ I. \tag{5}$$

Because *(a-b)/I* is a value between 0 and *I-1*, we can conclude that when the value of *I* is a prime number and is equal to or greater than *n-1*, each storage server can obtain optimal *recovery I/O parallelism*. Because the absolute value of $x_1 - x_2$ is smaller than *n-1*, the only condition that satisfies Equation (5) is that *a* is equal to *b*.

The storage servers in the first row of a given *ESetGroup* can always have optimal *recovery I/O parallelism*, and when *I* is a prime number and is equal to or greater than *n-1*, any storage server from other rows can obtain optimal *recovery I/O parallelism*. In this way, the recovery time of each storage server can approximate $(ns/\beta) \times ((k \times \omega)/\phi)/I$, where ns/β is the number of blocks in each storage server and $(k \times \omega)/\phi$ is the time needed to recover a single block, which was mentioned in the last paragraph of Section 3.

We consider the case for G_0 in this section. The storage servers for any *ESetGroup* G_i still satisfy the conditions of both Theorems 5.1 and 5.3.

The algorithm of standard parity declustering [33] also seeks to obtain an optimized recovery performance. However, our ESet differs from standard parity declustering in the following three aspects based on our analysis of ESet:

1) The standard parity declustering is designed for disk-level failures. ESet can tolerate rack-level, host-level, and disk-level failures.
2) The standard parity declustering relies on the Balanced Incomplete Block Design algorithm to distribute blocks, which will perform a complex calculation before distribution. For ESet, each erasure-coded set is one unit for distributing blocks in each stripe.
3) As we have proved, when *I* is a prime number and no less than $n-1$, it can achieve optimal recovery performance. However, standard parity declustering relies on the Balanced Incomplete Block Design algorithm to give optimal recovery performance, which is difficult to find.

In summary, our placement algorithm can obtain the desired *recovery I/O parallelism*. It can achieve optimal *recovery I/O parallelism* under some mild conditions. Our algorithm differ from existing algorithms such as standard parity declustering and is appropriate to achieve fast data recovery performance for erasure-coded storage systems.

6 Design and Implementation of ESetStore

We illustrate the data placement algorithm ESet to obtain efficient reconstruction for erasure-coded storage systems. In this section, we present the design and implementation of a prototype erasure-coded storage system that integrates the data placement algorithm ESet to harvest fast data recovery. We also consider how to achieve good read and write performance for erasure-coded storage systems. We developed ESetStore in C language with around 40,000 lines of source code.

We first present the overall architecture of our ESetStore. We then illustrate how to make efficient utilization of available I/O and computation resources in ESetStore to obtain good read and write performance. We also detail the efficient reconstruction with the placement algorithm ESet. Finally, we make a discussion about how to handle various kinds of I/O pressure in storage systems.

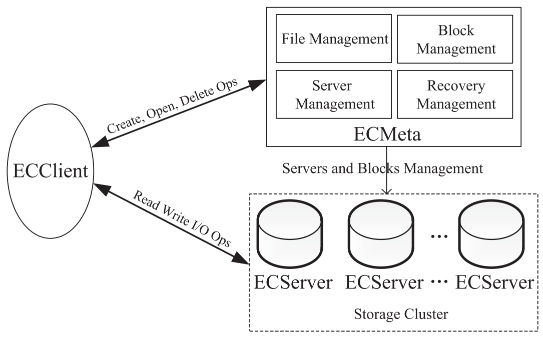


Fig. 4. Main components of ESetStore.

6.1 ESetStore Architecture

Fig. 4 illustrates the overview of ESetStore's system architecture. ESetStore consists of three major components: ECClient, a client library that allows users to interact with the storage system; ECMeta, the metadata service that manages the metadata information; and a Storage Cluster that contains many storage servers where each one is deployed with the storage service ECServer to store raw data.

ECClient. The ECClient is a library that provides fundamental operations for users to interact with the storage system. The three functions are create, open, and delete. When a client issues a create request to the system, the command is sent to the ECMeta. ECMeta returns a set of *n* storage servers for storing files and some additional information about how to encode the files. To open a file, a set of *k* storage servers that contain any data or parity blocks of the file will be returned. When it sends a delete command, the file information is deleted from the metadata server, and the file's content is deleted by the storage servers.

Two fundamental functions for accessing file content are read and write. When *n* storage servers are provided, the write function writes *n* data and parity blocks of a file to these storage servers. To read the content of a file from the servers, the ECClient first selects *k* storage servers that contain the data blocks to retrieve data from the storage system. In case that any storage server that contains data blocks failed, some parity blocks will be downloaded, and the decoding operation is performed to restore the file.

ECMeta. Each operation issued by users will first interact with the ECMeta. It records the file created by users via file management. It also manages the relationship between stripes and files by Block Management module. Monitoring the state of each storage server is also performed by the metadata service. It interacts with each storage server to periodically check its status. It is also responsible for conducting recovery of any failed storage server.

The metadata service is responsible for managing the metadata information of each file, and it records the location of each block and manages the relationship between blocks and stripes. When any storage server fails, it selects related erasure-coded sets for which each erasure-coded set contains the failed server to perform recovery.

Storage Cluster. The storage cluster contains many storage servers to store raw data. Each storage server contains a

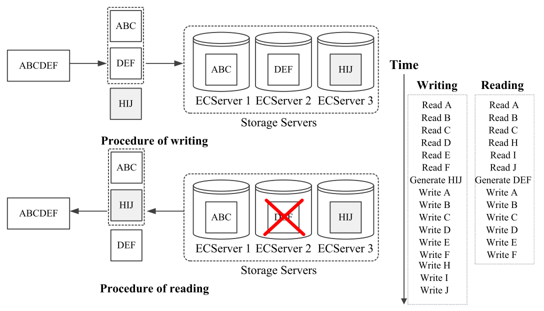


Fig. 5. Example of read and write operations.

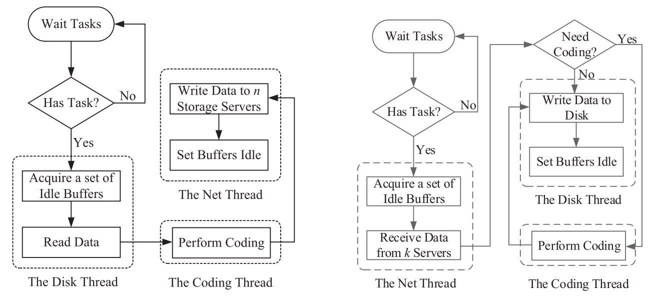


Fig. 7. Read and write operation in ESetStore.

storage service named ECServer. ECServer provides interfaces that help users access data on each storage server. When users write data to a storage server, it receives the data and writes them to the local disk. If the storage server receives the read request from users, the ECServer reads the data from the local disk and sends the data to the user. ECServer also assists ECMeta in recovering missing data.

6.2 Efficient Read and Write Operations

Accessing data is a constantly performed task in storage systems. Read and write operations are two fundamental operations for accessing data in a storage system. A write operation in erasure-coded storage systems involves three steps. First, the data are read from disk to memory. The encoding operation is then performed to generate k data chunks and n-k parity chunks. Finally, these chunks are written to n storage servers. When reading a file, the ECClient gathers data from k storage servers. If any of the k chunks retrieved from storage servers belong to the n-k parity chunk, the decoding operation must be performed before constructing the file. When decoding is required to read a file, we call it *degrade read*. The read operation without decoding is called *normal read*. To this end, effectively utilizing both I/O and computational resources are the key factors in achieving good read and write performance.

Fig. 5 presents a simple example of read and write operations. In this example, we set n as 3 and k as 2. The file content is denoted with a string *ABCDEF*. We can see that H is equal to $A \oplus D$, I is equal to $B \oplus E$, and J is equal to $C \oplus F$. The right side of Fig. 5 demonstrates the timeline of read and write operations. One read or write operation is performed at each time slot. Note that we assume the ECServer 2 failed in the read operation in the example. We perform a degrade read to retrieve the file with decoding operation involved. As for a normal read when ECServer 2 is available, we need only retrieve data chunks from ECServer 1 and ECServer 2. The read and write operations in this way may take a rather long time because each step takes an exclusive time slot.

One solution to improve read and write performance is to utilize the I/O resources of both the client side and the server side. Fig. 6 illustrates the method of overlapping operations to accelerate the performance of read and write operations. We divide each chunk into many small sub-chunks for each read and write step in our operation. To write data to servers, we read A and B and then encode them into O. We then write A, B, and O into three servers. We write C, D, and P in a similar way. The timeline at the bottom of Fig. 6 reveals that the writing, coding, and reading steps can proceed in parallel to better utilize the I/O resources.

In this way, we can make better use of I/O resources. Via numeric calculation, our proposed method takes 3/4 of the way in Fig. 5 for writing, and the time for reading is 9/13 of that in Fig. 5 for reading. To effectively overlap I/O and coding operations as illustrated in Fig. 6, we define a concept called *streaming size* as follows:

The streaming size is a configured fixed-length (e.g., 64 KB) to manipulate data. The block size is a multiple of the streaming size. When writing a file, the file is divided into one or more sets of k data blocks. Each block is further divided into many small chunks whose size is the streaming size. The streaming size can be the minimum size for handling both read and write operations.

For read and write operations, we have two sets of buffers. When we allocate a set of buffers to handle read and write operations in ESetStore, the size of each buffer is a multiple of the streaming size. Each set of buffers has two states. One state is idle when no thread is performing a task on them. Another state is busy when a thread is conducting a read or write operation on it. We have three threads to carry read and write operations with these buffers. One thread is the disk thread, which is responsible for reading data from disks into the buffers or writing data from buffers to a disk. The coding thread is responsible for coding. The network thread is responsible for sending data to other storage servers or receiving data from other storage servers.

The procedure of the write operation in ESetStore is presented in Fig. 7a. The disk thread first selects a set of n buffers that remain idle and mark the set's state as busy. It then reads data from the disk into the k buffers. After completing reading data into k buffers, it notifies the coding thread to handle the set of buffers. The disk thread continues to acquire a set of idle buffers and reading data into them until the file data are read from disks. The coding thread then generates parity data and fills them into n-k buffers. It then

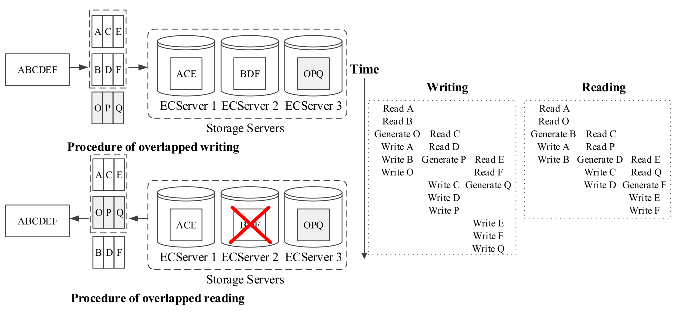


Fig. 6. Read and write operations with overlapped I/O and coding.

notifies the coding thread to send data to n storage servers. The network thread receives the set of buffers from the coding thread and sends data from n buffers to n storage servers. It then sets the state of the set of buffers to idle.

Reading works in the opposite way. The procedure of the read operation in ESetStore is presented in Fig. 7b. The network thread continues to request a set of buffers, marks the set's state as busy, and receives data from n storage servers until all tasks are complete. The coding thread performs decoding when any data block is missing. The disk thread writes data into a disk when a set of buffers is ready for writing. It then sets the set of buffers' state to idle when all data in the buffer have been written to a disk.

6.3 Data Distribution and Efficient Reconstruction

We use the data placement algorithm ESet to decide where to store each stripe in our ESetStore. After partitioning the storage servers into many erasure-coded sets, we use an array to store the entire erasure-coded sets with each erasure-coded set containing n storage servers. When storing a file to ESetStore, the file is encoded into a stripe, and we then select an erasure-coded set to carry the stripe for the file. To evenly distribute the data across storage servers, we select the erasure-coded sets in a round-robin fashion.

When a storage server fails, we must recover the missing blocks. Algorithm 3 presents a high-level workflow of the recovery process. First, it finds I erasure-coded sets that contain the failed server. Each of the I erasure-coded sets then selects k available storage servers to recover missing blocks. Each erasure-coded set recovers one missing block from the set at a time, and it continues to recover missing blocks until all missing blocks from the erasure-coded sets are recovered. The recovery is complete when each erasure-coded set has recovered its missing blocks.

Algorithm 3. High-Level Workflow of Recovery

```
1: Input: Erasure-coded sets, $N_{i,j}$
2: Initialize RecoverySet
3: Find I erasure-coded sets that contain $N_{i,j}$
4: Put I erasure-coded sets in RecoverySet
5: for each $E_i \in RecoverySet$ do
6:   Select k available storage servers in $E_i$
7:   for each missing block $\in E_i$ do
8:     Recover the missing block with selected servers
9:   end for
10: end for
```

6.3.1 I *Erasure-Coded Sets for a Failed Storage Server*

To recover a failed server, we need to locate I erasure-coded sets. We first consider the case in which we have only one group in our storage system. Erasure-coded sets are indexed from E_0 to E_{I^2-1}. The storage servers are indexed from $N_{0,0}$ to $N_{n-1,I-1}$. The algorithm to find I erasure-coded sets for a given storage server $N_{i,j}$ is presented in Algorithm 4.

When the failed server is in the first row of the group, its erasure-coded sets are located from line 5 to line 7 in Algorithm 4. They are indexed from $I * j$ to $(I+1) * j - 1$. For the case in which any failed storage server is in other rows, any erasure-coded set with its index satisfies the condition that at line 10 contains the storage server $N_{i,j}$. Here, the index of erasure-coded sets is from 0 to $I^2 - 1$. When the storage system contains many groups, locating I erasure-coded sets for a storage system with many *ESetGroups* is similar to the case in which we have only one *ESetGroup*. We must add the value, which is calculated by multiplying the index of the given *ESetGroup* by I^2, with each index of the located erasure-coded set by Algorithm 4.

Algorithm 4. Find I Erasure-Coded Sets That Contain $N_{i,j}$

```
1: Input: $E_0$ to $E_{I^2-1}$, I and $N_{i,j}$
2: Output: RecoverySet
3: Initialize RecoverySet
4: if $i == 0$ then
5:   for each $idx \in [I * j, (I+1) * j - 1]$ do
6:     Put $E_{idx}$ in RecoverySet
7:   end for
8: else
9:   for each $idx \in [0, I^2 - 1]$ do
10:    if j==((idx/I)(i-1)+idx)mod I then
11:      Put $E_{idx}$ in RecoverySet
12:    end if
13:  end for
14: end if
15: return RecoverySet
```

After locating I erasure-coded sets to recover a failed storage server, we select one storage server from each erasure-coded set responsible for recovering missing blocks. To recover a failed stripe in an erasure-coded set, the selected storage server in each erasure-coded set reads *k-1* blocks of the stripe from other *k-1* storage servers and reads one local block from its own disk. It then reproduces the missing block and caches the recovered block in its own disk. The selected storage servers in each erasure-coded set repeat the process until all missing blocks are recovered. The ESetStore is expected to have I times of performance growth to recover a failed storage server in an optimal situation compared with the case in which the value of I is 1.

6.3.2 *Recovery of Each Erasure-Coded Set in ESetStore*

We use a simple approach to recover missing blocks. After selecting k available storage servers from the given erasure-coded set, we select one storage server responsible for recovery. The selected storage server for gathering blocks and decoding missing blocks is called the head server.

When the head server receives the command to recover a missing block, it recovers a portion of data from the missing block as presented in Fig. 8. Note that the k blocks in Fig. 8 belong to the same stripe of the missing block. It first gathers k small chunks to the head server, and then uses a decoding operation to restore the part of data from the missing block and caches the restored data to its local disk. The operation in Fig. 8 is repeated until the whole missing block is recovered. It then informs the metadata server that the missing block is restored. The metadata server continues to send the command to the head server to recover a missing block until all missing blocks are recovered.

To make efficient use of available I/O and computation resources in the head server, it uses the same approach illustrated in Fig. 7b when reading *k-1* blocks.

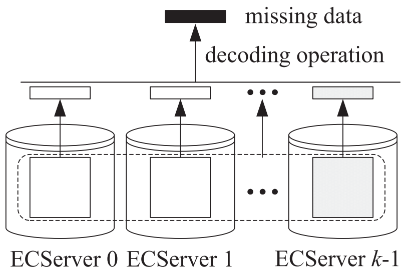


Fig. 8. Recovery missing data in an erasure-coded set.

6.3.3 Recovery With PPR in ESetStore

We can see that the above recovery performance for each erasure-coded set is limited by the head server's network bandwidth. A study named PPR [15] proposed a solution to make better use of the available network bandwidth from each set of k selected storage servers.

Our implementation of the PPR is illustrated in Fig. 9. The storage servers are indexed from ECServer 0 to ECServer *k-1*. The last storage server sends the required subblock to ECServer *k-2* to generate a temporary block. The temporary block is then sent to the former storage server. The head server continues to receive temporary blocks and restore missing small blocks, and the other servers continues to generate temporary blocks and send the temporary block to its former server until all missing data are restored. The entire process can be well pipelined so that we can make full use of the available I/O resources to perform recovery.

6.4 Limitations and Discussions

Our current design of ESetStore mainly addresses the issue of storage efficiency (i.e., using erasure coding) and recovery performance (i.e., through ESet), and meanwhile optimizes the read and write throughput. We consider recovery performance as the major design factor because in (n, k) erasure-coded system, the required I/O for recovery is k times of that of replication based system. Currently we have not considered other factors like data access dynamics and data locality.

Storage systems may encounter various kinds of I/O pressure. A storage system may have poor read performance when providing limited I/O resources for hot data, which are frequently accessed by clients in a short period. ESetStore does not implement any strategy to deal with hot data. We discuss some existing solutions that could be used to tackle this issue. As revealed in the study of Facebook's system f4 [7], newly created data in storage systems typically have a higher access rate. It is possible to distribute data in a round-robin manner to provide more I/O resources for hot data. The f4 system also uses a caching stack to reduces the load on the storage system. As a result, system administrators can use in-memory storage systems such as Memcache [39], [40] to reduce I/O pressure on our ESetStore. A recent erasure-coded storage system EC-Store is designed for various kinds of I/O workload [41]. It uses data access patterns and data movement strategies to improve the distribution of workload and provide efficient retrieval for erasure-coded storage systems. It is possible to adopt its strategy and run it on top of our ESet algorithm to provide better I/O performance for various access patterns.

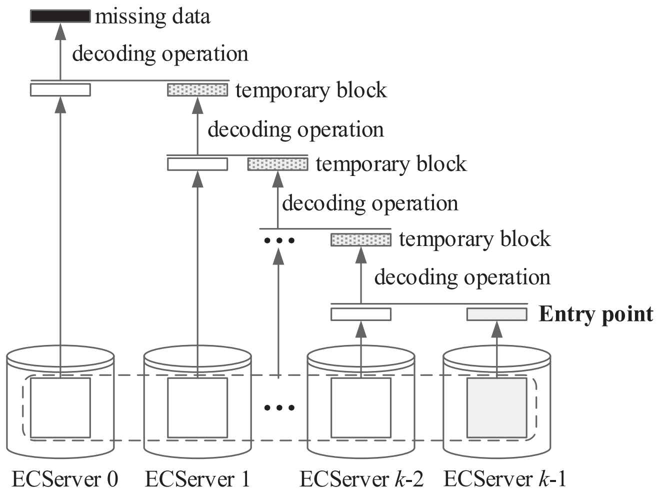


Fig. 9. Recovery of missing data with PPR in an erasure-coded set.

When using replication as a reliability mechanism, storage systems like HDFS can take advantage of data locality to better support computation tasks such as Map-Reduce. When one storage server failed, some other storage servers still contain the required data blocks. However, with erasure coding, any missing data block is required to be restored before performing computation tasks. The erasure-coded version of HDFS must gather k blocks to restore the missing block before performing computation tasks. A study has been proposed to run degraded tasks earlier to reduce the performance penalty [42]. The XORing Elephants used LRC codes to reduce repair costs for HDFS [23]. Currently, ESetStore does not consider how to exploit data locality to support computing-oriented applications. We will leave it as our future work.

7 Evaluation of ESetStore

In this section, we conduct a set of experiments to evaluate the performance of ESetStore using a cluster of up to 50 computers. We first introduce the experimental setup of our evaluation, and then measure the read and write throughput of ESetStore and compare it with HDFS using different settings of block size. The version of HDFS is Hadoop 3.0.0-alpha2 [8]. Many previous studies used HDFS as a baseline [14], [23], so we also choose it for comparison. We also evaluate the overall throughput of read and write operations on a small cluster and compare it with Ceph of version 13.2.6 [43]. Afterwards, we measure the recovery performance of ESetStore and compare it with both HDFS and Ceph. We demonstrate the recovery performance of ESetStore with the optimal I/O parallelism. Finally, we evaluate the system recovery performance with a single failure.

7.1 Experimental Setup

Fig. 10 presents the testbed used to the evaluate ESetStore, HDFS and Ceph. Each storage server is equipped with one 1 Gbps Ethernet card connected to a 1 Gbps switch. The testbed has 50 servers in total. Each storage server has a disk with the peak read and write throughput around 100 MB/s. The metadata service ECMeta is deployed in a high-performance server with 12 CPU cores and a 10 Gbps Ethernet card to provide low latency metadata service. It is connected to a

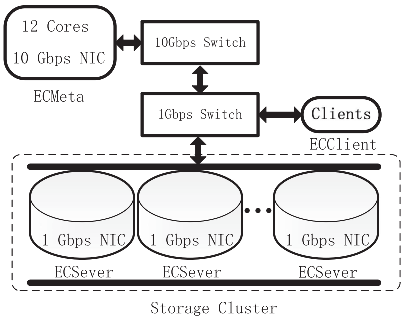


Fig. 10. Our testbed for performance evaluation.

10 Gbps switch which is connected to the 10 Gbps uplink port of the 1 Gbps switch.

To measure the read and write throughput achievable by a single client in Section 7.2.1, we vary the values of block size from 1 to 128 MB. We perform one read and write operation of a file for each measurement. The file size is the value of k multiplied by the block size. The total size written to the storage servers is the value of n multiplied by the block size, and the total size read from the storage server is the value of k multiplied by the block size. The throughput is calculated from dividing the file size by the whole execution time. We set the *streaming size* as 64 KB in each set of experiments. The buffer of each block is 256 KB. Each client can read and write 256 KB of data for each buffer.

When measuring the overall system throughput in Section 7.2.2, we set the size of each object as 16 MB, which means the file size of ESetStore is 16 MB. We evaluate a small cluster with 18 storage servers and set n as 3 and k as 2. We use a popular tool IOR [44] to measure the performance of our storage cluster with intensive I/O operations. We implement the necessary APIs for IOR to perform the read and write operations with ESetStore, and the source code can be found at [44]. We measure the read and write performance with up to 16 clients that are deployed on 16 machines in two cases: (1) without storage server failure; (2) with a single storage server failure.

When evaluating the single recovery performance of ESetStore in Sections 7.3 and 7.4, we set the value of I from one to seven. We have I hosts in each rack for each measurement. Each host uses one disk to store data. We have n racks, and the number of storage servers is nI. Our ESetStore has n racks to form one *ESetGroup* and contains I^2 erasure-coded sets. We use the same physical configuration to measure the performance of HDFS and Ceph for comparison.

The configuration for recovery experiments is presented in Table 2. The file size is $64 \times k$ MB. The default setting is used for ESetStore and HDFS. For Ceph, the block size refers to the data size written to each disk when writing an object. We use the default CRUSH setting and set the placement groups as the maximum number allowed, selected from 128, 256, 512, and 1024.

We write around 1 GB data to each storage server, and then manually shut down one server to measure the recovery time. The throughput is calculated by dividing the total recovered data size by the total recovery time in each measurement. We evaluate the recovery performance of ESetStore with the simple approach presented in Fig. 8. We also evaluate the recovery performance of ESetStore with the PPR adopted to achieve optimal recovery performance.

TABLE 2
The System Configuration for Recovery Performance Comparison

	OS	Block Size	Placement Algorithm
ESetStore	Ubuntu 14.04	64 MB	ESet
HDFS	Ubuntu 14.04	64 MB	Random Placement
Ceph	Ubuntu 16.04	4 MB	CRUSH

When evaluating the impact of recovery operation on the data access performance in Section 7.5, we write $256 \times I$ files to our storage system and then retrieve the files using multiple client machines. The size of each file is $4 \times k$ MB. So each storage server stores 1 GB of data. Both ESetStore and Ceph use $k \times I$ number of client machines to retrieve all files. We evaluate the performance of two cases: (1) no storage server is failed; (2) a single storage server is failed and the recovery procedure is performed.

7.2 Read and Write Throughput

In this subsection, we measure the read and write throughput of ESetStore and compare it with HDFS and Ceph. We evaluate both the normal data access and degraded data access when there is a single failure of storage server. In this subsection, data recovery will not be triggered upon the failure of a storage server. The performance of simultaneous data access and data recovery will be presented in Section 7.5.

7.2.1 *Single Client Performance: Comparison With HDFS*

We first investigate the impact of block size on the read/write throughput of a single client. Fig. 11 presents the read throughput of ESetStore and HDFS with different settings. In Fig. 11, ESetStore R, HDFS R, and Baseline R represent the cases without any storage server failure. On the contrary, ESetStore Degrade R, HDFS Degrade R, and Baseline Degrade R represent the cases with one storage server failure which contains the required data block. The baseline refers to the special version of ESetStore without the optimization strategies in Section 6.

The read throughput of ESetStore is about 50 MB/s when the block size is 1 MB when no server failure occurs. When the value of k is 3, the performance is slightly better than when k is 2. It is obvious that the read performance increases gradually as the data block size increases. The throughput reaches its maximum (which is limited by the network bandwidth) when the size of each block is 64 MB. The performance penalty of the degrade read is about 10 percent of normal read, attributed to the overlapping of I/O operations and computing operations.

We use the baseline and the HDFS for comparison. The results of the baseline reveal that the system can encounter various kinds of performance penalties at both the client side and the server side, when no optimization strategy is applied to our system. The HDFS does not make good use of available I/O resources to obtain high read throughput. It requires a larger block size to obtain better read performance. No obvious difference is observed between its normal read and degrade read. The under-utilization of I/O resources could be the major result of its performance degradation

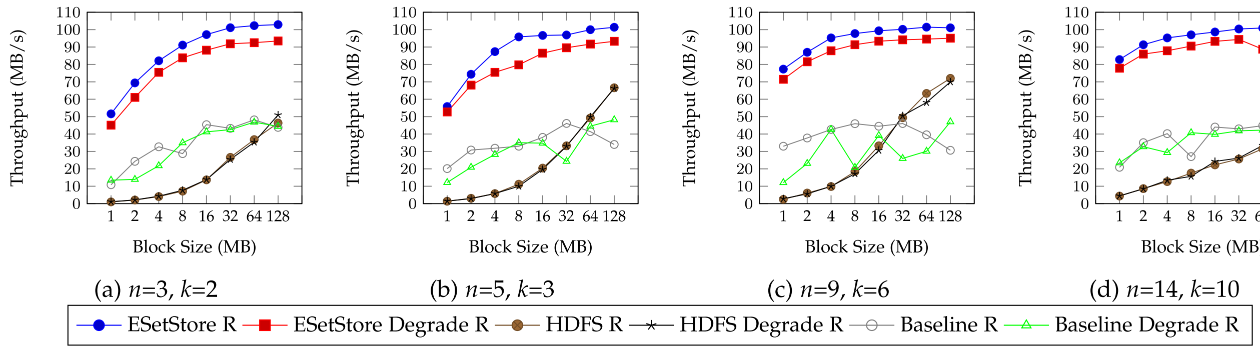


Fig. 11. Single client read throughput of ESetStore and HDFS.

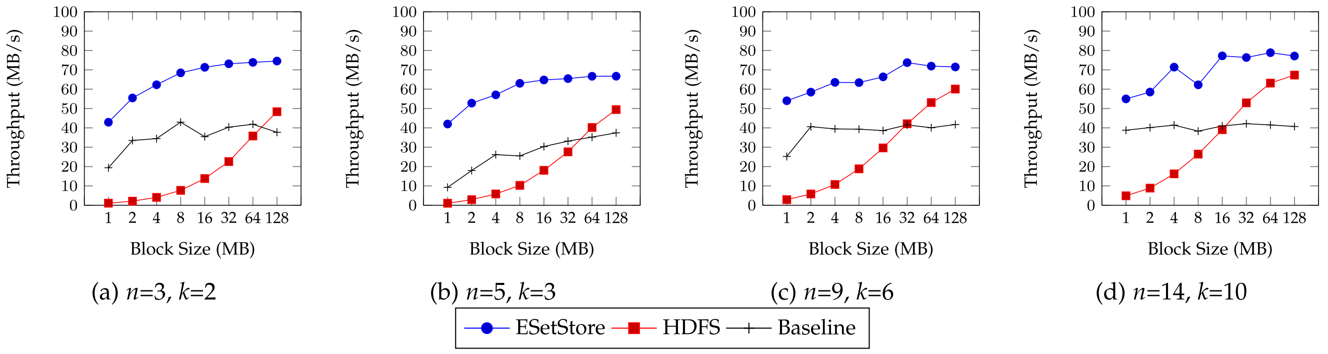


Fig. 12. Single client write throughput of ESetStore and HDFS.

when performing reads. This also indicates that our optimization strategy is effective and achieve show good read and write performance in erasure-coded storage systems.

Fig. 12 presents the write throughput of ESetStore and HDFS. The write throughput is k/n of the raw throughput because writing parity blocks account a part of the total execution time. This is why the case when n is 3 and k is 2 has better write throughput than the case when n is 5 and k is 3. The case of k is 2 has nearly the same throughput as that when k is 3 for the block size is 1 MB because the buffer is set to $256k$ bytes in each write operation.

The baseline version can have better performance with a larger block size. However, the write throughput is limited to around 45 MB/s because each step occupies a time slot exclusively. The write throughput of HDFS has similar performance to its read throughput. The performance increases as the block size increases. According to our experiments, ESetStore has much better write throughput than either the baseline version or the HDFS.

7.2.2 Overall Performance: Comparison With Ceph

The overall read/write throughput is also important to a distributed storage cluster. We use up to 16 client machines to measure the overall throughput of a cluster of 18 storage servers. We set I to 6, n to 3 and k to 2 in our storage cluster. We set the transfer size for each read and write to 2 MB, and the size of each file as 16 MB. Each measurement iterates 10 times and we pick the maximum overall throughput to represent the peak performance that the cluster can reach. The experimental results are shown in Fig. 13.

In Fig. 13, Single Failure denotes the case when the storage cluster has one failed storage server. From the figure, we can see that single failure has no obvious impact on the overall read/write throughput of both ESetStore and Ceph. When the number of clients increases from 1 to 2 and from 2 to 4, we can see the throughput is nearly doubled. But when the number of clients grows from 4 to 8 and 8 to 16, there are at most 50 percent performance growth. This could be caused by I/O contention when we have more clients issuing read/write requests. In all cases, we can see that ESetStore achieves much higher overall read/write throughput than Ceph.

7.3 Recovery Performance of Single Failure

The recovery performance of ESetStore is presented in Fig. 14. We set the value of k from 2 to 5. The value of $n-k$ can be set to 1 for two reasons. First, the value of $n-k$ as 1 is adequate to tolerate a single failure. Meanwhile, the smaller

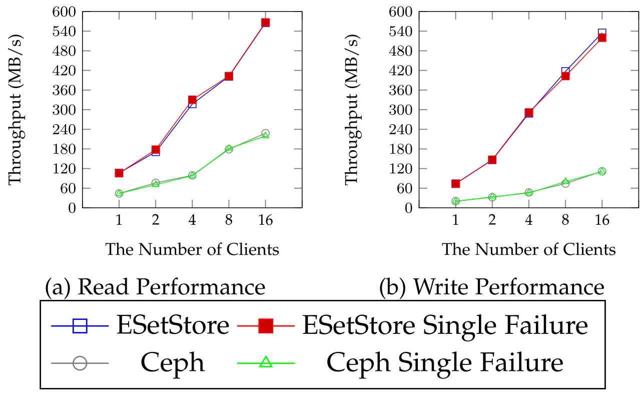


Fig. 13. The overall read/write throughput of ESetStore and Ceph. $n=3, k=2$.

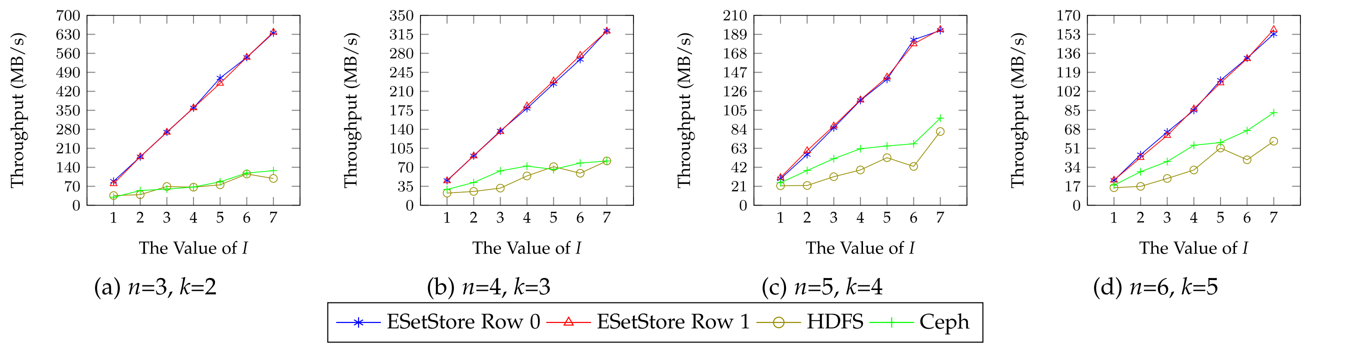


Fig. 14. Recovery performance of ESetStore.

value of $n-k$ means a reduction in coding overhead, which could better reveal the overhead of I/O operations.

We conduct two sets of experiments to measure the recovery performance of ESetStore. The first is ESetStore Row 0. We manually shutdown one storage server in the first row (row 0) in the *ESetGroup*. For ESetStore Row 1, we manually shutdown one storage server in the second row (row 1) in the *ESetGroup*. The storage servers in the first row always obtain optimal recovery performance because there will be k distinct storage servers to recover missing blocks in each *ESet*. However, whether the storage servers in other rows can obtain optimal recovery performance relies on the value of I as analyzed in Section 5.

From Fig. 14, we can see that the recovery performance is nearly $1/(k-1)$ of the maximum read throughput when the value of I is one because we must read $k-1$ blocks from other servers to recover a missing block.

A storage server from the first row will have k distinct servers in each erasure-coded set to recover all missing blocks that belong to the erasure-coded set. It can thus obtain I times of performance growth as compared to the situation in which I is equal to 1. When k is 3 when recovering a failed storage server from the second, a storage server that participates in recovery belongs to two erasure-coded sets when I is 2 and 4. However, because each erasure-coded set has only around 45 MB/s network bandwidth, its recovery performance is nearly the same as that of the first row. We can thus obtain I times of performance growth to recover a failed storage server in all cases compared with that in which the value of I is 1.

The evaluation result in ESetStore is close to our analysis that the recovery time is approximate $(ns/\beta) \times ((k \times \omega)/\phi)/I$ in Section 3. We didn't use the same setting as the one in Fig. 7b and 7, because the required machines $I * n$ for storage servers will exceed the number of our testbed. However, the recovery performance can still approximate $(ns/\beta) \times ((k \times \omega)/\phi)/I$ for these configuration. Because the main performance penalty for the recovery is the limited I/O resources, and our ESet can increase the I/O resources for recovery with the increases of value I.

The base performance, where I is 1, of HDFS and Ceph is slightly worse than that of our ESetStore, and we can see that there are fewer cases in which the recovery performance increases as I increases for HDFS. HDFS uses a random placement algorithm, which indicates that the algorithm cannot make as good use of the available disk I/O and network bandwidth resources as our ESetStore.

We set the number of threads for recovery to 8 for Ceph to achieve good recovery performance. Ceph showed slightly better performance than HDFS. It relies on more placement groups to achieve better performance. However, it can only achieve around 3x performance growth when I increases from 1 to 7. Thus, we can conclude that our ESetStore can harvest available I/O resources to improve the recovery performance with the placement algorithm ESet.

7.4 Performance of Optimal Recovery I/O Parallelism

We can observe from Fig. 14 that as k increases, the recovery performance suffers a great decrease because the recovery performance is limited by the network bandwidth of a single storage server. Here we implement the PPR algorithm in our ESetStore to evaluate the performance of optimal recovery I/O parallelism. With PPR, each erasure-coded set can make good use of the available I/O resources.

The evaluation results are presented in Fig. 15. We measured the recovery performance of a failed storage server from row 0 and a failed storage server from row 1. The evaluation results differ greatly from that in Fig. 14. The performance is far better due to the better utilization of available I/O resources in each erasure-coded set.

To recover a single failure from row 0, the performance can still show linear growth with an increase of I. However, when recovering a failed storage server from row 1, some cases show no obvious performance increase. For example, when k is 5, there is no performance growth for I from 1 to 4 because I is smaller than *n-1*. In these cases, some storage servers concurrently participate in the recovery of a failed storage server in more than one erasure-coded set, resulting in I/O contention. As a consequence, there is no increase of I/O resources to recover the failed server.

In addition, in some cases the performance of row 1 is around half that of row 0, such as when k is 3 and I is 4, because there is a storage server staying in two erasure-coded sets that contains the failed storage server. As a result, the I/O resources to recover the failed storage server is reduced to half for row 1.

In summary, the recovery performance is always optimal for the storage servers in row 0. The storage servers in other rows can obtain optimal recovery performance only when the configuration satisfies the optimal recovery I/O parallelism, namely, that I is a prime number and its value is equal to or greater than *n-1*. Our evaluation also indicates that an optimization solution, such as PPR, may become useless in

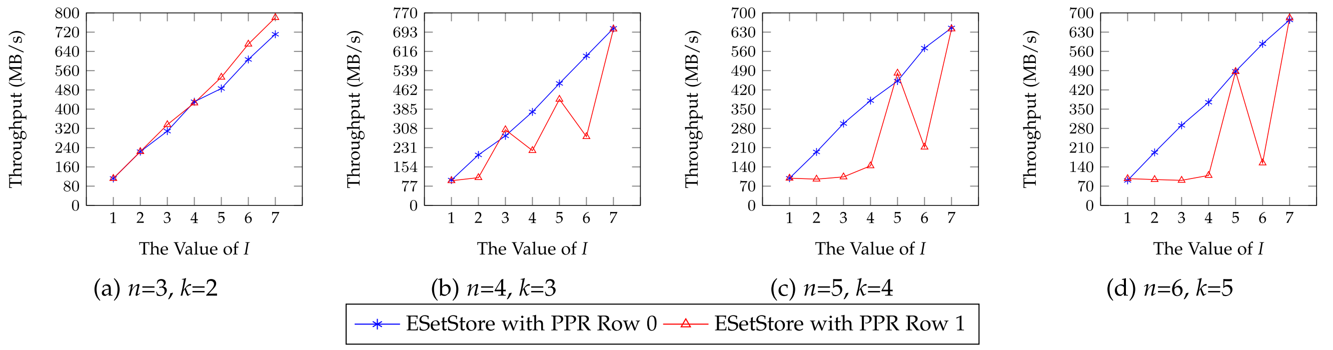


Fig. 15. Recovery performance of ESetStore with PPR.

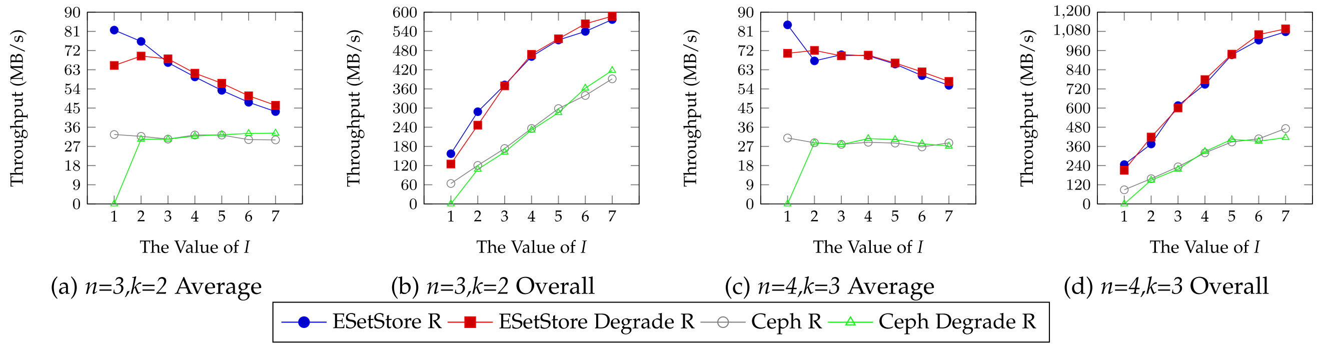


Fig. 16. Read throughput with recovery operation.

erasure-coded storage systems if the stripes are not placed in a proper way with a data placement algorithm.

7.5 Impact of Recovery on Data Access Performance

Although ESetStore is designed for efficient data recovery, it is still important to evaluate the data access performance while the system is during the recovery process. We present the experimental results in Fig. 16, where we measure the data access throughput while the system is undergoing the recovery process. The Average means the mean throughput of all clients, which is calculated by dividing the single file size by the average file downloading time. The Overall is the total aggregated system throughput for accessing all files, which is calculated by dividing the size of all retrieved files by the total time of downloading all files.

We can observe that the throughput of a single client in ESetStore decreases with the increase of I. This is because more clients result in more I/O contentions. It also explains why the overall throughput does not achieve linear performance growth with the increase of I. When I is 1 or 2, we observe that the data access performance with ongoing data recovery is slightly worse than the one without server failure. However, for larger values of I, the impact of data recovery on the data access performance is negligible.

For Ceph, the read operation cannot be performed when the value of I is 1 when one storage server failed. That is why the throughput is zero in Fig. 16 for Ceph Degrade R. For other cases, the Ceph Degrade R and Ceph R have similar performance. In all cases, ESetStore shows superior data access throughput than Ceph when the system is undergoing data recovery.

8 Conclusion

In this paper, we present the design and implementation of our data placement algorithm, ESet, on the prototype of an erasure-coded storage system ESetStore. Our results demonstrate that ESetStore can effectively improve the single recovery performance by harvesting available I/O resources with ESet. Meanwhile, our recovery solution can be an enhancement to existing optimizations such as the parallel partial repair algorithm. When the value of parameter recovery I/O parallelism I is a primer number and is equal to or greater than *n-1*, each storage server can make good use of the available I/O resources to obtain optimal recovery performance.

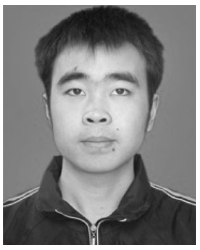
Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments. This work was supported by Hong Kong Innovation and Technology Fund ITS/443/16FX. Chengjian Liu and Qiang Wang contributed equally to this work.

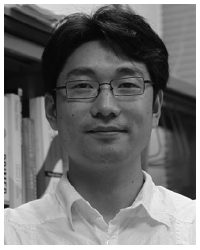
References

[1] M. Grawinkel, L. Nagel, M. Mäsker, F. Padua, A. Brinkmann, and L. Sorth, "Analysis of the ECMWF storage landscape," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 15–27.
[2] L. Papageorgiou, P. Eleni, S. Raftopoulou, M. Mantaiou, V. Megalooikonomou, and D. Vlachakis, "Genomic big data hitting the storage bottleneck," *EMBnet. J.*, vol. 24, 2018.
[3] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proc. 19th ACM Symp. Operating Syst. Princ.*, 2003, pp. 29–43.
[4] Z. Zhang, A. Deshpande, X. Ma, E. Thereska, and D. Narayanan, "Does erasure coding have a role to play in my data center," *Microsoft Res., Redmond, WA, USA, Tech. Rep. MSR-TR-2010*, 2010.

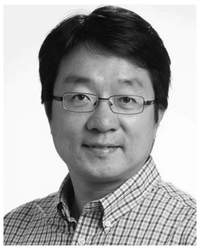
[5] C. Huang *et al.*, "Erasure coding in windows azure storage," in *Proc. Usenix Annu. Tech. Conf.*, 2012, pp. 15–26.
[6] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster," in *Proc. 5th USENIX Workshop Hot Topics Storage File Syst.*, 2013, Art. no. 8.
[7] S. Muralidhar *et al.*, "F4: Facebook's warm BLOB storage system," in *Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 383–398.
[8] A. S. Foundation, "HDFS erasure coding," 2017. [Online]. Available: https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html
[9] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. 7th Symp. Operating Syst. Des. Implementation*, 2006, pp. 307–320.
[10] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, "The quantcast file system," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1092–1101, 2013.
[11] O. Khan, R. C. Burns, J. S. Plank, W. Pierce, and C. Huang, "Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, Art. no. 20.
[12] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?" in *Proc. 5th USENIX Conf. File Storage Technol.*, 2007, pp. 1–16.
[13] Y. Zhu, P. P. Lee, Y. Xu, Y. Hu, and L. Xiang, "On the speedup of recovery in large-scale erasure-coded storage systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 7, pp. 1830–1840, Jul. 2014.
[14] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A Hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 331–342, 2014.
[15] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi, "Partial-parallel-repair (PPR): A distributed technique for repairing erasure coded storage," in *Proc. 11th Eur. Conf. Comput. Syst.*, 2016, Art. no. 30.
[16] R. Li, X. Li, P. P. Lee, and Q. Huang, "Repair pipelining for erasure-coded storage," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 567–579.
[17] C. Liu, X. Chu, H. Liu, and Y.-W. Leung, "ESet: Placing data towards efficient recovery for large-scale erasure-coded storage systems," in *Proc. 25th Int. Conf. Comput. Commun. Netw.*, 2016, pp. 1–9.
[18] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Ind. Appl. Math.*, vol. 8, no. 2, pp. 300–304, 1960.
[19] M. Blaum and R. M. Roth, "On lowest density MDS codes," *IEEE Trans. Inf. Theory*, vol. 45, no. 1, pp. 46–59, Jan. 1999.
[20] D. Ford *et al.*, "Availability in globally distributed storage systems," in *Proc. 9th USENIX Conf. Operating Syst. Des. Implementation*, 2010, pp. 1–7.
[21] A. Cidon, S. M. Rumble, R. Stutsman, S. Katti, J. K. Ousterhout, and M. Rosenblum, "Copysets: Reducing the frequency of data loss in cloud storage," in *Proc. Usenix Annu. Tech. Conf.*, 2013, pp. 37–48.
[22] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "CRUSH: Controlled, scalable, decentralized placement of replicated data," in *Proc. ACM/IEEE Conf. Supercomput.*, 2006, Art. no. 122.
[23] M. Sathiamoorthy *et al.*, "XORing elephants: Novel erasure codes for big data," *Proc. VLDB Endowment*, vol. 6, no. 5, pp. 325–336, 2013.
[24] M. L. Curry, A. Skjellum, H. Lee Ward, and R. Brightwell, "Gibraltar: A Reed-Solomon coding library for storage applications on programmable graphics processors," *Concurrency Comput.: Practice Experience*, vol. 23, no. 18, pp. 2477–2495, 2011.
[25] X. Chu, C. Liu, K. Ouyang, L. S. Yung, H. Liu, and Y.-W. Leung, "PErasure: A parallel cauchy reed-solomon coding library for GPUs," in *Proc. IEEE Int. Conf. Commun.*, 2015, pp. 436–441.
[26] C. Liu, Q. Wang, X. Chu, and Y.-W. Leung, "G-CRS: GPU accelerated cauchy reed-solomon coding," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 7, pp. 1484–1498, Jul. 2018.
[27] L. Xiang, Y. Xu, J. Lui, and Q. Chang, "Optimal recovery of single disk failure in RDP code storage systems," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 1, pp. 119–130, 2010.
[28] S. Xu *et al.*, "Single disk failure recovery for X-code-based parallel storage systems," *IEEE Trans. Comput.*, vol. 63, no. 4, pp. 995–1007, Apr. 2014.
[29] Z. Shen, J. Shu, P. P. Lee, and Y. Fu, "Seek-efficient I/O optimization in single failure recovery for XOR-coded storage systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 3, pp. 877–890, Mar. 2017.
[30] D. S. Papailiopoulos and A. G. Dimakis, "Locally repairable codes," *IEEE Trans. Inf. Theory*, vol. 60, no. 10, pp. 5843–5855, Oct. 2014.
[31] A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh, "A survey on network codes for distributed storage," in *Proc. IEEE*, vol. 99, no. 3, pp. 476–489, 2011.
[32] K. Rashmi, N. B. Shah, and K. Ramchandran, "A piggybacking design framework for read-and download-efficient distributed storage codes," in *Proc. IEEE Int. Symp. Inf. Theory Proc.*, 2013, pp. 331–335.
[33] M. Holland and G. A. Gibson, "Parity declustering for continuous operation in redundant disk arrays," in *Proc. 5th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 1992, pp. 23–35.
[34] G. A. Alvarez, W. A. Burkhard, L. J. Stockmeyer, and F. Cristian, "Declustered disk array architectures with optimal and near-optimal parallelism," *ACM SIGARCH Comput. Archit. News*, vol. 26, no. 3, pp. 109–120, 1998.
[35] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–10.
[36] R. J. Honicky and E. L. Miller, "A fast algorithm for online placement and reorganization of replicated data," in *Proc. Int. Parallel Distrib. Process. Symp.*, 2003, pp. 10–pp.
[37] R. Honicky and E. L. Miller, "Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution," in *Proc. 18th Int. Parallel Distrib. Process. Symp.*, 2004, Art. no. 96.
[38] Z. Shen, P. P. Lee, J. Shu, and W. Guo, "Cross-rack-aware single failure recovery for clustered file systems," *IEEE Trans. Dependable Secure Comput.*, vol. 17, no. 2, pp. 248–261, Mar./Apr. 2020.
[39] C. Liu, K. Ouyang, X. Chu, H. Liu, and Y. Leung, "R-Memcached: A reliable in-memory cache for big key-value stores," *Tsinghua Sci. Technol.*, vol. 20, no. 6, pp. 560–573, 2015.
[40] R. Nishtala *et al.*, "Scaling Memcache at Facebook," in *Proc. 10th USENIX Symp. Netw. Syst. Des. Implementation*, 2013, pp. 385–398.
[41] M. Abebe, K. Daudjee, B. Glasbergen, and Y. Tian, "EC-Store: Bridging the gap between storage and latency in distributed erasure coded systems," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 255–266.
[42] R. Li, P. P. C. Lee, and Y. Hu, "Degraded-first scheduling for MapReduce in erasure-coded storage clusters," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2014, pp. 419–430.
[43] T. C. Blog, "v13.2.6 mimic released," 2019. [Online]. Available: https://ceph.io/releases/v13-2-6-mimic-released/
[44] H. I. Benchmark, "IOR," 2020. [Online]. Available: https://github.com/stevenlcj/ESetStore/tree/master/IOR


Chengjian Liu received the MS degree from the College of Computer Science & Software Engineering, Shenzhen University, Shenzhen, P.R. China, in 2013, and the PhD degree in computer science from the Hong Kong Baptist University, Hong Kong, in 2018. Currently, he is an assistant professor with the College of Big Data and Internet, Shenzhen Technology University. His research interests include distributed storage, blockchain, and general-purpose GPU computing.

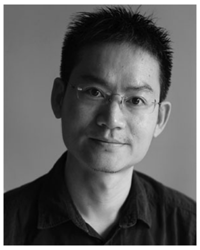
Qiang Wang received the BSc degree from the South China University of Technology, Guangzhou, China, in 2014. He is currently working toward the PhD degree in the Department of Computer Science, Hong Kong Baptist University, Hong Kong. His research interests include general-purpose GPU computing and power-efficient computing. He is a recipient of Hong Kong PhD Fellowship.



Xiaowen Chu (Senior Member, IEEE) received the BE degree in computer science from Tsinghua University, Beijing, P.R. China, in 1999, and the PhD degree in computer science from the Hong Kong University of Science and Technology, Hong Kong, in 2003. Currently, he is a professor with the Department of Computer Science, Hong Kong Baptist University. His research interests include distributed and parallel computing, deep learning systems, and wireless networks. He is currently serving as an associate editor of the *IEEE Access* and the *IEEE Internet of Things Journal*.



Yiu-Wing Leung received the BSc and PhD degrees from the Chinese University of Hong Kong, Hong Kong. He has been working with the Hong Kong Baptist University and currently he is professor of the Computer Science Department and programme director of two MSc programmes. His research interests include three major areas: 1) network design, analysis and optimization, 2) Internet and cloud computing, and 3) systems engineering and optimization. He has published more than 50 papers in these areas in various IEEE transactions and journals.



Hai Liu (Member, IEEE) received the BSc and MSc degrees in applied mathematics from the South China University of Technology, Guangzhou, China, and the PhD degree in computer science from the City University of Hong Kong, Hong Kong. He is currently an associate professor with the Department of Computing, Hang Seng University of Hong Kong. Before joining HSUHK, he held several academic posts with the University of Ottawa and Hong Kong Baptist University. His research interests includes wireless networking, cloud computing, and algorithm design and analysis. His h-index is 22 according to Google Scholar.

The Design of Fast Content-Defined Chunking for Data Deduplication Based Storage Systems

Wen Xia, *Member, IEEE*, Xiangyu Zou, *Student Member, IEEE*, Hong Jiang, *Fellow, IEEE*, Yukun Zhou, Chuanyi Liu, Dan Feng, *Member, IEEE*, Yu Hua, *Senior Member, IEEE*, Yuchong Hu, and Yucheng Zhang

Abstract—Content-Defined Chunking (CDC) has been playing a key role in data deduplication systems recently due to its high redundancy detection ability. However, existing CDC-based approaches introduce heavy CPU overhead because they declare the chunk cut-points by computing and judging the rolling hashes of the data stream byte by byte. In this article, we propose FastCDC, a Fast and efficient Content-Defined Chunking approach, for data deduplication-based storage systems. The key idea behind FastCDC is the combined use of five key techniques, namely, gear based fast rolling hash, simplifying and enhancing the Gear hash judgment, skipping sub-minimum chunk cut-points, normalizing the chunk-size distribution in a small specified region to address the problem of the decreased deduplication ratio stemming from the cut-point skipping, and last but not least, rolling two bytes each time to further speed up CDC. Our evaluation results show that, by using a combination of the five techniques, FastCDC is 3-12X faster than the state-of-the-art CDC approaches, while achieving nearly the same and even higher deduplication ratio as the classic Rabin-based CDC. In addition, our study on the deduplication throughput of FastCDC-based Destor (an open source deduplication project) indicates that FastCDC helps achieve 1.2-3.0X higher throughput than Destor based on state-of-the-art chunkers.

Index Terms—Data deduplication, content-defined chunking, storage system, performance evaluation

1 Introduction

Data deduplication, an efficient approach to data reduction, has gained increasing attention and popularity in large-scale storage systems due to the explosive growth of digital data. It eliminates redundant data at the file- or chunk-level and identifies duplicate contents by their cryptographically secure hash signatures (e.g., SHA1 fingerprint). According to deduplication studies conducted by Microsoft [1], [2] and EMC [3], [4], about 50 and 85 percent of the data in their production primary and secondary storage systems, respectively, are redundant and could be removed by the deduplication technology.

In general, chunk-level deduplication is more popular than file-level deduplication because it identifies and removes redundancy at a finer granularity. For chunk-level deduplication, the simplest chunking approach is to cut the file or data stream into equal, fixed-size chunks, referred to as Fixed-Size Chunking (FSC) [5]. Content-Defined Chunking (CDC) based approaches are proposed to address the *boundary-shift* problem faced by the FSC approach [6]. Specifically, CDC declares chunk boundaries based on the byte contents of the data stream instead of on the byte offset, as in FSC, and thus helps detect more redundancy for deduplication. According to some recent studies [1], [2], [7], [8], CDC-based deduplication approaches are able to detect about 10-20 percent more redundancy than the FSC approach.

Currently, the most popular CDC approaches determine chunk boundaries based on the Rabin fingerprints of the content, which we refer to as Rabin-based CDC [6], [9], [10]. Rabin-based CDC is highly effective in duplicate detection but time-consuming, because it computes and judges (against a condition value) Rabin fingerprints of the data stream byte by byte [11]. In order to speed up the CDC process, other hash algorithms have been proposed to replace the Rabin algorithm for CDC, such as SampeByte [12], Gear [13], etc. Meanwhile, the abundance of computation resources afforded by multi and manycore processors [14], [15] or GPU processors [16], [17], [18] has been leveraged for CDC acceleration.

Generally, CDC consists of two distinctive and sequential stages: *(1) hashing* in which fingerprints of the data contents are generated and *(2) hash judgment* in which fingerprints are compared against a given value to identify and declare chunk cut-points. Our previous study of delta compression, Ddelta [13], suggests that the Gear hash (i.e., $fp = (fp << 1) + G(b)$, see Section 3.2) is more efficient as a rolling hash for

- W. Xia is with the Harbin Institute of Technology, Shenzhen 518055, China, Cyberspace Security Research Center, Peng Cheng Laboratory, Shenzhen 518055, China, and also with the Wuhan National Laboratory for Optoelectronics, Wuhan 430074, China. E-mail: xiawen@hit.edu.cn.
- X. Zou and C. Liu are with the Harbin Institute of Technology, Shenzhen 518055, China, and also with the Cyberspace Security Research Center, Peng Cheng Laboratory, Shenzhen 518055, China. E-mail: xdnzxy@gmail.com, liuchuanyi@hit.edu.cn.
- H. Jiang is with the Department of Computer Science and Engineering, University of Texas at Arlington, TX 76019. E-mail: hong.jiang@uta.edu.
- Y. Zhou, D. Feng, Y. Hua, Y. Hu, and Y. Zhang are with the Wuhan National Laboratory for Optoelectronics, School of Computer Sci.&Tech., Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {ykzhou, dfeng, csyhua, yuchonghu, cszyc}@hust.edu.cn.

Manuscript received 29 July 2019; revised 21 Feb. 2020; accepted 25 Mar. 2020. Date of publication 2 Apr. 2020; date of current version 17 Apr. 2020. (Corresponding author: Chuanyi Liu.) Recommended for acceptance by T. Kosar. Digital Object Identifier no. 10.1109/TPDS.2020.2984632

CDC. To the best of our knowledge, Gear appears to be one of the fastest rolling hash algorithms for CDC at present since it use much less calculation operations than others. However, our empirical and analytical studies on Gear-based CDC obtain three important observations:

- Observation ①: the Gear-based CDC has the potential problem of low *deduplication ratio* (i.e., the percentage of redundant data reduced), about 10-50 percent lower on some datasets (detailed in Section 4.2). This is because, in the *hash judgment* stage of Gear-based CDC, the sliding window size is very small, only 13 bytes in its current implementation [13].
- Observation ②: the *hash judgment* stage becomes the new performance bottleneck in Gear-based CDC. This is because the accelerated *hashing* stage by Gear, has shifted the bottleneck to the *hash judgment* stage.
- Observation ③: Enlarging the predefined minimum chunk size (used in CDC to avoid the very small-sized chunks [6]) can further speed up the chunking process (called *cut-point skipping* in this paper) but at the cost of *decreasing the deduplication ratio* in Gear-based CDC. This is because many chunks with skipped cut-points are not divided truly according to the data contents (i.e., content-defined). Our large scale study (detailed in Section 4.3) suggests that skipping this predefined min chunk size usually increases the chunking speed by the ratio of $\frac{Predefined\ min\ chunk\ size}{Expected\ avg.\ chunk\ size}$ but decreases the deduplication ratio (about 15 percent decline in the worst case).

Therefore, motivated by the above three observations, we proposed FastCDC, a Fast and efficient CDC approach that addresses the problems of low deduplication efficiency and expensive hash judgement faced by Gear-based CDC. To address the problems observed in the 1st and 2nd observations, we use an approach of *enhancing and simplifying the hash judgment* to further reduce the CPU operations during CDC for data deduplication. Specifically, FastCDC pads several zero bits into the mask value in its hash-judging statement to enlarge the sliding window size to the size of 48 Bytes used by Rabin-based CDC, which makes it able to achieve nearly the same deduplication ratio as the Rabin-based CDC; Meanwhile, by further simplifying and optimizing the hash-judging statement, FastCDC decreases the CPU overhead for the hash judgment stage in CDC.

For the 3rd observation and to further speed up chunking, FastCDC employs a novel normalized Content-Defined Chunking scheme, called *normalized chunking*, that normalizes the chunk-size distribution to a specified region that *is guaranteed to be larger than the minimum chunk size* to effectively address the problem facing the cut-point skipping approach. Specifically, FastCDC selectively changes the number of mask bits '1' in the hash-judging statement of CDC, and thus it normalizes the chunk-size distribution to a small specified region (e.g., 8KB$\sim$16KB), i.e., the vast majority of the generated chunks fall into this size range, and thus minimizes the number of chunks of either too small or large in size. The benefits are twofold. ①, it increases the deduplication ratio by reducing the number of large-sized chunks. ②, it reduces the number of small-sized chunks, which makes it possible to combine with the cut-point skipping technique above to maximize the CDC speed without sacrificing the deduplication ratio.

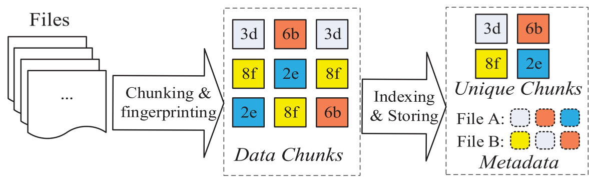


Fig. 1. General workflow of chunk-level data deduplication.

In addition, we propose a technique called "rolling two bytes each time" for FastCDC, which further reduces the calculation operations in the *hashing* stage by rolling two bytes each time to calculate the chunking fingerprints in the *hashing* stage, and then judging the even and odd bytes respectively in the *hash judgement* stage. This further accelerates the chunking process while achieving exactly the same chunking results.

Our evaluation results based on seven large-scale datasets, suggest that FastCDC is about 3-12$\times$ faster than the state of art, while ensuring a comparably high deduplication ratio. In addition, we have incorporated FastCDC in Destor [19], an open source data deduplication system, and evaluation shows that Destor using FastCDC helps achieve about 1.2-3.0X higher system throughout than using other CDC approaches. Meanwhile, due to its simplicity and effectiveness, FastCDC has been adopted as the default chunker by several known open source Github projects to speed up the detection of duplicate contents, such as Rdedup [20], Content Blockchain [21], etc. The released Rdedup version 2.0.0 states: "rdedupe store performance has been greatly improved by implementing many new techniques" and "our default CDC algorithm is now FastCDC".

2 Background

Recently, chunk-level data deduplication becomes one of the most popular data reduction method in storage systems for improving storage and network efficiency. As shown in Fig. 1, it splits a file into several contiguous chunks and removes duplicates by computing and indexing hash digests (or called fingerprints, such as SHA-1) of chunks [5], [6], [22], [23]. The fingerprints matching means that their corresponding chunks are duplicate, which thus simplifies the global duplicates detection in storage systems. In the past ten years, data deduplication technique has been demonstrated its space efficiency functionality in the large-scale production systems of Microsoft [1], [2] and EMC [3], [4].

Chunking is the first critical step in the operational path of data deduplication, in which a file or data stream is divided into small chunks so that each can be duplicate-identified. Fixed-Size Chunking (FSC) [5] is simple and fast but may face the problem of low deduplication ratio that stems from the *boundary-shift* problem [6], [24]. For example, if one or several bytes are inserted at the beginning of a file, all current chunk cut-points (i.e., boundaries) declared by FSC will be shifted and no duplicate chunks will be detected.

Content-Defined Chunking (CDC) is proposed to solve the *boundary-shift* problem. CDC uses a sliding-window technique on the content of files and computes a hash value (e.g., Rabin fingerprint [6], [9]) of the window. A chunk cut-point

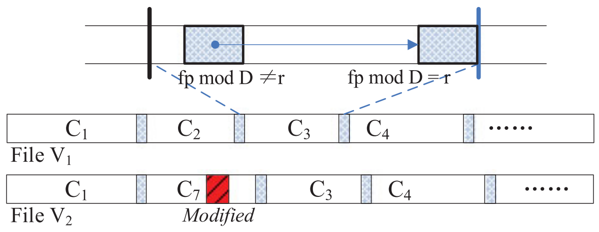


Fig. 2. The sliding window technique for the CDC algorithm. The hash value of the sliding window, *fp*, is computed via the Rabin algorithm (this is the *hashing stage* of CDC). If the lowest $log_2 D$ bits of the hash value matches a threshold value r, i.e., *fp* mod $D = r$, this offset (i.e., the current position) is marked as a chunk cut-point (this is the *hash-judging stage* of CDC).

is declared if the hash value satisfies some pre-defined condition. As shown in Fig. 2, to chunk a file V_2 that is modified from the file V_1, the CDC algorithm can still identify the correct boundary of chunks C_1, C_3, and C_4, whose contents have not been modified. As a result, CDC outperforms FSC in terms of deduplication ratio and has been widely used in backup [3], [22] and primary [1], [2] storage systems.

Although the widely used Rabin-based CDC helps obtain a high deduplication ratio, it incurs heavy CPU overhead [14], [16], [18], [25]. Specifically, in Rabin-based CDC, Rabin hash for a sliding window containing the byte sequence B_1, $B_2, \ldots, B_\alpha$ is defined as a polynomial

$$Rabin(B_1, B_2, \ldots, B_\alpha) = A(p) = \left\{\sum_{x=1}^{\alpha} B_x p^{\alpha-x}\right\} mod\ D, \quad (1)$$

where D is the average chunk size, α is the number of bytes in the sliding window, and p is a number representing an irreducible polynomial [9]. Rabin hash is a *rolling hash* algorithm since it is able to compute the hash in an iterative fashion, i.e., the current hash can be incrementally computed from the previous value as

$$\begin{aligned} &Rabin(B_2, B_3, \ldots, B_{\alpha+1}) \\ &= \{[Rabin(B_1, B_2, \ldots, B_\alpha) - B_1 p^{\alpha-1}]p + B_{\alpha+1}\} modS. \end{aligned} \quad (2)$$

However, Rabin-based CDC is time-consuming because it computes and judges the hashes of the data stream byte by byte, which renders the chunking process a performance bottleneck in deduplication systems. There are many approaches that accelerate the CDC process for deduplication systems and they can be broadly classified as either algorithmic oriented or hardware oriented. We summarize below some of these approaches that represent the state of the art.

Algorithmic-Oriented CDC Optimizations. Since the frequent computations of Rabin fingerprints for CDC are time-consuming, many alternatives to Rabin have been proposed to accelerate the CDC process [12], [13], [24]. SampleByte [12] is designed for providing fast chunking for fine-grained network redundancy elimination, usually eliminating duplicate chunks as small as 32-64 bytes. It uses one byte to declare a fingerprint for chunking, in contrast to Rabin that uses a sliding window, and skips $\frac{1}{2}$ of the expected chunk size before chunking to avoid generating extremely small-sized strings or chunks (they called "avoid oversampling"). Gear [13] uses fewer operations to generate rolling hashes by means of a small random integer table to map the values of the byte contents, so as to achieve higher chunking throughput. AE [24] is a non-rolling-hash-based chunking algorithm that employs an asymmetric sliding window to identify extremums of data stream as cut-points, which reduces the computational overhead for CDC. Yu *et al.* [26] adjust the function for selecting chunk boundaries such that if weak conditions are not met, the sliding window can jump forward, avoiding unnecessary calculation steps. RapidCDC [27] leverages the data locality to record the chunking positions to reduce the CDC computations for the duplicate chunks to appear next time.

Hardware-Oriented CDC Optimizations. StoreGPU [16], [17] and Shredder [18] make full use of GPU's computational power to accelerate popular compute-intensive primitives (i.e., chunking and fingerprinting) in data deduplication. P-Dedupe [14] pipelines deduplication tasks and then further parallelizes the sub-tasks of chunking and fingerprinting with multiple threads and thus achieves higher throughput. SS-CDC [28] proposes a two-stage prallel content-defined chunking approach without compromising deduplication ratio.

It is noteworthy that there are other chunking approaches trying to achieve a higher deduplication ratio but introduce more computation overhead on top of the conventional CDC approach. TTTD [29] and Regression chunking [2] introduces one or more additional thresholds for chunking judgment, which leads to a higher probability of finding chunk boundaries and decreases the chunk size variance. MAXP [30], [31], [32] treats the extreme values in a fixed-size region as cut-points, which also results in smaller chunk size variance. In addition, Bimodal chunking [33], Subchunk [34], and FBC [35] re-chunk the non-duplicate chunks into smaller ones to detect more redundancy.

For completeness and self-containment we briefly discuss *other relevant deduplication issues* here. A typical data deduplication system follows the workflow of chunking, fingerprinting, indexing, and storage management [19], [22], [36], [37]. The fingerprinting process computes the cryptographically secure hash signatures (e.g., SHA1) of data chunks, which is also a compute-intensive task but can be accelerated by certain pipelining or parallelizing techniques [14], [38], [39], [40]. Indexing refers the process of identifying the identical fingerprints for checking duplicate chunks in large-scale storage systems, which has been well explored in many previous studies [19], [22], [41], [42]. Storage management refers to the storage and possible post-deduplication processing of the non-duplicate chunks and their metadata, including such processes as related to further compression [13], defragmentation [43], reliability [44], security [45], etc. In this paper, we focus on designing a very fast and efficient chunking approach for data deduplication since the CPU-intensive CDC task has been widely recognized as a major performance bottleneck of the CDC-based deduplication system [17], [18], [27], [28].

3 FastCDC Design and Implementation

3.1 FastCDC Overview

FastCDC aims to provide high performance CDC. And there are three metrics for evaluating CDC performance, namely, deduplication ratio, chunking speed, and the average generated chunk size. Note that the average generated

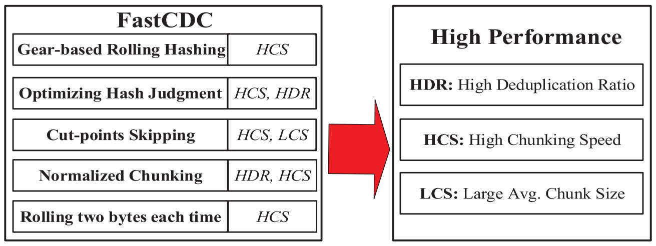

FastCDC
Gear-based Rolling Hashing | HCS
Optimizing Hash Judgment | HCS, HDR
Cut-points Skipping | HCS, LCS
Normalized Chunking | HDR, HCS
Rolling two bytes each time | HCS
High Performance
HDR: High Deduplication Ratio
HCS: High Chunking Speed
LCS: Large Avg. Chunk Size

Fig. 3. The five key techniques used in FastCDC and their corresponding benefits for high performance CDC.

chunk size is also an important CDC performance metric since it reflects the metadata overhead for deduplication indexing, i.e., the larger the generated chunk size is, the fewer the number of chunks and thus the less metadata will be processed by data deduplication.

Generally, it is difficult, if not impossible, to improve these three performance metrics simultaneously because they can be conflicting goals. For example, a smaller average generated chunk size leads to a higher deduplication ratio, but at the cost of lower chunking speed and high metadata overheads. Thus, *FastCDC is designed to strike a sensible trade-off among these three metrics so as to strive for high performance CDC, by using a combination of the five techniques with their complementary features* as shown in Fig. 3.

- Gear-based rolling hashing: due to its hashing simplicity and rolling effectiveness, Gear hash is shown to be one of the fastest rolling hash algorithms for CDC, and we introduce it and discuss its chunking efficiency in detail in Section 3.2.
- Optimizing hash judgment: using a zero-padding scheme and a simplified hash-judging statement to speed up CDC without compromising the deduplication ratio, as detailed in Section 3.3.
- Sub-minimum chunk cut-point skipping: enlarging the predefined minimum chunk size and skipping cut-points for chunks smaller than that to provide a higher chunking speed and a larger average generated chunk size, as detailed in Section 3.4.
- Normalized chunking: selectively changing the number of mask '1' bits for the hash judgment to approximately normalize the chunk-size distribution to a small specified region that is just larger than the predefined minimum chunk size, ensuring both a higher deduplication ratio and higher chunking speed, as detailed in Section 3.5.
- Rolling two bytes each time: furhter speeding up chunking without affecting the chunking results by reducing the calculation operations in the hashing stage by rolling two bytes each time to calculate the chunking fingerprints in the *hashing* stage, and then judging the even and odd bytes respectively in the *hash judgement* stage, as detailed in Section 3.7.

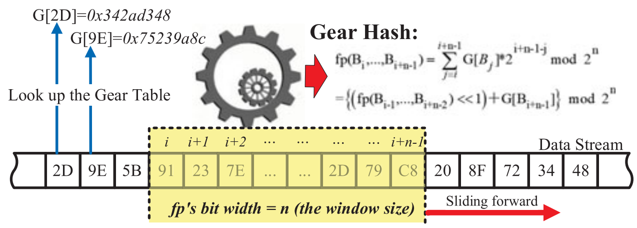

G[2D]=0x342ad348
G[9E]=0x75239a8c
Look up the Gear Table
Gear Hash:
$fp(B_i,...,B_{i+n-1}) = \sum_{j=i}^{i+n-1} G[B_j]*2^{i+n-1-j} \bmod 2^n$
$=\{(fp(B_{i-1},...,B_{i+n-2})<<1)+G[B_{i+n-1}]\} \bmod 2^n$
i i+1 i+2 ... ... i+n-1
Data Stream
2D 9E 5B 91 23 7E 2D 79 C8 20 8F 72 34 48
fp's bit width = n (the window size)
Sliding forward

Fig. 4. A schematic diagram of the Gear hash.

TABLE 1
The Hashing Stage of the Rabin- and Gear-Based CDC

Name	Pseudocode	Speed
Rabin	$fp = ((fp\hat{}U(a)) << 8) \mid b\hat{}T[fp >> N]$	Slow
Gear	$fp = (fp << 1) + G(b)$	Fast

Here 'a' and 'b' denote contents of the first and last byte of the sliding window respectively, 'N' is the length of the content-defined sliding window, and 'U', 'T', 'G' denote the predefined arrays [6], [11], [13]. 'fp' represents the fingerprint of the sliding window.

In general, the key idea behind FastCDC is the combined use of the above five key techniques for CDC acceleration, especially employing normalized chunking to address the problem of decreased deduplication ratio facing the cut-point skipping, and thus achieve high performance CDC on the three key metrics.

3.2 Gear-Based Rolling Hashing

In this subsection, we elaborate on and analyze the Gear-based rolling hash, and then introduce the new challenges and opportunities after we introduce Gear-based CDC. Gear-based rolling hash is first employed by Ddelta [13] for delta compression, which helps provide a higher delta encoding speed and is suggested to be a good rolling hash candidate for CDC.

A good hash function must have a uniform distribution of hash values regardless of the hashed content. As shown in Fig. 4, Gear-based CDC achieves this in two key ways: (1) It employs an array of 256 random 64-bit integers to map the values of the byte contents in the sliding window (i.e., the calculated bytes, whose size is the bit-width of the *fp*); and (2) The addition ("+") operation adds the new byte in the sliding window into Gear hashes while the left-shift ("$<<$") operation helps strip away the last byte of the last sliding window (e.g., B_{i-1} in Fig. 4). This is because, after the "$<<$" and modulo operations, the last byte B_{i-1} will be calculated into the *fp* as the $(G[B_{i-1}] << n)\ mod\ 2^n$, which will be equal to zero. As a result, Gear generates uniformly distributed hash values by using only three operations (i.e., "+", "$<<$", and an array lookup), enabling it to move quickly through the data content for the purpose of CDC. Table 1 shows a comparison among the two rolling hash algorithms: Rabin and Gear, which suggests Gear uses far fewer calculation operations than Rabin, thus being a good rolling hash candidate for CDC.

To better illustrate Gear-based CDC, Algorithm 3.2 provides the detailed chunking pseudo code that uses the *Gear* table for calculating the rolling fingerprints and the hash judging statement similar to the classical Rabin-based CDC. In Fig. 5, we compare the chunk-size distributions each generated by Rabin- and Gear-based CDC on the random-number workload, and against the mathematical analysis based on Equation (3) (see Section 3.4), which indicates that the three are almost identical (for more chunk-size distribution results, see Fig. 12 in Section 4.2). And the previous study Ddelta [13] also suggests Gear is considered to be a good rolling hash candidate for CDC both on the hashing efficiency and on the

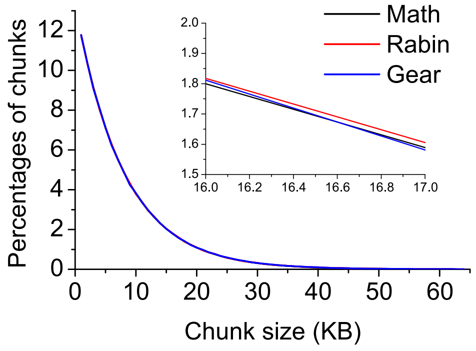


Fig. 5. Chunk-size distributions of Rabin- and Gear-based CDC approaches with average chunk size of 8KB (without max/min chunk size requirement). "Rabin" and "Gear" denote our experimental results after CDC and "Math" denotes theoretical analysis, where they are shown to be nearly identical.

chunking efficiency. However, according to our experimental analysis, there are still challenges and opportunities for Gear-based CDC, such as low deduplication ratio, expensive hash judgment, further acceleration by skipping, etc. We elaborate on these issues as follows.

Algorithm 1. GearCDC8KB

```
Input: data buffer, $src$; buffer length, $n$
Output: chunking breakpoint $i$
$MinSize \leftarrow 2KB$;    $MaxSize \leftarrow 64KB$;
$i \leftarrow MinSize$;    $fp \leftarrow 0$;
if $n \leq MinSize$ then
  return $n$
while $i < n$ do
  $fp = (fp << 1) + Gear[\ src[i]\ ]$;
  if $(fp\ \&\ 0x1fff\ ==\ 0x78)\ ||\ i\ >=\ MaxSize$ then
    return $i$;
return $i$;
```

Low Deduplication Ratio Due to the Limited Sliding Window Size. The Gear-based CDC has the potential problem of low *deduplication ratio*, about 10-50 percent lower on some datasets (see the evaluation results in Section 4.2). This is becase, the traditional hash judgment for the Rabin-based CDC, as shown in Fig. 2 (i.e., "*fp mod D==r*"), is also used by the Gear-based CDC [13] as shown in Algorithm 3.2. But this results in a smaller sized sliding window used by Gear-based CDC since it uses Gear hash for chunking. For example, as shown in Fig. 6, the sliding window size of the Gear-based CDC will be equal to the number of the '1' bits used by the mask value. Therefore, when using a mask value of 0x1fff (i.e., $2^{13}-1$, there are thirteen '1' bits) for the expected chunk size of 8 KB, the sliding window for the Gear-based CDC would be 13 bytes while that of the Rabin-based CDC would be 48 bytes [6]. The smaller sliding window size of the Gear-based CDC can lead to more chunking position collisions (i.e., randomly marking the different positions as the chunk cut-points), resulting in the decrease in deduplication ratio.

The Expensive Hash Judgment. In Gear-based CDC, the accelerated *hashing* stage by the fast Gear, has shifted the bottleneck to the *hash judgment* stage that requires more operations as shown in Algorithm 3.2. Our implementation and in-depth analysis of Gear-based CDC on several datasets

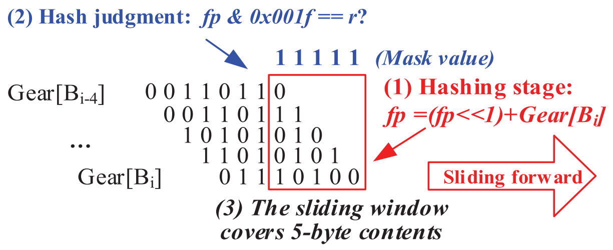


Fig. 6. An example of the sliding window technique used in the Gear-based CDC. Here CDC consists of two stages: hashing and hash judgment. The size of the sliding window used for hash judgment is only 5 bytes because of the computation principles of the Gear hash.

(detailed in Section 4) suggest that *its hash-judging stage accounts for more than 60 percent of its CPU overhead during CDC after the fast Gear hash is introduced*. Thus, there is a lot of room for the optimization of the hash judging stage to further accelerate the CDC process as discussed later in Section 3.3.

Further Speed up Chunking by Skipping. Another observation is that the minimum chunk size used for avoiding extremely small-sized chunks, can be also employed to speed up CDC by the cut-point skipping, i.e., eliminating the chunking computation in the skipped region. But this minimum chunk size for cut-point skipping approach decreases the deduplication ratio (as demonstrated in the evaluation results in Fig. 12c in Section 4.3) since many chunks are not divided truly according to the data contents, i.e., not really content-defined.

The last observation from the minimum chunk size skipping motivates us to consider a new CDC approach that (1) keeps all the chunk cut-points that generate chunks larger than a predefined minimum chunk size and (2) enables the chunk-size distribution to be normalized to a relatively small specified region, an approach we refer to as *normalized chunking* in this paper, as described in Section 3.5.

3.3 Optimizing Hash Judgment

In this subsection, we optimize the hash judgment stage on top of the Gear-based CDC, which helps further accelerate the chunking process and increase the deduplication ratio to reach that of the Rabin-based CDC. More specifically, FastCDC incorporates two main optimizations as elaborated below.

Enlarging the Sliding Window Size by Zero Padding. As discussed in Section 3.2, the Gear-based CDC employs the same conventional hash judgment used in the Rabin-based CDC, where a certain number of the lowest bits of the fingerprint are used to declare the chunk cut-point, leading to a shortened sliding window for the Gear-based CDC (see Fig. 6) because of the unique feature of the Gear hash. To address this problem, FastCDC enlarges the sliding window size by padding a number of zero bits into the mask value. As illustrated by the example of Fig. 7, FastCDC pads five zero bits into the mask value and changes the hash judgment statement to "*fp & mask == r*". If the masked bits of *fp* match a threshold value r, the current position will be declared as a chunk cut-point. Since Gear hash uses one left-shift and one addition operation to compute the rolling hash, this zero-padding scheme enables 10 bytes (i.e., $B_i, \ldots, B_{i+9}$), instead of the original five bytes, to be involved in the final hash judgment by the five masked one bits (as the red box shown in Fig. 7) and thus makes the

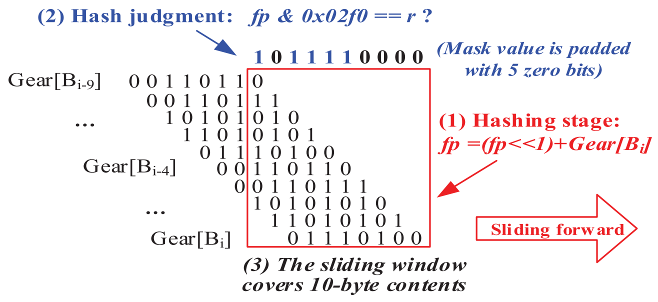

(2) Hash judgment: fp & 0x02f0 == r ?
1 0 1 1 1 1 0 0 0 0
(Mask value is padded with 5 zero bits)
Gear[Bi-9]
...
Gear[Bi-4]
...
Gear[Bi]
(1) Hashing stage: fp =(fp<<1)+Gear[Bi]
Sliding forward
(3) The sliding window covers 10-byte contents

Fig. 7. An example of the sliding window technique proposed for FastCDC. By padding y zero bits into the mask value for hash judgment, the size of the sliding window used in FastCDC is enlarged to about *5+y* bytes, where y*=5* in this example.

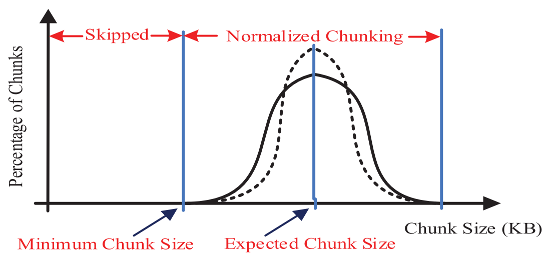


Fig. 8. A conceptual diagram of the normalized chunking combined with the subminimum chunk cut-point skipping. The dotted line shows a higher level of normalized chunking.

sliding window size equal or similar to that of the Rabin-based CDC [6], minimizing the probability of the chunking position collision. As a result, FastCDC is able to achieve a deduplication ratio as high as that by the Rabin-based CDC.

Simplifying the Hash Judgment to Accelerate CDC. The conventional hash judgment process, as used in the Rabin-based CDC, is expressed in the programming statement of "*fp mod D==r*" [6], [13]. For example, the Rabin-based CDC usually defines D and r as *0x02000* and *0x78*, according to the known open source project LBFS [6], to obtain the expected average chunk size of 8 KB. In FastCDC, when combined with the zero-padding scheme introduced above and shown in Fig. 7, the hash judgment statement can be optimized to "*fp & Mask*==0", which is equivalent to "!*fp & Mask*". Therefore, FastCDC's hash judgment statement reduces the register space for storing the threshold value r and avoids the unnecessary comparison operation that compares "*fp & Mask*" and r, thus further speeds up the CDC process as verified in Section 4.2.

3.4 Cut-Point Skipping

Most of CDC-based deduplication systems impose a limit of the maximum and minimum chunk sizes, to avoid the pathological cases of generating many extremely large- or small-sized chunks by CDC [1], [6], [33], [34], [37], [46]. A common configuration of the average, minimum, and maximum parameters follows that used by LBFS [6], i.e., 8, 2, and 64 KB. Our previous study [13] and experimental observations (see Fig. 11 in Section 4.2, using curve fitting) suggest that the cumulative distribution of chunk size X in Rabin-based CDC approaches with an expected chunk size of *8 KB* (without the maximum and minimum chunk size requirements) generally follows an exponential distribution as follows:

$$P(X \le x) = F(x) = (1 - e^{-\frac{x}{8192}}),\ x \ge 0. \qquad (3)$$

Note that this theoretical exponential distribution in Equation (3) is based on the assumption that the data content and Rabin hashes of contents (recall Equation (1) and Fig. 2 for CDC) follow a uniform distribution. Equation (3) suggests that the value of the expected chunk size will be 8 KB according to exponential distribution.

According to Equation (3), the chunks smaller than 2 KB and larger than 64 KB would account for about 22.12 and 0.03 percent of the total number of chunks respectively. This means that imposing the maximum chunk size requirement only slightly hurts the deduplication ratio but skipping cut-points before chunking to avoid generating chunks smaller than the prescribed minimum chunk size, or called *sub-minimum chunk cut-point skipping* , will impact the deduplication ratio significantly as evidenced in Fig. 12c. This is because a significant portion of the chunks are not divided truly according to the data contents, but forced by this cut-point skipping.

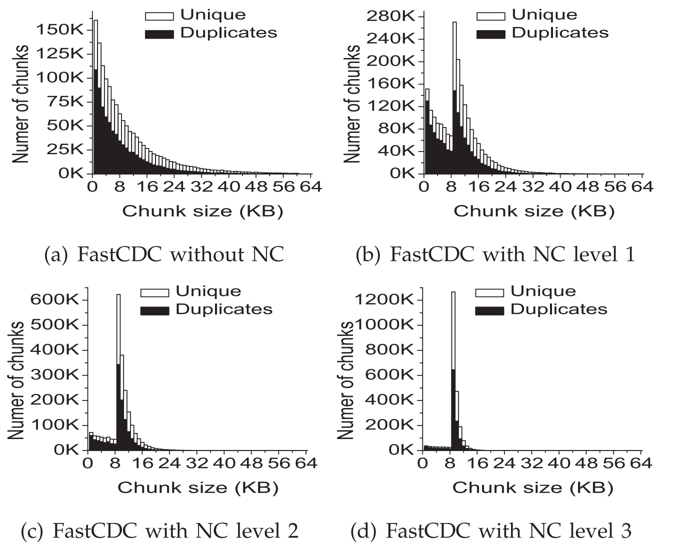
Given FastCDC's goal of maximizing the chunking speed, enlarging the minimum chunk size and skipping sub-minimum chunk cut-point will help FastCDC achieve a higher CDC speed by avoiding the operations for the hash calculation and judgment in the skipped region. This gain in speed, however, comes at the cost of reduced deduplication ratio. To address this problem, we will develop a normalized chunking approach, to be introduced in the next subsection.

It is worth noting that this cut-point skipping approach, by avoiding generating chunks smaller than the minimum chunk size, also helps increase the average generated chunk size. In fact, the average generated chunk size exceeds the expected chunk size by an amount equal to the minimum chunk size. This is because the F(x) in Equation (3) is changed to $(1 - e^{-\frac{x-MinSize}{8192}})$ after cut-point skipping, thus the value of the expected chunk size becomes 8 KB + minimum chunk size, which will be verified in Section 4.3. The speedup achieved by skipping the sub-minimum chunk cut-point can be estimated by $1+\frac{\text{the minimum chunk size}}{\text{the expected chunk size}}$. The increased chunking speed comes from the eliminated computation on the skipped region, which will also be evaluated and verified in Section 4.3.

3.5 Normalized Chunking

In this subsection, we propose a novel chunking approach, called normalized chunking, to solve the problem of decreased deduplication ratio facing the cut-point skipping approach. As shown in Fig. 8, normalized chunking generates chunks whose sizes are normalized to a specified region centered at the expected chunk size. After normalized chunking, there are almost no chunks of size smaller than the minimum chunk size, which means that normalized chunking enables skipping cut-points for subminimum chunks to reduce the unnecessary chunking computation and thus speed up CDC.

In our implementation of normalized chunking, we selectively change the number of effective mask bits (i.e., the number of '1' bits) for the hash-judging statement. For the traditional CDC approach with expected chunk size of 8 KB (i.e., 2^{13}), 13 effective mask bits are used for hash judgment



(a) FastCDC without NC (b) FastCDC with NC level 1

(c) FastCDC with NC level 2 (d) FastCDC with NC level 3

Fig. 9. Chunk-size distribution of FastCDC with normalized chunking (NC) at different normalization levels.

TABLE 2
Workload Characteristics of the Seven Datasets Used in the Performance Evaluation

Name	Size	Workload descriptions
TAR	56 GB	215 tarred files from several open source projects such as GCC, GDB, Emacs, etc. [47]
LNX	178 GB	390 versions of Linux source code files (untarred). There are totally 16, 381, 277 files [48].
WEB	237 GB	102 days' snapshots of the website: *news.sina.com*, which are collected by crawling software *wget* with a maximum retrieval depth of 3.
VMA	138 GB	90 virtual machine images of different OS release versions, including CentOS, Fedora, etc. [49]
VMB	1.9 TB	125 backups of an Ubuntu 12.04 virtual machine image in use by a research group.
RDB	1.1 TB	198 backups of the Redis key-value store database snapshots, i.e., dump.rdb files.
SYN	2.1 TB	300 synthetic backups. The backup is simulated by the file create/delete/modify operations [50].

(e.g., fp & $0x1fff{==}r$). For normalized chunking, more than 13 effective mask bits are used for hash judgment (e.g., fp & $0x7fff{==}r$) when the current chunking position is smaller than 8 KB, which makes it harder to generate chunks of size smaller than 8 KB. On the other hand, fewer than 13 effective mask bits are used for hash judgment (e.g., fp & $0x0fff{==}r$) when the current chunking position is larger than 8 KB, which makes it easier to generate chunks of size larger than 8 KB. Therefore, by changing the number of '1' bits in FastCDC, the chunk-size distribution will be approximately normalized to a specified region always larger than the minimum chunk size, instead of following the exponential distribution (see Fig. 5).

Generally, there are three benefits or features of normalized chunking (NC):

- NC reduces the number of small-sized chunks, which makes it possible to combine it with the cut-point skipping approach to *achieve high chunking speed without sacrificing the deduplication ratio* as suggested in Fig. 8.
- NC further improves the deduplication ratio by reducing the number of large-sized chunks, which compensates for the reduced deduplication ratio caused by reducing the number of small-sized chunks in FastCDC.
- The implementation of FastCDC does not add additional computing and comparing operations. It simply separates the hash judgment into two parts, before and after the expected chunk size.

Fig. 9 shows the chunk-size distribution after normalized chunking in comparison with FastCDC without NC on the TAR dataset (whose workload characteristics are detailed in Table 2 in Section 4.1). The normalization levels 1, 2, 3 indicate that the normalized chunking uses the mask bits of (14, 12), (15, 11), (16, 10), respectively, where the first and the second integers in the parentheses indicate the numbers of effective mask bits used in the hash judgment before and after the expected chunk size (or *normalized chunk size*) of 8 KB. Fig. 9 also suggests that the chunk-size distribution is a reasonably close approximation of the normal distribution centered on 8 KB at the normalization level of 2 or 3.

As shown in Fig. 9, there are only a very small number of chunks smaller than 2 or 4 KB after normalized chunking while FastCDC without NC has a large number of chunks smaller than 2 or 4 KB (consistent with the discussion in Section 3.4). Thus, when combining NC with the cut-point skipping to speed up the CDC process, only a very small portion of chunk cut-points will be skipped in FastCDC, leading to nearly the same deduplication ratio as the conventional CDC approaches without the minimum chunk size requirement. In addition, normalized chunking allows us to enlarge the minimum chunk size to maximize the chunking speed without sacrificing the deduplication ratio.

It is worth noting that the chunk-size distribution shown in Fig. 9 is not truly normal distribution but an approximation of it. Actually, it follows an improved exponential distribution calculating from Equation (3) as follows (taking the NC 2 as an example and using the average chunk size of 8 KB)

$$P(X \le x) = F(x) = \begin{cases} 1 - e^{-\frac{x}{8192*4}}, & 0 \le x \le 8192 \\ 1 - e^{-\frac{x}{8192/4}}, & x > 8192 \end{cases}. \tag{4}$$

Therefore, Figs. 9c and 9d shows a closer approximation of normal distribution of chunk size achieved by using the normalization levels 2 and 3. Interestingly, the highest normalization level of NC would be equivalent to Fixed-Size Chunking (FSC), i.e., all the chunk sizes are normalized to be equal to the expected chunk size. Since FSC has a very low deduplication ratio but extremely high chunking speed, it means that there will be a "sweet spot" among the normalization level, deduplication ratio, and chunking speed, which will be studied and evaluated in Section 4.

3.6 Putting It All Together

To put things together and in perspective. Algorithm 3.6 describes FastCDC combining the three key techniques: optimizing hash judgment, cut-point skipping, and normalized chunking (with the expected chunk size of 8 KB). The data structure "Gear" is a predefined array of 256 random 64-bit integers with one-to-one mapping to the values of byte contents for chunking [13].

Algorithm 2. FastCDC8KB (with NC)

```
Input: data buffer, $src$; buffer length, $n$
Output: chunking breakpoint $i$
$MaskS \leftarrow 0x0000d9f003530000LL$;        // 15 '1' bits;
$MaskA \leftarrow 0x0000d93003530000LL$;        // 13 '1' bits;
$MaskL \leftarrow 0x0000d90003530000LL$;        // 11 '1' bits;
$MinSize \leftarrow 2\ KB$;   $MaxSize \leftarrow 64\ KB$;
$fp \leftarrow 0$;  $i \leftarrow MinSize$;  $NormalSize \leftarrow 8\ KB$;
if $n \leq MinSize$ then
    return $n$;
if $n \geq MaxSize$ then
    $n \leftarrow MaxSize$;
else if $n \leq NormalSize$ then
    $NormalSize \leftarrow n$;
for ; $i < NormalSize$; $i{+}{+}$; do
    $fp = (fp << 1) + Gear[\ src[i]\ ]$;
    if ! ( $fp$ & $MaskS$ ) then
        return $i$;        //if the masked bits are all '0';
for ; $i < n$; $i{+}{+}$; do
    $fp = (fp << 1) + Gear[\ src[i]\ ]$;
    if ! ( $fp$ & $MaskL$ ) then
        return $i$;        //if the masked bits are all '0';
return $i$;
```

Algorithm 3. RabinCDC8KB(With NC)

```
Input: data buffer, $src$; buffer length, $n$
Output: chunking breakpoint $i$
$MinSize \leftarrow 2\ KB$;   $MaxSize \leftarrow 64\ KB$;
$fp \leftarrow 0$;  $i \leftarrow MinSize$;  $NormalSize \leftarrow 8\ KB$;
if $n \leq MinSize$ then
    return $n$;
if $n \geq MaxSize$ then
    $n \leftarrow MaxSize$;
else if $n \leq NormalSize$ then
    $NormalSize \leftarrow n$;
for ; $i < NormalSize$; $i{+}{+}$; do
    $fp = ((fp \hat{\ } U(a)) << 8) | b \hat{\ } T[fp >> N]$;
    if $fp$ & $(8192 * 2 - 1)$ $== 0x78$ then
        return $i$;
for ; $i < n$; $i{+}{+}$; do
    $fp = ((fp \hat{\ } U(a)) << 8) | b \hat{\ } T[fp >> N]$;
    if $fp$ & $(8192/2 - 1)$ $== 0x78$ then
        return $i$;
return $i$;
```

As shown in Algorithm 3.6, FastCDC uses normalized chunking to divide the chunking judgment into two loops with the optimized hash judgment. Note that FastCDC without normalized chunking is not shown here but can be easily implemented by using the new hash-judging statement "! fp & $MaskA$" where the $MaskA$ is padded with 35 zero bits to enlarge the sliding window size to 48 bytes as that used in the Rabin-based CDC [6]. Note that $MaskA$, $MaskS$, and $MaskL$ are three empirically derived values where the padded zero bits are almost evenly distributed for slightly higher deduplication ratio according to our large scale tests.

FastCDC implements normalized chunking by using mask value $MaskS$ and $MaskL$ to make the chunking judgment harder or easier (to generate chunks smaller or larger than the expected chunk size) when the current position is smaller or larger than the expected chunk size, respectively. And the number of '1' bits in $MaskS$ and $MaskL$ can be changed for different normalization levels. The minimum chunk size used in Algorithm 3.6 is $2\ KB$, which can be enlarged to $4\ KB$ or $8\ KB$ to further speed up the CDC process while combining with normalized chunking. Tuning the parameters of minimum chunk size and normalization level will be studied and evaluated in the next Section.

In addition, we implement the normalized chunking scheme in Rabin-based CDC as shown in Algorithm 3.6. This improved Rabin-based CDC is also evaluated in the next Section. Note that the hash judgment optimization for Gear is not applied for Rabin. This is because there are many zero hash values generated by Rabin [9], which results too many positions satisfying the chunking judgment "$(fp\&MaskValue) == false$" and thus the generated average chunk size after Rabin-based CDC will be far blow the expected average chunk size.

3.7 Rolling Two Bytes Each Time

Besides the techniques mentioned in the above subsection, we also propose another independent technique called "rolling two bytes each time" on top of FastCDC. As shown in Algorithm 3.7, the core idea of this technique is that the fingerprint fp is rolling two bytes each time (i.e., "$fp << 2$") in contrast to the traditional way of rolling one byte each time (see Algorithm 3.2), and then we judge the even and odd bytes respectively to determine the chunks' boundaries. Specifically, ① for the even bytes, we use the *Gear_ls* table (*Gear_ls* contains elements from the *Gear* table, which are all left shift one bit) and the mask value *MaskA_ls* (i.e., *MaskA* $<<1$) for the hash judgment in FastCDC, this is because when we judge the *fp* for the even bytes, *fp* has been already left shift two bits (as described in Algorithm 3.7); ② for the odd bytes, we process the fingerprints using *Gear* table and the mask value *MaskA* in the traditional way.

Algorithm 4. Rolling Two Bytes Each Time on FastCDC8KB (Without NC for Simplicity)

```
Input: data buffer, $src$; buffer length, $n$
Output: chunking breakpoint $i$
$MaskA \leftarrow 0x0000d93003530000LL$;      // 13 '1' bits ;
$MaskA\_ls \leftarrow (MaskA << 1)$;   $fp \leftarrow 0$;  $i \leftarrow MinSize$;
$MinSize \leftarrow 2KB$;     $MaxSize \leftarrow 64KB$;
if $n \leq MinSize$ then
    return $n$;
if $n \geq MaxSize$ then
    $n \leftarrow MaxSize$;
while $i < (n/2)$ do
    $fp = (fp << 2) + Gear\_ls[\ src[2 * i]\ ]$;
    if ! ( $fp$ & $MaskA\_ls$ ) then
        return $2 * i$;
    $fp += Gear[\ src[2 * i + 1]\ ]$;
    if ! ( $fp$ & $MaskA$ ) then
        return $2 * i + 1$;
return $n$;
```

TABLE 3
A Comparison Among the Rabin-Based CDC (RC), Gear-Based CDC (GC), and FastCDC (FC) Approaches in Terms of the Deduplication Ratio and the Average Size of Generated Chunks, as a Function of the Expected Chunk Size

Dataset	CDC	Expected Chunk Size of 4K (B)		Expected Chunk Size of 8K (B)		Expected Chunk Size of 16K (B)	
		Dedup Ratio	Avg. Chunk Size	Dedup Ratio	Avg. Chunk Size	Dedup Ratio	Avg. Chunk Size
TAR	RC	55.02%	5770	46.66%	12449	38.62%	25168
	GC	51.20% (–6.94%)	6786 (+17.6%)	43.55% (–6.67%)	14120 (+13.4%)	34.94% (–9.52%)	30919 (+22.9%)
	FC	54.39% (–1.14%)	5759 (–0.19%)	46.65% (-0.02%)	12334 (–0.92%)	38.68% (+1.55%)	25388 (+0.87%)
LNX	RC	96.65%	3847	96.30%	6021	95.94%	8261
	GC	96.72% (+0.07%)	3501 (–8.99%)	96.37% (+0.07%)	5684 (–5.59%)	96.01% (+0.07%)	8007 (–3.07%)
	FC	96.65% (–0.00%)	3860 (+0.33%)	96.31% (+0.01%)	6012 (–0.15%)	95.95% (+0.01%)	8246 (–0.18%)
WEB	RC	87.38%	5029	75.98%	11301	63.77%	23221
	GC	74.00% (–15.3%)	7264 (+44.4%)	57.37% (–24.5%)	19460 (+72.2%)	31.86% (–50.0%)	38888 (+67.5%)
	FC	90.02% (+3.02%)	5426 (+7.89%)	83.20% (+9.50%)	11552 (+2.22%)	72.92% (+14.3%)	23402 (+0.77%)
VMA	RC	41.63%	6535	36.70%	13071	31.38%	26191
	GC	41.11% (–1.24%)	5894 (–9.80%)	35.88% (–2.23%)	12419 (–4.98%)	30.52% (–2.74%)	24960 (–4.70%)
	FC	41.61% (–0.04%)	6468 (–1.02%)	36.40% (–0.81%)	13150 (+0.60%)	31.19% (–0.60%)	26334 (+0.54%)
VMB	RC	96.40%	5958	96.12%	11937	95.75%	24100
	GC	96.41% (+0.01%)	5622 (–5.63%)	96.05% (–0.07%)	11477 (–3.85%)	95.66% (–0.09%)	23260 (–3.48%)
	FC	96.39% (–0.01%)	6021 (+1.05%)	96.08% (+0.04%)	12138 (+1.68%)	95.70% (–0.05%)	24384 (+1.17%)
RDB	RC	95.35%	5473	92.57%	10964	87.38%	21946
	GC	95.15% (–0.20%)	5830 (+6.52%)	92.26% (–0.33%)	11666 (+6.40%)	86.82% (–0.64%)	23307 (+6.20%)
	FC	95.32% (–0.03%)	5479 (+0.10%)	92.58% (+0.01%)	10970 (+0.05%)	87.39% (+0.01%)	21909 (–0.16%)
SYN	RC	98.27%	5828	97.36%	11663	96.03%	23349
	GC	98.45% (+0.18%)	4922 (–15.5%)	97.65% (+0.29%)	9818 (–15.8%)	96.44% (+0.41%)	19624 (–16.0%)
	FC	98.28% (+0.01%)	5799 (–0.49%)	97.37% (+0.01%)	11598 (–0.55%)	96.04% (+0.01%)	23247 (–0.43%)

This technique is simple but very useful for FastCDC since it reduces one shift operation when chunking each two bytes compared with the traditional approach (rolling one byte each time as shown in Algorithm 3.2) while ensuring exactly the same chunking results. Note that it requires to lookup one more table and increases additional computation operations of '$2*i$' and '$2*i+1$', but these overheads are minor and FastCDC using this technique is about 30-40 percent faster than rolling one byte each time according to our evaluation results discussed in the next section.

4 Performance Evaluation

4.1 Experimental Setup

Experimental Platform. To evaluate FastCDC, we implement a prototype of the data deduplication system on the Ubuntu 18.04.1 operating system running on an Intel Xeon(R) Gold 6,130 processor at 2.1 GHz, with a 128 GB RAM. To better evaluate the chunking speed, another Intel i7-8700 processor at 3.2 GHz is also used for comparison.

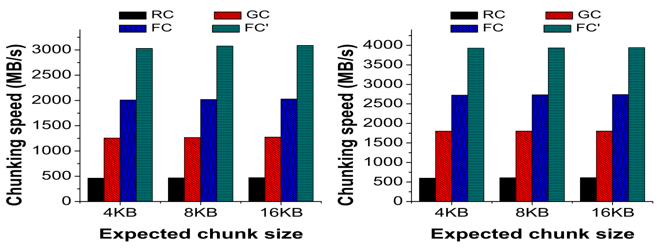
Configurations for CDC and Deduplication. Three CDC approaches, Rabin-, Gear-, and AE-based CDC, are used as the baselines for evaluating FastCDC. Rabin-based CDC is implemented based on the open-source project LBFS [6] (also used in many published studies [7], [19] or project [51]), where the sliding window size is configured to be 48 bytes. The Gear- and AE-based CDC schemes are implemented according to the algorithms described in their papers [13], [24], and we obtain performance results similar to and consistent with those reported in these papers. Here all the CDC approaches are configured with the maximum and minimum chunk sizes of $8\times$ and $\frac{1}{4}\times$ of the expected chunk size, the same as configured in LBFS [6]. The deduplication prototype consists of approximately 3,000 lines of C code, which is compiled by GCC 7.4.0 with the "-O3" compiler option to to maximize the speed of the resulting executable.

Performance Metrics of Interest. Chunking speed is measured by the in-memory processing speed of the evaluated CDC approaches and obtained by the average speed of five runs. *Deduplication ratio* is measured in terms of the percentage of duplicates detected after CDC, i.e., $\frac{\text{The size of duplicate data detected}}{\text{Total data size before deduplication}}$. *Average chunk size* after CDC is $\frac{\text{Total data size}}{\text{Number of chunks}}$, which reflects the metadata overhead for deduplication indexing.

Evaluated Datasets. Seven datasets with a total size of about 6 TB are used for evaluation as shown in Table 2. These datasets consist of the various typical workloads of deduplication, including the source code files, virtual machine images, database snapshots, etc., whose deduplication ratios vary from 40 to 98 percent, which will be detailed in Table 3 in the next subsection.

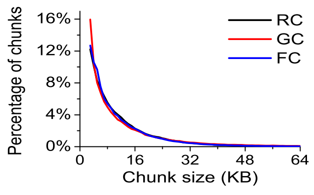
4.2 A Study of Optimizing Hash Judgment

This subsection discusses an empirical study of FastCDC using techniques of the optimized hash judgment and 'rolling two bytes each time'. Fig. 10 shows the chunking speed of the four CDC approaches running on the RDB dataset, as a function of the expected chunk size and all using the minimum chunk size of $\frac{1}{4}\times$ of that for cut-point skipping. In general, the Rabin-based CDC has the lowest speed, and Gear-based CDC are about $3\times$ faster than Rabin. FastCDC using optimized hash judgement (i.e., FC′ in Fig. 10) is

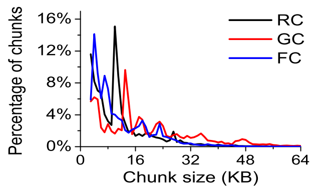


(a) Intel Gold 6130 (b) Intel i7-8700

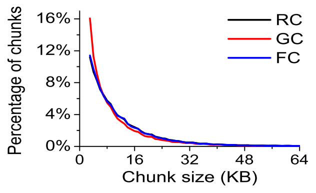
Fig. 10. Chunking speed, as a function of the expected chunk size, of Rabin-based CDC (RC), Gear-based CDC (GC), FastCDC using optimized hash judgement (FC) and rolling two bytes each time (FC′) on two CPU processors.



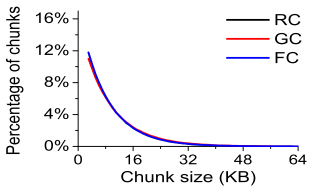
(a) Dataset TAR



(b) Dataset WEB

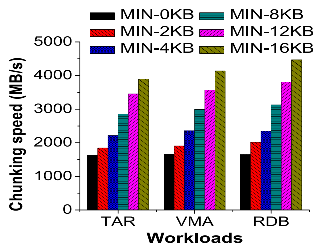


(c) Dataset VMA

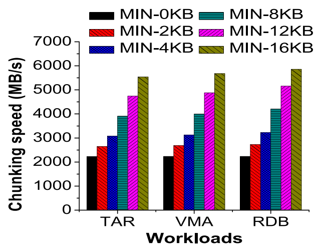


(d) Dataset RDB

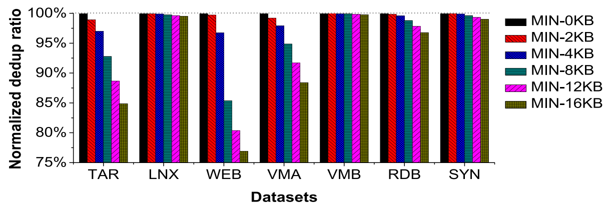
Fig. 11. Chunk-size distribution of the RC, GC, and FC approaches on the four typical datasets.



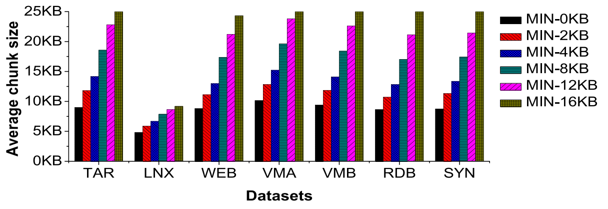
(a) Speed on intel Gold 6130



(b) Speed on intel i7-8700



(c) Deduplication ratio



(d) Average chunk size

Fig. 12. Chunking performance of FastCDC with the expected chunk size of 8KB but different minimum chunk sizes on two different CPU processors.

about 5× faster than Rabin and 1.5× faster than Gear regardless of the speed of the CPU processor and the expected chunk size. The high chunking speed of FastCDC stems from its simplification of the hash judgment after the fast Gear hash is used for chunking as described in Section 3.3. Meanwhile, FastCDC using 'rolling two bytes each time' (i.e., FC′ in Fig. 10) further increases the chunking speed by 40-50 percent since it further reduces calculation operation during CDC. Note that FC′ achieves exactly the same chunking results as FC, thus we do not discuss metrics of deduplication ratio and generated chunk size for FC′ in the remainder of this paper.

Table 3 shows the deduplication ratio and the average size of generated chunks (post-chunking) achieved by the three CDC approaches. We compare the Gear-based CDC (GC), and FastCDC (FC) approaches against the classic Rabin-based CDC (i.e., the baseline: RC) and record the percentage differences (in parentheses).

In general, FastCDC achieves nearly the same deduplication ratio as Rabin regardless of the expected chunk size and workload, and the difference between them is tiny as shown in the 3rd, 5th, 7th columns in Table 3 except on the WEB dataset. On the other hand, the Gear-based CDC has a much lower deduplication ratio on the datasets TAR and WEB due to its limited sliding window size as discussed in Section 3.2.

For the metric of the average size of generated chunks, the difference between the Rabin-based CDC and FastCDC is smaller than ±1.0 percent on most of the datasets. For the datasets WEB, FastCDC has 7.89 percent larger average chunk size than Rabin-based CDC, which is acceptable since the larger average chunk size means fewer chunks and fingerprints for indexing in a deduplication system (without sacrificing deduplication ratio) [3]. But for the Gear-based CDC, the average chunk size differs significantly in some datasets while its deduplication ratio is still a bit lower than other CDC approaches due to its smaller sliding window size.

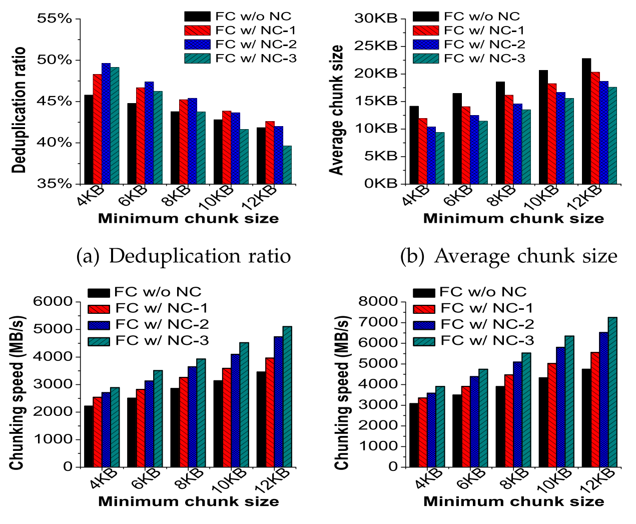
We also compare the chunk-size distributions of the three tested chunking approaches in Fig. 11: FastCDC has nearly the same chunk-size distribution as Rabin on datasets TAR, VMA, and RDB, which generally follows the exponential distribution as discussed in Section 3.4. Note that the results in Fig. 11b are very different from others. This is because there are many zero bytes in this dataset according to our observation, which makes the chunking fingerprints not so random (thus not follow the the exponential distribution). However, comparing with Rabin and Gear, FastCDC's chunk-size distribution on WEB is the most similar to other datasets, which explains why FastCDC achieves the highest deduplication ratio on WEB among the three tested chunking approaches (see Table 3).

In summary, FastCDC with the optimized hash judgment achieves a chunking speed that is 5× higher than Rabin-based CDC while satisfactorily solving the problems of low deduplication ratio and smaller sliding window size faced by Gear-based CDC.

4.3 Evaluation of Cut-Point Skipping

This subsection discusses the evaluation results of cut-point skipping technique. Figs. 12a and 12b show the impact of applying different minimum chunk sizes on the chunking speed of FastCDC. Since the chunking speed is not so sensitive to the workloads, we only show the three typical workloads in Fig. 12. In general, cut-point skipping greatly accelerates the CDC process since the skipped region will not be hash-processed by CDC. The speedup of the FastCDC applying the minimum chunk sizes of 4 and 2 KB over the FastCDC without the constraint of the minimum chunk size (i.e., Min-0 KB) is about 1.25× and 1.50× respectively, which is almost consistent with the equation $1+\frac{\textit{the minimum chunk size}}{\textit{the expected chunk size}}$ as discussed in Section 3.4.

Figs. 12c and 12d show the impact of applying different minimum chunk sizes on the deduplication ratio and average



(a) Deduplication ratio (b) Average chunk size

(c) Speed on intel Gold 6130 (d) Speed on intel i7-8700

Fig. 13. Evaluation of comprehensive performance of normalized chunking with different normalization levels.

generated chunk size of FastCDC. In general, deduplication ratio declines with the increase of the minimum chunk size applied in FastCDC, but not proportionally. For the metric of the average generated chunk size in FastCDC, it is approximately equal to the summation of the expected chunk size and the applied minimum chunk size. This means that the MIN-4 KB solution has the average chunk size of *8+4=12 KB*, leading to fewer chunks for fingerprints indexing in deduplication systems. Note that the increased portion of the average generated chunk size is not always equal to the size of the applied minimum chunk size, because the Rabin hashes of contents may not strictly follow the uniform distribution (as described in Equation (3) in Section 3.4) on some datasets.In addition, the average chunk sizes of dataset LNX are smaller than the minimum chunk size, which results from the many very small files whose sizes are much smaller than the minimum chunk size in LNX.

In summary, the results shown in Fig. 12 suggest that cut-point skipping helps obtain higher chunking speed and increase the average chunk size but at the cost of decreased deduplication ratio. The decreased deduplication ratio will be addressed by normalized chunking as evaluated in the next two subsections.

4.4 Evaluation of Normalized Chunking

In this subsection, we conduct a sensitivity study of normalized chunking (NC) on the TAR dataset, as shown in Fig. 13. Here the expected chunk size of FastCDC without NC is 8 KB and the normalized chunk size of FastCDC with NC is configured as the 4 KB + minimum chunk size. The normalization levels 1, 2, 3 refer to the three pairs of numbers of effective mask bits (14, 12), (15, 11), (16, 10) respectively that normalized chunking applies when the chunking position is smaller or larger than the normalized (or expected) chunk size, as discussed in Section 3.5.

Figs. 13a and 13b suggest that normalized chunking (NC) detects more duplicates when the minimum chunk size is about 4, 6, and 8 KB but slightly reduces the average generated chunk size, in comparison with FastCDC without NC. This is because NC reduces the number of large-sized chunks as shown in Fig. 9 and discussed in Section 3.5. The results also suggest that NC touches the "sweet spot" of deduplication ratio at the normalization level of 2 when the minimum chunk size is 4, 6, or 8 KB. This is because the very high normalization levels tend to have a similar chunk-size distribution to the Fixed-Size Chunking as shown in Fig. 9 in Section 3.5, which fails to address the *boundary-shift* problem and thus detects fewer duplicates. Figs. 13c and 13d suggest that NC, when combined with the approach of enlarging the minimum chunk size for cut-point skipping, greatly increases the chunking speed on the two tested processors.

In general, considering the three metrics of chunking speed, average generated chunk size, and deduplication ratio as a whole, as shown in Fig. 13, NC-2 with MinSize of 8 KB maximizes the chunking speed without sacrificing the deduplication ratio. Note that NC-2 with MinSize of 6 KB achieves the highest deduplication ratio among those NC approaches whose average chunk size $\geq$ that of Rabin and FastCDC tested in Table 3 (with the minimum chunk size of 2 KB).

4.5 Comprehensive Evaluation of FastCDC

In this subsection, we comprehensively evaluate the performance of FastCDC with the combined capability of the five key techniques: Gear-based rolling hash, optimizing hash judgment, cut-point skipping, rolling two bytes each time, and normalized chunking using "NC-2" and minimum chunk size of 6 KB/8 KB as suggested by the last subsection. Finally, twelve CDC approaches are tested for evaluation:

- RC-v1 (or RC-MIN-2 KB) is Rabin-based CDC used in LBFS [6]; RC-v2 and RC-v3 refer to Rabin-based CDC using normalized chunking with a minimum chunk size of 4 and 6 KB respectively.
- FC-v1 is FastCDC uses the techniques of optimizing hash judgment and cut-point skipping with a minimum chunk size of 2 KB; FC-v2 and FC-v3 refer to FastCDC using all the four techniques with a minimum chunk size of 6 and 8 KB, respectively.
- FC′-v1, FC′-v2, and FC′-v3 are FastCDC using the technique of rolling two bytes each time on top of FC-v1, FC-v2, and FC-v3 respectively.
- AE-v1 and AE-v2 refer to AE-based CDC [24] and its optimized version [52];
- Fixed-Size Chunking (FIXC) is also tested for comparison using the average chunk size of 10 KB (to better understand content-defined chunking).

Evaluation results in Table 4 suggest that FC-v1, FC-v2, AE-v2, and RC-v2 achieves nearly the same deduplication ratio as RC-v1 in most cases, which suggests that the normalized chunking scheme works well on both Rabin and FastCDC. Note that FIXC works well on the datasets LNX and VMB, because LNX has many files smaller than the fixed-size chunk of 10 KB (and thus the average generated chunk size is also smaller than 10 KB) and VMB has many structured backup data (and thus VMB is suitable for FIXC).

Table 5 shows that RC-v1, RC-v2, AE-v1, AE-v2, FC-v1, and FC-v2 generate similar average chunk size. But the approaches of RC-v3 and FC-v3 has a much larger average chunk size, which means that it generates fewer chunks and thus less metadata for deduplication processing. Meanwhile,

TABLE 4
Comparison of Deduplication Ratio Achieved by the Nine Chunking Approaches

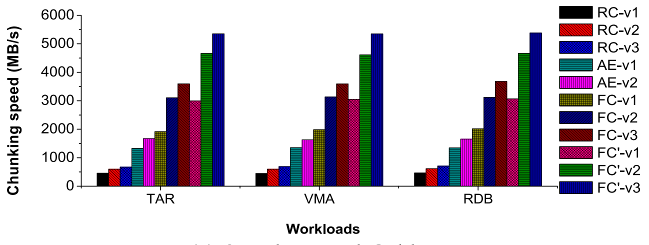
Dataset	FIXC	RC-v1	RC-v2	RC-v3	AE-v1	AE-v2	FC-v1	FC-v2	FC-v3
TAR	15.77%	46.66%	47.42%	45.37%	43.62%	46.41%	46.65%	47.39%	45.40%
LNX	95.68%	96.30%	96.28%	96.19%	96.25%	96.13%	96.31%	96.28%	96.19%
WEB	59.96%	75.98%	83.16%	80.39%	83.08%	83.18%	83.20%	83.29%	80.92%
VMA	17.63%	36.70%	37.79%	36.52%	38.10%	38.17%	36.40%	37.66%	36.39%
VMB	95.68%	96.12%	96.17%	96.11%	95.82%	96.15%	96.08%	96.17%	96.11%
RDB	16.39%	92.57%	92.96%	92.24%	88.82%	92.83%	92.58%	92.97%	92.23%
SYN	79.46%	97.36%	97.91%	97.67%	97.54%	97.86%	97.37%	97.90%	97.67%

TABLE 5
Average Chunk Size Generated by the Nine Chunking Approaches on the Seven Datasets

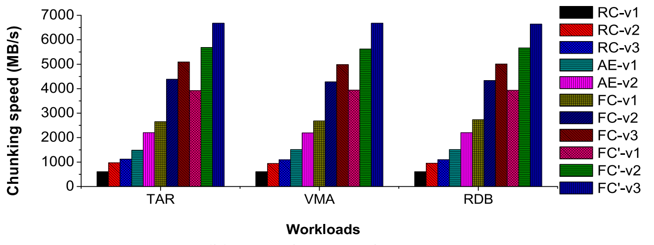
Dataset	FIXC	RC-v1	RC-v2	RC-v3	AE-v1	AE-v2	FC-v1	FC-v2	FC-v3
TAR	10239	12449	12664	14772	12187	12200	12334	12801	14918
LNX	6508	6021	7041	7636	6274	6162	6012	7042	7636
WEB	10240	11301	12174	14148	11977	11439	11552	11880	13951
VMA	10239	13071	13505	15628	13098	13559	13150	13595	15746
VMB	10239	11937	12970	15094	12303	12254	12138	13034	15166
RDB	10239	10964	12587	14728	11943	12102	10970	12583	14725
SYN	10240	11663	12221	14271	11956	11997	11598	12239	14289

RC-v3 and FC-v3 still achieves a comparable deduplication ratio, slightly lower than RC-v1 as shown in Table 4, while providing a much higher chunking speed as discussed later.

Fig. 14 suggests that FC′-v3 has the highest chunking speed, about 12× faster than the Rabin-based approach, about 2.5× faster than FC-v1. This is because FC′-v3 is the final FastCDC using all the five techniques to speed up the CDC process. In addition, FC′-v2 is also a good CDC candidate since it has a comparable deduplication ratio while also working well on the other two metrics of chunking speed and average generated chunk size. Meanwhile, normalized chunking also helps accelerate Rabin-based CDC (i.e., RC-v2 and RC-v3) while achieving comparable deduplication ratio and average chunk size. But this acceleration is limited since the main bottleneck for Rabin-based CDC is still the rolling hashing computation.



(a) Speed on intel Gold 6130



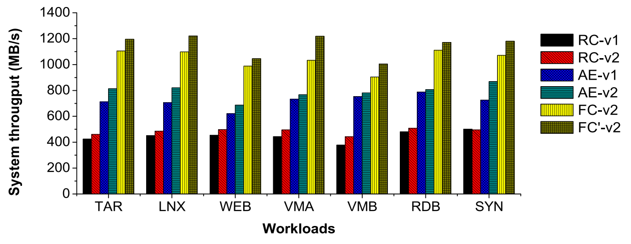
(b) Speed on intel i7-8700

Fig. 14. Chunking speed of the 11 CDC approaches.

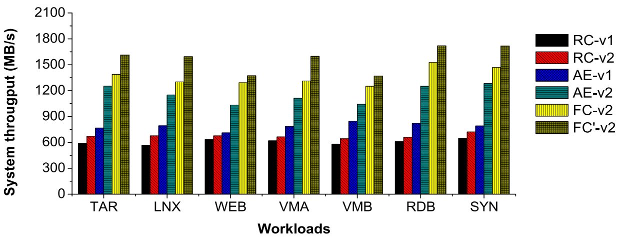
Table 6 further studies the CPU overhead among the eight CDC approaches. The CPU overhead is averaged on 1,000 test runs by the Linux tool "Perf". The results suggest that FC′-v3 has the fewest instructions for CDC computation, the higher instructions per cycle (IPC), and thus the least CPU time overhead, i.e., CPU cycles. Generally, FastCDC greatly reduces the number of instructions for CDC computation by using the techniques of Gear-based hashing, optimizing hash judgment, and rolling two bytes each time (i.e., FC′-v1), and then minimizes the number of computation instructions by enlarging the minimum chunk size for cut-point skipping and combining normalized chunking (i.e., FC′-v2 and FC′-v3). In addition, FastCDC increases the IPC for the CDC computation by well pipelining the instructions of hashing and hash-judging tasks in up-to-date processors. Therefore, these results clearly reveal the reason why FastCDC is much faster than Rabin- and AE-based CDC is that the former not only reduces the number of instructions and branches, but also increases the IPC for the CDC process.

TABLE 6
Number of Instructions, Instructions Per Cycle (IPC), and CPU Cycles Required to Chunk Data *Per Byte* by the 11 CDC Approaches on the Intel i7-8770 Processor

Approaches	Instructions	IPC	CPU cycles	branches
RC-v1	19.54	2.49	7.85	2.44
RC-v2	11.22	2.30	4.88	1.02
RC-v3	9.72	2.27	4.28	0.88
AE-v1	11.75	3.77	3.12	3.84
AE-v2	7.00	3.08	2.27	2.00
FC-v1	7.32	3.89	1.88	1.63
FC-v2	4.89	3.83	1.28	1.02
FC-v3	4.23	3.72	1.14	0.88
FC′-v1	5.28	3.87	1.36	1.13
FC′-v2	3.57	3.59	0.99	0.76
FC′-v3	3.09	3.47	0.89	0.66



(a) Throughputs on intel Gold 6130



(b) Throughputs on intel i7-8700

Fig. 15. System throughputs of Destor running with the six CDC approaches on the two CPUs.

In summary, as shown in Tables 4, 5, 6 and Fig. 14, FastCDC (i.e., FC′-v2 recommended) significantly speeds up the *Content-Defined Chunking* process and achieves a *comparable and even higher deduplication ratio* with the similar average chunk size by using a combination of the five key techniques proposed in Section 3.

4.6 Impact of CDC on Overall System Throughput

To understand the impact of the different CDC algorithms on the overall throughput of data deduplication system, we implemented them in the open-source Destor deduplication system [19]. In this evaluation, we use a Ramdisk-driven emulation to avoid the performance bottleneck caused by disk I/O. And for each dataset, we only use 5 GB, a small part of its total size in the evaluation. In addition, to examine the maximum impact of different CDC algorithms on the system throughput, we configure Destor with: (1) using the fast intel ISA-L library for SHA1 computation [53] (SHA1 speed would be about 3-4 GB/s on our tested CPUs); (2) indexing all fingerprints in RAM; (3) pipelining the deduplication subtasks (i.e., chunking, fingerprinting, indexing, etc.).

Fig. 15 shows that FastCDC (i.e., FC′-v2) helps achieve about 1.2-3.0X higher overall system throughout than RC-v1, RC-v2, AE-v1, and AE-v2, while achieving a comparable or even higher deduplication ratio as shown in Table 4. This is because when Destor pipelines the deduplication subtasks and the CDC becomes the bottleneck of the system, acceleration of the CDC can directly benefit the overall system throughput before the system meets another performance bottleneck.

5 Conclusion

In this paper, we propose FastCDC, a much faster CDC approach for data deduplication than the state-of-the-art CDC approaches while achieving a comparable deduplication ratio. The main idea behind FastCDC is the combined use of five key techniques, namely, Gear-based fast rolling hashing, optimizing the hash judgment for chunking, subminimum chunk cut-point skipping, normalized chunking, and rolling two bytes each time. Our experimental evaluation demonstrates that FastCDC obtains a chunking speed that is about 3-12× higher than that of the state-of-the-art CDC approaches while achieving nearly the same deduplication ratio as the classic Rabin-based CDC. In addition, our study of overall system throughput shows that Destor [19] using FastCDC helps achieve about 1.2-3.0X higher overall system throughout than using other CDC approaches.

FastCDC has been adopted as the default chunker in several Github projects (for quickly detecting duplicate contents), such as Rdedup [20], Content Blockchain [21], etc. We have also released the FastCDC source code at https://github.com/Borelset/destor/tree/master/src/chunkingto be shared with the deduplication and storage systems research community.

Acknowledgments

The authors are grateful to the anonymous reviewers for their insightful comments and constructive suggestions. This research was partly supported by NSFC No. 61972441, No. 61872110, No. 61872414, No. 61772212, No. 61821003, No. 61772222, and No. 61832007, the major project of National Science and Technology of China No. 2017ZX01032-101, the Open Project Program of Wuhan National Laboratory for Optoelectronics No. 2018WNLOKF008, the Shenzhen Science and Technology Program under Grant No. JCYJ20190806143405318, Key R&D Program for Guangdong Province under Grant No.2019B010136001, and the US National Science Foundation under Grant CCF-1704504 and CCF-1629625. The preliminary manuscript appeared in the proceedings of USENIX ATC 2016. In this journal version, we included more techniques on FastCDC and additional measurement results from our analysis and testbed experiments.

References

[1] D. Meyer and W. Bolosky, "A study of practical deduplication," in *Proc. 9th USENIX Conf. File Storage Technol.*, 2011, Art. no. 1.
[2] A. El-Shimi *et al.*, "Primary data deduplication–large scale study and system design," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2012, Art. no. 26.
[3] G. Wallace *et al.*., "Characteristics of backup workloads in production systems," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, Art. no. 4.
[4] P. Shilane *et al.*, "WAN optimized replication of backup datasets using stream-informed delta compression," in *Proc. FAST.*
[5] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage," in *Proc. 1st USENIX Conf. File Storage Technol.*, 2002, pp. 7–es.
[6] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *Proc. 18th ACM Symp. Operating Syst. Princ.*, 2001.
[7] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel, "A study on data deduplication in HPC storage systems," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, pp. 1–11.
[8] C. Policroniades and I. Pratt, "Alternatives for detecting redundancy in storage systems data," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 2004, Art. no. 6.
[9] M. O. Rabin, *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Lab. Univ., 1981.
[10] A. Broder, "Some applications of Rabin's fingerprinting method," in *Sequences II: Methods in Communications, Security, and Computer Science*. Berlin, Germany: Springer, 1993, pp. 1–10.
[11] C. Dubnicki, E. Kruus, K. Lichota, and C. Ungureanu, "Methods and systems for data management using multiple selection criteria," U.S. Patent App. 11/566,122, Dec. 1, 2006.

[12] B. Aggarwal *et al.*, "EndRE: An end-system redundancy elimination service for enterprises," in *Proc. 7th USENIX Conf. Netw. Syst. Des. Implementation*, 2010, Art. no. 28.

[13] W. Xia *et al.*,"Ddelta: A deduplication-inspired fast delta compression approach," *Perform. Eval.*, vol. 79, pp. 258–272, 2014.

[14] W. Xia *et al.*, "P-Dedupe: Exploiting parallelism in data deduplication system," in *Proc. IEEE 7th Int. Conf. Netw. Archit. Storage*, 2012, pp. 338–347.

[15] M. D. Lillibridge, "Parallel processing of input data to locate landmarks for chunks," U.S. Patent 8 001 273, Aug. 16, 2011.

[16] S. Al-Kiswany *et al.*, "StoreGPU: Exploiting graphics processing units to accelerate distributed storage systems," in *Proc. 17th Int. Symp. High Perform. Distrib. Comput.*, 2008, pp. 165–174.

[17] A. Gharaibeh *et al.*, "A GPU accelerated storage system," in *Proc. 19th ACM Int. Symp. High Perform. Distrib. Comput.*, 2010, pp. 167–178.

[18] P. Bhatotia, R. Rodrigues, and A. Verma, "Shredder: GPU-accelerated incremental storage and computation," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, Art. no. 14.

[19] M. Fu *et al.*, "Design tradeoffs for data deduplication performance in backup workloads," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 331–344.

[20] Rdedup Project. [Online]. Available: https://github.com/dpc/rdedup

[21] Content Blockchain Project. [Online]. Available: https://github.com/coblo

[22] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proc. 6th USENIX Conf. File Storage Technol.*, 2008, Art. no. 18.

[23] W. Xia *et al.*, "A comprehensive study of the past, present, and future of data deduplication,"

[24] Y. Zhang *et al.*, "AE: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication," in *Proc. IEEE Conf. Comput. Commun.*, 2015, pp. 1337–1345.

[25] Y. Cui *et al.*, "QuickSync: Improving synchronization efficiency for mobile cloud storage services," in *Proc. 21st Annu. Int. Conf. Mobile Comput. Netw.*, 2015, pp. 592–603.

[26] C. Yu, C. Zhang, Y. Mao, and F. Li, "Leap-based content defined chunking—Theory and implementation," in *Proc. 31st Symp. Mass Storage Syst. Technol.*, 2015, pp. 1–12.

[27] F. Ni and S. Jiang, "RapidCDC: Leveraging duplicate locality to accelerate chunking in CDC-based deduplication systems," in *Proc. ACM Symp. Cloud Comput.*, 2019, pp. 220–232.

[28] F. Ni, X. Lin, and S. Jiang, "SS-CDC: A two-stage parallel content-defined chunking for deduplicating backup storage," in *Proc. 12th ACM Int. Conf. Syst. Storage*, 2019, pp. 86–96.

[29] K. Eshghi and H. K. Tang, "A framework for analyzing and improving content-based chunking algorithms," Hewlett Packard Laboratories, Palo Alto, CA, USA, *Tech. Rep. HPL-2005–30(R.1)*, 2005.

[30] D. Teodosiu, N. Bjorner, Y. Gurevich, M. Manasse, and J. Porkka, "Optimizing file replication over limited bandwidth networks using remote differential compression," Microsoft Research TR-2006–157, 2006.

[31] A. Anand *et al.*, "Redundancy in network traffic: Findings and implications," in *Proc. 11th Int. Joint Conf. Meas. Model. Comput. Syst.*, 2009, pp. 37–48.

[32] N. Bjørner, A. Blass, and Y. Gurevich, "Content-dependent chunking for differential compression, the local maximum approach," *J. Comput. Syst. Sci.*, vol. 76, pp. 154–203, 2010.

[33] E. Kruus, C. Ungureanu, and C. Dubnicki, "Bimodal content defined chunking for backup streams," in *Proc. 8th USENIX Conf. File Storage Technol.*, 2010, Art. no. 18.

[34] B. Romański *et al.*, "Anchor-driven subchunk deduplication," in *Proc. 4th Annu. Int. Conf. Syst. Storage*, 2011, Art. no. 16.

[35] G. Lu, Y. Jin, and D. H. Du, "Frequency based chunking for data de-duplication," in *Proc. IEEE Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2010, pp. 287–296.

[36] W. Xia *et al.*, "SiLo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2011, pp. 26–28.

[37] M. Lillibridge *et al.*, "Sparse indexing: Large scale, inline deduplication using sampling and locality," in *Proc. 7th Conf. File Storage Technol.*, 2009, pp. 111–123.

[38] W. Xia *et al.*, "Accelerating data deduplication by exploiting pipelining and parallelism with multicore or manycore processors," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, pp. 1–2.

[39] F. Guo and P. Efstathopoulos, "Building a high-performance deduplication system," in *Proc. USENIX Annu. Tech. Conf.*, 2011, Art. no. 25.

[40] W. Xia *et al.*,"Accelerating content-defined-chunking based data deduplication by exploiting parallelism," *Future Gener. Comput. Syst.*, vol. 98, pp. 406–418, 2019.

[41] W. Xia, H. Jiang, D. Feng, and Y. Hua, "Similarity and locality based indexing for high performance data deduplication," *IEEE Trans. Comput.*, vol. 64, no. 4, pp. 1162–1176, Apr. 2015.

[42] B. Debnath, S. Sengupta, and J. Li, "ChunkStash: Speeding up inline storage deduplication using flash memory," in *Proc. USENIX Annu. Tech. Conf.*, 2010, Art. no. 16.

[43] M. Lillibridge, K. Eshghi, and D. Bhagwat, "Improving restore speed for backup systems that use inline chunk-based deduplication," in *Proc. 11th USENIX Conf. File Storage Technol.*, 2013, pp. 183–198.

[44] D. Bhagwat, K. Pollack, D. D. Long, T. Schwarz, E. L. Miller, and J. Paris, "Providing high reliability in a minimum redundancy archival storage system," in *Proc. 14th IEEE Int. Symp. Model. Anal. Simul.*, 2006, pp. 413–421.

[45] Y. Zhou *et al.*, "SecDep: A user-aware efficient fine-grained secure deduplication scheme with multi-level key management," in *Proc. 31st Symp. Mass Storage Syst. Technol.*, 2015, pp. 1–14.

[46] J. Min, D. Yoon, and Y. Won, "Efficient deduplication techniques for modern backup operation," *IEEE Trans. Comput.*, vol. 60, no. 6, pp. 824–840, Jun. 2011.

[47] GNU archives. [Online]. Available: http://ftp.gnu.org/gnu/

[48] Linux archives. [Online]. Available: ftp://ftp.kernel.org/

[49] VMs archives. [Online]. Available: http://www.thoughtpolice.co.uk

[50] V. Tarasov *et al.*, "Generating realistic datasets for deduplication analysis," in *Proc. USENIX Annu. Tech. Conf.*, 2012, Art. no. 24.

[51] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2008, pp. 72–81.

[52] Y. Zhang *et al.*,"A fast asymmetric extremum content defined chunking algorithm for data deduplication in backup storage systems," *IEEE Trans. Comput.*, vol. 66, no. 2, pp. 199–211, Feb. 2017.

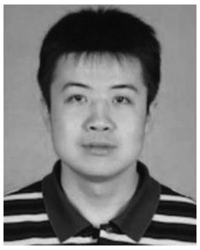
[53] Intel ISA-L: Intelligent Storage Acceleration Library. [Online]. Available: https://github.com/intel/isa-l

Wen Xia (Member, IEEE) received the PhD degree in computer science from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2014. He is currently an associate professor with the School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen. His research interests include data reduction, storage systems, cloud storage, etc. He has published more than 40 papers in major journals and conferences including the *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *Proceedings of the IEEE*, USENIX ATC, FAST, HotStorage, MSST, DCC, IPDPS, etc.

Xiangyu Zou (Student Member, IEEE) is currently working toward the PhD degree majoring in computer science at the Harbin Institute of Technology, Shenzhen, China. His research interests include data deduplication, storage systems, etc. He has published several papers in major journals and international conferences including the *Future Generation Computing Systems*, MSST, and HPCC.

Hong Jiang (Fellow, IEEE) received the BSc degree in computer engineering from the Huazhong University of Science and Technology, Wuhan, China, in 1982, the MASc degree in computer engineering from the University of Toronto, Toronto, Canada, in 1987, and the PhD degree in computer science from the Texas A&M University, College Station, Texas, in 1991. He is currently a chair and Wendell H. Nedderman endowed professor of Computer Science and Engineering Department, University of Texas at Arlington. Prior to joining UTA, he served as a program director with National Science Foundation (2013.1–2015.8) and he was with the University of Nebraska- Lincoln since 1991, where he was Willa Cather professor of computer science and engineering. His present research interests include computer architecture, computer storage systems and parallel I/O, high performance computing, big data computing, cloud computing, performance evaluation. He recently served as the associate editor of the *IEEE Transactions on Parallel and Distributed Systems* and the *IEEE Transactions on Computers*. He has more than 200 publications in major journals and international Conferences in these areas, including the *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, ISCA, MICRO, USENIX ATC, FAST, EUROSYS, SC, etc.

Yukun Zhou is currently working toward the PhD degree majoring in computer science at the Harbin Institute of Technology, Shenzhen, China. His research interests include data reduction, distributed systems, etc. He has published several papers in major journals and international conferences including the *Proceedings of the IEEE*, *Future Generation Computing Systems*, *Performance Evaluation*, USENIX ATC, MSST, IPDPS, INFOCOM, etc.



Chuanyi Liu received the PhD degrees in computer science and technology from Tsinghua University, Beijing, China, in 2010. He is currently an associate professor with the Harbin Institute of Technology, Shenzhen. His research interests include the mass storage systems, cloud computing and cloud security, and data security. He has published more than 30 papers in major journals and international conferences including the *IEEE Access*, CCS, ICDCS, ICS, etc.

Dan Feng (Member, IEEE) received the BE, ME, and PhD degrees in computer science and technology from the Huazhong University of Science and Technology (HUST), China, in 1991, 1994, and 1997, respectively. She is currently a professor and the dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, and parallel file systems. She has more than 100 publications in major journals and conferences, including the *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *ACM Transactions on Storage*, FAST, USENIX ATC, SC, etc.

Yu Hua (Senior Member, IEEE) received the BE and PhD degrees in computer science from the Wuhan University, Wuhan, China, in 2001 and 2005, respectively. He is currently a professor with the Huazhong University of Science and Technology, China. His research interests include file systems, cloud storage systems, non-volatile memory, etc. He has more than 100 papers to his credit in major journals and international conferences including *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Computers*, OSDI, MICRO, FAST, USENIX ATC, SC, etc. He serves for multiple international conferences, including ASPLOS, SOSP, FAST, USENIX ATC, ICS, RTSS, SoCC, ICDCS, INFOCOM, IPDPS, DAC, MSST, and DATE. He is a distinguished member of CCF and a senior member of ACM.

Yuchong Hu received the BE and PhD degrees in computer science from the University of Science and Technology of China, Hefei, China, in 2005 and 2010, respectively. He is currently an associate professor with the Huazhong University of Science and Technology. His research interests include network coding/erasure coding, cloud computing, and network storage. He has more than 30 publications in major journals and conferences, including the *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *IEEE Transactions on Information Theory*, *ACM Transactions on Storage* , FAST, INFOCOM, DSN, etc.

Yucheng Zhang received the PhD degree in computer science from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2018. He is currently an assistant professor with the Hubei University of Technology, Wuhan, China. His research interests include data deduplication, storage systems, etc. He has several papers in refereed journals and conferences including *IEEE Transactions on Computers*, *Future Generation Computing Systems*, *Proceedings of the IEEE*, USENIX ATC, FAST, IPDPS, INFOCOM, MSST, etc.

An Event-Driven Approach to Serverless Seismic Imaging in the Cloud

Philipp A. Witte, Mathias Louboutin, Henryk Modzelewski, Charles Jones, James Selvage, and Felix J. Herrmann

Abstract—Adapting the cloud for high-performance computing (HPC) is a challenging task, as software for HPC applications hinges on fast network connections and is sensitive to hardware failures. Using cloud infrastructure to recreate conventional HPC clusters is therefore in many cases an infeasible solution for migrating HPC applications to the cloud. As an alternative to the generic lift and shift approach, we consider the specific application of seismic imaging and demonstrate a serverless and event-driven approach for running large-scale instances of this problem in the cloud. Instead of permanently running compute instances, our workflow is based on a serverless architecture with high throughput batch computing and event-driven computations, in which computational resources are only running as long as they are utilized. We demonstrate that this approach is very flexible and allows for resilient and nested levels of parallelization, including domain decomposition for solving the underlying partial differential equations. While the event-driven approach introduces some overhead as computational resources are repeatedly restarted, it inherently provides resilience to instance shut-downs and allows a significant reduction of cost by avoiding idle instances, thus making the cloud a viable alternative to on-premise clusters for large-scale seismic imaging.

1 Introduction

Seismic imaging of the earth's subsurface is one of the most computationally expensive applications in scientific computing, as state-of-the-art imaging methods such as least-squares reverse time migration (LS-RTM), require repeatedly solving a large number of forward and adjoint wave equations during numerical optimization (e.g., [1], [2], [3]). Similar to training neural networks, the gradient computations in seismic imaging are based on backpropagation and require storage or re-computations of the state variables (i.e., of the forward modeled wavefields). Due to the large computational cost of repeatedly modeling wave propagation over many time steps using finite difference modeling, seismic imaging requires access to high-performance computing (HPC) clusters, but the high cost of acquiring and maintaining HPC cluster makes this option only viable for a small number of major energy companies [4]. For this reason, cloud computing has lately emerged as a possible alternative to on-premise HPC clusters, offering many advantages such as no upfront costs, a pay-as-you-go pricing model and theoretically unlimited scalability. Outside of the HPC community, cloud computing is today widely used by many companies for general purpose computing, data storage and analysis or machine learning. Customers of cloud providers include major companies such as General Electric (GE), Comcast, Shell or Netflix, with the latter hosting their video streaming content on Amazon Web Services (AWS) [5].

However, adapting the cloud for high-performance computing applications such as seismic imaging, is not straightforward, as numerous investigations and case studies have shown that performance, latency, bandwidth and mean time between failures (MTBF) in the cloud can vary significantly between platform providers, services and hardware, and are often inferior compared to on-premise HPC resources. Especially in the early days of cloud computing, performance and network connections were considerably slower than on comparable on-premise HPC systems, as discussed in a number of publications. An early performance analysis by Jackson [6] of a range of typical NERSC HPC applications on Amazon's Elastic Compute Cloud (EC2) found that, at the time of the comparison in 2010, applications on EC2 ran several orders of magnitude slower than on comparable HPC systems, due to low bandwidth and high latency. Other performance benchmarks from the late 2000s and early 2010s, similarly conclude that poor network performance severely limited the HPC capabilities of the cloud at that time [7], [8], [9], [10], [11], [12]. Cloud providers have since then responded by making significant improvements regarding network connections, now offering technologies such as InfiniBand, specialized network adapters such as Amazon's elastic fabric adapter (EFA, [13]) and improved virtualization techniques to improve performance (e.g., AWS Nitro [14]). Accordingly, more recent benchmarks on various cloud platforms, including AWS and Microsoft Azure, find that the performance on newly introduced HPC

- *P.A. Witte, M. Louboutin, and F.J. Herrmann are with the School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA 30308.*
  E-mail: {pwitte3, mlouboutin3, felix.herrmann}@gatech.edu.
- *H. Modzelewski is with the Department of Earth, Ocean and Atmospheric Sciences, University of British Columbia, Vancouver, BC V6T 1Z2, Canada. E-mail: hmodzelewski@eos.ubc.ca.*
- *C. Jones and J. Selvage are with Osokey Ltd., RG9 1AY Henley-on-Thames, United Kingdom. E-mail: {charles, james}@osokey.com.*

Manuscript received 20 Aug. 2019; revised 9 Mar. 2020; accepted 12 Mar. 2020. Date of publication 23 Mar. 2020; date of current version 17 Apr. 2020.
(Corresponding author: Philipp A. Witte.)
Recommended for acceptance by L. Wang.
Digital Object Identifier no. 10.1109/TPDS.2020.2982626

instances is oftentimes on par with modern on-premise HPC systems [15], [16]. Nevertheless, enhanced network technologies are typically limited to a small subset of specialized HPC instances, which are not as widely available as general purpose instances and which are accordingly more expensive [17].

Aside from network communication, several investigations [16], [18], [19] point out that embarrassingly parallel applications show very good performance that is comparable to (non-virtualized) HPC environments, even in the early days of the cloud and using standard (non-HPC optimized) nodes. Similarly, performance tests on single cloud nodes and bare-metal instances using HPCC and high-performance LINPACK benchmarks demonstrate good performance and scalability as well [20], [21]. These findings underline that the *lift and shift approach* for porting HPC applications to the cloud is unfavorable, as most HPC codes are based on highly synchronized message passing (i.e., MPI [22]) and rely on stable and fast network connections, which are only available on certain (limited) instance types and which are thus more expensive. On the other hand, individual compute nodes and architectures offered by cloud computing are indeed comparable to current supercomputing systems [21] and the cloud offers a range of novel technologies such as cloud object storage or event-driven computations [23]. These technologies are not available on traditional HPC systems and make it possible to address computational bottlenecks of HPC in fundamentally new ways. Porting HPC applications to the cloud in a way that is financially viable therefore requires a careful re-architecture of the corresponding codes and software stacks to take advantage of these technologies, while minimizing communication and idle times. This process is heavily application dependent and requires the identification of how specific applications can take advantage of specialized cloud services such as serverless compute or high throughput batch processing to mitigate resilience issues and minimize cost, while avoiding idle instances and fast network fabrics where possible.

Based on these premises, we present a workflow for large-scale seismic imaging on AWS, which does not rely on a conventional cluster of virtual machines, but is instead based on a serverless workflow that takes advantage of the mathematical properties of the seismic imaging optimization problem [24]. Similar to deep learning, objective functions in seismic imaging consist of a sum of (convex) misfit functions and iterations of associated optimization algorithms exhibit the structure of a MapReduce program [25]. The map part corresponds to computing the gradient of each element in the sum and is embarrassingly parallel to compute, but individual gradient computations are expensive as they involve solving partial differential equations (PDEs). The reduce part corresponds to the summation of the gradients and update of the model parameters and is comparatively cheap to compute, but I/O intensive. Instead of performing these steps on a cluster of permanently running compute instances, our workflow is based on specialized AWS services such as AWS Batch and Lambda, which are responsible for automatically launching and terminating the required computational resources [23], [26]. EC2 instances are only running as long as they are utilized and are shut down automatically as soon as computations are finished, thus preventing instances from sitting idle. This stands in contrast to alternative MapReduce cloud services, such as Amazon's Elastic Map Reduce (EMR), which is based on Apache Hadoop and relies on a cluster of permanently running EC2 instances [27]. In our approach, expensive gradient computations are carried out by AWS Batch, a service for processing embarrassingly parallel workloads, but with the possibility of using (MPI-based) domain decomposition for individual solutions of partial differential equations (PDEs). The cheaper gradient summations are performed by Lambda functions, a service for serverless computations, in which code is run in response to events, without the need to manually provision computational resources [23].

The following section provides an overview of the mathematical problem that underlies seismic imaging and we identify possible characteristics that can be taken advantage of to avoid the aforementioned shortcomings of the cloud. In the subsequent section, we describe our seismic imaging workflow, which has been developed specifically for AWS, but the underlying services are available on other cloud platforms (Google Compute Cloud, Azure) as well. We then present a performance analysis of our workflow on a real-world seismic imaging application, using a popular subsurface benchmark model [28]. Apart from conventional scaling tests, we also consider specific cloud metrics such as resilience and cost, which, aside from the pure performance aspects like scaling and time-to-solution, are important practical considerations for HPC in the cloud. An early application of our workflow is presented in an expanded conference abstract [29].

2 Problem Overview

Seismic imaging and parameter estimation are a set of computationally challenging inverse problems with high practical importance, as they are today widely used for geophysical exploration and monitoring geohazards. Exploration seismology is based on the manual excitation of seismic sources, which trigger sound and/or elastic waves that travel through the subsurface. At geological interfaces, waves are scattered and reflected, causing parts of the wavefield to travel back to the surface, where it is recorded by an array of receivers (Fig. 1). In a seismic survey, the source is fired repeatedly as it moves across the survey area and the observed data that is collected for each source location is denoted by $\mathbf{d}_i$. The objective of seismic imaging is to recover a physical parametrization of the subsurface from the recorded seismic data. In the setting of inverse problems, this is achieved by minimizing the misfit between recorded data and data that is predicted using numerical modeling. The *forward* problem is defined as the computation of a predicted seismic shot record through a forward modeling operator $\mathcal{F}(\mathbf{m}, \mathbf{q}_i)$, where $\mathbf{m}$ denotes the discretized unknown parameters, such as the seismic image or the acoustic wave speed and the vector $\mathbf{q}_i$ represents the location and the time signature of the seismic source. The evaluation of the forward modeling operator corresponds to numerically solving a discretized version of the wave equation for the given set of model parameters and current

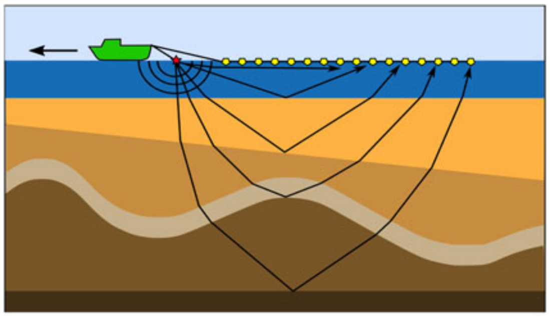


Fig. 1. In marine seismic data acquisition, a seismic vessel excites acoustic waves that travel through the subsurface. Waves are reflected and refracted at geological interfaces and travel back to the surface, where they are recorded by an array of seismic receivers. A typical seismic survey consists of several thousand of individual source experiments, during which the vessel moves across the survey area.

source location using for example finite differences (details are given in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2020.2982626.).

In the *inverse* problem, we are interested in recovering the parameters $\mathbf{m}$ from the observed seismic data $\mathbf{d}_i$. Mathematically, this is achieved by formulating an unconstrained optimization problem in which we minimize the ℓ_2-misfit between the observed and numerically modeled data [30], [31]:

$$\underset{\mathbf{m}}{\text{minimize}} \;\; \Phi(\mathbf{m}) = \sum_{i=1}^{n_s} \frac{1}{2} ||\mathcal{F}(\mathbf{m}, \mathbf{q}_i) - \mathbf{d}_i||_2^2. \tag{1}$$

In essence, the goal of seismic inversion is to find a set of model parameters $\mathbf{m}$, such that the numerically modeled data matches the observed data from the seismic survey. The total number of individual source experiments n_s for realistic surveys, i.e., the number of PDEs that have to solved for each evaluation of $\Phi(\mathbf{m})$, is quite large and lies in the range of 10^3 for 2D surveys and 10^5 for 3D surveys.

Seismic inverse problems of this form are typically solved with gradient-based optimization algorithms such as (stochastic) gradient descent, (Gauss-) Newton methods, sparsity-promoting minimization or constrained optimization (e.g., [32], [33]) and therefore involve computing the gradient of Equation (1) for all or a subset of indices i. The gradient of the objective function is given by:

$$\mathbf{g} = \sum_{i=1}^{n_s} \mathbf{J}^\top \Big(\mathcal{F}(\mathbf{m}, \mathbf{q}_i) - \mathbf{d}_i \Big), \tag{2}$$

where the linear operator $\mathbf{J} = \frac{\partial \mathcal{F}(\mathbf{m}, \mathbf{q}_i)}{\partial \mathbf{m}}$ is the partial derivative of the forward modeling operator with respect to the model parameters $\mathbf{m}$ and $\top$ denotes the matrix transpose. Both the objective function, as well as the gradient exhibit a sum structure over the source indices and are embarrassingly parallel to compute. Evaluating the objective function and computing the gradient are therefore instances of a MapReduce program [25], as they involve the parallel computation and subsequent summation of elements of the sum. However, computing the gradient for a single index i involves solving two PDEs, namely a forward wave equation and an adjoint (linearized) wave equation (denoted as a multiplication with $\mathbf{J}^\top$). For realistically sized 3D problems, the discretized model in which wave propagation is modeled has up to 10^9 variables and modeling has to be performed for several thousand time steps. The observed seismic data $\mathbf{d}_i$ $(i = 1, \ldots, n_s)$ is typically in the range of several terabytes and a single element of the data (a seismic *shot record*) ranges from several mega- to gigabytes.

The problem structure of Equation (1) is very similar to deep learning and the parallels between convolutional neural networks and PDEs have lately attracted strong attention [34]. As in deep learning, computing the gradient of the objective function (Equation (2)) is based on backpropagation and in principle requires storing the state variables of the forward problem. However, in any realistic setting the wavefields are too big to be stored in memory and therefore need to be written to secondary storage devices or recomputed from a subset of checkpoints [35]. Alternatively, domain decomposition can be used to reduce the domain size per compute node such that the forward wavefields fit in memory, or time-to frequency conversion methods can be employed to compute gradients in the frequency domain [3], [36]. In either case, computing the gradient for a given index i is expensive both in terms of necessary floating point operations, memory and IO and requires highly optimized finite-difference modeling codes for solving the underlying wave equations. Typical computation times of a single (3D-domain) gradient $\mathbf{g}_i$ (i.e., one element of the sum) are in the range of minutes to hours, depending on the domain size and the complexity of the wave simulator, and the computations have to be carried out for a large number of source locations and iterations.

The high computational cost of seismic modeling, in combination with the complexity of implementing optimization algorithms to solve Equation (1), leads to enormously complex inversion codes, which have to run efficiently on large-scale HPC clusters. A large amount of effort goes into implementing fast and scalable wave equation solvers [37], [38], as well as into frameworks for solving the associated inverse problem [39], [40], [41], [42]. Codes for seismic inversion are typically based on message passing and use MPI to parallelize the loop of the source indices (Equation (1)). Furthermore, a nested parallelization is oftentimes used to apply domain-decomposition or multi-threading to individual PDE solves. The reliance of seismic inversion codes on MPI to implement an embarrassingly parallel loop is disadvantageous in the cloud, where the mean-time-between failures (MTBF) is much shorter than on HPC systems [6] and instances using spot pricing can be arbitrarily shut down at any given time [43]. Another important aspect is that the computation time of individual gradients can vary significantly and cause load imbalances, which is problematic in the cloud, where users are billed for running instances by the second, regardless of whether the instances are in use or idle. For these reasons, we present an alternative approach for seismic imaging in the cloud based on batch processing and event-driven computations.

3 Event-Driven Seismic Imaging on AWS

3.1 Workflow

Optimization algorithms for minimizing Equation (1) essentially consists of three steps. First, the elements of the gradient $\mathbf{g}_i$ are computed in parallel for all or a subset of indices $i \in n_s$, which corresponds to the map part of a Map-Reduce program. The number of indices for which the objective is evaluated defines the batch size of the gradient. The subsequent reduce part consists of summing these elements into a single array and using them to update the unknown model/image according to the rule of the respective optimization algorithm (Algorithm 1). Optimization algorithms that fit into this general framework include variations of stochastic/full gradient descent (GD), such as Nesterov's accelerated GD [44] or Adam [45], as well as the nonlinear conjugate gradient method [46], projected GD or iterative soft thresholding [47]. Conventionally, these algorithms are implemented as a single program and the gradient computations for seismic imaging are parallelized using message passing. Running MPI-based programs of this structure in the cloud require that users request a set of EC2 instances and establish a network connection between all workers [48]. Tools like StarCluster [49] or AWS HPC [50] facilitate the process of setting up a cluster and even allow adding or removing instances to a running cluster. However, adding or dropping instances/nodes during the execution of an MPI program is not easily possible, so the number of instances has to stay constant during the entire length of the program execution, which, in the case of seismic inversion, can range from several days to weeks. This makes this approach not only prone to resilience issues, but it can result in significant cost overhead, if workloads are unevenly distributed and instances are temporarily idle.

Algorithm 1. Generic Algorithm Structure for Gradient-Based Minimization of Equation (1), Using a Fixed Number of Iterations n.

```
1: Input: batch size $n_b$, max. number of iterations $n$, step size $\alpha$,
   initial guess $\mathbf{m}_1$
2: for $i = 1$ to $n$ do
3:   Compute gradients $\mathbf{g}_i$, $i = 1, ..., n_b$ in parallel
4:   Sum gradients: $\mathbf{g} = \sum_{i=1}^{n_b} \mathbf{g}_i$
5:   Update optimization variable, e.g., using SGD:
     $\mathbf{m}_{k+1} = \mathbf{m}_k - \alpha \mathbf{g}$
6: end for
```

Instead of implementing and running optimization algorithms for seismic inverse problems as a single program that runs on a cluster of EC2 instances, we express the steps of a generic optimization algorithm through AWS Step Functions (Fig. 2) and deploy its individual components through a range of specialized AWS services [51]. Step functions allow the description of an algorithm as a collection of states and their relationship to each other using the JavaScript Object Notation (JSON). From the JSON definition of a workflow, AWS renders an interactive visual workflow in the web browser, as shown in Fig. 2. For our purpose, we use Step Functions to implement our iterative loop [52], during which we compute and sum the gradients, and use them to update the seismic image. We choose Step Functions to express our algorithm, as they allow composing different AWS Services such as AWS Batch and Lambda functions into a single workflow, thus making it possible to leverage preexisiting AWS services and to combine them into a single application. Another important aspect of Step Functions

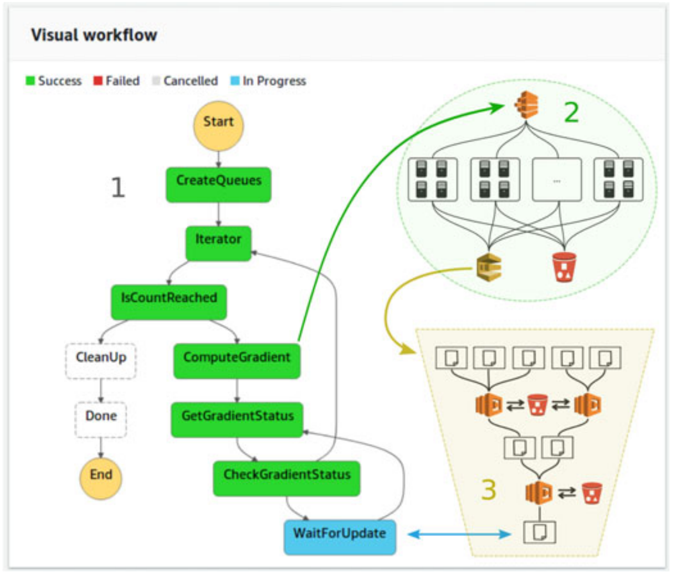


Fig. 2. A generic seismic imaging algorithm, expressed as a serverless visual workflow using AWS Step Functions (1). The workflow consists of a collection of states, which are used to implement an iterative optimization loop. Each iteration involves computing the gradient of Equation 1 using AWS Batch (2) and the subsequent event-driven summation of all gradient components using Lambda functions (3). The final Lambda function uses the summed gradient to update the optimization variable (i.e., the seismic image). Once the updated variable is detected by the `WaitForUpdate` state, the workflow automatically progresses to the next iteration.

is that the execution of the workflow itself is managed by AWS and does not require running any EC2 instances, which is why we refer to this approach as *serverless*. During execution time, AWS automatically progresses the workflow from one state to the next and users are only billed for transitions between states, but the cost is negligible compared to the cost of running EC2 instances (0.025$ per 1,000 state transitions).

States can be simple if-statements such as the `IsCountReached` state, which keeps track of the iteration number and terminates the workflow after a specified number of iterations, but states can also be used invoke other AWS services. Specifically, states can be used to invoke AWS Lambda functions to carry out serverless computations. Lambda functions allow users to run code in response to events, such as invocations through AWS Step Functions, and automatically assign the required amount of computational resources to run the code. Billing is based on the execution time of the code and the amount of used memory. Compared to EC2 instances, Lambda functions have a much shorter startup time in the range of milliseconds rather than minutes, but they are limited to 3 GB of memory and an execution time of 15 minutes. As such, Lambda functions themselves are not suitable for carrying out the gradient computations, but they can be used to manage other AWS services. In our workflow, we use Lambda functions invoked by the `ComputeGradient` state (Fig. 2) to launch AWS Batch jobs for computing the gradients. During the gradient computation, which can take up to several hours, the Step Functions check in a user-defined interval if the full gradient has been computed, before advancing the workflow to the next state. The `WaitForUpdate` state pauses

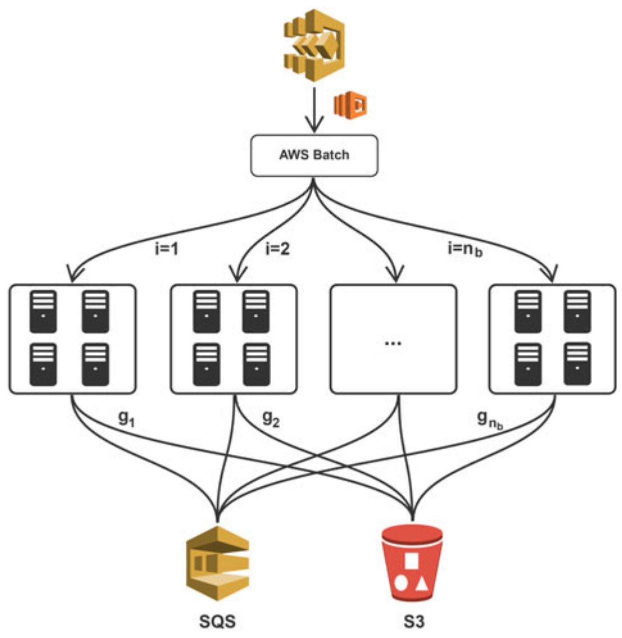


Fig. 3. The gradients of the LS-RTM objective function are computed as an embarrassingly parallel workload using AWS Batch. This process is automatically invoked by the AWS Step Functions (Fig. 2) during each iteration of the workflow. The gradients of individual source locations are computed as separate jobs on either a single or multiple EC2 instances. The resulting gradients are saved in S3 and the respective object names are sent to an SQS queue to invoke the gradient summation.

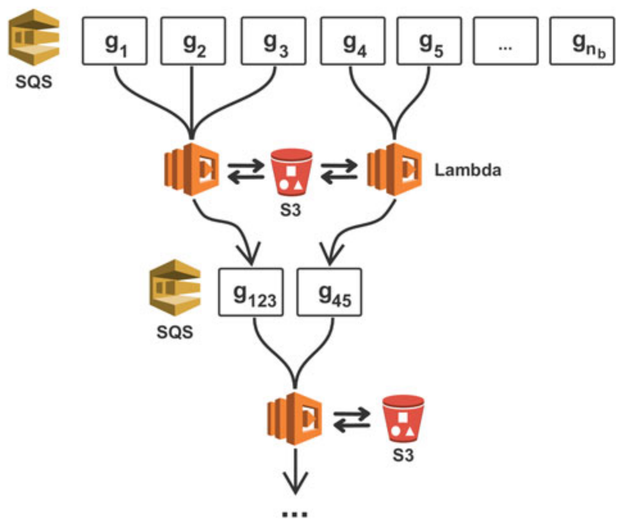


Fig. 4. Event-driven gradient summation using AWS Lambda functions. An SQS message queue collects the object names of all gradients that are currently stored in S3 and automatically invokes Lambda functions that stream up to 10 files from S3. Each Lambda function sums the respective gradients, writes the result back to S3 and sends the new object name to the SQS queue. The process is repeated until all gradients have been summed into a single S3 object. SQS has a guaranteed at-least-once delivery of messages to ensure that no objects are lost in the summation.

the workflow for a specified amount of time, during which no additional computational resources are running other than the AWS Batch job itself.

3.2 Computing the Gradient

The gradient computations (Equation (2)) are the major workload of seismic inversion, as they involve solving forward and adjoint wave equations, but the embarrassingly parallel structure of the problem lends itself to high-throughput batch computing. On AWS, embarrassingly parallel workloads can be processed with AWS Batch, a service for scheduling and running parallel containerized workloads on EC2 instances [26]. Parallel workloads, such as computing a gradient of a given batch size, are submitted to a batch queue and AWS Batch automatically launches the required EC2 instances to process the workload from the queue. Each job from the queue runs on an individual instance or set of instances, with no communication being possible between individual jobs.

In our workflow, we use the Lambda function invoked by the `ComputeGradient` state (Fig. 2) to submit the gradient computations to an AWS Batch queue. Each element of the gradient $\mathbf{g}_i$ corresponds to an individual job in the queue and is run by AWS Batch as a separate Docker container [53]. Every container computes the gradient for its respective source index i and writes its resulting gradient to an S3 bucket (Fig. 3), Amazon's cloud object storage system [54]. The gradients computed by our workflow are one-dimensional numpy arrays of the size of the vectorized seismic image and are stored in S3 as so-called objects [55]. Once an individual gradient $\mathbf{g}_i$ has been computed, the underlying EC2 instance is shut down automatically by AWS Batch, thus preventing EC2 instances from idling. Since no communication between jobs is possible, the summation of the individual gradients is implemented separately using AWS Lambda functions. For this purpose, each jobs also sends its S3 object identifier to a message queue (SQS) [56], which automatically invokes the reduction stage (Fig. 4). For the gradient computations, each worker has to download the observed seismic data of its respective source index from S3 and the resulting gradient has to be uploaded to S3 as well. The bandwidth with which objects are up- and downloaded is only limited by the network bandwidth of the EC2 instances and ranges from 10 to 100 Gbps [17]. Notably, cloud object storage such as S3 has no limit regarding the number of workers that can simultaneously read and write objects, as data is (redundantly) distributed among physically separated data centers, thus providing essentially unlimited IO scalability [54].

AWS Batch runs jobs from its queue as separate containers on a set of EC2 instances, so the source code of the application has to be prepared as a Docker container. Containerization facilitates portability and has the advantage that users have full control over managing dependencies and packages. Our Docker image contains the code for solving acoustic wave equations to compute gradients of a respective seismic source location. Since this is the most computational intensive part of our workflow, it is important that the wave equation solver is optimized for performance, but is also implemented in a programming language that allows interfacing other AWS services such as S3 or SQS. In our workflow, we use a domain-specific language compiler called Devito for implementing and solving the underlying wave equations using time-domain finite-difference modeling [38], [57]. Devito is implemented in Python and provides an application programming interface (API) for implementing forward and adjoint wave equations as high-level symbolic expressions based on the SymPy package [58].

During runtime, the Devito compiler applies a series of performance optimizations to the symbolic operators, such as reductions of the operation count, loop transformations, and introduction of parallelism [57]. Devito then generates optimized finite-difference stencil code in C from the symbolic Python expressions and dynamically compiles and runs it. Devito supports both multi-threading using OpenMP, as well as generating code for MPI-based domain decomposition. Its high-level API allows expressing wave equations of arbitrary stencil orders or various physical representations without having to implement and optimize low-level stencil codes by hand. The complexity of implementing highly optimized and parallel wave equation solvers is therefore abstracted and vertically integrated into the AWS workflow.

By default, AWS Batch runs the container of each job on a single EC2 instance, but recently AWS introduced the possibility to run multi-node batch computing jobs [59]. Thus, individual jobs from the queue can be computed on a cluster of EC2 instances and the corresponding Docker containers can communicate via the AWS network. In the context of seismic imaging and inversion, multi-node batch jobs enable nested levels of parallelization, as we can use AWS Batch to parallelize the sum of the source indices, while using MPI-based domain decomposition and/or multi-threading for solving the underlying wave equations. This provides a large amount of flexibility in regard of the computational strategy for performing backpropagation and how to address the storage of the state variables. AWS Batch allows to scale horizontally, by increasing the number of EC2 instances of multi-node jobs, but also enables vertical scaling by adding additional cores and/or memory to single instances. In our performance analysis, we compare and evaluate different strategies for computing gradients with Devito regarding scaling, costs and turnaround time.

3.3 Gradient Reduction

Every computed gradient is written by its respective container to an S3 bucket, as no communication between individual jobs is possible. Even if all gradients in the job queue are computed by AWS Batch in parallel at the same time, we found that the computation time of individual gradients typically varies considerably (up to 10 percent), due to varying network performance or instance capacity. Furthermore, we found that the startup time of the underlying EC2 instances itself is highly variable as well, so jobs in the queue are usually not all started at the same time. Gradients therefore arrive in the bucket over a large time interval during the batch job. For the gradient reduction step, i.e., the summation of all gradients into a single array, we take advantage of the varying time-to-solutions by implementing an event-driven gradient summation using Lambda functions. In this approach, the gradient summation is not performed by as single worker or the master process who has to wait until all gradients have been computed, but instead summations are carried out by Lambda functions in response to gradients being written to S3.

The event-driven gradient summation is automatically invoked through SQS messages, which are sent by the AWS Batch workers that have completed their computations and have saved their respective gradient to S3. Before being shut down, every batch worker sends a message with the corresponding S3 object name to an AWS SQS queue, in which all object names are collected (Fig. 4). Sending messages to SQS invokes AWS Lambda functions that read up to 10 messages at a time from the queue. Every invoked Lambda function that contains at least two messages, i.e., two object names, reads the corresponding arrays from S3, sums them into a single array, and writes the array as a new object back to S3. The new object name is sent to the SQS queue, while the previous objects and objects names are removed from the queue and S3. The process is repeated recursively until all n_b gradients have been summed into a single array, with n_b being the batch size for which the gradient is computed.

Since Lambda functions are limited to 3 GB of memory, it is not always possible to read the full gradient objects from S3. Gradients that exceed Lambda's available memory are therefore streamed from S3 using appropriate buffer sizes and are re-uploaded to S3 using the `multipart_upload` functions of the S3 Python interface [60]. As the execution time of Lambda functions is furthermore limited to 15 minutes, the bandwidth of S3 is not sufficient to stream and re-upload objects that exceed a certain size within a single Lambda invocation. For this case, we include the possibility that the workers of the AWS Batch job split the computed gradients into smaller chunks that are saved separately in S3, with the respective objects names being sent to multiple SQS queues. The gradient summation is then performed in chunks by separate queues and Lambda functions. The `CreateQueues` task of our Step Functions workflow (Fig. 2) automatically creates the required number of queues before starting the optimization loop and the `CleanUp` state removes them after the final iteration.

The advantage of the event-based gradient reduction is that that the summation is executed asynchronously, as soon as at least two S3 objects are available, while other batch jobs are still running. Therefore, by the time the last batch worker finishes the computation of its respective gradient, all remaining gradients have already been summed into a single object, or at least a small number of objects. Furthermore, summing files of a single queue happens in parallel (if enough messages are in the queue), as multiple Lambda functions can be invoked at the same time. Furthermore, splitting the gradients itself into chunks that are processed by separate queues leads to an additional layer of parallelism. In comparison to a fixed cluster of EC2 instances, the event-driven gradient summation using Lambda function also takes advantage of the fact that the summation of arrays is computationally considerably cheaper than solving wave equations and therefore does not require to be carried out on the expensive EC2 instances used for the PDE solves.

3.4 Variable Update

Once the gradients have been computed and summed into a single array that is stored as an S3 object, the gradient is used to update the optimization variables of equation 1, i.e., the seismic image or subsurface parameters such as velocity. Depending on the specific objective function and optimization algorithm, this can range from simple operations like multiplications with a scalars (gradient descent) to more computational expensive operations such as sparsity promotion or applying constraints [61]. Updates that use entry-wise operations only and are cheap to compute such as multiplications with scalars or soft-thresholding, can be applied directly by Lambda functions in the final step of the gradient summation. I.e., the Lambda function that sums the final two gradients, also streams the optimization variable of the current iteration from S3, uses the gradient to update it and directly writes the updated variable back to S3.

Many algorithms require access to the full optimization variable and gradient, such as Quasi-Newton methods and other algorithms that need to compute gradient norms. In this case, the variable update is too expensive and memory intensive to be carried out by Lambda functions and has to be submitted to AWS Batch as a single job, which is then executed on a larger EC2 instance. This can be accomplished by adding an extra state such as `UpdateVariable` to our Step Functions workflow. However, to keep matters simple, we only consider a simple stochastic gradient descent example with a fixed step size in our performance analysis, which is computed by the Lambda functions after summing the final two gradients [62]. The `CheckUpdateStatus` state of our AWS Step Functions advances the workflow to the next iteration, once the updated image (or summed gradient) has been written to S3. The workflow shown in Fig. 2 terminates the optimization loop after a predefined number of iterations (i.e., epochs), but other termination criteria based on gradient norms or function values are possible too. The update of the optimization variable concludes a single iteration of our workflow, whose performance we will now analyze in the subsequent sections.

4 Performance Analysis

In our performance analysis, we are interested in the performance of our workflow on a real-world seismic imaging application regarding scalability, cost and turn-around time, as well as the computational benefits and overhead introduced by our event-driven approach. We conduct our analysis on a popular 2D subsurface velocity model (Fig. 5a), called the 2004 BP velocity estimation benchmark model [28]. The seismic data set of this model contains 1,348 seismic source locations and corresponding observations $\mathbf{d}_i$ $(i = 1, \dots, 1,348)$. The (unknown) seismic image (Fig. 5b) has dimensions of $1,911 \times 10,789$ grid points, i.e., a total of almost 21 million parameters.

4.1 Weak Scaling

In our first performance test, we analyze the weak scaling behavior of our workflow by varying the batch size (i.e., the number of source locations) for which the gradient of the LS-RTM objective function (Equation (1)) is computed. For this test, we perform a single iteration of stochastic gradient descent (SGD) using our workflow and measure the time-to-solution as a function of the batch size. The workload per instance, i.e., per parallel worker, is fixed to one gradient. The total workload for a specified batch size is submitted to AWS Batch as a so-called *array job*, where each array entry corresponds to a single gradient $\mathbf{g}_i$. AWS Batch launches one EC2 instance per array entry (i.e., per gradient), runs

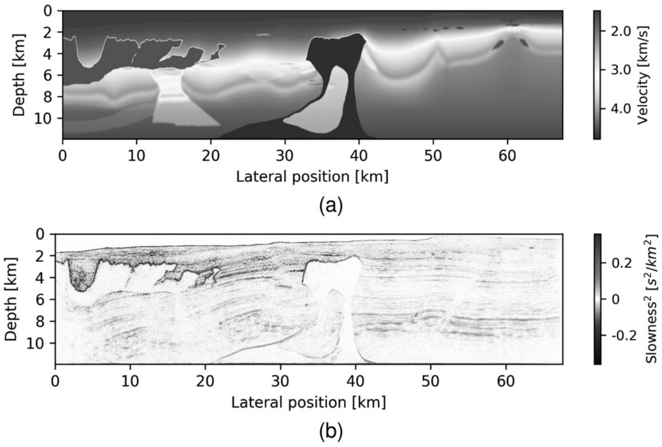


Fig. 5. The BP 2004 benchmark model, a 2D subsurface velocity model for development and testing of algorithms for seismic imaging and parameter estimation [28]. The velocity model and the unknown image have dimensions of $1,911 \times 10,789$ grid points, a total of 20.1 million unknown parameters (a). The inverted seismic image after 30 iterations of stochastic gradient descent and a batchsize of 80 sources per iteration, using our serverless workflow.

the respective container on the instance and then terminates the instance.

In the experiment, we measure the time-to-solution for performing a single iteration of our workflow, i.e., one stochastic gradient descent update. Therefore, each run involves the following steps:

1) A Lambda function submits the AWS Batch job for specified batch size n_b (Fig. 3)
2) Compute gradients $\mathbf{g}_i$ $(i = 1, \dots, n_b)$ in parallel (Fig. 3)
3) Lambda functions sum the gradients (Fig. 4): $\mathbf{g} = \sum_{i=1}^{n_b} \mathbf{g}_i$
4) A Lambda function performs the SGD update of the image: $\mathbf{x} = \mathbf{x} - \alpha \mathbf{g}$

We define the time-to-solution as the the time interval between the submission of the AWS Batch job by a Lambda function (step 1) and the time stamp of the S3 object containing the updated image (step 4). This time interval represents a complete iteration of our workflow.

The computations of the gradients are performed on `m4.4xlarge` instances (Appendix B, available in the online supplemental material) and the number of threads per instance is fixed to 8, which is the number of physical cores that is available on the instance. The `m4` instance is a general purpose EC2 instance and we chose the instance size (`4xlarge`) such that we are able to store the wavefields for backpropagation in memory. The workload for each batch worker consists of solving a forward wave equation to model the predicted seismic data and an adjoint wave equation to backpropagate the data residual and to compute the gradient. For this and all remaining experiments, we use the acoustic isotropic wave equation with a second order finite difference (FD) discretization in time and 8th order in space. We model wave propagation for 12 seconds, which is the recording length of the seismic data. The time stepping interval is given by the Courant-Friedrichs-Lewy condition with 0.55 ms, resulting in 21,889 time steps. Since it is not possible for the waves to propagate through the whole

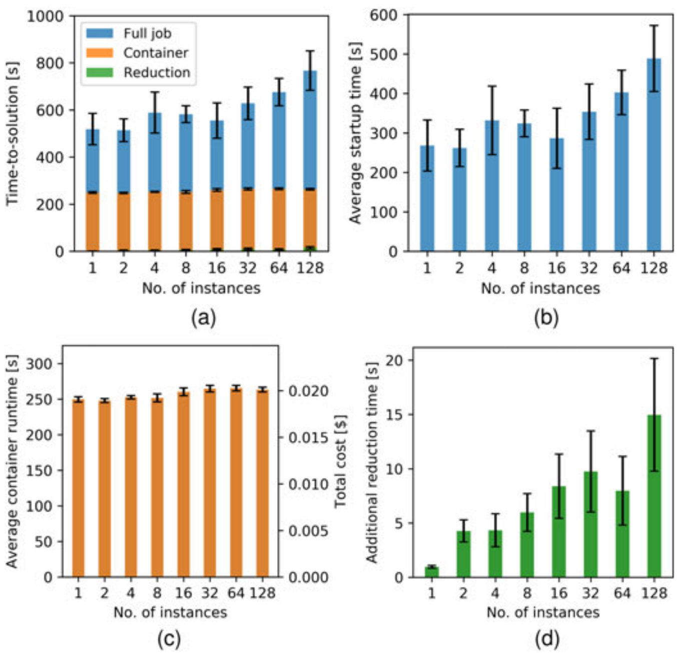


Fig. 6. Weak scaling results for performing a single iteration of stochastic gradient as a function of the batch size for which the gradient is computed (a). The gradient is computed as an AWS Batch job with an increasing number of parallel EC2 instances, while the gradient summation and the variable update are performed by Lambda functions. The total time-to-solution (a) consists of the average time it takes AWS Batch to request and start the EC2 instances (b), the average runtime of the containers (c) and the additional reduction time (d), i.e., the time difference between the final gradient of the respective batch and the updated image. All timings are the arithmetic mean over ten runs, with error bars representing the 90 percent confidence interval.

domain within this time interval, we restrict the modeling grid to a size of $1,911 \times 4,001$ grid points around the current source location. After modeling, each gradient is extended back to the full model size ($1,911 \times 10,789$ grid points). The dimensions of this example represent a large-scale 2D example, but all components of our workflow are agnostic to the number of physical dimensions and are implemented for three-dimensional domains as well. The decision to limit the examples to a 2D model was purely made from a financial viewpoint and to make the results reproducible in a reasonable amount of time.

The timings ranging from a batch size of 1 to 128 are displayed in Fig. 6a. The batch size corresponds to the number of parallel EC2 instances on which the jobs are executed. The time-to-solution consists of three components that make up the full runtime of each job:

1) The average time for AWS Batch to request and launch the EC2 instances and to start the Docker containers on those instances.
2) The runtime of the containers
3) The additional gradient reduction and image update time, given by the time interval between the termination of the AWS Batch job and the time stamp of the updated image.

The sum of these components makes up the time-to-solution as shown in Fig. 6a and each component is furthermore plotted separately in Figs. 6b, 6c, and 6d. All timings are the arithmetic mean over 10 individual runs and error bars represent the 90 percent confidence interval. The container runtimes of Fig. 6c are the arithmetic mean of the individual container runtimes on each instance (varying from 1 to 128). The average container runtime is proportional to the cost of computing one individual gradient and is given by the container runtime times the price of the `m4.4xlarge` instance, which was \$0.2748 per hour at the time of testing. No extra charges occurs for AWS Batch itself, i.e., for scheduling and launching the batch job.

The timings indicate that the time-to-solution generally grows as the batch size, and therefore the number of containers per job, increases (Fig. 6a). A close up inspection of the individual components that make up the total time-to-solution shows that this is mostly due to the increase of the startup time, i.e., the average time it takes AWS Batch to schedule and launch the EC2 instances for each job (Fig. 6b). We found that AWS Batch does generally not start all instances of the array job at the same time, but instead in several stages, over the course of 1 to 3 minutes. The exact startup time depends on the batch size and therefore on the number of instances that need to be launched, but also on the availability of the instance within the AWS region. The combination of these factors leads to an increase of the average startup time for an increasing batch size, but also to a large variance of the startup time between individual runs. Users have no control over the startup time, but it is important to consider that no cost is incurred during this time period, as no EC2 instances are running while the individual containers remain in the queue.

The average container runtime, i.e., the average computation time of a single gradient, is fairly stable as the batch size increases (Fig. 6c). This observation is consistent with the fact that each container of an AWS Batch array job runs as an individual Docker container and is therefore independent of the batch size. The container runtime increases only slightly for larger batch sizes and we observe a larger variance in some of the container runtimes. This variance stems from the fact that users do not have exclusive access to the EC2 instances on which the containers are deployed. Specifically, our containers run on `m4.4xlarge` instances, which have 8 cores (16 virtual CPUs) and 64 GB of memory. In practice, AWS deploys these instances on larger physical nodes and multiple EC2 instances (of various users) can run on the same node. We hypothesize that a larger batch size increases the chance of containers being deployed to a compute node that runs at full capacity, thus slightly increasing the average container runtime, as user do not have exclusive access to the full network capacity or memory bandwidth. The average container runtime also represents a lower bound on the time-to-solution (per iteration) that can be achieved by running the example as a classic (non-event driven) program on a fixed cluster. In this case there is no overhead from requesting EC2 instances, but other overhead may still occur, depending on how the source parallelization and gradient summation are implemented.

Finally, we also observe an increase in the additional gradient reduction time, i.e., the interval between the S3 timestamps of the final computed gradient $\mathbf{g}_i$ and the updated image $\mathbf{x}$. The batch size corresponds to the number of gradients that have to be summed before the gradient can be used to update the image. The event-driven gradient reduction invokes the summation process as soon as the first gradients are written to S3, so most gradients are already summed by the time the final worker finishes its gradient computation. For the event-driven gradient summation, the variance of the startup and container runtime is therefore advantageous, as it allows the summation to happen asynchronously. However, in our example, the time interval between the first two gradients being written to S3 (thus invoking the gradient reduction) and the final gradient being computed does not appear to be large enough to complete the summation of all gradients. Specifically, we see a general increase in the reduction time, as well as widening of the confidence interval. This variance is due to a non-deterministic component of our event-based gradient summation, resulting from a limitation of AWS Lambda. While users can specify a maximum number of messages that Lambda functions read from an SQS queue, it is not possible to force Lambda to read a minimum amount of two messages, resulting in most Lambda functions reading only a single message (i.e., one object name) from the queue. Since we need at least two messages to sum the corresponding gradients, we return the message to the queue and wait for a Lambda invocation with more than one message. The user has no control over this process and sometimes it takes several attempts until a Lambda function with multiple messages is invoked. The likelihood of this happening increases with a growing batch size, since a larger number of gradients need to be summed, which explains the increase of the reduction time and variance in Fig. 6d.

Overall, the gradient summation and variable update finish within a few seconds after the last gradient is computed and the additional reduction time is small compared to the full time-to-solution. In our example, the startup time (Fig. 6b) takes up the majority of the time-to-solution (Fig. 6a), as it lies in the range of a few minutes and is in fact longer than the average container runtime of each worker (Fig. 6c). However, the startup time is independent of the runtime of the containers, so the ratio of the startup time to the container runtime improves as the workload per container increases. For 3D imaging workloads, whose solution times for 3D wave equations are orders of magnitude higher than for two dimensions, it is therefore to be expected that the ratio between startup and computation time will shift considerably towards the latter. Indeed, in a follow-up application of our workflow to a 3D seismic data set on Microsoft Azure, the average container runtime to compute a gradient was 120 minutes, thus shifting the startup to computation time ratio to $1:25$ [63].

The cost of the batch job only depends on the container runtime and the batch size, but not on the startup time or reduction time. The cost for summing the gradients is given by the cumulative runtime of the Lambda functions, but is negligible compared to the EC2 cost for computing the gradients. This is illustrated in Fig. 7a, which shows a cost breakdown of running our workflow for 30 iterations of stochastic gradient descent with a batch size of of 80, which corresponds to 1.8 epochs. The corresponding data misfit as a function of the iteration number is shown in Fig. 7b. Similarly, SQS, Step Functions and S3 (i.e., the cost for storage and I/O) only contribute marginally to the full cost of running the imaging example, while the EC2 instances used by

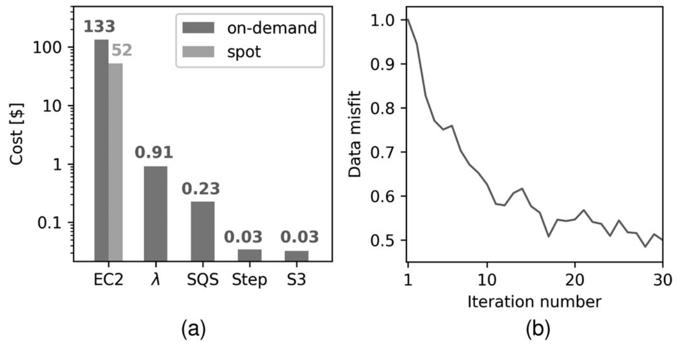


Fig. 7. Cost breakdown of running the imaging example for 30 iterations of stochastic gradient descent and a batch size of 80, which corresponds to approximately two passes through the data set (a). The data misfit as a function of the iteration number. Due to the fact that the underlying linear system is inconsistent and we stop iterating after two epochs only, the misfit only decays by about 50 percent.

AWS Batch contribute by far the largest share. Using spot instances as opposed to on-demand instances for AWS Batch reduces the cost of our example by a factor of 2.5, but the prices of the remaining services are fixed. The seismic image after the final iteration number is shown in Fig. 5b. In this example, every gradient was computed by AWS Batch on a single instance and a fixed number of threads, but in the subsequent section we analyze the scaling of runtime and cost as a function of the number of cores and EC2 instances. Furthermore, we will analyze in a subsequent example how the cost of running the gradient computations with AWS Batch compares to performing those computations on a fixed cluster of EC2 instances.

4.2 Strong Scaling

In the following set of experiments, we analyze the strong scaling behavior of our workflow for an individual gradient calculation, i.e., a gradient for a batch size of 1. For this, we consider a single gradient computation using AWS Batch and measure the runtime as a function of either the number of threads or the number of instances in the context of MPI-based domain decomposition. In the first experiment, we evaluate the vertical scaling behavior, i.e., we run the gradient computation on a single instance and vary the number of OpenMP threads. In contrast to the weak scaling experiment, we model wave propagation in the full domain ($1,911 \times 10,789$ grid points), to ensure that the sub-domain of each worker is not too small when we use maximum number of threads.

Since AWS Batch runs all jobs as Docker containers, we compare the runtimes with AWS Batch to running our application on a bare metal instance, in which case we have direct access to the compute node and run our code without any virtualization. All timings on AWS are performed on a `r5.24xlarge` EC2 instance, which is a memory optimized instance type that uses the Intel Xeon Platinum 8175M architecture (Appendix B, available in the online supplemental material). The `24xlarge` instance has 96 virtual CPU cores (48 physical cores on 2 sockets) and 768 GB of memory. Using the largest possible instance of the `r5` class, ensures that our AWS Batch job has exclusive access to the physical compute node, wile bare metal instances automatically give users exclusive access. We also include the Optimum HPC

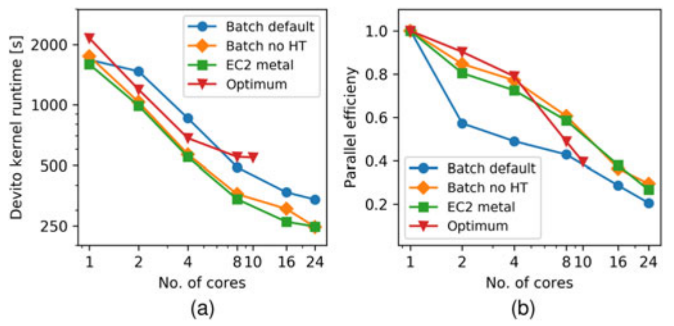


Fig. 8. Strong scaling results as a function of the number of cores on a single socket. Figure (a) shows the runtimes for AWS Batch with and without hyperthreading, as well as the runtimes on the r5 bare metal instance, in which case no containerization or virtualization is used. For reference, we also provide the runtime on a compute node of an on-premise cluster. Figure (b) shows the corresponding parallel efficiency.

cluster in our comparison, a small research cluster at the University of British Columbia based on the Intel's Ivy Bridge 2.8 GHz E5-2680v2 processor. Optimum has 2 CPUs per node and 10 cores per CPU. However, all OpenMP timings were conducted on single CPUs only, i.e., the maximum number of threads on each architecture corresponds to the maximum number of available cores per CPU (10 on Optimum and 24 on the `r5.24xlarge` instances).

Fig. 8a shows the comparison of the kernel runtimes on AWS and Optimum and Fig. 8b displays the corresponding parallel efficiency. As expected, the `r5` bare metal instance shows the best scaling, as it uses a newer architecture than Optimum and does not suffer from the virtualization overhead of Docker. We noticed that AWS Batch in its default mode uses hyperthreading (HT), even if we perform thread pinning and instruct AWS Batch to use separate physical cores. As of now, the only way to prevent AWS Batch from performing HT, is to modify the Amazon Machine Image (AMI) of the corresponding AWS compute environment. With HT disabled, the runtimes and speedups of AWS Batch are very close to the timings on the bare-metal instances, indicating that the overhead of Docker affects the runtimes and scaling of our memory-intensive application only marginally, which matches the findings of [64].

Next, we analyze the horizontal strong scaling behavior of running our application with AWS Batch. Once again, we consider the computation of one single gradient, but this time we vary the number of EC2 instances. We would like to emphasize that AWS Batch is used differently than in the weak scaling experiment, where AWS Batch was used to parallelize the sum over source locations and communication between workers of a separate jobs was not possible. Here, we submit a single workload (i.e., one gradient) as a multi-node AWS Batch job, in which case IP-based communication between instances is enabled. Since this involves distributed memory parallelism, we use domain decomposition based on message passing (MPI) to solve the wave equations on multiple EC2 instances [65], [66]. The code with the corresponding MPI communication statements is automatically generated by the Devito compiler. Furthermore, we use multi-threading on each individual instance and utilize the maximum number of available cores per socket, which is 24 for the `r5` instance and 18 for the `c5n` instance. To investigate the possible virtualization overhead

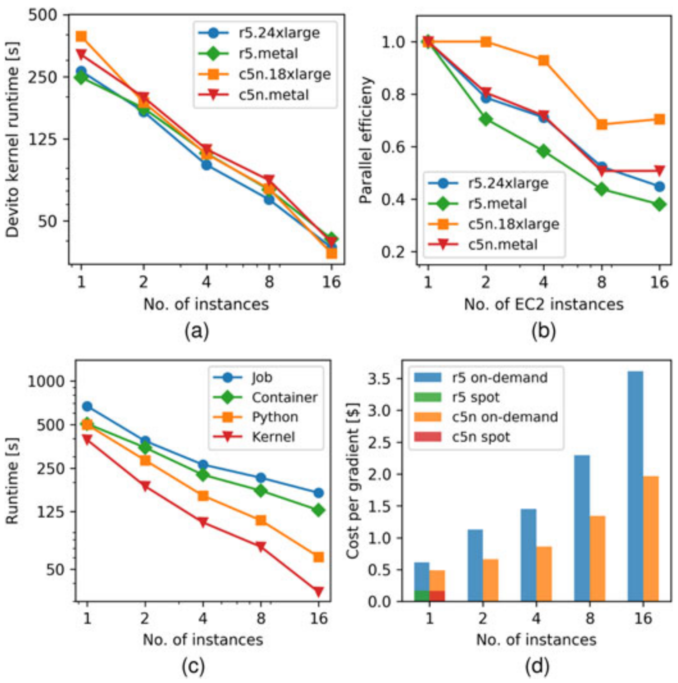


Fig. 9. Strong scaling results for computing a single gradient as an AWS Batch multi-node job for an increasing number of instances and in comparison to running on non-virtualized bare metal instances. Figures (a) and (b) show the Devito kernel times and parallel efficiency on two different instance types. Figure (c) shows a breakdown of the time-to-solution of each batch job into its individual components. Figure (d) shows the EC2 cost for computing the gradients.

of AWS Batch and Docker, we also conduct the same scaling experiments on clusters of EC2 bare metal instances, in which case our applications runs directly on the compute nodes without any form of virtualization.

We compare the `r5.24xlarge` (and `r5.metal`) instances from the last section with Amazon's recently introduced `c5n` HPC instance. Communication between AWS instances is generally based on ethernet and the `r5` instances have up to 25 GBps networking performance. The `c5n` instance type uses Intel Xeon Platinum 8142M processors with up to 3.4 GHz architecture and according to AWS provides up to 100 GBps of network bandwidth. The network is based on a proprietary AWS technology called elastic fabric adapter (EFA), but AWS has not disclosed whether this technology is based on InfiniBand or Ethernet [13]. Figs. 9a and 9b show the kernel runtimes and the corresponding parallel efficiency ranging from 1 to 16 instances for AWS Batch and on bare metal instances. The `r5` instance has overall shorter runtimes than the `c5n` instance, since the former has 24 physical cores per CPU socket, while the `c5n` instance has 18. However, as expected, the `c5n` instance exhibits a better parallel efficiency than the `r5` instance (in both batch mode and on bare metal), due to the better network performance. Interestingly, speedups and parallel efficiency of multi-node AWS Batch jobs are better than of the corresponding bare metal jobs, which is counter intuitive. To investigate this further, we measured the latency between two manually requested bare metal instances and two compute nodes assigned by AWS Batch and found that the latter set of instances have less than half the amount of latency (Appendix C, available in the online supplemental material). All nodes were requested in the same availability zone and EC2 placement group. This indicates that AWS possibly places instances for AWS Batch jobs closer to each other than manually requested EC2 instances, or that AWS Batch instances use a more efficient network gateway, but both of these explanations are purely speculative. Overall, the observed scaling and parallel efficiency of the AWS Batch jobs on both instances types are in the expected range of performance, as our application represents a strongly memory bound workload with an operational intensity of three FLOPs/Byte only, as shown in a roofline analysis of Devito's generated code for acoustic modeling [38].

The timings given in Fig. 9a are once again the pure kernel times for solving the PDEs, but a breakdown of the components that make up the total time-to-solution on the `c5n` instance is provided in Fig. 9c. The job runtime is defined as the interval between the job creation time and the S3 time stamp of the computed gradient. As in our weak scaling test, this includes the time for AWS Batch to request and launch the EC2 instances, but excludes the gradient summation time, since we are only considering the computation of a single gradient. The container runtime is the runtime of the Docker container on the master node and includes the time it takes AWS Batch to launch the remaining workers and to establish an `ssh` connection between all instances/containers, which was the only supported communication protocol for AWS Batch at the time [66]. Currently, AWS Batch requires this process to be managed by the user using a shell script that is run inside each container. The Python runtime in Fig. 9c is defined as the runtime of our application on the main node and includes IO, memory allocation and code generation time. Our timings in Fig. 9c show that the overhead from requesting instances and establishing a cluster, i.e., the difference between the Python and container runtime, is reasonable for a small number of instances, but grows significantly as the number instances is increased. Depending on the runtime of the application, the overhead thus takes up a significant amount of the time-to-solution, but for compute-heavy applications which run for one or multiple hours, this amount of overhead may still be acceptable. For 3D imaging applications that run for multiple hours, it is therefore to be expected that the ratio of application runtime to overhead will improve significantly.

Fig. 9d shows the cost for running our scaling test as a function of the cluster size. The cost is calculated as the instance price (per second) times the runtime of the container on the main node times the number of instances. The cost per gradient grows significantly with the number of instances, as the overhead from establishing an `ssh` connection to all workers increases with the cluster size. The communication overhead during domain decomposition adds an additional layer of overhead that further increases the cost for an increasing number of instances. This is an important consideration for HPC in the cloud, as the shortest time-to-solution does not necessarily correspond to the cheapest approach. Another important aspect is that AWS Batch multi-node jobs do not support spot instances [66]. Spot instances allow users to access unused EC2 capacities at significantly lower price than at the on-demand price, but AWS Batch multi-node jobs are, for the time being, only supported with on-demand instances.

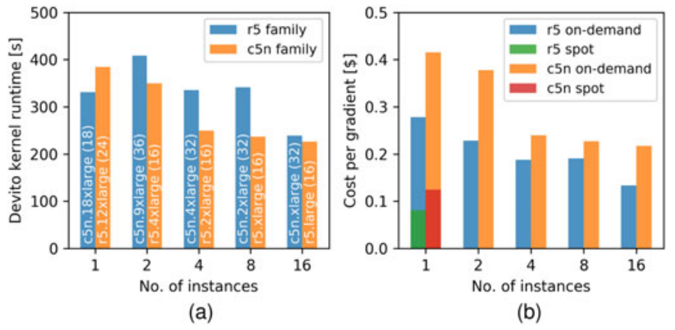


Fig. 10. Devito kernel runtimes for computing a single gradient as an AWS Batch job for an increasing number of instances. In comparison to the previous example in which both the instance type and number of threads were fixed (Fig. 10a), we use the smallest possible instance type for each job with as specified in each bar. Figure (b) shows the corresponding cost for computing the gradients.

The scaling and cost analysis in Figs. 9a, 9b, 9c and 9d was carried out on the largest instances of the respective instance types (`r5.24xlarge` and `c5n.18xlarge`) to guarantee exclusive access to the compute nodes and network bandwidth. Increasing the number of instances per run therefore not only increases the total number of available cores, but also the amount of memory. However, for computing a single gradient, the required amount of memory is fixed, so increasing the number of instances reduces the required amount of memory per instance, as wavefields are distributed among more workers. In practice, it therefore makes sense to chose the instance type based on the required amount of memory per worker, as memory is generally more expensive than compute. In our specific case, computing the gradient requires 170 GB of memory, which requires either a single `r5.12xlarge` instance or multiple smaller instances, which not only differ in the amount of memory, but also in the number of available CPU cores. We repeat our previous scaling test, but rather than using the same instance type in all runs, we choose the instance type based on the required amount of memory. Furthermore, for every instance type, we utilize the maximum amount of available cores using multi-threading with OpenMP. The kernel runtimes for an increasing number of instances is shown in Fig. 10a. In each bar, we indicate which instance type was used, as well as the total number of cores across all instances. The corresponding costs for computing each gradient is shown in Fig. 10a. Compared to the previous example, we observe that using 16 small on-demand instances leads to a lower cost than using a single more expensive large instance, but that using a single instance ultimately remains the most cost-effective way of computing a gradient, due to the possibility to utilize spot instances. An additional pricing model offered by AWS are reserved instances, which are EC2 instances that can be purchased at rates similar to spot prices if reserved for a minimum period of one year [67]. However, this pricing model is unsuitable for our event-driven approach, as users cannot save costs by minimizing idle time, since instances are paid for in advance.

In terms of cost, our scaling examples underline the importance of choosing the EC2 instances for the AWS Batch jobs based on the total amount of required memory, rather than based on the amount of CPU cores. Scaling horizontally by using an increasingly large number of instances

TABLE 1
Comparison of Parallelization Strategies on a Single EC2 Instance in the Context of AWS Batch

Grid	CPU (cores)	Parallelization	Runtime [s]
$1,911 \times 5,394$	1 (24)	OMP	190.17 ± 7.12
$1,911 \times 10,789$	1 (24)	OMP	378.94 ± 13.57
$1,911 \times 10,789$	2 (48)	OMP	315.92 ± 16.50
$1,911 \times 10,789$	2 (48)	OMP + MPI	249.13 ± 5.22

The Timings are the Devito Kernel Times for Computing a Single Gradient of the BP Model Using AWS Batch. The Program Runs as a Single Docker Container on an Individual EC2 Instance, Using Either Multi-Threading (OpenMP) or a Hybrid Approach of Multithreading and Domain-Decomposition (OpenMP + MPI).

expectedly leads to a faster time-to-solution, but results in a significant increase of cost as well (Fig. 9d). As shown in Fig. 10b, this increase in cost can be avoided to some extent by choosing the instance size such that the total amount of memory stays approximately constant, but ultimately the restriction of not supporting spot instances makes multi-node batch jobs not attractive in scenarios where single instances provide sufficient memory to run a given application. In practice, it makes therefore sense to use single node/instance batch jobs and to utilize the full number of available cores on each instance. The largest EC2 instances of each type (e.g., `r5.24xlarge`, `c5n.18xlarge`) have two CPU sockets with shared memory, making it possible to run parallel programs using either pure multi-threading or a hybrid MPI-OpenMP approach. In the latter case, programs still run as a single Docker container, but within each container use MPI for communication between CPU sockets, while OpenMP is used for multithreading on each CPU. For our example, we found that computing a single gradient of the BP model with the hybrid MPI-OpenMP approach leads to a 20 percent speedup over the pure OpenMP program (Table 1), which correspondingly leads to 20 percent cost savings as well.

4.3 Cost Comparison

One of the most important considerations of high performance computing in the cloud is the aspect of cost. As users are billed for running EC2 instances by the second, it is important to use instances efficiently and to avoid idle resources. In our specific application, gradients for different seismic source locations are computed by a pool of parallel workers, but as discussed earlier, computations do not necessarily complete at the same time. On a conventional cluster, programs with a MapReduce structure, are implemented based on a client-server model, in which the workers compute the gradients in parallel, while the master (the server) collects and sums the results. This means that the process has to wait until all gradients $\mathbf{g}_i$ have been computed, before the gradient can be summed and used to update the image. This inevitably causes workers that finish their computations earlier than others to the sit idle. This is problematic when using a cluster of EC2 instances, where the number of instances are fixed, as users have to pay for idle resources. In contrast, the event-driven approach based on Lambda functions and AWS Batch automatically terminates EC2 instances of workers that

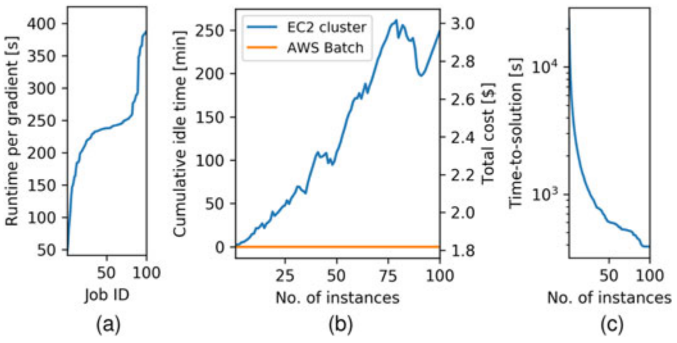


Fig. 11. (a) Sorted container runtimes of an AWS Batch job in which we compute the gradient of the BP model for a batch size of 100. Figure (b) shows the cumulative idle time for computing this workload as a function of the number of parallel workers on either a fixed cluster of EC2 instances or using AWS Batch. The right-hand y-axis shows the corresponding cost, which is proportional to the idle time. In the optimal case, i.e., no instances every sit idle, the cost for computing a gradient of batch size 100 is 1.8$. Figure (c) shows the time-to-solution as a function of the number of parallel instances, which is the same on an EC2 cluster and for AWS Batch, if we ignore the startup time of the AWS Batch workers or of the corresponding EC2 cluster.

have completed their gradient calculation, thus preventing resources from sitting idle.

We illustrate the difference between the event-driven approach and using a fixed cluster of EC2 instances by means of a specific example. We consider our previous example of the BP synthetic model and compute the gradient $\mathbf{g}_i$ for 100 random source locations and record the runtimes (Fig. 11a). We note that most gradients take around 250 seconds to compute, but that the runtimes vary due to different domain sizes and varying EC2 capacity. We now model the idle time for computing these gradients on a cluster of EC2 instances as a function of the the number of parallel instances, ranging from 1 instance (fully serial) to 100 instances (fully parallel). For a cluster consisting of a single instance, the cumulative idle time is naturally zero, as the full workload is executed in serial. For more than one instance, we model the amount of time that each instance is utilized, assuming that the workloads are assigned on a first-come-first-served basis. The cumulative idle time t_{idle} is then given as the sum of the differences between the runtime of each individual instance t_i and the instance with the longest runtime:

$$t_{\text{idle}} = \sum_{i=1}^{n_{\text{EC2}}} (\max\{t_i\} - t_i), \tag{3}$$

The cumulative idle time as a function of the cluster size n_{EC2} is plotted in Fig. 11b. We note that the cumulative idle time generally increases with the cluster size, as a larger number of instances sit idle while waiting for the final gradient to be computed. On a cluster with 100 instances each gradient is computed by a separate instance, but all workers have to wait until the last worker finishes its computation (after approximately 387 seconds). In this case, the varying time-to-solutions of the individual gradients leads to a cumulative idle time of 248 minutes. Compared to the cumulative computation time of all gradients, which is 397 minutes, this introduces an overhead of more than 60 percent, if the gradients are computed on a cluster with

100 instances. The cumulative idle time is directly proportional to the cost for computing the 100 gradients, which is plotted on the right axis of Fig. 11b. With AWS Batch, the cumulative idle time for computing the 100 gradients is zero, regardless of the number of parallel instances that AWS Batch has access to. Any EC2 instance that is not utilized anymore is automatically shut down by AWS Batch, so no additional cost other than the pure computation time of the gradients is invoked [68]. A follow up case study of this work, found that the ratio of cumulative idle time to cumulative compute time (about 60 percent) also held true for a 3D seismic imaging example, in which the average container runtime was substanially longer than in the 2D example (namely 120 minutes) [63].

While computing the 100 gradients on an EC2 cluster with a small number of instances results in little cumulative idle time, it increases the overall time-to-solution, as a larger number of gradients have to be sequentially computed on each instance (Fig. 11c). With AWS Batch this trade-off does not exist, as the cumulative idle time, and therefore the cost for computing a fixed workload, does not depend on the number of instances. However, it is to be expected that in practice the time-to-solution is somewhat larger for AWS Batch than for a fixed cluster of EC2 instances, as AWS Batch needs to request and launch EC2 instances for every new gradient computation.

4.4 Resilience

In the final experiment of our performance analysis, we analyze the resilience of our workflow and draw a comparison to running an MPI program on a conventional cluster of EC2 instances. Resilience is an important factor in high performance computing, especially for applications like seismic imaging, whose runtime can range from several hours to multiple days. In the cloud, the mean-time-between failures is typically much shorter than on comparable HPC systems [6], making resilience potentially a prohibiting factor. Furthermore, using spot instances further increases the exposure to instance shut downs, as spot instances can be terminated at any point in time with a two minute warning.

Seismic imaging codes that run on conventional HPC clusters typically use MPI to parallelize the sum of the source indices and thus often leverage the User Level Fault Mitigation (ULFM) Standard [69] or the MVAPICH2 implementation of MPI, which includes built-in resilience [70]. Both MVAPICH2 and ULFM enable the continued execution of an MPI program after node/instance failures. In our event-driven approach, resilience in the cloud is naturally provided by using AWS Batch for the gradient computations, as each gradient is computed by a separate container. In case of exceptions, AWS Batch provides the possibility to automatically restart EC2 instances that have crashed, which allows the completion of programs with the initial number of nodes or EC2 instances.

We illustrate the effect of instance restarts by means of our previous example with the BP model (Fig. 5). Once again, we compute the gradient of the LS-RTM objective function for a batch size of 100 and record the runtimes without any instance/node failures. In addition to the default strategy for backpropagation, we compute the gradients using optimal checkpointing, in which case the average runtime per gradient increases from 5 minutes to 45 minutes, as a smaller memory footprint is traded for additional computations.

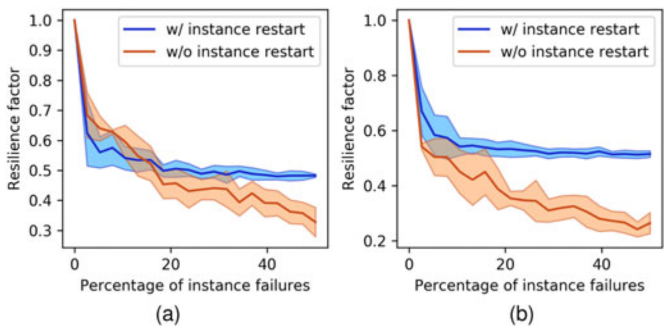


Fig. 12. Comparison of the resilience factor (RF) for an increasing percentage of instance failures with and without instance restarts. Figure (a) is the RF for an application that runs for 5 minutes without failures, while figure (b) is based on an example whose original time-to-solution is 45 minutes.

We then model the time that it takes to compute the 100 gradients for an increasing number of instance failures with and without restarts. We assume that the gradients are computed fully in parallel, i.e., on 100 parallel instances and invoke an increasing number of instance failures at randomly chosen times during the execution of program. Without instance restarts, we assign the workload of the failed instances to the workers of the remaining instances and model how long it takes complete the computations. With restarts, we add a two minute penalty to the failed worker and then restart the computation on the same instance. The two minute penalty represents the average amount of time it takes AWS Batch to restart a terminated EC2 instance and was determined experimentally.

Fig. 12 shows the ratio of the time-to-solution for computing the 100 gradients without events (i.e., without failures) to the modeled time-to-solution with events. This ratio is known as the resilience factor [71] and provides a metric of how instance failures affect the time-to-solution and therefore the cost of running a given application in the cloud:

$$r = \frac{\text{time-to-solution}_{\text{event-free}}}{\text{time-to-solution}_{\text{event}}}. \tag{4}$$

Ideally, we aim for this factor being as close to 1 as possible, meaning that instance failures do not significantly increase the time-to-solution. Figs. 12a and 12b compare the resilience factors with and without restarts for the two different backpropagation strategies, which represent programs of different runtimes. The resilience factor is plotted as a function of the percentage of instance failures and is the average of 10 realizations, with the standard deviation being depicted by the shaded colors. The plots show that the largest benefit from being able to restart instances with AWS Batch is achieved for long running applications (Fig. 12b). The resilience factor with instance restarts approaches a value of 0.5, since in the worst case, the time-to-solution is doubled if an instance fails shortly before completing its gradient computation. Without being able to restart instances, the gradient computations need to be completed by the remaining workers, so the resilience factor continuously decreases as the failure percentage increases. For short running applications (Fig. 12a), the overhead of restarting instances diminishes the advantage of instance restarts, unless a significant percentage of instances fail, which, however, is unlikely for programs that run in a matter of minutes. In fact, during our numerical experiments, we did not observed any spot-related shut-downs, as the runtime of our application was only in the order of five minutes. On the other hand, long running programs or applications with a large number of workers are much more likely to encounter instance shut downs and our experiment shows that these programs benefit from the automatic instance restarts of AWS Batch.

5 Discussion

The main advantage of an event-driven approach based on AWS Batch and Lambda functions for seismic imaging in the cloud is the automated management of computational resources by AWS. EC2 instances that are used for carrying out heavy computations, namely for solving large-scale wave equations, are started automatically in response to events, which in our case are Step Functions advancing the serverless workflow to the `ComputeGradients` state. Expensive EC2 instances are thus only active for the duration it takes to compute one element $\mathbf{g}_i$ of the full or mini-batch gradient and they are automatically terminated afterwards. Summing the gradients and updating the variables (i.e., the seismic image) is performed on cheaper Lambda functions, with billing being again solely based on the execution time of the respective code and the utilized memory. The cost overhead introduced by Step Functions, SQS messages and AWS Batch is negligible compared to the cost of the EC2 instances that are required for the gradient computations, while cost savings from spot instances and eliminating idle EC2 instances lead to significant cost savings, as shown in our examples. With the benefits of spot instances (factor 2-3), avoidance of idle instances and the overhead of spinning clusters (factor 1.5-2), as well as improved resilience, we estimate that our event-driven workflow provides cost savings of up to an order of magnitude in comparison to using fixed clusters of (on-demand) EC2 instances.

A second alternative to running cloud applications on fixed EC2 clusters are in-between approaches based on a combination of EC2 instances, task-based workflow tools and auto-scaling. These approaches potentially benefit from the possibility to avoid containerization by using bare metal instances, while leveraging automatic up- and down-scaling of EC2 instances to save cost. However, workflow tools like AWS Batch or Step Functions are currently not available for EC2 clusters and thus need to be replicated by the user. Furthermore, any cluster-based workflows, even with auto-scaling, require at least a single EC2 instance to be permanently running, while our serverless approach makes it hypothetically possible to suspend tasks or workloads indefinitely without incurring any cost at all. Regarding performance, our analysis showed that Docker containerization did not lead to considerable performance impairments, while latency between EC2 instances assigned by AWS Batch was in fact smaller than latency between user-requested instances. Using batch processing to compute an embarrassingly parallel workload is not only advantageous in the cloud, but also on on-premise HPC systems, as parallel jobs that are broken into multiple smaller and shorter jobs are oftentimes processed faster by HPC schedulers than single large workloads. In addition to the improved flexibility regarding nested parallelization, this makes tasked-based asynchronous batch processing interesting in the setting of traditional HPC as well.

Another major advantage of our proposed serverless approach is the handling of resilience. Instead of running as a single program, our workflow is broken up into its individual components and expressed through Step Function states. Parallel programs based on MPI rely on not being interrupted by hardware failures during the entire runtime of the code, making this approach susceptible to resilience issues. Breaking a seismic imaging workflow into its individual components, with each component running as an individual (sub-) program and AWS managing their interactions, makes the event-driven approach inherently more resilient, as the runtime of individual workflow components is much shorter than the runtime of the full program, thus minimizing the exposure to instance failures. Computing an embarrassingly parallel workload with AWS Batch, rather than as a MPI-program, provides a natural layer of resilience, as AWS Batch processes each item from its queue separately on an individual EC2 instance and Docker container, but also includes the possibility of automatic instance restarts in the event of failures.

The most prominent disadvantage of the event-driven workflow is that EC2 instances have to be restarted by AWS Batch in every iteration of the workflow. In our performance analysis, we found that the overhead of requesting EC2 instances and starting the Docker container lies in the range of several minutes and depends on how many instances are requested per gradient. However, items that remain momentarily in the batch queue, do not incur any cost until the respective EC2 instance is launched. The overhead introduced by AWS Batch therefore only increases the time-to-solution, but does not affect the cost negatively. Due to the overhead of starting EC2 instances for individual computations, our proposed workflow is therefore applicable if the respective computations (e.g., computing gradients) are both embarrassingly parallel and take a long time to compute; ideally in the range of hours rather than minutes. We therefore expect that the advantages of our workflow will be even more prominent when applied to 3D seismic data sets, where computations are orders of magnitude more expensive than in 2D.

Our application, as expressed through AWS Step Functions, represents the structure of a generic gradient-based optimization algorithm and is therefore applicable to problems other than seismic imaging and full-waveform inversion. The design of our workflow lends itself to problems that exhibit a certain MapReduce structure, namely they consists of a computationally expensive, but embarrassingly parallel Map part, as well as a computationally cheaper to compute Reduce part. On the other hand, applications that rely on dense communications between workers or where the quantities of interest such as gradients or functions values are cheap to compute, are less suitable for this specific design. For example, deep convolutional neural networks

TABLE 2
An Overview how the AWS Services Used in our Workflow Map to Other Cloud Providers

Amazon Web Services	Microsoft Azure	Google Cloud
Elastic Compute Cloud	Virtual Machines	Compute Engine
Simple Storage System	Blob storage	Cloud Storage
AWS Batch	Azure Batch	Pipelines
Lambda Functions	Azure Functions	Cloud Functions
Step Functions	Logic Apps	N/A
Simple Message Queue	Queue Storage	Cloud Pub/Sub
Elastic File System	Azure Files	Cloud Filestore

(CNNs) exhibit mathematically a very similar problem structure to seismic inverse problems, but forward and backward evaluations of CNNs are typically much faster than solving forward and adjoint wave equations, even if we consider very deep networks like ResNet [72]. Implementing training algorithms for CNNs as an event-driven workflow as presented here, is therefore excessive for the problem sizes that are currently encountered in deep learning, but might be justified in the future if the dimensionality of neural networks continues to grow.

The event-driven workflow presented in this work was specifically designed for AWS and takes advantage of specialized services for batch computing or event-driven computations that are available on this platform. However, in principle, it is possible to implement our workflow on other cloud platforms as well, as almost all of the utilized services have equivalent versions on Microsoft Azure or the Google Cloud Platform (Table 2) [73], [74]. Services for running parallel containerized workloads in the cloud, as well as event-driven cloud functions, which are the two main components of our workflow, are available on all platforms considered in our comparison. Furthermore, both Microsoft Azure as well as the GCP offer similar Python APIs as AWS for interfacing cloud services. We also speculate that, as cloud technology matures, services between different providers will likely grow more similar to each other. This is based on the presumption that less advanced cloud platforms will imitate services offered by major cloud providers in order to be competitive in the growing cloud market.

Overall, our workflow and performance evaluation demonstrate that cost-competitive HPC in the cloud is possible, but requires a fundamental software re-design of the corresponding application. In our case, the implementation of an event-driven seismic imaging workflow was possible, as we leverage Devito for expressing and solving the underlying wave equations, which accounts for the major workload of seismic imaging. With Devito, we are able to abstract the otherwise complex implementation and performance optimization of wave equation solvers and take advantage of recent advances in automatic code generation. As Devito generates code for solving single PDEs, with the possibility of using MPI-based domain decomposition, we are not only able to leverage AWS Batch for the parallelization over source experiments, but can also take advantage of AWS Batch's multi-node functionality to shift from data to model parallelism. In contrast, many seismic imaging codes are software monoliths, in which PDE solvers are enmeshed with IO routines, parallelization and manual performance optimization. Adapting codes of this form to the cloud is fundamentally more challenging, as it is not easily possible to isolate individual components such as a PDE solver for a single source location, while replacing the parallelization with cloud services. This illustrates that separation of concerns and abstract user interfaces are a prerequisite for porting HPC codes to the cloud such that the codes are able to take advantage of new technologies like object storage and event-driven computations. With a domain-specific language compiler, automatic code generation, high-throughput batch computing and serverless visual algorithm definitions, our workflow represents a true vertical integration of modern programming paradimgs into a framework for HPC in the cloud.

6 Conclusion

Porting HPC applications to the cloud using a lift and shift approach based on virtual clusters that emulate on-premise HPC clusters, is problematic as the cloud cannot offer the same performance and reliability as conventional clusters. Applications such as seismic imaging that are computationally expensive and run for a long time, are faced with practical challenges such as cost and resilience issues, which prohibit the cloud from being widely adapted for HPC tasks. However, the cloud offers a range of new technologies such as object storage or event-driven computations, that allow to address computational challenges in HPC in novel ways. In this work, we demonstrate how to adapt these technologies to implement a workflow for seismic imaging in the cloud that does not rely on a conventional cluster, but is instead based on serverless and event-driven computations. These tools are not only necessary to make HPC in the cloud financially viable, but also to improve the resilience of workflows. The code of our application is fully redesigned and uses a variety of AWS services as building blocks for the new workflow, thus taking advantage of AWS being responsible for resilience, job scheduling, and resource allocations. Our performance analysis shows that the resulting workflow exhibits competitive performance and scalability, but most importantly minimizes idle time on EC2 instances and cost and is inherently resilient. Our example therefore demonstrates that successfully porting HPC applications to the cloud is possible, but requires to carefully adapt the corresponding codes to to the new environment. This process is heavily dependent on the specific application and involves identifying properties of the underlying scientific problem that can be exploited by new technologies available in the cloud. Most importantly, it requires that codes are modular and designed based on the principle of separation of concerns, thus making this transition possible.

Acknowledgments

The authors would like to acknowledge Dr. Fabio Luporini and Dr. Gerard J. Gorman (Imperial College London) for their work on Devito and their valuable input on the performance evaluation. This research was funded by the Georgia Research Alliance and the Georgia Institute of Technology.

References

[1] A. A. Valenciano, "Imaging by wave-equation inversion," Ph.D. dissertation, Dept. Geophys., Stanford Univ., Stanford, CA, 2008.

[2] S. Dong *et al.*, "Least-squares reverse time migration: Towards true amplitude imaging and improving the resolution," in *Proc. 82nd Annu. Int. Meeting SEG Expanded Abstracts*, 2012, pp. 1–5.

[3] P. A. Witte, M. Louboutin, F. Luporini, G. J. Gorman, and F. J. Herrmann, "Compressive least-squares migration with on-the-fly Fourier transforms," *GEOPHYSICS*, vol. 84, no. 5, pp. R655–R672, 2019. [Online]. Available: https://doi.org/10.1190/geo2018–0490.1

[4] "Seismic processing and imaging," 2019. [Online]. Available: https://www.pgs.com/imaging/services/processing-and-imaging/

[5] AWS enterprise customer success stories, 2019. [Online]. Available: https://aws.amazon.com/solutions/case-studies/enterprise

[6] K. R. Jackson *et al.*, "Performance analysis of high performance computing applications on the Amazon Web Services cloud," in *Proc. IEEE 2nd Int. Conf. Cloud Comput. Technol. Science*, 2010, pp. 159–168.

[7] S. L. Garfinkel, "An evaluation of Amazon's grid computing services: EC2, S3, and SQS," Harvard Computer Science Group, Cambridge, MA, Tech. Rep. TR-08–07, 2007.

[8] J. Napper and P. Bientinesi, "Can cloud computing reach the Top500?" in *Proc. Combined Workshops Unconventional High Perform. Comput. Workshop Plus Memory Access Workshop*, 2009, pp. 17–20. [Online]. Available: http://doi.acm.org/10.1145/1531666.1531671

[9] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 6, pp. 931–945, Jun. 2011.

[10] K. R. Jackson, K. Muriki, L. Ramakrishnan, K. J. Runge, and R. C. Thomas, "Performance and cost analysis of the supernova factory on the Amazon AWS cloud," *Sci. Program.*, vol. 19, no. 2–3, pp. 107–119, 2011.

[11] L. Ramakrishnan, R. S. Canon, K. Muriki, I. Sakrejda, and N. J. Wright, "Evaluating interconnect and virtualization performance for high performance computing," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 2, pp. 55–60, 2012.

[12] P. Mehrotra *et al.*, "Performance evaluation of amazon elastic compute cloud for NASA high-performance computing applications," *Concurrency Comput., Practice Experience*, vol. 28, no. 4, pp. 1041–1055, 2016.

[13] AWS elastic fabric adapter, 2019. [Online]. Available: https://aws.amazon.com/hpc/efa/

[14] AWS Nitro system, 2019. [Online]. Available: https://aws.amazon.com/ec2/nitro/

[15] J. Scheuner and P. Leitner, "A cloud benchmark suite combining micro and applications benchmarks," in *Proc. Companion ACM/SPEC Int. Conf. Perform. Eng.*, 2018, Art. no. 161–166. [Online]. Available: https://doi.org/10.1145/3185768.3186286

[16] C. Kotas, T. Naughton, and N. Imam, "A comparison of Amazon Web Services and Microsoft Azure cloud platforms for high performance computing," in *Proc. IEEE Int. Conf. Consum. Electron.*, 2018, pp. 1–4.

[17] AWS documentation: Amazon EC2 instance types, 2019. [Online]. Available: https://aws.amazon.com/ec2/instance-types/

[18] A. Gupta and D. Milojicic, "Evaluation of HPC applications on cloud," in *Proc. 6th Open Cirrus Summit*, 2011, pp. 22–26.

[19] I. Sadooghi *et al.*, "Understanding the performance and potential of cloud computing for scientific applications," *IEEE Trans. Cloud Comput.*, vol. 5, no. 2, pp. 358–371, Apr. 2017.

[20] P. Rad, A. Chronopoulos, P. Lama, P. Madduri, and C. Loader, "Benchmarking bare metal cloud servers for HPC applications," in *Proc. IEEE Int. Conf. Cloud Comput. Emerg. Markets*, 2015, pp. 153–159.

[21] M. Mohammadi and T. Bazhirov, "Comparative benchmarking of cloud computing vendors with high performance LINPACK," in *Proc. 2nd Int. Conf. High Perform. Compilation Comput. Commun.*, 2018, pp. 1–5.

[22] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, Cambridge, MA, USA: MIT Press, vol. 1, 1999.

[23] AWS documentation: AWS Lambda, 2019. [Online]. Available: https://aws.amazon.com/lambda/

[24] A. Friedman and A. Pizarro, "Building high-throughput genomics batch workflows on AWS," May 2017. [Online]. Available: https://aws.amazon.com/blogs/compute/building-high-throughput-genomics-batch-workflows-on-aws-introduction-part-1-of-4/

[25] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[26] AWS documentation: AWS Batch, 2019. [Online]. Available: https://aws.amazon.com/ec2/

[27] AWS documentation: Amazon EMR, 2019. [Online]. Available: https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-overview.html

[28] F. Billette and S. Brandsberg-Dahl, "The 2004 BP velocity benchmark," in *Proc. 67th Annu. Int. Meeting EAGE Expanded Abstracts*, 2005, Art. no. B035.

[29] P. A. Witte, M. Louboutin, H. Modzelewski, C. Jones, J. Selvage, and F. J. Herrmann, "Event-driven workflows for large-scale seismic imaging in the cloud," in *Proc. 9th Annu. Int. Meeting SEG Expanded Abstracts*, 2019, pp. 3984–3988.

[30] A. Tarantola, "Inversion of seismic reflection data in the acoustic approximation," *Geophysics*, vol. 49, no. 8, 1984, Art. no. 1259. [Online]. Available: + http://dx.doi.org/10.1190/1.1441754

[31] J. Virieux and S. Operto, "An overview of full-waveform inversion in exploration geophysics," *Geophysics*, vol. 74, no. 6, pp. WCC127–WCC152, Nov./Dec. 2009.

[32] R. G. Pratt, "Seismic waveform inversion in the frequency domain, part 1: Theory and verification in a physical scale model," *Geophysics*, vol. 64, no. 3, pp. 888–901, 1999. [Online]. Available: https://doi.org/10.1190/1.1444597

[33] B. Peters, B. R. Smithyman, and F. J. Herrmann, "Projection methods and applications for seismic nonlinear inverse problems with multiple constraints," *Geophysics*, vol. 84, no. 2, pp. R251–R269, 2019. [Online]. Available: https://doi.org/10.1190/geo2018-0192.1

[34] L. Ruthotto and E. Haber, "Deep neural networks motivated by partial differential equations," *CoRR*, vol. abs/1804.04272, 2018. [Online]. Available: http://arxiv.org/abs/1804.04272

[35] A. Griewank and A. Walther, "Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation," *Assoc. Comput. Machinery Trans. Math. Softw.*, vol. 26, no. 1, pp. 19–45, Mar. 2000. [Online]. Available: http://doi.acm.org/10.1145/347837.347846

[36] C. M. Furse, "Faster than Fourier-ultra-efficient time-to-frequency domain conversions for FDTD," in *Proc. Inst. Elect. Electronics Engineers: Antennas Propag. Soc. Int. Symp.*, 1998, pp. 536–539.

[37] R. Abdelkhalek, H. Calandra, O. Coulaud, J. Roman, and G. Latu, "Fast seismic modeling and reverse time migration on a GPU cluster," in *Proc. Int. Conf. High Perform. Comput. Simul.*, 2009, pp. 36–43.

[38] M. Louboutin *et al.*, "Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration," *Geoscientific Model Develop.*, vol. 12, no. 3, pp. 1165–1187, 2019. [Online]. Available: https://www.geosci-model-dev.net/12/1165/2019/

[39] W. W. Symes, D. Sun, and M. Enriquez, "From modelling to inversion: Designing a well-adapted simulator," *Geophys. Prospecting*, vol. 59, no. 5, pp. 814–833, 2011. [Online]. Available: 10.1111/j.1365-2478.2011.00977.x

[40] L. Ruthotto, E. Treister, and E. Haber, "jInv–a flexible Julia package for PDE parameter estimation," *SIAM J. Sci. Comput.*, vol. 39, no. 5, pp. S702–S722, 2017. [Online]. Available: https://doi.org/10.1137/16M1081063

[41] C. D. Silva and F. J. Herrmann, "A unified 2D/3D large-scale software environment for nonlinear inverse problems," *ACM Trans. Math. Softw.*, vol. 45, pp. 7:1–7:35, 2017.

[42] P. A. Witte, M. Louboutin, F. Luporini, N. Kukreja, M. Lange, G. J. Gorman, and F. J. Herrmann, "A large-scale framework for symbolic implementations of seismic inversion algorithms in Julia," *Geophysics*, vol. 84, pp. A31–V183, 2019.

[43] AWS documentation: How spot instances work, 2019. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/how-spot-instances-work.html

[44] Y. Nesterov, *Lectures on Convex Optimization*, vol. 137, Berlin, Germany: Springer, 2018

[45] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *ArXiv e-prints*, vol. abs/1412.6980, 2014. [Online]. Available: https://arxiv.org/abs/1412.6980

[46] R. Fletcher and C. M. Reeves, "Function minimization by conjugate gradients," *Comput. J.*, vol. 7, no. 2, pp. 149–154, 1964.
[47] A. Beck and M. Teboulle, "A fast iterative shrinkage-thresholding algorithm for linear inverse problems," *SIAM J. Imaging Sci.*, vol. 2, no. 1, pp. 183–202, 2009.
[48] AWS documentation: Amazon Elastic Compute Cloud, 2019. [Online]. Available: https://docs.aws.amazon.com/batch/latest/userguide/what-is-batch.html
[49] Starcluster, 2019. [Online]. Available: http://star.mit.edu/cluster/
[50] AWS High Performance Computing, 2019. [Online]. Available: https://aws.amazon.com/hpc/
[51] AWS documentation: AWS Step Functions, 2019. [Online]. Available: https://aws.amazon.com/step-functions/
[52] Iterating a loop using Lambda, 2018. [Online]. Available: https://docs.aws.amazon.com/step-functions/latest/dg/tutorial-create-iterate-pattern-section.html
[53] Docker, 2019. [Online]. Available: https://www.docker.com/
[54] AWS documentation: Amazon Simple Storage Service, 2019. [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html
[55] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, "The NumPy array: A structure for efficient numerical computation," *Comput. Sci. Eng.*, vol. 13, no. 2, 2011, Art. no. 22.
[56] AWS documentation: Amazon Simple Queue Service, 2019. [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html
[57] F. Luporini *et al.*, "Architecture and performance of Devito, a system for automated stencil computation," *ACM Trans. Math. Softw.*, 2018. [Online]. Available: https://arxiv.org/abs/1807.03032
[58] D. Joyner, O. Čertík, A. Meurer, and B. E. Granger, "Open source computer algebra systems: SymPy," *Assoc. Comput. Machinery Commun. Comput. Algebra*, vol. 45, no. 3/4, pp. 225–234, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2110170.2110185
[59] J. Rad, A. Ragab, and A. Damodar, "Building a tightly coupled molecular dynamics workflow with multi-node parallel jobs in AWS Batch," Nov. 2018. [Online]. Available: https://aws.amazon.com/blogs/compute/building-a-tightly-coupled-molecular-dynamics-workflow-with-multi-node-parallel-jobs-in-aws-batch/
[60] Boto 3 documentation, 2019. [Online]. Available: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html#
[61] J. Nocedal and S. Wright, *Numerical Optimization.* Berlin, Germany: Springer, 2006.
[62] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proc. 15th Int. Conf. Comput. Statist.*, 2010, pp. 177–186.
[63] P. A. Witte, M. Louboutin, C. Jones, and F. J. Herrmann, "Serverless seismic imaging in the cloud," 2019. [Online]. Available: https://arxiv.org/abs/1911.12447
[64] M. T. Chung, N. Quang-Hung, M.-T. Nguyen, and N. Thoai, "Using Docker in high performance computing applications," in *Proc. IEEE 6th Int. Conf. Commun. Electronics*, 2016, pp. 52–57.
[65] A. Valli and A. Quarteroni, *Domain Decomposition Methods for Partial Differential Equations*, New York, NY, USA: Numerical Mathematics and Scientific Computation, The Clarendon Press, Oxford Univ. Press, 1999.
[66] AWS documentation: AWS Batch - multi node parallel jobs, 2019. [Online]. Available: https://docs.aws.amazon.com/batch/latest/userguide/multi-node-parallel-jobs.html
[67] Amazon EC2 reserved instances pricing, 2019. [Online]. Available: https://aws.amazon.com/ec2/pricing/reserved-instances/pricing/
[68] Announcing accelerated scale-down of AWS Batch managed compute environments, 2019. [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2017/10/announcing-accelerated-scale-down-of-aws-batch-managed-compute-environments/
[69] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "Post-failure recovery of MPI communication capability: Design and rationale," *Int. J. High Perform. Comput. Appl.*, vol. 27, no. 3, pp. 244–254, 2013.
[70] S. Chakraborty *et al.*, "EReinit: Scalable and efficient fault-tolerance for bulk-synchronous MPI applications," *Concurrency Comput.: Practice Experience*, 2018, Art. no. e4863. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4863
[71] S. Hukerikar, R. A. Ashraf, and C. Engelmann, "Towards new metrics for high-performance computing resilience," in *Proc. Workshop Fault-Tolerance HPC Extreme Scale*, 2017, pp. 23–30.
[72] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
[73] AWS to Azure services comparison, 2019. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/aws-professional/services
[74] Google Cloud Platform for AWS professionals, 2019. [Online]. Available: https://cloud.google.com/docs/compare/aws/

Philipp A. Witte received the MSc degree in geophysics from the University of Hamburg, and joined the Seismic Laboratory for Imaging and Modeling (SLIM), in the fall of 2014. He is currently working toward the PhD degree in the School of Computational Science and Engineering, Georgia Institute of Technology. His research interests include large-scale seismic inverse problems, cloud computing and physics-driven machine learning.

Mathias Louboutin received the MSc degree in applied mathematics and modeling from the University of Rennes 1. He is currently working toward the PhD degree at the Seismic Laboratory for Imaging and Modeling (SLIM) at the Georgia Institute of Technology. His research interests include seismic inverse problems, numerical modeling and domain-specific languages for high-performance computing.

Henryk Modzelewski received the MSc degree in geophysics from Warsaw University, and the PhD degree in atmospheric sciences from the University of British Columbia, Canada. He is currently a research associate with the Department of Earth, Ocean and Atmospheric Sciences, University of British Columbia. His research interests include high-performance and cloud computing, as well as scientific programming.

Charles Jones received the MSc degree in exploration geophysics from the University of Leeds. He is currently the head of development at Osokey Ltd. He is interested in public cloud to overcome the CapEx requirements of on-premise high performance computing.

James Selvage received the MSc degree in exploration geophysics from the University of Leeds. He is head of implementation at Osokey Ltd. It was in 2016 at an AWS summit in London that he first heard the phrase “serverless” and left wondering how this could be applied to geophysical datasets.

Felix J. Herrmann received the PhD degree in engineering physics from the Delft University of Technology, and has been a post-doctoral fellow at the Stanford’s Mathematics Department and MIT’s Earth Resources Laboratory. He is a professor at the Georgia Institute of Technology, where he is cross appointed in the Schools of Computational Science and Engineering, Earth and Atmospheric Sciences as well as Electrical and Computer Engineering. He is the founder of the Seismic Laboratory for Imaging and Modeling (SLIM) and the industry-supported SINBAD Consortium, both of which have been responsible for major innovations in compressed sensing-based seismic acquisition, domain-specific languages for seismic modeling and inversion, and large-scale optimization for wave-equation based seismic inverse problems. He also served for over a decade on the Faculty of the University of British Columbia. His research interests include theoretical and computational aspects of exploration seismology, compressive sensing, and deep learning.

Efficient Algorithms for Delay-Aware NFV-Enabled Multicasting in Mobile Edge Clouds With Resource Sharing

Haozhe Ren, Zichuan Xu, *Member, IEEE*, Weifa Liang, *Senior Member, IEEE*, Qiufen Xia, *Member, IEEE*, Pan Zhou, *Member, IEEE*, Omer F. Rana, *Senior Member, IEEE*, Alex Galis, *Senior Member, IEEE*, and Guowei Wu

Abstract—Stringent delay requirements of many mobile applications have led to the development of mobile edge clouds, to offer low latency network services at the network edges. Most conventional network services are implemented via hardware-based network functions, including firewalls and load balancers, to guarantee service security and performance. However, implementing hardware-based network functions usually incurs both a high capital expenditure (CAPEX) and operating expenditure (OPEX). Network Function Virtualization (NFV) exhibits a potential to reduce CAPEX and OPEX significantly, by deploying software-based network functions in virtual machines (VMs) on edge-clouds. We consider a fundamental problem of NFV-enabled multicasting in a mobile edge cloud, where each multicast request has both service function chain and end-to-end delay requirements. Specifically, each multicast request requires chaining of a sequence of network functions (referred to as a service function chain) from a source to a set of destinations within specified end-to-end delay requirements. We devise an approximation algorithm with a provable approximation ratio for a single multicast request admission if its delay requirement is negligible; otherwise, we propose an efficient heuristic. Furthermore, we also consider admissions of a given set of the delay-aware NFV-enabled multicast requests, for which we devise an efficient heuristic such that the system throughput is maximized, while the implementation cost of admitted requests is minimized. We finally evaluate the performance of the proposed algorithms in a real test-bed, and experimental results show that our algorithms outperform other similar approaches reported in literature.

Index Terms—Mobile edge clouds, network function virtualization, multicasting, approximation algorithms, algorithm design

1 Introduction

With increasing uptake and use of multimedia technologies, there is an associated increase in data being generated and transmitted (processed) over our network-based systems, often to multiple subscribers. Applications can include video-on-demand, high definition streaming, multimedia social networks (combing text, audio and video) and Internet-of-Things (IoTs). This paradigm of data transfer to multiple concurrent subscribers is referred to as *multicasting*, and can significantly stress our current networks. Multicasting not only requires use of various network functions such as firewalls, Intrusion Detection Systems (IDSs), proxies, and Wide Area Networks (WAN) optimizers to guarantee data transfer security, but also to meet stringent Quality-of-Service (QoS) requirements to ensure that the traffic is transferred on time. Considering that most multimedia data needs to be multicast to mobile users, Mobile Edge-Cloud Computing (MEC) [6], [15], [16], [18], [23], [27], [33], [46], [47], [50] has emerged as a promising platform to meet the QoS requirements of mobile users, by deploying data processing resources within the proximity of mobile users. Network Function Virtualization (NFV) moves network functions from dedicated hardware to (software-based) virtual machines (VMs) that can run on commodity hardware, thereby reducing the OPEX and CAPEX of network service providers. In this paper, we consider NFV-enabled multicasting in an MEC network, where each user request requires its traffic to pass through a sequence of network functions, referred to as a *service function chain*, before reaching its destination.

Provisioning NFV-enabled multicasting services in MEC networks poses many challenges. First, each cloudlet (resource hosting a software-based Virtual Network Function (VNF)) in an MEC network usually has limited computing resource to support VNFs. Allowing multicast requests to share existing VNF instances can significantly improve

- *H. Ren, Z. Xu, and G. Wu are with the Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, School of Software, Dalian University of Technology, Dalian 116620, China. E-mail: renhaozhe@mail.dlut.edu.cn, {z.xu, wgwdut}@dlut.edu.cn.*
- *W. Liang is with the Research School of Computer Science, Australian National University, Canberra, ACT 2601, Australia. E-mail: wliang@cs.anu.edu.au.*
- *Q. Xia is with the Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, International School of Information Science and Engineering, Dalian University of Technology, Dalian 116620, China. E-mail: qiufenxia@dlut.edu.cn.*
- *P. Zhou is with the School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, Hubei 430074, China. E-mail: panzhou@hust.edu.cn.*
- *O.F. Rana is with Cardiff University, CF10 3AT Cardiff, United Kingdom. E-mail: RanaOF@cardiff.ac.uk.*
- *A. Galis is with University College London, WC1E 6BT London, United Kingdom. E-mail: a.galis@ucl.ac.uk.*

Manuscript received 9 Nov. 2019; revised 25 Feb. 2020; accepted 17 Mar. 2020. Date of publication 30 Mar. 2020; date of current version 20 Apr. 2020. (Corresponding author: Zichuan Xu.) Recommended for acceptance by M. Guo. Digital Object Identifier no. 10.1109/TPDS.2020.2983918

resource utilization in MEC networks and reduce service cost. It is however challenging to efficiently utilize existing VNF instances or create new VNF instances to maximize the number of multicast requests and minimize overall cost – subject to the computing capacity constraint on each cloudlet in the MEC network and the end-to-end delay requirement of each admitted multicast request. The key challenge is to identify which cloudlets should be used to host VNFs required within a multicast request service chain, i.e., which existing VNF instances can be used for which request? Second, each NFV-enabled multicast request usually has a QoS requirement to guarantee that its traffic reaches the destinations within the specified end-to-end delay requirement. Identifying how to meet the end-to-end delay requirement of each admitted NFV-enabled multicast request is challenging. In this paper, we tackle the aforementioned challenges, by investigating efficient methods that investigate VNF sharing, service chaining, and routing that can meet QoS requirements of NFV-enabled multicast requests in an MEC network.

There are extensive studies on multicasting in conventional networks or software-defined networks, which do not consider service function chain requirements [17], [18], [51]. These solutions however cannot be directly applied to NFV-enabled multicasting. There are also recent investigations on NFV-enabled multicasting. However, these approaches do not consider end-to-end delay requirements [39], and they assume that only one service instance is included in the service function chain [51], or that the VNFs in each service chain are consolidated into a single location [45], [47]. For example, Zhang *et al.* [51] investigated the NFV-enabled multicast problem by assuming that there are sufficient computing and bandwidth resources in a Software Defined Network (SDN) to accommodate a multicast request. Xu *et al.* [47] investigated the problem of NFV-enabled multicasting, by devising an approximation algorithm with a provable approximation ratio for realizing a single NFV-enabled multicast request and an online algorithm with a guaranteed competitive ratio for the online NFV-enabled multicasting problem. Ren *et al.* [39] investigated the NFV-enabled multicasting in an SDN, by assuming that the traffic of each multicast request can be processed by multiple instances of the VNFs in its service chain. These methods are likely to increase the cost/delay of implementing such multicast requests, as placing VNFs into multiple cloudlets can lead to a greater delay to form a service function chain and incur a higher cost.

To the best of our knowledge, we are the first to consider the problem of delay-sensitive NFV-enabled multicasting problem in an MEC network, by designing both approximation algorithms and efficient heuristics. The main contributions of this paper are as follows.

- We study the NFV-enabled multicasting problem in an MEC network, with an aim to minimize the implementation cost of the request while meeting its delay requirement.
- We propose an efficient heuristic for the NFV-enabled multicasting problem. We also devise the very first approximation algorithm with an approximation ratio, if the delay requirement is neglected.
- We also consider a set of NFV-enabled multicast request admissions with the aim to maximize the weighted system throughput. We also propose a heuristic for this problem.
- We evaluate the performance of the proposed algorithms through experimental simulations in synthetic networks and within a real test-bed. Experimental results demonstrate that the proposed algorithms outperform existing reported approaches.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 introduces the system model, notations, and problem definition. Section 4 devises an approximation algorithm for the NFV-enabled multicasting problem without end-to-end delay requirements, and proposes an efficient heuristic for the problem with delay requirements using the proposed approximation algorithm as a subroutine. Section 5 devises an efficient heuristic algorithm for the the NFV-enabled multicasting problem with resource constraints on cloudlets. Section 6 evaluates the performance of the proposed algorithms experimentally in a real test-bed, and Section 7 concludes the paper.

2 Related Work

Recently, traffic steering has re-gained much attention due to the challenges introduced by software defined networking and network function virtualization [5], [6], [15], [16], [18], [23], [27], [33], [46], [47], [50]. Unicasting is one of the primary focus of existing studies. For example, Moens *et al.* [33] focused on hybrid networks with both hardware and software network functions. Cziva *et al.* [7] addressed the problem of the placement of virtual functions by minimizing the total number of VNF instances. Yu *et al.* [29] investigated profit maximization associated with placing VNFs onto a set of locations, and considered the delay requirement of each unicast request. Xu *et al.* [45] studied the offloading problem of delay-sensitive tasks with network function requirements in an MEC network, by proposing efficient heuristics and an online algorithm with a competitive ratio. Xie *et al.* [44] investigated the VNF sharing problem with an aim to improve resource utilization, by finding a common link for a set of service chains, so that the deployed service chains can be shared by all users. Kiji *et al.* [19] proposed a virtual network function placement and routing algorithm for multicast requests with service chain requests, through merging multiple service paths (MSC-M). Although there exist studies that consider the delay requirements of user requests [22], [29], [45], they only considered unicast requests and their solutions cannot be applied to the NFV-enabled multicasting problem, which is a generalization of the NFV-enabled unicasting problem. Chen and Wu [5] devised algorithms for the VNF placement to minimize the cost of implementing NFV-enabled unicast requests by balancing set-up and bandwidth consumption costs.

There are studies on multicasting in conventional networks [2], [14], [24], [25], [34], [43]. Recently, with the emergence of new networking technologies such as mobile edge computing, software-defined networking (SDN) and NFV, multicasting has re-gained the attention by the research community [17], [18]. For example, Huang *et al.* [18] studied online multicasting in software-defined networks with both node and link capacity constraints. Huang *et al.* [17] studied the scalability problem of multicasting in SDNs, by proposing an efficient algorithm to find a branch-aware Steiner Tree for each multicast request. These solutions however cannot be directly applied to the problem of NFV-enabled multicasting in MEC networks, because they did not consider the service chain requirements of multicast requests.

Investigations on NFV-enabled multicasting include [1], [30], [31], [39], [41], [47], [49], [51]. For instance, Zhang *et al.* [51] investigated the NFV-enabled multicasting problem in an SDN without resource capacity constraints, assuming that data traffic of each multicast request can only be processed by one server. Xu *et al.* [47], [48] considered the NFV multicasting problem by assuming the traffic of each request can be processed by multiple servers, with the objective to minimize the implementation cost. Approximation and online algorithms for the problems are proposed. They however assumed that the VNFs in each service chain is consolidated into a single data center. Ma *et al.* [30], [31] proposed an online algorithm for the NFV-enabled multicasting problem without taking into account the end-to-end delay requirement. Soni *et al.* [41] proposed a scalable multicast group management scheme and a load balancing method for the routing of best-effort traffic and bandwidth-guaranteed traffic. These studies however did not consider end-to-end delay requirements of multicast requests. Alhussein *et al.* [1] devised exact solutions for the problem of joint VNF placement and routing for multicast requests in 5G core networks, such that the cost of provisioning NFV-enabled multicast services is minimized, by formulating the problem into a mixed integer linear program (MILP). The delay requirement of NFV-enabled requests has not been considered and the MILP-based exact solutions might not be scalable for large problem sizes. Yi *et al.* [49] considered delay requirements of the NFV-enabled multicasting problem; however VNF sharing is not explored. To guarantee scalability and solution quality, Ren *et al.* [39] proposed approximation algorithm with an approximation ratio for the problem of embedding a service graph that consists of VNF instances into a substrate network, by assuming that the traffic of each multicast request can be processed by multiple instances of the VNFs in its service chain. The delay requirement of multicast requests however is not considered in the study. Similarly, the delay requirement of multicast requests is not considered [30], [31], and the authors only consider a single multicast request.

3 Preliminaries

In this section, we first introduce the system model, notation and key concepts. We then define the problem being considered more precisely.

3.1 System Model

We consider a mobile edge cloud (MEC) network $G = (V, E)$ with a set V of switches, a set of cloudlets and a set E of links between switches and cloudlets. Each cloudlet is attached to a switch in V via an optical fiber, and the communication delay between a switch and its attached cloudlet is negligible. Let V_{CL} be the set of switches with attached cloudlets. Clearly, $V_{CL} \subseteq V$. Cloudlets are usually deployed in shopping malls, airports, or base stations that are within the proximity of mobile users. Due to space limitation of installing cooling equipment in those places, each cloudlet is usually equipped with (a small number of) servers and thus has computing resource capacity to implement VNF instances. We denote by C_v the computing capacity of the cloudlet attached to switch node $v \in V_{CL}$. In addition, transferring data through links in E incurs a communication latency. Let d_e be the delay associated with transmitting a unit of data traffic via link $e \in E$. We assume that there is an SDN controller that both makes traffic steering decisions and manages network function instances that run on a server in the MEC network G. Fig. 1 is an illustrative example of an MEC network.

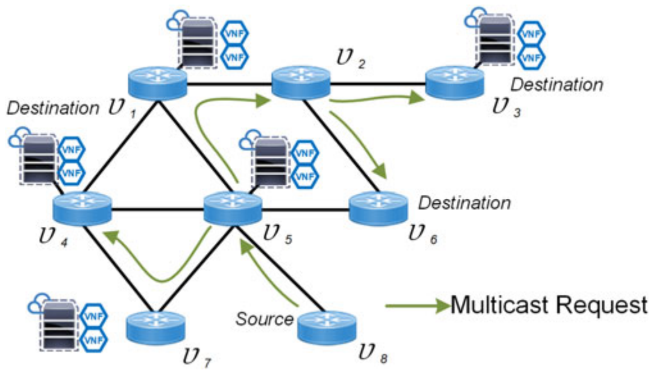


Fig. 1. An MEC network G.

3.2 NFV-Enabled Multicast Requests and Service Chains

A delay-aware NFV-enabled multicast request is a request that transfers an amount of data traffic from a source to a set of destinations. The data traffic must be processed by a sequence of VNFs before reaching their destinations, while also meeting delay constraints.

Let r_k be a delay-aware NFV-enabled multicast request, denoted by a quadruple $r_k = (s_k, D_k; b_k, SC_k)$, where $s_k \in V$ is the source, D_k is the set of destinations with $D_k \subseteq V$, b_k is the size of its data traffic, and SC_k is the *service chain* of r_k that consists of a sequence of VNFs. Without loss of generality, we consider that the data traffic b_k of request r_k is given (derived from historical information).

Let $\mathcal{F}$ be the set of VNFs provided by the network service provider in G. A VNF $f_l \in \mathcal{F}$ can be needed by request r_k to form its service function chain SC_k. Assume that there are L_k VNFs in SC_k, where $1 \le l \le L_k$ for each SC_k and $SC_k \subset \mathcal{F}$. We further assume that there is a number of already instantiated VNF instances for each type of network function f_l in cloudlets of G. Due to the resource capacity constraints on cloudlets, we allow the instances of VNF f_l to be shared among different requests.

To admit request r_k, all data traffic from source s_k of r_k needs to be processed through an instance of each VNF $f_l \in SC_k$ prior to reaching destinations in D_k, as illustrated in Fig. 2. An existing instance must therefore be selected for each VNF $f_l \in SC_k$, or a new instance of f_l must be instantiated in a cloudlet of G. Existing or newly created VNF instances of each service chain SC_k can be placed in multiple cloudlets, because a single cloudlet may not have all the instances of the VNFs in SC_k, or it may lack sufficient computing resources to create new instances for all VNFs in SC_k.

Each multicast request needs a certain amount of computing resource to process its data traffic. Let $C_{unit}(f_l)$ be

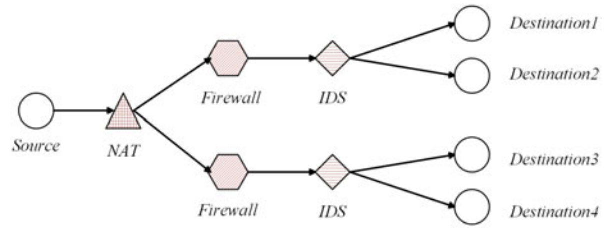


Fig. 2. A service chain $\langle$ NAT, Firewall, IDS $\rangle$ with one instance of NAT and two instances of Firewall and IDS.

the number of computing resource needed to process a unit amount of data traffic. If f_l is implemented as a newly created instance, the total number of computing resources that should be assigned to the new instance to process the data traffic of request r_k is $C_{unit}(f_l) \cdot b_k$. Otherwise, an existing instance of f_l should have at least an amount $C_{unit}(f_l) \cdot b_k$ of available computing resource to process the traffic of r_k. Notice that we assume that the accumulative available resources in the cloudlets of G are higher then the total resource demand of a single request r_k; however, for a specific cloudlet in V_{CL}, it may not have enough resources to meet the demand of r_k.

3.3 Delay Requirements of Multicast Requests

The end-to-end delay of implementing a multicast request plays a vital role in guaranteeing the quality of services of users. We thus consider that each multicast request has a delay requirement, which specifies the maximum delay it can tolerate for transmitting its data from its specified source to its destinations. For a delay-aware NFV-enabled multicast request, its experienced delay consists of the total processing delay in the selected cloudlets and the total transfer delay from the source to cloudlets and from the cloudlets to the destinations, which are defined in the following.

Processing Delay. The processing delay experienced by a multicast request r_k depends on both the amount of data traffic that needs to be processed and the computing resource assigned to process the traffic. Without loss of generality, we assume that the processing delay $d^p_{k,l}$ of each multicast request r_k by VNF f_l is proportional to the amount of traffic it needs to process, i.e.,

$$d^p_{k,l} = \alpha_l \cdot b_k, \tag{1}$$

where α_l is a given proportional factor of VNF f_l.

The accumulative processing delay incurred due to the traffic processing by network functions in SC_k of r_k is

$$d^p_k = \sum\nolimits_{f_l \in SC_k} d^p_{k,l}. \tag{2}$$

Transmission Delay. Let $\mathcal{P}_k$ be the set of routing paths from source s_k to destinations in D_k, where each path $p_m \in \mathcal{P}_k$ denotes a routing path from s_k to a destination $t_m \in D_k$. The transmission delay of each r_k is the maximum end-to-end delay incurred in the paths in $\mathcal{P}_k$. We denote by d^t_k the transmission delay of request r_k, which can be defined as follows:

$$d^t_k = \max\nolimits_{p_m \in \mathcal{P}_k} \sum_{e \in p_m} d_e \cdot b_k. \tag{3}$$

The delay experienced by multicast request r_k thus is

$$d_k = d^p_k + d^t_k, \tag{4}$$

which needs no greater than the specified delay requirement D_k, i.e.,

$$d_k \leq D_k. \tag{5}$$

3.4 Cost Models

As the network service provider of an MEC network G charges user requests on a pay-as-you-go basis, the major concern of the service provider is its *operational cost*, which consists of computing resource usage costs in cloudlets, bandwidth resource usage costs in links, and VNF instance instantiation costs. Let $c(e)$ and $c(v)$ be the usage costs of one unit of bandwidth and computing resources at link $e \in E$ and cloudlet $v \in V_{CL}$, respectively. Denote by $c_l(v)$ the cost of instantiating an instance of network function f_l in cloudlet $v \in V_{CL}$, and let $n'_{l,v}$ be the number of newly created instances for network function f_l in cloudlet v. Denote by $n_{l,v}$ the number of existing instances of f_l in v that are used to process the traffic of r_k.

The operational cost of admitting a delay-aware NFV-enabled multicast request r_k can be specified as

$$\begin{aligned} c_k = & \sum_{f_l \in SC_k} \sum_{v \in V_{CL,r_k}} ((n_{l,v} + n'_{l,v}) \cdot c(v) \cdot b_k + n'_{l,v} \cdot c_l(v)) \\ & + \sum_{e \in T_k} c(e) \cdot b_k, \end{aligned} \tag{6}$$

where V_{CL,r_k} is the set of cloudlets that are used to implement the instances of VNFs in SC_k of request r_k, and T_k is the obtained multicast tree that is used to route the data traffic of r_k.

3.5 The Directed Steiner Tree [4]

The Steiner tree problem is defined as follows: given a graph $G = (V, E)$ with a cost function c on the edges, and a subset of terminals $X \subset V$, the goal is to find a minimum cost tree that includes all the terminals in X. The found minimum cost tree is referred to as the *Steiner tree*.

3.6 Problem Definition

We consider a mobile edge cloud (MEC) network $G = (V, E)$ with a set V_{CL} of cloudlets with $V_{CL} \subset V$, and a set of multicast requests R. Given a snapshot of the MEC at a given time instant and a NFV-enabled multicast request r_k, understanding how request r_k can be realised across a set of VNFs remains a key challenge. We thus first consider the problem of admitting a single multicast request r_k, such that its operational cost is minimized. Further, considering that the accumulated computing resources in an MEC may be insufficient to implement all requests, another question is identifying how to carry out admission control for multicast requests to maximize weighted throughput. In the following we define these two optimization problems precisely.

Problem 1. *Assuming that each multicast request can be implemented using the computing resources assigned to existing VNF instances, the* NFV-enabled multicasting problem with a single multicast request *in MEC network G is to*

TABLE 1
Symbols

Symbols	Meaning
$G = (V, E)$	a mobile edge cloud (MEC) network with a set V of switches and a set E of links
R	a set of delay-aware NFV-enabled multicast requests
V_{CL}	the set of switches with attached cloudlets, and clearly $V_{CL} \subseteq V$
C_v	the computing capacity of the cloudlet attached to a switch node $v \in V_{CL}$
$r_k = (s_k, D_k; b_k, SC_k)$	a delay-aware NFV-enabled multicast request, where $s_k \in V$ is the source, D_k is the set of destinations with $D_k \subseteq V$, b_k is the size of its data traffic, and SC_k is the *service chain* of r_k that consists of a sequence of VNFs.
$\mathcal{F}$ and f_l	the set of VNFs provided by the network service provider in G and a VNF f_l
L_k	The number of VNFs in SC_k
$C_{unit}(f_l)$	the amount of computing resource needed to process a unit amount of data traffic
$d^p_{k,l}$	the processing delay of each multicast request r_k by VNF f_l
α_l	a given proportional factor of VNF f_l
d^p_k	the accumulative processing delay incurred due to the traffic processing by network functions in SC_k of r_k
$\mathcal{P}_k$	the set of routing paths from source s_k to destinations in D_k
$p_m \in \mathcal{P}_k$	a routing path from s_k to a destination $t_m \in D_k$
d^t_k	the transmission delay of request r_k
d_k and D_k	the delay experienced by multicast request r_k and its delay requirement
$c(e)$ and $c(v)$	the usage costs of one unit of bandwidth and computing resources at link $e \in E$ and cloudlet $v \in V_{CL}$
$c_l(v)$	the cost of instantiating an instance of network function f_l in cloudlet $v \in V_{CL}$
$n'_{l,v}$	the number of newly created instances for network function f_l in cloudlet v
$n_{l,v}$	the number of existing instances of f_l in v that are used to process the traffic of r_k
c_k	the operational cost of admitting a delay-aware NFV-enabled multicast request r_k
V_{CL,r_k}	the set of cloudlets that are used to implement the instances of VNFs in SC_k of request r_k
T_k	the obtained multicast tree that is used to route the data traffic of r_k
ST and R_{ad}	the weighted throughput and the set of admitted multicast requests
n'_k	the number of cloudlets that are used to implement the VNFs in SC_k in the current infeasible solution
n_k	the proper number of cloudlets in the feasible solution of algorithm `Heu_Delay`
n_{max} and n_{min}	the minimum and maximum bounds of the binary search range in algorithm `Heu_Delay`
$G' = (V', E')$	the auxiliary graph constructed in algorithm `Appro_NoDelay`
$f'_{i,l,v}$ and $f''_{i,l,v}$	the pair of virtual VNF nodes for the ith VNF instance of f_l in cloudlet $v \in V_{CL}$
$w(f'_{i,l,v}, f''_{i,l,v})$	the weight of edge $\langle f'_{i,l,v}, f''_{i,l,v}\rangle$ in the auxiliary graph G'.
$v'_{k,l}$ and $v''_{k,l}$	a pair of virtual cloudlets for the lth VNF and cloudlet v in G'
$W_{l,v}$	the widget that is built for network function f_l in cloudlet $v \in V_{CL}$
$ws_{l,v}$ and $wd_{l,v}$	a widget source node and a widget destination node for the widget for network function f_l and cloudlet $v \in V_{CL}$ in auxiliary graph G'
c^*	the optimal solution for the NFV-enabled multicasting problem
L_{max}	the maximum length of the service chains of the requests in R, i.e., $L_{max} = \arg\max_{r_k \in R} \lvert SC_k \rvert$.
L_{com} and $R(L_{com})$	the number of common VNFs that requests have in their service chains VNFs in common of their service chains, and the set of such requests.

route the traffic of request r_k to each destination in D_k by chaining either existing or newly created instances of VNF, such that the operational cost (i.e., Eq. (6)) of implementing r_k is minimized, while meeting the end-to-end delay requirement D_k of r_k and capacity constraint on each cloudlet $v \in V_{CL}$.

Problem 2. *Assuming that the computing resource in each cloudlet in the MEC network G has available capacity. For each request in R, the network may or may not have enough resources to admit it, the* NFV-enabled multicasting problem *in an MEC network G for a given set R of NFV-enabled multicast requests is to maximize the system throughput while minimizing the operational cost, subject to computing capacity on each cloudlet, where the system throughput is defined as the total amount of data that is processed and transferred by the system for admitted multicast requests. Let ST be the weighted throughput and R_{ad} the set of admitted multicast requests, then*

$$ST = \sum_{r_k \in R_{ad}} b_k. \tag{7}$$

The NFV-enabled multicasting problems are NP-hard, as its special case – the traditional multicast problem without NFV service chain constraints is NP-hard [8].

For clarity, the symbols used in this paper are summarized in Table 1.

4 Algorithms for the Admission of a Single NFV-Enabled Multicast Request

In this section, we deal with NFV-enabled multicasting for a single NFV-enabled multicast request admission. We first propose an efficient heuristic for the problem. We then consider a special case of the problem without delay requirements, by devising an approximation algorithm.

4.1 An Efficient Heuristic

The basic idea of the proposed heuristic is based on an observation that a feasible solution to the problem needs to meet the capacity constraints on cloudlets, service function chain requirements, and the end-to-end delay requirement of each multicast requests r_k. We thus adopt a two-phase heuristic that progressively considers the mentioned constraints and requirements.

Phase One. We first propose an algorithm to jointly consider the capacity constraint and the service chain requirement, by ignoring the delay requirement of r_k. The proposed algorithm smartly explores existing VNF instances in each cloudlet that can be shared with the VNF instances of r_k. Notice that the solution may not be feasible to the NFV-enabled multicasting problem, because the delay requirement of r_k is not considered in this phase. For the sake of clarity, we describe the proposed algorithm for the problem without delay requirement in the next subsection, which is referred to as `Appro_NoDelay`. By now, we assume we already obtained the multicast tree for r_k in G without considering its delay requirement.

Phase Two. We refine the obtained multicast tree into a feasible solution to meet the delay requirement of r_k. In particular, we observe that a longer delay will be the result if the VNFs of SC_k are implemented in multiple cloudlets. This is because that if the VNFs are distributed into different cloudlets, the data traffic transmission among two consecutive VNFs has to be performed by inter-cloudlet links, which incurs higher delays than those by intra-cloudlet data transfers. However, putting all VNFs into a single cloudlet may also incur a longer delay, since the selected cloudlet may be far away from the destinations of r_k. This means that a large or a small value for the number of cloudlets of a request may not be proper to meet the delay requirement of r_k. We thus adopt a binary search to narrow down the choices of the proper number of cloudlets for r_k, making the delay requirement of r_k being met quickly. Specifically, let n'_k be the number of cloudlets that are used to implement the VNFs in SC_k in the current infeasible solution, and denote by n_k the proper number of cloudlets in the feasible solution. We first set

$$n_k = \left\lfloor \frac{|V_{CL}|+1}{2} \right\rfloor. \tag{8}$$

The proposed algorithm first tries to re-assign the VNFs in service function chain SC_k such that they are implemented in exactly n_k cloudlets. If $n_k < n'_k$, we identify a number of $(n'_k - n_k)$ cloudlets that implements VNFs of SC_k in the obtained infeasible solution from the Steiner tree [4] (i.e., multicast tree) in G' and have the longest average data transfer delay from it to the destinations in D_k. Let F' be the set of instances of VNFs in SC_k that are implemented in the identified cloudlets. The VNFs in F' are pre-consolidated to the rest n_k cloudlets in V' one by one, by selecting a cloudlet with the lowest implementation cost for each $f_l \in F_{v'}$. If the pre-consolidation makes the delay requirement of r_k being met, the algorithm terminates with a feasible solution. Otherwise, if the experienced delay of r_k is reduced but still greater than its requirement, we continue the above procedure by searching the appropriate number of cloudlets in the range of $[1, n_k]$. The rationale is that the number of cloudlets in the multicast tree is still too many, and the inter-cloudlet communication leads to the delay requirement violation. The number of cloudlets still needs to be reduced. Instead, if the experienced delay is increased, we try to find the appropriate value for n_k in the range of $[n_k, |V_{CL}|]$. This means increasing the number of cloudlets for r_k may reduce the experienced delay of multicast request r_k. On the other hand, if $n_k > n'_k$, we need to find the additional $n_k - n'_k$ cloudlets that have the lowest implementation cost for VNFs of r_k, and pre-assign VNFs in F' to

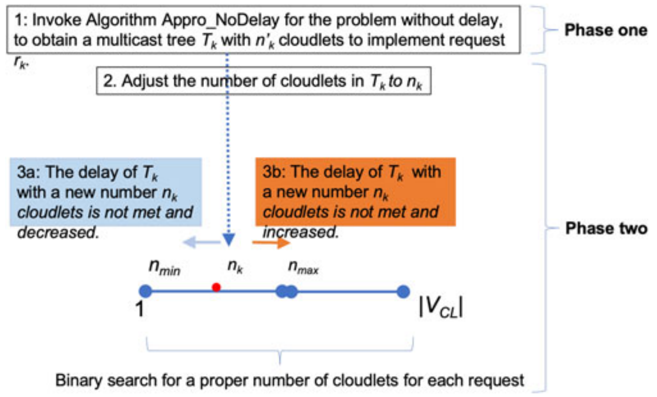

1: Invoke Algorithm Appro_NoDelay for the problem without delay, to obtain a multicast tree T_k with n'_k cloudlets to implement request r_k.
Phase one
2. Adjust the number of cloudlets in T_k to n_k
3a: The delay of T_k with a new number n_k cloudlets is not met and decreased.
3b: The delay of T_k with a new number n_k cloudlets is not met and increased.
Phase two
n_{min}
n_k
n_{max}
1
$|V_{CL}|$
Binary search for a proper number of cloudlets for each request

Fig. 3. An illustration of the algorithm `Heu_Delay`.

the cloudlets one by one. The above binary search procedure continues until a feasible solution is obtained or the multicast request is rejected. The detailed heuristic is described in Algorithm 1 and its basic idea is shown in Fig. 3. For simplicity, this algorithm is referred to as algorithm `Heu_Delay` in the rest of this paper.

Algorithm 1. `Heu_Delay`

```
Input: $G = (V, E)$, $V_{CL}$, computing capacity $C_v$ for each cloudlet $v \in V_{CL}$, and a multicast request $r_k = (s_k, D_k; b_k, SC_k)$ and its delay requirement $d_k^{req}$.
Output: The locations for the VNFs of service chain $SC_k$ of multicast request $r_k$ and the multicast tree $T_k$ to transfer its data.
1: /*Phase one: find cloudlets and routing paths for $r_k$ by considering its service chaining requirement and cloudlet capacity constraints.*/
2: Find a multicast tree for $r_k$ without considering its delay requirement $d_k^{req}$, by invoking algorithm Appro_NoDelay;
3: Let $n'_k$ be the number of cloudlets that are used to implement VNFs in $SC_k$ of the found multicast tree;
4: /*Phase two: adjust the multicast tree to meet the delay requirement of $r_k$.*/
5: $n_{min} \leftarrow 1$;
6: $n_{max} \leftarrow |V_{CL}|$;
7: while $n_{min} <= n_{max}$ do
8:     $n_k \leftarrow \lfloor \frac{n_{min}+n_{max}}{2} \rfloor$;
9:     if $n_k < n'_k$ then
10:        Identify the number of $n'_k - n_k$ cloudlets that implements VNFs of $SC_k$ in the obtained solution from the Steiner tree in $G'$ and has the top-($n'_k - n_k$) highest average data transfer delays from it to the destinations in $D_k$;
11:        Move the VNFs that were implemented in the $n'_k - n_k$ cloudlets of the infeasible solution to the rest cloudlets one by one.
12:    else
13:        Find the additional $n_k - n'_k$ cloudlets that have the lowest implementation cost for VNFs of $r_k$, and assign VNFs in $F_{v'}$ to the cloudlets one by one.
14:    if the experienced delay of $r_k$ is met then
15:        return;
16:    else
17:        if the experienced delay of $r_k$ is decreased then
18:            $n_{max} \leftarrow n_k$;
19:        else
20:            $n_{min} \leftarrow n_k$;
```

4.2 An Approximation Algorithm for the Problem Without Delay Requirements

The proposed approximation algorithm for the problem without delay requirements is to reduce the problem in G to the Steiner tree problem in an auxiliary graph G', via a non-trivial reduction. Since each cloudlet $v \in V_{CL}$ has computing capacity to implement the VNFs of each request, the VNFs in each service function chain SC_k can be implemented in multiple cloudlets or consolidated into a single cloudlet to save the communication cost due to the transmissions between different cloudlets. To ensure that each cloudlet has sufficient computing resource to implement the VNFs in SC_k of each multicast request r_k, we adopt a conservative method of reserving $\sum_{f_l \in SC_k} b_k \cdot C_{unit}(f_l)$ resource for r_k in each cloudlet. The cloudlet with an amount of available computing resource that is less than $\sum_{f_l \in SC_k} b_k \cdot C_{unit}(f_l)$ will be removed from the network G, where the available resource in idle VNF instances are also accounted.

The Construction of Auxiliary Graph $G' = (V', E')$. We now construct G' based on the sub-network of G.

We start by constructing the node set V' of G'. Specifically, we first add source node s_k into the auxiliary graph. We also add each node in V into V', i.e., $V' \leftarrow V' \cup V$. Notice that, since $V_{CL} \subset V$, all switch nodes in V_{CL} are added into V as well. However, only their functionalities of forwarding traffic will be used.

Recall that VNFs in SC_k of multicast request r_k can be assigned to existing VNFs or newly instantiated VNF instances. To determine whether making use of existing VNF instances or creating new ones, we create a *widget* for each cloudlet $v \in V_{CL}$ and each network function $f_l \in SC_k$ to represent the resource availability of the cloudlet v for f_l by two cases. Case 1: the amount of available computing resource to instantiate new instances of VNFs; Case 2: existing VNF instances of f_l in $v \in V_{CL}$ that are available to process the traffic of r_k. There is a widget for each pair of cloudlet and VNF, which actually means a possible placement of a VNF to a cloudlet.

For Case 1, we add a pair of *virtual VNF nodes* into the widget, to represent each of existing VNF instances of f_l with sufficient computing resource processing the data traffic of r_k in cloudlet $v \in V_{CL}$. Denote by $f'_{i,l,v}$ and $f''_{i,l,v}$ the pair of virtual VNF nodes for the ith VNF instance of f_l in cloudlet $v \in V_{CL}$. We then add an edge from $f'_{i,l,v}$ to $f''_{i,l,v}$ into the widget. The weight of edge $\langle f'_{i,l,v}, f''_{i,l,v}\rangle$ is the cost of processing a unit traffic by an existing VNF instance of f_l in cloudlet v, i.e., $w(f'_{i,l,v}, f''_{i,l,v}) = c(v_{f_l,r_k})$.

For Case 2, we add a pair of *virtual cloudlets* for each cloudlet $v \in V_{CL}$ into each widget to denote the amount of available computing resource to instantiate a new instance of f_l in cloudlet v, as shown in Fig. 4. Let $v'_{k,l}$ and $v''_{k,l}$ be such a pair of virtual cloudlets for the lth VNF and cloudlet v. To jointly consider the processing and transmission costs, we connect each pair of virtual cloudlets, $v'_{k,l}$ and $v''_{k,l}$, i.e., $E' \leftarrow E' \cup \{\langle v'_{k,l}, v''_{k,l}\rangle\}$. The weight of edge $\langle v'_{k,l}, v''_{k,l}\rangle$ is the sum of the instantiation cost of VNF f_l and the cost of processing a unit traffic by the lth VNF in SC_k for each multicast request r_k in cloudlet v. That is, $w(\langle v'_{k,l}, v''_{k,l}\rangle) = \frac{c_l(v)}{b_k} + c(v_{f_l,r_k})$.

We also add a *widget source node* $ws_{l,v}$ and a *widget destination node* $wd_{l,v}$ for the widget for network function f_l and cloudlet $v \in V_{CL}$. Node $ws_{l,v}$ is connected to node $v'_{k,l}$ and

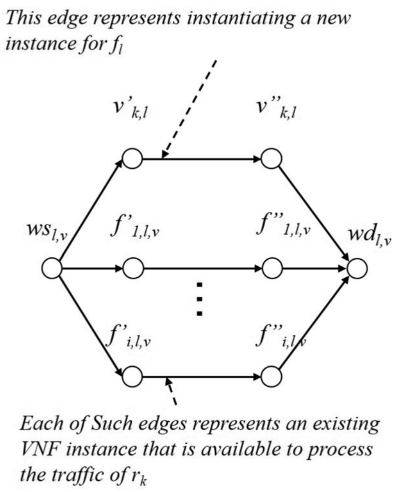

This edge represents instantiating a new instance for f_l
$v'_{k,l}$
$v''_{k,l}$
$ws_{l,v}$
$f'_{1,l,v}$
$f''_{1,l,v}$
$wd_{l,v}$
$f'_{i,l,v}$
$f''_{i,l,v}$
Each of Such edges represents an existing VNF instance that is available to process the traffic of r_k

Fig. 4. An example of the widget for the VNF f_l in SC_k and cloudlet $v \in V_{CL}$.

the node f'_l for each existing instance of network function f_l that has enough computing resource to process the data traffic of r_k. In addition, node $v'_{k,l}$ and node f'_l for each existing instance of network function f_l are both connected with the widget destination node $wd_{l,v}$. The weights of those edges are set to zeros. It must be mentioned that widget source and destination nodes are used to guarantee that either a new instance for f_l is created or an existing VNF instance of f_l is selected to process the traffic of r_k, which will be proved in the algorithm analysis part.

The widgets become part of the auxiliary graph G'.

We then connect the widgets and other nodes in the auxiliary graph G' as follows.

- *s_k to widget source nodes:* There is an edge from source node s_k to each widget source node $ws_{l,v}$ of the widget for the first VNF f_1 of SC_k and every $v \in V_{CL}$. The weight of edge $\langle s_k, ws_{l,v}\rangle$ is set as the transmission cost of data traffic of r_k.
- *widget destination to widget source nodes:* Since the data traffic of r_k may be processed by multiple cloudlets, there is an edge from the widget destination node of each widget for network function f_l to the widget source node of each widget for VNF f_{l+1}, for each l with $1 \le l \le L_k - 1$, i.e., $E' \leftarrow E' \cup \{\langle wd_{l,v}, ws_{l+1,u}\rangle\}$ for l with $1 \le l \le L_k - 1$ and v, u in V_{CL}. The weight of edge $\langle wd_{l,v}, ws_{l+1,u}\rangle$ is the transmission cost of a unit traffic along the shortest path from cloudlet v to cloudlet u.
- *widget destinations of f_{L_k} to cloudlet nodes:* We finally connect each of the widgets that are created for the last VNF $f_{L_k} \in SC_k$ with the cloudlet node. Specifically, there is an edge from node $wd_{L_k,v}$ to cloudlet node u in V', i.e., $E' \leftarrow E' \cup \{\langle wd_{L_k,v}, u\rangle\}$. The weight of edge $\langle wd_{L_k,v}, u\rangle$ is the transmission cost of a unit traffic along the shortest path from cloudlet v to cloudlet u.

An example of the constructed auxiliary graph is shown in Fig. 5.

Problem Reduction. We now reduce the NFV-enabled multicasting problem without delay requirements in G to the Steiner tree problem in the directed auxiliary graph G'. Recall that in the construction of G', the VNF processing

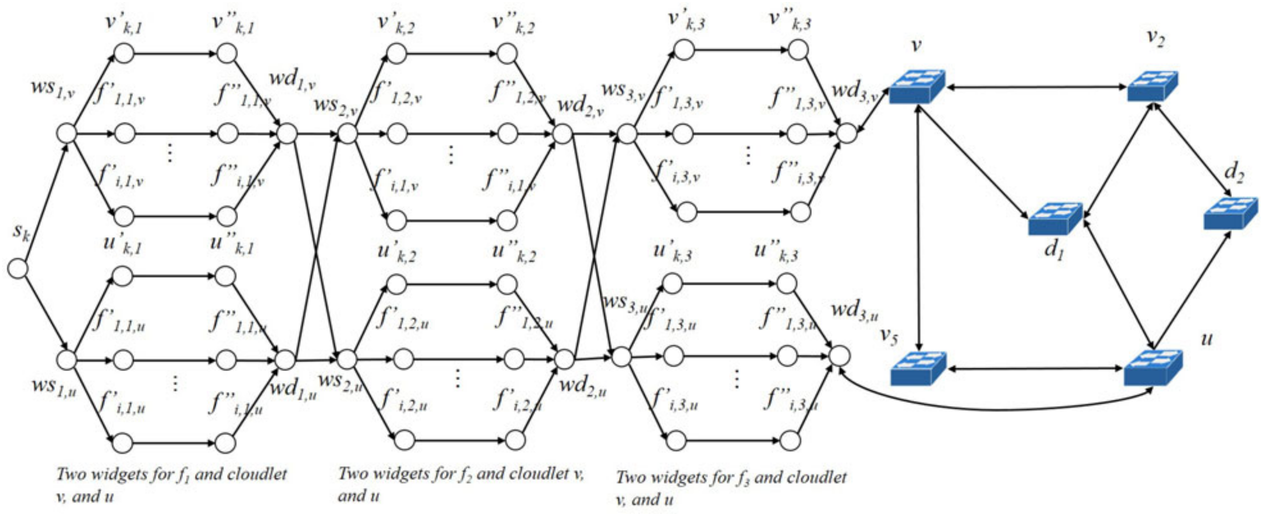


Fig. 5. An example of the auxiliary graph $G' = (V', E')$ with two servers attached at node v and node u and multicast request r_k transfer its data to destinations in $D_k = \{d_1, d_2\}$. Note that there is a widget for each pair of VNF f_l and cloudlet v, corresponding to a possible assignment of f_l. The original switches that attach the two cloudlets will just serve as normal forwarding switches.

and transmission costs are considered as the weights of edges. We thus find a directed Steiner tree in G' that spans nodes in $\{s_k\} \cup D_k$. We then transfer the Steiner tree in G' to routing paths for r_k in the original network G. Specifically, if a widget for $f_l \in SC_k$ of and cloudlet $v \in V_{CL}$ is included in the Steiner tree, either a newly created VNF instance or an existing one in cloudlet v will be used to implement f_l, depending on which edge of the widget is included in the Steiner tree. Notice that the edges among the widgets in G' correspond to the shortest paths of their endpoints of the edges in G. We thus replace each of such edges with its shortest path in G.

Algorithm 2. `Appro_NoDelay`

```
Input: $G = (V, E)$, $V_{CL}$, computing capacity $C_v$ for each cloudlet
    $v \in V_{CL}$, and a multicast request $r_k = (s_k, D_k; b_k, SC_k)$.
Output: The locations for the VNFs of service chain $SC_k$ of multicast request $r_k$ and the multicast tree $T_k$ to transfer its data.
1: Construct an auxiliary directed graph $G' = (V', E')$, as shown in Fig. 5;
2: Find a directed Steiner tree $T$ in $G'$ that spans nodes in
    $\{s_k\} \cup D_k$, using Charikar's algorithm [4];
3: For each path from the widget source node to the widget destination node of a widget in $T$, condense the path to a single node;
4: Replace each of all other edges in $T$ with its corresponding shortest path in network $G$; /*The edges among widgets correspond to shortest paths in the original network $G$. */
```

4.3 Algorithm Analysis

We now analyze the feasibility of the solution obtained and performance of the proposed algorithms.

We first show the feasibility of the solution delivered by Algorithm 2. Intuitively, if a solution to the NFV-enabled multicasting problem, it needs to satisfy the following three conditions:

- *Condition 1*: each VNF $f_l \in SC_k$ will be assigned to one or multiple cloudlets by either creating a new instance or using an existing instance
- *Condition 2*: the traffic of r_k will be processed by VNFs as the specified order in SC_k
- *Condition 3*: the processed traffic by the VNFs in SC_k is forwarded to destinations in D_k of r_k.

For Condition 1, we show that in each of the selected cloudlets for f_l, either a new instance is created or an existing instance is selected for it in the following lemma.

Lemma 1. *If a cloudlet $v \in V_{CL}$ is selected for VNF $f_l \in SC_k$ of multicast request r_k, either an existing instance of f_l or a newly created instance is used to process the traffic of r_k.*

Proof. Following the construction of G', showing the feasibility of the solution is to show that if the Steiner tree found in G' has one path from $ws_{l,v}$ to $wd_{l,v}$ of each selected widget, the path will be the only path in the Steiner tree, and no other paths in the widget will be included. Let $W_{l,v}$ be the widget that is built for network function f_l in cloudlet $v \in V_{CL}$. Assume that widget $W_{l,v}$ is included into the Steiner tree for the subgraph, and let p be the path from $ws_{l,v}$ to $wd_{l,v}$ of $W_{l,v}$ in G' that is included in the Steiner tree. We prove by contradiction. Assume that there is another instance (either newly created or existing one) of f_l is used to process the traffic of r_k. Let the ith instance of f_l be such an additional instance. This means that edge $\langle f'_{i,l,v}, f''_{i,l,v}\rangle$ has to be included in the Steiner tree found in G'. Edges $\langle ws_{l,v}, f'_{i,l,v}\rangle$ and $\langle f''_{i,l,v}, wd_{l,v}\rangle$ have to included, according to the structure of the widget; otherwise, edge $\langle f'_{i,l,v}, f''_{i,l,v}\rangle$ is a stand alone edge that can be removed. Let p' be the path that consisting of edges $\langle ws_{l,v}, f'_{i,l,v}\rangle$, $\langle f'_{i,l,v}, f''_{i,l,v}\rangle$, and $\langle f''_{i,l,v}, wd_{l,v}\rangle$, as shown in Fig. 6. Paths p' and p however make it not a tree. Therefore, only one path from $ws_{l,v}$ to $wd_{l,v}$ will be included in the Steiner tree for the subgraph of G' that is composed of source node s_k and the widgets, meaning that a newly created or existing instance of f_l will be selected in cloudlet $v \in V_{CL}$. The lemma holds. □

We consider Condition 2 in the following lemma.

Lemma 2. *The traffic of r_k will be processed by the VNF instances in SC_k in the specified order.*

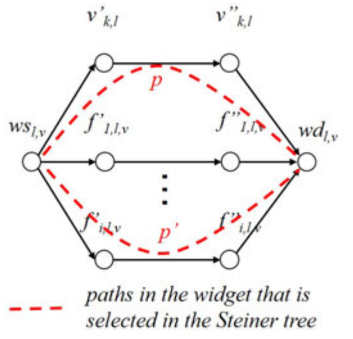


Fig. 6. A widget and its paths from its source to destination nodes that are selected in the Steiner tree.

Proof. Assume that the traffic of r_k is not processed by the specified order in SC_k. We have the following two cases: (1) two instances of the same VNF f_l processed the traffic, and (2) the traffic of r_k is processed by a previous VNF f_{l-1} after being processed by f_l.

For Case (1), the two instances must be in different cloudlets as shown in Lemma 1. This means that two widgets of the same VNF f_l is selected in the Steiner tree in G'. According to the construction of G' and Lemma 1, if the instances of f_l in two cloudlets are used, the source and destination nodes of the corresponding two widgets have to be included in the Steiner tree in G'; otherwise, the edges will be stand alone edges that can be removed from the Steiner tree. Therefore, according to the problem transformation method of the algorithm, this will correspond to the processing of r_k's traffic by two instances of f_l in different cloudlets, rather than a sequence processing of the two instances.

Case (2) can be dealt with similarly. Therefore, these two cases are not possible according to the construction of G'.

In addition, since each edge in G' may correspond to a shortest path in G, making the traffic being forwarded to a cloudlet more than once. this does not mean that the traffic is to be processed by the cloudlet twice. This is because we assume in such cloudlets will just forward the traffic instead of processing.

We thus conclude that the traffic of r_k will be processed by the VNFs in the specified order in SC_k. □

We now show Condition 3 as follows.

Lemma 3. *The traffic of r_k will be forwarded to its destinations in D_k after being processed by the instances of its VNFs in SC_k.*

Proof. In the construction of the auxiliary graph G', we can see that the destination nodes of the widgets for the last VNF f_{L_k} is connected to its corresponding switch node in the original network. For each $W_{L_k,k}$ of such widgets, if its edges are included in the Steiner tree, edge $\langle wd_{L_k,k}, v\rangle$ has to be included in the Steiner tree. The reasons include (1) this is the only edge to the destination nodes in D_k, and (2) as shown in Lemma 2, the traffic cannot be processed sequentially by other cloudlets of the same VNF f_{L_k} or the instances of its previous VNFs in SC_k. The lemma holds. □

Theorem 1. *Given an MEC network $G = (V, E)$ with a set V_{CL} of cloudlets and a multicast request r_k $(= (s_k, D_k; b_k, SC_k))$ that requires to transfer an amount b_k of data from its source to a set D_k of destinations and process its traffic by the VNFs in SC_k. There is an approximation algorithm, i.e.,* Algorithm 2, *for a special case of the NFV-enabled multicasting problem without delay requirements, which delivers a feasible solution that has an approximation ratio of $i(i-1)|D_k|^{1/i}$ [4], and the time complexity of $O((L_k \cdot |V| \cdot \frac{C_v}{C_{unit}(f_l)} + |V|)^i \cdot |D_k|^{2i})$, where L_k is the number of VNFs in the service chain SC_k of multicast request r_k, i.e., $L_k = |SC_k|$, and i is the level of the directed Steiner tree [4].*

Proof. From Lemmas 1, 2, and 3, we know that the solution obtained by finding a Steiner tree in G' is feasible. In the following, we analyze the approximation ratio and running time of the proposed approximation algorithm.

Assume c^* is the optimal solution for the NFV-enabled multicasting problem. In *Algorithm* 2, we find an approximate Steiner tree T' in the auxiliary graph G'. T' is then converted to routing paths for r_k in G by (1) selecting either an existing instance for a network function or a newly created instance of each VNF f_l in SC_k if the widget for f_l is included in the Steiner tree, and (2) replacing the edges between selected widgets using their corresponding shortest paths in G. In (1), the processing is determined according to which type of VNF instance is selected. In (2), the replaced auxiliary graph edge has the same weight as the total cost of its corresponding shortest path in G. Therefore, the cost do not change in the transfer from tree T' to the multicast tree T for multicast request r_k. Since the approximation ratio of the algorithm in [4] is $i(i-1)|D_k|^{1/i}$, the approximation of *Algorithm* 2 is $i(i-1)|D_k|^{1/i}$ as well.

We now show the time complexity of *Algorithm* 2. It can be seen that the most time consuming part of the algorithm is the finding of a Steiner tree in the auxiliary graph. The time complexity of Charikar's algorithm in auxiliary graph $G' = (V', E')$ is $O(|V'|^3)$ [21]. We can see that there are $O(\frac{C_v}{C_{unit}(f_l)})$ instances of VNF f_l in cloudlet $v \in V_{CL}$. According to the construction of the auxiliary graph, we thus have $O(\frac{C_v}{C_{unit}(f_l)} + 4) = O(\frac{C_v}{C_{unit}(f_l)})$ nodes for each widget. In total, we have $L_k \cdot |V_{CL}|$ widgets. Therefore, there are $O(L_k \cdot |V_{CL}| \cdot \frac{C_v}{C_{unit}(f_l)} + |V|)$ nodes in auxiliary graph G'. The time complexity thus is $O((L_k \cdot |V| \cdot \frac{C_v}{C_{unit}(f_l)} + |V|)^i \cdot |D_k|^{2i})$. □

We finally analyze the performance of Algorithm 1 the following theorem.

Theorem 2. *Given an MEC network $G = (V, E)$ with a set V_{CL} of cloudlets and a multicast request r_k $(= (s_k, D_k; b_k, SC_k))$ that requires to transfer an amount b_k of data from its source to a set D_k of destinations with an end-to-end delay requirement d_k^{req} and process its traffic by the VNFs in SC_k. There is a heuristic algorithm, i.e.,* Algorithm 1, *for the NFV-enabled multicasting problem for a single multicast request, which delivers a feasible solution in time $O(\lfloor \log V_{CL} + 1 \rfloor \cdot |V|^3 + (L_k \cdot |V| \cdot \frac{C_v}{C_{unit}(f_l)} + |V|)^i \cdot |D_k|^{2i})$, where L_k is the number of VNFs in the service chain SC_k of multicast request r_k, i.e., $L_k = |SC_k|$, and i is the level of the directed Steiner tree [4].*

Proof. We first show the solution feasibility of the proposed heuristic by showing that the end-to-end delay requirement of r_k is met. Algorithm 1 adopts a binary search based heuristic to find the proper number of cloudlets each multicast request r_k until the end-to-end delay requirement of r_k is met or it is rejected. Therefore, as long as the request is admitted, its end-to-end delay requirement is met.

We then analyze the time complexity of the proposed heuristic. Clearly, in the worse case, the binary search can make $\lfloor \log V_{CL} + 1 \rfloor$ iterations. Within each iteration, the most time consuming parts include (1) the identification of cloudlets that involved finding the delays from cloudlets to destinations in D_k via all pair shortest paths, which take $O(|V|^3)$ time, and (2) the assignment of VNFs one by one, taking $O(|SC_k|)$ time. In total, the time complexity of the proposed heuristic is $O(\lfloor \log V_{CL} + 1 \rfloor \cdot |V|^3 \cdot |SC_k| + (L_k \cdot |V| \cdot \frac{C_v}{C_{unit}(f_l)} + |V|)^i \cdot |D_k|^{2i}) = O(\lfloor \log |V| + 1 \rfloor \cdot |V|^3 + (L_k \cdot |V| \cdot \frac{C_v}{C_{unit}(f_l)} + |V|)^i \cdot |D_k|^{2i})$, assuming that $|SC_k|$ is a small constant. □

5 Algorithm for Admissions of a Set of NFV-Enabled Multicasting Requests

In this section, we consider a set of multicast request admissions. Given a set of NFV-enabled multicast request, we admit as many as requests in the set such that the weighted system throughput is maximized, while the accumulated implementation cost of all admitted requests is minimized, subject to computing capacities on cloudlets in an MEC.

5.1 Overview

Recall that we proposed both approximate and heuristic solutions for the NFV-enabled multicasting problem for the admission of a single multicast request, a simple method for the NFV-enabled multicasting problem is to consider algorithm `Heu_Delay` as a black-box and admit each request one by one invoking algorithm `Heu_Delay` iteratively. This method however may miss the opportunities of sharing VNFs among the requests, if the consecutively admitted requests do not have common VNFs in their service chains. Further, the constructed auxiliary graph G' in algorithm `Heu_Delay` for a request may no longer useful for the other. This consequently may lead to a prohibitively long time to make decisions of request admissions.

The basic idea behind the proposed algorithm is as follows. We observe that some requests have the same service chain requirements, and the VNFs in their service chains can be shared with high opportunities. Fig. 7 illustrates this idea, from which we can see that requests are classified into different *categories*, with each category having a set of requests that share a number of VNFs. Specifically, the algorithm first considers the category in which multicast requests the maximum number of common VNFs of their service chains. Then, the requests in this category, we start with the requests with smaller data traffic, and admit the requests one-by-one. This procedure continues until no more requests can be admitted in the category.

5.2 Heuristic Algorithm

We propose an efficient heuristic for the NFV-enabled multicasting problem for a set of requests with different service chain requirements, based on Algorithm 1.

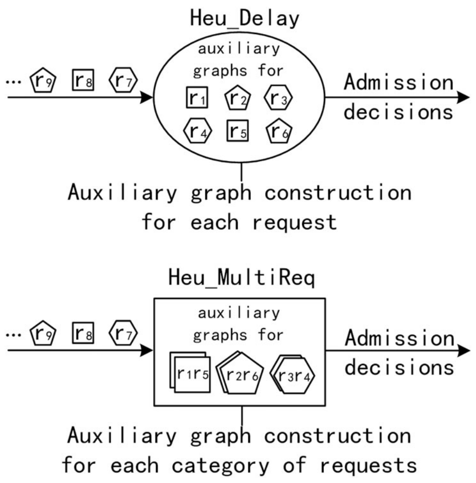


Fig. 7. The basic idea of the proposed heuristic for the NFV-enabled multicasting problem.

Specifically, the heuristic consists of a number of iterations within each iteration, a set of requests with the same number of VNFs in common are processed. Let L_{com} be the number of common VNFs that requests have in their service chains. Let L_{max} be the maximum length of the service chains of the requests in R, i.e., $L_{max} = \arg\max_{r_k \in R} |SC_k|$. Initially, $L_{com} = L_{max}$. It decreases by one in each iteration of the algorithm until $L_{com} = 0$.

Within each iteration, we first find the requests that have L_{com} VNFs in common of their service chains. Denote by $R(L_{com})$ the set of such requests. We then rank the requests in $R(L_{com})$ in increasing order of their data traffic. For each request $r_k \in R(L_{com})$, we invoke the proposed approximation algorithm in 2. Notice that the requests in $R(L_{com})$ may have different source nodes and different destination sets. We thus need to adjust the auxiliary graph after the admission of each multicast request, by removing the source node for the previous request, and add the source node of the current request. This means that, before admitting the next multicast request r_{k+1}, we make adjustments of the constructed auxiliary graph G' instead of constructing a new one. Specifically, the widgets that are built for the L_{com} VNFs are updated accordingly, if multicast request r_k is admitted. Also, the widgets for the VNFs that are not among the L_{com} of request r_{k+1} is added to the auxiliary graph. This iteration continues until no more requests can be admitted within this category. The steps of this algorithm are detailed in Algorithm 3.

We now analyze the feasibility of *Algorithm* `Heu_MultiReq` in the following theorem.

Theorem 3. *Given an MEC network $G = (V, E)$ with a set V_{CL} of cloudlets, a set R of NFV-enabled multicast requests with each multicast request r_k $(= (s_k, D_k; b_k, SC_k))$ that requires to transfer an amount b_k of data from its source to a set D_k of destinations with an end-to-end delay requirement d_k^{req} and process its traffic by the VNFs in SC_k. There is an efficient algorithm, Algorithm 3, for the NFV-enabled multicasting problem.*

Proof. To show the solution delivered by Algorithm 3 is feasible, we need to show the classification of requests does not affect the solution feasibility of Algorithm 2. Assume that the algorithm currently considers request r_{k+1}. If its previous request r_k is admitted, the widgets of the corresponding cloudlets that implement the VNFs of r_k are then updated, since the resource availabilities of these cloudlets or statuses of their existing VNF instances changed. Otherwise, there is not any change of the widgets in the auxiliary graph. Considering that the feasibility of admitting one request by Algorithm 2 can be shown by Lemma 2, Algorithm 1 delivers a feasible solution when multiple requests are considered. □

Algorithm 3. `Heu_MultiReq`

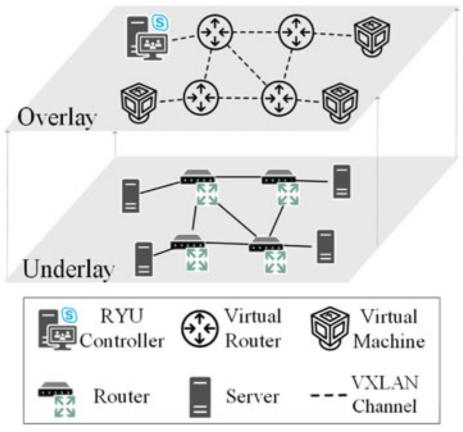
```
Input: $G = (V, E)$, $V_{CL}$, $C_e$ for each $e \in E$, $C_v$ for each $v \in V_{CL}$,
    and a set of multicast requests with each multicast request
    being denoted by $r_k = (s_k, D_k; b_k, SC_k)$.
Output: The system throughput achieved by the admitted
    requests in $R$.
1:  $N_{ad} \leftarrow 0$;
2:  for $L_{com} \leftarrow 0, 1 \ldots, L_{max}$ do
3:      Find the maximum number of requests in $R$ that have
        $L_{com}$ common VNFs in their service chains, and let
        $R(L_{com})$ be the set of requests;
4:      Rank the multicast requests in $R(L_{com})$ according to their
        data traffic;
5:      for each request $r_k \in R(L_{com})$ do
6:          $T \leftarrow \emptyset$;
7:          while $G$ is ($s_k$-$D_k$)-connected OR $r_k$ is admitted do
8:              Construct auxiliary graph $G' = (V', E')$, by creating
                $L_k \cdot |V_{CL}|$ widgets, adding all the switch nodes in $V$
                of the original network $G$, and interconnecting the
                added nodes as shown in Fig. 5, or adjust the auxil-
                iary graph if it is already constructed in the admis-
                sion of previous requests;
9:              Find a Steiner tree $T$ for in auxiliary graph $G'$;
10:             if the delay of each branch of $T$ is smaller than $d_k^{req}$
                then
11:                 Admit multicast request $r_k$;
12:             else
13:                 Find the branches of $T$ that violate delay require-
                    ment $d_k^{req}$;
14:                 For each of such found branch, identify an edge
                    with the maximum delay;
15:                 Remove the identified edges from graph $G$;
16:         if $T \neq \emptyset$ then
17:             For each path from the widget source node to the
                widget destination node of a widget in $T$, condense
                the path to a single node;
18:         The widgets that are built for the $L_{com}$ VNFs are
            updated according to the resource availabilities after
            admitting $r_k$;
19:         if $k + 1 < |R(L_{com})|$ then
20:             The widgets for the VNFs that are not among the
                $L_{com}$ of request $r_{k+1}$ is added to the auxiliary graph;
```

6 Performance Evaluation

In this section we evaluate the performance of the proposed algorithms in a real testbed.

6.1 Test-Bed Setup

We build a test-bed consisting of both an underlay network with hardware switches and an overlay network with virtual switches, as shown in Fig. 8. The physical underlay consists of five H3C S5560X-30S-EI switches [12], with the support for VXLAN for virtual tunnel building and SDN capabilities. It has also one server with E5 Gold 5218 CPU, 128G RAM and four PCs with i7-8700 CPU, 16G RAM. Netconf and SNMP protocols are used to manage the switches and the links that interconnect them [3], [35]. We considered a design approach that uses the VXLAN functionality provided by the switch, where VXLAN is a widely used overlay technology [37]. The H3C S5560X-30S-EI switch implements a VXLAN tunnel based on hardware, which can greatly improve performance compared to traditional methods. The overlay mechanism provides connectivity within, and potentially across multiple testbed sites as it can transit any routed layer-3 underlay. We use VXLAN as a point-to-point tunneling mechanism (VXLAN VNI identifies a single link between two nodes [37]). SDN-capable switches can also perform encapsulation and decapsulation of VXLAN tunnels, each tunnel corresponds to a port in the switch. Using VXLAN, we build an overlay network with a number of Open vSwitch (OVS) [36] nodes and VMs. The overlay network is built following the topology generated using a graph generation tool GT-ITM [10] and the real network topologies AS1755, AS4755. Its OVS nodes and VMs are controlled by a Ryu [40] controller. The proposed algorithms are implemented as Ryu applications.



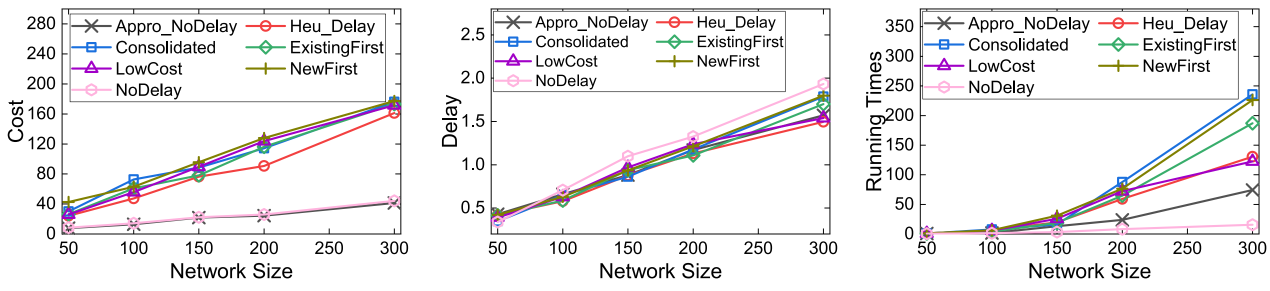
(a) The underlay and overlay of the test-bed.

(b) The hardware switches and servers.

Fig. 8. A test-bed with both hardware switches and virtual resources.

6.2 Environment Settings

We consider an MEC network consisting of the number of nodes from 50 to 250. The number of servers in each network is set to 10 percent of the network size, and the servers are randomly co-located with the switches. We also use real network topologies, i.e.,the GÉANT [9] and an ISP network from [42]. There are nine cloudlets for the GÉANT topology as set in [11] and the number of data centers in the ISP networks are provided by [38]. The computing capacity of cloudlet varies from 40,000 to 120,000 MHz [13] (cloudlets with around tens of servers). Five types of network functions, i.e., Firewall, Proxy, NAT, IDS, and Load Balancing, are considered, and their computing demands are adopted from [11], [32]. The source and destination nodes of each



(a) Average cost of implementing a multicast request. (b) Average delay experienced by a multicast request. (c) Running times.

Fig. 9. The performance of algorithms `Appro_NoDelay`, `Consolidated`, `NoDelay`, `ExistingFirst`, `NewFirst`, and `LowCost`.

multicast request is randomly generated, *the ratio* of the maximum number D_{max} of destinations of a multicast request to the network size $|V|$ is randomly drawn in the range of $[0.05, 0.2]$. The data of each request is randomly drawn from $[10, 200]$ Megabyte, and the delay requirement of transferring such data is randomly generated from $[0.05, 5]$ *seconds*. Notice that the transfer of larger amount of data can be divided into smaller amounts and transferred by multiple multicast requests. Unless otherwise specified, these parameters will be adopted in the default setting.

We compare the performance of the proposed approximation and heuristic algorithms against the following benchmarks.

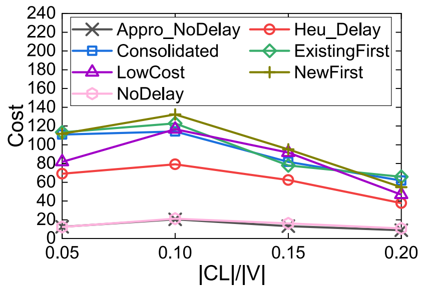
- We consider the case where the VNFs of each multicast request may be placed to multiple cloudlets for processing while there exist solutions that consolidate all VNFs of a multicast request into a single location. We thus compare our solutions with such a solution, which is referred to as algorithm `Consolidated`.
- We evaluate the performance of the proposed approximation and heuristic algorithms against the one in [39] that does not consider the delay requirement of multicast requests, and we use `NoDelay` to represent the algorithm.
- We also compare the performance of our algorithm against that of a greedy solution that prefers to select existing VNF instances for each multicast request r_k. Specifically, it finds the cloudlet that is the closest to source node s_k and has an VNF instance for its first VNF in SC_k, if there does not exist such cloudlets, a new VNF instance is created in the closest cloudlet. The procedure continues until all VNFs in SC_k are considered. This greedy algorithm is referred to as algorithm `ExistingFirst`.
- Another greedy benchmark prefers to create new instances for each of the VNFs in SC_k, which is referred to as algorithm `NewFirst`.
- The fifth benchmark selects the cloudlet that can achieve the lowest processing cost for each VNF in SC_k. For simplicity, it is referred to as algorithm `LowCost`. Specifically, algorithm `LowCost` finds the cloudlet that is the closest to the source s_k and then places as many VNFs in SC_k to the cloudlet until all existing VNF instances are used or no computing resource available to instantiate new ones. If there are still VNFs in SC_k that have not been assigned, it finds the next cloudlet that is the closest to the found cloudlets.

6.3 Performance Evaluation of Algorithms `Heu_Delay` and `Appro_NoDelay`

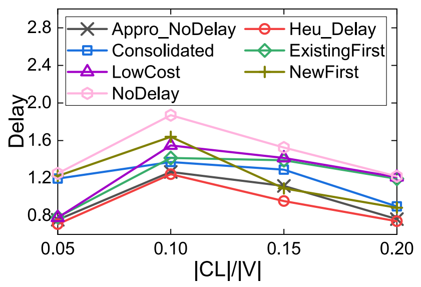
We first evaluate the performance of algorithms `Heu_Delay` and `Appro_NoDelay` against that of algorithms `Consolidated`, `NoDelay`, `ExistingFirst`, `NewFirst`, and `LowCost`, in terms of the average operational cost, the average end-to-end delay, and the running time, by varying the network size from 50 to 250 while fixing the number of requests at 100. Fig. 9 shows the results of the proposed algorithms.

From Fig. 9a, we can see that Algorithm `Heu_Delay` achieves a lower operational cost than these of algorithms `ExistingFirst`, `NewFirst`, and `LowCost`. The reason is that Algorithm `Heu_Delay` jointly considers existing VNF instances and newly instantiated ones. However, the greedy approaches `NewFirst`, `ExistingFirst`, and `LowCost` only prefer new, existing, or low processing cost VNF instances. They unfortunately could miss the opportunities of further reducing the operational cost. Specifically, if the use of existing VNF instances can save the processing cost, `NewFirst` has a higher cost due to creating new instances. Also, there are some cases when creating new VNF instances can save transmission costs, which can be missed by algorithm `ExistingFirst`. In addition, it can be seen from Fig. 9a that Algorithm `Heu_Delay` has a higher operational cost than algorithms `Appro_NoDelay` and `NoDelay`. This is because algorithms `Appro_NoDelay` and `NoDelay` do not consider the delay requirement of requests, making it choose cloudlets with lower operational costs.

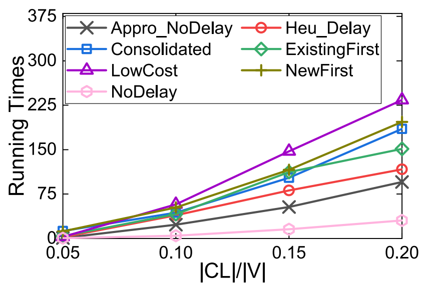
As shown in Fig. 9b, the average delay experienced by each multicast request by Algorithm `Heu_Delay` is much lower than its comparison counterparts. The reason is that Algorithm `Heu_Delay` carefully finds a trade-off between the delay and cost of implementing a NFV-enabled request. Also, from Fig. 9c, we can see that the running time of Algorithm `Heu_Delay` is around 50 seconds for network size 200, which is slightly larger than those of algorithms `Appro_NoDelay` and `NoDelay` and smaller than algorithms `ExistingFirst`, `NewFirst`, and `LowCost`. The reason is that `Heu_Delay` has an additional process of binary search to find a proper number of cloudlets for each request r_k. Algorithm `NoDelay` has a lower running time compared with algorithm `Appro_Delay` because the delay requirement of requests is not considered, which reduces the solution space.



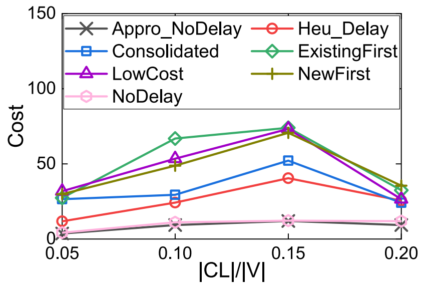
(a) Average cost of implementing a multicast request in network AS1755.



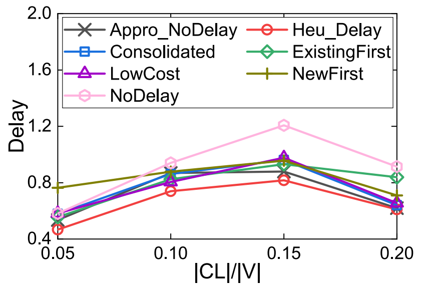
(b) Average delay experienced by a multicast request in network AS1755.



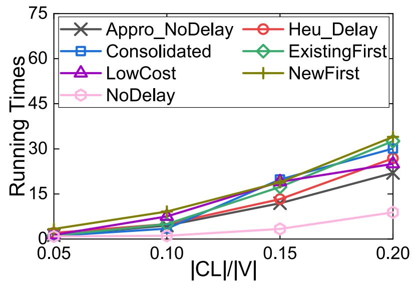
(c) Running times in network AS1755.



(d) Average cost of implementing a multicast request in network AS4755.

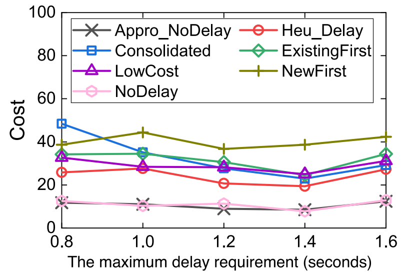


(e) Average delay experienced by a multicast request in network AS4755.

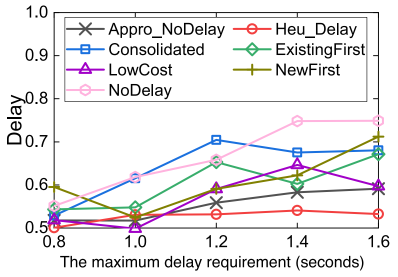


(f) Running times in network AS4755.

Fig. 10. The performance of algorithms `Appro_NoDelay`, `Consolidated`, `NoDelay`, `ExistingFirst`, `NewFirst`, and `LowCost` in networks AS1755 and AS4755.



(a) Average cost of implementing a multicast request.



(b) Average delay experienced by a multicast request.

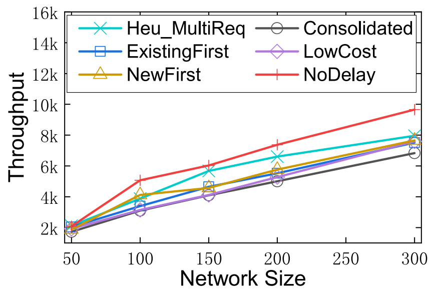
Fig. 11. The impact of the maximum delay requirement of each multicast request on the performance of algorithms `Appro_NoDelay`, `Consolidated`, `NoDelay`, `ExistingFirst`, `NewFirst`, and `LowCost`.

We then evaluate the performance of algorithms `Heu_Delay` and `Appro_NoDelay` against that of algorithms `Consolidated`, `NoDelay`, `ExistingFirst`, `NewFirst`, and `LowCost`, in real networks AS1755 and AS4755, by varying the ratio of the number of cloudlets to the number of switches, i.e., $|\mathcal{CL}|/|V|$ from 0.05 to 0.2. Fig. 10 illustrates the results. Figs. 10a and 10d show that algorithms `Heu_Delay` and `Appro_NoDelay` achieve lower operational costs than algorithms `Consolidated`, `ExistingFirst`, and `NewFirst`, while algorithms `Appro_NoDelay` and `NoDelay` has the highest delay. We can also see that the average cost of implementing a multicast increases first when the ratio $|\mathcal{CL}|/|V|$ increases from 0.05 to 0.1 and then decreases afterwards. The rationale behind is that VNFs of each multicast request may be assigned to more cloudlets with the increase of number of cloudlets, thereby pushing up the transmission cost from its source to the cloudlets and from the cloudlets to its destinations. However, with the further increase of cloudlets, it is more likely that these cloudlets are deployed in locations that are close to the source and destinations of the multicast request. The transmission cost then can be reduced afterwards.

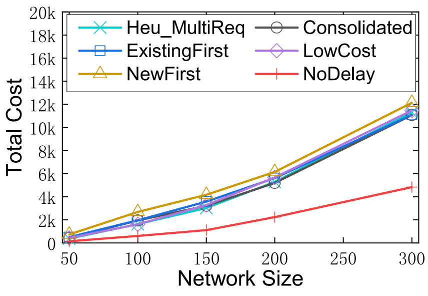
We then investigate the impact of the maximum delay requirement on algorithm performance in the real network AS1755, by varying the maximum delay requirement of each multicast request from 0.8 seconds to 1.8 seconds with an increment of 0.2 seconds. Fig. 11 illustrates that the cost of implementing a multicast request is decreasing with the increase of the maximum delay requirement. The rationale behind is that a higher delay requirement of a request allows the algorithm to select cloudlets with lower costs but further from the source node of the request. Obviously, the experienced delay will be higher, as shown in Fig. 11.

6.4 Performance Evaluation of Algorithm `Heu_MultiReq`

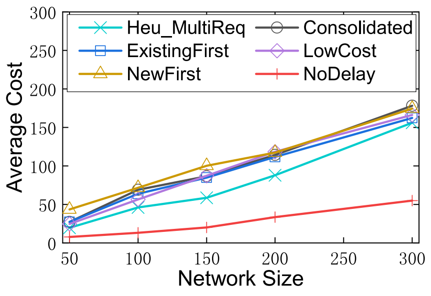
We now compare the performance of Algorithm `Heu_MultiReq` against that of algorithms `Consolidated`, `NoDelay`,



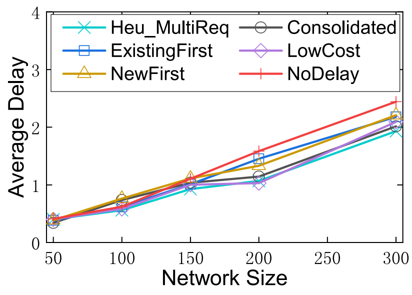
(a) System throughput.



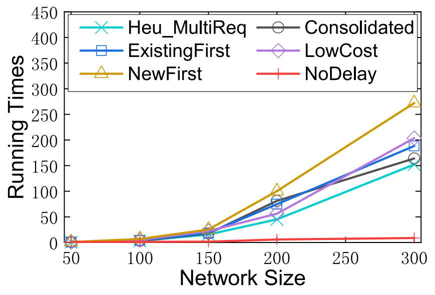
(b) Total cost of implementing a multicast request.



(c) Average cost of implementing a multicast request.

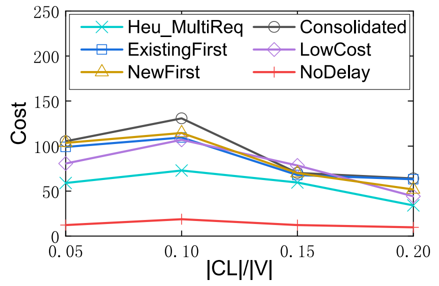


(d) Average delay experienced by a multicast request.

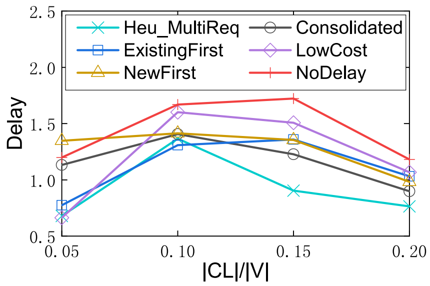


(e) Running times.

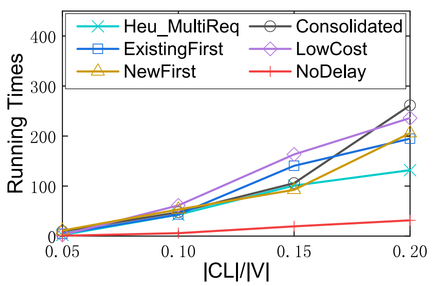
Fig. 12. The performance of algorithms `Heu_Multicast`, `Consolidated`, `NoDelay`, `ExistingFirst`, `NewFirst`, and `LowCost`.



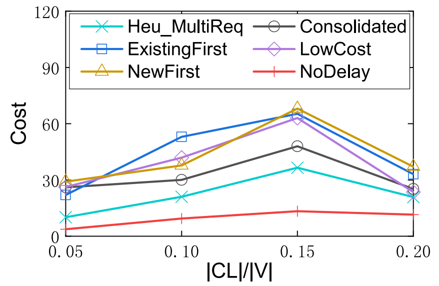
(a) Average cost of implementing a multicast request.



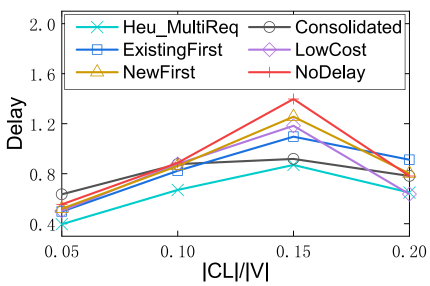
(b) Average delay experienced by a multicast request.



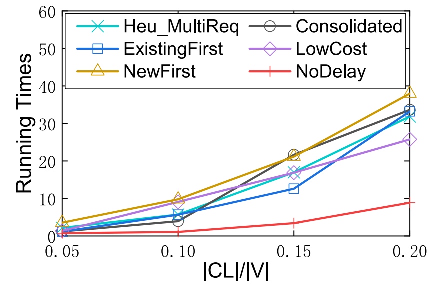
(c) Running times.



(d) Average cost of implementing a multicast request.



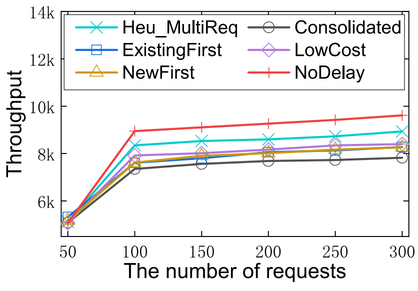
(e) Average delay experienced by a multicast request.



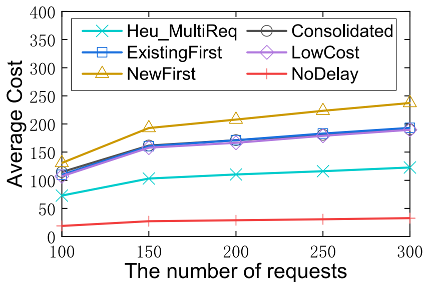
(f) Running times.

Fig. 13. The performance of algorithms `Heu_Multicast`, `Consolidated`, `NoDelay`, `ExistingFirst`, `NewFirst`, and `LowCost`.

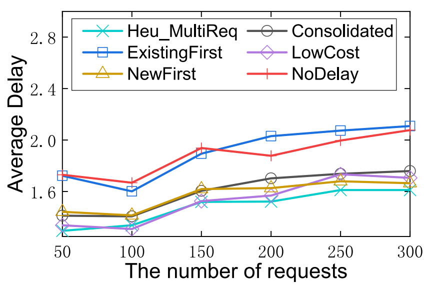
`ExistingFirst`, `NewFirst`, and `LowCost`, in terms of the system throughput, the total operational cost, the average end-to-end delay, and the running time, by varying the network size from 50 to 250 and fixing the number of requests to 100. Results are shown in Fig. 12, from which we can see that Algorithm `Heu_MultiReq` achieves around 30, 30, 35 percent higher system throughput than algorithms `ExistingFirst`, `NewFirst`, `LowCost`, and `Consolidated` when the network size is 200. The rationale behind is that algorithms `ExistingFirst`, `NewFirst`, and `LowCost` prefer existing, newly instantiated, and low processing cost VNF instances for each multicast request, and the cloudlets for those VNF instances may not have sufficient computing resource to implement the request, thereby leading to its rejection. Further, from Figs. 12a and 12b, it can be seen Algorithm `NoDelay` has a higher end-to-end delay than that of Algorithm



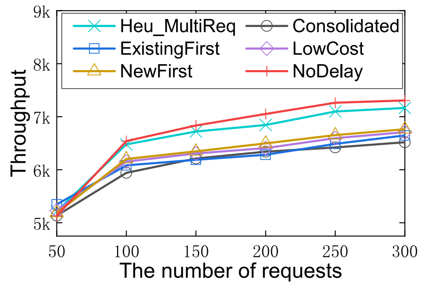
(a) System throughput in network AS1755



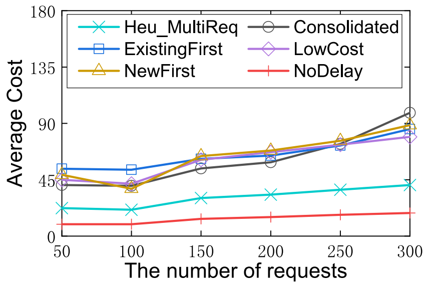
(b) Average cost in network AS1755



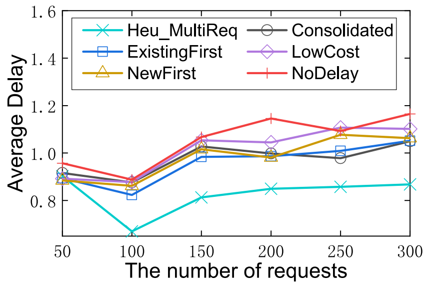
(c) Average delay in network AS1755



(d) System throughput in network AS4755



(e) Average cost in network AS4755



(f) Average delay in network AS4755

Fig. 14. The performance of algorithms `Heu_Multicast`, `Consolidated`, `NoDelay`, `ExistingFirst`, `NewFirst`, and `LowCost`.

`Heu_MultiReq`, although it delivers a slight higher system throughput. Similar results can be observed from Fig. 13 when the performance of Algorithm `Heu_MultiReq` is evaluated against that of algorithms `Consolidated`, `NoDelay`, `ExistingFirst`, `NewFirst`, and `LowCost`, in real networks AS1755 and AS4755.

We then investigate the impact of the number of requests on the performance of algorithms `Heu_MultiReq`, `Consolidated`, `NoDelay`, `ExistingFirst`, `NewFirst`, and `LowCost`, in terms of system throughput, average operational cost, average end-to-end delay, and running time, by varying the number of requests from 50 to 300 while fixing the network size to 100. Fig. 14 shows that the system throughput increases first with the growth on the number of requests from 50 to 100, and then keeps stable afterwards, because the cloudlet capacities are saturated. We can also see that the average cost of implementing a multicast increases with the growth of request number. The rationale behind is that each multicast request may be assigned to more cloudlets for processing with the increase of number of requests, considering that the resources in cloudlets are saturated and may not be enough to implement all VNFs of a service chain. This eventually increases the transmission cost for each multicast request.

7 Conclusion and Future Work

In this paper, we study the problem of delay-aware, NFV-enabled multicasting in a mobile edge cloud network, by exploring the sharing of VNF instances of requests. If cloudlets have sufficient computing resource to process traffic of a multicast request, with no delay requirement, we proposed an approximate solution with a provable approximation ratio; otherwise, we developed an efficient heuristic. We also considered a set of NFV-enable multicast request admissions with the aim to maximize the weighted system throughput, for which we proposed an efficient heuristic. We finally evaluate the performance of the proposed algorithms against state-of-the-arts approaches in a real testbed, and the results show that the performance of our algorithms is promising.

In this paper we considered the sharing of idle VNFs that have been released by other requests. The requests with the same service chain requirements may share resources with high probability. However, requests may have dynamic resource demands, and may share resources with others as long as they have complimentary demands. Understanding how to learn such dynamic complimentary resource demands among requests is challenging. Therefore, we consider the adoption of machine learning methods to classify requests with complimentary demands as our future research study – akin to existing efforts in *interference-aware* scheduling in cloud-based data centers. Existing efforts that make use of an interference index to characterize these competing/ complementary workloads can also be utilized in the proposed environment. Another is to explore the dynamic admission of NFV-enabled delay-aware requests, taking account of uncertainty (variability) of processing and transmission delays. The admission of requests in the current time slot can impact the admission of future requests. Understanding how online learning algorithms can adapt to support such admission control remains another potential research topic.

Acknowledgments

The authors would like to thank the three anonymous referees and the associate editor for their expertise comments and constructive suggestions, which have helped them improve the quality and presentation of this article greatly. The work of Zichuan Xu, Qiufen Xia, and Guowei Wu was supported in part by the National Natural Science Foundation of China (Grant No. 61802048 and 61802047), the fundamental research funds for the central universities in China (Grant No. DUT17RC(3)061, DUT17RC(3)070, DUT19RC(4)035, and DUT19GJ204), and the "Xinghai Scholar Program" in Dalian University of Technology, China. The work of Weifa Liang was supported by the Australian Research Council Discovery Project (Grant No. DP200101985). The work of Pan Zhou was supported by the National Natural Science Foundation of China (Grant No. 61972448).

REFERENCES

[1] O. Alhussein *et al.*, "Joint VNF placement and multicast traffic routing in 5G core networks," in *Proc. IEEE Global Commun. Conf.*, 2018, pp. 1–6.

[2] S. M. Banik, S. Radhakrishnan, and C. N. Sekharan, "Multicast routing with delay and delay variation constraints for collaborative applications on overlay networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 3, pp. 421–431, Mar. 2007.

[3] J. Case *et al.*, "A simple network management protocol (SNMP)," RFC 1098, IETF, 1990. [Online]. Available: https://tools.ietf.org/html/rfc1157

[4] M. Charikar *et al.*, "Approximation algorithms for directed Steiner problems," in *Proc. 9th Annu. ACM-SIAM Symp. Discrete Algorithms*, 1998, pp. 192–200.

[5] Y. Chen and J. Wu, "NFV middlebox placement with balanced set-up cost and bandwidth consumption," in *Proc. 47th Int. Conf. Parallel Process.*, 2018, Art. no. 14.

[6] R. Cohen, L. Eytan, J. Naor, and D. Raz, "Near optimal placement of virtual network functions," in *Proc. IEEE Conf. Comput. Commun.*, 2015, pp. 1346–1354.

[7] R. Cziva, C. Anagnostopoulos, and D. P. Pezaros, "Dynamic latency-optimal vNF placement at the network edge," in *Proc. IEEE Conf. Comput. Commun.*, 2018, pp. 693–701.

[8] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA, USA: Freeman, 1979.

[9] GÉANT. 2000. Accessed: Feb. 2020. [Online]. Available: http://www.geant.net

[10] E. W. Zegura, K. Calvert, and S. Bhattacharjee, "How to model an internetwork," in *Proc. IEEE INFOCOM*, 1996, pp. 594–602.

[11] A. Gushchin, A. Walid, and A. Tang, "Scalable routing in SDN-enabled networks with consolidated middleboxes," in *Proc. ACM SIGCOMM Workshop Hot Topics Middleboxes Netw. Function Virtualization*, 2015, pp. 55–60.

[12] H3C SDN Switches. 2019. Accessed: Feb. 2020. [Online]. Available: http://www.h3c.com/en/Product_Technology/Enterprise_Products/Switches/Campus_Switches/H3C_S5560X-EI/

[13] Hewlett-Packard Development Company, "L. P. servers for enterprise C bladeSystem, rack & tower and hyperscale," 2015. [Online]. Available: http://www8.hp.com/us/en/products/servers/

[14] K. Han, Y. Liu, and J. Luo, "Duty-cycle-aware minimum-energy multicasting in wireless sensor networks," *IEEE/ACM Trans. Netw.*, vol. 21, no. 3, pp. 910–923, Jun. 2013.

[15] H. Huang, S. Guo, J. Wu, and J. Li, "Service chaining for hybrid network function," *IEEE Trans. Cloud Comput.*, vol. 7, no. 4, pp. 1082–1094, Fourth Quarter 2019.

[16] H. Huang, P. Li, and S. Guo, "Traffic scheduling for deep packet inspection in software-defined networks," *Concurrency Comput., Practice Experience*, vol. 29, no. 16, 2016, Art. no. e3967.

[17] L. Huang, H. Hung, C. Lin, and D. Yang, "Scalable Steiner tree for multicast communications in software-defined networking," *CoRR*, vol. abs/1404.3454, 2014. [Online]. Available: http://arxiv.org/abs/1404.3454

[18] M. Huang, W. Liang, Z. Xu, W. Xu, S. Guo, and Y. Xu, "Dynamic routing for network throughput maximization in software-defined networks," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun.*, 2016, pp. 1–9.

[19] N. Kiji, T. Sato, R. Shinkuma, and E. Oki, "Virtual network function placement and routing model for multicast service chaining based on merging multiple service paths," in *Proc. IEEE 20th Int. Conf. High Perform. Switching Routing*, 2019, pp. 1–6.

[20] S. Knight *et al.*, "The internet topology zoo," *J. Sel. Areas Commun.*, vol. 29, pp. 1765–1775, 2011.

[21] L. Kou, G. Markowsy, and L. Berman, "A faster algorithm for Steiner trees," *Acta Informatica*, vol. 15, pp. 141–145, 1981.

[22] T.-W. Kuo, B.-H. Liou, K. C. Lin, and M.-J. Tsai, "Deploying chains of virtual network functions: On the relation between link and server usage," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun.*, 2016, pp. 1–9.

[23] Y. Li, L. T. X. Phan, and B. T. Loo, "Network functions virtualization with soft real-time guarantees," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun.*, 2016, pp. 1–9.

[24] D. Li *et al.*, "Reliable multicast in data center networks," *IEEE Trans. Comput.*, vol. 63, no. 8, pp. 2011–2024, Aug. 2014.

[25] W. Liang, "Approximate minimum-energy multicasting in wireless ad hoc networks," *IEEE Trans. Mobile Comput.*, vol. 5, no. 4, pp. 377–387, Apr. 2006.

[26] D. H. Lorenz and D. Raz, "A simple efficient approximation scheme for the restricted shortest path problem," *Operations Res. Lett.*, vol. 28, pp. 213–219, 2001.

[27] T. Lukovszki and S. Schmid, "Online admission control and embedding of service chains," in *Proc. Int. Colloq. Structural Inf. Commun. Complexity*, 2015, pp. 104–118.

[28] L. Mamatas, S. Clayman, and A. Galis, "A service-aware virtualized software-defined infrastructure," *IEEE Commun. Mag.*, vol. 53, no. 4, pp. 166–174, Apr. 2015.

[29] Y. Ma, W. Liang, Z. Xu, and S. Guo, "Profit maximization for admitting requests with network function services in distributed clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 5, pp. 1143–1157, May 2019.

[30] Y. Ma, W. Liang, J. Wu, and Z. Xu, "Throughput maximization of NFV-enabled multicasting in mobile edge cloud networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 2, pp. 393–407, Feb. 2020.

[31] Y. Ma, W. Liang, and J. Wu, "Online NFV-enabled multicasting in mobile edge cloud networks," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 821–830.

[32] J. Martins *et al.*, "ClickOS and the art of network function virtualization," in *Proc. 11th USENIX Conf. Netw. Syst. Des. Implementation*, 2014, pp. 459–473.

[33] H. Moens and F. D. Turck, "VNF-P: A model for efficient placement of virtualized network functions," in *Proc. 10th Int. Conf. Netw. Service Manage. Workshop*, 2014, pp. 418–423.

[34] M. Mongioví, A. K. Singh, X. Yan, B. Zong, and K. Psounis, "Efficient multicasting for delay tolerant networks using graph indexing," in *Proc. IEEE INFOCOM*, 2012, pp. 1386–1394.

[35] Netconf Working Group. 2018. [Online]. Available: https://datatracker.ietf.org/wg/netconf/about/

[36] Open vSwtich. 2016. [Online]. Available: https://www.openvswitch.org

[37] M. Mahalingam *et al.*, "Virtual extensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks," RFC 7348, IETF. 2014. [Online]. Available: https://tools.ietf.org/html/rfc7348

[38] Z. A. Qazi, C. C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying middlebox policy enforcement using SDN," in *Proc. ACM SIGCOMM Conf.*, 2013, pp. 27–38.

[39] B. Ren, D. Guo, G. Tang, X. Lin, and Y. Qin, "Optimal service function tree embedding for NFV Enabled multicast," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 132–142.

[40] Ryu SDN Controller. 2017. [Online]. Available: https://osrg.github.io/ryu/

[41] H. Soni, W. Dabbous, T. Turletti, and H. Asaeda, "NFV-based scalable guaranteed-bandwidth multicast service for software-defined ISP networks," *IEEE Trans. Netw. Service Manag.*, vol. 14, no. 5, pp. 1157–1170, Dec. 2017.

[42] N. Spring, R. Mahajan, and D. Wetherall, "Measuring ISP topologies with rocketfuel," in *Proc. ACM SIGCOMM Conf.*, 2002, pp. 133–145.

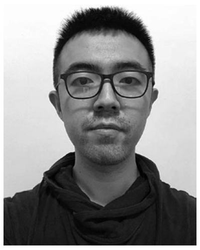
[43] J. M. Vella and S. Zammit, "A survey of multicasting over wireless access networks," *IEEE Commun. Surveys Tuts.*, vol. 15, no. 2, pp. 718–753, Second Quarter 2013.

[44] K. Xie, X. Zhou, T. Semong, and S. He, "Multi-source multicast routing with QoS constraints in network function virtualization," in *Proc. IEEE Int. Conf. Commun.*, 2019, pp. 1–6.

[45] Z. Xu, W. Liang, M. Jia, M. Huang, and G. Mao, "Task offloading with network function services in a mobile edge-cloud network," *IEEE Trans. Mobile Comput.*, vol. 18, no. 11, pp. 2672–2685, Nov. 2019.

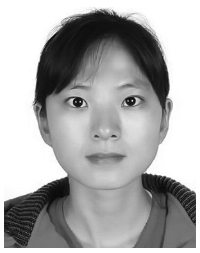
[46] Z. Xu, W. Liang, A. Galis, and Y. Ma, "Throughput maximization and resource optimization in NFV-enabled networks," in *Proc. IEEE Int. Conf. Commun.*, 2017, pp. 1–7.

[47] Z. Xu, W. Liang, M. Huang, M. Jia, S. Guo, and A. Galis, "Approximation and online algorithms for NFV-enabled multicasting in SDNs," in *Proc 37th IEEE Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 625–634.
[48] Z. Xu, W. Liang, M. Huang, M. Jia, S. Guo, and A. Galis, "Efficient NFV-enabled multicasting in SDNs," *IEEE Trans. Commun.*, vol. 67, no. 3, pp. 2052–2070, Mar. 2019.
[49] B. Yi, X. Wang, M. Huang, and A. Dong, "A multi-stage solution for NFV-enabled multicast over the hybrid infrastructure," *IEEE Commun. Lett.*, vol. 21, no. 9, pp. 2061–2064, Sep. 2017.
[50] Y. Zhang *et al.*, "StEERING: A software-defined networking for inline service chaining," in *Proc. 21st IEEE Int. Conf. Netw. Protocols*, 2013, pp. 1–10.
[51] S. Q. Zhang, Q. Zhang, H. Bannazadeh, and A. L. Garcia, "Network function virtualization enabled multicast routing on SDN," in *Proc. IEEE Int. Conf. Commun.*, 2015, pp. 5595–5601.


Haozhe Ren received the BSc degree from the University of Science and Technology Beijing, Beijing, China, in 2012, and the ME degree from the Xinjiang Normal University, Ürümqi, China, in 2018. He is currently working toward the PhD degree in the School of Software, Dalian University of Technology, Dalian, China. His current research interests include network function virtualization, software-defined networking, algorithmic game theory, and optimization problems.

Zichuan Xu (Member, IEEE) received the BSc and ME degrees from the Dalian University of Technology, Dalian, China, in 2008 and 2011, respectively and the PhD degree from the Australian National University, Canberra, Australia, in 2016, all in computer science. From 2016 to 2017, he was a research associate with the Department of Electronic and Electrical Engineering, University College London, United Kingdom. He is currently an associate professor with the School of Software, Dalian University of Technology. He is also a 'Xinghai Scholar' with the Dalian University of Technology. His research interests include cloud computing, network function virtualization, software-defined networking, wireless sensor networks, routing protocol design for wireless networks, algorithmic game theory, and optimization problems.

Weifa Liang (Senior Member, IEEE) received the BSc degree from Wuhan University, Wuhan, China, in 1984, the ME degree from the University of Science and Technology of China, Hefei, China, in 1989, and the PhD degree from the Australian National University, Canberra, Australia, in 1998, all in computer science. He is currently a full professor with the Research School of Computer Science, Australian National University. His research interests include design and analysis of energy efficient routing protocols for wireless ad hoc and sensor networks, mobile edge computing and cloud computing, network function virtualization, software-defined networking, design and analysis of parallel and distributed algorithms, approximation algorithms, combinatorial optimization, and graph theory.


Qiufen Xia (Member, IEEE) received the BSc and ME degrees from the Dalian University of Technology, Dalian, China, in 2009 and 2012, respectively and the PhD degree from the Australian National University, Canberra, Australia, in 2017, all in computer science. She is currently a lecturer with the Dalian University of Technology. Her research interests include mobile cloud computing, query evaluation, big data analytics, big data management in distributed clouds, and cloud computing.

Pan Zhou (Member, IEEE) received the BS degree from the Advanced Class of Huazhong University of Science and Technology (HUST), Wuhan, China, in 2006, and the PhD degree from the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, Georgia, in 2011. He is currently an associate professor with the School of Electronic Information and Communications, HUST, Wuhan, China. He was a senior technical member with Oracle, Inc., America, from 2011 to 2013, Boston, Massachusetts. His current research interests include security and privacy, machine learning and big data analytics, and information networks.

Omer F. Rana (Senior Member, IEEE) received the BS degree in information systems engineering from the Imperial College of Science, Technology and Medicine, London, United Kingdom, the MS degree in microelectronics systems design from the University of Southampton, Southampton, United Kingdom, and the PhD degree in neural computing and parallel architectures from the Imperial College of Science, Technology and Medicine, London, United Kingdom. He is a professor of performance engineering with Cardiff University, Cardiff, United Kingdom. His current research interests include problem solving environments for computational science and commercial computing, data analysis and management for large-scale computing, and scalability in high performance agent systems.

Alex Galis (Senior Member, IEEE) is currently a professor in networked and service systems with the University College London. He has co-authored 10 research books and more that 250 publications in the Future Internet areas: System management, networks and services, networking clouds, 5G virtualisation, and programmability. He was a member of the Steering Group of the Future Internet Assembly (FIA) and he led the Management and Service-aware Networking Architecture (MANA) Working Group. He acted as TPC chair of 14 IEEE conferences. He is also a co-editor of the *IEEE Communications Magazine* feature topic on Advances in Networking Software. He acted as a vice chair of the ITU-T SG13 Group on Future Networking. He is involved in IETF and ITU-T SG13 network slicing activities and he is also involved in IEEE SDN initiative.

Guowei Wu received the PhD degree from Harbin Engineering University, Harbin, China, in 2003. He is currently a professor with the School of Software, Dalian University of Technology (DUT) in China. His research interests include embedded real-time system, cyber-physical systems(CPS), and smart edge computing. He has published more than 100 papers in Journal and Conference.

Safety Enhancement for Real-Time Parallel Applications in Distributed Automotive Embedded Systems: A Stable Stopping Approach

Guoqi Xie, *Senior Member, IEEE*, Gang Zeng, *Member, IEEE*, and Renfa Li, *Senior Member, IEEE*

Abstract—In distributed automotive embedded systems, safety issues run through the entire life cycle, and safety mechanisms for error handling are desirable for risk control. This article focuses on safety enhancement (i.e., safety mechanisms for error handling) for a safety-critical automotive application within its deadline. A stable stopping approach used for safety enhancement for an automotive application is proposed based on the static recovery mechanism provided in ISO 26262. The Stable Stopping-based Safety Enhancement (SSSE) approach is proposed by combining known backward recovery, proposed forward recovery, and proposed forward-and-backward recovery through primary-backup repetition. The stable stopping (i.e., SSSE) approach is a convergence algorithm, which means that when the reliability value reaches a steady state and the algorithm can stop. Experimental results reveal that the exposure level defined in ISO 26262 drops from E3 to E1 after using SSSE, and such improvement enables a safety guarantee of higher level.

Index Terms—Distributed automotive embedded systems, safety enhancement, stable stopping

1 Introduction

1.1 Background

Safe driving has been the eternal theme of automobiles since their invention. Various safety components, such as safety belt, airbag, and brake-by-wire, have been developed for automobiles to enhance their safety. The 1st edition of the functional safety standard ISO 26262 for road vehicles was officially released in Nov. 2011 to enable different automotive software developers to follow the same principles of safety development [1], [2], [3]. Currently, the 2nd edition of the ISO 26262 standard has been released in Dec. 2018 to further strengthen safety development [4]. Functional safety refers to the absence of unreasonable risks due to hazards caused by the malfunctioning behavior of Electrical and Electronic (E/E) systems according to the definition in ISO 26262 [1], [4]. Safety issues run through the entire life cycle of automotive development, such that the maximum possible safety value should be known during the early design phase to help control risk in the actual development process. Safety enhancement (i.e., safety mechanisms for error handling) is extremely desirable for risk control.

Risk refers to the probability of occurrence of harm and the severity of that harm [1]; hence, safety can be enhanced by reducing risk through reducing the probability or the severity of harm or both. In ISO 26262, severity refers to the measure of the extent of harm to an individual in a specific situation [1], and it cannot be changed for a specific automotive application because it is determined by the nature of the harm. Therefore, risk can only be reduced by reducing the probability of occurrence of harm. In ISO 26262, the probability of occurrence of harm is represented by exposure [1]. Reliability is often associated with random hardware failures, and it is used to represent the probability of the non-occurrence of harm (i.e., reliability = 1 - exposure) in automotive safety issues [5].

1.2 Motivation

With the increasing distribution and complexity of distributed automotive embedded systems, safety-critical automotive applications such as brake-by-wire application [6], [7] and vehicle cruiser control application [8] are parallel applications, where some tasks can be performed simultaneously on different Electronic Control Units (ECUs) in parallel, thereby reducing end-to-end response time. Recently, Directed Acyclic Graph (DAG) has been used to represent the above parallel automotive applications in some works [7], [8] (details about parallel automotive application modeling can be found in Section 3).

A real-time application must guarantee correct response within a specified time constraint (i.e., deadline) [9]. In automotive embedded systems, many safety-critical applications must be real-time, such as anti-lock brake [10]; If the

• G. Xie and R. Li are with the Key Laboratory for Embedded and Cyber-Physical Systems of Hunan Province, College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan 410082, China. E-mail: {xgqman, lirenfa}@hnu.edu.cn.
• G. Zeng is with the Graduate School of Engineering, Nagoya University, Nagoya, Aichi 4648603, Japan. E-mail: sogo@ertl.jp.

*Manuscript received 19 June 2019; revised 2 Feb. 2020; accepted 29 Mar. 2020. Date of publication 2 Apr. 2020; date of current version 24 Apr. 2020. (Corresponding author: Renfa Li.)
Recommended for acceptance by B. Di Martino.
Digital Object Identifier no. 10.1109/TPDS.2020.2984719*

anti-lock brake cannot be finished within its deadline, vehicle collisions may happen due to the delay of the brake action output, thereby causing harm or injury to people (including drivers, passengers, and pedestrians) or damage to automobiles and roads. Violating the real-time constraint (i.e., deadline) is one of systematic failures, which may result in malfunctioning behaviors [5]. That is, a safety-critical automotive application must be completed correctly within its deadline, (i.e., guaranteeing the real-time constraint) [5]; otherwise, the probability of occurrence of harm is considered to be 100 percent. Therefore, the essence of safety enhancement is to enhance the reliability of the automotive application while guaranteeing its real-time constraint.

Ref. [5] studied the problem of enhancing the safety of a real-time parallel automotive application by presenting the Reliability Enhancement Technique (RET). RET is a Backward Safety Enhancement (BSE) approach because it tries to migrate each task to another ECU that generates maximum reliability value from the exit task to the entry task (i.e., backward recovery) (details about BSE can be found in Section 4.1). However, merely using BSE is insufficient to enhance safety due to the following reasons.

1) BSE merely enhances reliability through backward recovery (backward means that the recovery process is from exit to entry tasks), and it does not apply forward recovery (forward means that the recovery process is from entry to exit tasks). For an end-to-end parallel automotive application, applying forward-and-backward recovery to enhance safety could be efficient.
2) BSE is a non-repeated approach, thereby severely limiting the strength of safety enhancement. In fact, safety enhancement can be implemented through repeated recovery, which is an effective fault tolerance measure that uses redundancy. Particularly, the 2nd edition of ISO 26262 introduces the concept of fault tolerance, which means the ability to deliver a specified application in the presence of one or more faults [4].

1.3 Main Contributions

The development life cycle of safety-critical automotive applications includes analysis (concept), design, implementation, and running phases [5]. This study focuses on safety enhancement for a real-time parallel automotive application by using fault tolerance measure during the design phase. The novel contributions of this study include:

1) Considering that BSE in Ref. [5] is merely a backward recovery, we propose the Forward Safety Enhancement (FSE) algorithm (Algorithm 1). Different from BSE that it handles the recovery process from exit to entry tasks, FSE handles the recovery process from entry to exit tasks. FSE is a novel contribution as it tries to forward reallocate each task to another ECU that can generate maximum reliability value without violating given constraints. The forward-and-backward recovery is then proposed to further enhance safety by combining known BSE and proposed FSE algorithms.
2) Considering that BSE and FSE are merely non-repeated algorithms, we proposed the Repeated BSE (RBSE) algorithm (Algorithm 2) and Repeated FSE (RFSE) algorithm (Algorithm 3). Different from BSE and FSE that they handle the recovery process without repetition, RBSE and RFSE handle the recovery process through primary-backup repetition. RBSE (or RFSE) is a novel contribution as it tries to backward (or forward) add a new replica for each task to an available ECU that can generate maximum reliability value among all available ECUs without violating given constraints. The forward-and-backward recovery through primary-backup repetition is then proposed to further enhance safety by combining proposed RBSE and RFSE algorithms.
3) Considering that RBSE and RFSE could be invoked repeatedly until reaching a stable safety value, the Stable Stopping-based Safety Enhancement (SSSE) approach is proposed by combining the above four algorithms. In other words, SSSE is basically a combination of known BSE algorithm and proposed FSE, RBSE, and RFSE algorithms. The stable stopping (i.e., SSSE) approach is a convergence algorithm, which means that when the reliability value reaches a steady state, the algorithm can stop. SSSE is a novel contribution because it is a new combined approach. There are four algorithms to combine the SSSE approach, where we propose there new algorithms.

2 Related Research

The objective of this study is to enhance the safety for real-time parallel automotive applications through considering two safety properties, namely, reliability and response time. Therefore, this study mainly reviews existing research on the reliability and response time of DAG-based applications.

(1) Bi-Criteria (Bi-Objective) Optimization Between Response Time and Reliability. Simultaneously minimizing response time and maximizing reliability (i.e., minimizing exposure) is a bi-criteria optimization problem [11], [12], [13]. Ref. [11] provided the NP-hard complexity results and the optimal mapping algorithm for the bi-objective optimization problem under different variants of multiprocessors, including homogenous and heterogeneous speeds. Ref. [12] proposed meta-heuristic algorithm to solve the above bi-objective optimization problem for a parallel application while meeting the user-defined budget. Ref. [13] proposed a bi-objective algorithm for a parallel application based on Wind Driven Optimization (WDO) to implement the trade-off between minimizing response time and maximizing reliability.

(2) Response Time Optimization With Reliability Constraint. Ref. [5] presented an approach to minimize the response time of an automotive application while guaranteeing the reliability constraint without primary-backup repetition. Refs. [14], [15] presented the MaxRe and RR algorithms to minimize the resource cost of a DAG-based application while guaranteeing the reliability constraint through primary-backup repetition. To effectively assure the reliability requirement, Ref. [16] proposed reliability pre-allocation technique based on geometric mean, which makes the pre-allocated reliability values closer to the center. Particularly,

TABLE 1
Main Abbreviations in This Article

Abbreviation	Definition
RET	Reliability Enhancement Technique
BSE	Backward Safety Enhancement
RBSE	Repeated BSE
FSE	Forward Safety Enhancement
RFSE	Repeated FSE
SSSE	Stable Stopping-based Safety Enhancement
WCET	Worst Case Execution Time
WCRT	Worst Case Response Time
ECU	Electronic Control Unit
DAG	Directed Acyclic Graph
EST	Earliest Start Time
EFT	Earliest Finish Time
LFT	Latest Finish Time
AST	Actual Start Time
AFT	Actual Finish Time

non-repeated and repeated approaches are both proposed in Refs. [14], [15], [16].

(3) Reliability Optimization With Real-Time Constraint. Ref. [5] first studied the problem of enhancing the safety for an automotive application while guaranteeing its real-time constraint by presenting the BSE algorithm. As explained in Section 1, BSE is merely a backward recovery approach toward safety enhancement and is a non-repeated approach, thereby severely limiting the strength of safety enhancement. In other words, BSE is insufficient to enhance safety. This study aims to propose a novel safety enhancement approach by introducing backward-and-forward recovery and primary-backup repetition for parallel automotive applications.

3 Models

Readers can refer to Tables 1 and 2 for main abbreviation and notation, respectively, used in this study.

3.1 Application Model

Fig. 1 shows a simple execution process of the brake-by-wire application [6], which is represented by a DAG, in a

TABLE 2
Main Notations in This Article

Notation	Definition
n_i	A computing task in an automotive application
$m_{i,j}$	A CAN message from tasks n_i to n_j
$w_{i,k}$	WCET of task n_i executed in ECU u_k
$c_{i,j}$	WCRT of message $m_{i,j}$
λ_k	Failure rate for ECU u_k
$R(n_i, u_k)$	Reliability of task n_i executed in ECU u_k
$u_{allo(n_i)}$	Allocated ECU for task n_i
$R(G)$	Reliability of application G
$LB(G)$	Lower bound of application G
$D(G)$	Real-time constraint of application G
$EST(n_i, u_k)$	Earliest start time of task n_i executed in ECU u_k
$EFT(n_i, u_k)$	Earliest finish time of task n_i executed in ECU u_k
$LFT(n_i, u_k)$	Latest finish time of task n_i executed in ECU u_k
$AST(n_i)$	Actual start time of task n_i
$AFT(n_i)$	Actual finish time of task n_i
$avaBT(n_i, u_k)$	Available begin time of task n_i executed in ECU u_k
$avaET(n_i, u_k)$	Available end time of task n_i executed in ECU u_k

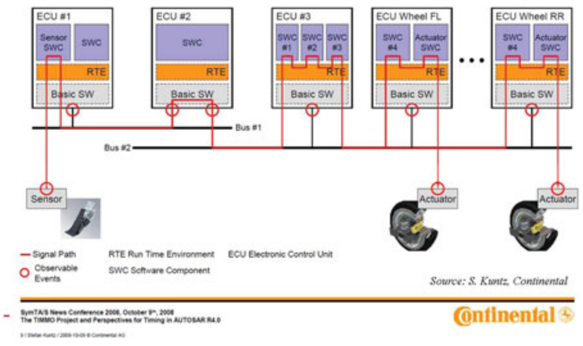


Fig. 1. Brake-by-wire application [6].

Controller Area Network (CAN)-based distributed automotive embedded system. For this parallel application, ECU u_1 receives the data from the sensor to trigger the entry task n_1, which is executed in u_1. n_1 finishes its execution and sends message $m_{1,2}$ to n_2 executed in ECU u_4. Notice that $m_{1,2}$ is transmitted in the CAN bus.

Let $U = \{u_1, u_2, \ldots, u_{|U|}\}$ be a heterogeneous ECU set in the system. In this study, we uniformly use $|X|$ to represent the size of the set X. Meanwhile, a motivational automotive application represented by a DAG $G = (N, W, M, C)$ is shown in Fig. 2.

1) Let n_i be the ith task of G. $pred(n_i)$ and $succ(n_i)$ represent the immediate predecessor task set and immediate successor task set of n_i, respectively. For instance, there are $pred(n_8) = \{n_2, n_4, n_6\}$ and $succ(n_8) = \{n_{10}\}$ in Fig. 2. The entry and exit tasks are denoted by n_{entry} and n_{exit}, respectively. For the application in Fig. 2, there are $n_{\text{entry}} = n_1$ and $n_{\text{exit}} = n_{10}$. The task model should support time-triggered and event-triggered paradigms according to the timing analysis of AUTOSAR standard [17], and the event-triggered task model (i.e., a task is released only if it receives the data from all its predecessor tasks) is considered in this study. Let $D(G)$ be the deadline of automotive application G, and assume that the period of G is not less than $D(G)$. Meanwhile, the application and all the tasks share the same deadline and period.
2) W represents a $|N| \times |U|$ matrix, where $w_{i,k}$ denotes the Worst Case Execution Time (WCET) of n_i executed in ECU u_k. $w_{i,k}$ is infinity (i.e., $w_{i,k} = +\infty$) if n_i cannot be allocated to u_k because some ECUs can only execute specific tasks. The WCETs of each task in three ECUs $\{u_1, u_2, u_3\}$ are listed in Table 3. The

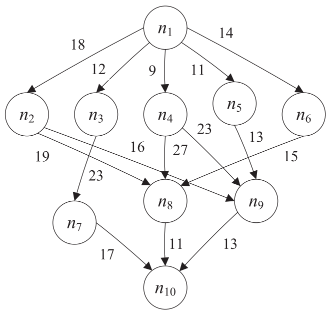


Fig. 2. Motivational automotive application with 10 tasks [5].

TABLE 3
WCETs of Tasks on Different ECUs of the Motivational Automotive Application in Fig. 2 [5]

Task	u_1	u_2	u_3
n_1	14	16	9
n_2	13	19	18
n_3	11	13	19
n_4	13	8	17
n_5	12	13	10
n_6	13	16	9
n_7	7	15	11
n_8	5	11	14
n_9	18	12	20
n_{10}	21	7	16

weight 9 of n_1, u_3 in Table 3 represents the WCET of n_1 in u_3 (i.e., $w_{1,3} = 9$).

3) Let $c_{i,j}$ be the Worst Case Response Time (WCRT) of message $m_{i,j}$, which is a CAN message from tasks n_i to n_j. Considering that the WCRT analysis generally involves a tight WCRT upper bound within a pseudo-polynomial computational time, the WCRTs in this study are the theoretical upper bounds [18].

4) The non-preemptive scheduling for ECUs is adopted to keep consistent with CAN buses. The motivational automotive application in Fig. 2 is used as an explanation of the proposed algorithms. For simplicity, the units of all parameters are ignored in the example.

3.2 Reliability Model

Random hardware failures occur unpredictably during the lifetime of a hardware element, but random hardware failure rates can be reasonably predicted because random hardware faults (including permanent and transient faults) occur based on a probabilistic distribution. [1]. This study considers transient faults (e.g., single bit faults in case of an Error Correcting Code (ECC) with single error correction, double error detection capability as pointed in the 2nd edition of the ISO 26262 standard [4]), which usually obey the Poisson distribution [5], [14], [15], [19], [20]. The reliability of n_i executed in u_k in its WCET is calculated by

$$R(n_i, u_k) = e^{-\lambda_k w_{i,k}}, \tag{1}$$

where λ is the failure rate of an ECU.

In general, the CAN link's failure rate (about 10^{-9}) is much less than ECU's failure rate (about 10^{-6}) due to the usage of CRC and ACK in CAN link [21], [22]. Therefore, communication faults can be disregarded when considering the ECU faults. Thus, the reliability of the application is the product of all its tasks [5], [14], [15], [19], [20]:

$$R(G) = \prod_{n_i \in N} R(n_i) = \prod_{n_i \in N} R(n_i, u_{allo(n_i)}), \tag{2}$$

where $u_{allo(n_i)}$ represents the allocated ECU of n_i.

3.3 Lower Bound of Application

Definition 1 (Lower Bound). *The lower bound means the minimum response time of an application without any constraints.* Scheduling tasks with minimum response time in multiprocessors (ECUs) is known to be an NP-hard optimization problem [23], [24]. Obtaining approximate lower bound is a fast process. The Heterogeneous Earliest Finish Time (HEFT) [23] and Optimistic Cost Table (OCT) [24] algorithms are two typical list scheduling algorithm to obtain approximate lower bounds. This study adopts HEFT to get the lower bound of the application.

There are two phases in HEFT: 1) tasks in the application are sorted in ascending order by the upward rank values (task prioritization phase); 2) each task is allocated to the ECU that has the minimum EFT combining the insertion strategy according to the task prioritization standard (task allocation phase). Ref. [23] provided enough details about how to obtain the lower bound of the motivational application in Fig. 2 by using HEFT.

1) The task prioritization is based on the descending order of $rank_{\mathrm{u}}$, which is calculated by [23]

$$rank_{\mathrm{u}}(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)} \{c_{i,j} + rank_{\mathrm{u}}(n_j)\},$$

where $\overline{w_i}$ represents the average WCET of task n_i. In the motivational example of Fig. 2, the task prioritization is organized by n_1 ($rank_{\mathrm{u}}(n_1)$ = 108), n_3 ($rank_{\mathrm{u}}(n_3)$ = 80), n_4 ($rank_{\mathrm{u}}(n_4)$ = 80), n_2 ($rank_{\mathrm{u}}(n_2)$ = 77), n_5 ($rank_{\mathrm{u}}(n_5)$ = 69), n_6 ($rank_{\mathrm{u}}(n_6)$ = 63.3), n_9 ($rank_{\mathrm{u}}(n_9)$ = 44.3), n_7 ($rank_{\mathrm{u}}(n_7)$ = 42.7), n_8 ($rank_{\mathrm{u}}(n_8)$ = 35.7), and n_{10} ($rank_{\mathrm{u}}(n_{10})$ = 14.7) using the HEFT algorithm [23].

2) The task allocation can get the lower bound of the application. Let $EFT(n_i, u_k)$ be the Earliest Finish Time (EFT) of task n_i executed in ECU u_k. The minimum EFT of the exit task should be the lower bound of the application:

$$LB(G) = \min_{u_k \in U} \{EFT(n_{\mathrm{exit}}, u_k)\}.$$

Therefore, we can iteratively get the minimum EFT of each task from the entry to the exit tasks. We can get the EFT of each task based on

$$EFT(n_i, u_k) = EST(n_i, u_k) + w_{i,k},$$

where $EST(n_i, u_k)$ represents the Earliest Start Time (EST) of task n_i executed in ECU u_k:

$$\begin{cases} EST(n_{\mathrm{entry}}, u_k) = 0; \\ EST(n_i, u_k) = \max\left(avaBT(n_i, u_k), \max_{n_h \in pred(n_i)}\{AFT(n_h) + c_{h,i}^{p,k}\}\right), \end{cases} \tag{3}$$

$avaBT(n_i, u_k)$ is the available begin time of u_k. n_h is an immediate predecessor task of n_i. $AFT(n_h)$ is the Actual Finish Time (AFT) of task n_h. n_i has different $EST(n_i, u_k)$ values depending on the allocations of predecessor tasks because $c_{h,i}^{p,k}$ is not a fixed value.

$$c_{h,i}^{p,k} = \begin{cases} c_{h,i} & p \neq k. \\ 0 & p = k. \end{cases}.$$

Table 4 shows the details of obtaining lower bound of the motivational example. First, n_1 is allocated to u_3 (denoted with bold text) as it has the minimum EFT of 9. Then, n_3 is

TABLE 4
Details of Obtaining Lower Bound of the Motivation Example

Task	$EFT(n_i, u_1)$	$EFT(n_i, u_2)$	$EFT(n_i, u_3)$
n_1	14	16	**9**
n_3	32	34	**28**
n_4	31	**26**	45
n_2	**40**	46	46
n_5	52	39	**38**
n_6	53	**42**	47
n_9	69	**68**	76
n_7	58	83	**49**
n_8	**62**	79	73
n_{10}	102	**80**	97

allocated to u_3 (denoted with bold text) as it has the minimum EFT of 28. The EFTs of n_4, n_2, n_5, n_6, n_9, n_7, n_8, and n_{10} are shown in Table 4. Finally, the lower bound of application is 80, as the AFT of the exit task (i.e., n_{10}) is 80. On the basis of Table 4, Fig. 3 shows the HEFT-generated lower bound ($LB(G) = 80$) of the motivational automotive application G in Fig. 2, whereas the arrows between tasks represent their communication.

The real-time constraint $D(G)$ must be larger than or equal to $LB(G)$. For the motivational automotive application in Fig. 2, we let the real-time constraint be $D(G) = 100$, as shown in Fig. 3.

3.4 Problem Statement and Two Constraints

Considering that the essence of safety enhancement is to enhance the reliability of the automotive application while guaranteeing its real-time constraint as explained in Section 1.2, the problem to be solved in this study is to enhance the reliability of the parallel automotive application G

$$R(G) = \prod_{n_i \in N} R(n_i),$$

towards enhancing the safety by task allocations, under its real-time constraint

$$RT(G) \leqslant D(G),$$

where $RT(G)$ represents the response time of application G. The problem statement in fact involves two constraints.

Definition 2 (Constraint 1). *The precedence constraints between the current task (i.e., n_i) and its immediate predecessor and immediate successor tasks, namely, the Actual Start Time (AST) and AFT of n_i must adhere to the data dependencies with other tasks.*

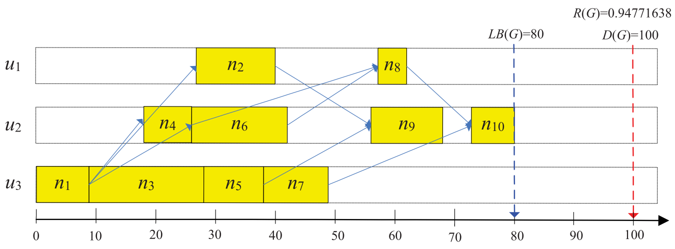


Fig. 3. HEFT-generated lower bound of the motivational automotive application in Fig. 2.

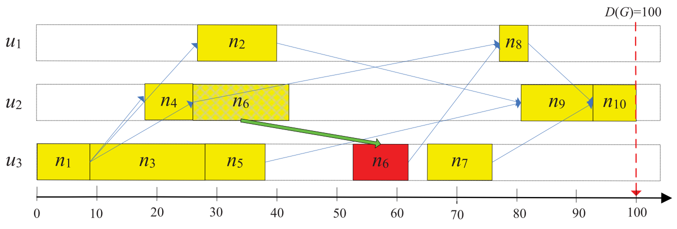


Fig. 4. n_{10}, n_9, n_8, and n_7 move the primary (i.e., the task itself) in fixed ECU without migration, and n_6 is migrated from u_2 to u_3.

For instance, n_8 cannot be scheduled before n_2, n_4, and n_6 according to the *Constraint 1* in Fig. 3.

Definition 3 (Constraint 2). *The real-time constraint of the automotive application, namely, the AFT of the exit task must be less than or equal to the deadline of the automotive application.*

In the process of safety enhancement, the above two constraints cannot be violated; otherwise, the safety enhancement is not valid.

4 Backward and Forward Safety Enhancement

4.1 Existing BSE

As explained previously, the BSE algorithm is a backward approach and its idea is as follows. BSE tries to migrate (i.e., re-allocate) the current task n_i to another ECU that generates maximum reliability value for n_i without violating *Constraint 1* and *Constraint 2*. The above process is explored from the exit task to the entry task (i.e., backward recovery). In the following, the idea of BSE is simply explained on the basis of the HEFT-generated lower bound (Fig. 3).

1) The backward sequence of tasks is sorted by the descending order of the AFT values generated by HEFT. For example, the backward sequence of tasks in Fig. 3 is n_{10}, n_9, n_8, n_7, n_6, n_2, n_5, n_3, n_4, and n_1.
2) The exit task n_{10} is first considered. n_{10} has moved its end time to 100 (i.e., $D(G) = 100$) in fixed ECU u_2 without violating *Constraint 2*, as shown in Fig. 4. We assume that n_{10}, n_9, n_8, and n_7 have been re-allocated using BSE, as shown in Fig. 4. Note that n_{10}, n_9, n_8, and n_7 are only rescheduled on the same ECU without migration. ECU migration is avoided due to one of the following reasons: 1) the current task n_i cannot be migrated to another ECU otherwise *Constraint 1* will be violated; 2) the current task n_i can be migrated to the other ECU without violating *Constraint 1*, but the new ECU cannot obtain a higher reliability value than the current ECU for n_i.
3) The fifth task n_6 is prepared to be re-allocated. n_6 (denoted with red color) can be migrated from u_2 to u_3 because it can generate maximum reliability for n_6 without violating *Constraint 1*, as shown in Fig. 4.
4) The following tasks use the same principle as the aforementioned tasks. As shown in Fig. 5, n_2, n_5, n_4, and n_1 are only moved in fixed ECUs without migration, whereas n_3 (denoted with red color) is migrated from u_3 to u_2. Finally, the reliability of the automotive application is enhanced from 0.9477163 (Fig. 3)

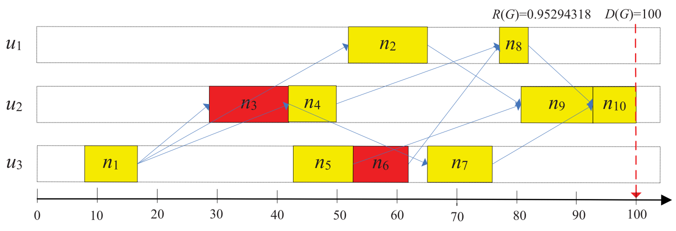


Fig. 5. BSE-generated lower bound of the motivational automotive application in Fig. 2.

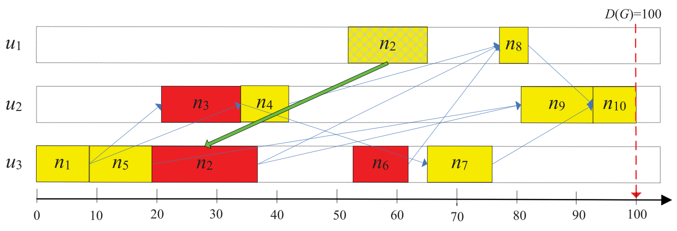


Fig. 6. n_1, n_3, n_4, and n_5 have moved their individual begin times and end times to small values in fixed ECUs without migration.

to 0.95294318 (Fig. 5) by using RET to migrate n_6 and n_3. Fig. 5 shows that *Constraint 1* and *Constraint 2* are not violated. In other words, the following facts are found: 1) each task adheres to the data dependencies with other tasks; 2) the application is finished within the deadline.

4.2 Proposed FSE Algorithm

Fig. 5 shows that the begin time of the entry task n_1 is not started at 0 but at 8 after using BSE. An intuitive feeling is that the begin time of n_1 can be set to 0 through making some possible migrations of n_1 and its successor tasks to enhance safety without violating *Constraint 1* and *Constraint 2*. The above process is explored from the entry task to the exit task (i.e., forward recovery).

1) The forward sequence of tasks is sorted by the ascending order of AST values generated by BSE. For example, the forward sequence of tasks in Fig. 5 is $n_1, n_3, n_4, n_5, n_2, n_6, n_7, n_8, n_9$, and n_{10}.
2) The entry task n_1 is first considered. n_1 has moved its start time and end time to 0 and 8, respectively, in fixed ECU u_3 without migration. The reason is that if n_1 can be migrated, it has already been migrated with the BSE algorithm, as shown in Fig. 5.
3) n_1, n_3, n_4, and n_5 have moved their individual begin times and end times to smaller values in fixed ECUs without migration, as shown in Fig. 6.
4) The fifth task n_2 is prepared to be re-allocated. n_2 can be migrated from u_3 to u_1 because it can generate maximum reliability for n_2 without violating *Constraint 1*, as shown in Fig. 6. n_2 cannot be migrated by using BSE. However, n_2 can be migrated when using FSE further. In the following, the migration of n_2 will be explained.

To determine whether the current task n_i can be migrated, its EST and Latest Finish Time (LFT) in each ECU are required to obtain in advance to ensure that *Constraint 1* and *Constraint 2* are not violated. The EST of n_i in u_k has been shown in Eq. (3), whereas the LFT of n_i in u_k is calculated by

$$\begin{cases} LFT(n_{\text{exit}}, u_k) = D(G); \\ LFT(n_i, u_k) = \min\left(avaET(n_i, u_k), \min_{n_j \in succ(n_i)}\{AST(n_j) - c_{i,j}^{k,q}\}\right), \end{cases} \tag{4}$$

$avaET(n_i, u_k)$ is the available end time of u_k for n_i. n_i has different $LFT(n_i, u_k)$ values depending on the allocations of successor tasks because $c_{i,j}^{k,q}$ is not a fixed value.

$$c_{i,j}^{k,q} = \begin{cases} c_{i,j} & k \neq q. \\ 0 & k = q. \end{cases}$$

Take n_2 as an example. The $avaBT(n_2, u_k)$ and $avaET(n_2, u_k)$ values are as follows:

$$\begin{cases} avaBT(n_2, u_1) = 0 \\ avaBT(n_2, u_2) = 42 \\ avaBT(n_2, u_3) = 19 \end{cases} \qquad \begin{cases} avaET(n_2, u_1) = 77 \\ avaET(n_2, u_2) = 81 \\ avaET(n_2, u_3) = 53. \end{cases}$$

Then, the $EST(n_2, u_k)$ and $LFT(n_2, u_k)$ values are as follows:

$$\begin{cases} EST(n_2, u_1) = 27 \\ EST(n_2, u_2) = 42 \\ EST(n_2, u_3) = 19 \end{cases} \qquad \begin{cases} LFT(n_2, u_1) = 65 \\ LFT(n_2, u_2) = 58 \\ LFT(n_2, u_3) = 53. \end{cases}$$

After *Constraint 1* and *Constraint 2* are determined, the ECU u_{allo} that has the maximum reliability value for n_i can then be selected.

$$R(n_i) = R(n_i, u_{allo(i)}) = \max_{w_{i,k} \leqslant (LFT(n_i,u_k) - EST(n_i,u_k))} R(n_i, u_k),$$

where $w_{i,k} \leqslant (LFT(n_i, u_k) - EST(n_i, u_k))$ should be satisfied to guarantee *Constraint 1* is not violated. Note that u_{allo} can be the currently allocated ECU u_{cur} to be moved in or a reallocated ECU u_{new} to be migrated to.

Continuing with n_2 as an example. The $R(n_2, u_k)$ values are as follows:

$$\begin{cases} R(n_2, u_1) = 0.99094128 \\ R(n_2, u_2) = \text{NULL} \\ R(n_2, u_3) = 0.99282586. \end{cases} \tag{5}$$

$R(n_2, u_2)$ is NULL because $w_{2,2}$ = 19, which is larger than 16 ($LFT(n_2, u_2)$ - $EST(n_2, u_2)$ = 58 - 42 = 16), such that allocating n_2 to u_2 will violate *Constraint 1*. Given that $R(n_2, u_3)$ has the maximum reliability values in Eq. (5), n_2 is migrated from u_1 to u_3.

The AST and AFT of n_i are correspondingly updated to

$$AST(n_i) = EST(n_i, u_{allo(i)}),$$

and

$$AFT(n_i) = AST(n_i) + w_{i,allo(i)},$$

respectively. For example, the AST and AFT of n_2 are as follows:

$$AST(n_2) = EST(n_2, u_3) = 19,$$

and

$$AFT(n_2) = AST(n_2) + w_{2,3} = 19 + 18 = 37.$$

5) The following tasks use the same principle as the aforementioned tasks. As shown in Fig. 7, n_6, n_7, n_8, n_9, and n_{10} are moved in fixed ECUs without migration. Finally, the reliability of the automotive application is enhanced from

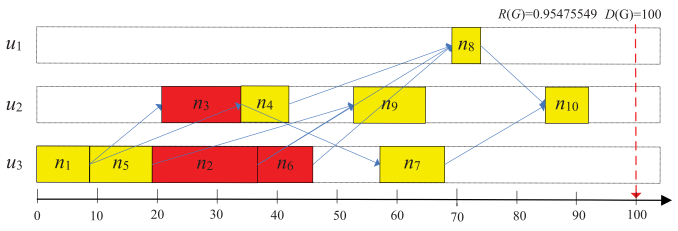


Fig. 7. FSE-generated task mapping of the motivational automotive application in Fig. 2.

0.95294318 (Fig. 5) to 0.95475549 (Fig. 7) by using FSE to migrate n_2. Fig. 7 shows that *Constraint 1* and *Constraint 2* are still not violated.

The FSE algorithm is proposed on the basis of the aforementioned analysis, as shown in Algorithm 1.

Algorithm 1. The FSE Algorithm

```
Input: $U = \{u_1, u_2, \dots, u_{|U|}\}$, $G$, $D(G)$, task mapping generated
       by the previous algorithm
Output: $R(G)$, $RT(G)$, and task mapping
 1: The tasks' forward sequence $forward\_seq$ is also sorted by
    the ascending order of the AST values generated by the
    previous algorithm;
 2: while (there are tasks in $forward\_seq$) do
 3:   $n_i \leftarrow forward\_seq.out()$;
 4:   $u_{\mathrm{cur(i)}}$ indicates the currently allocated ECU.
 5:   Move the $n_i$ in ECU $u_{cur(i)}$ by $AST(n_i) \leftarrow EST(n_i, u_{cur(i)})$
      and $AFT(n_i) \leftarrow \left(AST(n_i) + w_{\mathrm{i,cur(i)}}\right)$;
 6:   for (each ECU $u_k \in U$) do
 7:     Calculate $EST(n_i, u_k)$ using Eq. (3);
 8:     Calculate $LFT(n_i, u_k)$ using Eq. (4);
 9:     if (((LFT(n_i, u_k) - EST(n_i, u_k)) < w_{i,k}) then
10:       continue;
11:     end if
12:     Calculate $R(n_i, u_k)$ using Eq. (1);
13:   end for
14:   Select the ECU $u_{\mathrm{new(i)}}$ with the maximum reliability value
      under $R(n_i, u_{\mathrm{new(i)}}) > R(n_i, u_{\mathrm{new(i)}})$
15:   if ($u_{new(i)}$ is not NULL ) then
16:     Migrate the $n_i$ to ECU $u_{new(i)}$ by $AST(n_i) \leftarrow EST(n_i, u_{new(i)})$
        and $AFT(n_i) \leftarrow \left(AST(n_i) + w_{i,new(i)}\right)$;
17:   end if
18: end while
19: Calculate the reliability $R(G)$ using Eq. (2);
```

Overall, FSE tries to forward reallocate each task to another ECU that can generate maximum reliability value without violating *Constraint 1* and *Constraint 2*. The details are explained as follows. The time complexity of FSE is $O(|N|^2 \times |U|)$ and is explained below. (1) Traversing all tasks requires $O(|N|)$ time (Lines 2–18). (2) Calculating $EST(n_i, u_k)$ and $LFT(n_i, u_k)$ requires $O(|N| \times |U|)$ time (Lines 6–11). Therefore, the time complexity of the FSE algorithm is the same as those of HEFT and BSE.

5 Repeated Safety Enhancement

5.1 Proposed RBSE Algorithm

Fig. 7 shows that the end time of the exit task n_{10} is not ended at 100 but at 92 after using FSE. An intuitive feeling is that the end time of n_{10} can be set to 100 through making some possible migrations of tasks by BSE again. Setting the end time of n_{10} to 100 is necessary, but making possible migrations of tasks by BSE again is unfeasible. The reason is that through the backward-and-forward recovery of BSE and FSE in Section 4, the migration of tasks has reached the extreme. Continually invoking BSE or FSE recovery can not generate new migrations. Fortunately, primary-backup repetition is an effective fault tolerant measure (i.e., recovery through repetition) to implement safety enhancement.

Passive repetition and active repetition are two types of primary-backup repetition paradigms [14], [15]. Passive repetition aims to reschedule the task on a a backup ECU when this task fails in the primary ECU. Active repetition will simultaneously execute ε replicas of the task in ε ECUs. Each ECU only execute one replica for the same task, such that the maximum number of replicas is the number of ECU. Homogenous redundancy and heterogeneous redundancy are two types of redundancy paradigms [4]. Homogeneous redundancy means the duplication of homogeneous elements, whereas heterogeneous redundancy means the combination of hardware devices and software tasks. In this study, active repetition and homogeneous redundancy are adopted because such combination implements independence among tasks. In addition, as pointed out in the 2nd edition of ISO 26262, homogeneous redundancy during the design phase focuses primarily on controlling the effects of transient faults or random faults in the hardware (i.e., ECU in this study), on which a similar software is executed (e.g., temporal redundant execution of software). The aforementioned features are consistent with the automotive application and reliability models of this study.

Given that the reliability of n_i in u_k is $R(n_i, u_k) = e^{-\lambda_k w_{i,k}}$ according to Eq. (1), the failure probability of n_i in u_k without repetition is

$$Fail(n_i, u_k) = 1 - R(n_i, u_k) = 1 - e^{-\lambda_k w_{i,k}}.$$

Assume that there are num_i ($num_i \leqslant |U|$) replicas for n_i, the failure probability of n_i through active repetition is

$$\begin{aligned} Fail(n_i) &= \prod_{\beta=1}^{num_i} Fail\left(n_i^{\beta}, u_{allo(n_i^{\beta})}\right) \\ &= \prod_{\beta=1}^{num_i} \left(1 - R\left(n_i^{\beta}, u_{allo(n_i^{\beta})}\right)\right), \end{aligned}$$

where $u_{allo(n_i^{\beta})}$ represents the allocated ECU of replica n_i^{β}. Therefore, the reliability of n_i is

$$R(n_i) = 1 - Fail(n_i) = 1 - \prod_{\beta=1}^{num_i} \left(1 - R\left(n_i^{\beta}, u_{allo(n_i^{\beta})}\right)\right), \quad (6)$$

(1) The backward sequence of tasks is sorted by the descending order of the AFT values. For example, the backward sequence of the tasks in Fig. 7 is n_{10}, n_8, n_7, n_9, n_6, n_4, n_2, n_3, n_5, and n_1.

(2) The exit task n_{10} is first considered. n_{10} just moves its begin time and end time to 93 and 100, respectively, in fixed ECU u_2 without any repetition. Repetitions on u_1 or u_3 will violate *Constraint 1*. The details are explained below. The EST and LFT of n_{10} are

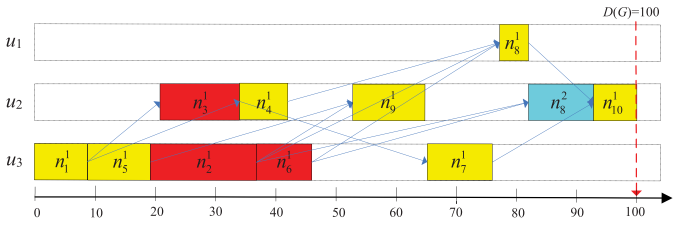


Fig. 8. A replica is added to u_2 for n_8.

$$\begin{cases} EST(n_{10}, u_1) = 85 \\ EST(n_{10}, u_2) = 85 \\ EST(n_{10}, u_3) = 85 \end{cases} \qquad \begin{cases} LFT(n_{10}, u_1) = 100 \\ LFT(n_{10}, u_2) = 100 \\ LFT(n_{10}, u_3) = 100. \end{cases}$$

The LFT values of n_{10} on u_1 and u_3 are

$$LFT(n_{10}, u_1) - EST(n_{10}, u_1) = 100 - 85 = 15 \; < \; w_{10,1} = 21,$$

and

$$LFT(n_{10}, u_3) - EST(n_{10}, u_3) = 100 - 85 = 15 \; < \; w_{10,3} = 16;$$

hence, neither u_1 nor u_3 can be added with replicas.

(3) The second task n_8 is considered to be repeated. A replica can be added to u_2 for n_8 because this operation does not violate *Constraint 1*, as shown in Fig. 8. Why a replica can be added to u_2 for n_8 is explained in the following.

The EST and LFT values of n_8 are

$$\begin{cases} EST(n_8, u_1) = 69 \\ EST(n_{10}, u_2) = 65 \\ EST(n_8, u_3) = 69 \end{cases} \qquad \begin{cases} LFT(n_8, u_1) = 82 \\ LFT(n_8, u_2) = 93 \\ LFT(n_8, u_3) = 82. \end{cases}$$

The LFT values of n_8 on u_2 and u_3 are

$$LFT(n_8, u_2) - EST(n_8, u_2) = 93 - 65 = 28 \; > \; w_{8,2} = 11,$$

and

$$LFT(n_8, u_3) - EST(n_8, u_3) = 82 - 69 = 13 \; < \; w_{10,3} = 14;$$

hence, a replica can be added to u_2, whereas a replica cannot be added to u_3 for n_8, as shown in Fig. 8.

(3) The following tasks use the same principle as the aforementioned tasks. As shown in Fig. 9, n_7, n_6, n_4, and n_3 add individual replicas in ECUs. Finally, the reliability of the automotive application is enhanced from 0.95475549 (Fig. 7) to 0.97584149 (Fig. 9) by using RBSE to add replicas for n_8, n_7, n_6, n_4, and n_3.

The RBSE algorithm is proposed on the basis of the aforementioned analysis, as shown in Algorithm 2.

Overall, RBSE tries to backward add a new replica for each task to an available ECU that can generate maximum reliability value among all available ECUs without violating the *Constraint 1* and *Constraint 2*. The time complexity of RBSE is also $O(|N|^2 \times |U|)$, which is the same as those of HEFT and BSE.

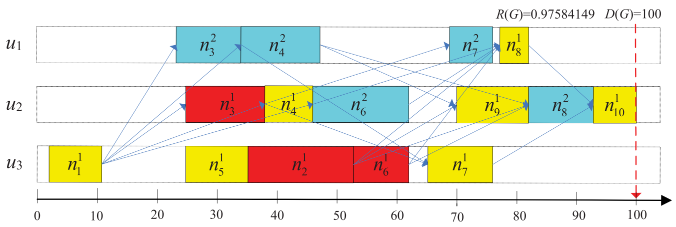


Fig. 9. RBSE-generated task mapping of the motivational automotive application in Fig. 2.

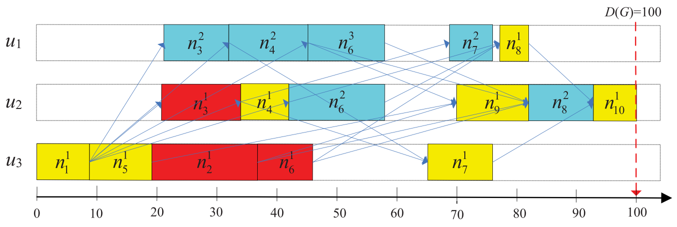


Fig. 10. A new replica is added to u_1 for n_6 by using RFSE.

Algorithm 2. The RBSE Algorithm

```
Input: $U = \{u_1, u_2, \ldots, u_{|U|}\}$, $G$, $D(G)$, task mapping generated
    by the previous algorithm
Output: $R(G)$, $RT(G)$, and task mapping
 1: The tasks' backward sequence $backward\_seq$ is also sorted
    by the descending order of the AFT values generated by
    the previous algorithm;
 2: while (there are tasks in $backward\_seq$) do
 3:   $n_i \leftarrow backward\_seq.out()$;
 4:   $U_{cur}$ indicates that the ECU set has been allocated;
 5:   for (each ECU $u_{cur(i)} \in (U_{cur})$) do
 6:     Move the $n_i$ in ECU $u_{cur(i)}$ by $AFT(n_i) \leftarrow LFT(n_i, u_{cur(i)})$
      and $AST(n_i) \leftarrow \big(AFT(n_i) - w_{i,k}\big)$;
 7:   end for
 8:   for (each ECU $u_k \in (U - U_{\text{cur}})$) do
 9:     Calculate $EST(n_i, u_k)$ using Eq. (3);
10:     Calculate $LFT(n_i, u_k)$ using Eq. (4);
11:     if ((($LFT(n_i, u_k) - EST(n_i, u_k)$) $<$ $w_{i,k}$) then
12:       continue;
13:     end if
14:     Calculate $R(n_i, u_k)$ using Eq. (1);
15:   end for
16:   Select the ECU $u_{new(i)}$ with the maximum reliability value
      under $R(n_i, u_{new(i)}) > R(n_i, u_{new(i)})$
17:   if ($u_{new(i)}$ is not NULL ) then
18:     Add the replica of $n_i$ to ECU $u_{new(i)}$ by $AFT(n_i) \leftarrow LFT(n_i,$
        $u_{new(i)})$ and $AST(n_i) \leftarrow \big(AFT(n_i) - w_{i,new(i)}\big)$;
19:   end if
20:   Calculate the reliability $R(n_i)$ using Eq. (6);
21: end while
22: Calculate the reliability $R(G)$ using Eq. (2);
```

5.2 Proposed RFSE Algorithm

Similar to the non-repeated FSE, a repeated FSE can be implemented by adding a possible replica in an ECU for each task.

1) Similar to FSE, the forward sequence of tasks is also sorted by the ascending order of the AST values generated by RBSE. For example, the forward sequence of tasks in Fig. 9 is n_1, n_3, n_5, n_4, n_2, n_6, n_7, n_9, n_8, and n_{10}.
2) The entry task n_1 is first handled. The begin time and end time of n_1 are just moved to 0 and 8, respectively, in fixed ECU n_3. No additional replicas are added for n_3, n_5, n_4, n_2 until n_6; a new replica for n_6

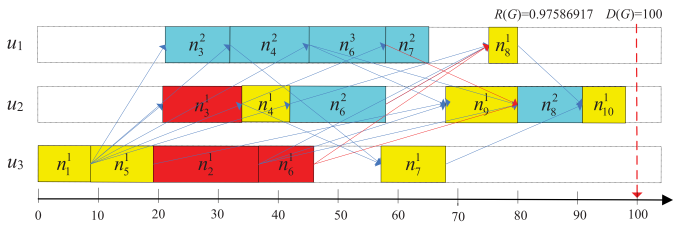


Fig. 11. RFSE-generated task mapping of the motivational automotive application in Fig. 2.

is added to u_1. That is, n_6 has a total of three replicas, as shown in Fig. 10.

3) The tasks n_1, n_3, n_5, n_4, n_2, and n_6 have been handled in Fig. 10. Similar to already handled tasks, the remaining tasks n_7, n_9, n_8, and n_{10} will be handled; however, no additional replicas are added for n_7, n_9, n_8, and n_{10}, as shown in Fig. 11. Finally, the reliability of the automotive application is enhanced from 0.97584149 (Fig. 9) to 0.97586917 (Fig. 11) by using RFSE to add replicas for n_6.

Finally, the RFSE algorithm is proposed and is shown in Algorithm 3.

Algorithm 3. The RFSE Algorithm

Input: $U = \{u_1, u_2, \ldots, u_{|U|}\}$, G, $D(G)$, task mapping generated by the previous algorithm
Output: $R(G)$, $RT(G)$, $rep(G)$, and task mapping

```
1:  The tasks' forward sequence $forward\_seq$ is also sorted by the ascending order of the AST values generated by the previous algorithm;
2:  while (there are tasks in $forward\_seq$) do
3:    $n_i \leftarrow forward\_seq.out()$;
4:    $U_{cur}$ indicates that the ECU set has been allocated;
5:    for (each ECU $u_{cur(i)} \in (U_{cur})$) do
6:      Move the $n_i$ in ECU $u_{cur(i)}$ by $AST(n_i) \leftarrow EST(n_i, u_{cur(i)})$ and $AFT(n_i) \leftarrow (AST(n_i) + w_{i,k})$;
7:    end for
8:    for (each ECU $u_k \in (U - U_{\text{cur}})$) do
9:      Calculate $EST(n_i, u_k)$ using Eq. (3);
10:     Calculate $LFT(n_i, u_k)$ using Eq. (4);
11:     if $(((LFT(n_i, u_k) - EST(n_i, u_k)) < w_{i,k})$ then
12:       continue;
13:     end if
14:     Calculate $R(n_i, u_k)$ using Eq. (1);
15:   end for
16:   Select the ECU $u_{new(i)}$ with the maximum reliability value under $R(n_i, u_{\text{new(i)}}) > R(n_i, u_{new(i)})$
17:   if ($u_{new(i)}$ is not NULL ) then
18:     Add the replica of $n_i$ to ECU $u_{new(i)}$ by $AST(n_i) \leftarrow EST(n_i, u_{new(i)})$ and $AFT(n_i) \leftarrow (AFT(n_i) + w_{i,new(i)})$;
19:   end if
20:   Calculate the reliability $R(n_i)$ using Eq. (6);
21: end while
22: Calculate the reliability $R(G)$ using Eq. (2);
```

5.3 Proposed SSSE Approach

After a round of backward-and-forward recovery using RBSE and RFSE, the second round, the third round, and so on can continue until the reliability value reaches a stable and fixed value. We have explained that continually doing non-repeated BSE or FSE recovery can not enhance safety in Section 5.1. Fortunately, invoking repeated RBSE and RFSE recovery could enhance safety by adding a possible new replica to an ECU for each task in each round instead of task migration. The flow chart of SSSE is shown in Fig. 12.

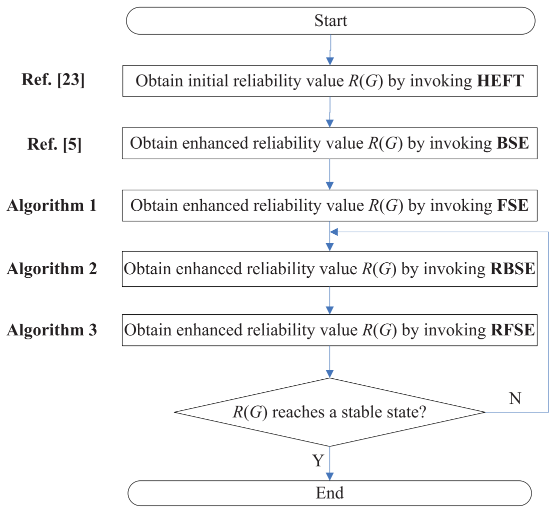

Start
Ref. [23]
Obtain initial reliability value R(G) by invoking HEFT
Ref. [5]
Obtain enhanced reliability value R(G) by invoking BSE
Algorithm 1
Obtain enhanced reliability value R(G) by invoking FSE
Algorithm 2
Obtain enhanced reliability value R(G) by invoking RBSE
Algorithm 3
Obtain enhanced reliability value R(G) by invoking RFSE
R(G) reaches a stable state?
N
Y
End

Fig. 12. Flowchart of the proposed SSSE approach.

Table 5 shows the reliability enhancement process of the motivational automotive application in Fig. 2 by using related algorithms. BSE, FSE, RBSE (first round), RFSE (first round) gradually enhance the reliability value from 0.94771638 to 0.97586917. 0.97586917 is a stable and fixed value because the same result is obtained using RFSE (first round), RBSE (second round), and RFSE (second round). As the motivational automotive application only includes three ECUs, the second round does not reflect the difference from the first round. The final response time $RT(G)$ is 98, and the number of replicas $rep(G)$ is 16 for the motivational automotive application in Fig. 2. SSSE is proposed by combing the existing HEFT and BSE algorithms and the proposed FSE, RBSE, RFSE algorithms. SSSE can invoke RBSE and RFSE repeatedly until reaching a stable value. Notice that RBSE and RFSE will not always enhance the reliability due to the limited ECUs and strict *Constraint 1* and *Constraint 2*. Therefore, SSSE can stop in the while loop.

It is worth pointing out that SSSE is consistent with the static recovery mechanism in the automotive functional safety standard ISO 26262, where backward recovery, forward recovery, and recovery through repetition have been recommended as static recovery mechanism as pointed in Section 7.4.12, Part 6 of the 2nd edition of ISO 26262 [4]. The

TABLE 5
Reliability Enhancement Process of the Motivational Automotive Application in Fig. 2

	$R(G)$	$RT(G)$	$D(G)$	$rep(G)$	Figure
HEFT	0.94771638	80	100	10	Fig. 3
BSE	0.95294318	92	100	10	Fig. 5
FSE	0.95475549	92	100	10	Fig. 7
RBSE (first round)	0.97584149	98	100	15	Fig. 9
RFSE (first round)	0.97586917	98	100	16	Fig. 11
RBSE (second round)	0.97586917	98	100	16	Fig. 11
RFSE (second round)	0.97586917	98	100	16	Fig. 11

SSSE approach proposed in this study is actually a recovery through repetition based on backward recovery and forward recovery of the static recovery mechanism. Therefore, the special idea of SSSE is the organic combination of the aforementioned three recoveries under the *Constraint 1* and *Constraint 2*. Therefore, the SSSE approach complies with the automotive functional safety standard from a practical perspective. In other words, the stable stopping (i.e., SSSE) approach is a convergence algorithm, which means that when the reliability value reaches a steady state, the algorithm can stop. There is no denying that the proposed SSSE approach has burden or disadvantages in terms of the ECU they have due to the use of repetition.

1) If repetition is not adopted, a task only causes one Message Receiving Interrupt (MRI) to one of its successor task in the ECU receiving its message. For instance, task n_8 is a successor task of task n_6 in the motivational automotive application of Fig. 2; after task n_6 is finished, it only needs to send one message to n_8, such that n_6 only causes one MRI to n_8.
2) If repetition is adopted, a task only causes multiple MRIs to its one successor task in the ECU receiving its message. For instance, after using the SSSE approach, n_6 causes four message MRIs to n_8. The details are shown in Fig. 11, where n_6 has three replicas of n_6^1, n_6^2, and n_6^3, while n_8 has two replicas of n_8^1, and n_8^2. In Fig. 11, n_6 cause four MRIs (denoted with red arrows) to n_8.

Too many unnecessary MRIs bring considerable ECU load due to the execution of Interrupt Service Routine (ISR) and interrupt-triggered switch overhead between tasks. These disadvantages are not negligible for safety-critical distributed automotive embedded systems, especially for ECUs running at relatively low clock speeds and with small memory space in automotive ECUs [25], [26], [27]. Therefore, although the SSSE approach enhances the reliability of the automotive application, it also brings considerable ECU load that would affect the quality of task scheduling.

6 Experiments

The state-of-the-art BSE algorithm is selected to compare with the proposed algorithms. The metrics are response time $RT(G)$, reliability $R(G)$, and replica number $rep(G)$. As this study focuses on the early design phase, the application parameters are known on the basis of their real deployment. The parameter values of a real automotive are adopted. The failure rate falls in the range of $10^{-6}/\mu$s - $16 \times 10^{-6}/\mu$s. The WCETs of the tasks and the WCRTs of the messages fall under the range of 100 μs - 400 μs generated by uniform distribution. The ECU number is 16.

According to the related description in ISO 26262, fault tolerance is merely a static recovery mechanism during the design phase to maintain the safe state of automotive application, and does not involve the final operational results [4]. Considering that fault tolerance does not address the context where the application is used to switch off the system and the context where a safe state can be directly reached by switching off the application, and not all imaginable faults can be tolerated as pointed out in the 2nd edition of ISO 26262 [4], this study focuses on safety prevention using safety enhancement technique during the design phase, and does not determine the final execution result during the running phase. Therefore, the automotive applications in this study will be tested by simulation to implement safety enhancement in a static recovery manner.

TABLE 6
ASIL Determination Formed by a Combination of Severity, Exposure, and Controllability Provided in ISO 26262 [4]

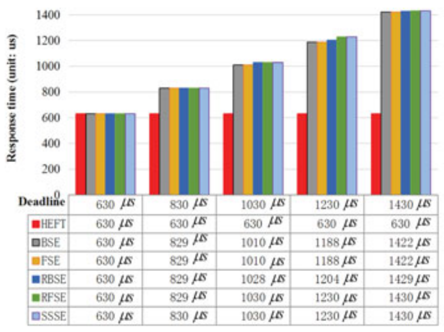
Severity	Exposure	Controllability		
		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A/QM
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

6.1 ASIL Determination in ISO 26262

We introduce the Automotive Safety Integrity Level (ASIL) in advance because it is related to the safety evaluation of an automotive application. In ISO 26262, ASIL is a risk classification scheme to define the safety requirements. The ASIL is established by performing a Hazard Analysis and Risk Assessment (HARA) of a potential hazard by evaluating at the severity, exposure and controllability of the automotive operating scenario. An ASIL is the combination of severity, exposure, controllability [1], [28]. There are four ASILs identified by the ISO 26262: ASIL A, ASIL B, ASIL C, ASIL D [1], [4]. The higher the ASIL, the greater the risk and the more effort required to reduce the risk.

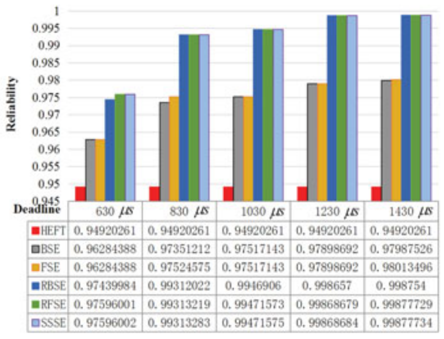
Severity means an estimate of the extent of harm that may occur in a potentially hazardous event; there are four severity levels: S0 (no injuries), S1 (light and moderate injuries), S2 (severe and life-threatening injuries), and S3 (fatal injuries). Exposure means the state of being in an operational situation where it is hazardous if the situation is coincident with the failure mode under the analysis; there are five exposure levels: E0 (incredibly), E1 (very low probability), E2 (low probability), E3 (medium probability), and E4 (high probability). Controllability means the ability to avoid specified harm or damage through the timely reactions of the persons involved; there are four controllability levels: C0 (controllable in general), C1 (simply controllable), C2 (normally controllable), and C3 (uncontrollable). Notice that the controllability is related to the driver's driving state rather than the system.

ISO 26262 provides the ASIL determination formed by a combination of severity, exposure, and controllability, as shown in Table 6 [4]. Each ASIL is the combination of severity, exposure, and controllability values. There is a subtle difference about ASIL determination between the 1st and 2nd editions of the ISO 26262 standard. In the 1st edition,



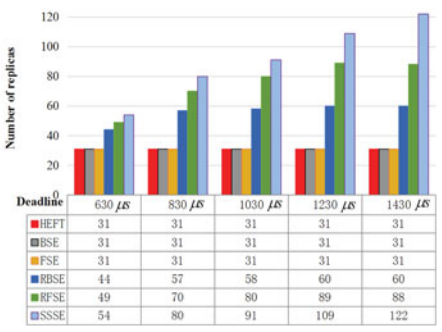
Deadline	630 μs	830 μs	1030 μs	1230 μs	1430 μs
🟥 HEFT	630 μs	630 μs	630 μs	630 μs	630 μs
▨ BSE	630 μs	829 μs	1010 μs	1188 μs	1422 μs
🟧 FSE	630 μs	829 μs	1010 μs	1188 μs	1422 μs
🟦 RBSE	630 μs	829 μs	1028 μs	1204 μs	1429 μs
🟩 RFSE	630 μs	829 μs	1030 μs	1230 μs	1430 μs
🟦 SSSE	630 μs	830 μs	1030 μs	1230 μs	1430 μs

(a) Response time values.



Deadline	630 μs	830 μs	1030 μs	1230 μs	1430 μs
🟥 HEFT	0. 94920261	0. 94920261	0. 94920261	0. 94920261	0. 94920261
▨ BSE	0. 96284388	0. 97351212	0. 97517143	0. 97898692	0. 97987526
🟧 FSE	0. 96284388	0. 97524575	0. 97517143	0. 97898692	0. 98013496
🟦 RBSE	0. 97439984	0. 99312022	0. 9946906	0. 998657	0. 998754
🟩 RFSE	0. 97596001	0. 99313219	0. 99471573	0. 99868679	0. 99877729
🟦 SSSE	0. 97596002	0. 99313283	0. 99471575	0. 99868684	0. 99877734

(b) Reliability values.



Deadline	630 μs	830 μs	1030 μs	1230 μs	1430 μs
🟥 HEFT	31	31	31	31	31
▨ BSE	31	31	31	31	31
🟧 FSE	31	31	31	31	31
🟦 RBSE	44	57	58	60	60
🟩 RFSE	49	70	80	89	88
🟦 SSSE	54	80	91	109	122

(c) Numbers of replicas.

Fig. 13. Values of the real-life parallel automotive application in different real-time constraints.

the combination of S3, E1, and C3 is ASIL A. However, in the 2nd edition, this combination may be Quality Management (QM) if several unlikely situations are combined and the change may result in a lower probability of exposure than E1. QM indicates that software can be developed according to the quality management process to manage the identified risk, and no safety-related design needs to be considered. QM is not an ASIL but may be specified in the HARA [1], [4]. Severity has been decided after HARA and cannot be changed. Controllability is a fixed value during the design phase. Therefore, to enhance safety for a real-time parallel automotive application, the feasible measure is to drop the exposure of the application according to Table 6.

6.2 Real-Life Parallel Automotive Application

The real-life parallel automotive application with 31 tasks from Ref. [5] are adopted. This application contains six blocks: engine controller (n_1-n_7), automatic gear box (n_8-n_{11}), anti-locking brake (n_{12}-n_{17}), wheel angle sensor (n_{18}-n_{19}), suspension controller (n_{20}-n_{24}), and body work (n_{25}-n_{31}).

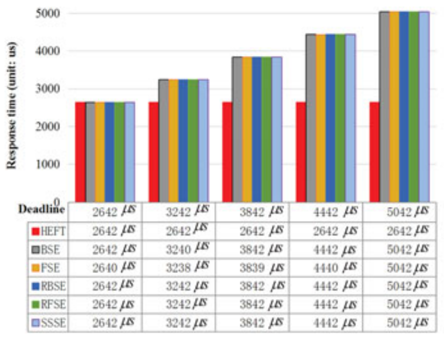
The HEFT-generated lower bound of the application is 630 μs. As the real-time constraint must be larger than or equal to the lower bound explained in Section 3.3, The real-time constraint is changed from 630 μs to 1430 μs with 200 μs increments. The results are shown in Fig. 13.

1) Fig. 13(a) shows that the response time values using all the algorithms are less than or equal to corresponding real-time constraints. As the HEFT-generated response time is the lower bound, it is fixed at 630 μs. Except that the real-time constraint is equal to the lower bound (i.e., 630 μs), Fig. 13(a) shows the BSE and FSE-generated response time values are less than the real-time constraints; such results indicate that BSE and FSE still leave a larger optimization space for RBSE, RFSE, and SSSE.
2) Fig. 13(b) shows that a long response time could lead to a high reliability, and the results basically confirm that minimizing response time and maximizing reliability is a bi-criteria optima problem. In addition, Fig. 13(b) clearly shows that the reliability values are divided into three gradients: 1) HEFT; 2) BSE and FSE; and 3) RBSE, RFSE, and SSSE. As expected, SSSE generates the maximum reliability values, followed by RFSE, RBSE, FSE, BSE, and HEFT. HEFT has the lowest reliability value of 0.94920261. By backward-and-forward recovery using BSE and FSE, the reliability value are increased to 0.962–0.980. Through further primary-backup repetition using RBSE and RFSE, the reliability values are increased to 0.974–0.998. With the continuous involvement of RBSE and RFSE (i.e., using SSSE), the reliability values can reach a stable state in each case. SSSE only increases a small amount of reliability compared with RFSE because SSSE only invokes RBSE and RFSE twice at most. Even so, as long as the slack response time increases, safety enhancement can be maximized as much as possible.

Automotive functional safety standard ISO 26262 provides the duration/probability of exposure levels of E1 (very low probability with reliability goal of $<$ 0.99), E2 (low probability with reliability goal 0.99), E3 (medium probability with reliability goal $>$ 0.9 and $<$ 0.99), and E4 (high probability with reliability goal $<=$ 0.9) (see Table 7). Take Fig. 15(b) as an example, the maximum reliability value using BSE is 0.97987526 (E3), whereas using SSSE reaches 0.99877734 (E1). That is, the exposure level drops from E3 to E1 after using SSSE. Such exposure drop (i.e., reliability enhancement, safety enhancement) enables a safety guarantee of higher level. For example, Table 6 shows the ASIL determination formed by a combination of severity, exposure, and controllability in the 2nd edition of ISO 26262 [4]. According to the ASIL determination in Table 6, there are two combinations of ASIL C. We consider one combination, where the severity of the application is S3, the exposure of the application is E3, and the controllability of the driver is C3. Severity and controllability are fixed values during the design phase as explained in Section 3.2. Therefore, to achieve QM availability for the ASIL C application, one possible measure is to drop the exposure of the application from E3 to E1 according to Table 6.

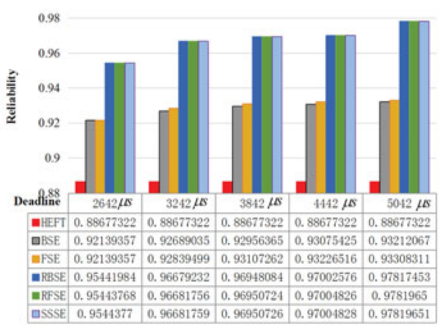
TABLE 7
Classes of Probability of Exposure Regarding Duration/Probability of Exposure in ISO 26262 [1], [4]

Exposure level	Probability of exposure	Reliability goal
E1 Very low probability	Not specified	At least exceeds 0.99
E2 Low probability	$<$ 1%	0.99
E3 Medium probability	[1%, 10%]	$>$0.9
E4 High probability	$>$ 10%	$<=$0.9



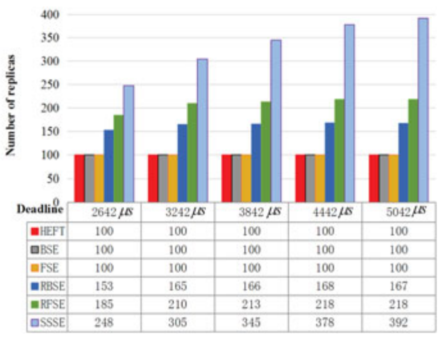
Deadline	2642 μs	3242 μs	3842 μs	4442 μs	5042 μs
🟥 HEFT	2642 μs	2642 μs	2642 μs	2642 μs	2642 μs
■ BSE	2642 μs	3240 μs	3842 μs	4442 μs	5042 μs
🟧 FSE	2640 μs	3238 μs	3839 μs	4440 μs	5042 μs
🟦 RBSE	2642 μs	3242 μs	3842 μs	4442 μs	5042 μs
🟩 RFSE	2642 μs	3242 μs	3842 μs	4442 μs	5042 μs
□ SSSE	2642 μs	3242 μs	3842 μs	4442 μs	5042 μs

(a) Response time values.



Deadline	2642 μs	3242 μs	3842 μs	4442 μs	5042 μs
🟥 HEFT	0. 88677322	0. 88677322	0. 88677322	0. 88677322	0. 88677322
■ BSE	0. 92139357	0. 92689035	0. 92956365	0. 93075425	0. 93212067
🟧 FSE	0. 92139357	0. 92839499	0. 93107262	0. 93226516	0. 93308311
🟦 RBSE	0. 95441984	0. 96679232	0. 96948084	0. 97002576	0. 97817453
🟩 RFSE	0. 95443768	0. 96681756	0. 96950724	0. 97004826	0. 9781965
□ SSSE	0. 9544377	0. 96681759	0. 96950726	0. 97004828	0. 97819651

(b) Reliability values.



Deadline	2642 μs	3242 μs	3842 μs	4442 μs	5042 μs
🟥 HEFT	100	100	100	100	100
■ BSE	100	100	100	100	100
🟧 FSE	100	100	100	100	100
🟦 RBSE	153	165	166	168	167
🟩 RFSE	185	210	213	218	218
□ SSSE	248	305	345	378	392

(c) Numbers of replicas.

Fig. 14. Values of synthetic application in different real-time constraints.

Even though SSSE has very little reliability enhancement over RBSE and RFSE in Fig. 13(b), the results are still useful for safety-sensitive distributed automotive embedded systems. The classes of probability of exposure in ISO 26262 shown in Table 7 are informative, not prescriptive, and leave a great deal of discretion to whoever is building each component system and ultimately to the automakers and suppliers [29]. For instance, the probability of exposure E1 in ISO 26262 is not specified, and its corresponding reliability goal at least exceeds 0.99, as shown in Table 7. Therefore, different automakers and suppliers can choose different reliability goals according to their product and market orientations as long as the values belong to the same exposure. For instance, it is feasible that the reliability goal for E3 is 0.99877730 because this value exceeds 0.99, and ISO 26262 requires high accuracy in reliability value. When the real-time constraint is 1430 μs shown in Fig. 13(b), SSSE drops to E1, whereas RBSE and RFSE only drops to E2. This result sufficiently reflects the advantages of SSSE.

3) Fig. 13(c) clearly shows that the HEFT-, BSE-, and FSE-generated replicas are fixed at 31 because they do not implement primary-backup repetition. The replicas produced by RBSE, RFSE, and SSSE progressively increase; for example, when the real-time constraint is 630 μs, the numbers of RBSE-, RFSE-, and SSSE-generated replicas are 44, 49, and 54, respectively. As the reliability value increases with the increased real-time constraint, the number of replicas required also rises through primary-backup repetition; for example, when the real-time constraint is 1430 μs, the numbers of RBSE, RFSE, and SSSE-generated replicas reach 60, 88, and 122, respectively. Although primary-backup repetition is a practical fault tolerance measure to enhance safety, it has some weaknesses. That is, RBSE, RFSE, and SSSE, especially SSSE increase the burden on ECUs due to the additional replicas and affect the control efficiency of ECUs due to the time-consuming extra code. These problems should be considered the weaknesses of fault tolerance measure.

(4) In each case, the time required to calculate the result using SSSE is very short and is within 1 s. The reason is that SSSE only invokes RBSE and RFSE twice at most.

6.3 Synthetic Parallel Application

In addition to using a real-life parallel automotive application to confirm the advantage of the proposed algorithms, an additional synthetic application with 100 tasks are adopted to analyze the results. This synthetic application has the same parameter values as those of a real-life parallel automotive application. A randomly generated application can be obtained by using a task graph generator [30]. The following parameters are set in this experiment: the communication-to-computation ratio is 1, the shape parameter is 1, and the heterogeneity factor is 0.5. The heterogeneity factor values are in the 0-1 scope in the task graph generator, where 0.1 and 1 are the lowest and highest heterogeneity factors, respectively. The HEFT-generated lower bound of the application is 2642 μs. As the real-time constraint must be larger than or equal to the lower bound explained in Section 3.3, the real-time constraint is changed from 2642 μs to 5042 μs with 600 μs increments. The results are shown in Fig. 14.

1) Similar to Fig. 13(a), Fig. 14(a) shows that the response time values using all the algorithms are also less than or equal to the corresponding real-time constraints. Overall, Fig. 13(a) and Fig. 14(a) have the same regular pattern.
2) Similar to Fig. 13(b), Fig. 14(b) also clearly divides the reliability values into three gradients: 1) HEFT; 2) BSE and FSE; and 3) RBSE, RFSE, and SSSE. HEFT has the lowest reliability value of 0.88677322, which is much lower than 0.94920261 shown in Fig. 13(b). The reason is that the reliability value of the application is the product of all tasks, such that the application reliability value generated by 100 tasks is naturally lower than that generated by 31 tasks (the real-life parallel automotive application contains 31 tasks). Overall, Fig. 13(b) and Fig. 14(b) present the same regular pattern for all the algorithms.
3) Fig. 14(c) shows the same regular pattern as Fig. 13(c). By synthesizing the data in Figs. 13(c) and 14(c), the following facts are confirmed. 1) HEFT, BSE, and FSE have fixed numbers of replicas $|N|$ due to the absence of primary-backup repetition, and $|N|$ represents the size of task set of the application. 2) The numbers of RBSE-generated replicas are between $|N|$ and $2 \times |N|$ because each task is repeated once at most. 3) The numbers of RFSE-generated replicas are between $|N|$ and $3 \times |N|$ because each task is repeated at most once on the basis of RBSE; 3) SSSE-generated replicas are theoretically between $|N|$ and $|U| \times |N|$, where $|U|$ represents the size of processor set, but approximately $4 \times |N|$ in the actual situations. However, as mentioned

earlier, the main weaknesses of RBSE, RFSE, and SSSE are that they increase the burden on ECUs due to the additional replicas and affect the control efficiency of ECUs due to the time-consuming extra code.

4) In each case, the time required to calculate the result using SSSE is still within 1 s because SSSE only invokes RBSE and RFSE five times at most for an automotive application with 100 tasks.

7 Conclusion

The SSSE approach for a real-time parallel automotive application to enhance safety to a stable value was proposed in this study. SSSE consists of existing HEFT and BSE algorithms and pretested FSE, RBSE, and RFSE algorithms. SSSE enhances the safety by using stable stopping approach on the basis of the forward-and-backward recovery through primary-backup repetition. SSSE can make the exposure level drops from E3 to E1 towards safety guarantee for higher level. SSSE is actually a recovery through repetition based on backward recovery and forward recovery of the static recovery mechanism pointed out in the 2nd edition of ISO 26262. It is believed that the stable stopping approach can serve as a guideline in the safety design of automotive applications. The future work could further study the safety enhancement of distributed automotive embedded systems by considering multiple automotive applications with different deadlines.

Acknowledgments

The authors would like to express their gratitude to the associate editor and three anonymous reviewers for their constructive comments which have helped to improve the quality of this article. This work was supported in part by the National Natural Science Foundation of China under Grants 61702172, 61932010, 61672217, and 61972139, the CCF-Tencent Open Fund under Grant CCF-TecentRAGR20190119, the Open Research Project of the State Key Laboratory of Synthetical Automation for Process Industries (SAPI), Northeastern University, China under Grant PAL-N201803, the Natural Science Foundation of Hunan Province under Grant 2018JJ3076, and the Fundamental Research Funds for the Central Universities, Hunan University, China.

References

[1] "Road vehicles-functional safety, iso 26262," Nov. 2011. [Online]. Available: https://www.iso.org/standard/43464.html

[2] S. S. Williamson, A. K. Rathore, and F. Musavi, "Industrial electronics for electric transportation: Current state-of-the-art and future challenges," *IEEE Trans. Ind. Electron.*, vol. 62, no. 5, pp. 3021–3032, May 2015.

[3] A. Nardi and A. Armato, "Functional safety methodologies for automotive applications," in *Proc. 36th Int. Conf. Comput.-Aided Des.*, 2017, pp. 970–975.

[4] "*Road vehicles-functional safety*, ISO 26262," Dec. 2018. [Online]. Available: https://www.iso.org/standard/68383.html

[5] G. Xie, G. Zeng, Y. Liu, J. Zhou, R. Li, and K. Li, "Fast functional safety verification for distributed automotive applications during early design phase," *IEEE Trans. Ind. Electron.*, vol. 65, no. 5, pp. 4378–4391, May 2018.

[6] E. Rolf, "Formal performance analysis in automotive systems design - a rocky ride to new grounds," in *Proc. 23rd Int. Conf. Comput. Aided Verification*, 2011. [Online]. Available: http://formalverification.cs.utah.edu/cav2011/content/presentations/CAV 2011V3.pdf

[7] G. Xie *et al.*, "Reliability enhancement towards functional safety goal assurance in energy-aware automotive cyber-physical systems," *IEEE Trans. Ind. Informat.*, vol. 14, pp. 5447–5462, Dec. 2018.

[8] W. Jiang, P. Pop, and K. Jiang, "Design optimization for security-and safety-critical distributed real-time applications," *Microprocessors Microsyst.*, vol. 52, pp. 401–415, Jul. 2017.

[9] J. M. Rivas, J. J. Gutiérrez, J. C. Palencia, and M. G. Harbour, "Deadline assignment in edf schedulers for real-time distributed systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 10, pp. 2671–2684, Sep. 2015.

[10] "Real-time computing - wikipedia," Apr. 2014. [Online]. Available: https://en.wikipedia.org/wiki/Real-time_computing

[11] A. Benoit, F. Dufossé, A. Girault, and Y. Robert, "Reliability and performance optimization of pipelined real-time systems," *J. Parallel Distrib. Comput.*, vol. 73, no. 6, pp. 851–865, Jun. 2013.

[12] N. Kaur and S. Singh, "A budget-constrained time and reliability optimization bat algorithm for scheduling workflow applications in clouds," *Procedia Comput. Sci.*, vol. 98, pp. 199–204, Jan. 2016.

[13] P. Singh, M. Dutta, and N. Aggarwal, "Bi-objective hwdo algorithm for optimizing makespan and reliability of workflow scheduling in cloud systems," in *Proc. 14th IEEE India Council Int. Conf.*, 2017, pp. 1–9.

[14] L. Zhao, Y. Ren, Y. Xiang, and K. Sakurai, "Fault-tolerant scheduling with dynamic number of replicas in heterogeneous systems," in *Proc. 12th IEEE Int. Conf. High Perform. Comput. Commun*, 2010, pp. 434–441.

[15] L. Zhao, Y. Ren, and K. Sakurai, "Reliable workflow scheduling with less resource redundancy," *Parallel Comput.*, vol. 39, no. 10, pp. 567–585, Jul. 2013.

[16] G. Xie, Z. Li, N. Yuan, R. Li, and K. Li, "Toward effective reliability requirement assurance for automotive functional safety," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 23, no. 5, pp. 1–26, Aug. 2018.

[17] 2020. https://www.autosar.org/standards/classic-platform/

[18] X. Yong, Z. Gang, C. Yang, R. Kurachi, H. Takada, and L. Renfa, "Worst case response time analysis for messages in controller area network with gateway," *IEICE Trans. Inf. Syst.*, vol. 96, no. 7, pp. 1467–1477, Jul. 2013.

[19] A. Girault and H. Kalla, "A novel bicriteria scheduling heuristics providing a guaranteed global system failure rate," *IEEE Trans. Dependable Secure Comput.*, vol. 6, no. 4, pp. 241–254, Oct./Dec. 2009.

[20] B. Zhao, H. Aydin, and D. Zhu, "On maximizing reliability of real-time embedded applications under hard energy constraint," *IEEE Trans. Ind. Informat.*, vol. 6, no. 3, pp. 316–328, Aug. 2010.

[21] M. Di Natale, H. Zeng, P. Giusto, and A. Ghosal, *Understanding and Using the Controller Area Network Communication Protocol: Theory and Practice*. Berlin, Germany: Springer, 2012.

[22] A. Kazeminia, "Reliability optimization of hardware components and systems topology during early design phase," *J. Amer. Assoc. Pediatric Ophthalmology Strabismus*, vol. 18, no. 4, 2014, Art. no. e9.

[23] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002.

[24] H. Arabnejad and J. G. Barbosa, "List scheduling algorithm for heterogeneous systems by an optimistic cost table," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 3, pp. 682–694, Mar. 2014.

[25] F. Pölzlbauer, R. I. Davis, and I. Bate, "Analysis and optimization of message acceptance filter configurations for controller area network (CAN)," in *Proc. 25th Int. Conf. Real-Time Netw. Syst.*, 2017, pp. 247–256.

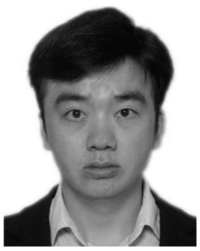
[26] R. I. Davis, S. Altmeyer, and A. Burns, "Mixed criticality systems with varying context switch costs," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2018, pp. 140–151.

[27] P. Patel, M. Vanga, and B. B. Brandenburg, "Timershield: Protecting high-priority tasks from low-priority timer interference (outstanding paper)," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2017, pp. 3–12.

[28] G. Xie, Y. Chen, Y. Liu, R. Li, and K. Li, "Minimizing development cost with reliability goal for automotive functional safety during design phase," *IEEE Trans. Rel.*, vol. 67, no. 1, pp. 196–211, Mar. 2018.

[29] C. Hobbs and P. Lee, "Understanding iso 26262 asils," Jul. 2013. [Online]. Available: https://www.electronicdesign.com/embedded/understanding-iso-26262-asils

[30] "Task graph generator," 2015. https://sourceforge.net/projects/taskgraphgen/



Guoqi Xie (Senior Member, IEEE) received the PhD degree in computer science and engineering from Hunan University, in 2014. He is currently an associate professor of embedded and cyber-physical systems with Hunan University. He was a postdoctoral research fellow with Nagoya University. His current research interests include embedded and cyber-physical systems, parallel and distributed systems, and safety- and security-critical systems. He received the Best Paper Award at IEEE ISPA 2016 and the 2018 IEEE TCSC Early Career Researcher Award. He is currently serving on the editorial boards of *Journal of Systems Architecture*, *Journal of Circuits*, *Systems and Computers*, and *Microprocessors and Microsystems*. He is an ACM senior member.

Gang Zeng (Member, IEEE) received the PhD degree in information science from Chiba University, in 2006. He is an associate professor at the Graduate School of Engineering, Nagoya University. From 2006 to 2010, he was a researcher, and then assistant professor at the Center for Embedded Computing Systems (NCES), the Graduate School of Information Science, Nagoya University. His research interests mainly include power-aware computing and real-time embedded system design. He is a member of IPSJ.

Renfa Li (Senior Member, IEEE) is currently the professor and chair of embedded and cyber-physical systems with Hunan University. He is the chair of the Key Laboratory for Embedded and Cyber-Physical Systems. His major interests include computer architectures, embedded computing systems, cyber-physical systems, and Internet of things. He is a member of the council of CCF, and a senior member of ACM.

Customizable Scale-Out Key-Value Stores

Ali Anwar, Yue Cheng, Hai Huang, Jingoo Han, Hyogi Sim, Dongyoon Lee, Fred Douglis, *Fellow, IEEE*, and Ali R. Butt

Abstract—Enterprise KV stores are often not well suited for HPC applications, and thus cumbersome end-to-end KV design customization is required to meet the needs of modern HPC applications. To this end, in this article we present BESPOKV, an adaptive, extensible, and scale-out KV store framework. BESPOKV decouples the KV store design into the control plane for distributed management and the data plane for local data store. For the control plane, BESPOKVprovides pre-built modules, called *controlets*, supporting common distributed functionalities (e.g., replication, consistency, and topology) and their various combinations. This decoupling allows BESPOKV to take a user-provided single-server KV store, called a *datalet*, and transparently enables a scalable and fault-tolerant distributed KV store service. The resulting distributed stores are also adaptive to consistency or topology requirement changes and can be easily extended for new types of services. Such specializations enable innovative uses of KV stores in HPC applications, especially for emerging applications that utilize KV-friendly workloads. We evaluate BESPOKV in a local testbed as well as in a public cloud settings. Experiments show that BESPOKV-enabled distributed KV stores scale horizontally to a large number of nodes, and performs comparably and sometimes 1.2× to 2.6× better than the state-of-the-art systems.

Index Terms—Key-value stores, HPC KV stores, scale-out KV stores, application tailored storage

1 Introduction

The underlying storage and I/O fabric of modern high performance computing (HPC) increasingly employ new technologies such as flash-based systems and non-volatile memory (NVM). While improving I/O performance, e.g., via providing more efficient and fast I/O burst buffer, such technologies also provide for opportunities to explore the use of in-memory storage such as key-value (KV) stores in the HPC setting. Distributed KV stores are beginning to play an increasingly critical role in supporting today's HPC applications. Examples of this use include dynamic consistency control [1], coupling applications [2], [3], and storing intermediate results [4], among others. Relatively simple data schemas and indexing enable KV stores to achieve high performance and high scalability, and allow them to serve as a cache for quickly answering various queries, where user experience satisfaction often determines the success of the applications. Consequently, a variety of distributed KV stores have been developed, mainly in two forms: natively-distributed and proxy-based KV stores.

The *natively-distributed* KV stores [5], [6], [7], [8], [9], shown in Fig. 1a, are designed with distributed services (e.g., topology, consistency, replication, and fault tolerance) in mind from the beginning, and are often specialized for one specific setting. For example, HyperDex [10] supports Master-Slave topology and Strong Consistency (MS+SC). Facebook relies on its own distributed Memcache [8] with Master-Slave topology and Eventual Consistency (MS+EC). Amazon employs Dynamo [6] with Active-Active[1] topology and Eventual Consistency (AA+EC).

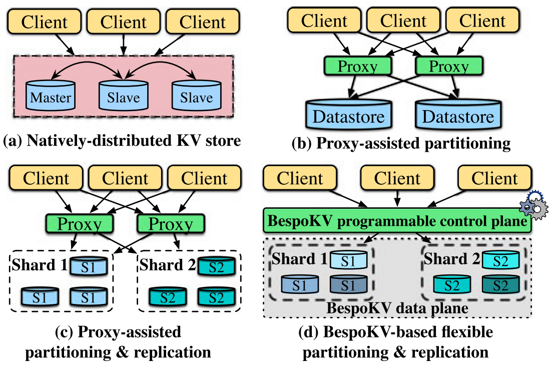
The key limitation of natively-distributed KV stores lie in their *inflexible* monolithic design where distributed features are deeply baked with backend data stores. Such a design allows the developers to highly optimize the KV store performance. However, such optimizations are not portable to any other KV store. The rigid design implies that these KV stores are not adaptive to ever-changing user demands for different backend, topology, consistency, or other services. For instance, Social Artisan [11] and Behance [12] moved from MongoDB to Cassandra for scalability and maintenance reasons [13]. Conversely, Flowdock [14] migrated from Cassandra to MongoDB due to stability issues. Unfortunately, this migration process is very frustrating and time/money-consuming as requires data remodeling and extra migration resources [13].

Alternatively, *proxy-based* distributed KV stores leverage a proxy layer to add distributed services into existing backend data stores. For example, Mcrouter [15], and Twemproxy [16] can be used as a proxy to enable a basic form of distributed Memcached [17] with partitioning, as shown in Fig. 1b. Twemproxy supports additional Redis [18] backend

• *A. Anwar is with IBM Research–Almaden, San Jose, CA 95120-6099. E-mail: ali.anwar2@ibm.com.*
• *Y. Cheng is with George Mason University, Fairfax, VA 22030. E-mail: yuecheng@gmu.edu.*
• *H. Huang is with IBM Research–T.J. Watson, Ossining, NY 10562. E-mail: haih@us.ibm.com.*
• *J. Han and A.R. Butt are with Virginia Tech, Blacksburg, VA 24061. E-mail: jingoo@vt.edu, butta@cs.vt.edu.*
• *H. Sim is with Oak Ridge National Laboratory, Oak Ridge, TN 37830. E-mail: simh@ornl.gov.*
• *D. Lee is with Stony Brook University, Stony Brook, NY 11794, and also with Virginia Tech, Blacksburg, VA 24061. E-mail: dongyoon@cs.stonybrook.edu.*
• *F. Douglis is with Perspecta Labs, Basking Ridge, NJ 07920. E-mail: fd-ic@douglis.org.*

Manuscript received 11 June 2019; revised 25 Feb. 2020; accepted 8 Mar. 2020. Date of publication 30 Mar. 2020; date of current version 24 Apr. 2020. (Corresponding author: Ali Anwar.) Recommended for acceptance by J. Wang. Digital Object Identifier no. 10.1109/TPDS.2020.2982640

1. Active-Active is also called multi-master in database literature.



Client
Client
Client
Master
Slave
Slave
(a) Natively-distributed KV store
Client
Client
Client
Proxy
Proxy
Datastore
Datastore
(b) Proxy-assisted partitioning
Client
Client
Client
Proxy
Proxy
Shard 1
S1
Shard 2
S2
(c) Proxy-assisted partitioning & replication
Client
Client
Client
BespoKV programmable control plane
Shard 1
S1
Shard 2
S2
BespoKV data plane
(d) BespoKV-based flexible partitioning & replication

Fig. 1. Different approaches to enable distributed KV stores: (a) natively-distributed (b-d) proxy-based.

as well. Recently, Netflix Dynomite [19] extended Twemproxy to support high availability and cross-datacenter replication, as illustrated in Fig. 1c.

Unlike monolithic natively-distributed KV stores, the use of a separate proxy layer enables support for multiple backends. Each single-server KV store such as Memcached [17], Redis [18], LevelDB [20], and Masstree [21] has own its merit, so the ability to choose one or mix is an ample reward. However, existing proxy-based KV stores are still limited to a single topology and consistency: e.g., Dynomite supports AA +EC only. We see that existing solutions have not yet extracted the full potential of proxy-based distributed KV stores. Table 1 summarizes the limitations of existing proxy-based KV solutions such as Dynomite and Twmemproxy.

This paper presents BESPOKV, a flexible, ready-to-use, adaptive, and extensible distributed KV store framework. Fig. 1d illustrates BESPOKV's distributed KV store architecture. BESPOKV takes as input a single-server KV store, which we call *datalet*, and transparently enables a distributed KV store service, supporting a variety of cluster topologies, consistency models, replication options, and fault tolerance (Section 3). For the control plane, BESPOKV provides a set of distributed

TABLE 1
BESPOKV versus State-of-the-Art Systems for KV Stores

System	S	R	MB	MC	MT	AR	P
Single-server	✗	✗	✗	✗	✗	✗	✗
Twemproxy	✓	✗	✓	✗	✗	✗	✗
Mcrouter	✓	✓	✗	✗	✗	✗	✗
Dynomite	✓	✓	✓	✗	✗	✗	✗
BESPOKV (Our work)	✓	✓	✓	✓	✓	✓	✓

S: Sharding; R: Replication; MB: Multiple backends; MC: Multiple consistency techniques, e.g. strong, eventual, per-request, etc.; MT: Multiple network topologies, e.g. Master-Slave, Master-Master, Peer-to-Peer, etc.; AR: Automatic failover recovery; P: Programmable.

management units, referred to as *controlets*. To the best of our knowledge, BESPOKV is *the first system supporting multiple consistency techniques, multiple network topologies, dynamic topology/consistency adaptation, automatic failover, and programmability, all at the same time.*

Table 2 shows the benefit of the proposed BESPOKV framework. Here, four snippet implementations of the core functions for a simple KV store are presented in pseudocode. To implement everything from scratch (`(a) Vanilla`), a developer creates her own concurrency control functionality (`Lock()`, `Unlock()`), and consistency and quorum management logic (`Sync()`, `Quorum()`). Using a distributed lock server (`(b) Lockserver-based`), the developer can avoid implementing synchronization functions. Similarly, using Vsync [22] library (`(c) Vsync-based`) for consistency management further reduces engineering effort. However, there are two limitations. First, developers still need to familiarize themselves with a large collection of system/library interfaces to use them appropriately in the application code. Second, such approaches often provide only a single technique for replication or consistency: e.g., Vsync uses only a virtual synchrony to replicate data. In contrast, using BESPOKV (option (d)), developers only need to implement a non-distributed version of the KV store (*datalet*), and then BESPOKV transparently scales it out to a variety of distributed environments with different requirements.

TABLE 2
An Example of Four Possible Approaches to Developing a Distributed KV Store With the Last One Being the Proposed Approach

(a) Vanilla

```
void Put(Str key, Obj val) {
  if (this.master):
    Lock(key)
    Table.insert(key, val)
    Unlock(key)
  Sync(master.slaves)
}

Obj Get(Str key) {
  if (this.master)
    Obj val = Quorum(key)
  Sync(master.slaves)
  return val
}

void Lock(Str key) {
  ... // Acquire lock
}

void Unlock(Str key) {
  ... // Release lock
}

void Sync(Replicas peers) {
  ... // Update replicas
}

void Quorum(Str key) {
  ... // Select a node
}
```

(b) Lockserver-based

```
void Put(Str key, Obj val) {
  if (this.master):
    ls.Lock(key) // lockserver
    Table.insert(key, val)
    ls.Unlock(key) // lockserver
  Sync(master.slaves)
}

Obj Get(Str key) {
  if (this.master)
    Obj val = Quorum(key)
  Sync(master.slaves)
  return val
}

void Sync(Replicas peers) {
  ... // Update replicas
}

void Quorum(Str key) {
  ... // Select a node
}
```

(c) Vsync-based

```
#include <vsynclib>

void Put(Str key, Obj val) {
  if (this.master):
    ls.Lock(key) // lockserver
    Table.insert(key, val)
    ls.Unlock(key) // lockserver
  Vsync.Sync(master.slaves)
}

Obj Get(Str key) {
  if (this.master)
    Obj val = Vsync.Quorum(key)
  Vsync.Sync(master.slaves)
  return val
}
```

(d) BESPOKV-based

```
void Put(Str key, Obj val) {
  Table.insert(key, val)
}

Obj Get(Str key) {
  return Table(key)
}
```

In case of `(a) vanilla`, LoC of `Lock`, `Unlock`, `Sync`, and `Quorum` is not shown. Similarly, LoC to implement `Lock` and `Unlock` recipe for ZooKeeper is not shown. Vsync is available in C# and requires use of proper APIs but for the sake of simplicity and consistency we assume a C++ language grammar.

BESPOKV's decoupled control and data plane architecture, configurability, and extensibility enable new solutions for emerging HPC systems and workloads. First, BESPOKV makes it easy for HPC developers to explore different design tradeoffs in future HPC systems with heterogeneous hardware resources. Prior solutions are developed for one architecture. For instance, SKV [4] is designed for the IBM Blue Gene Active Storage I/O nodes equipped with flash storage, while PapyrusKV [1] is designed to leverage non-volatile memory (NVM) in HPC systems. Future HPC architectures are expected to have hierarchical, heterogeneous resources such as DRAM, NVM, and high-bandwidth memory (HBM). BESPOKV seamlessly supports the use of different datalets, each of which can be tuned for different memory and storage architecture. BESPOKV's proxy-based design may add performance overhead with an additional layer in theory, but we found them they remain small during our evaluation.

Second, BESPOKV enables new HPC services for emerging workloads such as deep learning and massive IoT data processing: (1) Data layout: While existing KV solutions are rigid/fixed for one setting, BESPOKV allows storing data in different datalets, adapt and switch datalets as needed, and thus can handle diverse characteristics of new data workloads. For example, a datalet using B-tree as main data structure is better suited for read-intensive workloads (e.g., deep learning), while Log Structured Merge (LSM) tree based datalet is a better choice for write-intensive workloads due to high write amplification and no fragmentation. (2) Multi-tenancy and geo-distribution: IoT applications increasingly require multi-tenancy support, e.g., smart road big data used by different applications. Different tenant would require different consistency and topologies. Even for a single tenant the topology requirements may change. For example, simple MS topology may be sufficient for sensors deployed in one building but as the scale of deployment increases, AA may become more beneficial. Existing systems do not provide such support. (3) Low latency: deep learning queries require ultra low latency to take advantage of in-memory KV storage. For this purpose, we added support for DPDK kernel bypassing in BESPOKV.

This paper makes the following contributions:

- We propose a novel distributed KV store architecture that follows best architectural practices such as decoupling of control and data planes. Decoupling allows BESPOKV to transparently turn a user-provided (single-server) datalet into scalable, fault-tolerant distributed KV stores. Such specialization will enables innovative uses of KV stores in HPC applications, especially for emerging applications that utilize KV-friendly workloads. Our implementation of BESPOKV is publicly available at https://github.com/tddg/bespokv.
- We demonstrate that BESPOKV can be easily extended to offer advanced features such as range query, per-request consistency, polyglot persistence, and more. To the best of our knowledge, BESPOKV is first to support a seamless on-the-fly topology/consistency adaptation. As examples, we present a novel mechanism to make transitions from MS+EC to MS+SC, and from AA+EC to MS+EC. We also present several use cases to show effectiveness of BESPOKV.
- We deploy BESPOKV-enabled distributed KV stores in a local testbed as well as in a public cloud (Google Cloud Platform [23]) and evaluate their performance. Using five (two new and three existing) datalets, We show that with all the aforementioned benefits, BESPOKV-enabled distributed KV stores scale horizontally and performs comparable (and sometimes 1.2× to 2.6× better) to state-of-the-art distributed KV stores.

2 CHALLENGES

Several challenges arise when designing BESPOKV to meet the competing goals of compatibility, versatility, modularity, and performance.

Compatibility. BESPOKV strives to transparently make a non-distributed KV store into a distributed one. It should be easy to use, such that a developer simply "drops" the non-distributed version of the store into BESPOKV; in turn, BESPOKV will automatically clone and convert the store into various types of highly scalable and reliable distributed clusters.

However, in reality, every datalet is different, resulting in compatibility issues. Moreover, KV stores use different communication protocols. For instance, Redis's protocol is different from Cassandra's. This implies that BESPOKV's communication substrate should be designed to understand the basic message semantics, e.g., request routing. We describe this in Section 3.1.

Versatility. Due to the diversity of data storage and retrieval requirements, almost all the points on the cluster topology (MS, AA, etc.), consistency (strong, eventual, etc.), replication, and fault tolerance spectrum are valid. However, existing systems only support a fixed single design point, which limits flexibility and adaptability. Therefore, BESPOKV architecture should be versatile enough to cover various design options, and be flexible to support reconfiguration.

Different storage applications implement their distributed management and protocols with preference on diverse dimensions such as cluster topology and consistency. To support applications with tradeoffs among these different dimensions through a generic framework, one should ensure that each configurable dimension has a clear boundary and well defined interface. Hence, different dimensions can be seamlessly combined with each other to form a highly versatile choice of options for application developers. Moreover, the distributed network architecture should be flexible enough to support these wide range of options. Section 3 presents this aspect of BESPOKV's versatile architecture.

Modularity. Building various design options using different implementations is simply a matter of putting in more engineering effort and not as challenging. In fact, such a naive monolithic redesign approach would essentially be similar to the current approach of per-application implementations. Instead, BESPOKV should be designed in a modular fashion, which makes it possible to reuse a previously developed component. For instance, a controlet supporting MS+SC or AA+EC can be reused for multiple backend data stores. Furthermore, the modules in BESPOKV should be expandable to meet the ever-growing needs for advanced features.

Performance. Achieving the above goals is the major focus of our work. However, BESPOKV should be realized without

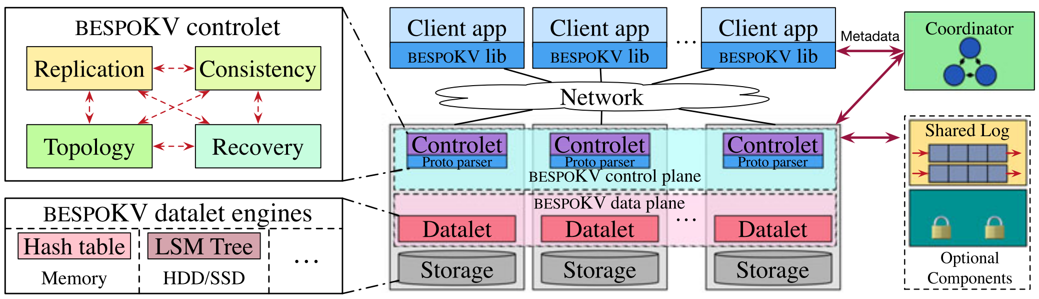


Fig. 2. BESPOKV architecture and the interactions between components. LSM Tree: Log-structured merge-tree. DLM: Distributed lock manager.

sacrificing performance. Design choices for protocol handling, network architecture, and diverse components should be carefully made with efficiency in mind.

3 BESPOKV DESIGN

In this section, we describe the design of BESPOKV and how it provides compatibility, versatility, modularity, and high performance for supporting distributed KV stores. Fig. 2 shows the overall architecture of BESPOKV comprising five modules: datalet, controlet, coordinator, client library, and optional components. A collection of datalets form the data plane, the rest of the modules makes up the control plane.

Datalet is supplied by the user and responsible for storing data within a single node. Datalet should provide the basic I/O interfaces (e.g., Put and Get) for the KV stores to be implemented. We refer to this interface as the datalet API. For example, a user can develop a simplest form of in-memory hash table. Users can also mix and match datalets with each datalet using a different data structure.

Controlet is supplied by BESPOKV and provides a datalet with distributed management services to realize and enable the distributed KV stores associated with the datalet. The controlet processes client requests and routes the requests to the associated entities: e.g., to a datalet for storing data. BESPOKV provides a default set of controlets, and allows advanced users to extend and design new controlets as needed for realizing a service that may require specialized handling in the controlet.

BESPOKV allows an arbitrary mapping between a controlet and a datalet. A controlet may handle N (≥ 1) instances of datalets, depending on the processing capacity of the controlet and its datalets, and can leverage physical resource (datacenter) heterogeneity [24], [25] for better overall utilization. For instance, a controlet running on a high-capacity node may manage more datalet nodes than a controlet running on a low-capacity node. For simplicity, we use one-to-one controlet–datalet mappings in the rest of the paper.

Coordinator provides three main functions. (1) It maintains the metadata regarding the whole cluster topology and provides a query service as a metadata server. (2) It tracks the liveness of the cluster by exchanging periodic heartbeat messages with the controlets. (3) It coordinates failover in case of a node failure. The coordinator can run on separate node or alongside other controlets.

BESPOKV implements the coordinator on top of ZooKeeper [26] for better resilience. Similar to designing specialized controlets, advanced users have the option to design customized coordinators if needed. It is also possible to design a new coordinator as a special form of controlet from scratch using the BESPOKV-provided controlet programming abstraction as shown in Section 3.2. Nonetheless, because it is widely used across many KV stores, BESPOKV includes the coordinator as a default module in the control plane.

Client library is provided by BESPOKV and used by the client applications to utilize the services created by BESPOKV. The library provides a flexible means for mapping data to controlets. The client application uses the library interface to consult with the coordinator and fetch data partitioning and mapping information, which is then used to route requests to appropriate controlets. BESPOKV allows different developers to choose their own partitioning techniques such as consistent hashing and range-based partitioning.

Optional Components. BESPOKV provides two optional components facilitating the controlet development: 1) a distributed lock manager (DLM) for a locking service, and 2) a Shared Log for an ordering service. One can build such a distributed management service as a special form of controlets from the scratch, but given its common use in distributed KV store development, BESPOKV imports existing solutions (e.g., Redlock [27] for DLM, and ZLog [28], [29], [30] for Shared Log) and provides interface libraries (Section 3.2, Table 4).

3.1 Data Plane

A collection of datalets running on different distributed nodes form the data plane for BESPOKV. A single-server datalet is completely unaware of other datalets.

Datalet Development. BESPOKV supports multiple backends. Users can make use of off-the-shelf single-serve data stores such as Redis [18], SSDB [31], and Masstree [21]. In addition, BESPOKV provides datalet templates based on commonly used data structures: currently, a hash-table-based *tHT*, a log-based *tLog*, and a tree-based *tMT*. For the ease of development, BESPOKV furnishes an asynchronous event-driven network programming framework in which developers can design new datalets, starting from existing templates. We evaluate the reduced engineering effort in Section 8.

APIs and Protocol Parsers. For compatibility and modularity, BESPOKV provides a clean set of datalet APIs (between controlet and datalet) and client APIs (between client app and client library). Table 3 presents example datalet and client APIs. As these APIs are consistent with existing I/O interfaces of existing KV stores. Datalet developers can adopt them in a straightforward manner to enable distributed services. This is much easier than library-based replication solutions such as Vsync [22] where developers should learn complex new APIs.

TABLE 3
APIs to `Put`, `Get`, and `Del` a KV Pair

Datalet API (provided by application developers)	
`Put(key, val)`	Write the `{key,val}` pair to the datalet
`val=Get(key)`	Read `val` of `key` from the datalet
`Del(key)`	Delete `{key,val}` pair from the datalet
Client API (provided by BESPOKV)	
`CreateTable(T)`	Create a table `T` to insert data
`Put(key, val, T)`	Write the `{key,val}` pair to table `T`
`val=Get(key, T)`	Read `val` of `key` from table `T`
`Del(key, T)`	Delete `{key,val}` pair from table `T`
`DeleteTable(T)`	Delete table `T`

Datalet and Client APIs are for using pre-built controlets.

To offer compatibility and be able to understand application protocols to process incoming requests properly, BESPOKV's communication substrate supports two options. (1) It provides a BESPOKV-defined protocol using Google Protocol Buffers [32]. This option is suitable for new datalets and is preferred due to its ease of use and better programmability. (2) BESPOKV allows developers to provide a parser for their own protocols. This option is mainly available for porting existing datalets such as Redis or SSDB.

3.2 Control Plane

BESPOKV provides a set of pre-built controlets that provide datalets with common distributed management. Given a datalet, BESPOKV makes distributed KV stores immediately ready-to-use. Developers can also extend these pre-built controlets or design new ones from scratch for advanced services.

Pre-Built Controlets. BESPOKV identifies four core components for distributed management, and provides pre-built controlets that support common design options in existing distributed KV stores. The choice is based on our comprehensive study of existing systems that revealed three key observations: (1) cluster topology, consistency model, replication, and fault tolerance generally define distributed features of KV stores; (2) for the topology, MS and AA are common; and (3) for the consistency model, SC and EC are popular. Detailed descriptions of exemplary controlets supporting MS+SC, MS+EC, AA+SC, and AA+EC options follow in Section 4.

Controlet Development. To support advanced users and new kinds of services, BESPOKV provides an asynchronous event-driven network programming framework for controlet development as well. For each event (e.g., `Put` request, timeout, etc.), developer can define event handlers to instruct how the controlet should process the event to enable versatile distributed management services in the control plane. The aforementioned pre-built controlets indeed consist of a set of pre-defined event handlers for common distributed services.

Discussion. Load imbalance due to hot keys (i.e., hotspots) can be solved by integrating a small metadata cache at BESPOKV's client library to keep track of hot keys [33]; once the popularity of hot keys exceeds a certain predefined threshold, client library replicates this key on a shadow server that is rehashed by adding a suffix to the key. In fact, our proxy-based architecture naturally fits for adding a controlet-side small cache or data migration/replication for load balancing purpose [34], [35], [36], [37], [38].

TABLE 4
APIs for Events, Shared Log, DLM, and Coordinator for New Controlet Development

Events API (provided by BESPOKV)	
`Register(c,e,cb)`	Register basic event `e` for conn `c` to call func `cb`
`Enable(c,e)`	Enable event `e` to be triggered onc time for conn `c`
`On(e,cb)`	Register extended event `e` to call func `cb`
`Emit(e)`	Emit event `e`
Shared Log API (provided by BESPOKV)	
`CreateLog`	Creates a new log instance `L`
`PutSharedLog(m, L)`	Append message `m` to log `L`
`AsyncFetch(L)`	Asynchronous read from log `L`
DLM API (provided by BESPOKV)	
`Lock(key)`	Acquire lock on `key`
`Unlock(key)`	Unlock `key`
Coordinator API (provided by BESPOKV)	
`LogHeartbeat(c,d)`	Log heartbeat for controlet `c` & datalet `d`
`map=GetShardInfo(s)`	Get controlet & datalet list for shard `s`
`c=LeaderElect(s)`	Elect new Master controlet for shard `s`

Due to space limitation, we list only important APIs.

Control Plane Configuration. To configure the system, each controlet takes as input (1) a JSON configuration file that specifies the basic system deployment parameters such as topology, consistency model, the number of replicas, and coordinator address; and (2) a datalet host file containing the list of datalets to be managed. BESPOKV loads the runtime configuration information at the coordinator, which serves as the query point for the client library and controlets to periodically retrieve configuration updates. Any change in configuration at runtime (e.g., topology/consistency switch) results in replacing old controlets with new ones. We describe dynamic adaptation mechanisms in Section 5 in detail.

Controlet Programming Abstraction. BESPOKV uses asynchronous event-driven programming model to achieve high throughput. For each event (e.g., incoming network input, timer, etc.), developers are asked to define event handlers to process the event. There are two types of events in BESPOKV: basic and extended events. Basic events represent pre-defined conditions. Developers can create their own extended events by using basic or existing extended events.

Other Controlet APIs. BESPOKV provides a set of libraries and APIs with common features for controlet development, shown in Table 4.

4 BESPOKV-BASED DISTRIBUTED KV STORES

BESPOKV, to be specific its control plane, transparently turns a user-provided single-server datalet to a scalable, fault-tolerant distributed KV store. Using hash-based *tHT* datalet and consistent hashing for the client library as an example, this section presents support for MS+SC, MS+EC, AA+SC, AA+EC and four examples to enable new forms of distributed services by combining existing controlets or extending ones.[2]

2. Please note that these examples present just one way to implement each combination. Controlet developers can easily implement their own versions.

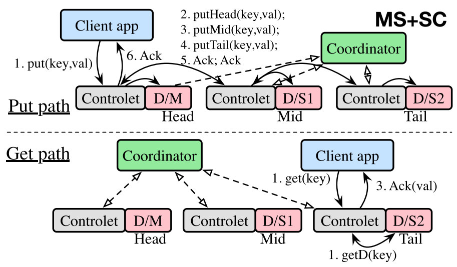


Fig. 3. `Put/Get` paths in MS+SC. `M` means master; `S`n means the nth slave; `D` means datalet.

4.1 Master-Slave & Strong Consistency

We start from a KV store supporting the MS topology with the SC model (MS+SC). Perhaps the simplest way to ensure SC is to rely on a locking mechanism using ZooKeeper [26] at the cost of serialization. However, alternative scalable designs exist such as chain replication (CR) [39], value-dependent chaining [10], and their variants. The pre-built BESPOKV controlet for MS+SC leverages CR algorithm. Our modular design allows BESPOKV to adopt other optimizations for CR [40], [41] as well, but so far we have not implemented those. The original CR paper describes the tail sending a message directly back to the client; but similar to CRAQ [41], our implementation lets the head respond after it receives an acknowledgment from the tail, given its pre-existing network connection with the client.

Example. Fig. 3 shows how MS+SC is implemented in BESPOKV. Here, clients route `Puts` to the head of the corresponding controlet–datalet chains via consistent hashing (step 1). The head controlet forwards the incoming `Put` request to its local datalet (step 2) and then to mid node (step 3), which forwards the request to its local datalet and then to tail (step 4). Tail first forwards the request to local datalet and then sends `Ack`back to mid, which sends `Ack` back to head (step 5). Once the head controlet receives the `Ack` from the mid, the head controlet marks the request completed and responds to the client (step 6). `Gets` are routed to the tail node of the corresponding chains. This provides the SC guarantee as clients are only notified of the successful completions of `Puts` after the data is persisted through the tail nodes.

Failover. In all cases (MS+SC, MS+EC, AA+SC, and AA +EC), when the coordinator detects a node failure using a periodic heartbeat message, it launches a new controlet–datalet pair in recovery mode on one of the standby nodes. The new controlet then recovers the data from one of the datalets.

In particular, for MS+SC using chain replication, the coordinator performs the chain recovery process and adds the new pair as the new tail to the end of the chain. The former chain recovery process depends on the location of the failure in the replica chain as follows. If a middle node fails, the coordinator notifies the head controlet to skip forwarding requests to the failed node. In case the tail node fails, the coordinator informs the head controlet to skip forwarding requests to the tail datalet and temporarily marks the second to the last node as the new tail so that future incoming `Get` requests can be redirected properly. If the head node fails, the coordinator appoints the second node in chain as the new head, and updates the cluster metadata. Upon seeing the change, the clients redirect future writes to the new head. Every node maintains a list of requests received but not yet processed by the tail, which is used to resolve in-flight requests [39], [41].

4.2 Master-Slave & Eventual Consistency

BESPOKV's pre-built controlet takes a simple approach to support MS+EC where the master copies the data to slaves asynchronously.

Example. Fig. 4a shows an example for MS+EC. Here, upon receiving an incoming `Put` request (step 1), the master node commits the request to the local datalet (step 2) before it sends an acknowledgement back to the client (step 3). Unlike the previous SC case, the master does not wait until the propagation finishes.[3] Subsequently, BESPOKV provides EC by asynchronously forwarding `Put` requests to other datalets (step 4).

Failover. Upon a node failure, the coordinator launches a new controlet–datalet pair, and then the new controlet recovers the requests from another datalet. For MS+EC, the new pair is added as a slave. If the master node fails, the coordinator promotes one of the slave nodes to master after a leader election process. The coordinator then updates the cluster topology metadata so that future incoming writes can be routed to the new master, similar to the case of head failure in MS+SC.

4.3 Active-Active & Strong Consistency

Supporting AA and SC is expensive in general. AA allows multiple nodes to handle `Put` requests and SC requires global ordering (serialization) between them. Thus, CR-like optimization is not applicable under AA. For simplicity and comparison purposes, the current BESPOKV's AA+SC controlet takes the distributed locking based implementation, using the DLM library (Section 3.2). For performance improvement, optimistic concurrency control [42] and inconsistent replication [9] can be added. Instead of using DLM, one can also enable SC using a Shared Log to maintain a global and sequential order of concurrent requests, which we used for AA+EC later in a relaxed manner.

Example. Fig. 4b shows a DLM-based AA+SC example. Clients' `Put` requests are routed to any controlet (step 1 and step 2). Concurrent `Puts` from another client (step 2 in our example) are synchronized via the distributed locking service. The first receiving controlet acquires a write lock (step 3) on the key and updates all the relevant datalets (step 4 & 5), releases the lock (step 6), and finally acknowledges to the client (step 7). For a `Get`request, the controlet that receives the request acquires a read lock on that key, reads the value from the local datalet, releases the lock, and then sends a response back to the client.

*Failover.*Like the previous cases, when a node fails the coordinator launches a new controlet–datalet pair. The new controlet then performs data recovery from another datalet. As AA+SC uses locking, ensuring SC for the new node and adding it as an active node are trivial because all writes are synchronized using locks. However, deadlock freedom should

3. This way at least one datalet is written straight away as in Cassandra [7]. An alternative design choice is to forward the request to more than one datalet and then acknowledge back. However, this decision solely depends on the type of eventual consistency that is desired.

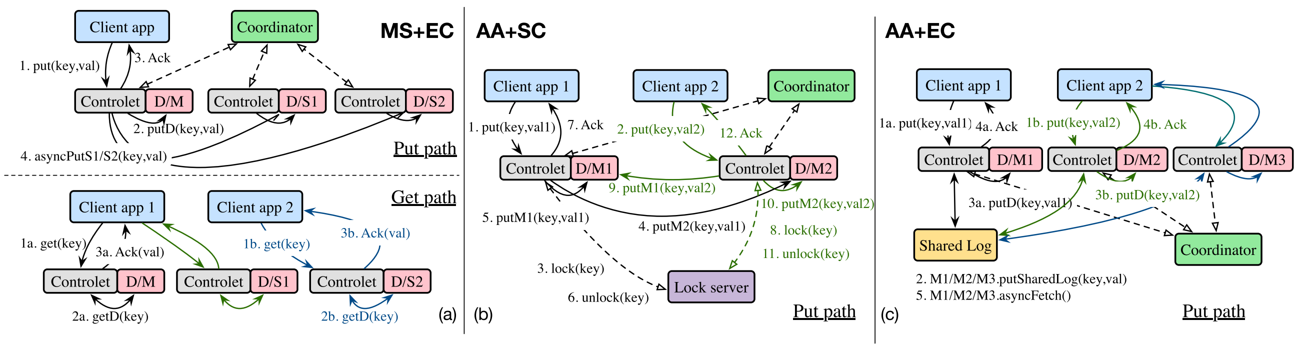


Fig. 4. The `Put/Get` paths in MS+EC (a), AA+SC (b), and AA+EC (c). The `Get` path is same in all three, except in AA+SC, where the difference is that each `Get` needs to acquire a read lock before proceeding. `Mn` means the nth master.

be guaranteed. Thus, BESPOKV enforces that locks are released after a configurable period of time. If a controlet fails after acquiring a lock, the lock is auto-released after it expires. Note that if a lock is auto-released, but a controlet has not failed and was simply unresponsive for a while, it is terminated to ensure proper continuation of operations. Also, one of the master nodes cleans up the in-flight requests.

4.4 Active-Active & Eventual Consistency

For an AA topology, relaxed data consistency is more widely used in practice for performance as in Dynamo [6], Cassandra [7]), and Dynomite [19]. In particular, these systems use gossip-based protocols and provide a weaker data consistency model, e.g., acknowledging back to the client if a `Put` request is written to one node, N nodes, or a quorum [43].

In order to ensure EC, when multiple masters receive concurrent PUT requests, AA should be able to resolve conflicts and agree on the global order of them, unlike MS where one master gets all the writes. In this sense, Dynomite does not support (a strict form of) EC when conflicting PUT requests arrive within a time period less than the latency of replication [44].

To address this issue, BESPOKV's AA+EC controlet uses a Shared Log to keep track of the request ordering. From the Shared Log, asynchronous propagation of writes occur to support EC. One disadvantage of this approach is that we need to scale the Shared Log setup as BESPOKV scales. Alternative approach is to add anti-entropy/reconciliation [45].

Example. Fig. 4c depicts how BESPOKV supports AA+EC. In AA, clients can route `Get/Put` to any of the master controlets (step 1a). On a `Put`, the receiving controlet (in our example the leftmost one) writes to the Shared Log first (step 2a), commits the request on its local datalet (step 3a), and then responds back to the client (step 4a). All the controlets asynchronously fetch the request (step 5). `Gets` can be handled by any of the corresponding controlets by retrieving the data from their local datalets. The duration to keep the requests in Shared Log is configurable.

Failover.For AA+EC, the failover is handled like with MS +EC, except that leader election is not needed in this case.

5 Dynamic Adaptation to Consistency and Topology Model Changes

Separating the control and data planes bring another benefit: BESPOKV-enabled distributed KV stores can seamlessly adapt to consistency and topology model changes at runtime by switching the controlets while keeping the datalets unchanged. At a high level, upon a consistency and/or topology change request, Coordinator launches a new set of controlets that will provide new services. Two old and new controlets are mapped to one datalet during the transition phase. The old controlet provides the old service with no downtime, and forwards some requests to the new controlet so that it can prepare the new service. When the transition completes, the new controlet takes over the old one. The transition protocol differs per each case. BESPOKV supports any transition between four aforementioned topology and consistency combinations, among which we describe two interesting cases in detail. Section 9.4 presents the experimental results on this aspect.

5.1 Transition From MS+EC to MS+SC

To make a transition from EC to SC, the master node needs to make sure that all the `Put` requests 1) that have arrived before the transition starts and 2) that arrive during transition are fully propagated to the slave nodes. For the former, the old master keeps flushing out any pending propagation. For the latter, the old master forwards an incoming `Put` request to the new master controlet which uses chain replication for SC, instead of propagating it asynchronously. When there is no more pending propagation left in the old controlet, the transition is over. SC guarantees will be enforced after the transition has completed. During the transition, any node may respond to `Get` requests, providing EC guarantee. This means that a `Get` request, even after the reconfiguration was requested, may experience EC until the transition is over. As controlet developers are responsible for developing the transition functionality for the various consistency/topology modes. A controlet developer can choose an alternative route to fence all writes as soon as the reconfiguration is requested so that all reads observe the same and latest applied value.

Fig. 5a shows transition from MS+EC to MS+SC.[4] Client 1 sends a `Put` request (Step 1a) to the old master controlet C1. A concurrent `Get` request (Step 1b) from Client 2 gets serviced as it used to be. The old master forwards `Put` request (Step 2) to the new master controlet which guarantees SC.

4. Reverse transition from MS+SC to MS+EC is trivial as the new master just needs to start using asynch. propagation instead of chain replication.

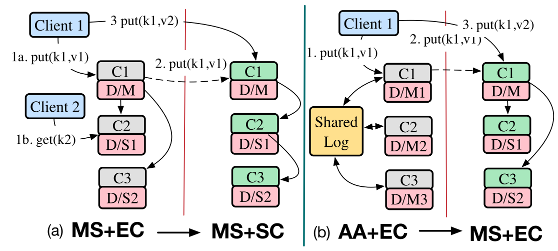


Fig. 5. Transition: MS+EC to MS+SC and AA+EC to MS+EC.

When the new master completes its chain replication process, it acknowledges the old master, which in turn acknowledges Client 1. When the transition completes, a Put request (Step 3) is routed to the new master controlet.

5.2 Transition From AA+EC to MS+EC

In AA+EC, any active node can get a Put request. To maintain a global ordering between concurrent Puts, an active node relies on the Shared Log that propagates Puts to the other nodes on its behalf. On the other hand, in MS +EC, only the master node gets Put requests and is in charge of propagating them to the slaves. Therefore, the key operation in the transition from AA+EC to MS+EC is to move the role of propagating Puts from the Shared Log to the new master. To this end, when the transition starts, the new master node takes the in-flight Puts that have not been propagated yet from the Shared Log and starts propagating them by itself. When an old active controlet receives a Put request during transition, it does not consult with Shared Log, but forwards the request to the new master node which will eventually propagates the request. The Get requests are not affected. Fig. 5b shows an example where a Put request (Step 1) is forward to the new master (Step 2) during transition. When the transition completes, a Put request (Step 3) is serviced by the new master. The transition from MS+EC to AA+EC can be supported by the reverse step order.

6 Extensions to KV Stores

BESPOKV is immediately ready-to-use for popular distributed KV store use cases. If desired, BESPOKV's control plane can be extended to enable new forms of distributed services by combining existing controlets or extending ones. This section demonstrates four examples. We evaluated performance of Scan requests (range query) in Section 9.2, and the next two per-request consistency and polyglot persistence in Section 9.5.

6.1 Range Query

We support range query or scan operations as follows. For datalets, the Masstree-based *tMT* template is used and extended to expose a range query API such as GetRange (Start, End). The client library supports range-based partitioning, e.g., dividing the name space by alphabetical order (e.g., A-C on one node, D-F on another node, and so on). The controlet divides a client request into sub-requests and forwards the sub-range query requests to corresponding datalets that store the specified range.

6.2 Per-Request Consistency

We extend the client library GET API to support consistency/topology specification on a per-request basis. For instance, under MS+SC, if the user specifies a lower value of consistency level, GETs can go to any of the replicas, thus only eventual consistency is guaranteed.

6.3 Polyglot Persistence

A use case for KV store is to support businesses that may be divided into different components, and each component requires its own private data storage. BESPOKV supports such polyglot persistence [46] by launching custom controlets for cross-app lazy synchronization (eventual consistency).

6.4 Other Topologies

BESPOKV also supports an AA-MS hybrid topology by configuring an MS topology for each shard on top of the logical AA overlay. Similarly, a P2P-like topology can also be enabled by allowing clients to send a request to any controlet, which then routes the request to the actual controlet that manages the requested data. In this case, a controlet needs to maintain a routing map similar to a finger table [47] to determine the location of keys.

Variants of AA. BESPOKV can support Adding routing flexibility to the AA topology is straightforward. Clients can simply send a request to any of the controlets (or use some load balancing techniques such as round robin), which then routes the request to the actual controlet–datalet that holds the requested data. The extra logic we need to add into controlets is a routing map similar to a finger-table [47] to determine the location of keys.

6.5 Other Consistency Models

Our Shared Log-based asynchronous fetches for eventual consistency can be easily configured to support bounded staleness. Developers simply specify a T-sec polling period, so that clients are guaranteed not to see the stale data for more than T sec. Similarly, causal consistency can also be supported if the controlet serving the Get request fetches all the pending data from the shared log and communicates it to other controlets before replying back to the client.

7 BESPOKV's Use Cases

7.1 Hierarchical and Heterogeneous Storage of HPC

HPC big data problems require efficient and scalable storage systems, but load balancing I/O servers at scale remains a challenge. Statistical analysis [48] and Markov chain model [49] have been used to predict shared resource usage. A KV store can be used to collect runtime statistics from HPC storage systems for accurate prediction. However, existing KV stores are designed for one type of storage architecture (in-memory, SSD, NVM, etc.), leading to suboptimal performance.

BESPOKV supports the use of different datalets to store replicas of a KV pair, where each of these datalet can be tuned for different memory and storage architecture. By doing so, BESPOKV unifies multiple data abstraction together and enables multifaceted view on shared data with configurable consistency and topology. Fig. 6 shows an example of

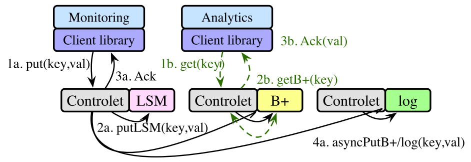


Fig. 6. `Put/Get` paths in MS+EC for HPC monitoring to perform I/O load balancing.

how BESPOKV unifies three different data abstractions – a log-structure merge-tree, Masstree, and log, and transparently provides master-slave topology (MS) and eventual consistency (EC). Data is replicated asynchronously in batch mode from master to slaves. In this design, it is possible to run applications with different properties (e.g., write-intensive and read-intensive apps) together.

There are two advantages of this design architecture. First, different applications can choose datalet that best suits their need. As a typical use case, monitoring data collection is write-intensive workload, and prefers a scalable solution that is able to persist all data on persistent storage. Whereas, analytical models incur read intensive workload which could benefit from high read throughput. Second, replicas in different datalets are not evicted simultaneously. For instance, a replica of a KV pair may evict from in-memory based datalet due to size restriction but another replica may stay longer in NVM/SSD based datalet or stay forever in log based datalet that uses HDD.

7.2 Building Burst Buffer File Systems

Burst buffer file systems are becoming an indispensable framework to quickly absorb application I/O requests in exascale computing [50], [51], [52], [53]. Many burst buffer file systems adopt KV stores to manage file system metadata. BESPOKV allows to develop similar file systems with less development effort. In particular, the dynamic and flexible nature of BESPOKV well suits with ephemeral burst buffer file systems [50]. An ephemeral burst buffer file system has to be dynamically constructed and destroyed within compute nodes assigned to a corresponding job. In such a scenario, BESPOKV can quickly initialize the distributed KV store for storing file system metadata.

Furthermore, BESPOKV also allows to dynamically tune the file system behavior. For instance, it is often preferred to relax the strong POSIX consistency semantics for certain HPC workloads (e.g., checkpointing) to maximize the parallel I/O performance [54]. BESPOKV can simplify the development of such a file system, because it natively supports an instantiation of the distributed KV store with desired consistency and reliability levels.

7.3 Accelerating the File System Metadata Performance

KV store is also widely adopted to enhance the performance of file system metadata operations in HPC systems. Metadata performance is one of the major limitations in HPC parallel file systems. A popular approach to address this limitation is to stack up a special file system atop the parallel file system [55], [56], [57]. The stacked file system then quickly absorbs the metadata operations by exploiting a distributed KV store. BESPOKV can accelerate the development of such a stacked file system (evaluated in Section 9.2). Specifically, BESPOKV allows to explore various datalets in backend, and also dynamically tune the file system behavior to comply with the desired performance, consistency and reliability levels.

7.4 Resource and Process Management

KV store has also been used to aid the resource and process management in HPC systems [2], [58]. BESPOKV can help develop an advanced job launching system, because it can adapt to different topology and consistency models on the fly. For example, the simple MS topology may be sufficient for handling jobs on a single cluster, but the AA topology may become more suitable when jobs spans multiple clusters (evaluated in Sections 9.2 and 9.4).

8 BESPOKV IMPLEMENTATION

Current implementation of BESPOKV consists of $\tilde{}69k$ lines of C++/Python code without counting comments or blank lines. Except controlets, BESPOKV consists of five components. (1) *Control Core* implements the control plane backbone with support for event and message handling. (2) *Client library* helps clients route requests to appropriate controlets, and is extended from libmc [59], a in-memory KV store client library. (3) *Coordinator* uses ZooKeeper [26] to store topology metadata of the whole cluster and coordinates leader elections during failover. It includes a Python-written failover manager that directly controls the data recovery as well as handling BESPOKV process failover. (4) *Lock server APIs* implement two lock server options—ZooKeeper-based [60] and Redlock-based [27]. (5) *Shared Log handler* is implemented using ZLog [28], based on CORFU.

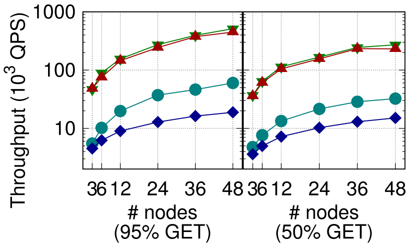
The BESPOKV prototype has four pre-built controlets as described in Section 4. All controlet shares the sample event-handling controlet template of 150 LoC. In addition, BESPOKV supports multiple backend datalets with protocol parsers. Using the common datalet template of 966 LoC, we implemented three new datalets with a Protobuf-based [32] parser: *tHT*, an in-memory hash table; *tLog*, a persistent log-structured store that uses *tHT* as the in-memory index; and *tMT*, a Masstree-based [61] store. In addition, BESPOKV are compatible with existing single-server KV stores SSDB [31] and Redis [18] that use a simple text-based protocol parser. With protocol parsers, we refer them *tSSDB* and *tRedis*, respectively. Docker based BESPOKV is partially supported right now. We plan to use Kubernetes [62] to simplify deployment in near future.

Using the template-based design approach, we note that for developers (with few years of C/C++ programming experience) non familiar to BESPOKV it took almost three and six person-days time to develop datalet and controlet, respectively. This underscores BESPOKV's ability to ease development of distributed KV stores.

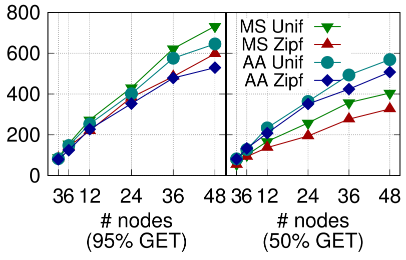
9 EVALUATION

Our evaluation answers the following questions:

- Are BESPOKV-enabled distributed KV stores scalable (Section 9.2), adaptive to topology and consistency changes (Section 9.4), and extensible (Section 9.5)?



(a) SC. On log scale.



(b) EC.

Fig. 7. BESPOKV scales *tHT* horizontally.

- How does BESPOKV compare to existing proxy-based (Section 9.6), and natively-distributed (Section 9.7) KV stores?
- How well BESPOKV handles a node failure? (Section 10)

9.1 Experimental Setup

Testbeds and Configuration. We perform our evaluation on Google Cloud Engine (GCE) and a local testbed. For larger scale experiments (Sections 9.2, 9.3, 9.4, 9.5, and 9.6), we make use of VMs provisioned from the `us-east1-b` Zone in GCE. Each controlet–datalet pair runs on an `n1-standard-4` VM instance type, which has 4 virtual CPUs and 15 GB memory. Workloads are generated on a separate cluster comprising nodes of `n1-highcpu-8` VM type with 8 virtual CPUs to saturate the cloud network and server-side CPUs. A 1 Gbps network interconnect was used.

For performance stress test (Section 9.7) and fault tolerance experiments (Section 10), we use a local testbed consisting of 12 physical machines, each equipped with 8 2.0 GHz Intel Xeon cores, 64 GB memory, with a 10 Gbps network interconnect. The coordinator is a single process (backed-up using ZooKeeper [26] with a standby process as follower) configured to exchange heartbeat messages every 5 sec with controlets. We deploy the DLM, Shared Log, Coordinator and ZooKeeper on separate set of nodes. BESPOKV's coordinator communicate with ZooKeeper for storing metadata.

Workloads. We use two workloads obtained from typical HPC services: job launch, and I/O forwarding and three workloads from the Yahoo! Cloud Serving Benchmark (YCSB) [63].

We use approach similar to [2] to generate HPC workloads. The job launch workload is obtained by monitoring the messages between the server and client during a MPI job launch. Control messages from the distributed servers are treated as `Get` whereas results from the compute nodes back to the servers as `Put`. The I/O forwarding workloads is generated by running SeaweedFS [64], a distributed file system which supports KV store for metadata management. The clients first create 10,000 files, and then performs reads or writes (with 50 percent probability) on each file. We collect the log of the metadata server. We extend these workloads several times until reaching 10M requests with the goal to reflect the time serialization property of the obtained messages.

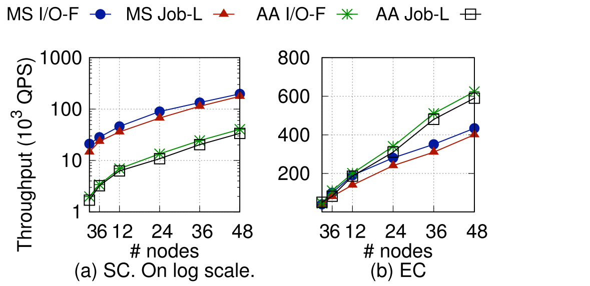


Fig. 8. BESPOKV scales HPC workloads.

For YCSB we use an update-intensive workload (`Get`:`Put` ratio of 50%:50%), a read-mostly workload (95 percent `Get`), and a scan-intensive workload (95 percent `Scan` and 5 percent `Put`). All workloads consist of 10 million unique KV tuples, each with 16 B key and 32 B value, unless mentioned otherwise. Each benchmark process generates 10 million operations following a balanced uniform KV popularity distribution and a skewed Zipfian distribution (where $\text{Zipfian constant} = 0.99$). The reported throughput is measured in terms of thousand queries per second (kQPS) as an arithmetic mean of three runs.

9.2 Scalability

In this test, we evaluate the scalability of the BESPOKV-enabled distributed KV store using four datalets: tHT and tLog, as examples of newly developed datalets; and tSSDB and tMT, as representatives of existing persistent KV stores. Fig. 7 shows the scalability of BESPOKV-enabled distributed tHT. We measure the throughput when scaling out tHT from 3 to 48 nodes on GCE. The number of replicas is set to three. We present results for all four topology and consistency combinations: MS+SC, MS+EC, AA+SC, and AA+EC. For all cases, BESPOKV scales tHT out linearly as the number of nodes increases for both read-intensive (95 percent `Get`) and write-intensive (50 percent `Get`) workloads. For SC, MS +SC using chain replication scales well, while AA+SC performs worse as expected in locking based implementation. For EC, the results show that our EC support scales well for both MS+EC and AA+EC. Performance comparison to existing distributed KV stores will follow in Section 9.7.

Fig. 8 shows similar trend for HPC oriented workloads. We again observe that MS outperforms AA for SC whereas the trend is opposite for EC where AA performs better than MS. We also observe that performance of I/O forwarding is slightly better than Job launch. This is because I/O forwarding workload has 12 percent more reads than Job launch with `Get`:`Put` ratio of 62%:38%.

Fig. 9 shows the scalability when varying the number of nodes from 3 to 48, with tSSDB, tLog, and tMT as datalet. Due to space constraints, we only present the result with the MS +EC configuration. While enabling eventual consistency with fault tolerance, BESPOKV provides good scalability for all three. In terms of performance, tMT is an in-memory database and thus outperforms both tLog and tSSDB which persist data on

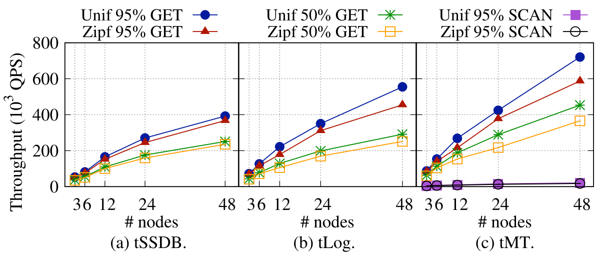


Fig. 9. BESPOKV scales *tSSDB*, *tLog*, and *tMT* with MS+EC.

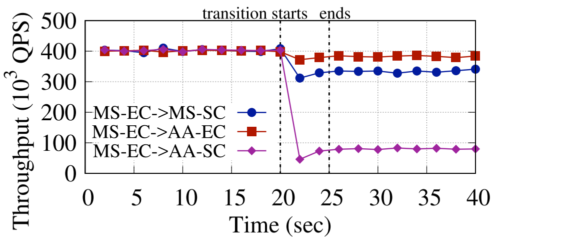


Fig. 11. BESPOKV seamlessly adapts service from MS-EC to MS-SC, AA-EC, and AA-SC.

disk. It is as expected that the throughput of `Scans` (range queries) is much lower than point queries. A 48 node tMT cluster gives $18k$ QPS on Zipfian 95 percent `Scan`, while Uniform yields slightly higher throughput ($21k$). Interestingly, this test covers a potential use case of BESPOKV+tLog for flash storage disaggregation, where users can exploit the scale-out capacity of an array of fast SSD (flash) devices/nodes with low-latency datacenter network [65], [66].

9.3 Impact of Varying Replication Factor

Next, we analyze the impact of varying the replication factor on performance. Fig. 10 shows the average throughput of an 8-shard cluster when varying the number of replicas from 1 to 3.

For cluster configurations with EC under read-intensive (uniform and Zipfian 95 percent `Get`) workloads, a larger replication factor results in higher performance. This is because there are more nodes that can serve `Get` requests. The effect is more significant for uniform workload, as the load is more balanced.

On the other hand, for MS+SC, scaling the number of replicas does not improve performance. Performance stops scaling above 2 replicas under read-intensive workloads, and it actually degrades by a factor of 2 to 3 for write-intensive workloads. This is because BESPOKV uses chain replication to support MS+SC, and all `Puts` are going to the head controlets, which need to do more work as the length of the chain increases.

Increasing the number of replicas does increase the performance for AA+SC under read-intensive workloads but with limited improvement. This is due to the high cost of distributed locking. Zipfian workloads severely increase the lock contention at the lock server, leading to the observed performance drop.

9.4 Adaptability

We evaluate BESPOKV's adaptability in switching online consistency levels and topology configurations (Section 5). In all the tests we use 3 shards with a Zipfian workload of 95 percent `Get`. As shown in Fig. 11, the transition is scheduled to be triggered at 20 sec. The throughput drops to the lowest point for all three cases. This is because clients switch connection to the new controlets. Performance stabilizes in ~5 sec, because all the in-flight requests are handled during this process. We observe similar trends for other possible transitions that can be enabled by BESPOKV. This demonstrates BESPOKV's flexibility and adaptability in switching between different key designs & configurations. This also shows that BESPOKV is able to complete switching in extremely short time compared to existing solutions because BESPOKV does not require data migration or down time.

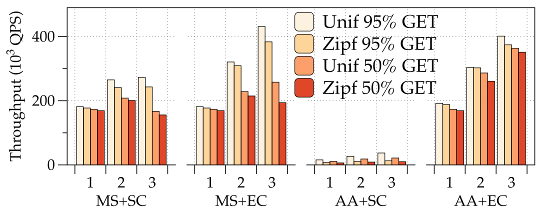


Fig. 10. Varying the number of replicas in BESPOKV.

9.5 Extensibility and New Services

As sketched in Section 4, BESPOKV can be extended to support new forms of distributed services. This section evaluates two examples: per-request consistency and polyglot persistence.

We evaluate the per-request consistency service (Section 6.2) under MS+SC and a Zipfian workload with a 25:75 percent ratio of SC:EC as the desired consistency. We observed the performance to be between MS+SC and MS +EC as shown in Fig. 7; for example, with 24 nodes, we obtain $\sim 300k$ QPS for 95 percent `Get` and $\sim 270k$ QPS for 50 percent `Get` workloads. We also evaluate the average latency of each request. With a weaker consistency requirement, the `GET` latency is 0.67 ms. We get an average of 1.02 ms latency with default strong consistency.

We test polyglot persistence (Section 6.3) by storing each replica in a different type of datalet. We use *tHT*, *tLog* and *tMT* in MS topology with eventual consistency. The performance of the resulting configuration under Uniform workload is very similar to the numbers in Figs. 7 and 9; for example, with 24 nodes, we obtain $375k$ QPS for 95 percent `Get` and $200k$ QPS for the 50 percent `Get` workload.

9.6 Comparison to Proxy-Based Systems

This section shows that BESPOKV can support new topologies and consistency models for existing single-server KV store, and them compares BESPOKV with two state-of-the-art Proxy-based KV stores. We test BESPOKV+Redis (*tRedis*) running in MS+SC, MS+EC and AA+EC modes, reusing SSDB's text-based protocol parser for Redis. We measure the throughput of *tRedis* on eight 3-replica shards across 24 nodes on GCE, and compare it with Dynomite [19] supporting AA+EC only, and Twemproxy [16] supporting MS+EC only. We perform each test at three different periods of time to capture interfernce caused by cloud-based multi tenancy.

Fig. 12 shows the throughput. BESPOKV enables new MS +SC ($\sim 500k$ QPS under Zipfian 95 percent `Get`) and AA+EC ($\sim 750k$ QPS under Zipfian 95 percent `Get`) configurations

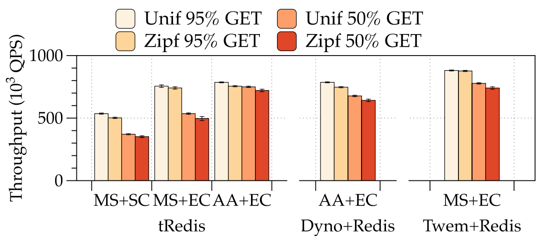


Fig. 12. BESPOKV adds MS+SC and AA+EC for Redis. Comparison with Dynomite (Dyno) and Twemproxy (Twem).

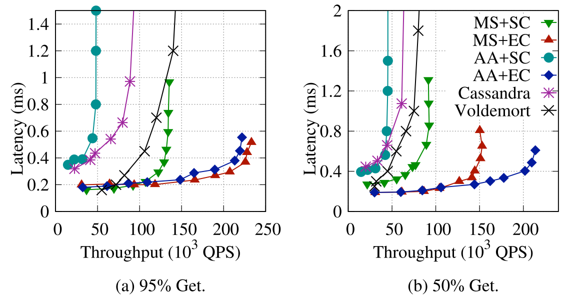


Fig. 13. Average latency versus throughput achieved by various systems under Zipfian workloads.

with reasonable performance. As expected, MS+SC is more expensive than MS+EC. Twemproxy is just a proxy to route requests using consistent hashing to a pool of backend servers. Hence, Twemproxy+Redis in supporting MS+EC performs slightly better than BESPOKV in supporting MS+EC.[5] However, we observed the same performance for Dynomite +Redis in supporting AA+EC configuration for Redis as BESPOKV in supporting AA+EC. The small error bars in Fig. 12 show that inherent multi tenancy effect of cloud-based environment is almost negligible.

9.7 Comparison to Natively-Distributed Systems

In this experiment, we compare BESPOKV-enabled KV stores with two widely used natively-distributed (off-the-shelf) KV stores: Cassandra [7] and LinkedIn's Voldemort [67]. These experiments were conducted on our 12-node local testbed in order to avoid confounding issues arising from sharing a virtualized platform. We launch the storage servers on six nodes and YCSB clients on the other four nodes to saturate the server side. The coordinator, lock server (only for AA +SC), ZLog (only for AA+EC), and ZooKeeper are launched on separate nodes. We use tHT as a datalet to show high efficiency of BESPOKV-enabled KV stores.

For Cassandra, we specify consistency level of *one* to make consistency requirements less stringent. Cassandra's replication mechanism follows the AA topology with EC [68]. For Voldemort we use a *server*-side routing policy, *all-routing* as the routing strategy, a replication factor of *three*, *one* as the number of reads or writes that can succeed without client getting an exception, and persistence set to *memory*.

Fig. 13 shows the latency and throughput for all tested systems/configurations when varying the number of clients

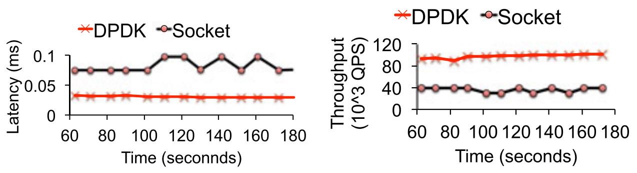


Fig. 14. Latency and throughput improvements by using DPDK.

to increase the throughput in units of kQPS.[6] For AA+EC, BESPOKV outperforms Cassandra and Voldemort. For read-intensive workload, BESPOKV's throughput gain over Cassandra and Voldemort is 4.5× and 1.6×, respectively. For write-intensive workload, BESPOKV's throughput gain is 4.4× over Cassandra and 2.75× over Voldemort. In this experiment Cassandra was configured to use persistent storage. However even using *tLog* as a datalet for BESPOKV(also uses persistent storage) we observed a throughput gain of 2.6× and 1.2× over Cassandra and Voldemort, respectively. We suspect that this is because Cassandra uses compaction in its storage engine which significantly effects the write performance and increases the read latency due to use of extra CPU and disk usage [69]. Voldemort uses the same design and both are based on Amazon's Dynamo paper [6]. Furthermore, our findings are consistent with Dynomite in terms of the performance comparison with Cassandra [69].

As an extra data point, we also see interesting tradeoffs when experimenting with different configurations supported by BESPOKV. For instance, MS+EC achieves performance comparable to AA+EC under 95 percent Get workload since both configurations serve Gets from all replicas. AA+EC achieves 47 percent higher throughput than MS+EC under 50 percent Get workload, because AA+EC serves Puts from all replicas. For AA+SC, lock contention at the DLM caps the performance for both read- and write-intensive workloads. As a result, MS+SC performs 3.2× better than AA+SC for read-intensive workload and ˜2× better for the write-intensive workload.

9.8 DPDK Optimization

We recently added support for DPDK based communication between clients, controlets, and datalets in to BESPOKV. In this experiment, we show performance of socket versus DPDK based communication. We deployed a single shard on our local testbed (Section 9.1) and measured latency and throughput using YCSB. Each node in our local setup is equipped with Intel ethernet controller X540-AT2. We used Intel's DPDK framework version 17.05. Fig. 14 shows that DPDK reduces latency by up to 65 percent. We also observe 3× improvement in throughput compared to socket based communication. Another interesting finding is that DPDK based communication results in more stable performance.

10 Failover & Data Recovery

We also evaluate how BESPOKV performs in case of a node failure, and compare it with Redis's replication used by Dynomite for failover recovery. In this set of tests, we use 3 shards (each with 3 replicas) to clearly reflect the impact of a failure on throughput. The workload consists of 1 million

5. Twemproxy itself does not provide any consistency support.

6. Uniform workloads show similar trend, hence are omitted.

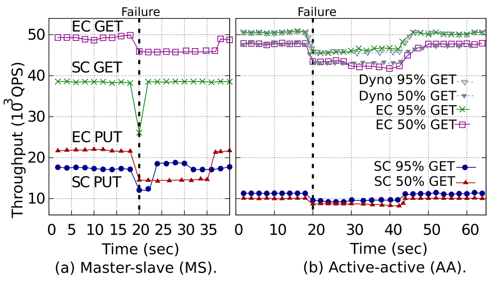


Fig. 15. Throughput timeline on failover. `EC`: eventual consistency; `SC`: strong consistency; `Dyno`: Dynomite.

KV tuples generated with a Zipfian distribution. We intentionally crash a node to emulate a failure, and Fig. 15 shows the resulting throughput change.

MS topology. For MS+SC, we bring down the head node under the write-intensive workload (50 percent `Put`, as shown in the bottom half of Fig. 15a) and the tail node for the read-intensive workload (95 percent `Get`, as shown in top half of the figure), to maximize the performance disruption on the respective workloads. For MS+EC, we take down the master node for the write-intensive workload and a random slave node for the read-intensive workload.

We observe that for MS+SC, `Put` throughput goes down by about $1/3$ when the head node crashes at 20 sec, as we have 3 shards. The coordinator detects the node failure from the lack of heartbeat message before assigning the master role to the second node in the chain. The coordinator then launches a new controlet–datalet pair in recovery mode, and inserts the pair to the end of the chain once data recovery completes at around 35 sec. Meanwhile the throughput stabilizes. MS+EC failover shows a similar trend. The top half of Fig. 15a shows the impact of node failure on `Get` performance under MS topology. For MS+SC, killing the tail brings down `Get` throughput by $1/3$. Once failure is detected, the coordinator makes the 2nd-from-last node in the chain the new tail, and updates the topology metadata. Once clients see the update, they reroute the corresponding `Gets` to the new tail. Hence, the throughput goes back to normal in˜5 sec. MS+EC behaves differently as `Gets` are served by any of the 3 replicas. Thus, the slave failure drops throughput by only˜$1/9$.

AA Topology. In BESPOKV's AA and Dynomite (with Redis) failover test, we randomly kill a node at 20 sec and record the overall throughput. As shown in Fig. 15b, the throughput is slightly impacted in all cases, because both BESPOKV AA and Dynomite serve reads and writes from all replicas. Dynomite leverages Redis' master-slave replication to recover data directly from the surviving nodes. We observe trend similar to Dynmoite as BESPOKV also uses datalet's callback functions to import and export the data. Please note that users can choose to add more replicas to increase the overall performance so that in case of a single node failure or a topology/consistency switch the performance drop is not significant enough to affect the HPC application.

11 Related Work

Dynomite [19] adds fault tolerance and consistency support for simple data stores such as Redis. Dynomite only supports eventual consistency with AA topology. It also requires the single-server applications to support distributed management functions such as Redis' streaming data recovery/migration mechanism. BESPOKV's datalet is completely oblivious of the upper-level distributed management, which offers improved flexibility and programmability.

Pileus [70] is a cloud storage system that offers a range of consistency-level SLAs. Some storage systems offer tunable consistency, e.g., ManhattanDB [71]. Flex-KV [72] is another flexible key-value store that can be configured to act as a non-persistent/durable store and operates consistently/inconsistently. Morphus [73] provides support towards reconfigurations for NoSQL stores in an online manner. MOS [74], [75] and hatS [76], [77] provide flexible and elastic resource-level partitioning for serving heterogeneous object store workloads. ClusterOn [78] proposes to offer generic distributed systems management for a range of distributed storage systems. MBal [36], [37] provides fine-grained service-level differentiation via flexible data partitioning. To the best of our knowledge, BESPOKV is the first generic framework that offers a broad range of consistency/topology options for both users and KV store application developers.

Vsync [22] is a library for building replicated cloud services. BESPOKV embeds single-node KV store application code and automatically scales it with a rich choice of services. Going one step further, BESPOKV can be an ideal platform to leverage Vsync to further enrich flexibility. EventWave [79] elastically scales inelastic cloud programs. PADS [80] provides policy architecture to build distributed applications. Similarly, mOS [81] provides reusable networking stack to allows developers to focus on the core application logic instead of dealing with low-level packet processing. BESPOKV focuses on a specific domain with a well-defined limited set of events–KV store applications.

12 Conclusion

We have presented the design and implementation of BESPOKV, a framework, which takes a single-server data store and transparently enables a scalable, fault-tolerant distributed KV store service. BESPOKV's decoupled control and data plane architecture, configurability, and extensibility enable new solutions for emerging HPC systems and workloads. BESPOKV can be easily extended to offer advanced features such as range query, per-request consistency, polyglot persistence, and more. To the best of our knowledge, BESPOKV is first to support a seamless on-the-fly topology/consistency adaptation. As examples, we present a novel mechanism to make transitions from MS+EC to MS+SC, and from AA+EC to MS+EC. We also present several use cases to show effectiveness of BESPOKV to support HPC applications. Evaluation shows that BESPOKV is flexible, adaptive to new user requirements, achieves high performance, and scales horizontally. BESPOKV has been open-sourced and is available at https://github.com/tddg/bespokv

Acknowledgments

This work was sponsored in part by the US National Science Foundation under the Grants: CNS-1565314, CNS-1405697, CNS-1615411, CCF-1919075, CCF-1919113, and CNS-1814430. This work was performed while Fred Douglis was employed by Dell EMC and gratefully acknowledges Dell EMC's support.

References

[1] J. Kim, S. Lee, and J. S. Vetter, "PapyrusKV: A high-performance parallel key-value store for distributed NVM architectures," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2017, Art. no. 57.

[2] K. Wang, A. Kulkarni, M. Lang, D. Arnold, and I. Raicu, "Using simulation to explore distributed key-value stores for extreme-scale system services," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2013, Art. no. 9.

[3] Z. W. Parchman, F. Aderholdt, and M. G. Venkata, "SharP hash: A high-performing distributed hash for extreme-scale systems," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2017, pp. 647–648.

[4] S. Eilemann *et al.*, "Key/value-enabled flash memory for complex scientific workflows with on-line analysis and visualization," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2016, pp. 608–617.

[5] MongoDB, Accessed: Aug. 2019. [Online]. Available: https://www.mongodb.com/

[6] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," in *Proc. 21st ACM SIGOPS Symp. Operating Syst. Princ.*, 2007, pp. 205–220.

[7] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Syst. Rev.*, vol. 44, pp. 35–40, 2010.

[8] R. Nishtala *et al.*, "Scaling Memcache at Facebook," in *Proc. 10th USENIX Conf. Netw. Syst. Des. Implementation*, 2013, pp. 385–398.

[9] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports, "Building consistent transactions with inconsistent replication," in *Proc. 25th Symp. Operating Syst. Princ.*, 2015, pp. 263–278.

[10] R. Escriva, B. Wong, and E. G. Sirer, "HyperDex: A distributed, searchable key-value store," in *Proc. ACM SIGCOMM Conf. Appl. Technol., Archit. Protocols Comput. Commun.*, 2012, pp. 25–36.

[11] Social Artisan, Accessed: Aug. 2019. [Online]. Available: http://socialartisan.co.uk/

[12] Behance, Accessed: Aug. 2019. [Online]. Available: https://www.behance.net/

[13] The Migration Process, Accessed: Aug. 2019. [Online]. Available: https://academy.datastax.com/planet-cassandra//mongodb-to-cassandra-migration/#data_model

[14] Why flowdock migrated from cassandra to mongodb, Accessed: Aug. 2019. [Online]. Available: http://blog.flowdock.com/2010/07/26/flowdock-migrated-from-cassandra-to-mongodb/

[15] R. Nishtala *et al.*, "Scaling Memcache at Facebook," in *Proc. 10th USENIX Conf. Netw. Syst. Des. Implementation*, 2013, pp. 385–398.

[16] Twitter's Twemproxy, Accessed: Aug. 2019. [Online]. Available: https://github.com/twitter/twemproxy

[17] Memcached, Accessed: Aug. 2019. [Online]. Available: https://memcached.org/

[18] Redis, Accessed: Aug. 2019. [Online]. Available: http://redis.io/

[19] Netflix's Dynomite, Accessed: Aug. 2019. [Online]. Available: https://github.com/Netflix/dynomite

[20] LevelDB, Accessed: Aug. 2019. [Online]. Available: https://github.com/google/leveldb

[21] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 183–196.

[22] Vsync, Accessed: Aug. 2019. [Online]. Available: https://vsync.codeplex.com/

[23] Google Cloud Platform, Accessed: Aug. 2019. [Online]. Available: https://cloud.google.com/compute/

[24] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware scheduling for heterogeneous datacenters," in *Proc. 18th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2013, pp. 77–88.

[25] J. Mars, L. Tang, and R. Hundt, "Heterogeneity in "Homogeneous" warehouse-scale computers: A performance opportunity," *IEEE Comput. Archit. Lett.*, vol. 10, no. 2, pp. 29–32, Jul.–Dec. 2011.

[26] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2010, Art. no. 11.

[27] Distributed locks with Redis, Accessed: Aug. 2019. [Online]. Available: http://redis.io/topics/distlock

[28] ZLog, Accessed: Aug. 2019. [Online]. Available: https://github.com/noahdesu/zlog

[29] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, and J. D. Davis, "CORFU: A shared log design for flash clusters," in *Proc. 9th USENIX Symp. Netw. Syst. Des. Implementation*, 2012, pp. 1–14.

[30] M. A. Sevilla *et al.*, "Malacology: A programmable storage system," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 175–190.

[31] SSDB, Accessed: Aug. 2019. [Online]. Available: https://github.com/ideawu/ssdb

[32] Google Protocol Buffers, Accessed: Aug. 2019. [Online]. Available: https://developers.google.com/protocol-buffers/

[33] Y.-J. Hong and M. Thottethodi, "Understanding and mitigating the impact of load imbalance in the memory caching tier," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, Art. no. 13.

[34] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky, "Small cache, big effect: Provable load balancing for randomly partitioned cluster services," in *Proc. 2nd ACM Symp. Cloud Comput.*, 2011, pp. 1–12.

[35] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, "Be fast, cheap and in control with SwitchKV," in *Proc. 13th Usenix Conf. Netw. Syst. Des. Implementation*, 2016, pp. 31–44.

[36] Y. Cheng, A. Gupta, and A. R. Butt, "An in-memory object caching framework with adaptive load balancing," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, pp. 1–16.

[37] Y. Cheng, A. Gupta, A. Povzner, and A. R. Butt, "High performance in-memory caching through flexible fine-grained services," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, Art. no. 56.

[38] X. Jin *et al.*, "NetCache: Balancing key-value stores with fast in-network caching," in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 121–136.

[39] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *Proc. 6th Conf. Symp. Operating Syst. Des. Implementation*, 2004, Art. no. 7.

[40] S. Almeida, J. A. Leitão, and L. Rodrigues, "ChainReaction: A causal+ consistent datastore based on chain replication," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 85–98.

[41] J. Terrace and M. J. Freedman, "Object storage on CRAQ: High-throughput chain replication for read-mostly workloads," in *Proc. Conf. USENIX Annu. Tech. Conf.*, 2009, Art. no. 11.

[42] H.-T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, 1981.

[43] Cassandra Configuration, Accessed: Aug. 2019. [Online]. Available: http://docs.datastax.com/en/cassandra/2.0/cassandra/dml/dml_config_consistency_c.html

[44] How Dynomite handles the data conflict, Accessed: Aug. 2019. [Online]. Available: https://github.com/Netflix/dynomite/issues/274

[45] R. Van Renesse, D. Dumitriu, V. Gough, and C. Thomas, "Efficient reconciliation and flow control for anti-entropy protocols," in *Proc. ACM Workshop Large-Scale Distrib. Syst. Middleware*, 2008, Art. no. 6.

[46] Polyglot persistence, Accessed: Aug. 2019. [Online]. Available: https://en.wikipedia.org/wiki/Polyglot_persistence

[47] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 31, pp. 149–160, 2001.

[48] B. Xie *et al.*, "Characterizing output bottlenecks in a supercomputer," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, Art. no. 8.

[49] A. K. Paul *et al.*, "I/O load balancing for big data HPC applications," in *Proc. IEEE Int. Conf. Big Data*, 2017, pp. 233–242.

[50] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu, "An ephemeral burst-buffer file system for scientific applications," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, Art. no. 69.

[51] T. Wang, S. Oral, Y. Wang, B. Settlemyer, S. Atchley, and W. Yu, "BurstMem: A high-performance burst buffer system for scientific applications," in *Proc. IEEE Int. Conf. Big Data*, 2014, pp. 71–79.

[52] D. Shankar, X. Lu, and D. K. D. Panda, "Boldio: A hybrid and resilient burst-buffer over lustre for accelerating big data I/O," in *Proc. IEEE Int. Conf. Big Data*, 2016, pp. 404–409.

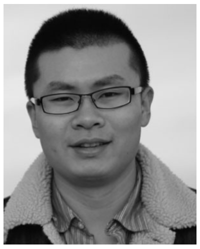
[53] X. Shi, M. Li, W. Liu, H. Jin, C. Yu, and Y. Chen, "SSDUP: A traffic-aware SSD burst buffer for HPC systems," in *Proc. Int. Conf. Supercomput.*, 2017, Art. no. 27.

[54] R. Rajachandrasekar, A. Moody, K. Mohror, and D. K. Panda, "A 1 PB/s file system to checkpoint three million MPI tasks," in *Proc. 22nd Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2013, pp. 143–154.

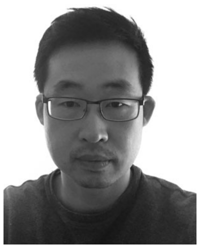
[55] S. Patil and G. Gibson, "Scale and concurrency of GIGA+: File system directories with millions of files," in *Proc. 9th USENIX Conf. File Stroage Technol.*, 2011, Art. no. 13.

[56] K. Ren, Q. Zheng, S. Patil, and G. Gibson, “IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 237–248.
[57] Q. Zheng, K. Ren, G. Gibson, B. W. Settlemyer, and G. Grider, “DeltaFS: Exascale file systems scale better without dedicated servers,” in *Proc. 10th Parallel Data Storage Workshop*, 2015, pp. 1–6.
[58] R. H. Castain, D. G. Solt, J. Hursey, and A. Bouteiller, “PMIx: Process management for exascale environments,” in *Proc. 24th Eur. MPI Users’ Group Meet.*, 2017, pp. 14:1–14:10.
[59] libmc, Accessed: Aug. 2019. [Online]. Available: https://github.com/douban/libmc
[60] ZooKeeper Recipes and Solutions, Accessed: Aug. 2019. [Online]. Available: https://zookeeper.apache.org/doc/r3.1.2/recipes.html
[61] Embedded Masstree, Accessed: Aug. 2019. [Online]. Available: https://github.com/rmind/masstree
[62] Kubernetes: Production-Grade Container Orchestration, Accessed: Aug. 2019. [Online]. Available: https://kubernetes.io/
[63] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.
[64] SeaweedFS, Accessed: Aug. 2019. [Online]. Available: https://github.com/chrislusf/seaweedfs
[65] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar, “Flash storage disaggregation,” in *Proc. 11th Eur. Conf. Comput. Syst.*, 2016, Art. no. 29.
[66] N. Zhao *et al.*, “Chameleon: An adaptive wear balancer for flash clusters,” in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2018, pp. 1163–1172.
[67] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, “Serving large-scale batch computed data with project Voldemort,” in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, Art. no. 18.
[68] N. Carvalho *et al.*, “Finding consistency in an inconsistent world: Towards deep semantic understanding of scale-out distributed databases,” in *Proc. 8th USENIX Conf. Hot Topics Storage File Syst.*, 2016, pp. 66–70.
[69] Why not Cassandra, Accessed: Aug. 2019. [Online]. Available: http://www.dynomitedb.com/docs/dynomite/v0.5.6/faq/
[70] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, “Consistency-based service level agreements for cloud storage,” in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 309–324.
[71] Manhattan, our real-time, multi-tenant distributed database for Twitter scale, Accessed: Aug. 2019. [Online]. Available: http://goo.gl/7ETHfo
[72] A. Phanishayee, D. G. Andersen, H. Pucha, A. Povzner, and W. Belluomini, “Flex-KV: Enabling high-performance and flexible KV systems,” in *Proc. Workshop Manage. Big Data Syst.*, 2012, pp. 19–24.
[73] M. Ghosh, W. Wang, G. Holla, and I. Gupta, “Morphus: Supporting online reconfigurations in sharded NoSQL systems,” *IEEE Trans. Emerg. Topics Comput.*, vol. 5, no. 4, pp. 466–479, Fourth Quarter 2017.
[74] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt, “Taming the cloud object storage with MOS,” in *Proc. 10th Parallel Data Storage Workshop*, 2015, pp. 7–12.
[75] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt, “MOS: Workload-aware elasticity for cloud object stores,” in *Proc. 25th ACM Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2016, pp. 177–188.
[76] K. Krish, A. Anwar, and A. R. Butt, “hatS: A heterogeneity-aware tiered storage for hadoop,” in *Proc. 14th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2014, pp. 502–511.
[77] K. Krish, A. Anwar, and A. R. Butt, “[phi] Sched: A heterogeneity-aware hadoop workflow scheduler,” in *Proc. IEEE 22nd Int. Symp. Modelling Anal. Simul. Comput. Telecommun. Syst.*, 2014, pp. 255–264.
[78] A. Anwar, Y. Cheng, H. Huang, and A. R. Butt, “ClusterOn: Building highly configurable and reusable clustered data services using simple data nodes,” in *Proc. 8th USENIX Conf. Hot Topics Storage File Syst.*, 2016, pp. 51–55.
[79] W.-C. Chuang, B. Sang, S. Yoo, R. Gu, M. Kulkarni, and C. Killian, “EventWave: Programming model and runtime support for tightly-coupled elastic cloud applications,” in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, pp. 1–16.
[80] N. M. Belaramani, J. Zheng, A. Nayate, R. Soulé, M. Dahlin, and R. Grimm, “PADS: A policy architecture for distributed storage systems,” in *Proc. 6th USENIX Symp. Netw. Syst. Des. Implementation*, 2009, pp. 59–73.
[81] M. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park, “mOS: A reusable networking stack for flow monitoring middleboxes,” in *Proc. 14th USENIX Conf. Netw. Syst. Des. Implementation*, 2017, pp. 113–129.

Ali Anwar received the PhD degree in computer science from Virginia Tech, Blacksburg, Virginia. He is currently a research staff member with IBM Almaden Research Center. In his earlier years, he worked as a tools developer (GNU GDB) at Mentor Graphics. His research interests include distributed computing systems, cloud storage management, file and storage systems, AI platforms, and intersection of systems and machine learning.



Yue Cheng received the PhD degree in computer science from Virginia Tech, Blacksburg, Virginia, in 2017. He is currently an assistant professor of computer science with George Mason University. His research interests include distributed systems, cloud computing, and high-performance computing.



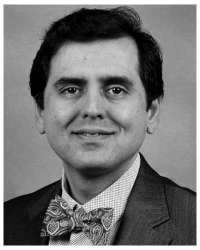
Hai Huang received the BSE degree in computer science and engineering (CSE) from the Ohio State University, Columbus, Ohio, in 2000, and the MS and PhD degrees in computer science and engineering from the University of Michigan, Ann Arbor, Michigan, in 2006. He is a currently research staff with the Cloud Computing Department, IBM Research. His research interests include cloud computing, operating systems, distributed systems management, software testing, and anomaly detection.

Jingoo Han is currently working toward the PhD degree with the Department of Computer Science, Virginia Tech, Blacksburg, Virginia. In his earlier years, he has worked as a senior software engineer with Samsung Electronics. His research interests include distributed systems, deep learning, and high-performance computing.

Hyogi Sim received the BS degree in civil engineering and the MS degree in computer engineering from Hanyang University, Seoul, South Korea, and the MS degree in computer science from Virginia Tech, Blacksburg, Virginia, in 2014 and is currently working toward the PhD degree at Virginia Tech, Blacksburg, Virginia. He joined Oak Ridge National Laboratory in 2015, as a post-masters associate. During this appointment, he conducted research and development on active storage systems and scientific data management for HPC systems. He is currently an HPC systems engineer with Oak Ridge National Laboratory. His primary role is to design and develop a checkpoint-restart storage system for the exascale computing project. His areas of interest include storage systems and distributed systems.

Dongyoon Lee received the MS and PhD degrees in computer science and engineering from the University of Michigan, Ann Arbor, Michigan. He is currently an assistant professor of computer science with Stony Brook University, where he works on computer systems, software reliability, program analysis, concurrency, security, computer architecture, and software engineering. From 2014 to 2019, he was an assistant professor of computer science with Virginia Tech. He has won a Distinguished Paper Award at FSE 2018 and a Best Paper Award at ASPLOS 2011. His SC 2016 paper was nominated as a Best Student Paper Finalist. He has received a Virginia Tech ICTAS Junior Faculty Award in 2017, a Google Research Award in 2015, a ProQuest Distinguished Dissertation Award in 2013, and a VMWare Graduate Fellowship in 2011. His co-authored papers won the best student paper finalist at SC 2016, and the best paper at ASPLOS 2011.

Fred Douglis (Fellow, IEEE) received the PhD degree in computer science from the U.C. Berkeley, Berkeley, California. He is a chief research scientist with Perspecta Labs since January 2018, where he works on applied research in the areas of blockchain, network optimization, and security. He was previously with companies including Matsushita, AT&T, IBM, and (Dell) EMC. His research interests included storage, distributed systems, web tools and performance, and mobile computing. He is a member of the IEEE Computer Society Board of Governors.



Ali R. Butt is currently a professor of computer science with Virginia Tech. He is a recipient of an NSF CAREER, IBM Faculty Awards, NetApp Faculty Fellowships, and a VT COE Faculty Fellowship. He has served on the editorial board of the *IEEE Transactions on Cloud Computing*, *ACM Transactions on Storage*, and *IEEE Transactions on Parallel and Distributed Systems*. He is an alumni of the NAE FOE and NAS AA symposia (2010 organizer). His research interests include scalable distributed computing systems, cloud computing, edge computing, file and storage systems, and Internet-of-Things. At Virginia Tech, he leads the Distributed Systems & Storage Laboratory (DSSL).

Energy-Efficient Parallel Real-Time Scheduling on Clustered Multi-Core

Ashikahmed Bhuiyan, Di Liu, Aamir Khan, Abusayeed Saifullah, Nan Guan, and Zhishan Guo

Abstract—Energy-efficiency is a critical requirement for computation-intensive real-time applications on multi-core embedded systems. Multi-core processors enable intra-task parallelism, and in this work, we study energy-efficient real-time scheduling of constrained deadline sporadic parallel tasks, where each task is represented as a directed acyclic graph (DAG). We consider a clustered multi-core platform where processors within the same cluster run at the same speed at any given time. A new concept named *speed-profile* is proposed to model per-task and per-cluster energy-consumption variations during run-time to minimize the expected long-term energy consumption. To our knowledge, no existing work considers energy-aware real-time scheduling of DAG tasks with *constrained deadlines*, nor on a *clustered multi-core platform*. The proposed energy-aware real-time scheduler is implemented upon an *ODROID XU-3* board to evaluate and demonstrate its feasibility and practicality. To complement our system experiments in large-scale, we have also conducted simulations that demonstrate a CPU energy saving of up to 67 percent through our proposed approach compared to existing methods.

Index Terms—Parallel task, real-time scheduling, energy minimization, cluster-based platform, heterogeneous platform

1 Introduction

Multi-core processors appear as an enabling platform for embedded systems applications that require real-time guarantees, energy efficiency, and high performance. Intra-task parallelism (a task can be executed on multiple cores simultaneously) enables us to exploit the capability of the multi-core platform, and facilitates a balanced distribution of the tasks among the processors. Such a balanced distribution leads to energy efficiency [1]. Directed Acyclic Graph (DAG) task model [2] is one of the most generalized workload model for representing deterministic intra-task parallelism. Recently, quite some effort has been spent on developing real-time scheduling strategies and schedulability analysis of DAG tasks, few to mention [2], [3], [4], [5], [6], [7], [8].

There are several real-world application that uses the DAG model. For example, the work in [3] studies problems related to scheduling parallel real-time tasks, modeled as DAG, on multiprocessor architectures. In a homogeneous computing environment, a low-complexity compile-time algorithm for scheduling DAG tasks is proposed in [9]. Another example would be systems that control asynchronous devices, such as the local-area network adapters that implement real-time communication protocols.

Since many of those applications are battery-powered, considering energy-efficient approaches for designing such a platform is crucial. Thanks to the fact that modern generation processors support dynamic voltage and frequency scaling (DVFS), where each processor can adjust the voltage and frequency at runtime to minimize power consumption, per-core energy minimization becomes possible during runtime. Despite the hardness of the problem [10], a significant amount of work has considered power minimization for non-parallel tasks on a multi-core platform (refer to [11] for a survey). Regarding parallel tasks, Guo *et al.* studied energy-efficient real-time scheduling for DAG tasks as an early research effort [12]. They adopted the federated scheduling and task decomposition framework [2] for minimizing system energy consumption via per-core speed modulation. As the only step (that we are aware of) towards energy-aware scheduling of real-time DAG tasks, they targeted an exciting problem and laid some of the foundations of this work. However, the attention of [12] is restricted to implicit deadline tasks with a system model of per-core DVFS.

Unfortunately, per-core DVFS becomes inefficient as it increases the hardware cost [13]. For balancing the energy efficiency and the hardware cost, there is an ongoing trend to group processors into islands, where processors in the same island execute at the same speed. For example, a big. LITTLE platform (e.g., ODROID XU-3 [14]) consists of high performance (but power-hungry) cores integrated into 'big' clusters and low-power cores into 'LITTLE' clusters. Such a platform executes several real-life applications with heavy computational demands (e.g., video streaming [15]) in an energy-efficient manner. Apart from the energy consumption issue, a multi-core platform enables task execution with

- *A. Bhuiyan and Z. Guo are with the Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL 32816. E-mail: abvn2@mst.edu, zsguo@ucf.edu.*
- *D. Liu is with Yunnan University, Kunming 650106, China. E-mail: d.liu@liacs.leidenuniv.nl.*
- *A. Khan is with Brainco Inc., Somerville, MA 02143. E-mail: aamir.khan@brainco.tech.*
- *A. Saifullah is with the Department of Computer Science, Wayne State University, Detroit, MI 48202. E-mail: saifullah@wayne.edu.*
- *N. Guan is with the Department of Computing, Hong Kong Polytechnic University, Hong Kong. E-mail: csguannan@comp.polyu.edu.hk.*

Manuscript received 24 July 2019; revised 6 Mar. 2020; accepted 1 Apr. 2020. Date of publication 14 Apr. 2020; date of current version 24 Apr. 2020. (Corresponding author: Ashikahmed Bhuiyan.) Recommended for acceptance by Q. Zheng. Digital Object Identifier no. 10.1109/TPDS.2020.2985701

high-performance demand and tight deadlines, essential for computation-intensive real-time systems, e.g., autonomous vehicles [16].

Despite the urgent need, to our knowledge, no work has been done that considered the energy-efficient scheduling of DAG tasks in clustered multi-core platforms, where cores form a group of frequency/voltage clusters. Such kind of system balances the energy efficiency and hardware cost compared to the traditional (with individual frequency scaling feature) multi-core models. The scheduling problem becomes highly challenging on such platforms because:

(i) The relationship between the execution time, frequency, and the energy consumption is nonlinear, making it highly challenging to minimize energy consumption while guaranteeing real-time correctness, i.e., none of the tasks miss their deadline.

(ii) Existing solution (e.g., [12]) relies on the assumption that each processor can freely adjust its speed. That solution performs poorly as the assumption is no longer valid under a more realistic platform model considered in this paper.

(iii) The speed of a cluster becomes unpredictable when shared by multiple tasks with sporadic release patterns.

Contribution. In this paper, we propose a novel technique for energy-efficient scheduling of constrained deadline DAG tasks in a cluster-based multi-core system. To the best of our knowledge, no work has investigated the energy-efficient scheduling of DAG tasks on such a cluster-based platform. It is also the first work that has addressed the power awareness issue considering constrained deadline DAG tasks. Specifically, we make the following contributions:

- We consider a more practical *cluster-based system model* where the cores must execute at the same speed at any time instant within each cluster.
- To better handle constrained deadlines, one need to capture the gaps between deadlines and upcoming releases, as well as handling sporadic releases. Considering a continuous frequency scheme, we first propose a novel concept of *speed-profile* to present the energy-consumption behavior for each task as well as each cluster, such that they could guide task partitioning in an energy-efficient manner. An efficient *greedy* algorithm is proposed to partition DAG tasks according to the constructed speed-profiles.
- We propose an approach to creating the speed-profile to adapt to the discrete frequency scheme. Also, we extend our approach to apply to the heterogeneous platform.
- To evaluate the effectiveness of our proposed technique, we implement it on the ODROID XU-3 board, a representative multi-core platform for embedded systems [14]. The experiments report that our approach can save energy consumption by 18 percent compared to a reference approach. For larger-scale evaluation, we perform simulations using synthetic workloads and compare our technique with two existing baselines [12], [17]. The simulation results demonstrate that our method can reduce energy consumption by up to 66 percent compared to the existing ones under the cluster-based platform setting.

Organization. The rest of the paper is organized as follows. Section 2 presents the workload, power, and platform models, and the problem statement. Section 3, describes the importance of creating a speed-profile for an individual task and the whole cluster. Section 4 discusses the approaches to create the speed-profile (considering both the continuous and discrete frequency mode) for each task. In this section, we also propose a greedy algorithm to allocate multiple tasks in the same cluster. Sections 5 and 6 presents the experimental and simulation results. Section 7 discusses related work including a detailed comparison with our existing work [12], [18]. Section 8 concludes this paper.

2 System Model, Problem Statement, and Background

2.1 System Model and Problem Statement

Workload Model. We consider a set of sporadic parallel task denoted by $\tau = \{\tau_1, \ldots, \tau_n\}$, where each $\tau_i \in \tau$ $(1 \leq i \leq n)$ is represented as a DAG with a minimum inter-arrival separation (i.e., period) of T_i time units, and a relative deadline of $D_i (\leq T_i)$ time units. An implicit deadline task has the same relative deadline and period, i.e., $D_i = T_i$. As a DAG task, the execution part of task τ_i contains a total of N_i nodes, each denoted by $\mathcal{N}_i^j (1 \leq j \leq N_i)$. A directed edge from $\mathcal{N}_i^j$ to $\mathcal{N}_i^k (\mathcal{N}_i^j \rightarrow \mathcal{N}_i^k)$ implies that execution of $\mathcal{N}_i^k$ can start if $\mathcal{N}_i^j$ finishes for every instance (precedence constraints). In this case, $\mathcal{N}_i^j$ is called a parent of $\mathcal{N}_i^k$ ($\mathcal{N}_i^k$ is a child of $\mathcal{N}_i^j$). A node may have multiple parents or children. The *degree of parallelism*, $\mathcal{M}_i$, of τ_i is the number of nodes that can be simultaneously executed. c_i^j denotes the execution requirement of node $\mathcal{N}_i^j$. $C_i := \sum_{j=1}^{N_i} c_i^j$ denotes the worst case execution requirement (WCET) of τ_i.

A *critical path* is a directed path with the maximum total execution requirements among all other paths in a DAG. L_i is the sum of the execution requirements of all the nodes that lie on a critical path. It is the minimum make-span of τ_i, i.e., in order to make τ_i schedulable, at least L_i time units are required even when number of cores is unlimited. Since at least L_i time units are required for τ_i, the condition $T_i \geq L_i$ (implicit deadline tasks) and $D_i \geq L_i$ (constrained deadline tasks) must hold for τ_i to be schedulable. A schedule is said to be feasible if upon satisfying the precedence constraints, all the sub-tasks (nodes) receive enough execution from their arrival times, i.e., C_i within T_i (implicit deadline) or D_i (constrained deadline) time units. These terms are illustrated in Fig. 2a.

Platform Model. We consider a clustered multi-core platform, where processors within the same cluster run at the same speed (frequency and supply voltage) at any given time. Such additional restriction comparing to traditional multi-core platform makes the model more realistic in many senarios. For example, our experiment is conducted on the ODROID XU-3 platform with one 'LITTLE' cluster of four energy-efficient ARM Cortex-A7 and one 'big' cluster of four performance-efficient ARM Cortex-A15. Note that we do not restrict the hardware-dependent energy parameters (e.g., α, β and γ in the power model discussed below) to be identical across different clusters—these parameters can be derived using any curve-fitting method, e.g., [19].

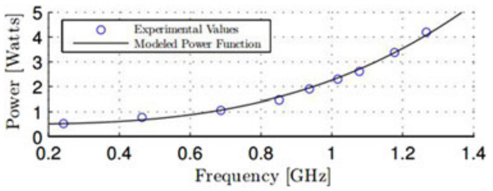


Fig. 1. Comparison of the power model (Equation (1)) with experimental results in [24]. Here, $\alpha = 1.76 Watts/GHz^3$, $\gamma = 3$, and $\beta = 0.5$ Watts. This figure is adopted from [20].

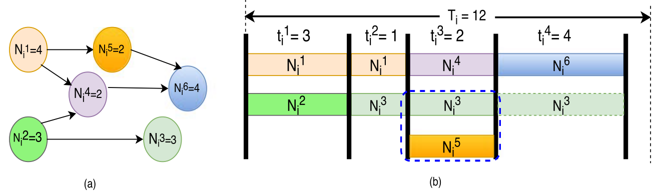


Fig. 2. (a) A DAG, τ_i (b) transformed DAG τ_i after applying task decomposition. Both of them are adopted from [12].

Energy Model. Assuming frequency (speed) of a processor at a specific instant t is $s(t)$ (in short, denoted as s), then its power consumption $P(s)$ can be calculated as

$$P(s) = P_s + P_d(s) = \beta + \alpha s^\gamma. \tag{1}$$

Here, P_s and $P_d(s)$ respectively denote the static and dynamic power consumption. Whenever a processor remains on, it introduces P_s in the system (due to leakage current). Switching activities introduce $P_d(s)$ which is frequency dependent and represented as αs^γ. Here, the $\alpha > 0$ depends on the effective switching capacitance [20]; $\gamma \in [2, 3]$ is a fixed parameter determined by the hardware; $\beta > 0$ represents the static part of power consumption. From this model, the energy consumption over any given period $[b, f]$ is calculated as $E(b,f) = \int_b^f P(s(t))\, dt$.

Our motivation behind selecting this power model comes from the fact that it complies with many existing works in the community, few to mention [10], [12], [18], [20], [21], [22], [23]. Beside this, recently this model was shown to be highly realistic by showing its similarity with actual power consumption [21]. Fig. 1 shows comparison between the original power consumption results from [24] and the power model in Equation (1).

Assumptions. In this paper, we make the following assumptions: (i) we focus on CPU power consumption, and (ii) Dynamic power management (DPM) is not considered. Appendix B, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2020.2985701, provides the details behind these assumptions, their impacts, and some hints to overcome the drawbacks.

Problem Statement. Considering a constrained deadline sporadic DAG task-set on a clustered multi-core platform, we focus on finding a correct scheduling strategy, while the CPU power consumption is minimized.

2.2 Background and Existing Concepts

In this section, we describe some existing concepts and techniques for handling real-time parallel task scheduling, and that constitute an initial step for our proposed work.

Task Decomposition. The well-known task decomposition technique [2] transforms a parallel task τ_i into a set of sequential tasks as demonstrated in Fig. 2b. Upon task decomposition, each node $\mathcal{N}_i^l \in \tau_i$ is converted into an individual sub-task with its scheduling window (defined by its own release time and deadline) and execution requirement (c_i^l). The allocation of release time and deadline respect all the dependencies (represented by edges in the DAG). Considering that a task is allowed to execute on an unlimited number of cores, starting from the beginning, a vertical line is drawn at every time instant where a node $\mathcal{N}_i^l$ starts or ends. So the DAG is partitioned into several segments which may contain single/multiple thread(s). Threads assigned to the same segment share equal amount of execution length; e.g., $\mathcal{N}_i^3$, $\mathcal{N}_i^4$, and $\mathcal{N}_i^5$ all have 2-time units assigned to the 3rd segment, as demonstrated in Fig. 2b.

Segment Extension. The deadline for each node via task decomposition may be unnecessarily restrictive, e.g., the decomposition of the DAG in Fig. 2a will restrict $\mathcal{N}_i^3$ within the 2nd and 3rd segment. To eliminate such unnecessary restriction and allow $\mathcal{N}_i^3$ to execute in the 4th segment, segment extension should be applied, e.g., the green rectangle for $\mathcal{N}_i^3$ in the 4th segment in Fig. 2b.

Intra-Task Processor Merging. After applying task decomposition and segment extension upon a DAG task τ_i, some of these cores (where τ_i is allocated) can be very lightly loaded. Those core cause massive leakage power consumption in the long run and should be avoided when necessary. Intra-task merging [12] seeks to merge those cores to gain overall energy efficiency by reducing the total number of active cores. For example, in Fig. 2b, the third core (executing $\mathcal{N}_i^5$) is lightly loaded, and thus it is better to merge all the execution into the second core and shut it off completely. Such a reduction on the number of active cores minimizes leakage power consumption (see Equation (1) and Fig. 2 in [12]) as well as the total number of clusters.

3 Speed-Profile for Task and Cluster

This section discusses how different tasks share a cluster where all processors in a cluster execute at the same speed. When multiple tasks share a cluster, they may not align well due to sporadic releases and different periods. In a cluster-based platform, the processor having the maximum speed dominates the others in the same cluster. Hence, existing energy-saving techniques may perform poorly in a cluster-based platform. To tackle this problem, we propose a new concept called *speed-profile*. We provide the definition of speed-profile and its motivation in Section 3.1. Section 3.2 describes how speed-profiles are handled when two tasks are partitioned into the same cluster.

3.1 Speed-Profile for Each DAG

Interesting energy-saving techniques (e.g., segment extension) have been proposed in [12] for the implicit deadline tasks. For the constrained deadline tasks, this technique becomes incompetent because of the non-negligible idle gaps between the task deadline and its next release. For example, consider the task τ_i in Fig. 2b with $D_i = 10$ and $T_i = 12$. Segment extension can stretch $\mathcal{N}_i^3$ to the end of the 4th segment but cannot utilize the idle time of 2 units. Besides, the suboptimal solution provided in [12] becomes *non-convex* (in a convex function, we can find the global maximum or minimum, for some variables of this function, which does not hold for a non-convex function) in a cluster-based platform (see Lemma 1).

Lemma 1. *In a cluster-based platform, the convex optimization problem constructed in [12] becomes non-convex.*

Proof. The following set of constraints ensure the real-time correctness for each node $\mathcal{N}_i^l \in \tau_i$, i.e., $\mathcal{N}_i^l$ receives enough time to finish execution within its scheduling window

$$\forall l : \mathcal{N}_i^l \in \tau_i :: \sum_{j=b_i^l}^{d_i^l} t_j^c s_{i,j} \geq c_i^{\mathcal{N}_i^l}. \tag{2}$$

We introduce the following inequalities to bound the total length for all segments in task τ_i

$$\sum_{j=1}^{Z} t_j^c \leq T_i. \tag{3}$$

Any value of execution speed and segment length ensures real-time correctness if Equations (2) and (3) are respected. However, the work in [12] considered that the execution speed of a node, $\mathcal{N}_i^l$, is *constant* within its scheduling window (from b_i^l to d_i^l), and can be represented by a function of nodes execution requirement and its scheduling window. Also, the work in [12] considered that a single DAG executes at a time, and, hence the execution speed of a node is not affected by the execution speed of other nodes (of other tasks). In this work, we consider the cluster-based platform, and the execution speed of a node depends on the execution speed of other nodes (of other tasks) in the same cluster. As a result, we cannot express the execution speed of a node as a function of its execution requirement, resulting in quadratic inequality constraints (Equation (2)). This makes the optimization problem *non-convex*. □

Due to the characteristics of a clustered platform, at each instant, all cores in a cluster must execute at the speed of the fastest one. If these tasks are not well aligned (concerning their execution speed), the cluster as a whole will perform poorly (w.r.t. energy efficiency). Assigning tasks with similar speed shape on the same cluster may not be an energy efficient option (due to their sporadic releases pattern). Fig. 3 and Example 1 demonstrates one such scenario.

Example 1. In this example, we describe how the sporadic arrival pattern of a task influences the energy efficiency of the whole cluster. Consider two tasks τ_1 and τ_2 with the predefined necessary speed of execution on two processors each, to be partitioned on to the same cluster (of four processors). Fig. 3a shows the synchronous release case, where the whole cluster could run at 0 speed between [3,4) and [7,8). While Fig. 3b shows the scenario when τ_1's initial release is delayed by one-time unit, where the whole cluster will need to run at a higher speed (of 0.8) most (75 percent) of the time and thus consumes more energy.

In this example, from τ_2's perspective, direct energy reduction with existing per-task WCET based techniques may not help much, as it may be another task dominating the speed of the whole cluster most of the time. The critical observation is that, due to the extra restriction of the more realistic platform model, the speed of a cluster is determined by the heavier DAG running on it, as well as how synchronous are the releases, which could be entirely random. Moreover, even a task finishes its execution early (say, τ_2 requires no execution over [5,7)), we may not be able to reduce the cluster speed at all.

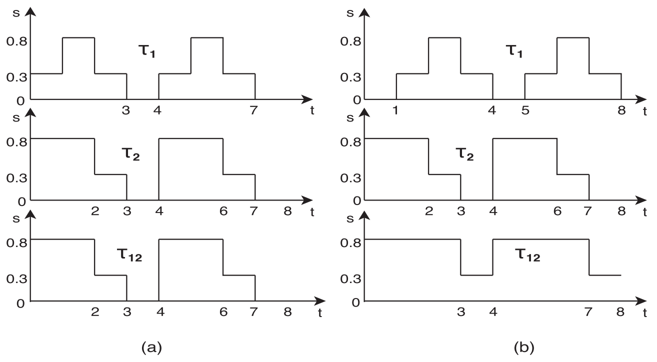


Fig. 3. When two tasks τ_1 and τ_2 with fixed speed patterns each are partitioned on to the same cluster, the resultant speed pattern (τ_{12}) of the cluster may vary for their (τ_1 and τ_2) different release offsets. In order to satisfy platform model restrictions while guaranteeing the correctness, the processors (of the same cluster) must run at the maximum/larger of the two individual speeds at each instant.

To address this issue, we propose a novel concept of speed-profile to capture the energy consumption behavior of all possible alignment scenarios.

Definition 1. *The* Speed-profile *of a task describes the percentage/likelihood of all possible speeds that the task may execute at over a period. It is a random variable $\mathcal{S}$ with an associated probability function (PF) $f_{\mathcal{S}}(s) = \mathbb{P}(\mathcal{S} = s)$, where s is a speed from the finite set of possible speeds, and $f_{\mathcal{S}}(s)$ represents the portions of the time (likelihoods) when it is running at speed s.*

Example 2. Let us consider a task τ_i with $T_i = 15$ executing at a speed of 0.6 for 5 time units (not necessarily to be continual), and at a speed of 0.5 for the rest of the time. The speed-profile of the task is thus $\mathcal{S}_i = \begin{pmatrix} 0.6 & 0.5 \\ 5/15 & 10/15 \end{pmatrix} = \begin{pmatrix} 0.6 & 0.5 \\ 0.33 & 0.67 \end{pmatrix}$. At any specific time, t, there is about 33 percent probability that the cores are running at the speed of 0.6 unit and about 67 percent probability that the cores are running at the speed of 0.5 unit.

It is evident that from another task's point of view, the speed-profile provides probabilistic information on how the task of interest would restrict the lower bound to the speed of the cluster over time. As the alignment of releases between any two tasks is unknown, we assume in the future analysis that any phase difference is of equal chance over the long run.

Remark 1. The speed-profile $\mathcal{S}_i$ of a given task τ_i remains the same for an initial phase (release offset) $\phi_i \geq 0$. Regarding inter-task combinations, we assume uniform distribution for the phase of any task; i.e., $\phi_i \sim U[0, T_i)$.

Section 4.1 details the calculation for task speed-profile. Here, we describe the calculation of the cluster speed-profile when two tasks are combined on to the same cluster.

3.2 Speed-Profile for the Cluster

As stated earlier, the property of the clustered platform and sporadic arrival pattern of a task makes the exact speed of the cluster unpredictable at a specific time instant (see Fig. 3 and Example 1). As a result, when two tasks τ_i and τ_j (with speed-profiles) are being considered allocating to the same cluster, we need to construct the merged speed-profile of the cluster (executing them both). To perform such calculation, we introduce a special $\odot$ operator that takes the maximum of the two profiles on a probability basis.[1]

Definition 2. *The special operator $\odot$ operates on two (or more) random variables $\mathcal{X}$ and $\mathcal{Y}$. During this operation, each entry $\mathcal{X}_i \in \mathcal{X}$ is compared with each entry $\mathcal{Y}_j \in \mathcal{Y}$ and the value $\mathcal{Z}_{ij}$ is calculated as $\mathcal{Z}_{ij} = max(\mathcal{X}_i, \mathcal{Y}_j)$, with a combined (multiplied) probability. If there are multiple occurrences of an entry, all of them are merged into a single entry, and their associated probability are summed together.*

Example 3. Let $\mathcal{S}_i = \begin{pmatrix} 6 & 5 \\ 0.4 & 0.6 \end{pmatrix}$, $\mathcal{S}_j = \begin{pmatrix} 6 & 2 \\ 0.4 & 0.6 \end{pmatrix}$. Then $\mathcal{S}_i \odot \mathcal{S}_j = \begin{pmatrix} 6 & 6 & 6 & 5 \\ 0.16 & 0.24 & 0.24 & 0.36 \end{pmatrix} = \begin{pmatrix} 6 & 5 \\ 0.64 & 0.36 \end{pmatrix}$.

Note that we allocate two different DAGs (with same/different periods) to the same cluster. The speed-profile indicates how long a DAG executes at different speeds within its deadline, i.e., the probability that a DAG executes at a specific speed. The task's period becomes irrelevant as speed-profile is a probability-based measurement. Once τ_i and τ_j are allocated to the same cluster, we use $\mathcal{S}_{ij}$ to denote the speed-profile of the cluster (see Example 3).

In summary, energy minimization in a cluster-based platform is challenging because of sporadic release pattern and the idle gaps between a task deadline and its period. To tackle these problems, we have introduced the concept of speed-profile for both an individual task and a cluster where multiple tasks can be allocated.

4 Task Partitioning Algorithm

The ultimate goal of the paper is to partition all DAGs into clusters, such that overall platform energy consumption is minimized. Recall that on a clustered multiprocessor platform, at a given instant, all processors in the same cluster must execute at the same speed. Due to this property of a cluster-based platform, if two tasks that are not well-aligned (in terms of execution speed) are allocated to the same cluster, it will result in reduced energy efficiency. So, we have proposed the concept of speed-profiles (refer to Section 3) which is a tool to measure the potential long-term energy saving of a cluster when partitioning any pair of DAGs into this cluster. So far we have discussed the importance of the concept of speed-profile but did not mention how to create them given a DAG task, which is the focus on Section 4.1. Then, Section 4.4 describes the task-to-cluster partitioning algorithm.

4.1 Creating the Speed-Profile of a Task

Given a DAG task τ_i, we provide two approaches to create the speed-profile $\mathcal{S}_i$.

Approach A: Considering the Maximum Speed from all the Cores. Upon applying the task decomposition, segment extension, and intra-task processor merging techniques (Section 2), some vital information (e.g., the speed of a core at a specific time and number of cores required) becomes available. This information plays a role to calculating the speed-profile S_i of task τ_i. At any time instant t, we consider the maximum speed from all the cores available. It ensures the sufficient speed so that even the heaviest node can finish execution within its scheduling window (defined after task decomposition). We consider constrained deadline (i.e., $D_i \leq T_i$), so the task must have to finish by D_i and rest of the time $(T_i - D_i)$ it remains idle. For each segment $j \in \tau_i$, (summation of the length of these segments is equal to D_i), we create a pair $(s_{i,j}, p_{i,j})$. For the jth segment, $s_{i,j}$ and $p_{i,j}$ respectively denote the maximum execution speed and the probability that the cluster will run at this speed. Let, M cores are allocated to τ_i. At jth segment, we calculate $s_{i,j}$ and $p_{i,j}$ as follows:

$$s_{i,j} = \max_{k \leq M}\{s_{i,j,k}\}, p_{i,j} = \frac{t^c_j}{T_i}.$$

Here, $s_{i,j,k}$ denotes the speed of kth core at jth segment and t^c_j is the length of jth segment. The speed-profile $\mathcal{S}_i$ will be

$$\mathcal{S}_i = \begin{pmatrix} s_{i,1} & s_{i,2} & \cdots & s_{i,z} & 0 \\ p_{i,1} & p_{i,2} & \cdots & p_{i,z} & (T_i - D_i)/T_i \end{pmatrix}.$$

The last pair reflects the fact that the core remains idle for the $(T_i - D_i)$ time units at the end of each period.

Example 4. Consider a task τ_i with $T_i = 15$, $D_i = 12$ and $C_i = 6.5$. Let, the task is partitioned into three segments of length 5, 7 and 3 time units respectively, where the processor is executing at a (maximum) speed of 0.6 in the first segment, speed of 0.5 in the second segment, and remain idle in the third segment The speed-profile is

$$\mathcal{S}_i = \begin{pmatrix} 0.6 & 0.5 & 0 \\ 0.33 & 0.47 & 0.2 \end{pmatrix}.$$

Note that, if a cluster contains a single task τ_i, then $\mathcal{S}_i$ also represents the cluster speed-profile. If τ_i and τ_j (or more tasks) are executing on the same cluster, then the technique described in Section 3.2 needs to be applied before making any choices. The greedy choosing approach for task partition is detailed in Section 4.4.

Approach B: A Single Speed Throughout. Theorem 4 of [12] shares a valuable insight: *The total energy consumption (assuming processor remains on) is minimized in any scheduling window when execution speed remains uniform (the same) throughout the interval.*Motivated by it,[2] we propose another approach of selecting a single speed for a DAG task (job) during the whole duration from its release until its deadline.

1. Although the appearance of the proposed operator is identical to [25], the calculation is quite different. This is due to the "larger value dominating" nature of the platform model considered in this paper.

2. Note that [12] considered that the speed remains constant within a scheduling slot for each processor. Also, they assumed per core speed scaling and calculated the speed within each scheduling slot through a convex optimization method. This paper considers the clustered platform where the objective function becomes *non-convex* (see Lemma 1) and thus the existing approach is inefficient.

In this approach, we consider the maximum workload (or the execution requirement) from all the cores available and determine the aggregated workload. Upon dividing the aggregated workload by the deadline, we get the desired single speed. Let M cores be allocated to task τ_i. At jth segment, the execution requirement of the kth core is denoted by $w_{i,j,k}$, which is calculated as follows:

$$w_{i,j,k} = s_{i,j,k} \times t^c_j.$$

We determine the maximum execution requirement as follows:

$$w_{i,j} = \max_{k \le M}\{w_{i,j,k}\}.$$

Let Z denotes the total number of segments in τ_i. The maximum total workload w_i and the desired single speed s_i is calculated using the following equations:

$$w_i = \sum_{j=1}^{Z} w_{i,j}, \qquad s_i = \frac{w_i}{D_i}. \tag{4}$$

Other than the idle pair $(0, (T_i - D_i)/T_i)$, we consider a single speed throughout the deadline so only a single pair (s_i, p_i) is required, where $s_i = w_i/D_i$ and $p_i = D_i/T_i$.

Example 5. Consider the task described in Example 4 ($T_i = 15,\ D_i = 12$ and $C_i = 6.5$). It must finish 6.5 unit of workloads within 12-time units. Using this approach its speed-profile is

$$\mathcal{S}_i = \begin{pmatrix} 0.54 & 0 \\ 0.8 & 0.2 \end{pmatrix}.$$

Lemma 2. *If a task τ_i executes according to the speed-profile $\mathcal{S}_i$, it guarantees real-time correctness.*

Proof. It has been observed in [12] that the following constraint guarantees the real-time correctness

$$\forall l : \mathcal{N}^l_i \in \tau_i :: \sum_{k=b^l_i}^{d^l_i} t^c_k S^{\mathcal{M}^l_i}_k \ge c^{\mathcal{N}^l_i}_i. \tag{5}$$

Here, b^l_i and d^l_i denotes the release time and deadline of $\mathcal{N}^l_i$, $\mathcal{M}^l_i$ denotes the node-to-processor mapping and $S^{\mathcal{M}^l_i}_k$ is the speed of the processor (where $\mathcal{N}^l_i$ is allocated) at kth segment. Unlike to [12], at any time instant t, we choose either the maximum speed from all the cores running on the same cluster (Approach A) or a single speed that can guarantee the maximum execution requirement for the whole duration up to τ_i's deadline (Approach B). So, at any time instant, the cluster speed is larger or equals to the speed of any individual core. Considering Equations (2) and (5) we can deduce that

$$\forall l : \mathcal{N}^l_i \in \tau_i :: \sum_{k=b^l_i}^{d^l_i} t^c_k s_{i,k} \ge \sum_{k=b^l_i}^{d^l_i} t^c_k S^{\mathcal{M}^l_i}_k \ge c^{\mathcal{N}^l_i}_i.$$

So, we conclude that Executing a task with speed according to the speed-profile $\mathcal{S}_i$ guarantees real-time correctness. □

An Efficient Approach for Implicit Deadline System. By adopting simple modification in Equation (4), it is possible to apply the process mentioned above for the implicit deadline tasks also. The workload w_i should be divided by the period instead of the deadline. We consider the same speed through the task period, so only a single pair (s_i, p_i) is required, where $s_i = w_i/T_i$ and $p_i = 1$.

Example 6. Now we create the speed-profile for the task described in Examples 4 and 5 considering implicit deadline. So it has $T_i = D_i = 15$ and $C_i = 6.5$. Let's assume that it is executed at a speed of 0.6 for 5-time units, at a speed of 0.35 for 10-time units. According to Approach A, the speed-profile is

$$\mathcal{S}_i = \begin{pmatrix} 0.6 & 0.35 \\ 0.33 & 0.67 \end{pmatrix},$$

and according to Approach B, the speed-profile is

$$\mathcal{S}_i = \begin{pmatrix} 0.43 \\ 1 \end{pmatrix}.$$

4.2 Discretization of the Speed-Profile

In Section 4.1, we have described two approaches to create the speed-profile for an individual task. While creating the speed-profiles, those approaches assume a continuous frequency scheme. From a practical point of view, discrete frequency mode should be preferred over the continuous frequency mode, because a real platform supports only a set of frequencies. Now, we describe the technique to discretize all the speeds available in a speed-profile (assuming that the speed-profile is already created).

Suppose, we execute a task τ_i (and its speed-profile is $\mathcal{S}_i$) in a real-platform, and this platform supports only those speeds available on a speed-set $\mathcal{Z}$. Note that the content of $\mathcal{Z}$ is dependent on the platform. For example, ODROID XU-3 supports a frequency range of 200-1400 MHz (LITTLE cluster) and 200-2000 MHz (big cluster) with scale steps of 100 MHz). Now, for each entry $s_{i,j} \in \mathcal{S}_i$, we find the minimum speed $\mathcal{Z}_k \in \mathcal{Z}$, where $\mathcal{Z}_k \ge s_{i,j}$. Once, we find an appropriate $\mathcal{Z}_k$; we set the value of $s_{i,j}$ as $s_{i,j} = \mathcal{Z}_k$.

Example 7. Consider a task τ_i with the same speed-profile from Example 4. Let us assume that we will execute τ_i in a platform where $\mathcal{Z} = \{0, 0.2, 0.4, 0.55, 0.75, \text{and } 1\}$, i.e., this platform supports only six discrete speeds, and all the speeds are normalized w.r.t. the maximum speed supported by this platform. Considering the speed-profile $\mathcal{S}_i$ (from Example 4) and the speed-set $\mathcal{Z}$, we find that:

(a) $s_{i,1} \le \{\mathcal{Z}_5 \text{ and } \mathcal{Z}_6\}$
(b) $s_{i,2} \le \{\mathcal{Z}_4, \mathcal{Z}_5 \text{ and } \mathcal{Z}_6\}$, and
(c) $s_{i,3} \le \{\mathcal{Z}_1, \mathcal{Z}_2, \mathcal{Z}_3, \mathcal{Z}_4, \mathcal{Z}_5 \text{ and } \mathcal{Z}_6\}$.

Now, we choose the minimum $\mathcal{Z}_k \in \mathcal{Z}$ such that $\mathcal{Z}_k \ge s_{i,j}$. So, we assign $\mathcal{Z}_5$ to $s_{i,1}$, $\mathcal{Z}_4$ to $s_{i,2}$, and $\mathcal{Z}_1$ to $s_{i,3}$. Now, the updated (i.e., discretized) speed-profile becomes

$$\mathcal{S}_i = \begin{pmatrix} 0.75 & 0.55 & 0 \\ 0.33 & 0.47 & 0.2 \end{pmatrix}.$$

Theorem 1. *When a task executes with its discretized speed-profile, it guarantees that the task will not miss the deadline.*

TABLE 1
Estimated Parameters for Different Cluster of an ODROID XU-3 Board

Cluster Type	$\beta(W)$	$\alpha(W/MHz^{\gamma})$	γ
big	0.155	3.03×10^{-9}	2.621
LITTLE	0.028	2.62×10^{-9}	2.12

This table is adopted from [15].

Proof. We have shown in Lemma 2 that a task τ_i will not miss the deadline if executed according to its speed-profile $\mathcal{S}_i$. If we discretize τ_i's speed-profile and execute τ_i according to this speed-profile, then the task still guarantees the real-time correctness. This is because any speed $s_{i,j}$ of the discretized speed-profile is greater than or equal to $s_{i,j}$ when it was continuous. □

4.3 Handling Platform Heterogeneity

In this section, we discuss a specific type of multi-core platform with diverse computing abilities: heterogeneous multi-core platform. We first discuss different types of heterogeneous platforms, and then explain how our proposed techniques can be extended to handle heterogeneity. In a heterogeneous platform, different cores have different computational capabilities. In terms of speed, Funk defined a widely-accepted classification of the heterogeneous platform [26] as follows, where the speed of the processor denotes the work completed (while executing a task) in a single-time unit by this processor.

(i) *Identical multiprocessors*: On Identical multiprocessors, all tasks are executed at the same speed on any processor;
(ii) *Uniform multiprocessors*: On Uniform multiprocessors, all the tasks execute at the same speed if allocated on the same processor, but at a different speed on different processors. So, the execution speed of a task depends on the processor where the task is allocated.
(iii) *Unrelated multiprocessors*: On Unrelated multiprocessors, execution speeds of different tasks may vary on the same processor, i.e., a task's execution speed depends on both the task itself and the processor where it is allocated.

In a heterogeneous platform, each core is designed with a different computational capability, and an efficient task-to-core mapping improves the system resource efficiency. In the context of energy efficiency, two major directions have been mentioned in [27] for any heterogeneous platform:

(i) Find an appropriate core/cluster for task mapping to reduce the overall power consumption of the whole platform.
(ii) Deploy energy-aware scheduling techniques on each core/cluster to reduce power consumption.

Our proposed approach covers both directions. First, we use speed profile to identify efficient core/cluster to task mapping and then try to reduce the overall cluster speed as much as possible. It works for an identical heterogeneous platform (a.k.a., homogeneous multiprocessor) as task-to-core mapping does not impact energy consumption much.

TABLE 2
The "Uncore" Power Consumption for Different Cluster of an ODROID XU-3 Board

Freq(GHz)	2	1.8	1.6	1.4	1.2	1.0
big cluster(W)	0.8	0.528	0.39	0.309	0.244	0.182
Freq(GHz)	1.4	1.2	1.0	0.8	0.6	0.4
LITTLE cluster(W)	0.04	0.04	0.04	0.04	0.04	0.04

This table is adopted from [15].

Now, we extend our approach to apply to the uniform heterogeneous platform by modifying the parameters in the power model in Equation (1), i.e., setting different $\alpha, \beta,$ and γ values for the 'big' and 'LITTLE' cluster. Under such consideration, different clusters no longer share the same power model, and the same task may have different execution requirements on different clusters. We report the estimated values of $\alpha, \beta,$ and γ in Table 1. These parameters are adopted from [15]. The work in [15] estimated these parameters for the ODROID XU-3 board using the real power measurements along with a curve fitting method. They have also assumed that there is another contributor to the total power consumption of a cluster, i.e., the "uncore" power consumption (reported in Table 2). The "uncore" power consumption introduced in the system from some components other than a processor, e.g., a shared cache. Similar to the dynamic power consumption, the "uncore" power consumption also depends on the processor frequency. However, unlike the dynamic power consumption, there is always some "uncore" power consumption as long as the cluster remains on (even if there is no workload on a processor).

Considering all the parameters from Tables 1 and 2, we bring the following modification in Equation (1)

$$P(s) = N_p\beta + \alpha s^{\gamma} + P_s(f). \tag{6}$$

Here, N_p denotes the number of cores per cluster, and $P_s(f)$ denotes the "uncore" power consumption.

We have a different power model for the "big" and the "LITTLE" cluster, but we still don't know what the basis of assigning a task to a cluster is. Recall that, while creating the speed-profile, some vital information (e.g., the speed of a core at a specific time) were known to us (Section 4.1). If the execution speed of a task is greater than a certain threshold at any point from its release to its deadline, then we assign this task to the big cluster. Else, we assign this task to the LITTLE cluster. For the platform we consider (ODROID XU-3), we set the threshold to 0.7. It is the ratio of the maximum speed supported by the big cluster and the LITTLE cluster (see Table 2).

4.4 Task Partition: Greedy Merging With Speed-Profiles

We are now equipped with tools (speed-profiles) to measure the potential long-term energy saving of a cluster when partitioning any pair of DAG tasks into it. This subsection describes the scheme for selecting pair by pair so that the total number of clusters can be significantly smaller than the total number of tasks.

Let, we decide for each task whether it should be allocated on a LITTLE or a big cluster using the technique described in Section 4.3. To select a (task) pair that will share the same cluster, we greedily choose the pair that provides maximum power saving, as depicted in Algorithm 1. Note that we allow the pairing of two DAGs that are not merged previously. Also, if any task uses more cores than what is available in a cluster, that task cannot be merged with that cluster.

Algorithm 1. Greedy Merging

```
1:  Input: Task-set $\tau$, with speed-profile $\mathcal{S}_i$ (computed using
    approach A or approach B) for each task
2:  Output: Speed-profile $\tilde{\mathcal{S}}$ (with processor power saving).
3:  $\bar{\mathcal{S}}, \tilde{\mathcal{S}} \leftarrow \emptyset$ ▷ All the possible/selected speed-profiles
4:  for $i = 1$ to $n$ do
5:      for $j = i + 1$ to $n$ do
6:          $\mathcal{S}_{ij} \leftarrow \mathcal{S}_i \odot \mathcal{S}_j$; $\bar{\mathcal{S}} \leftarrow \bar{\mathcal{S}} \cup \mathcal{S}_{ij}$;
7:      end for
8:  end for
9:  while $\exists \mathcal{S}_{xy} \in \bar{\mathcal{S}}$ and $\mathcal{S}_{xy}$ provides non-zero power saving do
10:     $\mathcal{S}_{xy} \leftarrow$ the pair from $\bar{\mathcal{S}}$ with maximum power saving
11:     $\tilde{S} \leftarrow \tilde{S} \cup \mathcal{S}_{xy}$
12:     for $k = 1$ to $n$ do
13:         $\bar{\mathcal{S}} \leftarrow \bar{\mathcal{S}} - \mathcal{S}_{kx} - \mathcal{S}_{xk} - \mathcal{S}_{ky} - \mathcal{S}_{yk}$
14:     end for
15: end while
16: return$\tilde{S}$.
```

Algorithm 1 creates two empty lists $\bar{\mathcal{S}}$ and $\tilde{\mathcal{S}}$ that will contain all the possible and selected speed-profiles (Line 3). Lines 4–8, calculate all the possible speed-profiles and insert them into $\bar{\mathcal{S}}$. We greedily select a pair of DAGs that provide the maximum power saving (calculated using Equations (6) and (10) from [12]) and update the list $\bar{\mathcal{S}}$ by removing the pair from any further merging (Lines 9–15). The list $\tilde{\mathcal{S}}$ is also updated by adding the selected pair (Line 11). We conclude by returning the updated list $\tilde{\mathcal{S}}$ (Line 16).

Theorem 2. *Executing a task with a speed according to the cluster speed-profile guarantees real-time correctness.*

Proof. We have shown in Lemma 2 that a task τ_i will not miss the deadline if executed according to its speed-profile $\mathcal{S}_i$. If τ_i share a cluster with another task τ_j and executes according to the merged (i.e., cluster) speed-profile $\mathcal{S}_{ij}$, then it still guarantees the real-time correctness, because $\mathcal{S}_{ij} \geq \mathcal{S}_i$ holds at any time instant. □

Remark 2. For n tasks, the time complexity to generate all possible speed-profiles, $\bar{\mathcal{S}}$, is $O(n^2 Z)$, where Z is the maximum number of segments of all DAGs in the set after decomposition (related to the structure and size of the DAGs). Algorithm 1 greedily choose a speed-profile by iterating through $\mathcal{S}$ and then update, which takes $O(n^2)$ time as well. Thus the total complexity of the proposed method is $O(n^2)$.

In summary, we have proposed two methods (Section 4.1) to create the speed-profile for a constrained-deadline DAG. We also show that if a task executes according to the speed-profile, it ensures real-time correctness. According to the techniques provided in Section 3, we could evaluate and compare all potential pairs of the combination by calculating the cluster speed-profile after merging. Finally, Section 4.4 discussed how to use these speed-profiles to find suitable partners to share a cluster greedily.

5 System Experiments

In this section, we present experimental results conducted on an ODROID XU-3 board. The platform runs on Ubuntu 16.04 LTS with Linux kernel 3.10.105. It is fabricated with Samsung Exynos5422 Octa-core SoC, consisting of two quad-core clusters, one 'LITTLE' cluster with four energy-efficient ARM Cortex-A7 and one 'big' cluster with four performance-efficient ARM Cortex-A15. Four TI INA231 power sensors are integrated onto the board to provide real-time power monitoring for the A-7 and A-15 clusters, GPU, and DRAM. An energy monitoring script, `emoxu3` [28], is used to log energy consumption of the workloads.

DAG Generation. In this experiment, we generate two task sets each with 300 DAGs, and use the widely used Erdos-Renyi [29] method to generate a DAG. We tune a parameter p, that denotes the probability of having an edge between two nodes. In this experiment, we set p to 0.25 generate DAGs with an uncomplicated structure. If a disconnected DAG is generated, we add the fewest number of edges to make it connected. For experimentation, we have considered arbitrary task periods, and it is determined using Gamma distribution [30]. We set the periods with $T_i = L_i + 2(C_i/m)(1 + \Gamma(2,1)/4)$ [2]. Here, L_i is the critical path length of τ_i, calculated according to the definition of L_i (refer to Section 2).

After generating the topology of each DAG of a set, we partition them into two subsets according to the proposed approach, one to the "big" and the other one to the "LITTLE" cluster, and measure the energy consumption over the hyper-period of all DAGs. We use `rt-app` [31] to emulate the workload for each node. rt-app simulates a real-time periodic load and utilizes the POSIX threads model to call and execute threads. For each thread, an execution time needs to be assigned. In this experiment, for each node, we randomly select an execution time ranged between $[300\ \mathrm{ms}, 700\ \mathrm{ms}]$. rt-app itself has a latency that varies randomly between $13 - 150\ \mathrm{ms}$ per thread. Therefore, we add the maximum latency of rt-app, i.e., 150 ms, to the execution time of each thread from an analytical point of view.

DAG Scheduling. We use the Linux built-in real-time scheduler `sched_FIFO` to schedule the DAGs. Compared to the other system tasks, DAGs are assigned with higher priorities so that they can execute without interference. Our approach is also applicable to other preemptive schedulers which feature the work-conserving property.

Frequency Scaling. According to the frequency/speed-profile (Section 4), we use `cpufreq-set` program (from `cpufrequtils` package) to change the system's frequency online. We use the ODROID XU-3 board, where scaling-down (up) the frequency of the big cluster takes at most $60\ (40)\ \mathrm{ms}$, respectively. On the LITTLE cluster, both the operation takes at most $15\ \mathrm{ms}$. Due to this delay, the hyper-period of all DAGs becomes large ($230s$, in this experiment). We detail the reasons behind this delay in Appendix B.2, available in the online supplemental material.

The Reference Approach. Since no work has studied the same problem considered in this paper, we do not have a

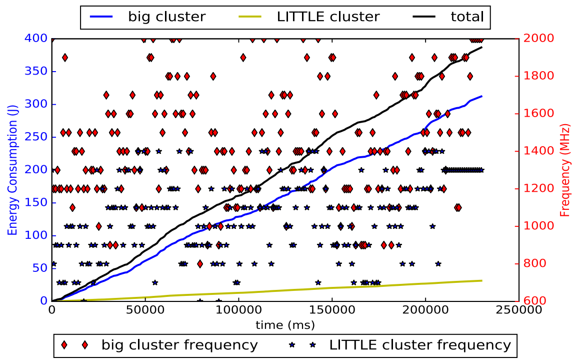


Fig. 4. The energy consumption and the frequency variation of our proposed approach on ODROID XU-3.

direct baseline for comparison. So, we propose a reference approach based on the studies for energy-efficient scheduling of sequential tasks [32]. They assigned an operational frequency to each task, and at run-time, schedule them according to their frequency. In this reference approach, we compute an operational frequency for each DAG. This frequency stretches out execution length of these DAGs as much as possible without violating their deadlines. As stated earlier, the reference approach executes the DAGs with the same partition, but without the merging techniques proposed in Section 4.

Results. The experimental results are plotted in Figs. 4 and 5. In these figures, we show (i) the energy consumption over the hyper-period (230*s*), where the three lines show the energy consumption of the big and LITTLE cluster, and the total system; and (ii) frequency variation during the run-time, where the diamond and star marks denote the operational frequency of the big and the LITTLE cluster at a specific time instant, respectively. Note that the GPU and DRAM also contribute the energy consumption of the total system. Hence, the total energy consumption is a bit higher than the summation of the contribution of the big and the LITTLE cluster, but it is observed that there is a negligible difference for the energy consumption of GPU and DRAM between the two approaches. Besides, it is worth noticing that this energy consumption also accounts for energy consumption of the operating system.

Table 3 summarizes the comparison of the experimental results, where the energy consumption of the two clusters and the total system is presented, and the energy saving from our approach is given. As can be seen, our approach

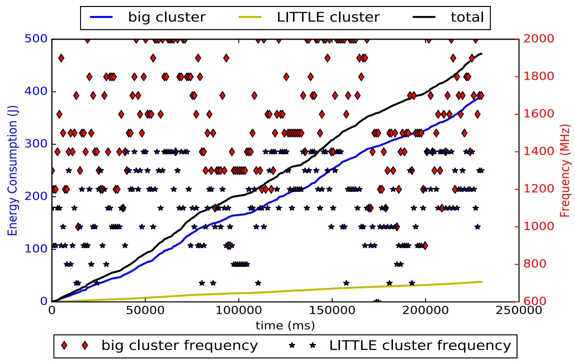


Fig. 5. The energy consumption and the frequency variation of the reference approach on ODROID XU-3.

TABLE 3
Summary of Experimental Results

	Ours (J)	Ref (J)	Energy Saving (%)
big cluster	312	389	20
LITTLE cluster	32	38	16
Total	387	472	18

consumes 312J and 32J on the big and the LITTLE cluster, respectively. Comparing to the reference approach, we save energy consumption by 20 and 16 percent. In total, our approach saves energy consumption by 18 percent.

The result can be justified as the reference approach changes the frequency for each DAG, while ours have a fine-grained frequency adjustment at each segment (Section 4.1), and could scale down the frequency if required. Fig. 6 presents the frequency occurrence probability of two clusters which is recorded per second by emoxu3. We observe that within the same time interval the reference approach has a higher probability to execute at a higher frequency, while our approach is more likely to execute at the lower frequencies, thus reducing the energy consumption.

Remark 3. Each heavy DAG ($C_i > T_i$) needs two or more cores while executing and the ODROID XU-3 board contains four cores per cluster. So, in this experimental setup, we can not execute more than four heavy DAGs at a time. Such a restriction is not applicable to the light DAGs ($C_i \leq T_i$). We also consider that a heavy DAG cannot be allocated in multiple clusters.

6 Simulations

For large-scale evaluation, we perform simulations and compare the results with existing baselines. We generate DAGs using the Erdos-Renyi method (Section 5). We consider two types of task periods; *(a) harmonic periods*, where the task period T_i is enforced to be an integral power of 2. We define T_i as $T_i = 2^\alpha$, where α is the minimum value such that $2^\alpha \geq L_i$, where L_i is the critical path length of τ_i*(b) arbitrary periods*, T_i is determined using Gamma distribution (see Section 5).

We compare our approaches with some existing baselines studied in [12], [17], [33]. Total power consumption by our approach and by these baselines are calculated using

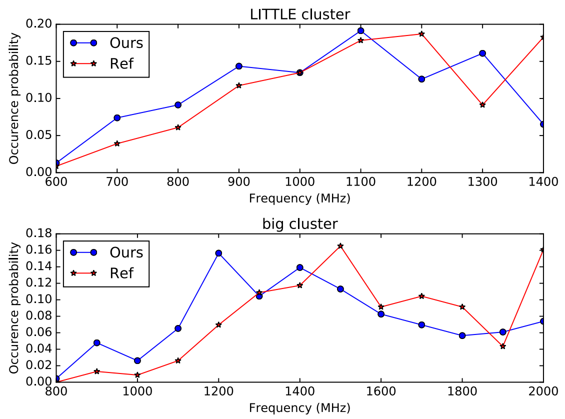


Fig. 6. Frequency occurrence probabilities.

Equation (6). As mentioned earlier, [12] considered per-core DVFS, i.e., each core individually is an island of the cluster-based platform. For a fair comparison, according to the scheduling policy of [12], when a task is allocated on some cores at any time instant t, we choose the maximum speed among all these cores. We consider [12] as a baseline because that work is closely related to ours. Although they have considered per-core DVFS and restrict their attention only to implicit deadline tasks, the task and the power model are same. Besides, although this work and [12] propose different approaches to power saving, the initial (preparation) steps of both approaches are based on commonly known techniques like task decomposition, task merging, etc.

The work in [17] studied a greedy slack stealing (GSS) scheduling approach considering inter-dependent sequential tasks. It considered the DAG model to represent dependencies among the tasks. In GSS, the *slack* (unused time in actual computation requirement of a task) is reclaimed by one task by shifting others towards the deadline. They did not consider repetitive tasks; hence it can be regarded as scheduling a single task. Besides this, the power and graph model used in [17] is different from ours. To ensure a fair comparison, we execute the GSS algorithm using the power model in Equation (1) and assume that once introduced in the system; a processor remains active. We also consider a minimum inter-arrival separation for a DAG. That work considered three different kinds of nodes: *AND, OR*, and *Computation* nodes (Section 2.1 in [17]). A computation node has both the maximum and average computation requirement. To comply with our work where the focus in energy reduction while guaranteeing worst-case temporal correctness, we execute the GSS algorithm considering only the computation nodes with their maximum computation requirement. We made all the changes in order to provide a fair comparison. Despite these differences, we chose [17] as a baseline because they studied a *GSS* approach for energy minimization. They considered the inter-dependent sequential tasks and their dependencies was represented by a DAG, which is similar to our task model.

We also consider [33] as a baseline because this work considered scheduling a set of independent periodic applications, where each application is modeled as a DAG. They proposed an approach for energy minimization combining the DVFS and the DPM policy. Similar to [12] and [17], the work in [33] considered per-core DVFS.

We compare power consumption by varying two parameters for each task: *task periods (utilization)* and *the number of nodes*. We randomly generate 25 sets of DAG tasks and compare the average power consumption.

Notations of Referenced Approaches. For the task partitioning step, either we randomly choose any two and allocate them to the same cluster, or greedily choose the ones with lowest speed as proposed. Regarding speed-profile calculation, there are also two options (Approaches A and B in Section 4.1). Combining these options in two steps lead to four baselines: *MaxSpeed_Greedy*, *SingleSpeed_Greedy*, *MaxSpeed_Random*, *SingleSpeed_Random*. Also, three baselines mentioned above are included for comparison:

- Federated scheduling with intra-task processor merging [12], denoted by *Fed_Guo*;
- GSS algorithm [17], denoted by *GSS_Zhu*.
- DVFS and DPM combination [33], denoted by *com_Chen*.

6.1 Uniform Heterogeneous Platform With a Continuous Frequency Scheme

In this section, considering the uniform heterogeneous platform and a continuous frequency scheme, we report the power consumption comparison for different approaches mentioned earlier. Under such a platform, different clusters no longer share the same power model and we use the power model described in Equation (6). We present the power comparison results in an identical heterogeneous platform (from [34]) in Appendix A, available in the online supplemental material.

6.1.1 Constrained Deadline Task

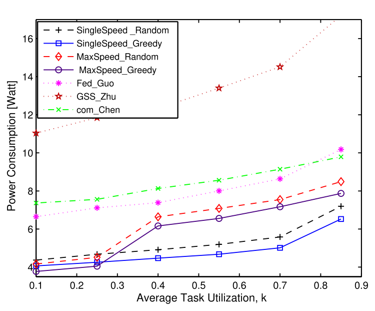
Here, we report the power consumption under the scheme for constrained deadline tasks mentioned in Section 4. We evaluate the efficiency of our proposed method by changing two parameters; task period (utilization) and the number of nodes in the task.

Effect of Varying Task Periods (Utilization). Here we control the average task utilization through varying the task period. In order to make the task schedulable, the critical path length L_i of task τ_i should not exceed its deadline D_i. We vary the period in a range ($L_i \leq T_i \leq C_i$). The parameter L_i and C_i are measured once the DAG is generated according to the technique described in Section 5. We also use the following equation (according to [12]) to ensure that the value of T_i satisfies the range ($L_i \leq T_i \leq C_i$)

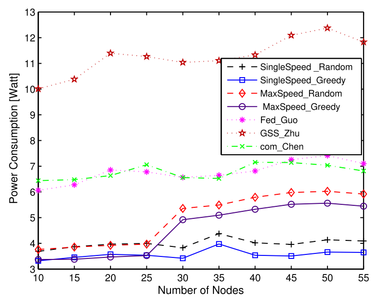
$$T_i = L_i + (1-k)(C_i - L_i). \tag{7}$$

Here, $k \in [0, 1]$ is task utilization. As we are considering the constrained deadline tasks, D_i is randomly picked from the range ($L_i \leq D_i \leq T_i$). The results are presented in Fig. 7a. Note that when any parameter (e.g., number of nodes in a DAG, task utilization) changes, savings in energy randomly vary within a small range and we consider the minimum value among them. The results indicate a proportional relationship between the average power consumption and average task utilization. It happens because a higher task utilization imposes tighter real-time restrictions. It restricts (refer to Fig. 2b) the space for the segment length optimization. In this experiment, the number of nodes is fixed to 30. Fig. 7a shows that *SingleSpeed_Greedy* approach performs better for a higher utilization value. On average, the *SingleSpeed_Greedy* approach leads to a power saving of at least 30.23 and 60.2 percent compared to *Fed_Guo* and *GSS_Zhu* approaches, respectively. In *SingleSpeed_Greedy* approach, a task executes with a single speed throughout the deadline. During the task partitioning step, a suitable partner (with similar speed-profile) leads to energy efficiency. However, for the other approaches task speed may vary throughout the deadline. In that case, evil alignment and a significant variation in the speed may reduce energy efficiency (see Fig. 3 and Example 1).

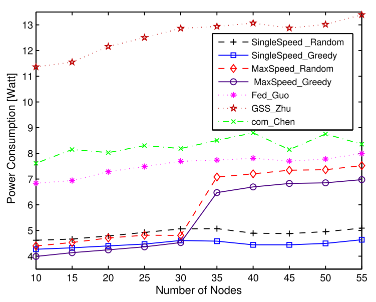
Effect of Varying the Numbers of Nodes. Now we vary the number of nodes (10 to 55) (T_i is fixed) and report the average power consumption. We report the average power consumption for harmonic deadline tasks in Fig. 7b and



(a) Under Varying Utilization

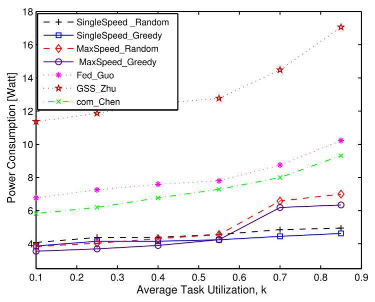


(b) Under Varying Number of Nodes (Harmonic Period)

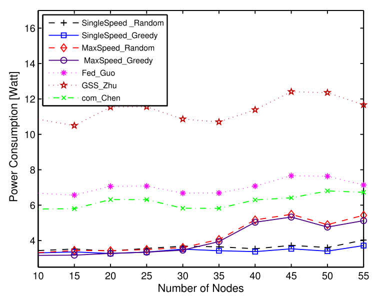


(c) Under Varying Number of Nodes (Arbitrary Period)

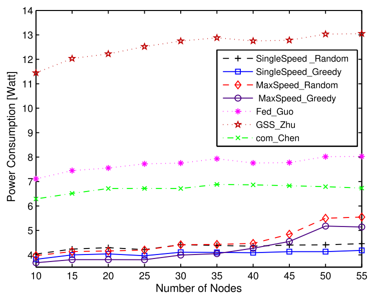
Fig. 7. Power consumption comparison between different approaches for the *constrained* deadline tasks considering a *continuous* frequency scheme on the *uniform* heterogeneous platform.



(a) Under Varying Utilization

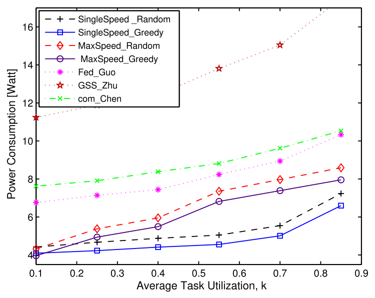


(b) Under Varying Number of Nodes (Harmonic Period)

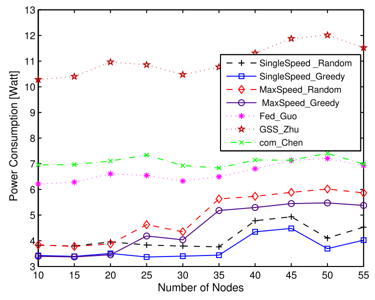


(c) Under Varying Number of Nodes (Arbitrary Period)

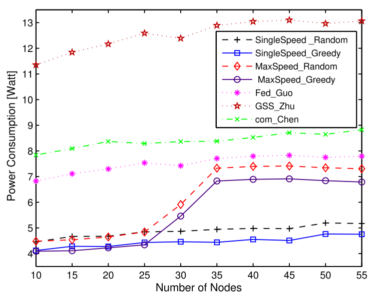
Fig. 8. Power consumption comparison between different approaches for the *implicit* deadline tasks considering a *continuous* frequency scheme on the *uniform* heterogeneous platform.



(a) Under Varying Utilization



(b) Under Varying Number of Nodes (Harmonic Period)



(c) Under Varying Number of Nodes (Arbitrary Period)

Fig. 9. Power consumption comparison between different approaches for the *constrained* deadline tasks considering a *discrete* frequency scheme on the *uniform* heterogeneous platform.

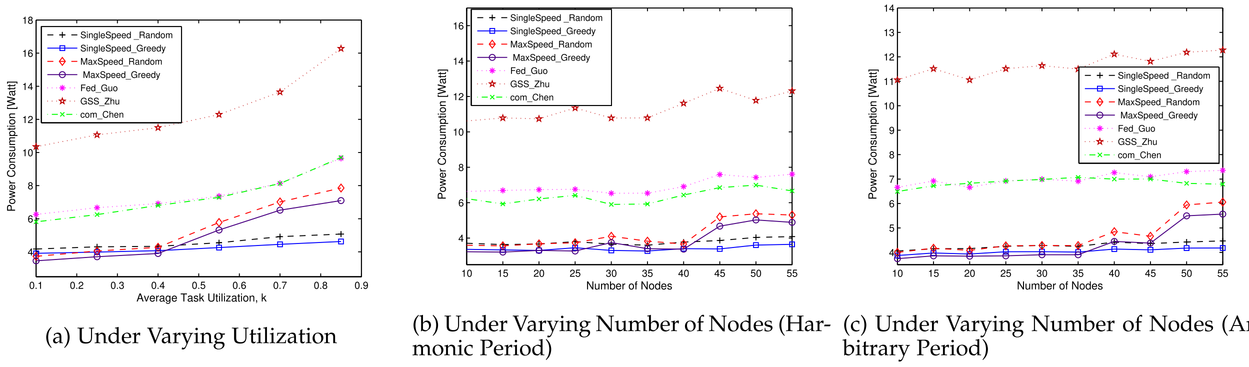
arbitrary deadline tasks in Fig. 7c. We observe that the power consumption pattern does not change that much, i.e., SingleSpeed_Greedy approach outperforms other approaches especially when the number of nodes (in each DAG) are high, 35 or higher. Specifically, under harmonic task periods, the SingleSpeed_Greedy incurs 40.19 and 65.9 percent less power on average compared to Fed_Guo and GSS_Zhu; under arbitrary task periods, the savings potential are 33.43 and 61.96 percent, respectively.

6.1.2 Implicit Deadline Task

Effect of Varying Task Periods (Utilization). Using previous setup (Section 6.1.1), We observe that the average energy consumption is directly proportional to the average task utilization.

Fig. 8a shows that *SingleSpeed_Greedy* approach performs better for a higher utilization value and on average, saves at least 35.21 and 62.52 percent compared to *Fed_Guo* and *GSS_Zhu* approaches, respectively.

Effect of Varying the Numbers of Nodes. Figs. 8b and 8c report the average power consumption for the harmonic and arbitrary deadline tasks, respectively. We observe that the SingleSpeed_Greedy approach outperforms other approaches when the number of nodes (in each DAG) are high. Under harmonic task periods, the SingleSpeed_Greedy incurs 44.84 and 67.55 percent less power on average compared to



(a) Under Varying Utilization (b) Under Varying Number of Nodes (Harmonic Period) (c) Under Varying Number of Nodes (Arbitrary Period)

Fig. 10. Power consumption comparison between different approaches for the *implicit* deadline tasks considering a *discrete* frequency scheme on the *uniform* heterogeneous platform.

Fed_Guo and GSS_Zhu; under arbitrary task periods, the savings potential are 42.33 and 67.19 percent, respectively.

6.2 Uniform Heterogeneous Platform With a Discrete Frequency Scheme

In this section, we report the power consumption comparison for the (previously mentioned) approaches considering the uniform heterogeneous platform and a discrete frequency scheme. Under such a platform, we discretize the frequency using the technique described in Section 4.2.

6.2.1 Constrained Deadline Task

Here, we consider the constrained deadline tasks and report their average power consumption by changing two parameters: task period (or utilization) and the number of nodes.

Effect of Varying Task Periods (Utilization). Similar to the Figs. 7a, and 8a, we observe that the (i) average energy consumption is directly proportional to the average task utilization. (ii) *SingleSpeed_Greedy* approach consumes less power than other approaches (see Fig. 9a).

Effect of Varying the Numbers of Nodes. We vary the number of nodes (10 to 55) and report the average power consumption for harmonic (arbitrary) deadline tasks in Fig. 9b (Fig. 9c). Similar to the Figs. 7 and 8, we observe that the (i) Performance of *SingleSpeed_Random*, *SingleSpeed_Greedy*, *MaxSpeed_Greedy*, and *MaxSpeed_Random* does not vary that much for a small number of nodes (typically 10 to 25) per DAG. (ii) *SingleSpeed_Greedy* approach performs better (i.e., consume less power) than other approaches when the number of nodes per DAG is high.

6.2.2 Implicit Deadline Task

Now, we report the average power consumption using the same setup as described in Section 6.2.1, i.e., (a) for a fixed number of nodes (30) per task, change their utilization value, and (b) vary the number of nodes (10 to 55) per task, while keeping T_i fixed. We report the average power consumption in Fig. 10. From this figure, we observe that the (i) Performance of *SingleSpeed_Random*, *SingleSpeed_Greedy*, *MaxSpeed_Greedy*, and *MaxSpeed_Random* does not vary that much for a smaller task utilization or when the number of nodes per DAG is small (typically 10 to 35). (ii) *SingleSpeed_Greedy* approach performs better (i.e., consume less power) compared to the other approaches when the number of nodes per DAG is high.

7 Related Work

Much work has been done aimed at energy-efficient scheduling of sequential tasks in a homogeneous multi-core platform (see [11] for a survey). Considering the mixed-criticality task model and varying-speed processors, the works on [23], [35], [36], [37], [38] proposed an approach to handle the energy minimization problem. The work in [27], [39], [40], [41], [42], [43] presented an energy-efficient approach for the heterogeneous platform. Considering the real-time tasks in clustered heterogeneous platforms, the work in [39] studied the partitioned EDF scheduling policy, while [42] proposed an optimal task-core mapping technique that is fully-migrative. Considering the heterogeneous multi-core platform, a two-phase algorithm was proposed by [27]. In the first phase, they proposed a tasks-core allocation approach with the aim of reducing the dynamic energy consumption, while the second phase seeks for a better sleep state to reduce the leakage power consumption. A low overhead, DVFS-cum-DPM enabled energy-aware approach, HEALERS, was proposed by [43]. However, none of them considered the intra-task parallelism. Considering a clustered heterogeneous MPSoC platform, a migrative cluster scheduling approach was proposed by [15]. In this approach, run-time migration (within different cores in the same cluster) for a task is allowed to improve resource utilization. The work in [44] studied the technique to utilize the parallelism in a hard real-time streaming application (represented as a Synchronous Data Flow (SDF) graph) in a clustered heterogeneous platform.

Till date, considering both the intra-task parallelization and power minimization has received less attention. A greedy slack stealing algorithm is proposed in [17] that deals with task represented by graphs but did not consider the periodic DAGs. Assuming per-core DVFS, [33] provided the technique to combine DVFS and DPM. Considering the real-time jobs (represented as a DAG) in cloud computing systems and in a heterogeneous multi-core platform, the work in [45], [46] studied a QoS-aware and energy-efficient scheduling strategy. They proposed a scheduling policy that utilizes per-core DVFS. With the aim of improving energy-efficiency in a heterogeneous real-time platform, [47] proposed a combined approach considering the approximate computation and bin packing strategy. [48] investigated the energy awareness for cores that are grouped into blocks, and each block shares the same power supply scaled by DVFS. Benefits of (in terms of power saving) intra-task parallelism is proven theoretically in [1]. Considering the fork-join model, [49] reported an empirical evaluation of the power savings in a real test-bed. Based on level-packing, [50] proposed an energy efficient algorithm for implicit deadline tasks with same arrival time and deadline.

None of these works allows intra-task processor sharing considering the sporadic DAG task model. The recent work in [12], [18] is most related to ours. However, these works are significantly different from ours w.r.t the task model, platform, real-time constraints (deadlines), solution techniques, and the evaluation. *First*, the work in [12] considered a simplified model where only one DAG task executes at a time, while the work in [18] extends this work by allowing inter-task processor sharing. However, both of these works assumed that the number of cores are unlimited. *Second*, Both the works in [12], [18] assumed per-core speed scaling. However, many of the existing platforms (e.g., ODROID XU-3) do not support such speed scaling—speeds of processors under the same cluster must execute at the same speed. As the number of cores fabricated on a chip increases, per-core speed scaling design is less likely to be supported due to the inefficiency on hardware levels [13]. *Third*, Both of these works have studied only the implicit deadline tasks and did not consider the *constrained* deadline tasks. Hence, the non-negligible idle gaps between the task deadline and its next release remain un-utilized. *Finally*, the evaluations in [12], [18] were done based on simulations without any implementation on a real platform.

8 Conclusion

In this paper, we have studied real-time scheduling of a set of implicit and constrained deadline sporadic DAG tasks. We schedule these tasks on the cluster-based multi-core platforms with the goal of minimizing the CPU power consumption. In a clustered multi-core platform, the cores within the same cluster run at the same speed at any given time. Such design better balances energy efficiency and hardware cost and appears in many systems. However, from the resource management point of view, this additional restriction leads to new challenges. By leveraging a new concept, i.e., *speed-profile*, which models energy consumption variations during run-time, we can conduct scheduling and task-to-cluster partitioning while minimizing the expected overall long-term CPU energy consumption. To our knowledge, this is the first work that has investigated energy-efficient scheduling of DAGs on clustered multi-core platform. Also, no work considered energy-aware real-time scheduling of constrained deadline DAG tasks.

We have implemented our result on an ODROID XU-3 board to demonstrate its feasibility and practicality. We have also complemented our system experiments on a larger scale through realistic simulations that demonstrate an energy saving of up to 57 percent through our proposed approach compared to existing methods. In this work, we have restricted our attention mainly to the CPU power consumption. In the future, we plan to consider other components that may affect the total power consumption, e.g., cache misses, context switches, I/O usage, etc. We also plan to study the effect of tasks sporadic release patterns (to the overall power consumption) and propose a task reallocation scheme.

Acknowledgments

This work was partially supported by the National Science Foundation CNS-1850851, CNS-1742985 and National Natural Science Foundation of China 61801418.

References

[1] A. Paolillo, J. Goossens, P. M. Hettiarachchi, and N. Fisher, “Power minimization for parallel real-time systems with malleable jobs and homogeneous frequencies,” in *Proc. IEEE 20th Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, 2014, pp. 1–10.

[2] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill, “Parallel real-time scheduling of DAGs,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 12, pp. 3242–3252, Dec. 2014.

[3] M. Qamhieh, F. Fauberteau, L. George, and S. Midonnet, “Global EDF scheduling of directed acyclic graphs on multiprocessor systems,” in *Proc. 21st Int. Conf. Real-Time Netw. Syst.*, 2013, pp. 287–296.

[4] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, “A generalized parallel task model for recurrent real-time processes,” in *Proc. IEEE 33rd Real-Time Syst. Symp.*, 2012, pp. 63–72.

[5] J. Li, K. Agrawal, C. Lu, and C. Gill, “Analysis of global EDF for parallel tasks,” in *Proc. 25th Euromicro Conf. Real-Time Syst.*, 2013, pp. 3–13.

[6] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese, “Feasibility analysis in the sporadic DAG task model,” in *Proc. 25th Euromicro Conf. Real-Time Syst.*, 2013, pp. 225–233.

[7] J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah, “Analysis of federated and global scheduling for parallel real-time tasks,” in *Proc. 26th Euromicro Conf. Real-Time Syst.*, 2014, pp. 85–96.

[8] S. Baruah, V. Bonifaci, and A. Marchetti-Spaccamela, “The global EDF scheduling of systems of conditional sporadic DAG tasks,” in *Proc. 27th Euromicro Conf. Real-Time Syst.*, 2015, pp. 222–231.

[9] T. Hagras and J. Janecek, “A high performance, low complexity algorithm for compile-time job scheduling in homogeneous computing environments,” in *Proc. Int. Conf. Parallel Process. Workshops*, 2003, pp. 149–155.

[10] H. Aydin and Q. Yang, “Energy-aware partitioning for multiprocessor real-time systems,” in *Proc. Int. Parallel Distrib. Process. Symp.*, 2003, p. 9

[11] M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo, “Energy-aware scheduling for real-time systems: A survey,” *ACM Trans. Embedded Comput. Syst.*, vol. 15, no. 1, 2016, Art. no. 7.

[12] Z. Guo, A. Bhuiyan, A. Saifullah, N. Guan, and H. Xiong, “Energy-efficient multi-core scheduling for real-time DAG tasks,” in *Proc. 29th Euromicro Conf. Real-Time Syst.*, 2017, pp. 22:1–22:21.

[13] S. Herbert and D. Marculescu, “Analysis of dynamic voltage/frequency scaling in chip-multiprocessors,” in *Proc. Int. Symp. Low Power Electron. Des.*, 2007, pp. 38–43.

[14] 2017. [Online]. Available: http://www.hardkernel.com/

[15] D. Liu, J. Spasic, G. Chen, and T. Stefanov, “Energy-efficient mapping of real-time streaming applications on cluster heterogeneous MPSoCs,” in *Proc. 13th IEEE Symp. Embedded Syst. Real-Time Multimedia*, 2015, pp. 1–10.

[16] J. Kim, H. Kim, K. Lakshmanan, and R. R. Rajkumar, “Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car,” in *Proc. ACM/IEEE Int. Conf. Cyber-Physical Syst.*, 2013, pp. 31–40.

[17] D. Zhu, D. Mosse, and R. Melhem, “Power-aware scheduling for AND/OR graphs in real-time systems,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 9, pp. 849–864, Sep. 2004.

[18] A. Bhuiyan, Z. Guo, A. Saifullah, N. Guan, and H. Xiong, “Energy-efficient real-time scheduling of DAG tasks,” *ACM Trans. Embedded Comput. Syst.*, vol. 17, no. 5, 2018, Art. no. 84.

[19] W. M. Kolb, "Curve fitting for programmable calculators," IMTEC, 1984.

[20] S. Pagani and J.-J. Chen, "Energy efficient task partitioning based on the single frequency approximation scheme," in *Proc. IEEE 34th Real-Time Syst. Symp.*, 2013, pp. 308–318.

[21] S. Pagani and J.-J. Chen, "Energy efficiency analysis for the single frequency approximation (SFA) scheme," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 5s, 2014, Art. no. 158.

[22] P. Huang, P. Kumar, G. Giannopoulou, and L. Thiele, "Energy efficient DVFS scheduling for mixed-criticality systems," in *Proc. Int. Conf. Embedded Softw.*, 2014, pp. 1–10.

[23] S. Narayana, P. Huang, G. Giannopoulou, L. Thiele, and R. V. Prasad, "Exploring energy saving for mixed-criticality systems on multicores," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2016, pp. 1–12.

[24] J. Howard *et al.*, "A 48-core IA-32 processor in 45 nm CMOS using on-die message-passing and DVFS for performance and power scaling," *IEEE J. Solid-State Circuits*, vol. 46, no. 1, pp. 173–183, Jan. 2011.

[25] D. Maxim and L. Cucu-Grosjean, "Response time analysis for fixed-priority tasks with multiple probabilistic parameters," in *Proc. IEEE 34th Real-Time Syst. Symp.*, 2013, pp. 224–235.

[26] S. H. Funk, *EDF Scheduling on Heterogeneous Multiprocessors*. Chapel Hill, NC, USA: Univ. of North Carolina, 2004.

[27] M. A. Awan, D. Masson, and E. Tovar, "Energy efficient mapping of mixed criticality applications on unrelated heterogeneous multicore platforms," in *Proc. 11th IEEE Symp. Ind. Embedded Syst.*, 2016, pp. 1–10.

[28] 2017. [Online]. Available: https://github.com/tuxamito/emoxu3

[29] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner, "Random graph generation for scheduling simulations," in *Proc. 3rd Int. ICST Conf. Simul. Tools Techn.*, 2010, Art. no. 60.

[30] 2017. [Online]. Available: http://en.wikipedia.org/wiki/Gamma distribution

[31] 2017. [Online]. Available: https://github.com/scheduler-tools/rt-app/

[32] J.-J. Chen and C.-F. Kuo, "Energy-efficient scheduling for real-time systems on dynamic voltage scaling (DVS) platforms," in *Proc. 13th IEEE Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, 2007, pp. 28–38.

[33] G. Chen, K. Huang, and A. Knoll, "Energy optimization for real-time multiprocessor system-on-chip with optimal DVFS and DPM combination," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 3s, 2014, Art. no. 111.

[34] Z. Guo, A. Bhuiyan, D. Liu, A. Khan, A. Saifullah, and N. Guan, "Energy-efficient real-time scheduling of DAGs on clustered multi-core platforms," in *Procc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2019, pp. 156–168.

[35] S. Baruah and Z. Guo, "Mixed-criticality scheduling upon varying-speed processors," in *Proc. IEEE 34th Real-Time Syst. Symp.*, 2013, pp. 68–77.

[36] S. Baruah and Z. Guo, "Scheduling mixed-criticality implicit-deadline sporadic task systems upon a varying-speed processor," in *Proc. IEEE Real-Time Syst. Symp.*, 2014, pp. 31–40.

[37] Z. Guo and S. Baruah, "The concurrent consideration of uncertainty in WCETs and processor speeds in mixed-criticality systems," in *Proc. 23rd Int. Conf. Real-Time Netw. Syst.*, 2015, pp. 247–256.

[38] A. Bhuiyan, S. Sruti, Z. Guo, and K. Yang, "Precise scheduling of mixed-criticality tasks by varying processor speed," in *Proc. 27th Int. Conf. Real-Time Netw. Syst.*, 2019, pp. 123–132.

[39] A. Colin, A. Kandhalu, and R. Rajkumar, "Energy-efficient allocation of real-time applications onto heterogeneous processors," in *Proc. IEEE 20th Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, 2014, pp. 1–10.

[40] J.-J. Chen, A. Schranzhofer, and L. Thiele, "Energy minimization for periodic real-time tasks on heterogeneous processing units," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2009, pp. 1–12.

[41] C. Liu, J. Li, W. Huang, J. Rubio, E. Speight, and X. Lin, "Power-efficient time-sensitive mapping in heterogeneous systems," in *Proc. 21st Int. Conf. Parallel Archit. Compilation Techn.*, 2012, pp. 23–32.

[42] H. S. Chwa, J. Seo, J. Lee, and I. Shin, "Optimal real-time scheduling on two-type heterogeneous multicore platforms," in *IEEE Real-Time Syst. Symp.*, 2015, pp. 119–129.

[43] S. Moulik, R. Devaraj, and A. Sarkar, "HEALERS: A heterogeneous energy-aware low-overhead real-time scheduler," *IET Comput. Digit. Techn.*, vol. 13, no. 6, pp. 470–480, Nov. 2019.

[44] J. Spasic, D. Liu, and T. Stefanov, "Energy-efficient mapping of real-time applications on heterogeneous MPSoCs using task replication," in *Proc. Int. Conf. Hardware/Softw. Codes. Syst. Synthesis*, 2016, pp. 1–10.

[45] G. L. Stavrinides and H. D. Karatza, "Energy-aware scheduling of real-time workflow applications in clouds utilizing DVFS and approximate computations," in *Proc. IEEE 6th Int. Conf. Future Internet Things Cloud*, 2018, pp. 33–40.

[46] G. L. Stavrinides and H. D. Karatza, "An energy-efficient, QoS-aware and cost-effective scheduling approach for real-time workflow applications in cloud computing systems utilizing DVFS and approximate computations," *Future Gener. Comput. Syst.*, vol. 96, pp. 216–226, 2019.

[47] G. L. Stavrinides and H. D. Karatza, "Scheduling real-time DAGs in heterogeneous clusters by combining imprecise computations and bin packing techniques for the exploitation of schedule holes," *Future Gener. Comput. Syst.*, vol. 28, no. 7, pp. 977–988, 2012.

[48] X. Qi and D.-K. Zhu, "Energy efficient block-partitioned multicore processors for parallel applications," *J. Comput. Sci. Technol.*, vol. 26, no. 3, 2011, Art. no. 418.

[49] A. Paolillo, P. Rodriguez, N. Veshchikov, J. Goossens, and B. Rodriguez, "Quantifying energy consumption for practical fork-join parallelism on an embedded real-time operating system," in *Proc. 24th Int. Conf. Real-Time Netw. Syst.*, 2016, pp. 329–338.

[50] H. Xu, F. Kong, and Q. Deng, "Energy minimizing for parallel real-time tasks based on level-packing," in *Proc. IEEE Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, 2012, pp. 98–103.

[51] A. Dunkels, F. Osterlind, N. Tsiftes, and Z. He, "Software-based on-line energy estimation for sensor nodes," in *Proc. 4th Workshop Embedded Netw. Sensors*, 2007, pp. 28–32.

[52] H.-Y. Zhou, D.-Y. Luo, Y. Gao, and D.-C. Zuo, "Modeling of node energy consumption for wireless sensor networks," *Wireless Sensor Netw.*, vol. 3, no. 01, 2011, Art. no. 18.

Ashikahmed Bhuiyan received the bachelor of science degree in computer science and engineering from the Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh, in 2013. He is currently working toward the PhD degree in the Department of Electrical and Computer Engineering, University of Central Florida (UCF), Orlando, Florida, under the supervision of Zhishan Guo and Abusayeed Saifullah (from the Wayne State University). He is a member of the Real-Time & Intelligent Systems Lab, UCF. His research focuses on improving energy efficiency in real-time embedded systems, parallel computing, and mixed-criticality scheduling. He has received the Best Student Paper Award at the 40th IEEE Real-Time Systems Symposium (RTSS 2019).

Di Liu received the BEng and MEng degrees from Northwestern Polytechnical University, Xi'an, China, in 2007 and 2011, respectively, and the PhD degree from Leiden University, Leiden, The Netherlands, in 2017. He is currently an assistant professor with the School of Software, Yunnan University, China. His research interests include the fields of real-time systems, energy-efficient multicore/many core systems, and cyber-physical systems.

Aamir Khan received the MS degree from the Computer Engineering Department, Missouri University of Science and Technology, Rolla, Missouri, in 2018. He is currently an embedded systems engineer with BrainCo Tech in Boston. His areas of interest includes real-time systems and body-machine interface products.

Abusayeed Saifullah received the PhD degree in computer science and engineering with Turner Dissertation Award from Washington University in St Louis, St. Louis, Missouri, in 2014. He is an assistant professor of the Computer Science Department, Wayne State University. His research primarily concerns Internet-of-Things, cyber-physical systems, real-time systems, embedded systems, and low-power wide-area networks. He received seven Best Paper Awards/Nominations in highly competitive conferences including ACM SenSys (2016 nomination), IEEE RTSS (2019, 2014, 2011), IEEE ICII (2018), and IEEE RTAS (2012 nomination). He also received multiple young investigator awards including the CAREER award (2019) and the CRII award (2016) of the National Science Foundation (NSF). He is serving as the program chair of IEEE ICESS 2020, served as a track chair of IEEE ICCCN 2019 and as a program committee member for various conferences including ACM SenSys, IEEE RTSS, ACM/IEEE IoTDI, IEEE RTAS, ACM/IEEE ICCPS, ACM MobiHoc, IEEE INFOCOM, EWSN, and ACM IWQoS. He also served as a guest editor of the *IEEE Transactions on Industrial Informatics*, and is currently an editor of the *Elsevier Pervasive and Mobile Computing Journal*.

Nan Guan received the PhD degree from Uppsala University, Uppsala, Sweden. He is currently an assistant professor with the Department of Computing, Hong Kong Polytechnic University. He worked with Northeastern University, China before joining The Hong Kong Polytechnic University. His research interests include the design and analysis of real-time systems, embedded systems, cyber-physical systems, and Internet-of-Things (IoT) systems. He received the EDAA Outstanding Dissertation Award in 2014, the CCF Outstanding Dissertation Award in 2013, the Best Paper Award of IEEE RTSS in 2009, the Best Paper Award of DATE in 2013, the Best Paper Award of ACM e-Energy 2018 and the Best Paper Award of IEEE ISORC 2019. He served as the TPC co-chair of EMSOFT 2015, ICESS 2017, SETTA 2019, the TPC track chair of RTAS 2018.

Zhishan Guo received the BE (with honor) degree in computer science and technology from Tsinghua University, Beijing, China, in 2009, the MPhil degree in mechanical and automation engineering from the Chinese University of Hong Kong, Hong Kong, in 2011, and the PhD degree in computer science from the University of North Carolina at Chapel Hill, Chapel Hill, North Carolina, in 2016. He is an assistant professor with the Department of Electrical and Computer Engineering, University of Central Florida. His research and teaching interests include real-time scheduling, cyber-physical systems, and neural networks and their applications.

SaberLDA: Sparsity-Aware Learning of Topic Models on GPUs

Kaiwei Li, Jianfei Chen, Wenguang Chen, and Jun Zhu

Abstract—Latent Dirichlet Allocation (LDA) is a popular tool for analyzing discrete count data such as text and images, which are required to model datasets and a large number of topics, e.g., tens of thousands of topics for industry scale applications. Although distributed CPU systems have been used to address this problem, they are slow and resource inefficient. GPU-based systems have emerged as a promising alternative because of their high computational power and memory bandwidth. However, existing GPU-based LDA systems can only learn thousands of topics, because they use dense data structures, and have linear time complexity to the number of topics. In this article, we propose SaberLDA, a GPU-based LDA system that implements a sparsity-aware algorithm to achieve sublinear time complexity to learn a large number of topics. To address the challenges introduced by sparsity, we propose a novel data layout, a warp-based sampling kernel, an efficient sparse matrix counting method, and a fine-grained load balancing strategy. SaberLDA achieves linear speedup on 4 GPUs and is 6–10 times faster than existing GPU systems in thousands of topics. It can learn 40,000 topics from a dataset of billions of tokens in two hours, which was previously only achievable using clusters of tens of CPU servers.

Index Terms—GPU acceleration, latent dirichlet allocation, topic models, machine learning

1 Introduction

Big data, such as web pages, user activities and images, are pervasive nowadays. Machine learning helps extract underlying information from the data and make predictions.

Among various machine learning algorithms, Probabilistic graphical models (PGMs) is a class of popular unsupervised machine learning algorithms. PGMs provide a flexible way of defining models that incorporate human knowledge and have been used extensively in various scientific and engineering domains (see [13] for an overview).

In this paper, we focus on topic modeling, an important subclass of PGMs, and demonstrate the challenges and solutions encountered to accelerate PGMs with GPUs. Topic models provide a suite of widely adopted statistical tools for feature extraction and dimensionality reduction for bag-of-words (i.e., discrete count) data, such as text documents and images in a bag-of-words format [7]. Given an input corpus, topic models automatically extract a number of latent topics, which are in unigram distributions over the words in a given vocabulary. The high-probability words in each topic are semantically correlated. Latent Dirichlet Allocation (LDA) [4] is the most popular topic model due to its simplicity. It has been deployed as a key component in data visualization [16], text analysis [5], [42], computer vision [7], network analysis [8], [11], and recommendation systems [12].

In practice, it is not uncommon to encounter large-scale datasets, e.g., text analysis typically consists of hundreds of millions of documents [36], and recommendation systems need to tackle hundreds of millions of users [1]. Furthermore, as the scale of datasets increases, model sizes need to increase as well because we need a larger number of topics to exploit the richer semantic structure underlying the data. Having ten thousands of topics is a reasonable design goal for modern topic modeling systems, to have good coverage for both industry-scale applications [32] and researching [5], [7], [16].

However, it is highly challenging to efficiently train large LDA models. The time complexity of training LDA is high because it involves iteratively scanning the input corpus many times (e.g., 100), and the time complexity of processing each token is not constant but is related to the number of topics.

To train LDA in acceptable time, CPU clusters are often used. However, due to the limited memory bandwidth and low computational power of CPUs, large clusters are typically required to learn large topic models [1], [35], [36]. For example, a 32-machine cluster is used to learn 1,000 topics from a 1.5-billion-token corpus.

A promising alternative is to train LDA with GPUs, leveraging their high computational power and memory bandwidth. There have been several previous attempts using this approach. For example, Yan *et al.* [33] implemented the collapsed Gibbs sampling algorithm, BIDMach [41] implemented the variational Bayes algorithm as well as a modified Gibbs sampling algorithm, and Tristan *et al.* [28] proposed an expectation-maximization algorithm. These GPU-based systems are reportedly to achieved superior performance compared to CPU-based systems [28], [33], [41].

Unfortunately, current GPU-based LDA systems can only learn a few hundred topics (See Table 1), which may not be

- *K. Li, J. Chen, and J. Zhu are with the Department of Computer Sciences and Technology, Tsinghua University, Beijing 100084, China. E-mail: {likw17, chenjian14}@mails.tsinghua.edu.cn, dcszj@tsinghua.edu.cn.*
- *W. Chen is with the Department of Computer Sciences and Technology, Tsinghua University, Beijing 100084, China, and also with Beijing National Research Center for Information Science and Technology (BNRist). E-mail: cwg@tsinghua.edu.cn.*

Manuscript received 8 Aug. 2019; revised 20 Feb. 2020; accepted 28 Feb. 2020. Date of publication 8 Apr. 2020; date of current version 24 Apr. 2020.
(Corresponding authors: Wenguang Chen and Jun Zhu.)
Recommended for acceptance by S. He.
Digital Object Identifier no. 10.1109/TPDS.2020.2979702

TABLE 1
Summary of LDA Systems Where D is the Number of Documents, K is the Number of Topics, V is the Size of Vocabulary, and T is the Number of Tokens

Implementation	D	K	V	T
Yan *et al.* [33]	300k	128	100k	100M
BIDMach [41]	300k	256	100k	100M
Steele and Tristan [28]	50k	20	40k	3M
AGA-LDA [22]	300k	128	100k	100M

sufficient to capture the rich semantic structure underlying the large datasets in industry-scale applications [31]. It is fundamentally difficult for these systems to learn more topics because they use algorithms on dense data structures whose time and space complexity is linear to the number of topics.

To address this problem, we propose SaberLDA, a novel GPU-based system that adopts a state of the art sparsity-aware algorithm for LDA. Sparsity-aware algorithms are based on the insight that a single document is not likely to have many topics. A Sparsity-aware algorithm has the same model accuracy as the algorithm without sparsity optimization, and it is able to achieve sub-linear (or even amortized constant) time complexity with respect to the number of topics. Representative examples include AliasLDA [17], F+LDA [35], LightLDA [36], WarpLDA [9], and ESCA [39], which are implemented in general-purpose CPU systems.

However, it is considerably more challenging to design and implement sparsity-aware algorithms on GPUs than on CPUs.

- Compared with CPUs, GPUs have considerably higher thread counts, smaller per thread cache size, and larger cache line size, which makes it very difficult to use caches to mitigate the random access problems introduced by sparsity.
- The branch divergence issues of GPUs suggest that we should fully vectorize the code. However, for sparse data structures, loop length is no longer fixed and the data are not aligned, indicating that straightforward vectorization is not feasible.
- The limited GPU memory capacity requires streaming input data and model data. This adds another dimension of complexity for data partition and layout to simultaneously enable parallelism, good locality and efficient sparse matrix updates.
- Finally, the calculation based on the skew vocabulary requires fine-grained parallelism in multi-GPU with hundreds of streaming multiprocessors, or it would be imbalanced because the workload of the largest word exceeds the average workload of all GPU blocks.

SaberLDA addresses all these challenges by supporting sparsity-aware algorithms on multi-GPUs. Our key technical contributions are as follows:

- A novel hybrid data layout named *partition-by-document and order-by-word* (PDOW) that simultaneously maximizes the locality and reduces the GPU memory consumption;
- A warp-based sampling kernel that is fully vectorized, and is equipped with a W-ary sampling tree that supports both efficient construction and sampling;
- A multi-GPU based *shuffle and segmented count* (SSC) algorithm for updating sparse count matrices; and
- A *split index* optimization based on the Compressed Row (CSR) format representing a sparse matrix of vocabulary, which solves the imbalance of multiple GPU workloads.

Our experimental results demonstrate that SaberLDA is able to train LDA models with up to 40,000 topics, which is more than an order of magnitude larger than previous GPU-based systems [28], [33], [41]. SaberLDA is also highly efficient; using a single GPU, SaberLDA converges five times faster than previous GPU-based systems and four times faster than CPU-based systems. With 4 GPUs, SaberLDA is able to learn 40,000 topics from a dataset of billions of tokens, which was previously only achievable using clusters of tens of machines [35].

The rest the paper is organized as follows. Section 2 introduces the basics of LDA. Section 3 presents the design of SaberLDA. Section 4 contains the experiments. Section 5 discusses related work. Section 6 discusses the limitations of this study and future work. In Section 7 we summarize the paper.

2 Latent Dirichlet Allocation

In this section, we introduce the Latent Dirichlet Allocation (LDA) model, its sampling algorithm, and the sparsity-aware optimization.

2.1 Definition

LDA is a statistical model that infers topics from a given corpus. A corpus consists of multiple documents. Each document is represented as a bag of words, disregarding grammar and even word order, but keeping multiplicity. Each occurrence of a word in such document is called a *token*. The set of all words forms the vocabulary. The definitions of the corpus are:

- D: The number of documents in a corpus;
- V: The size of the vocabulary; and
- T: The total number of tokens among all documents.

We also define the number of topics to infer, K, which is given by the user.

- K: The number of topics to infer from the corpus.

The text corpus is represented as a token list $\mathbf{T} : \text{List}[(d, v, k)]$. The terms d, v, and k refer to the document-id, word-id, and topic-id, respectively. Training LDA involves assigning a *topic assignment* k for each token (d, v, k). Note that duplicated words occuring in the same document are treated as multiple tokens since they can be assigned to different topics.

According to the topics from each token in $\mathbf{T}$, we can calculate the topic count for a document or a word. Hence, we define the document-topic count matrix $\mathbf{A}$ and word-topic count matrix $\mathbf{B}$, as well as the global topic distribution vector $\mathbf{C}$:

- $\mathbf{A}$: $D \times K$ sparse matrix, A_{dk} is the number of occurrences of topic k in document d;
- $\mathbf{B}$: $V \times K$ sparse matrix, but usually more dense than $\mathbf{A}$. B_{vk} is the count of those tokens with word v which has been assigned to topic k; and

Token List T

	Word	Topic
Doc1	iOS	3
	Android	3
Doc 2	apple	2
	apple	1
	iPhone	1
	iOS	3
Doc 3	apple	2
	orange	2

Doc-Topic Matrix A

	1	2	3
Doc 1			2
Doc 2	2	1	1
Doc 3		2	

Word-Topic Matrix B

	1	2	3
iOS			2
Android			1
apple	1	2	
iPhone	1		
orange		1	

Fig. 1. An example of token list and count matrices.

- **C**: a dense vector of K elements, where C_k is the count of all tokens with topic k.

Fig. 1 is an example of token list and count matrices.

2.2 Inference

Given the token list **T**, our goal is to infer the topic assignment k for each token $t = (d, v, k)$. Informally, the LDA model is designed in a way that maximizes some objective function (the likelihood) related to the topic assignments of each token. We omit the mathematical details and refer the interested readers to standard LDA literature [4], [15].

There are many inference algorithms, such as variational inference [4], Markov chain MonteCarlo [15], and expectation maximization [9], [28]. We choose the state-of-the-art algorithm [28] to implement on GPU for its following advantages:

- It is sparsity-aware, so the time complexity is sublinear with respect to the number of topics. This property is critical to support the efficient training of large models;
- It enjoys the best degree of parallelism because the count matrices **A** and **B** only need to be updated once per iteration. This matches with the massively parallel nature of GPU to achieve high performance.

The pseudo code of ESCA is shown in Algorithm 1, which is a bulk synchronous parallel (BSP) programming model. In each iteration, the topics of all tokens are sampled independently. Initially, all tokens are assigned randomly. The inference method has multiple iterations. Each iteration first calculates **A**, **B**, and **C** by their definitions and then samples the topic k of each token based on:

$$P_k = \frac{(A_{dk} + \alpha)(B_{vk} + \beta)}{C_k + V\beta}. \tag{1}$$

Here, **P** is the measure of topic distribution of token (d, v) where P_k is proportional to the probability of choosing topic k. The token (d, v) has larger probability in topic k when its document d has more tokens in topic k (larger A_{dk}), and the word v in all documents are more assigned by topic k (larger B_{vk}). α and β are two user specified parameters that control the granularity of topics. Large α and β values mean that we want to discover a few general topics, while small α and β values mean that we want to discover many specific topics.

2.3 Sampling

We now introduce the sampling function at line 12 in Algorithm 1. The input distribution array **P** consists of K floating point numbers. The output is a topic k, where k is an integer between 1 and K. Algorithm 2 is a straightforward method to generate the output topic from a given array of probabilities, which takes the following steps. Compute the the prefix-sum of **P** and set the result to the variable **S**. Random a float number m from the uniform distribution of 0 to S_K, where S_K is the summation of **P**. Use binary search to find the sampled topic k where where $S_{k-1} < m \leq S_k$.

Algorithm 1. ESCA Algorithm for LDA

```
1: Input: token list T
2: for $(d, v, k) \in \mathbf{T}$ do
3:     $k \leftarrow \textsc{Random}(1, K)$
4: end for
5: for $i \leftarrow 1$ to num_iteration do
6:     $\mathbf{A} \leftarrow \mathbf{Int}[D][K], \mathbf{B} \leftarrow \mathbf{Int}[V][K], \mathbf{C} \leftarrow \mathbf{Int}[K]$
7:     for $(d, v, k) \in \mathbf{T}$ do
8:         $\textsc{Increase}(A_{dk}, B_{vk}, C_k)$
9:     end for
10:    for $(d, v, k) \in \mathbf{T}$ do
11:        $\mathbf{P} \leftarrow (\mathbf{A}_d + \alpha) \times (\mathbf{B}_v + \beta)/(\mathbf{C} + V \times \beta)$
12:        $k \leftarrow \textsc{Sample}(\mathbf{P})$
13:    end for
14: end for
```

Algorithm 2. Sample From Topic Distribution Array

```
1: Input: $K$: Int, P: float[$K$] //Starting from index 1
2: $\mathbf{S} \leftarrow \textsc{PrefixSum}(\mathbf{P})$ where $S_0 = 0, S_k = \sum_{i=1}^{k} P_i$
3: Random $m \in (0, S_K]$
4: Binary search $k$ where $S_{k-1} < m \leq S_k$
5: return $k$
```

2.4 Sparsity-Aware Optimization

The time complexity of constructing and sampling of **P** is $O(K)$. However, the complexity is not sparsity-aware, which means that it is proportional to the number of topics. For example, for a given corpus, inferring 1000 topics will be 10 times slower than inferring 100 topics. The sparsity-aware optimizations [1], [28] utilize the sparsity of **A**, and improve the time complexity to $O(K_D)$, where $O(K_D) \ll O(K)$. K_D is the average number of non-zero entries per row of **A**.

According to Eq. (1), for a given token with document d and word v, its topic distribution is $\mathbf{P}_{dv}$. The sparsity-aware optimization decomposes the original sampling problem $\mathbf{P}_{dv}$ as two easier sampling subproblems $\mathbf{X}_{dv}$ and $\mathbf{Y}_v$. Sampling $\mathbf{P}_{dv}$ is exactly equal to sample the subproblem $\mathbf{X}_{dv}$ with the probability of $\frac{|\mathbf{X}_{dv}|}{|\mathbf{X}_{dv}|+|\mathbf{Y}_v|}$, and sampling $\mathbf{Y}_v$ with the probability of $\frac{|\mathbf{Y}_v|}{|\mathbf{X}_{dv}|+|\mathbf{Y}_v|}$, where $|\mathbf{V}|$ means the summation of the elements in vector **V**. The two sub-problems are derived by the following equations:

$$\mathbf{P}_{dv} = \frac{(\mathbf{A}_d + \alpha)(\mathbf{B}_v + \beta)}{\mathbf{C} + V\beta} = \mathbf{A}_d\mathbf{Q}_v + \alpha\mathbf{Q}_v = \mathbf{X}_{dv} + \mathbf{Y}_v \tag{2a}$$

$$\text{where:} \quad \mathbf{X}_{dv} = \mathbf{A}_d\mathbf{Q}_v, \mathbf{Y}_v = \alpha\mathbf{Q}_v, \mathbf{Q}_v = \frac{\mathbf{B}_v + \beta}{\mathbf{C} + V\beta} \tag{2b}$$

We now analysis the time complexity of constructing and sampling of these two subproblems $\mathbf{X}_{dv}$ and $\mathbf{Y}_v$.

$\mathbf{X}_{dv}$ has a factor $\mathbf{A}_d$, which is a sparse vector most of the time, so the sparsity of $\mathbf{X}_{dv}$ is equal to $\mathbf{A}_d$. Since $\mathbf{A}_d$ is the topic count of document d, the number of the non-zero element $\mathrm{nnz}(\mathbf{A}_d)$ is no more than the number of tokens of document d. In most corpora like NYTimes and PubMed, a document has dozens to hundreds of tokens. However, it is common to infer thousands to millions of topics, which is far more than the tokens in one document. This can be described by the following inequation:

$$K_D \approx \mathrm{nnz}(\mathbf{A}_d) \leq |\ \mathbf{T}_d| \approx 100 \ll K, \tag{3}$$

where the function *nnz* is the number of non-zero elements of a vector, $\mathbf{T}_d$ is the group of all tokens of document d, K_D is the average of $\mathrm{nnz}(\mathbf{A}_d)$ among all documents. Consequently, $\mathbf{X}_{dv}$ is a sparse vector with $O(K_D)$ elements, and the time complexity of construction and sampling of $\mathbf{X}_{dv}$ are both $O(K_D)$, which is sparsity-aware.

$\mathbf{Y}_v$ is the product of α and $\mathbf{Q}_v$ which depends on the topic count vector $\mathbf{B}_v$ and $\mathbf{C}$. Since many tokens belong to the same word v in the vocabulary, it is possible to reuse $\mathbf{Y}_v$ for those tokens. The construction of $\mathbf{Y}_v$ can be pre-processed before Before sampling tokens. Hence the time complexity is $O(K)$ for constructing each $\mathbf{Y}_v$, and $O(VK)$ for all unique words in the vocabulary.

At last, we look at sampling $\mathbf{Y}_v$. Since $\mathbf{Y}_v$ is a dense vector of K non-zero elements, the straightforward sampling Algorithm 2 is not sparsity-aware here. Previous works [17], [36] used AliasTable with $O(K)$ built time for each word v and $O(1)$ for sampling each token. Yu *et al.* [35] used a Fenwick tree, which was also built in $O(K)$ and can then sample a topic in $O(\log_2 K)$. For convenience, we respectively name the initialize and sampling functions PreBuilt() (line 3) and FastSample() (line 17).

Algorithm 3. Sparsity Aware Optimization

```
1:  for $i \leftarrow 1$ to num_iteration do
2:      $\mathbf{A} \leftarrow$ DocTopicCount($\mathbf{T}$)
3:      $\mathbf{B} \leftarrow$ WordTopicCount($\mathbf{T}$)
4:      $\mathbf{C} \leftarrow$ TopicCount($\mathbf{T}$)
5:      $\mathbf{Q}, \mathbf{Y} \leftarrow \mathbf{Float}[V][K]$
6:      for $v \leftarrow 1$ to $V$ do
7:          $\mathbf{Q}_v \leftarrow (\mathbf{B}_v + \beta)/(\mathbf{C} + V\beta)$ // $O(K)$
8:          $\mathbf{Y}_v \leftarrow \alpha \times \mathbf{Q}_v$
9:          $\mathrm{Tree}_v \leftarrow$ PreBuilt($\mathbf{Y}_v$)
10:     end for
11:     for $(d, v, k) \in \mathbf{T}$ do
12:         $\mathbf{X}_{dv} \leftarrow \mathbf{A}_d \times \mathbf{Q}_v$ // $O(K_D)$
13:         $R \leftarrow$ RandFloat($0, |\mathbf{X}_{dv}| + |\mathbf{Y}_v|$)
14:         if $R \leq |\mathbf{X}_{dv}|$ then
15:             $k \leftarrow$ SparseSample$(\mathbf{X})_{dv}$ // $O(K_D)$
16:         else
17:             $k \leftarrow$ FastSample($\mathrm{Tree}_v$) // $O(\log_2 K)$
18:         end if
19:     end for
20: end for
```

The sparsity-aware optimization is shown in Algorithm 3. At the beginning of each iteration, the time complexity of constructing count matrices **A**, **B** and **C** is $O(T)$. Before sampling tokens, constructing $\mathbf{Q}_v$, $\mathbf{Y}_v$ and additional data structure Tree $_v$ has $O(KV)$ time complexity. When sampling each token, the time complexity is $O(K_D + \log_2 K)$ for Fenwick tree and $O(K_D + 1)$ for AliasTable. Hence the the time complexity for sampling all tokens is $O(KV + TK_D)$ Consequently, the overall time complexity in each iteration is $O(KV + TK_D)$, which sparsity-aware for large topics and it is far less than the original $O(TK)$ algorithm.

3 Design and Implementation

In this section, we illustrate the design and implementation of SaberLDA. We re-iterate the design goals of SaberLDA here:

- Supporting industrial-grade large models on GPUs, e.g., tens of thousands topics;
- Supporting large dataset with billions of tokens; and
- Providing significant speedup with GPUs to CPU solutions.

To achieve these goals, SaberLDA adopts the state-of-the-art embarrassingly parallel LDA algorithm with sparsity-aware optimization (Algorithm 3) in multiple GPUs. In the rest part of this section, we organize the content by demonstrating how different segments of Algorithm 3 are designed and implemented in SaberLDA.

The first challenge we faced is to design the layout of large datasets and matrices on memory-limited multi-GPU architecture. Section 3.1 discusses the computing and memory impacts of different kinds of data layouts.

We next focus on the most time-consuming part of the sparsity-aware algorithm, which is the sampling of tokens (lines 11-19). We analyze the memory access localities of different for-looping orders in Section 3.2.

Section 3.3 discusses how to assign tokens into massive GPU cores with fine-grained parallelism. It also demonstrates the vectorized optimization used in element-wise product (line 12) and sampling methods (line 15 and 17).

Since it is challenging to achieve good load balance with skew vocabulary distributions especially in multi-GPU. Section 3.4 proposes a hybrid parallelism strategy named *split index* to further parallelize the most frequent words.

The matrices counting methods (line 2 and line 3) take lower order of complexity than sampling, however, it is still challenging to use GPU efficiently if compute them into sparse format. We named this problem *shuffle and segmented count* and propose a multi-GPU solution in Section 3.5.

3.1 Data Layouts

A modern GPU has more than 10 gigabytes memory on board. A typical industrial LDA model [32] learns 10k topics with 210k unique words in the vocabulary, which forms 8.4 gigabytes of a count matrix **B** storing 32-bit integers as its elements. Using the dense format for storing the model matrix **B** is suitable for most cases of LDA training.

The web-scale datasets for LDA training have no less than millions of documents, which makes the size of document-topic count matrix **A** tens to hundreds of times larger than word-topic count matrix **B**. To reduce the memory requirements and enable the sparsity-aware algorithm, SaberLDA uses a sparse layout CSR to store **A**. However, the size of **A** still has dozens to hundreds of gigabytes, hence it has to be partitioned before computing in GPUs.

Since the token list **T** and the document-topic count matrix **A** are too large to be kept in GPU memory, they are

TABLE 2
Data Layout Overview of SaberLDA for a Typical Short Video Dataset for Recommendation

	T	**A**$(D \times K)$	**B**$(V \times K)$
#	10B	10M × 10k	100k × 10k
Size	80 GB	Sparse 40 GB	Dense 4 GB
Partitioning	By d ~2 GB	By d ~2 GB	Suitable in GPUs
Grouping	By v	Replicated	By v

divided into small partitions. A partition contains the partition of **A** and **T** that belongs to the same document group. The size of each partition is some gigabytes, so it is able to be entirely transferred to GPU memory.

During the computation, partitions of **A** and **T** are successively transferred to GPUs. When the computation of a partition finishes, the data are transferred back to the CPU. The GPU memory space is recycled for the next partition.

In multiple GPUs, the computation of tokens **T** can be separated evenly in multiple GPU, because they are embarrassingly parallelized. However, computing token (d, v) with document-id d and word-id v requires reading the rows $\mathbf{A}_d$ and $\mathbf{B}_v$, which causes the matrices must be shared across multiple GPUs for reading. The shared data can either be *replicated* in all GPUs or be *remote* accessed through PCI-e bus or NVLink. Fortunatedly, if *grouping* the tokens of the same document (or word) into single GPU, visiting the data of count matrix **A** (or **B**) is localized.

It must be decided how to group the tokens. If grouping tokens with the same document in a GPU, we can either replicate the whole matrix **B** into all GPUs, or distribute **B** and remotely access the different parts. Unfortunatedly, reaplicating the large dense matrix **B** is not scalable and wasting memory, and remoting access data from other GPUs has very high latency. Grouping tokens with the same word is much more better. At this time, visiting matrix **B** is localized. Replicating the matrix **A** and the token list **T** into multiple GPUs take no significant overhead, because they are already spending time on CPU-GPU transferring.

Table 2 demonstrates the data layout of a typical short video dataset for recommendation. The first row is the num- ber of elements. The second row is the estimated size. The third row is the partition strategy and the size of each partition for CPU-GPU transferring, and the last row is the grouping strategy in multi-GPU.

3.2 Choosing Sampling Order

In this subsection, we introduce the design and optimization of token sampling in SaberLDA. The procedure of token sampling is found in lines 11-19 in Algorithm 3. The order of sampling tokens in the GPU is the looping order of tokens **T**. Theoretically, these tokens can be sampled in any order, but the ordering greatly impacts the locality.

As illustrated in Fig. 2a, sampling k for a token (d, v, k) requires evaluating an element-wise product of two rows (Line 12 of Algorithm 3), the dth row of the document-topic count matrix $\mathbf{A}_d$ and the vth row of the word-topic factor matrix $\mathbf{Q}_v$. The element-wise product involves accessing all non-zero entries of $\mathbf{A}_d$ (sequential), and accessing elements of $\mathbf{Q}_v$ indexed by the non-zero entries of $\mathbf{A}_d$ (random).

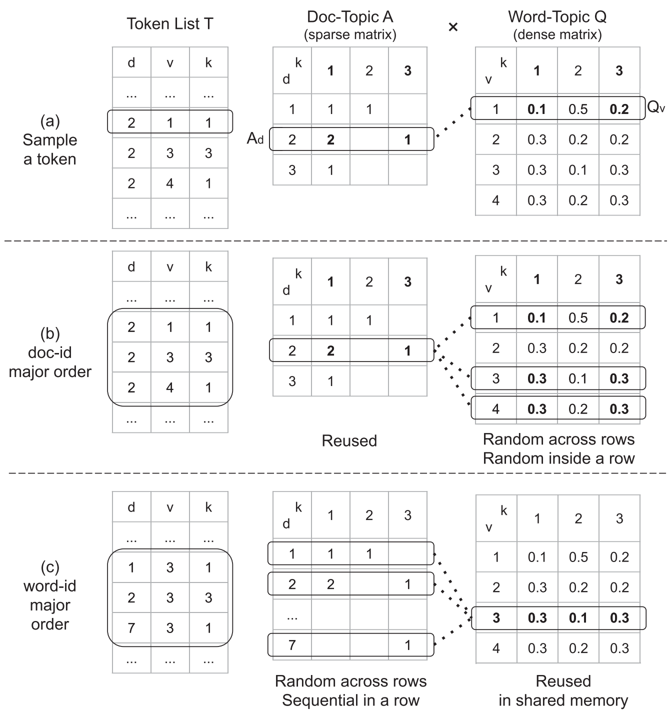


Fig. 2. Memory Access Pattern.

There are two particular visiting orders that reuse the result of previous memory accesses for better locality. The *doc-major order* sorts the tokens by their document-ids, so that tokens belonging to the same document are adjacent in the token list. Before processing the tokens in the dth document, the row $\mathbf{A}_d$ can be fetched into the shared memory and reused to sample all the subsequent tokens in that document. By contrast, the access of $\mathbf{Q}_v$ cannot be reused because each token needs to *access random elements* (indexed by non-zero entries of $\mathbf{A}_d$) *in a random row* of the word-topic count matrix **Q**, as shown in Fig. 2b. The bottleneck of this approach is accessing **Q**, where both the row index and column index are random.

By constract, the *word-major order* sorts the tokens by their word-ids, so that tokens belonging to the same word are adjacent in the token list. Before processing the tokens of the vth word, the row $\mathbf{Q}_v$ can be calculated in the shared memory on the fly and reused to sample all the subsequent tokens of that word. Each token needs to *access all the elements of a random row* $\mathbf{A}_d$ (Fig. 2c). The bottleneck of this approach is accessing **A**, where only the row index d is random.

The memory hierarchies can be quite different on CPUs and GPUs. CPUs have larger cache size (>30 MB) and smaller cache line (64B), whereas GPUs have smaller cache size (>2 MB) and larger cache line. On CPUs, when K and V are small, the document-major order can have better cache locality than the word-major order because the matrix **Q** can fit in the cache [39] for small datasets. However, for GPUs, the word-major order has a clear advantage in that it fully utilizes the cache line by accessing whole rows of $\mathbf{A}_d$ instead of random elements. Therefore, we choose the word-major order for SaberLDA. The pseudo code of sampling tokens in word-major is shown in Algorithm 4.

Combining this sampling order with the document partition strategy, SaberLDA adopts a *hybrid* data layout called partition-by-document and order-by-word (PDOW), which means to first partition the token list by document-id, and then sort the tokens within each partition by word-id. Unlike simple layouts such as sorting all tokens by document-id or by word-id in previous systems [28], [33], [35], [36], [39], [41], PDOW combines the advantages of both by-document partitioning and word-major ordering, and simultaneously maximizes cache locality with the word-major order, while keeping the GPU memory consumption small with the by-document partitioning.

3.3 Warp-Based Sampling

We now turn to the inner loop of sampling tokens within a word, i.e., line 7 in Algorithm 4, which is the most time consuming part of LDA. To understand the challenges and efficiency-related considerations, it is helpful to briefly review GPU architecture.

Algorithm 4. Sampling Tokens in Word-Major Order

```
1: for $i \leftarrow 1$ to num_iteration do
2:   $\mathbf{A}, \mathbf{B}, \mathbf{C} \leftarrow$ Count$\mathbf{T}$
3:   for $v \leftarrow 1$ to $V$ do // by block
4:     shared $\mathbf{Q}_v \leftarrow (\mathbf{B}_v + \beta)/(\mathbf{C} + V\beta)$
5:     shared $S \leftarrow \alpha \times |\mathbf{Q}_v|$
6:     shared Tree$_v \leftarrow$ PreBuilt$(\mathbf{Q}_v)$
7:     for $(d, k) \in \mathbf{T}_v$ do // by warp
8:       $\mathbf{X}_{dv} \leftarrow \mathbf{A}_d \times \mathbf{Q}_v$
9:       $R \leftarrow$ RandFloat$(0, |\mathbf{X}_{dv}| + S)$
10:      if $R \leq |\mathbf{X}_{dv}|$ then
11:        $k \leftarrow$ SparseSample$(\mathbf{X}_{dv})$
12:      else
13:        $k \leftarrow$ TreeSample(Tree$_v$)
14:      end if
15:    end for
16:  end for
17: end for
```

GPUs follow a single instruction multiple data (SIMD) pattern, where the basic SIMD unit is a *warp*, which has 32 *data lanes*. Each lane has its own arithmetic logic unit (ALU) and registers, and all the lanes in a warp execute the same instruction. In NVIDIA's CUDA, each thread is executed on a lane, and every adjacent 32 threads share the same instruction. Readers can make an analogy between GPU warp instruction and CPU vector instruction.

3.3.1 Thread-Based Versus Warp-Based

The most straightforward implementation of sampling on a GPU is *thread-based sampling*, which samples each token with a GPU thread. Therefore, 32 tokens are processed in parallel with a warp. Thread-based sampling is acceptable when **A** is dense because the number of product operations (Line 8 of Algorithm 4) is always K, and there is no need to use if-branches for sparse sampling.

However, this approach has several disadvantages when using sparse sampling in word-major order. First, because each token corresponds to different rows of **A**, the numbers of product operations are different (Line 8 of Algorithm 4). In this case, all the threads need to wait for the slowest one, creating long waiting time. Second, the branch choosing the sampling function causes the thread divergence problem, i.e., if some of the threads go to one branch and other threads go to another branch, the warp needs to perform the instructions of both branches, which increases the waiting time. Finally, the memory access to **A** is *unconcealed* (Line 8 of Algorithm 4) because the GPU threads are accessing different and discontinuous addresses of the global memory.

To overcome the disadvantages of thread-based sampling, SaberLDA adopts *warp-based sampling*, in which all the threads in a warp collaboratively sample a single token. However, there are also challenges for warp-based sampling–all the operations need to be vectorized to maximize the performance. In the next part of this section, we introduce how to individually vectorize the element-wise product, the sparse sample and the tree sampling functions.

3.3.2 Element-Wise Product

The element-wise product step is vectorizable by simply letting each thread process an index, and a warp then computes the element-wise product for 32 indices at a time. All the threads are fully utilized except for the last 32 indices in the case if the number of non-zero entries of $\mathbf{A}_d$ is not a multiple of 32. The waste is very small since the entries of $\mathbf{A}_d$ ary typically many more than 32, e.g., 100.

3.3.3 Choosing the Branch

This step only consists of a random number generation and a comparison, whose costs are negligible. Note that thread divergence will not occur since the whole warp goes to one branch.

3.3.4 Sparse Sampling

This step samples a topic from the sparse probability array $\mathbf{X}_{dv}$, by vectorizing the sampling function listed in Algorithm 2,

First, we vectorize the computation of the prefix sum by using the *__shuf_down* intrinsic $O(\log_2 32)$ times [20]. We refer to this routine as the *warp_prefix_sum*.

Given the prefix sum, we need to determine the index of the first element that is greater than or equal to the random value just generated. This can be achieved in two steps:

1) Use the warp-vote intrinsic *__ballot* to make a 32-bit integer, whose ith bit is set to one if the ith prefix sum is greater than or equal to the random value , vise versa.
2) Use the intrinsic *__ffs* to return the index of the first bit 1 of the 32-bit integer.

We call these two steps the *warp_vote*, which returns an index that is greater than or equal to the given value, or is equal to -1 if there is no such index. The threads in the warp continue to fetch the next 32 values in the loop, or stop if they find such an index.

In addition to the eliminated waiting time and thread divergence, the memory access behavior of our implementation is also good. For the element-wise product, the access to **A** is continuous. Specifically, the warp accesses two 128-byte cache lines from the global memory, and each thread consumes two 32-bit numbers (an integer index and a float32 value). The accesses to **Q** are random, but they are still efficient since the current row $\mathbf{Q}_v$ has been loaded into the shared memory or cached.

3.3.5 Tree Sampling

We now present the details for sampling $\mathbf{Q}_v$ (Lines 6 and 13 of Algorithm 4). Note that Tree_v is built from $\mathbf{Y}_v$ which has the same distribution as $\mathbf{Q}_v$ ($\mathbf{Y}_v = \alpha\mathbf{Q}_v$). Hence, $\mathbf{Q}_v$ can also be used to build the sampling tree. As discussed in Section 2.3, this sampling problem is essentially the same problem as sampling from $\mathbf{X}_{dv}$, but the basic sampling algorithm has $O(K)$ time complexity for each token, so we want another approach. From the literature, there are two main optimizations based on the pre-processing of $\mathbf{Y}_v$, and we briefly review their data structures.

- An alias table [29] can then be built in $O(K)$ time, and each sample can be obtained in $O(1)$ time. However, building the alias table is sequential.
- The Fenwick tree [35] can be built in $O(K)$ time, and each sample can then be obtained in $O(\log_2 K)$ time. However, the branching factor of the Fenwick tree is only two, so the 32-thread GPU warp cannot be fully utilized.

Both approaches are designed for CPUs, and are slow to construct on a GPU because they cannot be vectorized. Vectorization is critical because using only one thread for pre-processing is much slower than using a full warp for pre-processing.

To allow vectorized pre-processing, we propose a W-ary tree (each node has no more than W children), which can be constructed in $O(K)$ time *with full utilization of GPU warp*. Subsequent samples can be obtained in $O(\log_W K)$ time, where W is the number of threads in a GPU warp, i.e., 32.

We emphasize that our main focus is efficient construction of the tree instead of efficient sampling using the tree, because the cost of sampling using the tree is negligible compared with sampling from $\mathbf{X}_{dv}$. Moreover, the sampling using our W-ary tree is efficient because In SaberLDA, K is no more than 1 million, and $\log_W K = 4$, so $O(\log_W K) = O(4)$ is on the same level as the $O(1)$ alias table algorithm.

Our W-ary tree is designed to efficiently find the location of a given number in the prefix-sum array. Each node of the tree stores a number, where the bottom-most level nodes stores the prefix-sums of the given array. The length of an upper level is equal to the length of the lower level divided by W, and the ith node in an upper level is equal to the $(iW-1)$-th node in the lower level. Fig. 3 illustrates the construction of the tree. This procedure is efficient because all the nodes in one layer can be constructed in parallel. Therefore, the GPU warp can be fully utilized with the aforementioned *warp_prefix_sum* function, which uses W threads to compute the prefix-sum of W numbers in parallel.

To find the position of a given value in the prefix-sum array, we recursively find the position on each level, from top to bottom (Fig. 3). Based on the particular construction of our tree, if the position on the lth level is i, then the position on the $l+1$-th level is between iW and $iW+W-1$. Therefore, only W nodes need to be checked on each level. This checking can be achieved efficiently using the aforementioned *warp_vote* function. Also, the memory access is efficient because the tree can be cached in the shared memory

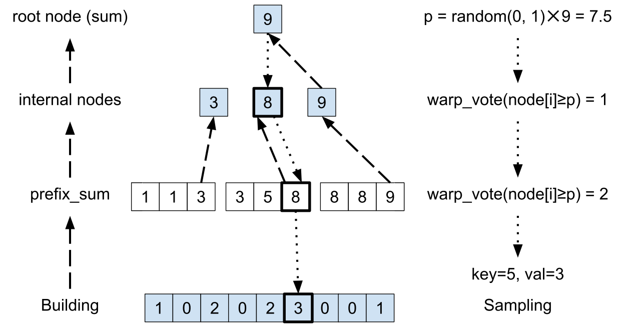


Fig. 3. Building a 3-ary tree and sampling from the tree.

(Section 3.2), and it only needs to read W continuous floating point numbers, i.e., a 128-byte cache line for each level.

3.4 Load Balancing

We now analyze the load balance of warp-based sampling in GPUs. A GPU can schedule GPU blocks overall computing units (SMs), and schedule warps of a block within one computing unit. We let a GPU block compute all tokens with the same word and each warp in the GPU block fetches one token at a time. We randomly partition the words into groups for multiple GPUs, so the computing workloads among GPUs are nearly the same. The word scheduling order is based on the descending order of word frequencies; hence, the GPU can use many blocks of small words to achieve balance among SMs.

However, using only one GPU block to compute the most frequent word may take longer than computing all other words in other blocks. For instance, one V100 GPU has 80 SMs and two V100s have 160 SMs. Each SM has to compute the 0.625 percent of the total workload for perfect balancing. By constrast,in the PubMed dataset, the computing workload of the most popular word is more than 1 percent. If we limit the computation of one word in a single GPU block, this will cause 60 percent imbalance overhead, which is unacceptable. Therefore, it is necessary to split the computation of popular words into multiple GPU blocks.

Our work proposes a split index format, which makes changes to the CSR index of the token list. In the traditional CSR format, each index represents the range of tokens belonging to a word in the vocabulary. In our split index format, the indices of those popular words are split into multiple pieces. A split parameter P is given to split the indices of words that have more than P tokens. If a word has t tokens, the index is split into $\lceil t/P \rceil$ pieces.

Fig. 4 shows an example. Without the split index format, block 0 computes the entire workload of word 1, which is larger than any other blocks and creates imbalance. With the help of the split index, the workload of word 1 is split into blocks 0 and 1, so each block has a similar workload and the heavy block is eliminated, avoiding the imbalance.

3.5 Efficient Sparse Matrix Counting

In this section, we introduce how to count the sampled topics into sparse count matrices $\mathbf{A}$ and $\mathbf{B}$ (line 2 in Algorithm 4). Use of a sparse matrix format can greatly reduce memory space consumption and accelerate construction time. The theoretical time complexity of counting topics is

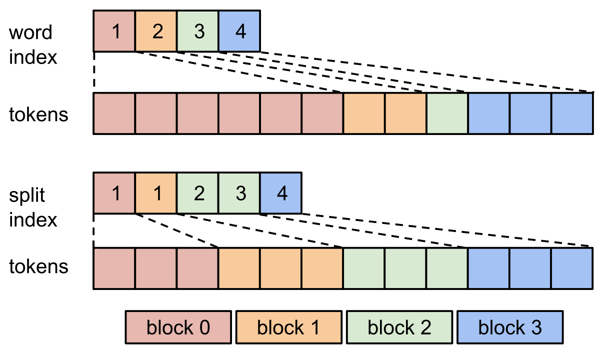


Fig. 4. Split Index Format.

proportional to the number of tokens, i.e., $O(T)$, which is less than the sampling tokens (TK_D). However, sparse matrix counting has an undeterministic result size, which may lead to extra space exceeding the limited GPU memory. Moreover, if not well optimized in the GPU, the time consumption of counting becomes greater than the sampling step.

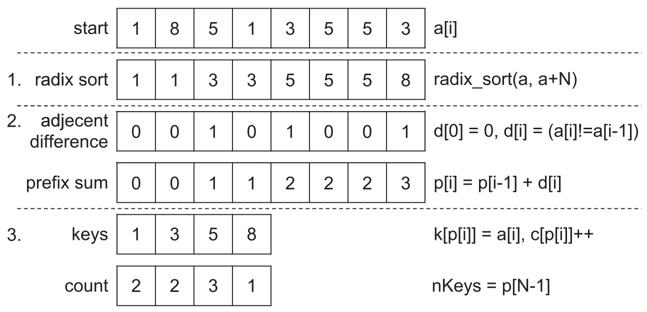
3.5.1 Count Doc-Topic Matrix $\mathbf{A}$

Based on the word-major sampling order, the topics sampled from tokens are grouped by word, but are with random documents. It is difficult to directly update the document count matrix. Fortunately, since the tokens are partitioned by document, we can calculate the partition of count matrix $\mathbf{A}$ when the sampling of the partition is finished.

A naïve approach to count the matrix is to sort all the tokens by first document-id d, then topic assignment k, and then perform a linear scan. However, the sorting is expensive since the global memory must be frequently accessed. Moreover, the efficiency of sorting decreases as the size of the token list increases.

We propose a procedure called *shuffle and segmented count* (SSC) to address this problem. We first perform a *shuffle* to organize the token list by the document-id's, i.e., segment the tokens into D small lists where the tokens in each list share the same document. The shuffle can be achieved by sorting all the tokens by the document-id d, but the sorting need not be actually performed because the target place of each token is fixed. We pre-process an index array of target places of tokens in the CPU, and transfer it to the GPU following the document partition.

Then, we parallelly count each segment (tokens of each document) in the GPU, which is a common problem known as *segmented count*. Unlike similar problems, such as


start
a[i]
1. radix sort
radix_sort(a, a+N)
2. adjecent difference
d[0] = 0, d[i] = (a[i]!=a[i-1])
prefix sum
p[i] = p[i-1] + d[i]
3. keys
k[p[i]] = a[i], c[p[i]]++
count
nKeys = p[N-1]

start	1	8	5	1	3	5	5	3	a[i]
1. radix sort	1	1	3	3	5	5	5	8	radix_sort(a, a+N)
2. adjecent difference	0	0	1	0	1	0	0	1	d[0] = 0, d[i] = (a[i]!=a[i-1])
prefix sum	0	0	1	1	2	2	2	3	p[i] = p[i-1] + d[i]
3. keys	1	3	5	8					k[p[i]] = a[i], c[p[i]]++
count	2	2	3	1					nKeys = p[N-1]

Fig. 5. Segmented count.

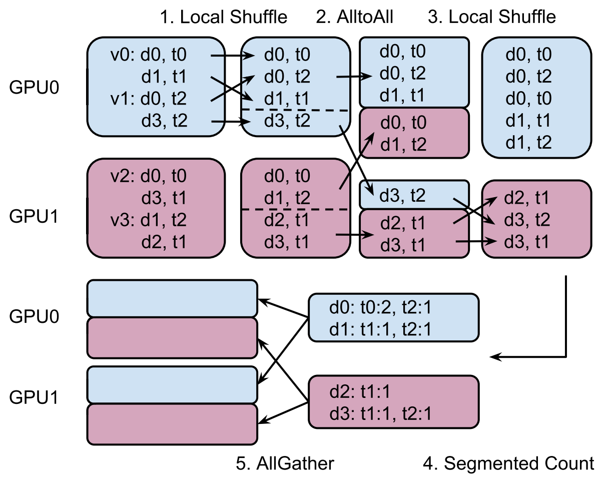


Fig. 6. Construct Doc-Topic Matrix $\mathbf{A}$ in multiple GPUs.

segmented sort [25], and segmented reduce [24], which have fully optimized algorithms for GPUs, an efficient GPU solution of segmented count has not yet been well studied.

We propose a solution of segmented count that is sufficiently efficient for SaberLDA. Our procedure consists of three steps, as illustrated in Fig. 5:

1) Perform a radix sort of each segment. Since a segment is short, it can be accelerated by caching in the shared memory;
2) Calculate the prefix sum of the adjacent difference, to get the number of different topics and the order number of each topic; and
3) Assign the topic number at the corresponding order number, and increase the count of the same topic.

In the multi-GPU system, tokens are grouped by different words and split into different GPUs. Our goal is to construct the doc-topic model $\mathbf{A}$ and replicate it to all GPUs. Although its computing time is considerably shorter than the kernel sampling function, using only one GPU to compute still hurts the multi-GPU scalability. Our work extends the *shuffle and segmented count* method into a multi-GPU system. To achieve good scalability, we rearrange the tokens in document order and re-partition them to all GPUs. Therefore, the segmented count calculation is evenly separated in each GPU. We implement a distributed doc-topic matrix construction according to the following steps, shown in Fig. 6:

1) For each GPU, shuffle the tokens into document order.
2) Use the All-to-All transfer method to send tokens to the corresponding GPU.
3) Shuffle the tokens in each GPU again, ordering all tokens by document id.
4) Call the segmented count kernel of each document partition. Thus each document has its topic count distribution.
5) Use the All-gather method to broadcast every document count partition to all GPUs. Each GPU receives all document count partitions and assembles them into a complete doc-topic matrix.

Like the single GPU version, the shuffle index of each token is unchanged among all estimate iterations. All shuffle indices and all-to-all metadata can be pre-possessed in the

TABLE 3
Test Platforms

Platform	A	B
CPU	Intel Xeon E5-2670 v3 12 cores x2	Intel Xeon E5-2620 v4 8 cores x2
Memory	128 GB	256 GB
GPU	NVIDIA GTX 1080	NVIDIA Tesla V100 x4
Memory	8 GB	16 GB x4

CPU before the first iteration. To save the transfer bandwidth between the CPU and the GPU, the doc-topic matrix is only temporally calculated and used inside GPUs. Only indices and metadata are transferred from CPU to GPUs.

We use NVLink instead of PCI-e for communications between GPUs for its high bandwidth and extra-communication channels. For example, while learning 1000 topics from PubMed dataset in our experiment platform (in Table 3), the computation time per iteration takes about 500 ms with 4 GPUs. The communication between GPUs needs 17ms via NVLink, while it requires more than 70 ms via PCI-e.

3.5.2 Count Word-Topic Matrix $\mathbf{B}$

Finally, we discuss how to efficiently update the dense count matrices $\mathbf{B}$. When the sampling of all tokens with the same word is finished, the corresponding row $\mathbf{B}_v$ of the word-topic count matrix is ready to be updated. The *atomicAdd* function must be used because there may be GPU blocks updating the same row, but the overhead is very low since the time complexity of updating is lower than the time complexity of sampling. The word-topic probability matrix $\mathbf{Q}$ can be easily generated according to Eq. (2b) from $\mathbf{B}$ after all the updates are finished. Maximal parallel performance can be achieved since both matrices are dense.

4 Evaluation

In this section, we provide an extensive set of experiments to analyze various aspects of the performance of SaberLDA, and to compare SaberLDA with other cutting-edge open source implementations. We analyze the performance on a modern multi-GPU platform and process it on large datasets. The code of SaberLDA is about 3,000 lines, written in CUDA and C++.

4.1 Experiment Setup

4.1.1 Test Platforms

We have two test platforms to test our program as well as other existing solutions.

Platform A has two Intel E5-2670v3 CPUs, with 12 cores per CPU, 128 GB main memory and an NVIDIA GTX 1080 GPU. Platform B is a multi-GPU machine which consists of 4 NVIDIA V100 NVLink-connected GPUs. The bi-directional bandwidth between each pair of GPUs is 96 GB/s. A PCI-e x16 bus is in charge of transferring data between CPU and GPUs. It has 12 GB/s bandwidth for a single direction.

4.1.2 Dataset

We use the dataset from the UCI machine learning repository. NYTimes consists of news articles and PubMed has biomedical literature abstracts. We also use ClueWeb12 [6] dataset, which is a crawl of web pages, as a large dataset. We first extract text from the HTML pages, remove stop words, and tokenize them. A subset of ClueWeb12 that can be fit in main memory is randomly selected. The statistic details of the datasets we used for testing are listed in Table 4.

TABLE 4
Statistics of Tested Datasets

Dataset	D	T	V	T/D
NYTimes [2]	300k	100M	102k	332
PubMed [2]	8.2M	738M	141k	90
ClueWeb12 subset [6]	33.1M	8.9B	100k	268

The hyper-parameters $\alpha = 50/K$ and $\beta = 0.01$ are set according to previous works [9], [17], [34], [35].

4.1.3 Experiment Metrics

The training time of LDA depends on both the number of iterations to converge and the time for each iteration. The former depends on the algorithm, e.g., variational Bayes algorithm [4] typically requires fewer iterations than ESCA [39] to converge. The latter depends on the time complexity of sampling each token as well as the implementation, e.g., the sparsity-aware algorithm has $O(K_D)$ time complexity, and faster performance than the $O(K)$ vanilla algorithms. We use various metrics to compare LDA implementations as follows.

We use *time per iteration* or *throughput* to compare different implementations of the same algorithm, e.g., SaberLDA with a CPU implementation of the ESCA algorithm, because they require the same number of iterations to converge. The throughput is defined as

$$\text{Throughput} = \frac{\text{number of tokens } T \text{ processed}}{\text{running time}},$$

and the unit is million tokens per second (MToken/s).

We compare different algorithms by *the required time to converge to a certain model quality*, because some algorithms may require more iterations to converge, but cost less for each iteration. The model quality is assessed by *holdout log-likelihood*, using the partially-observed document approach [30]. The log-likelihood of each token $L_{d,v}$ is calculated by the following equation:

$$L_{d,v} = \log \frac{\sum_k P_k}{|\mathbf{T}_d| + K\alpha}, \tag{4}$$

where P_k and $\mathbf{T}_d$ are defined in Eqs. (1) and 3, respectively. Higher log-likelihood indicates better model quality.

4.2 Impact of Optimizations

We first investigate the impact of each optimization technique proposed in previous sections by training the LDA on the NYTimes dataset with 1,000 topics for 100 iterations, with the result shown in Fig. 7. The total elapsed time is decomposed as the *Sample* function, rebuilding the document-topic matrix $\mathbf{A}$, constructing pre-processed data structures for sampling, and data transferring the between CPU and GPU.

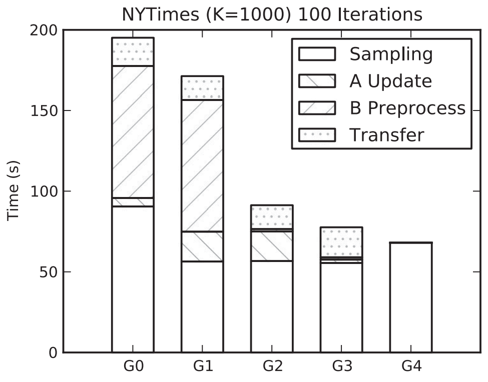


Fig. 7. Impact of optimizations tested in Platform A. G0: Baseline; G1: World-major order; G2: W-ary tree; G3: Shuffle and Segmented Count; G4: Asynchronous.

G0 is the most straightforward sparsity-aware implementation on GPU which sorts all tokens by documents, performs the pre-processed sampling with the alias table, and builds count matrices by naïve sorting of all tokens. G1 adopts the *word-major order* proposed in Section 3.2, and the time of sampling is reduced by almost 40 percent because of the improved locality for sampling. Note that G1 takes more time than G0 to rebuild the doc-topic matrix A because the tokens are ordered by word, and the sorting becomes slower.

The bottleneck of G1 is the construction of the alias table, since it is hard to vectorize. In G2, we replace the alias table with the W-ary tree in Section 3.3.5, which fully utilizes warps to greatly reduce the construction time by 98 percent.

We optimize the rebuilding of the document-topic matrix $\mathbf{A}$ with shuffle and segmented count (Section 3.5.1) in G3, and reduce the rebuilding time by 89 percent. Now, the time for updating $\mathbf{A}$ and pre-processing is negligible. Finally, in G4, we enable multiple workers running asynchronously to hide the data transfer between the CPU and GPU. This reduces the total running time by 12.3 percent. Overall, these optimizations combined achieve 2.9x speedup comparing with the baseline version G0. We emphasize that G0 is already highly optimized, and handle more topics than previous GPU implementations because it adopts the sparsity-aware algorithm with time complexity $O(K_D)$.

4.3 Profiling Analysis

Next, we analyze the utilization of hardware resources with NVIDIA visual profiler in Platform A. We focus on memory bandwidth because LDA is a memory intensive task [9].

Table 5 is the memory bandwidth utilization of the first 10 iterations on NYTimes with $K = 1,000$. Statistics show that the throughput of device memory reaches more than 140 GB/s, which is approximately 50 percent of the bandwidth.

TABLE 5
Memory Bandwidth Utilization, NYTimes K=1000

	Throughput (GB/s)	Utilization
Device memory	144	50%
L2 cache	203	30%
L1 unified cache	894	20%
Shared memory	458	20%

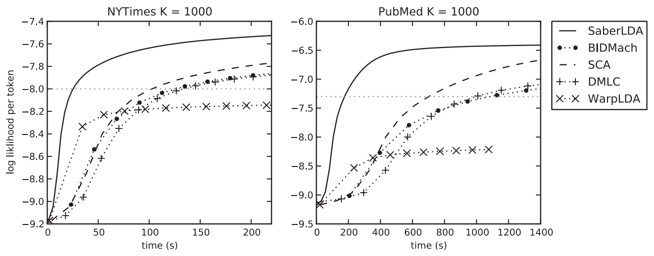


Fig. 8. Convergence over time with 1000 topics in Platform A.

This shows clear advantage over CPUs, given that the bandwidth between the main memory and CPU is only 40-80 GB/s. The throughputs of the L2 cache, unified L1 cache, and shared memory are 203 GB/s, 894 GB/s, and 458 GB/s respectively, while the utilization is lower than 25 percent. Therefore, these are not the bottlenecks of the overall system performance.

We further use performance counters to examine the kernel function, which shows that the memory dependency is the main reason of instruction stall (47 percent), and the second reason for execution dependency (27 percent). The hotspot is computing the element-wise product between the sparse vector $\mathbf{A}_d$ and the dense vector $\mathbf{Q}_v$, which is expected because it accesses global memory.

4.4 Comparison With Other Implementations

We compare SaberLDA with one previous GPU-based implementation, BIDMach [41], as well as three CPU-based implementations, ESCA (CPU), DMLC [38], and WarpLDA [9]. BIDMach [41] is the open-source GPU-based implementation. It is a general GPU-based machine learning tool whose algorithms are implemented in Scala, and the basic kernel functions, such as matrix manipulation are compiled to the CUDA library. BIDMach reports better performance than Yan *et al.*'s implementation [33], [41], and Steele and Tristan's implementation only reports tens of topics in their paper. Therefore, we think it is reasonable to compare SaberLDA with BIDMach, and we choose the batched Variational Bayes algorithm of BIDMach because it has better convergence than other algorithms on thousands of topics. ESCA (CPU) is a carefully optimized CPU version of the ESCA algorithm that SaberLDA also adopts. DMLC has a collection of multi-thread LDA algorithms on CPU , and we choose the FTreeLDA algorithm that achieved the best performance among all algorithms provided by DMLC. WarpLDA uses a state-of-art Metropolis-Hastings sampling algorithm with $O(1)$ time complexity per token.

We compare the time to converge of these implementations on four settings on the NYTimes and PubMed datasets, with the number of topics $K = 1000$. Fig. 8 shows the convergence over time. We compare the time to converge to the log-likelihood of -8.0 and -7.3, for NYTimes and PubMed, respectively. SaberLDA is 5.0 - 6.2 times faster than BIDMach. We also attempt to perform the comparison with 3,000 and 5,000 topics, and find that BIDMach is more than 10 times slower than SaberLDA with 3,000 topics, and reports an out-of-memory error with 5,000 topics. This is as expected because the time consumption of BIDMach grows linearly with respect to the number of topics, and its dense matrix format is much more memory consuming than SaberLDA. SaberLDA

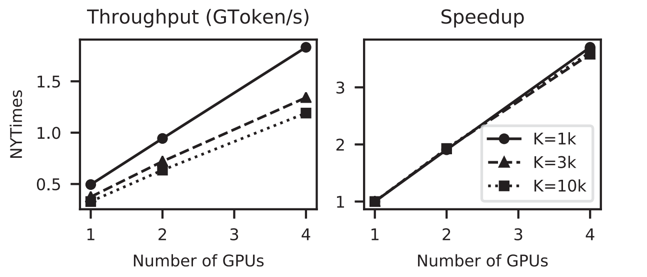


Fig. 9. Multi-GPU Speedup of NYTimes Dataset with 100 Iterations in Platform B.

is about 4 times faster than ESCA (CPU) and 5.4 times faster than DMLC on the two datasets with $K = 1,000$. WarpLDA converges to a worse local optimum possibly because of its inexact algorithm and the different metric we used to assess model quality compared with its paper [9]. This shows that SaberLDA is more efficient than other implementations.

4.5 Multi-GPU Performance

We now analyze the performance of SaberLDA on Platform B, which has 4 V100 NVLink-connected GPUs.

4.5.1 *Scalability*

Fig. 9 shows the throughput and speedup in multi-GPUs. The throughput is the average among 100 iterations. We use the NYTimes dataset and split the documents into two partitions for overlapping of the peer-to-peer transferring and calculating steps. All data stays in the GPU, avoiding the PCI-e transferring bottleneck between the CPU and GPUs. The shared memory is used to fetch the current row of $\mathbf{Q}_v$ when estimating 1000 topics while processing 3k and 10k topics would use the global memory.

Our work achieves throughput of 1.83 GToken/s with the NYTimes dataset and 1000 topics with 4 GPUs. When the number of topics increases, the throughput reduces to 1.34 GToken/s for 3k topics and 1.19 MToken/s for 10k topics. This is because of the increasing size of the vector $\mathbf{B}$, which leads to more cache miss rate in the sampling kernel. The right subfigure of Fig. 9 shows that the speedup achieves about 1.92x with 2 GPUs and 3.61x to 3.7x with 4 GPUs.

4.5.2 *Benefits of Split Index*

We analyze the benefit of the *Split Index* proposed in Section 3.4. We test the throughput of 100 iterations with the PubMed dataset with 1000 topics. Those word indices are split if its percentage of workload is larger than the average workload per GPU SM. The results shows are presented in Fig. 10. Both versions have the same performance with a single GPU. With 4 GPUs, the original version has less than 3x speedup, whereas the split-index version has nearly 4x speedup. The split index method is effective when estimating skew vocabularies in multi-GPUs.

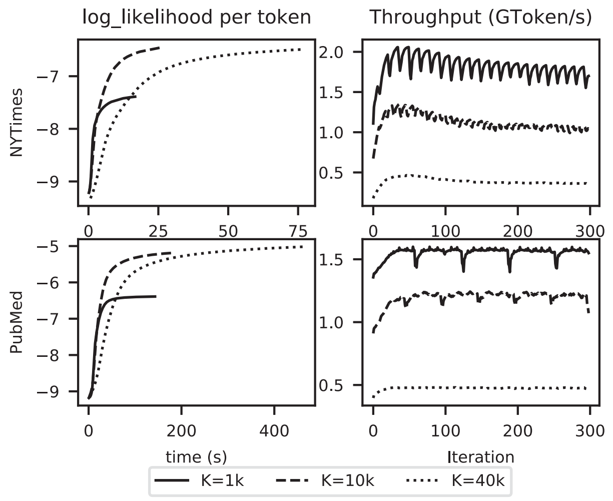


Fig. 11. Convergence and throughput for NYTimes and PubMed datasets with 300 iterations in Platform B.

4.5.3 *Convergence and Throughput*

We then analyze the convergence and throughput in the small datasets of NYTimes and PubMed. The numbers of topics are 1k, 10k, and 40k and we use an estimated 300 iterations, which is enough for the model to attain convergence. The results are shown in Fig. 11. SaberLDA finishes in 16, 27, and 77 seconds for NYTimes with the different numbers of topics, and takes 143, 186, and 466 seconds to finish for PubMed. The throughput achieves 2.05 GToken/s for NYTimes with 1000 topics and it becomes stable at 490 MToken/s when estimating for PubMed with 40k topics.

4.5.4 *Large Dataset*

Finally, to demonstrate the ability of SaberLDA to process large datasets, we test the performance of SaberLDA on a large subset of the ClueWeb12 dataset. In this experiment, we used the 4-GPU platform B as our test machine. We run SaberLDA in 300 iterations with 1,000, 10,000 and 40,000 topics separately, and the result are shown in Fig. 12. The throughput of estimating 1,000 and 10,000 topics are about 500 Mtoken/s, which is limited by the PCI-e x16 link connected between the CPU and the PCI-e hub in this platform. The throughput of estimating 40,000 topics is approximately

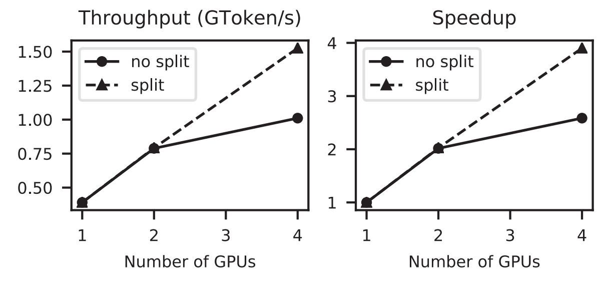


Fig. 10. Benefits of Split Index: PubMed 1000 Topics in Platform B.

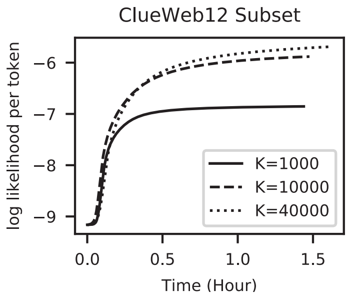


Fig. 12. ClueWeb12 subset in Platform B.

460 MToken/s. It takes 1.6 hours iterations and get the convergence of likelihood at -5.688.

5 Related Work

LDA Algorithms. The classic inference algorithms such as variational Bayes [4], [26], expectation propagation [21], and Collapsed Gibbs sampling (CGS) [15] have linear time complexity $O(K)$ for each token. Various algorithms have been proposed to optimize the time complexity to average topics per word $O(K_W)$ (SparseLDA [34]), average topics per documents $O(K_D)$ (AliasLDA [18], ESCA [39], F+LDA [35]), and constant time complexity $O(1)$ (LightLDA [36], WarpLDA [9]).

Distributed CPU LDA Systems. Distributed LDA systems were designed to accelerate training large datasets. They are based on MPI and OpenMP primitives (PLDA [31], AD-LDA [23], Peacock [32], F+LDA [35], WarpLDA [9] and [27]), parameter server abstraction (LDA* [37], LightLDA [36], Yahoo!LDA [1]), and Spark (ZenLDA [40]). The state-of-the-art distributed implementation [9] can infer 1 million topics and reach good scalability at 256 servers.

GPU LDA Systems. Yan *et al.* [33], BIDMach [41], Steele and Tristan [28], AGA-LDA [22], and Pyro [3] implemented LDA in GPU(s), supporting at most a thousand topics because of their $O(K)$ algorithms.

LDA Model Extensions. As extensions of the classic flat topic models. Hierarchical [14] and DAG-structured [19] LDA models can learn topics of different levels of abstraction. Chen *et al.* [10] implemented a distributed hierarchical LDA system with 50 machines.

6 Limitations, Discussion, and Future Work

One limitation of SaberLDA is that it can only scale-up to a server with multiple NVLink connected GPUs. We plan to extend the current implementation to multiple servers to make the training process even faster.

The other limitation is that SaberLDA holds all input data in the main memory of the CPU server and streams this data to GPUs. This can be improved by using NVMe SSD disks to support larger input data size.

Although we focus on the classic LDA algorithm in this paper, we believe the techniques proposed in this paper are generally applicable to hierarchical and DAG-structured LDA models, which also have sparse models.

7 Conclusion

We presented SaberLDA, a high-performance sparsity-aware LDA system on a multi-GPU system. By adopting sparsity-aware algorithms, SaberLDA overcomes the problems of previous GPU-based systems, which support only a small number of topics. We proposed a novel data layout, a warp-based sampling kernel, and an efficient sparse count matrix updating algorithm to address the challenges induced by sparsity. We then demonstrated the power of SaberLDA with extensive experiments. It can efficiently handle large-scale datasets with up to 8.9 billion tokens and learn large LDA models with up to 40,000 topics, which is orders of magnitude larger than existing GPU-based LDA systems. With the split index structure, and workload balancing algorithm, SaberLDA has linear speedup on 4 V100 NVLink connected GPUs and is more than 20 times faster than a modern dual CPU server. We hope our work will motivate more research on supporting more sparsity machine learning algorithms with GPUs.

Acknowledgments

This work was supported in part by NSFC for Distinguished Young Scholars under Grant No. 61525202 and Beijing Academy of Artificial Intelligence (BAAI).

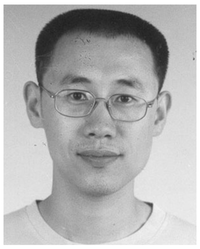
References

[1] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola, “Scalable inference in latent variable models,” in *Proc. 5th ACM Int. Conf. Web Search Data Mining*, 2012, pp. 123–132.
[2] D. Dua and C. Graff, “UCI machine learning repository,” University of California, Irvine, School of Information and Computer Sciences, 2017. [Online]. Available: http://archive.ics.uci.edu/ml
[3] E. Bingham *et al.*, “Pyro: Deep universal probabilistic programming,” *The J. Mach. Learn. Res.*, vol. 20, no. 1, pp. 973–978, 2019
[4] D. M. Blei, A. Y. Ng, and M. I. Jordan, “Latent dirichlet allocation,” *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, 2003.
[5] J. L. Boyd-Graber, D. M. Blei, and X. Zhu, “A topic model for word sense disambiguation,” in *Proc. Conf. Empir. Methods Natural Language Process. Comput. Natural Lang. Learn.*, 2007, pp. 1024–1033.
[6] J. Callan, “The lemur project and its ClueWeb12 dataset,” in *Proc. Invited Talk SIGIR Workshop Open-Source Inf. Retrieval*, 2012, pp. 11–20
[7] L. Cao and L. Fei-Fei, “Spatially coherent latent topic model for concurrent segmentation and classification of objects and scenes,” in *Proc. IEEE 11th Int. Conf. Comput. Vis.*, 2007, pp. 1–8.
[8] J. Chang and D. Blei, “Relational topic models for document networks,” in *Proc. 12th Int. Conf. Artif. Intell. and Statist.*, 2009, pp. 81–88.
[9] J. Chen, K. Li, J. Zhu, and W. Chen, “WarpLDA: A cache efficient o (1) algorithm for latent dirichlet allocation,” in *Proc. VLDB Endowment*, vol. 16, 2016, pp. 744–755.
[10] J. Chen, J. Zhu, J. Lu, and S. Liu, “Scalable training of hierarchical topic models,” *Proc. VLDB Endowment*, vol. 11, no. 7, pp. 826–839, 2018.
[11] N. Chen, J. Zhu, F. Xia, and B. Zhang, “Discriminative relational topic models,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 37, no. 5, pp. 973–986, May 2015.
[12] W.-Y. Chen, J.-C. Chu, J. Luan, H. Bai, Y. Wang, and E. Y. Chang, “Collaborative filtering for orkut communities: Discovery of user latent behavior,” in *Proc. 18th Int. Conf. World Wide Web*, 2009, pp. 681–690.
[13] Z. Ghahramani, “Probabilistic machine learning and artificial intelligence,” *Nature*, vol. 521, no. 7553, 2015, Art. no. 452.
[14] T. L. Griffiths, M. I. Jordan, J. B. Tenenbaum, and D. M. Blei, “Hierarchical topic models and the nested chinese restaurant process,” in *Proc. Advances Neural Inf. Process. Syst.*, 2004, pp. 17–24.
[15] T. L. Griffiths and M. Steyvers, “Finding scientific topics,” *Proc. Nat. Acad. Sci.*, vol. 101, no. (suppl 1), pp. 5228–5235, 2004.
[16] T. Iwata, T. Yamada, and N. Ueda, “Probabilistic latent semantic visualization: Topic model for visualizing documents,” in *Proc. 14th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2008, pp. 363–371.
[17] A. Q. Li, A. Ahmed, S. Ravi, and A. J. Smola, “Reducing the sampling complexity of topic models,” in *Proc. 20th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2014, pp. 891–900.
[18] M. Li *et al.*, “Scaling distributed machine learning with the parameter server,” in *Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 583–598.
[19] W. Li and A. McCallum, “Pachinko allocation: Dag-structured mixture models of topic correlations,” in *Proc. 23rd Int. Conf. Mach. Learn.*, 2006, pp. 577–584.
[20] M. Harris, S. Sengupta, and J. Owens, “Parallel prefix sum (scan) with CUDA,” 2017. [Online]. Available: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html
[21] T. Minka and J. Lafferty, “Expectation-propagation for the generative aspect model,” in *Proc. 18th Conf. Uncertainty Artif. Intell.*, 2002, pp. 352–359.

[22] G. E. Moon, I. Nisa, A. Sukumaran-Rajam, B. Bandyopadhyay, S. Parthasarathy, and P. Sadayappan, "Parallel latent dirichlet allocation on gpus," in *Proc. Int. Conf. Comput. Sci.*, 2018, pp. 259–272.
[23] D. Newman, A. Asuncion, P. Smyth, and M. Welling, "Distributed algorithms for topic models," *J. Mach. Learn. Res.*, vol. 10, pp. 1801–1828, 2009.
[24] NVIDIA, "Segmented reduction," 2013. [Online]. Available: https://nvlabs.github.io/moderngpu/segreduce.html
[25] NVIDIA, "Segmented sort and locality sort," 2013. [Online]. Available: https://nvlabs.github.io/moderngpu/segsort.html
[26] Y. W. Teh, D. Newman, and M. Welling, "A collapsed variational Bayesian inference algorithm for latent dirichlet allocation," in *Proc. 19th Int. Conf. Neural Inf. Process. Syst.*, 2006, pp. 1353–1360.
[27] S. Tora and K. Eguchi, "Mpi/openmp hybrid parallel inference for latent dirichlet allocation," in *Proc. 3rd Workshop Large Scale Data Mining: Theory Appl.*, 2011, Art. no. 5.
[28] J.-B. Tristan, J. Tassarotti, and G. Steele, "Efficient training of LDA on a GPU by mean-for-mode estimation," in *Proc. 32nd Int. Conf. Mach. Learn.*, 2015, pp. 59–68.
[29] A. J. Walker, "An efficient method for generating discrete random variables with general distributions," *ACM Trans. Math. Softw.*, vol. 3, no. 3, pp. 253–256, 1977.
[30] H. M. Wallach, I. Murray, R. Salakhutdinov, and D. Mimno, "Evaluation methods for topic models," in *Proc. 26th Annu. Int. Conf. Mach. Learn.*, 2009, pp. 1105–1112.
[31] Y. Wang, H. Bai, M. Stanton, W.-Y. Chen, and E. Y. Chang, "PLDA: Parallel latent dirichlet allocation for large-scale applications," in *Proc. Int. Conf. Algorithmic Aspects Inf. Manage.*, 2009, pp. 301–314.
[32] Y. Wang *et al.*, "Peacock: Learning long-tail topic features forindustrial applications," *ACM Trans. Intell. Syst. Technol.*, vol. 6, no. 4, pp. 1–23, 2015.
[33] F. Yan, N. Xu, and Y. Qi, "Parallel inference for latent dirichlet allocation on graphics processing units," in *Proc. Advances Neural Inf. Process. Syst.*, 2009, pp. 2134–2142.
[34] L. Yao, D. Mimno, and A. McCallum, "Efficient methods for topic model inference on streaming document collections," in *Proc. 15th ACM SIGKDD Int. Conf. Knowl. Discovery and Data Mining*, 2009, pp. 937–946.
[35] H.-F. Yu, C.-J. Hsieh, H. Yun, S. Vishwanathan, and I. S. Dhillon, "A scalable asynchronous distributed algorithm for topic modeling," in *Proc. 24th Int. Conf. World Wide Web*, 2015, pp. 1340–1350.
[36] J. Yuan *et al.*, "Lightlda: Big topic models on modest compute clusters," in *Proc. 24th Int. Conf. World Wide Web*, 2015, pp. 1351–1361.
[37] L. Yut, C. Zhang, Y. Shao, and B. Cui, "LDA*: A robust and large-scale topic modeling system," *Proc. VLDB Endowment*, vol. 10, no. 11, pp. 1406–1417, 2017.
[38] M. Zaheer, "Dmlc experimental-LDA," 2016. [Online]. Available: https://github.com/dmlc/experimental-lda
[39] M. Zaheer, M. Wick, J.-B. Tristan, A. Smola, and G. L. Steele Jr, "Exponential stochastic cellular automata for massively parallel inference," in *Proc. 19th Int. Conf. Artif. Intell. Statist.*, 2015, pp. 966–975.
[40] B. Zhao, H. Zhou, G. Li, and Y. Huang, "Zenlda: Large-scale topic model training on distributed data-parallel platform," *Big Data Mining Analytics*, vol. 1, no. 1, pp. 57–74, 2018.
[41] H. Zhao, B. Jiang, J. F. Canny, and B. Jaros, "Same but different: Fast and high quality gibbs parameter estimation," in *Proc. 21th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2015, pp. 1495–1502.
[42] J. Zhu, A. Ahmed, and E. P. Xing, "MedLDA: Maximum margin supervised topic models," *J. Mach. Learn. Res.*, vol. 13, pp. 2237–2278, 2012.

Kaiwei Li received the bachelor's degree from Tsinghua University, in 2014, and currently he is working toward the PhD degree in the Institute of High Performance Computing of Tsinghua University. His major research interests include parallel computing, machine learning accelerating, and GPU programming and distributed systems.

Jianfei Chen received BS and PhD degrees in computer science from Tsinghua University, in 2014 and 2019, respectively. He is currently a postdoctoral researcher with the Department of Computer Science and Technology, Tsinghua University. His major research interests efficient machine learning algorithms, probabilistic inference, and topic models.



Wenguang Chen received the BS and PhD degrees in computer science from Tsinghua University, in 1995 and 2000, respectively. He was the CTO of Opportunity International Inc. from 2000–2002. Since January 2003, he joined Tsinghua University. He is currently a professor and associate head with the Department of Computer Science and Technology, Tsinghua University. His research interest includes parallel and distributed computing and programming model.

Jun Zhu received the BS, MS, and PhD degrees from the Department of Computer Science and Technology, Tsinghua University. He is currently a professor with the Department of Computer Science and Technology, Tsinghua University. He was an adjunct faculty and a post-doctoral fellow at Machine Learning Department, Carnegie Mellon University. His research interest is on machine learning. He has published extensively at leading journals and conferences. He was selected as one of IEEE AI's to Watch, MIT TR35 China and China Computer Federation (CCF) Young Scientists.

CURE: A High-Performance, Low-Power, and Reliable Network-on-Chip Design Using Reinforcement Learning

Ke Wang, *Student Member, IEEE* and Ahmed Louri, *Fellow, IEEE*

Abstract—We propose CURE, a deep reinforcement learning (DRL)-based NoC design framework that simultaneously reduces network latency, improves energy-efficiency, and tolerates transient errors and permanent faults. CURE has several architectural innovations and a DRL-based hardware controller to manage design complexity and optimize trade-offs. First, in CURE, we propose reversible multi-function adaptive channels (RMCs) to reduce NoC power consumption and network latency. Second, we implement a new fault-secure adaptive error correction hardware in each router to enhance reliability for both transient errors and permanent faults. Third, we propose a router power-gating and bypass design that powers off NoC components to reduce power and extend chip lifespan. Further, for the complex dynamic interactions of these techniques, we propose using DRL to train a proactive control policy to provide improved fault-tolerance, reduced power consumption, and improved performance. Simulation using the PARSEC benchmark shows that CURE reduces end-to-end packet latency by 39 percent, improves energy efficiency by 92 percent, and lowers static and dynamic power consumption by 24 and 38 percent, respectively, over conventional solutions. Using mean-time-to-failure, we show that CURE is 7.7× more reliable than the conventional NoC design.

Index Terms—Computer architecture, network-on-chip(NoC), reliability, deep reinforcement learning

1 Introduction

Network-on-Chips (NoCs) [1], [2] have emerged as the standard interconnect solutions for connecting multiple cores, memory modules, and other hardware components. With continuous aggressive technology scaling, the reliability issue of NoCs becomes considerably more pronounced because the transistors and wires in NoCs are becoming increasingly vulnerable to faults, which are predominantly classified as permanent faults (induced by hardware aging) and transient errors (caused by runtime variations, overheated hardware, transistor delays, etc.).

A significant amount of work has been proposed to enhance the robustness of NoC [3], [4], [5], [6], [7], [8], [9], [10]. Unfortunately, these existing fault-tolerant methodologies have some critical defects. First, these techniques are limited, because they only focus on either faults within routers (at the gate-level) [3], [4] or faults that occur on inter-router links (at the link-level) [6], [7], [8], [9], [10]. Second, these conventional error-handling techniques can be expensive: SHIELD [3] and Vicis [4] duplicate the logic circuitry in routing pipeline stages or use redundant input ports, which consume massive chip area, while other retransmission-based fault-handling schemes, such as [5] and [6], generally incur substantial power consumption and prohibitive latency.

Deploying power-saving and performance-enhancing techniques to compensate the cost and performance degradation caused by fault-tolerant methodologies is indispensable yet extremely complex. Different optimization techniques, when being used simultaneously, can conflict and offset each other's desired goals, which presents various design tradeoffs. For example, channel buffers [7], [11], [12] replace the power-consuming router buffers with link storage to save power, but result in performance loss due to the limited throughput of link storage. Power-gating techniques [13], [14], [15] are proposed to take advantage of idle router periods to save power; however, they incur prohibitive wake-up latency. Dynamic voltage and frequency scaling [13], [16] intends to balance power savings and network throughput, but it can lead to increased transient faults [17]. Due to the explosion and complexity of the design space, we intend to use machine learning techniques to optimize the dynamic interactions of different techniques and automatically learn an optimal control policy to address the challenges of simultaneously decreasing power consumption, increasing performance, and improving reliability.

In this paper, we propose *CURE*, a learning-based NoC design framework, that handles permanent and transient faults at both the gate-level and link-level in a high-performance, low-power-consumption manner. CURE uniquely utilizes a per-router deep reinforcement learning (DRL)-based [18], [19], [20] control policy to explore the dynamic interactions among NoC components and system-level performance metrics as well as optimizing the trade-offs

• *The authors are with the Department of Electrical and Computer Engineering, George Washington University, Washington, DC 20052. E-mail: {cory, louri}@gwu.edu.*

Manuscript received 16 July 2019; revised 18 Dec. 2019; accepted 4 Apr. 2020. Date of publication 8 Apr. 2020; date of current version 1 May 2020. (Corresponding authors: Ke Wang.) Recommended for acceptance by M. Guo. Digital Object Identifier no. 10.1109/TPDS.2020.2986297

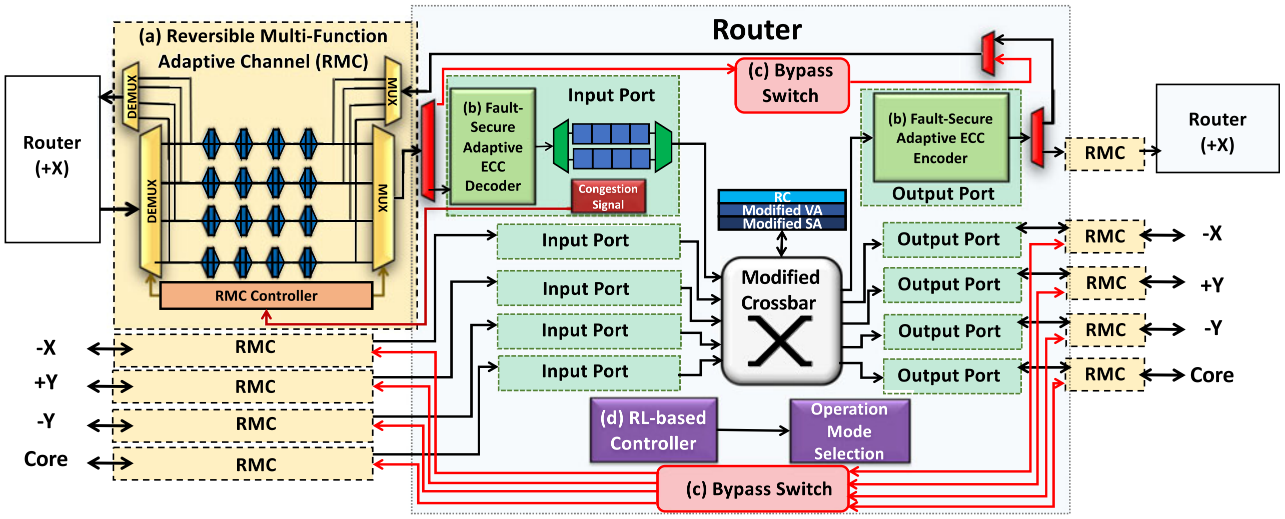


Fig. 1. CURE architecture design. CURE consists of (a) a reversible multi-function adaptive channel (RMC) between adjacent routers, (b) a new fault-tolerant router design with modified VA & SA & ST (crossbar) and adaptive error detection/correction hardware, (c) a router bypass route, and (d) a reinforcement learning (DRL)-based control policy.

at runtime. The major contributions of this paper are as follows:

- *Improved Inter-Router Channel Design:* We improve our previous proposed channel buffers [8] by designing reversible multi-function adaptive channel (RMC) buffers. RMC buffers have three basic functions: (1) forward/backward regular repeaters for flit propagation, (2) forward/backward buffers for link storage, and (3) forward/backward re-transmission buffers for handling faults. RMC is beneficial in several ways. First, with the additional storage available on the inter-router channel, dynamic power consumption for on-chip storage is reduced at high network loads without any performance degradation. Second, RMC provides flexibility for improving reliability at the link-level via re-transmission buffers and reversing the propagation direction to avoid faulty links. Third, the reversibility of RMC enhances performance by allowing the NoC to dynamically adapt to traffic because it provides extra link bandwidth in a specific direction at high network loads. Additionally, we utilize the bypass route proposed in [8], for the purpose of enhancing reliability, to retain NoC connectivity when permanent faults occur on RMCs.
- *Robust Router Microarchitecture Design:* We significantly improve the NoC robustness for both transient errors and permanent faults. First, we propose per-router self-diagnosis adaptive error control hardware to detect/correct faults at the link-level. The proposed error correction hardware adapts to the error level of each port and dynamically deploys the most efficient error detection/correction and flit re-transmission schemes with minimized power and latency overheads. Additionally, a low-cost, fault-vulnerable detector is implemented inside the error control hardware to detect malfunctions of the error control hardware itself. Second, we modify the circuitry of the router to mitigate transient errors and permanent faults that occur in the routing pipeline stages at the gate-level. Additionally, we propose a power-gating scheme that dynamically powers off NoC components (router buffers, crossbar, error control hardware, etc.) when needed to achieve power savings and mitigate aging effects.
- *DRL-Based Control Policy Design:* We propose a set of unique operation modes for each router with a DRL-based control policy to handle the dynamic interactions and optimize the trade-offs. The goal of dynamically utilizing different operation modes is to achieve improved performance, maximized power savings, and enhanced reliability. At runtime, per-router DRL agents observe and learn from the entire NoC environment and automatically evolve optimal per-router control policies that select the optimal operation modes at any given time.

We evaluate the performance of the proposed CURE architecture using a modified Booksim2 [21] simulator with PARSEC benchmarks on an 8×8 2D mesh architecture. We show that the proposed *CURE* provides significant power savings, enhanced reliability, higher performance, and lower area overhead compared to multiple state-of-the-art NoC designs with power-saving and fault-tolerant mechanisms.

2 CURE Microarchitecture

In this section, we demonstrate the architectural innovations of the proposed framework. The overall microarchitecture of the proposed design is shown in Fig. 1. The proposed design consists of inter-router links based on reversible multi-function channels (RMCs), a fault-tolerant router design, dynamic fault-secure error correction hardware, and a router bypass route for power savings and stress management. Additionally, a DRL-based controller is located in each router to handle the dynamic interactions and optimize the trade-offs. The implementation of RMCs is described in Section 2.1. The fault-tolerant router design is presented in Section 2.2. The adaptive error correction (ECC) hardware is presented in Section 2.3, and the bypass route is discussed in Section 2.4.

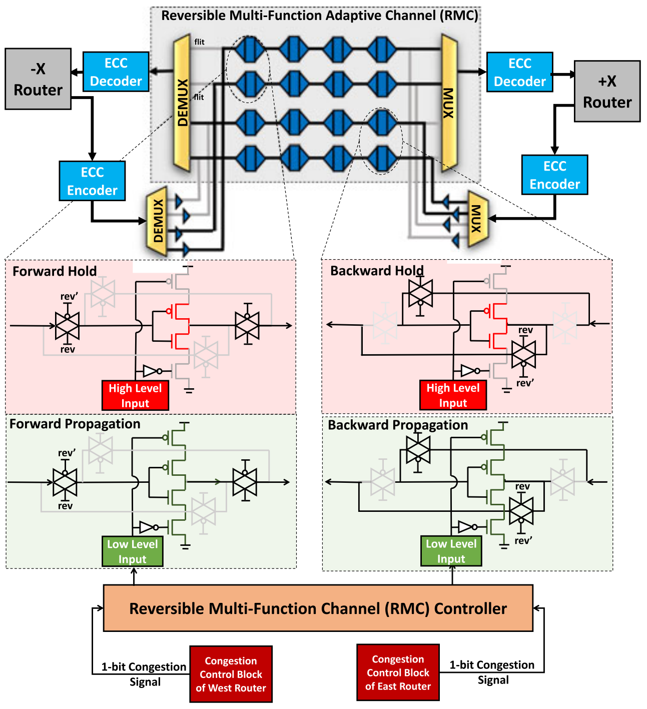


Fig. 2. Proposed reversible multi-function adaptive channel (RMC). Each RMC consists of four physical links with four buffer stages per link.

2.1 Reversible Multi-Function Channels (RMCs)

Although previous research [7], [11], [12] has shown that the excessive power consumption of router buffers can be reduced by moving storage to inter-router links or channels, the performance loss has not been thoroughly considered. In this paper, we borrow the idea of reversible links in [7] and extend the multi-function adaptive channel (MFAC) buffers in [8] to reversible multi-function adaptive channel buffers (RMC buffers), with the objective of improving network performance. With the newly designed channel architecture, RMC storage, links, and router storage are dynamically allocated according to traffic patterns to reduce power consumption. As shown in Fig. 2, each RMC buffer consists of an inverter and two tri-state transistors to enable store/propagation and four transmission control gates to reverse the link direction. The reversibility of RMC allows the links to adapt to different traffic loads for enhanced network throughput and the wear-out on different links for enhanced reliability.

Each RMC between two adjacent routers has four physical links, and the channel buffers are evenly allocated on those links.

Next, we use an RMC controller to dynamically and independently configure the transmission/buffer functions of the physical links to perform multiple RMC functions, as shown in Fig. 3. As described in detail in Section 2.1.1, RMCs can function as (1) forward/backward transmission repeaters, (2) forward/backward link storage, and (3) forward/backward re-transmission buffers. The dynamic selection of RMC functionalities is explained in Section 3. Because our design may lead to a latency penalty (due to the control overhead of the RMC buffers) and potential congestion (due to the head-of-line blocking of router buffers), we use dynamic router buffer allocation to maximize network throughput, as detailed in Section 2.1.2.

2.1.1 RMC Buffer Functionalities

Previous designs [8], [12] show that tri-state transistors can be used for propagating or storing flits. Fig. 2 shows the proposed RMC buffers based on those tri-state transistors. The proposed RMC between two adjacent routers consists of four physical links, with four buffer stages per link. Each physical link can be used for either storage or transmission as regular repeaters, in both directions, controlled by the additional four transmission gates. The direction and functionality of each RMC link are configured by the added RMC controller. The RMC controller first uses the a single-bit reversal (*rev* and *rev′*) signal to enable/disable the four transmission gates. In this way, the link can be configured in a specific direction. For example, when the rev signal is high, the RMC buffer will store/propagate flits forward. After configuring the direction, the RMC controller will send a function selection signal along with the 1-bit congestion signal from the downstream router to enable one of the functions of the RMC buffer. The RMC buffer can implement three basic functions in both directions, namely, link storage, transmission repeaters, and re-transmission buffers, which are illustrated in Fig. 3 and discussed below.

(1) Forward/Backward Transmission Repeater (Fig. 3a): In this case, the RMC buffer links are configured as repeaters. When the RMC controller is set to forward the congestion signal and the 1-bit congestion signal is low, the transistors connected to GND and V_{dd} are enabled, allowing the RMC to act as a transmission channel.

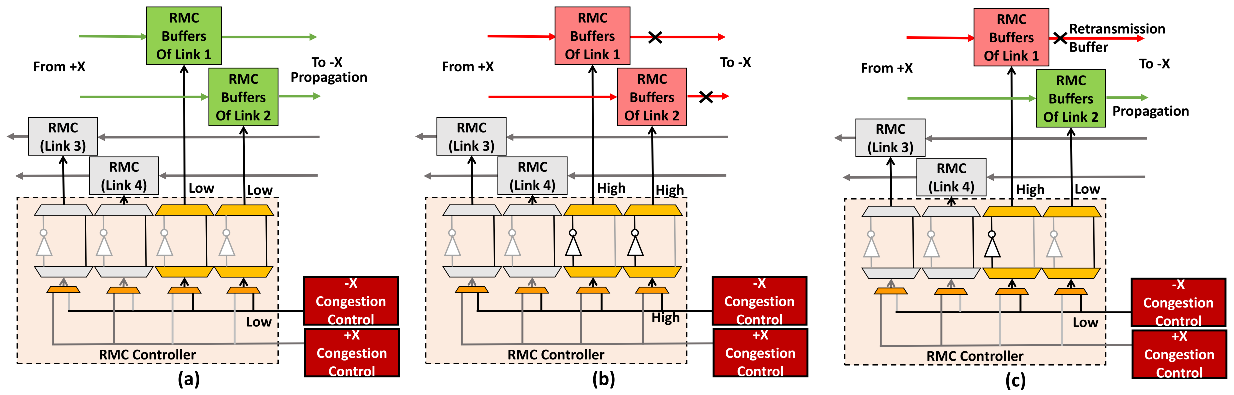


Fig. 3. Multi-function adaptive channel (RMC) buffers assume three different functions: (a) forward/backward regular repeaters for flit transmission, (b) forward/backward link buffers for storage on the link itself, and (c) forward/backward re-transmission buffers to store a copy of error-free flit for fault-tolerance.

(2) Forward/Backward Link Storage (Fig. 3b): In this case, the RMC buffer links are configured as link storage. When the RMC controller is set to forward the congestion signal and the 1-bit congestion signal is high, the transistors connected to GND and V_{dd} are disabled. Flits are then buffered in the transistors' capacitance.

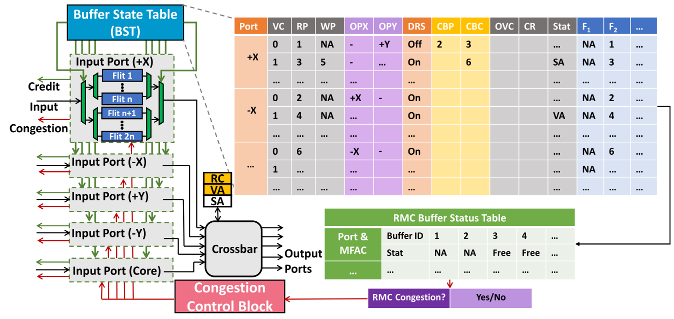
(3) Forward/Backward Re-transmission Buffer (Fig. 3c): This functionality can only be activated when at least two physical RMC links are in the same direction. In this case, we use one of the RMC buffer links to store flits for re-transmission purposes, whereas all other RMC buffer links are used for transmission. In conventional re-transmission-based error control design, a copy of the transmitted flit is stored in the local re-transmission buffer (in the upstream router) until it receives an acknowledgment (ACK) message back from the downstream router. The implementation of local re-transmission buffers can lead to excessive power and area overhead, especially when these re-transmission buffers are underutilized in low-error scenarios. Therefore, replacing the traditional in-router re-transmission buffers with RMC buffers is beneficial because the original flit will only be stored when needed (under higher error rates). Under this condition, the RMC controller will send the same packets/flits to both RMC buffer links. The RMC controller configures one of the RMC buffer links for storage (by applying a "hold" signal) and the other RMC buffer links for forwarding the flit (regular transmission). Upon receiving a NACK signal, the RMC controller releases the flit for re-transmission. If an ACK signal is received, the flit is discarded because the original transmission is error-free.

2.1.2 RMC Buffer Allocation and Flow Control

In CURE, to ensure connectivity in all directions, at least one of the four physical links in each RMC is allocated to each direction. For instance, as shown in Fig. 2, the top link is always facing -X, while the bottom link is facing +X. However, the middle links can be dynamically configured to face either direction. Therefore, the RMC has three different configurations (with the top link always facing -X, and the bottom link always facing +X):(1) the middle links are facing -X and +X, (2) both the middle links are facing -X, and (3) both the middle links are facing +X. For the ease of explanation, we name these three configurations $RMC_{2:2}$, $RMC_{3:1}$, and $RMC_{1:3}$, according to the link number of each direction.

As mentioned above, determining which direction to allocate the RMC links is critical. Network traffic is measured using link utilization and buffer utilization values monitored by the corresponding router. The RMC controller analyzes the network throughput in each direction, using the number of propagated packets, and calculates the ratio of these two throughput numbers. The RMC controller will allocate the RMC links with the configuration that is nearest to the ratio. For example, if the ratio of the -X traversal to the +X traversal is 2.6, the $RMC_{3:1}$ configuration will be utilized.

After allocating the direction of RMC links, the RMC buffers will be allocated using a unified buffer state table (BST), as shown in Fig. 4. The proposed BST is router-associated and shared by all the input ports within the router and remains accessible even if the router is power-gated (power-gating information is recorded in the yellow and orange



Port	VC	RP	WP	OPX	OPY	DRS	CBP	CBC	OVC	CR	Stat	F_1	F_2	...
+X	0	1	NA	-	+Y	Off	2	3			...	NA	1	...
	1	3	5	-	...	On		6			SA	NA	3	...
	...	...	...	...		...					...	...	...	...
-X	0	2	NA	+X	-	On					...	NA	2	...
	1	4	NA	...		On					VA	NA	4	...
	...	...	...	...		...					...	...	...	...
...	0	6		-X	-	On					...	NA	6	...
	1	...		...		...					...	NA	...	...
	...	...	...	...		...					...	...	...	...

RMC Buffer Status Table						
Port & MFAC	Buffer ID	1	2	3	4	...
	Stat	NA	NA	Free	Free	...
...	...	...	...	...	...	...

Fig. 4. Proposed unified buffer state table (BST). The green arrows indicate buffer slot allocation and credit signals by BST, while congestion signals are shown with red arrows.

entries shown in Fig. 4). The proposed BST is a modified version of the conventional virtual channel (VC) state table. The conventional VC table consists of the following information, or entries: the VC identifier (VC), read pointer (RP), write pointer (WP), allocated output port (OP), output VC (OVC), status (Stat), and credit count (CR). In the conventional VC state table, the header flit carries the packet information for route computation (RC) and VC allocation. The VC state table allocates a free VC slot to the header flit and records the VC information (VCID) and output information (output VC and output port). The body flits of the packet simply follow the VCID to find the correct output port from the VC state table. Thus, the packet is routed correctly.

In CURE, we modify the VC state table and implement BST to support RMC channel buffers and record routing information when the router is power-gated. Compared to input-port-associated VC state tables, the proposed BST is router-associated and shared by all input ports within the router. Other than the entries of the conventional VC state table, the proposed BST also consists of additional entries for allocating channel buffers: input port identifier (Port) that indicates the input port of the incoming flit, downstream router status (DRS) that indicates if the downstream router is power-gated, a channel buffer pointer (CBP) and channel buffer credit (CBC) that indicate the occupancy status of the associated RMC buffers. We also create two entries, OPX and OPY, to replace the conventional output port entry to support adaptive XY/YX routing. To enable BST functioning when the router is power-gated, we consider a separate supply voltage that is not powered-off for BST.

The flow control is simple. While the router is powered on, the body flits simply follow the VC and output port information carried by the header flit using the BST. Similarly, when the router is power-gated, the BST also records the VC and OP information of the header flit. Thus, the body flits can be routed to the associated output port by looking up the information. When the flit leaves the bypass switch, a credit is sent back to its upstream router for updating the credit information. This guarantees that the flow control operates normally, irrespective of whether the router is powered on or off. The BST is powered by a separate voltage supply. Additionally, the congestion control block monitors and updates the BST by recording all the available router buffer and RMC buffer slots. If all the router buffer slots and RMC buffer slots of an input direction are occupied, a congestion signal will be triggered.

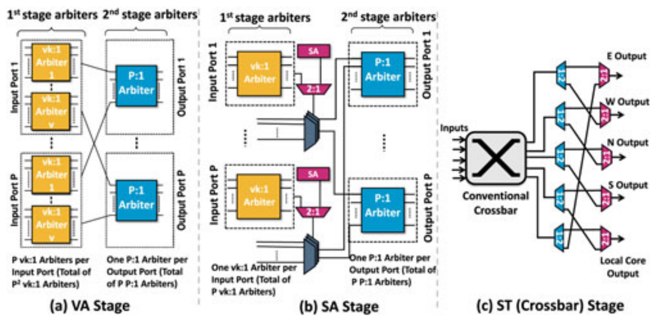


Fig. 5. Proposed fault-tolerant router.

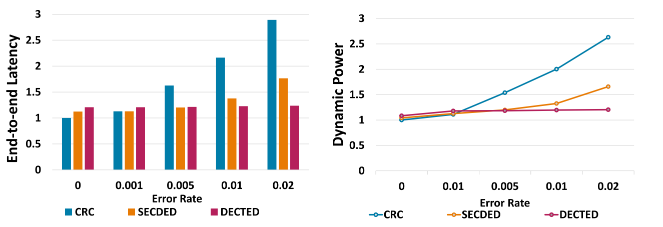


Fig. 6. Trade-offs of different static error control schemes.

To further balance traffic and utilize RMC resources, we propose an adaptive XY/YX routing algorithm to mitigate traffic in high-intensity ports/links to underutilized ports/links after RMC is configured. This can provide more routing options when flits compete for the same output port. At the RC stage, for a packet from the local injection port, the RC unit calculates the output ports out of both the XY and YX routes and stores the results in the corresponding VC state table entry. For a packet from other input ports, the routing identification bit in its head flit decides which routing computation unit is used. Because XY and YX routes can be calculated simultaneously with symmetric logic, the proposed RC unit does not introduce extra delays in the RC stage. The VC allocation stage utilizes the unified BST. All VC states are stored in the unified BST. The OP entry is extended to two parts: the output port for XY routing (OPX) and the output port for YX routing (OPY). For a packet newly injected from the injection port of the current router, the information of the output ports of both XY and YX routing is useful in the VA stage. For a packet just passing by the current router, the routing algorithm has already been determined; thus, only one of two output ports is valid. The other output port will be ignored based on the additional routing identification bit in the head flit. The proposed routing scheme is deadlock-free.

Note that in the 4-pipeline-stage router, the arbitration logic for VA/SA stages dominates the critical path and determines the minimum clock period. Using the Synopsys Design Compiler, we determined the critical delay of VA/SA to be 0.318 nsec (VA)/0.386 nsec (SA) for conventional routers and 0.246 nsec (VA)/0.453 nsec (SA) for CURE. Both of the critical delays are fulfilled within 0.5nsec clock (i.e., 2.0 GHz clock frequency).

2.2 Fault-Tolerant Router Design

A typical NoC router consists of input/output ports, buffers, routing logic, and a crossbar that connects the input ports to output ports for packet routing. The conventional router has four pipeline stages: routing computation stage, virtual channel allocation stage (VA), switch allocation stage (SA), and switch traversing (ST) , which is also known as the crossbar stage. Permanent faults occasionally occur in the last three stages if the corresponding hardware (i.e., arbiter, switch, link, and crossbar) is faulty [3]. In this paper, we propose a new router design that can tolerate permanent faults in the VA, SA, and ST (crossbar) pipeline stages.

VA and SA Stages. The CURE architecture uses the unified BST to allocate VC and switches. Conventional VA has two stages. In the first stage, each input VC that has a head flit arbitrates for an empty VC at the downstream router using the RC results. In the second stage, head flits across different input VCs that have been allocated the same virtual channel in the downstream router compete with each other. Similar to the VA stage, the SA stage also has two stages: the first stage decides which VC of an input port can propagate its flit to the crossbar stage, while the second stage resolves the competition between VCs of different input ports trying to access the same output port. Each of these stages is composed of a set of arbiters associated with a specific VC. Permanent faults may occur when any of the arbiters are faulty.

In this paper, several simple MUXes/DEMUXes are added to the conventional VA and SA stages to allow the router to bypass the faulty arbiter and borrow the unoccupied arbiter to perform VC and switch allocation. We modified the SHIELD [3] architecture to fit the proposed CURE using RMC buffers and unified BST. The proposed architecture is shown in Fig. 5. As shown in Fig. 5, in the VA and SA stages, arbiters from the virtual channels of the same input port can be shared.

ST (Crossbar) Stage. A crossbar is considered faulty if any of the MUXes/DEMUXes located in the crossbar are faulty and the packet fails to be forwarded to the output port. To overcome such faults, additional MUXes and DEMUXes are added to each output port to create a backup flit ejection path. The modified crossbar is shown in Fig. 5c. The crossbar is fault-free as long as either the original path or the backup path is not faulty.

2.3 Fault-Secure Adaptive ECC Hardware Design

Conventional static error correction hardware is either not power efficient or not powerful enough to handle transient faults: lightweight error control schemes (e.g., end-to-end CRC) can lead to excessive re-transmission traffic at high error rates, and powerful error correction schemes (e.g., double-error correction triple-error detection (DECTED)) are power consuming. We use three existing static error control schemes, namely end-to-end CRC, per-hop SECDED, and per-hop DECTED, to evaluate the trade-offs between NoC performance metrics of different static error mitigation techniques. The evaluation result is shown in Fig. 6. As shown in Fig. 6, for low error levels, SECDED and DECTED both negatively impact system performance due to their encoding and decoding overheads. However, as the error level increases, it is beneficial to deploy SECDED and DECTED to mitigate re-transmission packets. Therefore, there is a strong need for a dynamic error control hardware that can adapt to different error levels and apply the most efficient error mitigation technique at runtime. To this end, we propose a per-router-based adaptive error control hardware to

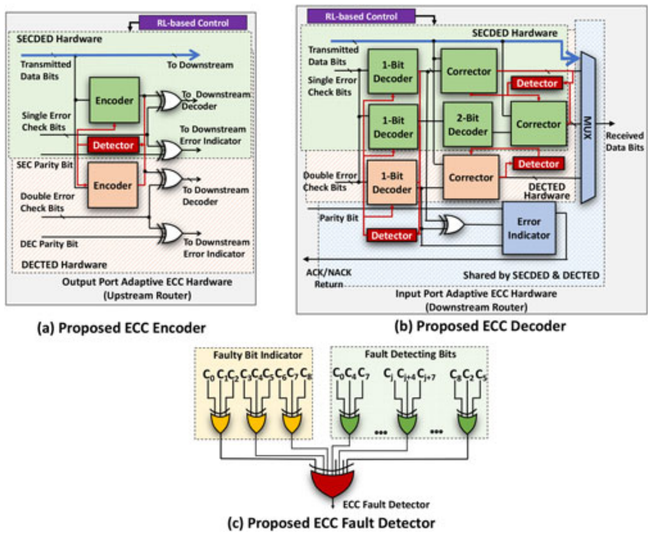


Fig. 7. Proposed fault-secure adaptive error control hardware. (a) Fault-secure adaptive error control hardware, including encoders located in the router's output port and decoders located in the router's input port. SECDED is active when logic circuits in green and blue are enabled, DECTED is active when logic circuits in green, orange, and blue are enabled. The red arrow shows flits with CRC enabled. (b)Proposed ECC fault detector. The green OR gates detect malfunctions in ECC hardware, while the yellow OR gates indicate if a permanent fault occurs.

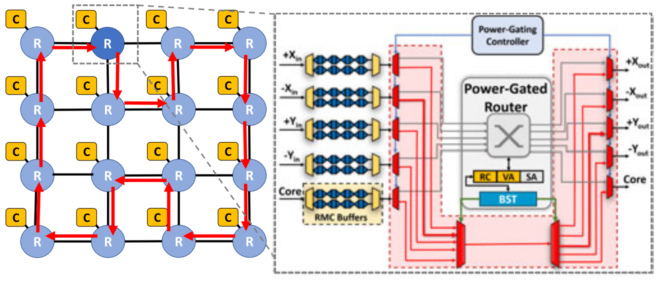


Fig. 8. Microarchitecture of a bypass route in each router. An example of the constructed bypass ring is shown on the left side.

mitigate soft errors in RMC links for balanced NoC power, performance, and reliability. Additionally, since transistors are less reliable with aggressive technology scaling, the combinational logic of ECC hardware is also vulnerable to faults. Thus, we also enhance the router with self-diagnosis function to ensure that the error control circuitries (i.e., ECC encoder and decoder) are fault-secure.

The proposed adaptive error correction hardware is shown in Fig. 7. The proposed error control hardware can be configured as end-to-end CRC, per-hop SECDED, and per-hop DECTED. At runtime, the ECC hardware dynamically deploys the most appropriate error control scheme guided by the DRL-based control policy discussed in Section 4. We reinforce the fault-prone ECC hardware design by proposing a self-diagnosis ECC fault detector to achieve fault-tolerance in both the communication channels and the ECC circuitry. The proposed self-diagnosis detector can verify the correctness of the ECC hardware operations using low-density parity-check (LDPC) codes [22]. As shown in Figs. 7a and 7b, each ECC encoder, decoder, and corrector is assigned to an ECC fault detector. The detector applies LDPC codes to each syndrome vector (the 9-bit Hamming code C_0 to C_8) from the outputs of different ECC hardware. The circuitry details of the proposed ECC fault detector are shown in Fig. 7c. The last OR gate has 12 inputs. The first 9 inputs are the required LDPC input. Because LDPC codes are proven to detect all the error combinations in the 9-bit syndrome vector [23], the proposed detector can detect malfunctions in the ECC hardware due to transient errors. To enhance the tolerance of the ECC hardware to permanent faults, we uniquely add three more inputs to the last OR gate of the error detector. These three bits can indicate the location of the faulty bit in the 9-bit Hamming code. If an error occurs in the same bit repeatedly, the corresponding gates in the ECC hardware will be marked as faulty, and the specific faulty gate will be power-gated, while the bit-level operations will be performed by the other underutilized gates (e.g., the XOR gates for DECTED). Additionally, note that the proposed ECC fault detectors are only activated when the SECDED/DECTED scheme is enabled.

2.4 Router Bypass Route Design

In this paper, we modify the stress-relaxing bypass technique originally proposed in [8], which proactively power-gates and bypasses the NoC router to save power and prevent overheating by adding an extra escape link. All of the inter-router escape links together construct a bypass-ring network to retain connectivity when there are a significant number of failures. The bypass-ring network fully connects each router in the NoC. An example of a chip-level bypass ring in a 4×4 2D mesh is demonstrated with red arrows on the left-hand side in Fig. 8. When all the possible links are faulty in one direction (i.e., all the corresponding RMCs are faulty), the escape link will be activated. Each escape link has one single-flit latch as the link in the stress-relaxing bypass route. The incoming flit will be stored at that single-flit latch and propagated using a round robin scheme. In this way, packets can pass through the faulty router and proceed to the next router. In addition, the proposed design continues to utilize the BST for routing information while the router is bypassed. This retains the connectivity of that direction and eliminates resource starvation to prevent deadlock. In this way, the new bypass route with escape links can further enhance reliability in the presence of faults. Additionally, to reduce the area overhead of the cross-router bypass-ring network, the ring network does not support bypassing in all directions, which means that each network interface can accept packets from only one specific upstream router and forward packets to specific downstream router.

3 Proactive Operation Modes

In this section, we propose ten proactive operation modes (1 power-gating mode and 9 fault-tolerant modes) for CURE routers. Each operation mode has various configurations of RMC, adaptive error control hardware, and power-gating strategies. Each CURE router occasionally and independently selects and deploys an operation mode proactively using a DRL-based control policy (described in Section 4). The operation modes are detailed below.

- *Operation Mode 0 - Power-Gating Mode:* In this mode, the router is power-gated, while the bypass route is enabled. The RMC channel is configured as 2:2 for both directions, and the RMC buffers are used to store the incoming flits. This operation mode is activated either when the router is underutilized or when a high risk of overheating is predicted. This mode mitigates permanent faults and saves static power.

- *Operation Modes 1, 2, and 3 - CRC-Only Mode:* Operation modes 1, 2 , and 3 have different RMC direction configurations: 2:2 for mode 1, 1:3 for mode 2, and 3:1 for mode 3. However, these operation modes share the same error control configurations. In these modes, the RMCs are configured as storage buffers, and the entire adaptive ECC hardware is power-gated, so that only CRC is enabled. These operation modes are beneficial to save power and eliminate ECC computational overhead when the error level is low.
- *Operation Modes 4, 5, and 6 - Per-hop SECDED Mode:* In these modes, the router's adaptive ECC hardware is partially activated to perform per-hop SECDED. This configuration is beneficial when SECDED can handle most of the faults. Otherwise, it will either lead to unnecessary power and latency penalties (when the error level is low) or excessive re-transmissions (when errors cannot be corrected by SECDED). The RMC buffers are configured as re-transmission buffers. Similar to operation modes 1, 2, and 3, the RMC direction configurations are set to 2:2 for mode 4, 1:3 for mode 5, and 3:1 for mode 6.
- *Operation Modes 7, 8, and 9 - Per-hop DECTED Mode:* In these modes, the router activates the entire adaptive ECC hardware to enable DECTED. This is the situation where the flits are more likely to contain errors of 2 or more bits. The RMC buffers are configured as re-transmission buffers. Similarly, the RMC direction configurations are set to: 2:2 for mode 7, 1:3 for mode 8, and 3:1 for mode 9.

The dynamic selection of operation modes is performed by each router independently in a sequence of discrete time steps using the DRL-based control policy presented in Section 4. At each time step, each router decides which operation mode to apply for the following time step and passes the decision to the downstream router. In this way, the downstream router will be informed to configure the ECC decoder located in the corresponding input port to apply the correct ECC coding, such that it is synchronized with the ECC coding of the upstream router's encoder at output port at the next time step. As demonstrated, the proposed dynamic operation modes allow individual routers to utilize the most suitable technique at runtime, resulting in greater benefits for the entire network.

4 Proposed Deep Reinforcement Learning (DRL)-Based Control Policy

We present a new per-router deep reinforcement learning-based control policy to dynamically select operation modes that can lead to the maximum system-level performance. Reinforcement learning is an online learning algorithm that learns and optimizes the behavior of autonomous RL agents [20](e.g., routers) from the dynamic interactions between the agents and the environment (e.g., NoC system)

Category	Features	Description
Router Input Related Metrics	1. +X link utilization	Input flits/cycle of +X input port
	2. –X link utilization	Input flits/cycle of –X input port
	3. +Y link utilization	Input flits/cycle of +Y input port
	4. –Y link utilization	Input flits/cycle of –Y input port
	5. Local port link utilization	Input flits/cycle of local port
Buffer Related Metrics	6. +X buffer utilization	the buffer utilization of +X input port
	7. –X buffer utilization	the buffer utilization of –X input port
	8. +Y buffer utilization	the buffer utilization of +Y input port
	9. –Y buffer utilization	the buffer utilization of –Y input port
	10. Local port buffer utilization	the buffer utilization of local port
Router Output Related Metrics	11. +X Link utilization	Output flits/cycle of +X input port
	12. –X Link utilization	Output flits/cycle of –X input port
	13. +Y Link utilization	Output flits/cycle of +Y input port
	14. –Y Link utilization	Output flits/cycle of –Y input port
	15. Local port Link utilization	Output flits/cycle of local port
RMC Related Metrics	16. +X RMC Previous Mode	Operation mode at the last timestep
	17. –X RMC Previous Mode	Operation mode at the last timestep
	18. +Y RMC Previous Mode	Operation mode at the last timestep
	19. –Y RMC Previous Mode	Operation mode at the last timestep
Other Metrics	20. Temperature	Local router temperature in °C

Fig. 9. Router attributes selected in the state vector.

at runtime. Specifically, in CURE, each router (agent), acts as a learner and a decision-maker and interacts with the NoC system (environment), in a sequence of discrete time steps t = 0, 1, 2, 3, and so on. At time step t, the router observes the current *state* by extracting runtime system attributes(e.g., buffer utilization, temperature, etc.), takes an *action* by selecting one of the proposed operation modes and applies it at the next step $t' = t + 1$. Next, at time step t +1, upon taking the action selected in the previous step, the NoC attributes change and result in a new state s'. The new state is fed back to the agent, and the time step is incremented. In addition to observing the new state, the agent also receives a *reward* r. A *policy* π maps states to actions, specifying how to choose actions given the state of the environment to maximize the cumulative reward. For the router, the cumulative reward will be a function of energy, performance, and reliability over the entire sequence of actions. An RL algorithm continually evolves the policy based on the router's past interactions with the NoC system.

Design Space and Action-Value Function. In CURE, the state space S is defined as a vector of several NoC system metrics, or features, listed in Fig. 9. These features contain input-related metrics (attributes 1 to 10), output-related metrics (attributes 11 to 15), RMC related metrics (attributes 16 to 19), and local operation temperature (attribute 20). At each time step, the agent takes an action according to the monitored state. The action space A = $\{a_0, a_1, a_2, \ldots, a_9\}$ contains the ten operation modes from which the routers can select.

The goal of the agent is to optimize its long-term *return*, which is represented by the discounted sum of future rewards. The return at time step *t* is defined as

$$\mathbb{R}_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots . \quad (1)$$

The variable γ (where $0 \leq \gamma \leq 1$) is a *discount rate*, which determines the impact of future rewards on the total return: as γ approaches 1, the agent becomes less near-sighted by giving more weight to future rewards.

In this paper, with the goal of simultaneously improve performance, energy-efficiency, and reliability, we design the reward function for router i at time step t as

$$r_{i,t} = -\log_{\alpha}(Latency_{i,t}) - \log_{\beta}(Power_{i,t}) - \log_{\lambda}(Aging_{i,t}). \tag{2}$$

The *Latency* refers to the average end-to-end latency of the specific router i, *Power* contains both static and dynamic power consumption. Additionally, the aging factor is calculated using the aging model, which is described in detail in Section 5.1. The α, β, and λ is used to emphasize the importance of each individual desired goal. In this paper, we set all three parameters, α, β, and λ, to 1.

Deep Q-Learning Approach. In DRL, a *model* of the environment, specified through a probability distribution $p(s_{t+1},\ r_{t+1}|s_t, a_t)$, characterizes how the state of the environment changes as a result of an agent action, and the reward that the agent receives after each action. Correspondingly, DRL agents compute an action-value function $Q^{\pi}(s,a)$ that estimates the return that they are expected to receive in this model of the environment if they start in state s, take action a, and follow the policy π for the remaining actions.

To find the optimal Q-value function *Q*(s, a)* that maximizes the expected return, we use the tabular Q-learning algorithm [18]. Q-values are initialized with zeros for all possible *(s, a)* pairs at the beginning. At each time step, the Q-learning algorithm chooses actions based on the current Q such that, over many time steps, all actions are taken in all states. After taking an action *a* and observing the reward *r* and new state s', the action-value entry *Q(s, a)* is changed using the following temporal difference rule:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha[r + \gamma\ \max_{a'} Q(s',a')]. \tag{3}$$

The learning rate α can be reduced over time and determines how well Q-learning will converge. It can be shown that for appropriate values of α, Q-learning converges to the optimal Q-value function Q* and its corresponding optimal policy π^* [18]. To explore unvisited regions of the state-action space, an ϵ-greedy policy is also applied to π^*, where agents also have a probability of ϵ to select a random action rather than always taking the action with the maximum Q-value [24].

Note that we intentionally design a reward function with a negative value to achieve a better high-level performance of RL. Because the Q-values of all the visited state-action pairs are negative and all the unvisited pairs have Q-values equal to zero, the RL agent will always select unvisited state-action pairs (zero is greater than any negative number). It allows the RL agents to explore the state-action pairs as much and as fast as possible, which leads to shorter convergence time for optimal decision making.

In conventional RL, the optimal policy π^* is recorded in a state-action mapping table called Q-table, where all Q-values are stored. As mentioned, each state vector consists of a number of features. When the RL agent observes any new state-action pair, it creates a new entry in the Q-table to record its actions and associated Q-values. Although the feature values are discretized into limited bins, the area consumption for the Q-table is excessive. To address this problem, we implement deep Q-learning by replacing the state-action table with an offline-trained artificial neural network (ANN) to reduce the hardware costs. The ANN calculates the state-action value rather than storing the entire state-action table in the router, thus eliminating the storage space for state-action pairs.

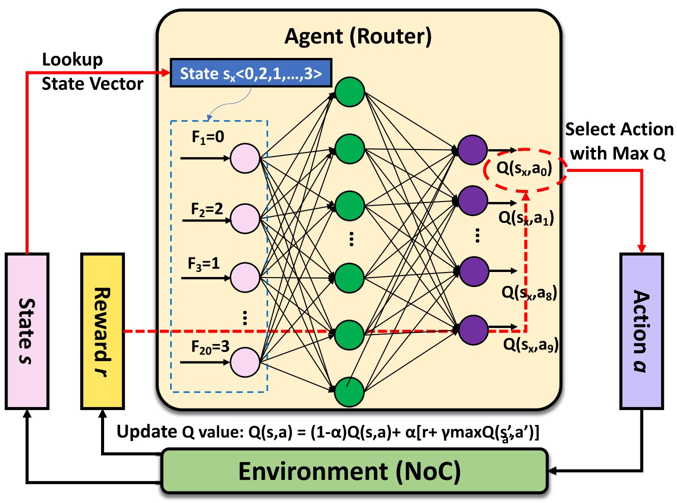


Fig. 10. Q-Learning process. For time step t, the action a_0 with the maximum Q-value of current state s_x is selected. The reward for action a_0 will be calculated after a_0 impacts the NoC environment. The Q-value will be updated following the Q-value updating rule. α is a learning rate , and γ is the discount rate.

In CURE, each ANN consists of 20 neurons at the input layer, 30 neurons at the hidden layer, and 10 neurons at the output layer. For training, we use a cycle-accurate network simulator to simulate the dynamic interactions and the online process of reinforcement learning. At this stage, Q-values of different state-action pairs are explored using real-world applications. All the visited state vectors and their corresponding Q-values (calculated with the reward function) are recorded as training sets for the ANN. Specifically, the attribute values of the state vector are ANN inputs, and the Q-values are the desired outputs. During each time step, similarly to updating the Q-value using reward values in conventional RL, the ANN follows the Q-value update rule and record ΔQ as the output error. Then, it uses mini-batch gradient descent to back propagate this error to the hidden layer to tune the weights. The training time is not counted to the timing overhead of the proposed NoC system. During the testing (or inference) phase, the ANN monitors the attribute values and uses the input values and weights to calculate 10 Q-values. The action with the highest Q-value will be selected.

The timing and area overheads of the proposed deep Q-learning are discussed in Section 6.4.

The Working of the Proposed DRL-based Controller. Fig. 10 demonstrates the working of the DRL-based control logic when running a benchmark. At each time step, the process goes through several stages. In the first stage, the router uses the feature values $F_1, F_2, \ldots, F_{20}$ in the state vector s as inputs of the ANN. In Fig. 10, we assume current state s matches state s_x in the mapping table. In the second stage, the ANN calculates the Q-values of all possible state-action pairs in the current state entry, and the router selects an action a, which has the maximum $Q(s,a)$-value for the next time step (we assume that a_0 in Fig. 10 has the maximum Q-value). Upon taking the action a, the NoC system transits to a new state s'. In the last stage of the current time step, the

TABLE 1
Simulation Environment Setup

# of cores	64 out-of-order CPUs @ 32 nm
Voltage and Frequency	1.0 Volt, 2.0 GHz
NoC Parameters	8 × 8 2D Mesh, 4-stage routers
Packet Size	4 flits
Flit Size	128 bits
Cycle Delay	4 cycle to L1 cache 8 cycle to L2 cache 160 cycle to main memory
Buffer Numbers* of Different Technologies	4RB-4VC-0CB (SECDED, SHIELD, Vicis) Avg. 8CB, 2 Directions (QORE) 2RB-4VC-8CB (IntelliNoC) Avg. 2RB-4VC-8CB, 2 Directions (CURE)

* *RB: router buffer, VC: virtual channel, and CB: channel buffer.*

NoC system provides a reward *r* (defined in 2) to the router. The reward will be used to update $Q(s,a)$. Each router will repeat all of these stages at each time step.

The initialization of the per-router controller and the state-action mapping table will be discussed in Section 5.2.

5 Evaluation Methodology

In this section, we present the fault injection model used in our experiments to inject timing errors into NoC, the simulation setup, and the benchmarks that we use. Faults that occur in the routing table and machine learning module are beyond the scope of this paper.

5.1 Fault Injection Models

To quantify the reliability improvement of the proposed designs for timing errors, we first create a transient error injection model to realistically produce a probability of timing errors for each link. The proposed transient fault injection model is a combination of the existing VARIUS [25] fault model and HotSpot [26] thermal model. At runtime, first, HotSpot uses the values of router supply voltage, operation frequency, and link utilization to obtain router operating temperature at runtime. The temperature values are fed into the VARIUS transient error model to generate the probability of timing errors (R_e) for each transmitted bit. Using R_e, the probability of which n-bit flit is faulty can be calculated as follows:

$$P_{fault} = 1 - (1 - R_e)^n. \qquad (4)$$

To assess the reliability improvement for permanent faults, we utilize the permanent fault model proposed in [8]. Specifically, we model and calculate the *aging* factor in (2) by correlating the shift in the threshold voltage of the transistor (ΔV_{th}). The ΔV_{th} is calculated using both ΔV_{th_NBTI} given by [27], [28] and ΔV_{th_HCI} given by [29], [30].

We model the *aging* factor given in (2) as follows:

$$\begin{cases} \Delta V_{th} = \Delta V_{th\_NBTI} + \Delta V_{th\_HCI} \\ Aging = 1 + \frac{\Delta V_{th}}{V_{th0}} \times 100\% \end{cases}. \qquad (5)$$

Note that the *aging* factor is designed to have a value greater than 1 such that it can be used in the reward function.

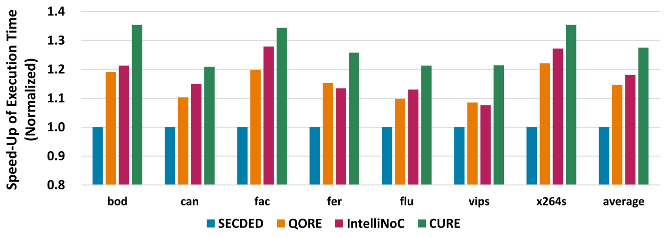


Fig. 11. Speed-up of full application execution time comparison, normalized to the SECDED baseline (higher is better).

5.2 Simulation Setup

We evaluate our proposed architecture using a modified version of the cycle-accurate network simulator *Booksim2* [21]. We also use Netrace [31] to capture cycle-accurate benchmark traces for the network simulator. Table 1 shows the simulation setup.

As mentioned, DRL parameters (α, γ, and ϵ) and ANN setups (e.g., neuron numbers) can impact the performance of DRL [32], [33]. In this paper, we set α, γ and ϵ to 0.1, 0.9 and 0.05, respectively. The operation modes of all routers are initialized to mode 1.

Workloads from PARSEC benchmarks [34] are tested. The benchmarks from PARSEC are transformed into trace files that contain trace format packet injection/ejection events and offer runtime information (such as time, packet size, transmission source, destination, and event type). Because CURE is the first NoC design framework that is fault-secure to both link failure and router failure, it is difficult to find a single state-of-the-art technology to compare with. Therefore, we conduct two sets of experiments to evaluate the performance of CURE in Section 6. In Section 6.1, we compare the performance of the proposed RL framework to the performances of the following state-of-the-art techniques: a static baseline using SECDED, QORE [7], and IntelliNoC [8] while link failures are injected. In Section 6.2, we compare CURE with SHIELD [3] and Vicis [4] while intra-router permanent faults are injected. For the RL-based IntelliNoC and DRL-based CURE, we train the per-router policy using a subset of PARSEC (blk, dedup, fre, and swa). Then, we use the remaining applications in PARSEC to test performance. The testing phase for each benchmark lasts a full application execution time. The control policy is dynamically updated by applying the temporal difference rule (3) every 1,000 cycles.

6 Evaluation Results and Analysis

6.1 Performance Analysis With Faulty Links

In this subsection, system-level performance metrics are evaluated when link failures are injected. Before runtime, permanent faults are randomly inserted into a percentage of links with a probability of 5 percent. At runtime, transient errors are injected into the links using the error injection model presented in Section 5. The evaluation results are presented below.

Execution Speed-up. The speed-up is obtained by calculating the ratio of the full application execution time of various techniques (baseline, QORE, and IntelliNoC) to the execution time using the proposed CURE technique running different benchmarks. The speed-up comparison is shown in Fig. 11. As shown in Fig. 11, CURE achieves the largest

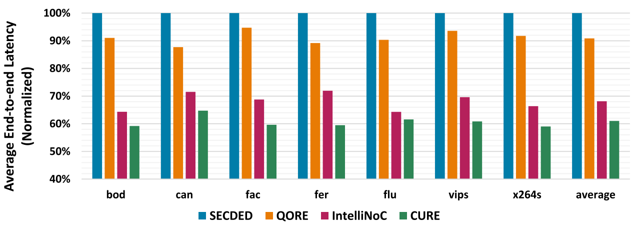


Fig. 12. Network latency comparison using average end-to-end latency, normalized to the SECDED baseline (lower is better).

speed-up of all evaluated techniques. QORE achieves a 14 percent speed-up over the SECDED baseline since the dual-direction links can improve the NoC throughput by dynamically allocating the limited link bandwidth for unbalanced traffic. IntelliNoC results in an average speed-up of 18 percent due to the ability to reduce ECC overhead and re-transmission traffic. However, in some benchmarks (i.e., fer and vips), IntelliNoC achieves worse performance than QORE. This is because IntelliNoC's fixed-direction MFAC can be the bottleneck when multiple link failures occur on the same channel. However, because CURE does not have such limitations, it successfully accelerates benchmark execution by 27 percent.

Network Latency. We define network latency as the average end-to-end latency of all transmitted packets of the full execution of each benchmark application. To measure the end-to-end latency, first, when a packet is injected from the source node, the injection time is recorded. Second, when the packet reaches the destination node and accepted by the destination node (after passing the error checking), the packet acceptance time is also recorded. The end-to-end latency for that packet is the time difference between injection time and acceptance time, which is recorded using the number of clock cycles. Fig. 12 shows the normalized network latency for different techniques. CURE achieves an average of end-to-end latency reduction of 39 percent. Note that QORE only achieves a 9 percent latency reduction over the baseline due to the timing overhead of channel buffer allocation and static error control scheme. Techniques with RL-based policies (i.e., IntelliNoC and CURE) both achieve latency reductions of over 30 percent, which implies that dynamic proactive policy such as the RL-based policy can minimize re-transmission and thus improve overall latency.

Energy-Efficiency. We define energy-efficiency as

$$Energy - E\ f\ ficiency = \left[\left(P_{static} + P_{dynamic}\right) \times T_{exec}\right]^{-1}. \tag{6}$$

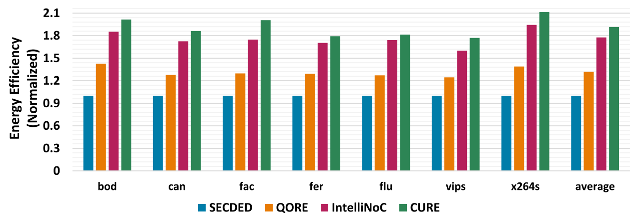


Fig. 13. Energy-efficiency comparison, normalized to the SECDED baseline (higher is better).

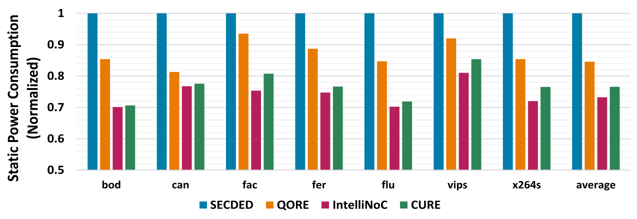


Fig. 14. Overall static power consumption comparison, normalized to the SECDED baseline (lower is better).

P_{static} and $P_{dynamic}$ are static and dynamic power consumption, respectively. We first model the static power with Synopsys Library Compiler for the designed NoC. Since Synopsys cannot evaluate the dynamic power accurately for different benchmark applications, we fed the static power parameters captured by Synopsys to DSENT [35] power model. During application execution, DSENT calculates the average dynamic power by the number of buffer writes, crossbar, and VA/SA activities within full application execution time. T_{exec} is the benchmark execution time. Fig. 13 shows the energy-efficiency measurements for all techniques studied and normalized to the SECDED baseline. CURE improves energy-efficiency by 92 percent compared to the baseline, while the energy-efficiency improvements using QORE and IntelliNoC are 36 and 77 percent, respectively.

Overall Static Power Consumption. Fig. 14 shows the overall static power consumption for the various techniques. QORE reduces static power consumption by 11 percent due to the elimination of router buffers. IntelliNoC achieves a static power reduction of 27 percent thanks to its power-gating and bypass scheme. Similar to IntelliNoC, the use of power-gating and bypass is beneficial in CURE. However, due to the extra logic and circuitry in the modified ECC and pipeline stages, the proposed CURE only achieves a static power reduction of 24 percent.

Overall Dynamic Power Consumption. As discussed, dynamic power consumption can be lowered by reducing the number of router buffers and/or by mitigating faults and reducing the number of re-transmissions. CURE, using RMC and dynamic error control, is able to significantly reduce re-transmission traffic. Consequently, CURE outperforms all other techniques in reducing dynamic power consumption as shown in Fig. 15.

6.2 Performance Analysis With Intra-Router Failures

In this subsection, permanent faults are injected into the router circuit (including VA, SA, crossbar, and ECC hardware). Before runtime, faults are randomly inserted into a

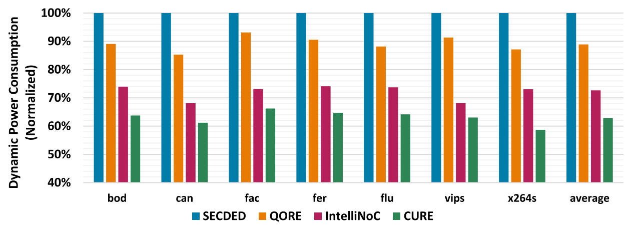


Fig. 15. Overall dynamic power consumption comparison, normalized to the SECDED baseline (lower is better).

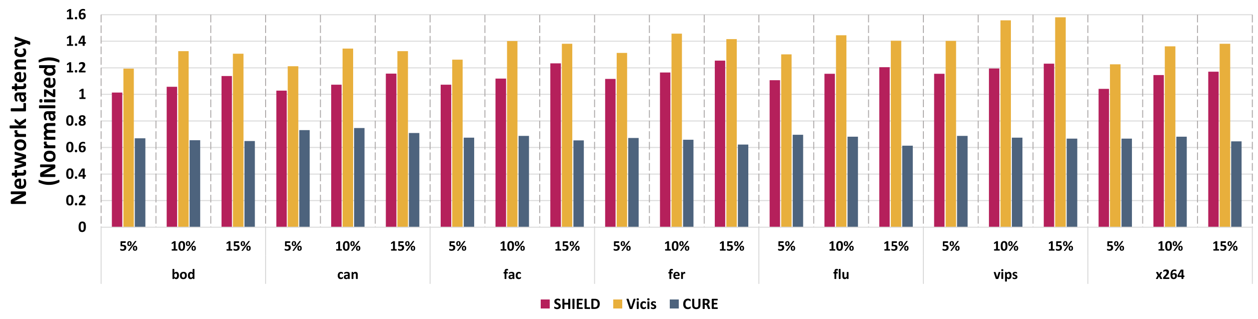


Fig. 16. Network latency comparison using average end-to-end latency, normalized to the SECDED baseline (lower is better). Each benchmark is tested under 5, 10, and 15 percent permanent fault rates in router.

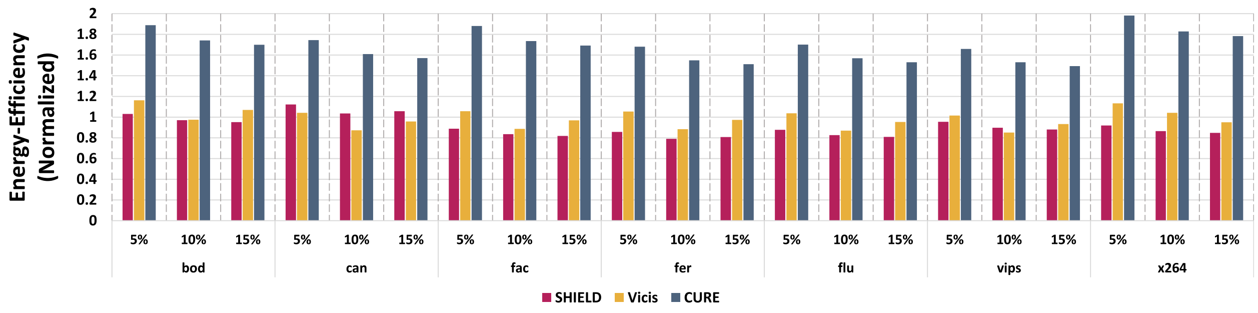


Fig. 17. Energy-efficiency comparison, normalized to the SECDED baseline (higher is better). Each benchmark is tested under 5, 10, and 15 percent permanent fault rates in router.

percentage of the router components ranging from 5 to 15 percent. At runtime, transient faults are injected only to the links using the proposed error injection model. The transient fault-handling technique used by SHIELD and Vicis is static SECDED, while CURE utilizes the proposed adaptive error control hardware. The network latency and energy efficiency of different techniques are evaluated, and the results are normalized to the static SECDED baseline. Because the SECDED baseline cannot tolerate router failures, the intra-router error rate is set to 0 percent for the baseline.

Network Latency. Fig. 16 shows the normalized network latency for different techniques at various error rates. The proposed CURE framework achieves at least 30 percent network latency reduction under 5, 10, and 15 percent router failure rates. However, both of the other fault-tolerant techniques incur excessive network latency.

Energy-Efficiency. Fig. 17 shows the normalized energy-efficiency for different techniques. The energy-efficiency of all the techniques tends to decrease as the error rate increases. However, there are some exceptions to Vicis. When the error rate exceeds 10 percent, Vicis re-configures itself less often, which can significantly reduce execution time. As shown in the figure, the proposed CURE framework achieves the highest energy-efficiency among all the other techniques. This result implies that the use of channel buffers and adaptive error control hardware can substantially reduce power and execution time.

6.3 Reliability Improvement Analysis

We use mean-time-to-failure (MTTF) to evaluate the reliability of the proposed design for permanent faults [36], [37]. Fig. 18 shows the normalized MTTF comparison. As shown in Fig. 18, the proposed CURE framework is 7.7× more reliable than the baseline router which is unprotected from permanent faults, while the highest normalized MTTF value of other fault-tolerant designs is 4.6× (SHIELD). This improvement is achieved thanks to the reduced operation temperature by the RL-based control policy and the modified pipeline and ECC hardware in each router.

6.4 Overhead Analysis

We evaluated the area overhead of each technique with Synopsys and 32 nm technology library with the supply voltage set to 1.0 Volt, and clock frequency set to 2.0 GHz. The area overhead is shown in Table 2.

As shown in this table, IntelliNoC, QORE, and CURE consume less area than the baseline due to the use of link storage. Due to the additional circuitry design, Vicis has the largest area overhead. CURE and IntelliNoC incur additional area overheads thanks to the RL-based controlling modules. The area overhead of the DRL-based controller

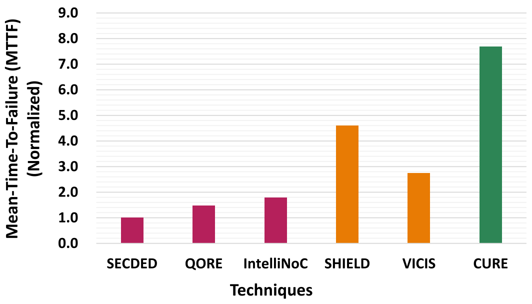


Fig. 18. Mean-time-to-failure (MTTF) comparison, normalized to the SECDED baseline (higher is better).

TABLE 2
Area Overhead Comparison (μm^2)*

	Baseline	QORE	SHIELD	Vicis	IntelliNoC	CURE
Router Buffer	1248.3 ×16 /port	-	1248.3 ×16 /port	1248.3 ×16 /port	1248.3 ×8 /port	1248.3 ×8 /port
Control logic & Crossbar	9004.7	9662.4	12066.3	41712.0	9004.7	11613.6
Channel	136.7	5948.4	136.7	136.7	2869.6	2974.2
ECC	3325.4	3325.4	3325.4	3325.4	3940.3	4058.1
Total	119807.0	79764.4	83953.1	152514.3	89313.7	89256.2
%Change	-	-33.4%	+2.6%	+27.3%	-25.4%	-25.5%

includes the area consumption of both ALUs (adder, multiplier, and Sigmoid function) and SRAM for calculating, updating, and storing the ANN. The simulation shows that the area overhead for ALUs and SRAM are 992.2μm^2 and 838.6μm^2, respectively. This implies 0.8 and 0.7 percent of the area consumption of the baseline router. Therefore, the area cost is reduced as compared to the 4 percent area overhead of table-based RL in previous designs. Moreover, CURE requires additional timing overhead for calculating the Q-values using ANN. Using [38], it shows that at each time step, the timing overhead of CURE is estimated to be 160 ns. Using the similar method as [39], this latency can be overlapped by a large time step. Specifically, we use two sets of different intervals for monitoring the attributes and the controlling to minimize the negative effect of this latency. The two sets of intervals are offset by the ANN computation time, which can pipeline the overhead effectively. By doing so, the ANN control computing does not block either the monitoring process or the controlling. Therefore, the use of ANN will not negatively impact the overall performance metrics. Additionally, the timing overheads for the MFAC (IntelliNoC) and RMC (CURE) configurations are both estimated around 45 to 50 cycles. Furthermore, The control logic of the proposed DRL consumes an additional 0.26 mw power, which implies 4 percent of the static power consumption of the baseline router.

6.5 Sensitivity Study

Impact of Input Data Size. In this test, we explore the impact of different input data sizes on network latency and energy-delay product (EDP), and the blackscholes application in the PARSEC benchmark is used. Lower network latency and EDP indicate better system performance. The evaluation result illustrated in Fig. 19 shows that as input data size increases, the performance metrics remain the same. This is because the traffic patterns, network intensity, and workload allocations for the same benchmark processing phase are the same, for different data input sizes. However, by reducing the NoC size to a 4×4 mesh, which incurs a higher network intensity, the proposed adaptive ECC hardware can achieve better network performance, as compared to the static SECDED baseline.

Impact of Time Step Length. In this test, we varied the time step t starting from 200 to 10,000 clock cycles. The evaluation results using the blackscholes application are illustrated in Fig. 20a. As shown in Fig. 20a, a longer cycle time (10K cycles) has a negative impact on performance due to coarse-grain control. Meanwhile, aggressively reducing the length of time steps (200 clock cycles) also leads to a degradation in performance because the computational overhead of DRL-based control policy will dominate performance.

Impact of Discount Rate γ. Fig. 20b indicates the impact of the discount rate γ on network latency and EDP. The blackscholes application is used in this test. As shown in Fig. 20b, the network latency and EDP initially improve with larger γ, yet aggressively increasing γ can also lead to Q-learning failing to converge, which negatively affects the system performance. The best performance is achieved when γ equals 0.9.

Impact of Exploration Probability ϵ. Fig. 20c shows the impact of ϵ values on network latency and EDP, using the blackscholes application. When ϵ is 0, the router always selects the initial mode most of the time. As ϵ increases from 0 to 1, the router tends to explore new state-action pairs more frequently. Further, when ϵ equals 1, the router will take actions entirely at random. As shown in Fig. 20c, the best system performance is achieved when ϵ equals 0.05.

Impact of the Hidden Layer Size of the ANN. Because the number of neurons used in the hidden layer of the ANN can affect the accuracy of calculating $Q(s, a)$, thereby impacting the decision making of the DRL controller, we vary the size of

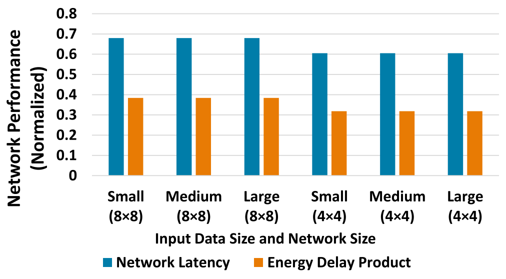


Fig. 19. Impact of different input data sizes and different NoC sizes.

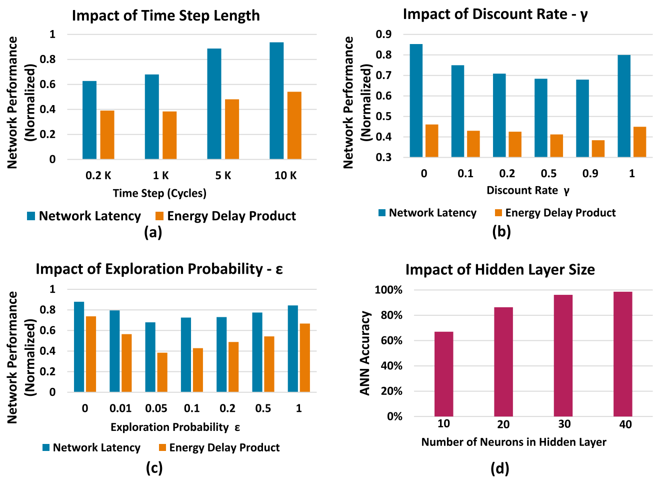


Fig. 20. Impact of (a) time step length, (b)discount rate γ, (c) exploration probability ϵ, and (d) hidden layer size of of DQL on network performance metrics. Results are normalized to the baseline.

the hidden layer to study its impact on calculation accuracy, as shown in Fig. 20d. Fig. 20d shows that the calculation accuracy improves as the hidden layer size increases. To save area consumption and reduce the timing overhead of the ANN, we select the hidden layer of 30 neurons.

7 Related Works

There has been considerable work in improving the energy-efficiency and reliability in NoCs. In the following, we briefly highlight some of the directly relevant works.

Power-Saving Designs for NoCs. Because static power consumption has become a substantial portion of overall network power, power-gating (PG) techniques that power off underutilized network components have been shown to be effective for static power savings [14], [15], [40]. However, conventional power-gating schemes for routers tend to substantially increase network latency due to a reduced number of active routers in the network and extra control overhead in managing power-gating. Another approach proposed for reducing network power is reducing router buffers. Michelogiannakis and Dally [11] and Kodi *et al.* [7], [12]have shown that eliminating router buffers is beneficial for both static and dynamic power reduction. However, simply replacing router buffers with channel buffers leads to penalties in network congestion and latency [7], [11], [12].

Fault-Tolerant Designs for NoCs. In NoC, both transient and permanent faults can manifest during transmission. CRC [6] is a basic transient fault detection technique that is often used for NoCs. Flits are encoded by a local CRC encoder in the router before transmission, and are decoded by the destination CRC decoder to perform error detection. If the destination router detects errors, a re-transmission request is sent to the source router to re-transmit the flit. To mitigate transient faults, per-hop error correction codes (ECCs) are generally deployed. SECDED is one of the most commonly used ECC techniques in NoCs [4]. To handle permanent faults caused by transistor aging [29], a number of techniques have been proposed using load-balancing [7], circuitry redundancy [37], and adaptive routing techniques [4] among others. Note that most of the techniques are static in nature, with CRCs or SECDEDs being deployed all the time, regardless of whether faults are present. Reliability-enhancement mechanisms based on static techniques have been shown to require excessive power consumption, and longer delays, thereby significantly degrading NoC performance [17], [37], [41].

Learning-Enabled NoC Designs. Multiple machine learning techniques have been introduced to balance design trade-off or predict traffic in NoCs. These works target achieving high power efficiency [8], [16], [24], [39], [42], [43], [44], [45], and enable fault-tolerant design [5], [8], [45]. For example, [44] discovers that the wake-up latency of PG and performance degradation of using DVFS are the bottlenecks of implementing PG and DVFS simultaneously. Hao *et al.* [44] have shown that applying machine learning to handle the dynamic trade-offs of DVFS and PG can achieve optimal power savings. [45] introduces a proactive fault-tolerant mechanism to optimize energy efficiency and performance with reinforcement learning (RL). Further, [8] proposes to use channel buffers to achieve higher power savings in addition to [45]. We extend these existing works by proposing link reversibility, adding hard error tolerance, and using DRL to reduce control overhead.

8 Conclusion

In this paper, we propose *CURE*, a learning-based NoC design that can simultaneously improve performance, energy-efficiency, and reliability. CURE consists of reversible multi-function adaptive channel, enhanced fault-tolerant router circuitry, ten unique operation modes, and a DRL-based dynamic control policy. With DRL, each router learns from the NoC behavior and updates a control policy to select an optimal operating mode at any given time. The experimental results illustrate that CURE decreases network latency by 39 percent, improves energy efficiency by 92 percent over the static SECDED baseline. Using the mean-time-to-failure metric, we show that the proposed framework is 7.7$\times$ more reliable than the baseline NoC architecture.

Acknowledgments

This research was supported in part by NSF Grants CCF-1420718, CCF1513606, CCF-1703013, CCF-1547034, CCF-1547035, CCF-1540736, and CCF-1702980. The authors would like to sincerely thank the anonymous reviewers for their excellent feedback.

References

[1] L. Benini and G. De Micheli, “Networks on chips: A new SoC paradigm,” *Computer*, vol. 35, no. 1, pp. 70–78, 2002.
[2] W. J. Dally and B. Towles, “Route packets, not wires: On-chip interconnection networks,” in *Proc. 38th Annu. Des. Autom. Conf.*, 2001, pp. 684–689.
[3] P. Poluri and A. Louri, “Shield: A reliable network-on-chip router architecture for chip multiprocessors,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 10, pp. 3058–3070, Oct. 2016.
[4] D. Fick, A. DeOrio, J. Hu, V. Bertacco, D. Blaauw, and D. Sylvester, “Vicis: A reliable network for unreliable silicon,” in *Proc. 46th ACM/EDAC/IEEE Annu. Des. Autom. Conf.*, 2009, pp. 812–817.
[5] D. DiTomaso, T. Boraten, A. Kodi, and A. Louri, “Dynamic error mitigation in NoCs using intelligent prediction techniques,” in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2016, pp. 1–12.
[6] S. Lin and D. J. Costello, *Error Control Coding*, 2nd ed. NJ, USA: Prentice Hall, 2004.
[7] D. DiTomaso, A. Kodi, and A. Louri, “QORE: A fault tolerant network-on-chip architecture with power-efficient quad-function channel (QFC) buffers,” in *Proc. 20th Int. Symp. High Perform. Comput. Archit.*, 2014, pp. 320–331.
[8] K. Wang, A. Louri, A. Karanth, and R. Bunescu, “IntelliNoC: A holistic design framework for energy-efficient and reliable on-chip communication for manycores,” in *Proc. 46th Int. Symp. Comput. Archit.*, 2019, pp. 589–600.
[9] K. Aisopos, A. DeOrio, L.-S. Peh, and V. Bertacco, “ARIADNE: Agnostic reconfiguration in a disconnected network environment,” in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2011, pp. 298–309.
[10] J. Kim, C. Nicopoulos, D. Park, V. Narayanan, M. S. Yousif, and C. R. Das, “A gracefully degrading and energy-efficient modular router architecture for on-chip networks,” *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 2, pp. 4–15, 2006.
[11] G. Michelogiannakis and W. J. Dally, “Elastic buffer flow control for on-chip networks,” *IEEE Trans. Comput.*, vol. 62, no. 2, pp. 295–309, Feb. 2013.
[12] A. K. Kodi, A. Sarathy, and A. Louri, “iDEAL: Inter-router dual-function energy and area-efficient links for network-on-chip (NoC) architectures,” *ACM SIGARCH Comput. Archit. News*, vol. 36, no. 3, pp. 241–250, 2008.
[13] H. Matsutani, M. Koibuchi, D. Ikebuchi, K. Usami, H. Nakamura, and H. Amano, “Ultra fine-grained run-time power gating of on-chip routers for CMPs,” in *Proc. 4th ACM/IEEE Int. Symp. Netw.-on-Chip*, 2010, pp. 61–68.

[14] R. Das, S. Narayanasamy, S. K. Satpathy, and R. G. Dreslinski, "Catnap: Energy proportional multiple network-on-chip," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 320–331, 2013.

[15] A. Samih, R. Wang, A. Krishna, C. Maciocco, C. Tai, and Y. Solihin, "Energy-efficient interconnect via router parking," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit.*, 2013, pp. 508–519. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2013.6522345

[16] M. Clark, A. Kodi, R. Bunescu, and A. Louri, "LEAD: Learning-enabled energy-aware dynamic voltage/frequency scaling in NoCs," in *Proc. 55th ACM/ESDA/IEEE Des. Autom. Conf.*, 2018, pp. 82:1–82:6. [Online]. Available: http://doi.acm.org/10.1145/3195970.3196068

[17] Y. Chen, M. F. Reza, and A. Louri, "DEC-NoC: An approximate framework based on dynamic error control with applications to energy-efficient NoCs," in *Proc. IEEE 36th Int. Conf. Comput. Des.*, 2018, pp. 480–487.

[18] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.

[19] L. Busoniu, R. Babuska, and B. De Schutter, "Multi-agent reinforcement learning: A survey," in *Proc. 9th Int. Conf. Control Autom. Robot. Vis.*, 2006, pp. 1–6.

[20] V. François-Lavet *et al.*, "An introduction to deep reinforcement learning," *Found. Trends® Mach. Learn.*, vol. 11, no. 3/4, pp. 219–354, 2018.

[21] N. Jiang *et al.*, "A detailed and flexible cycle-accurate network-on-chip simulator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2013, pp. 86–96.

[22] H. Naeimi and A. DeHon, "Fault secure encoder and decoder for nanomemory applications," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 4, pp. 473–486, Apr. 2009.

[23] S. Lin and D. J. Costello, *Error Control Coding*, 2nd ed. Upper Saddle River, NJ, USA: Prentice-Hall, 2004.

[24] Q. Fettes, M. Clark, R. Bunescu, A. Karanth, and A. Louri, "Dynamic voltage and frequency scaling in NoCs with supervised and reinforcement learning techniques," *IEEE Trans. Comput.*, vol. 68, no. 3, pp. 375–389, Mar. 2019.

[25] S. R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas, "VARIUS: A model of process variation and resulting timing errors for microarchitects," *IEEE Trans. Semicond. Manuf.*, vol. 21, no. 1, pp. 3–13, Feb. 2008.

[26] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. R. Stan, "HotSpot: A compact thermal modeling methodology for early-stage vlsi design," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 14, no. 5, pp. 501–513, May 2006.

[27] M. A. Alam and S. Mahapatra, "A comprehensive model of PMOS NBTI degradation," *Microelectronics Rel.*, vol. 45, no. 1, pp. 71–81, 2005.

[28] S. Bhardwaj, W. Wang, R. Vattikonda, Y. Cao, and S. Vrudhula, "Predictive modeling of the NBTI effect for reliable design," in *Proc. Custom Integr. Circuits Conf.*, 2006, pp. 189–192.

[29] D. Lorenz, G. Georgakos, and U. Schlichtmann, "Aging analysis of circuit timing considering NBTI and HCI," in *Proc. 15th IEEE Int. On-Line Testing Symp.*, 2009, pp. 3–8.

[30] H. Kim, A. Vitkovskiy, P. V. Gratz, and V. Soteriou, "Use it or lose it: Wear-out and lifetime in future chip multiprocessors," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2013, pp. 136–147.

[31] J. Hestness, B. Grot, and S. W. Keckler, "Netrace: Dependency-driven trace-based network-on-chip simulation," in *Proc. 3rd Int. Workshop Netw. Chip Archit.*, 2010, pp. 31–36.

[32] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," *ACM SIGARCH Comput. Archit. News*, vol. 36, no. 3, pp. 39–50, 2008.

[33] Y. Bai, V. W. Lee, and E. Ipek, "Voltage regulator efficiency aware power management," in *Proc. 22nd Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2017, pp. 825–838.

[34] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors," in *Proc. 5th Annu. Workshop Model. Benchmarking Simul.*, 2009, vol. 2011, p. 37.

[35] C. Sun *et al.*, "DSENT-A tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling," in *Proc. IEEE/ACM 6th Int. Symp. Netw.-on-Chip*, 2012, pp. 201–210.

[36] J. Shin, V. Zyuban, Z. Hu, J. A. Rivers, and P. Bose, "A framework for architecture-level lifetime reliability modeling," in *Proc. 37th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2007, pp. 534–543.

[37] K. Constantinides *et al.*, "BulletProof: A defect-tolerant CMP switch architecture," in *Proc. 12th Int. Symp. High-Perform. Comput. Archit.*, 2006, pp. 5–16.

[38] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *Proc. IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers*, 2014, pp. 10–14.

[39] J.-Y. Won, X. Chen, P. Gratz, J. Hu, and V. Soteriou, "Up by their bootstraps: Online learning in artificial neural networks for CMP uncore power management," in *Proc. 20th Int. Symp. High Perform. Comput. Archit.*, 2014, pp. 308–319.

[40] L. Chen and T. M. Pinkston, "NoRD: Node-router decoupling for effective power-gating of on-chip routers," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2012, pp. 270–281.

[41] M. Koibuchi, H. Matsutani, H. Amano, and T. M. Pinkston, "A lightweight fault-tolerant mechanism for network-on-chip," in *Proc. 2nd ACM/IEEE Int. Symp. Netw.-on-Chip*, 2008, pp. 13–22.

[42] S. Van Winkle, A. K. Kodi, R. Bunescu, and A. Louri, "Extending the power-efficiency and performance of photonic interconnects for heterogeneous multicores with machine learning," in *Proc. 24th IEEE Int. Symp. High-Perform. Comput. Archit.*, 2018, pp. 480–491.

[43] D. DiTomaso, A. Sikder, A. Kodi, and A. Louri, "Machine learning enabled power-aware network-on-chip design," in *Proc. Des. Autom. Test Eur. Conf.*, 2017, pp. 1354–1359.

[44] H. Zheng and A. Louri, "An energy-efficient network-on-chip design using reinforcement learning," in *Proc. 56th Annu. Des. Autom. Conf.*, 2019, pp. 47:1–47:6. [Online]. Available: http://doi.acm.org/10.1145/3316781.3317768

[45] K. Wang, A. Louri, A. Karanth, and R. Bunescu, "High-performance, energy-efficient, and fault-tolerant network-on-chip design using reinforcement learning," in *Proc. Des. Autom. Test Europe Conf. Exhib.*, 2019, pp. 1166–1171.

Ke Wang (Student Member, IEEE) received the BS degree in electrical engineering from Peking University, Beijing, China, in 2013, and the MS degree in electrical engineering from Worcester Polytechnic Institute, Worcester, Massachusetts, in 2015. He is currently working toward the PhD degree in computer engineering in the School of Engineering and Applied Science, George Washington University, Washington, DC. His research work focuses on optimized NoC design of high performance, power efficiency, and reliability using machine learning.

Ahmed Louri (Fellow, IEEE) received the PhD degree in computer engineering from the University of Southern California, Los Angeles, California, in 1988. He is the David and Marilyn Karlgaard Endowed chair professor of electrical and computer engineering with the George Washington University, Washington, DC, and the director of the High Performance Computing Architectures and Technologies Laboratory. From 2010 to 2013, he served as a program director with the National Science Foundations (NSF) Directorate for computer and information science and engineering. He conducts research in the broad area of computer architecture and parallel computing, with emphasis on interconnection networks, optical interconnects for scalable parallel computing systems, reconfigurable computing systems, and power-efficient and reliable Network-on-Chips (NoCs) for multicore architectures. He is currently serving as the editor-in-chief of the *IEEE Transactions on Computers*.

Heterogeneous Edge Offloading With Incomplete Information: A Minority Game Approach

Miao Hu, *Member, IEEE*, Zixuan Xie, Di Wu, *Senior Member, IEEE*, Yipeng Zhou, Xu Chen, and Liang Xiao, *Senior Member, IEEE*

Abstract—Task offloading is one of key operations in edge computing, which is essential for reducing the latency of task processing and boosting the capacity of end devices. However, the heterogeneity among tasks generated by various users makes it challenging to design efficient task offloading algorithms. In addition, the assumption of complete information for offloading decision-making does not always hold in a distributed edge computing environment. In this article, we formulate the problem of heterogeneous task offloading in a distributed environment as a minority game (MG), in which each player must make decisions independently in each turn and the players who end up on the minority side win. The multi-player MG incentivizes players to cooperate with each other in the scenarios with incomplete information, where players don't have full information about other players (e.g., the number of tasks, the required resources). To address the challenges incurred by task heterogeneity and the divergence of naive MG approaches, we propose an MG based scheme, in which tasks are divided into subtasks and instructed to form into a set of groups as possible, and the left ones are scheduled to perform decision adjustment in a probabilistic manner. We prove that our proposed algorithm can converge to a near-optimal point, and also investigate its stability and price of anarchy in terms of task processing time. Finally, we conduct a series of simulations to evaluate the effectiveness of our proposed scheme and the results indicate that our scheme can achieve around 30% reduction of task processing time compared with other approaches. Moreover, our proposed scheme can converge to a near-optimal point, which cannot be guaranteed by naive MG approaches.

Index Terms—Heterogeneous edge offloading, incomplete information, minority game

1 Introduction

Edge computing is a new computing paradigm that allows data produced by end devices (e.g., vehicles, cameras, IoT devices) to be processed at the edge of the network where the data is being generated, instead of sending it back to the cloud (or data center) along long routes. By exploiting computation resources of edge nodes to process offloaded tasks, edge computing can provide real-time local data analytics capability, which cannot be offered by the cloud. As an example, self-driving cars are commonly equipped with a multitude of sensory devices, including cameras, LIDAR, sonar devices, and so on. For these cars to operate safely, it is essential to process the huge amount of collected data (e.g., over 1.38 GB per second for a self-driving car [1]) in a timely manner. The latency in processing speed is critical for vehicle safety, but transmitting such a large volume of data back-and-forth over a network is very time-consuming. One feasible solution is to process the sensory data at the edge, so that the cars can react to their surroundings right away and take driving actions accordingly [2]. Although at a nascent stage, edge computing has proven its superiority in quite a few areas. According to the report by CB Insights[1], the global market of edge computing is estimated to reach $6.72 billion by 2022.

It should be pointed out that the edge and the cloud are complementary components of a whole computing system. Compared to high-end cloud servers, edge servers (or edge nodes) in the proximity commonly have much lower capacity. The edge needs to work with the cloud in tandem to process tasks offloaded by end users more efficiently [3], [4], [5]. However, it is challenging to determine how to conduct efficient task offloading in a distributed environment, where cloud (or edge) servers are physically distributed in different locations. The reasons are multi-fold: first, edge servers have limited processing capacity and transmission bandwidth, and an improper offloading decision may overload edge servers and cause wastage of CPU cycles; second, given that complete information is not always available in a distributed environment, it is difficult to find the optimal offloading scheme. We need to design offloading algorithms that only require local and incomplete information, especially for users with their own interests.

From the perspective of any end user, its objective is to minimize the amount of processing time for its tasks. The user needs to determine whether to conduct task offloading, where to offload the task (e.g., to the edge or the cloud), the

- *M. Hu, Z. Xie, D. Wu, and X. Chen are with the School of Data and Computer Science, Sun Yat-sen University, Guangzhou, Guangdong 510275, China, and also with the Guangdong Key Laboratory of Big Data Analysis and Processing, Guangzhou, Guangdong 510006, China. E-mail: humiao@outlook.com, xiezx6@mail2.sysu.edu.cn, {wudi27, chenxu35}@mail.sysu.edu.cn.*
- *Y. Zhou is with the Department of Computing, Faculty of Science and Engineering, Macquarie University, Sydney, NSW 2109, Australia, and also with Peng Cheng Laboratory, Shenzhen, Guangdong 518000, China. E-mail: yipeng.zhou@mq.edu.au.*
- *L. Xiao is with the Department of Communication Engineering, Xiamen University, Xiamen, Fujian 361005, China. E-mail: adalittlel@gmail.com.*

Manuscript received 25 June 2019; revised 2 Mar. 2020; accepted 11 Apr. 2020. Date of publication 17 Apr. 2020; date of current version 1 May 2020. (Corresponding author: Di Wu.)
Recommended for acceptance by R. Yahyapour.
Digital Object Identifier no. 10.1109/TPDS.2020.2988161

amount of offloaded tasks, and so on. However, as complete information is not available, each user has no idea about decisions made by others when making its own decision. Therefore, they have to make offloading decisions independently, which may cause unexpected results. It is possible that too many tasks are assigned to the same edge server and the processing time is significantly prolonged due to server overloading. In another extreme, if too few tasks are assigned to an edge server, it will lead to low execution efficiency of edge resources.

Due to the importance of task offloading in edge computing, researchers have made significant efforts in this field. In most of previous studies, the scheduler of task offloading is very aggressive in collecting task-related information. It is based on the assumption that a better offloading decision can be made with more information. Centralized schedulers proposed by [6], [7], [8], [9], [10] extensively collected information from end users before making offloading decisions. Due to the high volume of message traffic, the above centralized approaches cannot work well in a large-scale distributed environment. Some work [11], [12] started to investigate the design of distributed schedulers for task offloading in edge computing systems. It normally requires pair-wise information exchange among all users to obtain a complete view of the system. Inevitably, the communication overhead will increase considerably with the increase of user population in the system. The problem will be further complicated by the loss of direct communication between two users. In the real-world environment, it is more common that each user needs to make decisions with partial information.

In this paper, we formulate the problem of heterogeneous task offloading in a distributed environment as a multi-player game, in which complete information is not required by the scheduler to make offloading decisions. We apply the theory of minority game (MG) to design the scheduling algorithm for task offloading. MG is efficient in modeling collective behaviors of users where they have to compete for limited resources without complete information [13]. However, a naive MG solution cannot guarantee to converge to the optimal point and also is not applicable in the case with heterogeneous task loads. To overcome the above obstacles, we propose an MG based scheme for the heterogeneous edge computing scenario, in which tasks generated by the same user are divided into subtasks and instructed to form into groups, and compete for computation resources in a batch in order to minimize the task processing time. The left subtasks are scheduled to make their offloading decisions in a probabilistic manner iteratively. We prove that our proposed algorithm can converge to a near-optimal point, and also investigate its stability and price of anarchy in terms of task processing time. The communication overhead is low as the scheduler does not need to collect complete information for decision-making.

In summary, our contributions in this paper can be concluded as below:

- We formulate the problem of heterogeneous task offloading in the distributed edge computing environment as a multi-player game, in which end users are modeled as players, and each user chooses to make offloading decisions independently with incomplete information.
- We propose an MG based scheme for efficient task offloading in heterogeneous edge computing, in which groups are formed among subtasks to compete resources with others. For subtasks not included in any group, they are scheduled by a probabilistic decision adjustment policy. We further prove that the proposed scheme can converge to a near-optimal point, and an expression of the gap is derived following derivations on attendance and price of anarchy.
- We conduct a series of simulations to evaluate the effectiveness of our proposed scheme and the results indicate that our scheme can achieve around 30% reduction of task processing time compared with other approaches. Moreover, our scheme is verified that can converge to a near-optimal point.

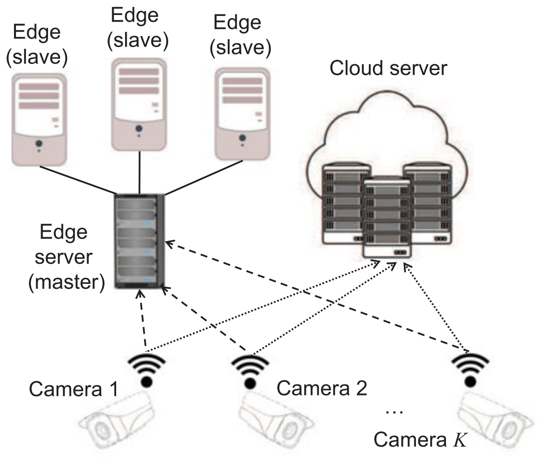


Fig. 1. A partially distributed architecture of a heterogeneous edge computing system.

The remainder of this paper is organized as follows. Section 2 introduces the system model and formulates a task offloading game with incomplete information. The MG based offloading framework and properties are introduced in Section 3. An extension to a fully distributed edge computing architecture is presented in Section 4. Experimental results are shown in Section 5, and some related studies are summarized in Section 6. Finally, Section 7 concludes our work and points out possible future work.

2 System Model and Game Formulation

In this section, we introduce the system model and formulate the problem of heterogeneous task offloading in a distributed environment as a multi-player game with incomplete information.

2.1 System Model

Fig. 1 shows a typical heterogeneous edge computing system for processing video streams captured by cameras, where the video analytics tasks (e.g., object detection, tracking, recognition) have different resource requirements [14]. In such a system, there exist multiple edge servers with different capacity (e.g., CPU, memory), which work together with remote clouds for task execution. We first consider a partially distributed architecture, in which all edge servers

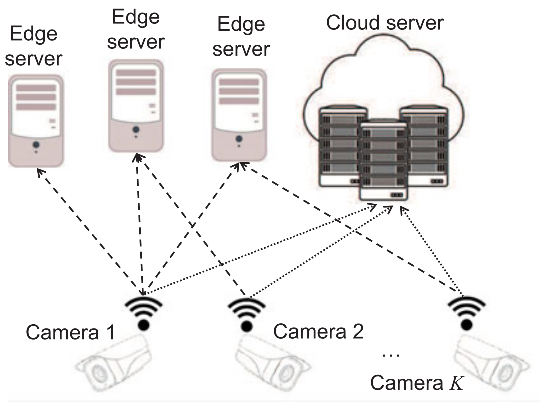


Fig. 2. A fully distributed architecture of a heterogeneous edge computing system.

can be abstracted as a single cluster and the master edge server is the entry point of task offloading, as illustrated in Fig. 1. Such a partially distributed architecture has been proven to be effective in managing large-scale networked systems (e.g., SDN controllers, Kubernetes). Anyway, the whole edge computing system can also be organized in a fully distributed way (as illustrated in Fig. 2).

In Fig. 1, end users can choose to offload their tasks to either cloud servers or edge servers. The internal scheduling of task offloading within the edge server cluster is transparent to end users. Our main contribution in this architecture is to determine which side (i.e., edge server side, cloud server side) should be chosen when only incomplete information is available. The extension to a fully distributed architecture will be discussed in Section 4.

Suppose that there are K users (e.g., cameras) in the system, each of which generates tasks with different amounts of computation load. Each task is further divided into a set of *subtasks* with equal size in terms of computation load. For a parallelizable task focused in this paper, it can be divided into a set of subtasks that can be run in parallel. A typical example is video analytics task, in which a video can be divided into a series of frames and the video analytics task (e.g., object detection) can be conducted on each frame separately [15], [16]. The size of a subtask is denoted by C_0 in the unit of CPU cycles. Let R_k be the computation requirement of a user k, and N_k be the number of subtasks generated by the user k. Thus, we have $R_k = N_k C_0$. In one decision period, all users generate $N = \sum_k N_k$ subtasks in total. Similar to many previous studies (e.g., [11] and [12]), to enable tractable analysis and get useful insights, we consider a quasi-static scenario where the set of end users remains unchanged during a computation offloading period (e.g., several hundred milliseconds). For a better understanding of the work, we list the key defined symbols in Table 1.

As the key performance metric, the task processing time consists of task delivery time and task execution time. The task delivery time from the user to the server is influenced by the communication model. While task execution time refers to the time to process a task on the server, which is mainly affected by the computation model. In the following, we will introduce the communication model and the computation model, respectively.

(1) Communication Model: To analyze our scheme in the wireless environment, we introduce a wireless communication model [17] into our system model, based on which we derive the user uplink rate and transmission delay. The wireless communication model has been widely adopted in the previous literature (e.g., [12], [17], [18]). According to the model, with the channel state information at the user side, we have the uplink capacity from user k to server m as:

$$C_{m,k}^{\mathrm{UL}} = B \log_2(1 + \mathrm{SNR}_k), \tag{1}$$

where B represents the transceiver bandwidth, and $\mathrm{SNR}_k = |h_{m,k}|^2 P_k / \sigma^2$ denotes the signal to noise ratio (SNR) for user k, where $|h_{m,k}|^2$ is the gain of the channel between the transmitter k and the base station for server m and P_k represents the transmission power of user k. The uplink capacity $C_{m,k}^{\mathrm{UL}}$ can be achieved by certain physical layer technologies, which has been commonly assumed in many previous studies, e.g., [19], [20], [21]. For the partially distributed case, the uplink rate is denoted as $C_{\mathrm{e},k}^{\mathrm{UL}}$ and $C_{\mathrm{c},k}^{\mathrm{UL}}$ for the edge server cluster and cloud server, respectively.

Let $\tau_{k,\mathrm{e}}$ and $\tau_{k,\mathrm{c}}$ denote the task transmission time from user k to the edge server cluster and the cloud server, respectively. Then we have

$$\tau_{\mathrm{e},k} = \frac{D}{C_{\mathrm{e},k}^{\mathrm{UL}}} \quad \text{and} \quad \tau_{\mathrm{c},k} = \frac{D}{C_{\mathrm{c},k}^{\mathrm{UL}}} + \tau_{\mathrm{RTT}}, \tag{2}$$

where D is the data size of one subtask in terms of bits, and τ_{RTT} denotes the expected transmission delay between the base station and the cloud server. The reference value of τ_{RTT} can be obtained from statistics in field experiments [22].

(2) Computation Model: In our system, servers support the processing of multiple tasks by using the technique of virtual parallel processing [23]. Let $\zeta_{\mathrm{e},k}$ and $\zeta_{\mathrm{c},k}$ denote the task execution time on the edge server and the cloud server, respectively. Then we have

TABLE 1
A List of the Key Defined Symbols

Notation	Definition
$K, \mathbb{K}$	the number of users and the user set
C_0	the baseline computation load
N_k	the number of subtasks from user k
$\mathbb{N}_k$	the subtask set from user k
$N, \mathbb{N}$	the number of subtasks and the subtask set
$C_\mathrm{e}, C_\mathrm{c}$	computation capacity for edge/cloud server
ρ	a value to balance the server importance
τ	observation/decision time slot
$\mathbf{s}$	subtask scheduling profiles
s_n	the subtask scheduling decision for subtask n
$\mathbf{v}$	user scheduling profiles
v_k	# subtasks offloaded to edge server for user k
$\mathbf{v}_{-k}$	scheduling profiles of all other users except k
$U(v_k, \mathbf{v}_{-k})$	the utility function for user k given strategy v_k
$\mathbb{E}, \mathbb{C}$	the subtask set offloaded to edge/cloud server
$n_\mathrm{e}, n_\mathrm{c}$	the number of subtasks offload to edge/cloud
$n_{\mathrm{e},k}, n_{\mathrm{c},k}$	# subtasks offload to edge/cloud from user k
$T_\mathrm{e}(n_\mathrm{e})$	processing time for tasks in edge server
$T_\mathrm{c}(n_\mathrm{c})$	processing time for tasks in cloud server
$T_\mathrm{s}(n_\mathrm{e}, n_\mathrm{c})$	task processing time from system viewpoint
$\mathbb{V}_k$	the set of strategies for user k
$n_{\mathrm{r},k}$	# subtasks not in a group formed by user k
ψ	the cut-off value
b	one-bit control information
p	decision adjustment probability
M	the number of distributed edge servers

$$\zeta_{\mathrm{e},k} = \frac{n_{\mathrm{e},k} C_0}{C_{\mathrm{e},k}} \quad \text{and} \quad \zeta_{\mathrm{c},k} = \frac{n_{\mathrm{c},k} C_0}{C_{\mathrm{c},k}}, \tag{3}$$

where $n_{\mathrm{e},k}$ is the number of subtasks from user k processed by the edge, and $n_{\mathrm{c},k}$ represents the number of subtasks from user k processed by the cloud. The parameters $C_{\mathrm{e},k}$ and $C_{\mathrm{c},k}$ are defined as the computation capacity of the edge server and cloud server respectively. In terms of the unit of measures for computation capacity, similar to the definitions in the previous work (e.g., [9], [20], [21]), we focus more on CPU speed, which can be measured by the number of CPU cycles per unit time (e.g., second). Specifically, the amount of allocated computation resource on the edge (or cloud) (namely, $C_{\mathrm{e},k}$ or $C_{\mathrm{c},k}$) is determined by the results of competition among multiple users. Our goal in this paper is to minimize the overall task processing time, and thereby it is important to avoid the congestion of any particular edge server, unless the whole system is overloaded with too many tasks for processing.

2.2 Game Formulation

We model the decision-making problem of heterogeneous task offloading as a multi-player game, in which users are modeled as players. Within a decision-making period τ, each user can choose its own task offloading strategy independently. A strategy is a vector, in which each element indicates the number of subtasks that should be offloaded to each resource provider.

Let $\mathbf{v} = [v_1, v_2, \ldots, v_K]^{\mathrm{T}}$ denote a vector of user offloading strategies[1], where v_k is the number of subtasks generated by user k that are offloaded to the edge server. Furthermore, we let $\mathbf{s} = [s_1, s_2, \ldots, s_N]^{\mathrm{T}}$ denote the subtask scheduling profile, where s_n denotes the decision on subtask n and $s_n \in \{0, 1\}$. If $s_n = 1$, subtask n is offloaded to an edge server; otherwise, subtask n is offloaded to the cloud. Thus, we have $v_k = \sum_{n \in \mathbb{N}_k} s_n$, where $\mathbb{N}_k$ denotes the subtask set of user k. Apparently, the payoff of a user depends on the actions conducted by the other users. Let $\mathbf{v}_{-k} = [v_1, v_2, \ldots, v_{k-1}, v_{k+1}, \ldots, v_K]^{\mathrm{T}}$ denote the offloading decisions made by all other users except user k.

Following the classical definition [13], the game can be formally defined as below.

Definition 1. A minority game model *for task offloading with incomplete information. We formulate a minority game $\Gamma = \langle \mathbb{K}, \mathbb{V}_k, U(v_k, \mathbf{v}_{-k}) \rangle$, where the set of users $\mathbb{K}$ is regarded as the set of players, $\mathbb{V}_k$ is the set of strategies for user k, and the payoff for user k can be defined as the utility function $U(v_k, \mathbf{v}_{-k})$. The set of strategies $\mathbb{V}_k$ are only known by the user instead of any opponent, however, the payoff function $U(v_k, \mathbf{v}_{-k})$ depends not only on your own strategy but on strategies chosen by your opponents.*

Our formulated minority game belongs to the game with incomplete information due to the following two reasons:

1) Each player in our game does not know its opponents' possible action spaces. In our paper, an action strategy is defined by a profile indicating the number of subtasks that are offloaded to each server. Formally, let $\mathbf{s} = [s_1, s_2, \ldots, s_{N_k}]^{\mathrm{T}}$ denote the action profile, where s_n denotes the decision on subtask n and $s_n \in \{0, 1\}$. If $s_n = 1$, subtask n will be offloaded to an edge server; otherwise, subtask n will be offloaded to the cloud. The action spaces are related to the value of N_k, i.e., the number of generated tasks. As N_k varies with k and a player may not know the properties of tasks generated by its opponents, the action spaces of players can be treated as incomplete information.
2) Each player in our game does not know its opponents' payoffs. Recall that the payoff function is defined as the utility $U(v_k, \mathbf{v}_{-k})$, $\mathbf{v} = [v_1, v_2, \cdots, v_K]^{\mathrm{T}}$. This indicates that the payoff is determined by not only the individual action v_k, but also actions of the other players $\mathbf{v}_{-k}$. Diverse action spaces induce different payoff functions for our game.

In the formulated MG model, user k aims to make a decision v_k to maximise the utility for its own tasks, i.e., $\max_{v_k \in \{0,1,\cdots,N_k\}} U(v_k, \mathbf{v}_{-k})$. For an edge (or cloud) server, the processing time is a function of the amount of assigned task load. Let $T_{\mathrm{e}}(n_{\mathrm{e}})$ denote the task processing time on an edge server, and $T_{\mathrm{c}}(n_{\mathrm{c}})$ denote the task processing time on a cloud server. We consider the total latency determined by the slowest subtask, then the payoff function can be represented as:

$$U(v_k, \mathbf{v}_{-k}) = \begin{cases} 1/T_{\mathrm{e}}(n_{\mathrm{e},-k} + v_k), & \text{if } v_k = N_k \\ 1/T_{\mathrm{c}}(n_{\mathrm{c},-k} + N_k - v_k), & \text{if } v_k = 0 \\ 1/\max\{T_{\mathrm{e}}(n_{\mathrm{e},-k} + v_k), T_{\mathrm{c}}(n_{\mathrm{c},-k} + N_k - v_k)\}, & \\ & \text{otherwise} \end{cases}, \tag{4}$$

where $n_{\mathrm{e},-k}$ denotes the number of subtasks being offloaded to the edge server from the other users except user k, and $n_{\mathrm{c},-k}$ denotes the number of subtasks being offloaded to the cloud server from other users except user k.

Combining Eqs. (2) and (3), we have the time experienced by tasks that are offloaded to the edge server as:

$$T_{\mathrm{e}}(n_{\mathrm{e}}) = \tau_{\mathrm{e}} + \tfrac{n_{\mathrm{e}} C_0}{C_{\mathrm{e}}}, \tag{5}$$

where $\tau_{\mathrm{e}} = \mathrm{E}_k\{\tau_{\mathrm{e},k}\}$ refers to the expected value and $n_{\mathrm{e}} = \sum_{n \in \mathcal{N}} s_n$. Similarly, we have the task processing time on the cloud server as:

$$T_{\mathrm{c}}(n_{\mathrm{c}}) = \tau_{\mathrm{c}} + \tfrac{n_{\mathrm{c}} C_0}{C_{\mathrm{c}}}, \tag{6}$$

where $\tau_{\mathrm{c}} = \tau_{\mathrm{e}} + \tau_{\mathrm{RTT}}$ and $n_{\mathrm{c}} = N - n_{\mathrm{e}}$.

For each player, minimizing its task processing time is the strategy to win the game. However, without the knowledge of actions chosen by the other users, it is not easy to make the right choice. A solution to the MG based task offloading design with incomplete information will be presented in the following sections.

2.3 Preliminary of Minority Game

Minority game is efficient in modeling collective behaviors of users when they have to compete for limited resources with incomplete information. In its original form, namely, the El Farol Bar problem, players make their decisions on whether to attend a bar each night. Going to a bar is only enjoyable only if it is not too crowded, otherwise people would rather

1. The time symbol t is omitted to keep our presentation concise since the scheduling rule is identically designed for each round decision.

stay at home. Intuitively, players adjust their behavior based on their expectations on what other players are going to do next, and these expectations are generated by the attendance decision of the other players. At each round of MG, each player determines his/her own action based on historical and preference factors. After all decisions are made, the action associated with less players is declared as minority side strategy and those players get the chance to win certain payoffs. The game result is broadcast back to all players such that they can update their information and make necessary adjustments in future expectations. As each player makes his/her own decision independently, the game is carried out in a decentralized manner [23], [24], [25], [26].

In a basic MG, the players select between two alternatives and the players belonging to the minority group win. The minority is typically defined by using a cut-off value. The collective sum of the selected actions by all players is referred to as the attendance. These two key terminologies, the cut-off value and attendance, are defined as below.

Definition 2. Cut-off value ψ *is defined as a threshold value for the number of subtasks offloaded to the edge server such that the task processing time is minimized, i.e.,*

$$\min_{\psi\in\mathbb{R}} \; l(\psi) = \min_{\psi\in\mathbb{R}} \; \max\{T_{\rm e}(\psi), T_{\rm c}(N-\psi)\}.$$

From Eqs. (5) and (6), $T_{\rm c}(n_c)$ and $T_{\rm e}(n_e)$ are monotonically increasing functions with the number of subtasks. Then, $T_{\rm e}(\psi)$ monotonically increases with ψ, while $T_{\rm c}(N-\psi)$ monotonically decreases with ψ. To obtain an optimal cut-off value ψ^*, it should be guaranteed that $T_{\rm c}(N-\psi^*) = T_{\rm e}(\psi^*)$, and we have

$$\psi^* = \frac{NC_0C_{\rm e} - (\tau_{\rm c}-\tau_{\rm e})C_{\rm c}C_{\rm e}}{C_0(C_{\rm c}+C_{\rm e})}. \tag{7}$$

In the practical system, the cut-off value ψ can be approximated by an rounding integer, and we have

$$\psi = \begin{cases} \lfloor\psi^*\rfloor, & \text{if } l(\lfloor\psi^*\rfloor) \le l(\lceil\psi^*\rceil), \\ \lceil\psi^*\rceil, & \text{otherwise.} \end{cases} \tag{8}$$

The cut-off value ψ, as the key factor of minority game, explains how users play the game by making offloading decisions. If $n_{\rm e} \ge \psi$ (equivalent to $T_{\rm e} \ge T_{\rm c}$), subtasks offloaded to the edge (i.e., majority) lose and subtasks offloaded to the cloud (i.e., minority) win. In contrast, when $n_{\rm e} \le \psi$ (equals to $T_{\rm e} \le T_{\rm c}$), the subtasks offloaded to the edge (i.e., minority) win and the others lose.

In the standard MG, attendance is defined as the collective sum of the selected actions by all players, originally for decision scenarios with homogeneous tasks [13]. For example, in the El Farol Bar problem mentioned in Section 2.3, players make their decisions on whether to attend a bar every night. Let "to attend the bar" as strategy 1 and "not to attend the bar" as strategy 2. It is introduced in MG by labelling strategy 1 as 1 and strategy 2 as -1, which is called a spin. Then the attendance A can be represented as $A = n_1 - n_2$, where n_i denotes the number of users choosing strategy i. To accommodate heterogeneous computing capacity in the practical system, we redefine it as below.

Definition 3 (Attendance). *Let ρ denote a ratio on the capacities between edge servers and cloud servers, where $0 < \rho \le 1$. The derivation of ρ will be introduced later. We have redefined "attendance" as:*

$$A = n_{\rm e} - \rho n_{\rm c}. \tag{9}$$

Specifically, this definition considers the asymmetric processing capacity between edge servers and cloud servers [23], [24], [25], [26]. As our objective is to minimize the task processing time, we derive a lemma to establish the connection between attendance and the payoff function.

Lemma 1. *The system performance (e.g., task processing time) is optimized when the absolute value of the attendance is minimized.*

Proof. Recall Eq. (9), since $n_{\rm c} = N - n_{\rm e}$, we have

$$A = n_{\rm e} - \rho(N - n_{\rm e}) = (1+\rho)n_{\rm e} - \rho N. \tag{10}$$

With Eq. (5), we have that $T_{\rm e}$ increases as $n_{\rm e}$ increases, which is linearly proportional to a positive A from Eq. (10). Similarly, $T_{\rm c}$ increases with the decrease of A. Overall, the task processing time increases with $|A|$. Therefore, we can conclude that the task processing time can be optimized when the absolute value of A is minimized. □

Recall the optimal cutoff value defined in Eq. (7), we define the ratio ρ such that $A = \psi^* - \rho(N-\psi^*) = 0$. By solving this equation, we have

$$\rho = \frac{NC_{\rm e} + (\tau_{\rm c}-\tau_{\rm e})C_{\rm c}C_{\rm e}}{NC_{\rm c} - (\tau_{\rm c}-\tau_{\rm e})C_{\rm c}C_{\rm e}}, \tag{11}$$

and $0 < \rho \le 1$. Especially, in the general condition $C_{\rm e}(\tau_{\rm c}-\tau_{\rm e}) < C_{\rm c}(\tau_{\rm c}-\tau_{\rm e}) << N$, we have $\rho \approx C_{\rm e}/C_{\rm c}$.

3 Task Offloading for Partially Distributed Architecture

In this section, we propose an MG based task offloading algorithm for the partially distributed architecture of a heterogeneous edge computing scenario with incomplete information, and prove that the proposed algorithm can approach the optimality.

3.1 MG-Based Task Offloading Schedule

In the standard MG based offloading schemes, tasks are assumed to be homogeneous in terms of computation requirements, based on which the offloading decisions are made. However, tasks generated by users in real environments are commonly heterogeneous. Moreover, a naive MG solution cannot guarantee to converge to the optimal point. To address these challenges, we propose a novel MG based offloading algorithm, which not only makes decision with incomplete information but handles heterogeneous tasks generated by different users and servers with heterogeneous processing capacities. Besides, our algorithm is proven to be able to converge to a near-optimal point.

To handle the general case that tasks are commonly heterogeneous in the computation requirements, we introduce the concept of group, based on which our MG based task offloading algorithm is proposed.

Definition 4 (Group). *A "group" is the smallest unit when conducting the offloading procedure in our algorithm. Namely,* *all the subtasks are offloaded in groups and one group of subtasks is mapped to only one server.*

By introducing groups, we can achieve high flexibility when offloading heterogeneous tasks and reduce the complexity of offloading decision-making. In our algorithm, the problem of task offloading is decomposed into two phases:

1) Server selection, i.e., mapping each group to a server side, either edge server side or cloud server side.
2) Task scheduling, i.e., matching each subtask with a specific server.

These two phases will be introduced as follows.

3.1.1 MG Based Server Selection

In Algorithm 1, we propose an MG based server selection algorithm, which contains three major steps:

Algorithm 1. *MG Based Server Side Selection*

```
1: Given: $N, K, N_k, \rho, s_n = \{0, 1\}, t = 1$
2: % initial allocation
3: for $k = 1 : K$ do
4:     Calculate $n_{e,k}$, $n_{c,k}$ and $n_{r,k}$ following Eq. (12);
5:     Allocate $n_{e,k}$ subtasks to side 1, $n_{c,k}$ subtasks to side 0;
6:     Allocate $n_{r,k}$ subtasks following Eq. (14);
7: end for
8: repeat
9:     Calculate the attendance $A$ following Eq. (9);
10:    if $A > 0$ then
11:        $b(t) = 0$
12:    else
13:        $b(t) = 1$
14:    end if
15:    % scheduler broadcasts the winner information $b(t)$
16:    % user adjusts decision with probability $p$ in Eq. (17)
17:    for $n = 1 : N$ do
18:        if $s_n \neq b(t)$ then
19:            if $(\text{rand}()\%N) < \text{abs}(A) - 1$ then
20:                $s_n = (s_n + 1)\%2$
21:            end if
22:        end if
23:    end for
24:    $t = t + 1$
25: until $p = 0$
```

1) *Subtask grouping and initial allocation*. First, subtasks are grouped to match servers with different processing capacities. Given the capacity balancing ratio ρ, we initially allocate the number of subtasks generated by user k to the edge server cluster as:

$$n_{e,k} = \left\lfloor \frac{\rho N_k}{1+\rho} \right\rfloor. \tag{12}$$

Our allocation follows the same procedure as that in the classical papers (e.g., [12], [27], [28]). The crux of the allocation infers from the fact that the transmission time increases with the workload offloaded to the server. Similarly, the number of subtasks offloaded to the cloud server is initially set as

$$n_{c,k} = n_{e,k}/\rho. \tag{13}$$

After initial subtask grouping, there may exist residual subtasks without being scheduled, i.e., $n_{r,k} = N_k - n_{e,k} - n_{c,k}$, where $n_{r,k}$ denotes the number of subtasks that are not in an initial group division. The offloading decisions for these subtasks follow probability

$$p_e = \frac{\rho}{1+\rho} \quad \text{and} \quad p_c = \frac{1}{1+\rho}. \tag{14}$$

This principle is proposed to minimize the absolute value of the attendance.

2) *Information collection and winner broadcast*. Following the offloading principle in Eqs. (12)-(14), the scheduler can calculate the values of n_e and n_c as

$$n_e = \sum_{k=1}^{K} n_{e,k} \quad \text{and} \quad n_c = \sum_{k=1}^{K} n_{c,k}. \tag{15}$$

Then, the scheduler broadcasts the winning group with one-bit control information $b(t)$, defined as

$$b(t) = \begin{cases} 1, & \text{if} \quad n_e < \psi \\ 0, & \text{otherwise.} \end{cases} \tag{16}$$

When $b(t) = 1$, the subtasks offloaded to the edge server win; otherwise, the subtasks offloaded to the cloud server win. Given the control information, users could change their strategies to improve decision-making for the next round offloading.

3) *Decision adjustment and iterative scheduling*. It is essential for the subtasks in the majority to adjust their decisions to achieve a better performance. For example, suppose that the subtasks offloaded to the "cloud server" win, then, in the next round, we keep decision for subtasks that choose the "cloud server" action, however, subtasks in the "edge server" may set to change their decisions by the user.

We propose a decision adjustment policy. The subtasks in the majority change their offloading decisions according to the probability

$$p = \begin{cases} \frac{|A|-1}{N}, & \text{if } |A| > 1 \\ 0, & \text{otherwise.} \end{cases} \tag{17}$$

The task offload decision adjustment and scheduling iteratively continue until $p = 0$.

For clarity, we use an example with two users to describe our algorithm (as shown in Fig. 3). User 1 generates seven subtasks in one time slot, while user 2 generates five subtasks in one slot. We set the edge/cloud server capacity ratio as $\rho = 1/2$. In Step 1, initial groups are formed among users. Following the principles defined in Eqs. (12) and (13), we have $n_{e,1} = 2$ and $n_{c,1} = 4$ for user 1. While for user 2, we have $n_{e,2} = 1$ and $n_{c,2} = 2$ in the initial two groups. With these operations, we have the number of residual subtasks that are not included in any initial group, i.e., $n_{r,1} = 1$ and $n_{r,2} = 2$. In total, we have $N_r = n_{r,1} + n_{r,2} = 3$ subtasks for random offloading schedule. In this example, we assume the three residual subtasks selecting "edge server", i.e., the edge offloading group is in "majority" while the cloud offloading group is in "minority". In Step 2, we have the attendance $A = n_e - \rho n_c = 3$, which is broadcast to all the users in the network. Since the attendance value is greater than 1, we need to adjust the decision profile in Step 3. As defined

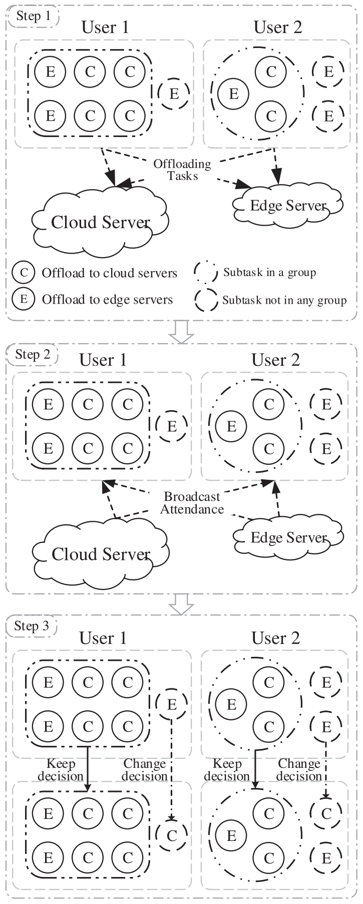


Fig. 3. An example of the proposed MG based offloading procedure.

in Eq. (17), we have the decision adjustment probability as $p = (|A| - 1)/N = (3-1)/12 = 1/6$. That is, the subtasks in the majority change their decisions with probability p, while the subtasks in the minority stay unchanged. With such a policy, the decision profile iteratively changes until achieving the balanced state, when no subtask can change decision to achieve a smaller attendance value. Finally, we have four subtasks offloaded to the edge server and eight subtasks to the cloud server.

3.1.2 Intra-Cluster Task Scheduling

Here we propose an intra-cluster task scheduling algorithm. Assume that the edge cluster is composed of J edge servers. Let $n_{e,j}$ denote the number of subtasks allocated to the edge server j, and $\sum_j n_{e,j} = n_e$. Recall that edge servers support computation on multiple tasks with virtual parallel processing. The number of subtasks allocated to each edge server should be proportional to its capacity. This model has been widely used for designing allocation policies (e.g., [12], [27], [28]) for computation resources. For example, in [27], the CPU cycles at the edge server are proportionally allocated to each device such that they experience the same execution latency. Then we have the optimal number of subtasks scheduled to the edge server j as

$$n_{e,j} = \frac{n_e C_j}{\sum_j C_j}. \tag{18}$$

Note that the number of subtasks in Eq. (18) might not be an integer, and we refine the allocation as

$$n'_{e,j} = \left\lfloor \frac{n_e C_j}{\sum_j C_j} \right\rfloor, \tag{19}$$

and we have $\sum_j n'_{e,j} \le n_e$. Let Δ_j denote the gap between $n_{e,j}$ defined in Eq. (18) and $n'_{e,j}$ defined in Eq. (19), and we have $\Delta_j = n_{e,j} - n'_{e,j}$. Let $\mathcal{J}_\Delta$ denote the set containing the first $n_e - \sum_j n'_{e,j}$ values on Δ_j. Then we have the revised subtask allocation among edge servers as

$$n^*_{e,j} = \begin{cases} \left\lceil n_e C_j / \sum_j C_j \right\rceil, & \text{for } j \in \mathcal{J}_\Delta \\ \left\lfloor n_e C_j / \sum_j C_j \right\rfloor, & \text{otherwise.} \end{cases} \tag{20}$$

The intra-cluster task allocation, conducted by the master node of the edge cluster, is illustrated in Algorithm 2.

Algorithm 2. *Intra-Cluster Task Scheduling*

```
1: Given: the number of subtasks offloaded to edge server cluster $n_e$, the capacity $C_j$ for edge server $j$.
2: % Step 1: Subtasks Allocation Policy Calculation
3: for $j = 1 : J$ do
4:     Calculate $n^*_{e,j}$ following Eqs. (18)-(20).
5: end for
6: % Step 2: Subtasks Allocation
7: for $j = 1 : J$ do
8:     Allocate $n^*_{e,j}$ subtasks to edge node $j$.
9: end for
```

To theoretically verify the performance of our intra-cluster scheduling, we derive the gap of task processing time between the practical case with a single edge server cluster and the optimal case with resource pool. In the optimal case, edge servers are perceived as a resource pool by consolidating the computation resources of all edge servers.

In the optimal case, we have the task processing time as

$$T^*_e = \tau_e + \frac{n_e C_0}{\sum_j C_j}. \tag{21}$$

With the intra-cluster scheduling, we have the practical task processing time as

$$T'_e = \tau_e + \max\left\{ \frac{n_{e,1} C_0}{C_1}, \frac{n_{e,2} C_0}{C_2}, \ldots, \frac{n_{e,J} C_0}{C_J} \right\} \tag{22}$$

which might be greater than T^*_e due to the heterogeneity of servers. For the performance gap between T'_e and T^*_e, we have the following proposition.

Proposition 2. *The performance gap on task processing time between the practical case with a single edge server cluster and the optimal case is not greater than $C_0/\min_j\{C_j\}$, where C_0 is the computation requirement for one unit subtask and C_j is the computing capacity of edge server j.*

Proof. Comparing Eqs. (21) and (22), we can conclude that the practical task processing performance can achieve the optimal value if and only if Eq. (18) holds. As we assume that servers support computation on multiple tasks with virtual parallel processing, subtasks allocated to edge servers are proportional to the capacity. It should be noted that the difference between $n^*_{e,m}$ and $n_{e,m}$ is not greater than one. As the system performance is mainly constrained by the slowest edge server, we have the difference on task processing time between the practical system and optimal value as $T'_e - T^*_e \leq C_0/\min_j\{C_j\}$. □

In practical, the computing capacity of an edge server is generally greater than 10 Gigacycles per second. Meanwhile, with efficient task partition, the computation requirement for each subtask can be lower than 100 CPU Megacycles. Thus, the performance gap will be lower than 10 milliseconds, showing the effectiveness of our design. With Proposition 2, a way to reduce the gap is to set a small size of partitioned subtask C_0.

3.2 Performance Analysis

In this section, we analyze the properties of our MG based scheduling algorithm, and check its equilibrium, convergence, and performance bound.

3.2.1 Nash Equilibrium

Nash equilibrium (NE) is used to discuss the equilibrium state in general game models. An Nash equilibrium is defined as a strategy profile that maximizes the expected payoff for each player given their beliefs and the strategies played by the other players. That is, a strategy profile $\mathbb{V}_k$ is an NE if and only if for every player keeping the strategies of every other player fixed, strategy v_k maximizes the expected payoff of player k according to his/her belief. In the following, we prove the existence of at least one NE point for the proposed game.

Based on the utility function defined in Eq. (4), the NE in a pure strategy for the MG is given as below.

Definition 5. *For player k, the strategy profile $\{v^*_k, N_k - v^*_k\}$ is a NE of the MG if at the equilibrium, the utility cannot be improved by unilaterally changing the strategy, i.e.,*

$$U(v^*_k, \mathbf{v}_{-k}) \geq U(v^*_k + 1, \mathbf{v}_{-k}), \tag{23}$$

and

$$U(v^*_k - 1, \mathbf{v}_{-k}) \leq U(v^*_k, \mathbf{v}_{-k}). \tag{24}$$

Thus, the individual player's utility cannot be improved by unilaterally deviating from the equilibrium.

Proposition 3. *The NE state can be reached if and only if v^*_k subtasks are offloaded to the edge server from player k, given that $l(\lfloor\psi^*\rfloor) \neq l(\lceil\psi^*\rceil)$.*

Proof. Please see Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2020.2988161, for details. □

Proposition 4. *If $l(\lfloor\psi^*\rfloor) = l(\lceil\psi^*\rceil)$, NE can be achieved by assigning either $\lfloor\psi^*\rfloor$ or $\lceil\psi^*\rceil$ subtasks to the edge server.*

Proof. Similar to the proof for Proposition 3, we can obtain two NE points $\lfloor\psi^*\rfloor - n_{\mathrm{e},-k}$ and $\lceil\psi^*\rceil - n_{\mathrm{e},-k}$, since both of these points satisfy Eqs. (23) and (24) simultaneously. That is, if $l(\lfloor\psi^*\rfloor) = l(\lceil\psi^*\rceil)$, NE can be achieved by assigning either $\lfloor\psi^*\rfloor - n_{\mathrm{e},-k}$ or $\lceil\psi^*\rceil - n_{\mathrm{e},-k}$ subtasks from user k to the edge server. This completes the proof. □

3.2.2 Convergence

For the convergence of our MG based task offloading algorithm, we have the following proposition.

Theorem 5. *With the MG based algorithm, the attendance $|A|$ can converge to a value not greater than 1. That is, $\exists\epsilon \leq 1$ and $\exists T > 0$, such that $|A(t)| = \epsilon$ for $t \geq T$.*

Proof. Please see Appendix B, available in the online supplemental material, for details. □

With Proposition 5, we find that the attendance $|A|$ can converge to a value no more than one with our proposed MG based offloading scheme. Recall that attendance measures how tasks are allocated to servers with different capacities. Although we cannot directly acquire task processing performance with attendance, the conclusion is still essential as the state with large attendance will result in bad performance in terms of task processing time.

3.2.3 Performance Bound

We then study the performance bound of the task processing time by utilizing the price of anarchy (PoA).

Let $\mathbb{Z}$ denote the set of Nash equilibria of the multi-player offloading game and $\mathbf{v}^* = \{v_1, v_2, \ldots, v_K\}$ denote the optimal solution that maximizes the number of users that can maximize their payoffs. Then the PoA is defined as

$$\mathrm{PoA} = \frac{\max_{\mathbf{v}\in\mathbb{Z}} \sum_{k\in\mathbb{K}} t_k(\mathbf{v})}{\min_{\mathbf{v}\in\mathbb{V}} \sum_{k\in\mathbb{K}} t_k(\mathbf{v})}. \tag{25}$$

Note that, for the metric of the task processing time, a smaller PoA is better.

For our proposed MG based offloading scheme, we have the following theorem on PoA.

Theorem 6. *For the multi-player MG offloading game, the PoA of the task processing time satisfies that*

$$1 \leq \mathrm{PoA} \leq \frac{\max\left\{\tau_{\mathrm{e}} + \frac{(n^*_{\mathrm{e}}+1)C_0}{C_{\mathrm{e}}}, \tau_{\mathrm{c}} + \frac{n^*_{\mathrm{c}}C_0}{C_{\mathrm{c}}}\right\}}{\min\left\{\tau_{\mathrm{e}} + \frac{n^*_{\mathrm{e}}C_0}{C_{\mathrm{e}}}, \tau_{\mathrm{c}} + \frac{(n^*_{\mathrm{c}}-1)C_0}{C_{\mathrm{c}}}\right\}}, \tag{26}$$

where $n^*_{\mathrm{e}} = \lfloor\psi^*\rfloor$.

Proof. Please see Appendix C, available in the online supplemental material, for details. □

Theorem 6 indicates that when the computation of one subtask decreases (i.e., C_0 is small), the worst-case performance of Nash equilibrium can be improved. Moreover, when servers have plenty of computing resources (i.e., C_{c} and C_{e} are large), the worst-case Nash equilibrium is closer to the optimum and hence the PoA is lower.

4 Task Offloading for Fully Distributed Architecture

To further prove the feasibility of our approach, we extend our design to a fully distributed scenario with multiple distributed edge servers. This is a general case since edge servers deployed by different operators might be restricted in mutually resource sharing. Although the standard MG theory provides a scalable extension (e.g., simplex game [29]) to the cases where each decision maker owns multiple choices, the task offloading scheduling for the heterogeneous edge computing scenario is still challenging.

Let M denote the number of edge servers and $\mathbb{M}$ the server set. The cut-off value can be generalized as below.

Definition 6. General cut-off value. *The general cut-off value ψ_m denotes a threshold for the number of subtasks offloaded to edge server m, such that the task processing time can be minimized. Let C_m denote the processing capacity of server m, then we have*

$$\psi_m = \frac{NC_m}{\sum_{m=1}^{M} C_m}. \tag{27}$$

With the cut-off value set $\Psi = \{\psi_1, \psi_2, \ldots, \psi_M\}$ in conjunction with multiple distributed servers as offloading choices, there will be more than one winner (i.e., minority). The tasks offloaded to server m is in minority if $n_m \leq \psi_m$. Otherwise, the tasks in server m are in majority.

With the general cut-off values, we can extend the MG based algorithm to the case with multiple choices, with three key procedures presented as follows.

1) *Subtask grouping and initial scheduling*. For the N_k subtasks generated from the user k, the number of subtasks offloaded to edge server m is initially set as

$$n_{m,k} = \left\lfloor \frac{C_m}{\sum_{m=1}^{M} C_m} N_k \right\rfloor. \tag{28}$$

The key idea of the initial subtask scheduling is to make the most efficient decision from the user's perspective. After initial scheduling, some residual subtasks might still be unscheduled, i.e.,

$$n_{\mathrm{r},k} = N_k - \sum_m n_{m,k}, \tag{29}$$

where $n_{\mathrm{r},k}$ denotes the number of subtasks that have not been scheduled initially. These subtasks are initially scheduled to edge server m with probability

$$p_m = \frac{C_m}{\sum_{m=1}^{M} C_m}. \tag{30}$$

2) *Information collection and winner broadcast*. After the offloading decision is made, the scheduler can collect the information on the number of subtasks offloaded to edge server m as

$$n_m = \sum_{k=1}^{K} n_{m,k}. \tag{31}$$

With network information for the tth iteration, the edge server m is in minority, if n_m is not greater than ψ_m, then, the subtasks offloaded to edge server m win; otherwise, the subtasks offloaded to edge server m lose. After information collection, the scheduler broadcasts the winning side by sending M-bit control information $\{b_m(t)\}_{m\in\mathbb{M}}$ defined as

$$b_m(t) = \begin{cases} 1, & \text{if} \quad n_m < \psi_m \\ 0, & \quad \text{otherwise.} \end{cases} \tag{32}$$

After receiving the control information, users could evaluate their strategies to improve decision-making for the $(t+1)$-th round offloading.

3) *Decision adjustment and iterative scheduling*. Following the insights from the MG theory, subtasks in the majority side should change their decisions to achieve better performance. Let $\mathbb{M}^+$ denote the server set in majority, i.e., $m \in \mathbb{M}^+$ if $n_m > \psi_m$. Otherwise, let $\mathbb{M}^-$ denote the server set in minority, i.e., $m \in \mathbb{M}^-$ if $n_m \leq \psi_m$.

Suppose that edge server m wins, then we keep decision unchanged for subtasks that choose edge server m in the next round. Otherwise, subtasks in edge server m are set to adjust their decisions following probability defined as follows. For $m \in \mathbb{M}^+$, $m' \in \mathbb{M}^-$,

$$q_{mm'} = \underbrace{\frac{n_m - \psi_m}{n_m}}_{\text{PartI}} \cdot \underbrace{\frac{\psi_{m'} - n_{m'}}{\sum_{m'}(\psi_{m'} - n_{m'})}}_{\text{PartII}} \cdot \underbrace{\frac{1}{1 + e^{-\kappa\tau_m/\tau_{m'}}}}_{\text{PartIII}} \tag{33}$$

where Part I represents a high decision adjustment probability with a large gap between the actual offloading subtasks n_m and the optimal ψ_m for servers in the majority, Part II denotes a high decision adjustment probability for servers in the minority with a large gap between the optimal ψ_m and the actual offloading subtasks n_m, Part III illustrates the probability that the offloading decision changes from a server in the majority with transmission delay τ_m to a server in the minority with transmission delay $\tau_{m'}$.

Note that some subtasks in the majority should keep the current offloading decision with the probability of

$$q_{mm} = \frac{n_m - \psi_m}{n_m}, \tag{34}$$

where Part II and Part III in Eq. (33) are not included as the offloading decision is not changed.

As a normalized form of $q_{mm'}$, we define the practical decision change probability $p_{mm'}$ as

$$p_{mm'} = \frac{q_{mm'}}{\sum_{m'\in\mathbb{M}^-} q_{mm'} + q_{mm}}. \tag{35}$$

The task offloading determination is conducted until $p_{mm'} = 0,\ \forall m, m' \in \mathbb{M}$.

Note that the cut-off value defined in Eq. (27) cannot be guaranteed to be an integer, which might cause the algorithm not to be convergent since $p_{mm'} \neq 0$. Alternatively, we refine the definition of the general cut-off value as

$$\psi'_m = \left\lfloor NC_m / \sum_{m=1}^{M} C_m \right\rfloor. \tag{36}$$

However, this might cause $\sum_{m=1}^{M} \psi'_m \neq N$, which might also introduce a divergent case. Nevertheless, let Δ_m denote the gap between ψ_m defined in Eq. (27) and ψ'_m defined in Eq. (36), and we have

$$\Delta_m = \frac{NC_m}{\sum_{m=1}^{M} C_m} - \left\lfloor \frac{NC_m}{\sum_{m=1}^{M} C_m} \right\rfloor. \tag{37}$$

Let $\mathcal{M}_\Delta$ denote the set containing the first $N - \sum_{m=1}^{M} \psi'_m$ values on Δ_m. Then we have the revised cut-off generalization value as

$$\psi^*_m = \begin{cases} \lceil NC_m / \sum_{m=1}^{M} C_m \rceil, & \text{for } m \in \mathcal{M}_\Delta \\ \lfloor NC_m / \sum_{m=1}^{M} C_m \rfloor, & \text{otherwise.} \end{cases} \tag{38}$$

Following the idea in Proposition 4, we can also prove that our MG based offloading algorithm with multiple choices can converge to the revised cut-off generalization value defined in Eq. (38). The detailed proof is omitted here due the page limit.

The MG based task offloading algorithm for the fully distributed scenario is illustrated in Algorithm 3.

Algorithm 3. *MG Based Task Offloading for Fully Distributed Architecture*

```
1: Given: $C_m, N, N_k, s_n = \{1, 2, \dots, M\}$
2: % Step 1: Subtask grouping and initial allocation
3: Calculate general cut-off value $\psi^*_m$ following Eq. (38);
4: for $k = 1 : K$ do
5:     Calculate $n_{m,k}$ and $n_{r,k}$ following Eqs. (28) and (29);
6:     Allocate $n_{m,k}$ subtasks to edge server $m$;
7: end for
8: % Step 2: Information collection and winner broadcast
9: for $t = 1 : T$ do
10:    Collect information and calculate $n_m$ following Eq. (31);
11:    if $n_m < \psi^*_m$ then
12:        $b_m(t) = 1$
13:    else
14:        $b_m(t) = 0$
15:    end if
16:    % Step 3: Decision adjustment and iterative scheduling
17:    Calculate normalized prob. $p_{mm'}$ following Eq. (35)
18: end for
```

With multiple edge servers, the competition between distributed servers and the diverse communication delays to different servers cause a gap of task processing time compared to the scenario where edge servers can form a single cluster. Thus, we introduce an upper bound on the gap of the task processing time between the case with fully distributed edge servers and the optimal solution with an edge resource pool. The result is shown as follows.

Proposition 7. *The performance gap between the practical case with fully distributed servers and the optimal case with a resource pool is not greater than $C_0/\min_m\{C_m\} + \max\{\tau_m - \tau_{m'}\}$, where C_0 is the computation requirement for one unit subtask, C_m is the computing capacity of edge server m, and τ_m is the transmission delay to edge server m.*

Proof. As the processing capacities of servers are identical for the scenario with distributed multiple servers, and the scenario where edge servers form a cluster. They all support computations on multiple tasks with virtual parallel processing, and the number of subtasks allocated to edge servers is proportional to their capacity. Recall the conclusion in Proposition 2, the task execution time difference is less than $C_0/\min_m\{C_m\}$.

Different from Proposition 2, we should take the transmission delay into account as the task delivery time to different servers is usually different. We have the gap shown as the maximum difference between delivery time of all tasks, i.e., $\max\{\tau_m - \tau_{m'}\}$.

In total, the performance gap between T_c and T_d is not greater than $C_0/\min_m\{C_m\} + \max\{\tau_m - \tau_{m'}\}$. □

TABLE 2
Default Simulation Settings on Key Parameters

Parameter	Description	Value	Units
K	the number of users	20	
N	the number of subtasks	$125 - 2000$	
C_e	edge server capacity	2.5	GHz
C_c	cloud server capacity	$2.5 - 12.5$	GHz
N_{it}	iterations per run	32	
N_{pd}	the number of periods	1000	
τ_c	cloud communication delay	100	ms
τ_e	edge communication delay	10-50	ms

5 Performance Evaluation

In this section, we conduct simulations with various settings to verify our analysis and demonstrate the performance gain achieved by the proposed algorithm.

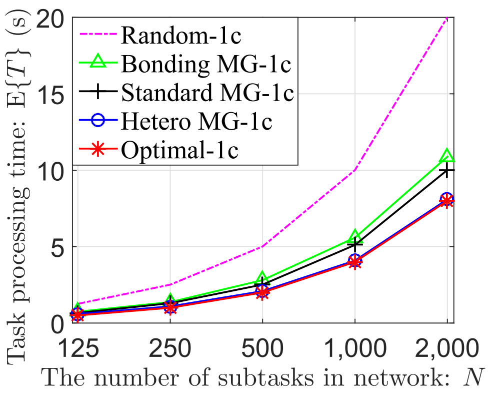
5.1 Simulation Settings

In our experiments, we simulate an edge computing system that contains both cloud and edge servers. In default, there are 20 users in the system and each user can generate tasks independently. The default number of CPU cycles to complete the basic unit computation load of one subtask C_0 equals to 50 Megacycles. Based on the empirical results in [28], [30], we set the range of the workload as $[2.5 \cdot 10^8, 5 \cdot 10^9]$ CPU cycles. For each user, the number of generated task loads obeys a uniform distribution as referred in [27], which is set to be in the range of (0, 100] in our simulations.

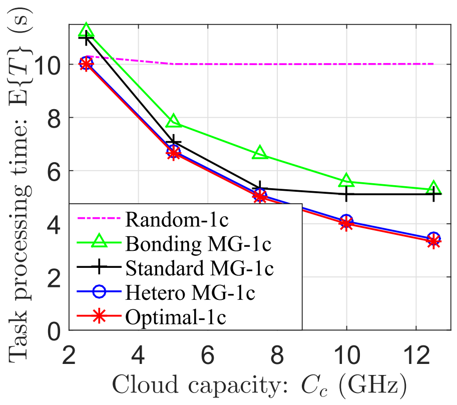
Considering the difference in terms of distance to the cloud (or edge) servers, the round-trip-time from a user to cloud (or edge) servers is set as $\tau_c = 100$ ms (or $\tau_e = 10$ ms) respectively according to [22]. In addition, the computing capacity of the edge server and the cloud server is set as $C_e = 2.5$ GHz and $C_c = 10$ GHz in default, respectively. In reality, the cloud capacity is configurable and can be expanded by installing more CPU cores if necessary [27]. In each experiment, algorithms are executed 32 times to obtain the average results. In each simulation run, decisions on 1000 periods are evaluated and statistically collected [26]. Table 2 lists the key parameters with corresponding values adopted in our simulations.

For performance evaluation, the scheduling results are compared among the following schemes:

- *Random scheme*, in which each task randomly determines whether offloading its computation to the edge server or the cloud server [31].
- *Standard MG scheme*, where each subtask is scheduled based on strategy iteration techniques [23], and subtasks respond to the aggregate action of all subtasks in the previous round.
- *Bonding MG scheme*, where users make offloading decisions according to the standard MG principle, where tasks generated by one user are scheduled in a bonding manner [26]. The bonding MG scheme is

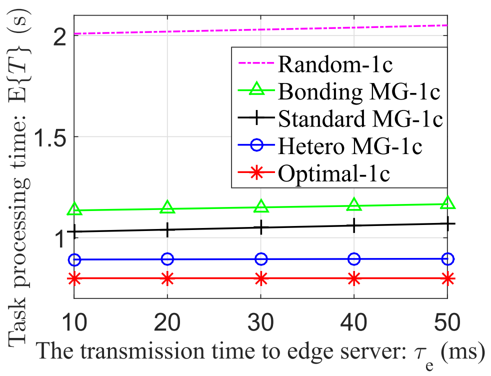


(a) $\mathrm{E}(T)$ v.s. N

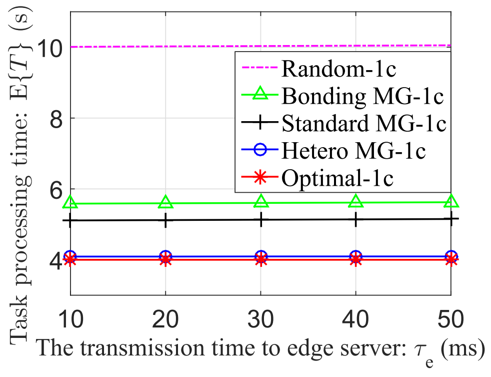


(b) $\mathrm{E}(T)$ v.s. C_c

Fig. 4. The task processing time $\mathrm{E}(T)$ versus the number of subtasks in the system N and the cloud processing capacity C_c, where "-1c" denotes the case with a single cluster (partially distributed architecture).



(a) $C_0 = 10$ Megacycles



(b) $C_0 = 50$ Megacycles

Fig. 5. The task processing time $\mathrm{E}(T)$ versus the task delivery time to the edge server τ_e with different basic task loads.

an extension of the standard MG scheme under scenarios for heterogeneous task load.

- *Heterogeneous MG scheme* ("hetero MG" for short), which is proposed in Algorithm 1 and applicable for the heterogeneous edge computing scenarios.
- *Optimal scheme*, where each players know the actions taken by the rivals, i.e., this is the solution for game with complete information.

5.2 Results for Partially Distributed Architecture

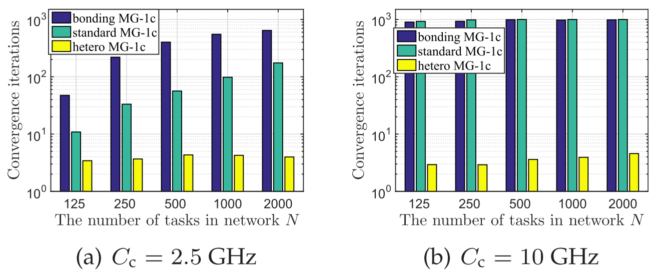
In this section, we will report and analyze our simulation results in terms of major metrics, including task processing delay and convergence time.

5.2.1 *Task Processing Time*

Fig. 4a compares the task processing time versus the number of subtasks in the system, given a constant subtask load C_0. Among the compared schemes, our proposed "hetero MG" scheme outperforms all other schemes by achieving a minimum task processing time. With the increase of N (the number of subtasks in the system), all curves go upwards indicating that the system load is critical for task processing performance. Meanwhile, the gap between our scheme and others increases with the increase of N, showing that our scheme is even better in heavy load scenarios. From the result, we also find that the "hetero MG" scheme achieves pretty good performance which is close to the optimal solution for the scenario with complete information.

Fig. 4b presents the task processing time versus the cloud processing capacity C_c, given a static edge capacity C_e. Again, the "hetero MG" scheme is the best one consuming the least amount of time to process tasks. As the cloud capacity increases, we can observe that the task processing time decreases for all schemes except the random scheme. The "hetero MG" scheme is always better than baselines, while approaches the optimal solution with complete information. Again, the random scheme, which does not take the beliefs of the other players into the offloading design, is the worst one in most cases.

Fig. 5 compares the task processing time versus the edge communication delay τ_e. In the simulations, we fix the task delivery time to the cloud server at $\tau_c = 100$ ms, and vary the task delivery time to the edge server τ_c from 10 ms to 50 ms with two basic subtask loads (i.e., $C_0 = 10$ and $C_0 = 50$ Megacycles). From the figure, we can observe that our "hetero MG" scheme is always the best one achieving the lowest task processing time, and the task processing time will be prolonged significantly for all schemes because of the increase of the basic subtask load from $C_0 = 10$ to $C_0 = 50$ Megacycles. By comparing the two cases with different basic subtask loads, we find that the task processing time T will go upwards slightly as we increase τ_e with $C_0 = 10$ Megacycles. However, such phenomena cannot be observed for the case with $C_0 = 50$ Megacycles, in which the task processing time is very stable with τ_e. This



(a) $C_c = 2.5$ GHz (b) $C_c = 10$ GHz

Fig. 6. Convergence time (i.e., the number of iterations before reaching condition $|A| \leq 1$) versus the number of subtasks in the system.

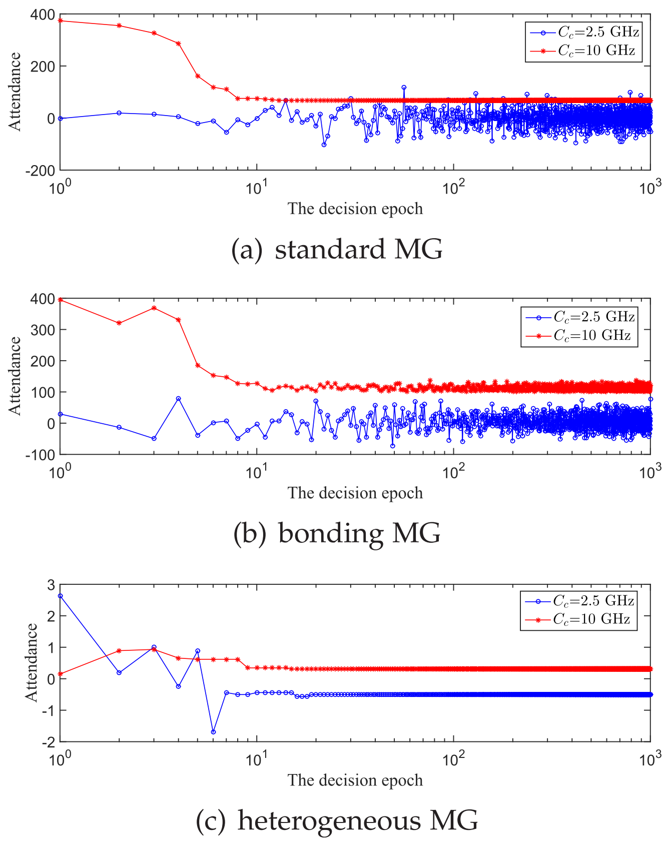


Fig. 7. The convergence on attendance A versus the decision epoch.

difference can be explained from the influence of the task delivery time with different C_0. The larger the value C_0 is, the smaller influence the delivery time will play on the task processing time.

5.2.2 *Algorithm Convergence*

Fig. 6 evaluates the algorithm convergence by comparing the number of iterations taken to reach the equilibrium state versus the number of subtasks in the system. For the proposed "hetero MG" scheme, the criteria of the equilibrium state is the attendance less than 1 (i.e. $|A| \leq 1$). Since the other two schemes might not converge to the equilibrium, we set the identical convergence principle for a convenient comparison. The simulation results show that all schemes except our "hetero MG" scheme need more iterations to converge to the optimal state with the increase of C_c. In Fig. 6a, we can see that there is a clear upward trends of the number of iterations for both "standard MG" and "bonding MG" with the increase of the number of subtasks in the system. We can also observe that C_c increases the number of iterations for "standard MG" and "bonding MG" schemes significantly, especially when the number of subtasks is not very large by comparing Figs. 6a and 6b. The large number of iterations consumed by "standard MG" and "bonding MG" schemes imply that it is difficult for them to achieve the equilibrium state in reality. In contrast, the number of iterations of our scheme is almost unchanged for all cases. In fact, our "hetero MG" scheme will organize subtasks (generated by the same user) to form groups, which can substantially reduce convergence iterations.

Fig. 7 illustrates the converging tendency on the attendance A versus the decision epoch. Importantly, neither the "standard MG" nor the "bonding MG" could guarantee the convergence to a small value on the attendance, which is an essential factor representing the server utilization efficiency. The task processing performance cannot be guaranteed because of a large difference between the number of subtasks offloaded to the two kinds of servers. While for our "hetero MG" scheme, it can fast converge to a small attendance value, which verifies the effectiveness of the group formation and the decision adjustment policy.

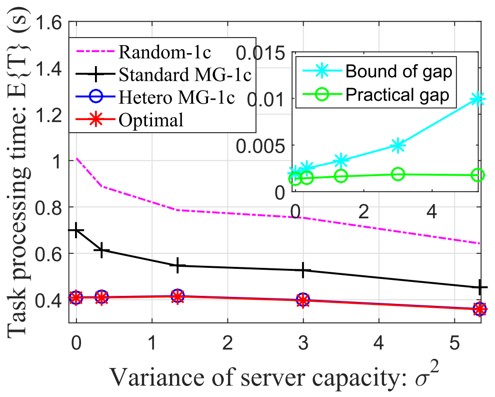
5.3 Results for Fully Distributed Architecture

The experiment results on the extended case for the fully distributed architecture in Section 4 are shown as follows. In our experiments, the computing capacity of the edge servers follows the uniform distribution with the expectation μ and the variance σ^2, i.e., $C_m \sim U(\mu, \sigma^2)$. The variance σ^2 is an essential factor indicating the heterogeneity of the processing capacity of the servers. With a higher value of σ^2, it achieves a larger gap between the server with maximum capacity and the server with minimum capacity. In the following experiments, the expected server capacity is set as a constant (e.g., 5 GHz), while we evaluate the offloading performance versus different levels of server heterogeneity (i.e., variance of server capacity).

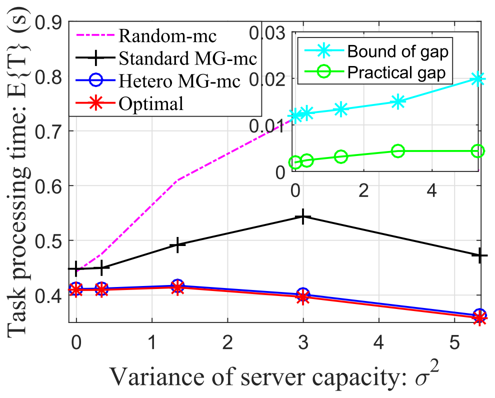
Fig. 8a illustrates the task processing time versus the variance of server capacity for the scenarios where edge servers form a cluster. We find that our "hetero MG" scheme outperforms the other two schemes, including the "standard MG" and the random schemes. The performance improvement mainly comes from taking the heterogeneity issues of the edge computing environment into account.

Fig. 8a also illustrates the performance gap between the "hetero MG" scheme and the optimal solution. Apparently, the gap on the task processing time increases with the diversity of the edge capacity. This might be due to that a small server processing capacity exists with a high probability given a large variance on server capacity.

For the fully distributed scenarios, Fig. 8b illustrates the task processing time versus the variance of server capacity. Again, we find that our proposed "hetero MG" scheme outperforms the other two schemes, including the "standard MG" scheme and the random scheme. With the increase of the capacity variance, the results from the other two schemes show an obvious increasing trend. This might be due to the mismatch between the scheduled tasks and their computation requirements without considering the influence of the heterogeneity of the edge computing scenario.

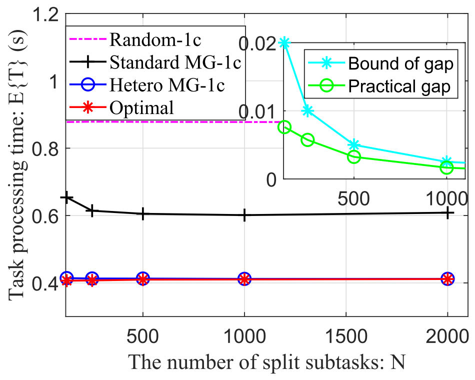


(a) edge servers form a cluster

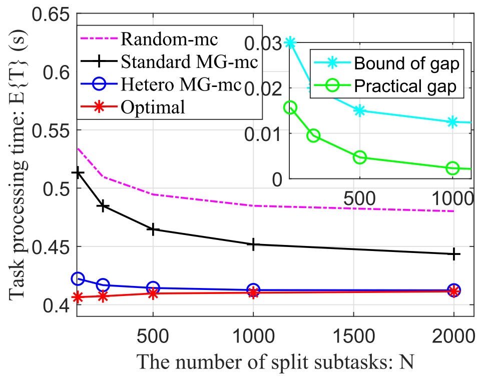


(b) fully distributed edge servers

Fig. 8. The task processing time versus the variance of server capacity, where “-1c” denotes the case with a single cluster (partially distributed architecture) and “-mc” denotes the case with multiple clusters (fully distributed architecture).



(a) edge servers form a cluster



(b) fully distributed edge servers

Fig. 9. The task processing time versus the number of split tasks, where the task load of the network is set as a constant.

Similarly, Fig. 8b illustrates the performance gap between the results from “hetero MG” scheme and the optimal solution. Again, the gap on the task processing time increases with the diversity among edge capacities. Compared to the gap in Fig. 8a, the gap of the scenario with multiple edge servers is a bit larger, which might be caused by various transmission delays to different edge servers.

Fig. 9 compare the task processing time versus the number of split tasks. Different from Fig. 4a with constant subtask load, the task load of the network is set as a constant in Fig. 9. Again, for both partially and fully distributed architectures, the proposed “hetero MG” scheme outperforms the other two compared schemes, while approaches the optimal solution with complete information. This verifies the performance of the MG based algorithm design for heterogeneous edge offloading.

For the performance gap, in Fig. 9a with a single edge cluster, we find a decreasing trend on the practical gap between “hetero MG-1c” solution and the optimal solution with the increase of the number of split tasks. This might be due to that, the finer the granularity of task split is, the better the match between subtasks and heterogeneous computing resources will be. In Fig. 9b, we can find a similar trend on the gap between the “hetero MG-mc” solution and the optimal solution. Differently, the gap for the “hetero MG-mc” solution is relatively larger than that for the “hetero MG-1c” solution. This is reasonable since the computing resources are more heterogeneous for the fully distributed architecture compared to the partially distributed architecture.

6 Related Work

The edge computing framework has been designed to meet the time-critical real-world use cases, as referred in many studies [32], [33], [34]. In the environment of edge computing, a task can be offloaded to a nearby edge server or a remote cloud server to reduce latency and increase throughput. Many previous studies formulated the two-way offload decision as a knapsack problem and solved it using methods, such as dynamic programming and Markov decision process (MDP). For example, Wu *et al.* [6] modeled the task scheduling problem as a knapsack problem, and proposed both offline and online algorithms to maximize the user’s quality of experience satisfaction degree cost effectively. Goudarzi *et al.* [8] proposed a genetic algorithm to timely achieve the best offloading outcome in a heterogeneous multisite context. Sun *et al.* [9] developed a user-centric task offloading scheme to optimize task processing delay, based on Lyapunov optimization and multi-armed bandit theories. Kochovski *et al.* [35] proposed a new MDP based management method for service level agreements to facilitate an automated decision-making process. However, these schemes require a scheduler to collect information and make decisions, which is time consuming and not suitable for decentralized edge computing scenarios.

For distributed scheduling, game theory-driven schemes have attracted much attention. Chen *et al.* [11], [12] formulated the computation offloading decision among devices as a multi-player strategic game. Some work (e.g., [36]) established a non-cooperative game framework to systematically study the stabilization of a competitive mobile edge computing environment, and proposed an iterative algorithm to find the Nash equilibrium of the games. He *et al.* [37] proposed a game-theoretic approach that formulates the cost-effective edge user allocation problem as a potential game, and designed a novel decentralized algorithm for finding a Nash equilibrium. To consider the computation requirement difference between users, these game based schemes always require state information exchange between task initiators, which might introduce problems in scalability and internode interaction overhead.

The above-mentioned games assume the players have complete information about other players' strategies, including the knowledge of other players' choices (e.g., Stackelberg game) in some cases. But in some real scenarios, there is a lack of information about the environment. In the cases with incomplete information, minority game shows its efficiency on players' distributed cooperation decision [13]. Ranadheera *et al.* [23] studied the applicability of MG to solve the distributed decision-making problems in wireless networks. The authors in [26] developed an MG based distributed server activation mechanism for computation offloading in order to guarantee energy-efficient activation of servers. Furthermore, the asymmetric processing capacity between edge servers and clouds has been investigated by a revision of MG with arbitrary cut-offs [23], [26]. However, these works assumed that users are homogeneous with respect to tasks potentially to be offloaded, which cannot be directly adapted to heterogeneous task scheduling scenario.

7 Conclusion

In this paper, to address the challenges incurred by incomplete information and task heterogeneity, we proposed a minority game (MG) based offloading algorithm to perform task offloading in a heterogeneous environment with incomplete information. In our scheme, tasks are divided into subtasks and instructed to form into a set of groups as possible, and the left ones are scheduled to perform decision adjustment in a probabilistic manner. We investigate the properties of our proposed algorithm theoretically and prove that our algorithm can approach a near optimal point. Finally, we conducted extensive simulations to validate our heterogeneous MG scheme and compared its performance with other alternatives. In the future, we plan to explore task offloading in more realistic scenarios with user dynamics and time-varying network conditions.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under Grants 61802452, U1911201, 61972432, 61971366, and 61872420, Guangdong Special Support Program under Grant 2017TX04X148, the Natural Science Foundation of Guangdong under Grant 2018A030310079, the Program for Guangdong Introducing Innovative and Enterpreneurial Teams under Grant 2017ZT07X355, the Pearl River Talent Recruitment Program under Grant 2017GC010465, and the project of "FANet: PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications" under Grant LZC0019.

References

[1] CBInsights, "Market sizings of edge computing," 2018. [Online]. Available: https://www.cbinsights.com/market-sizings

[2] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, May 2016.

[3] L. Xiao, Y. Li, X. Huang, and X. Du, "Cloud-based malware detection game for mobile devices with offloading," *IEEE Trans. Mobile Comput.*, vol. 16, no. 10, pp. 2742–2750, Oct. 2017.

[4] L. Xiao, X. Wan, C. Dai, X. Du, X. Chen, and M. Guizani, "Security in mobile edge caching with reinforcement learning," *IEEE Wireless Commun.*, vol. 25, no. 3, pp. 116–122, Jun. 2018.

[5] M. Min *et al.*, "Learning-based privacy-aware offloading for healthcare IoT with energy harvesting," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4307–4316, Jun. 2019.

[6] T. Wu, W. Dou, Q. Ni, S. Yu, and G. Chen, "Mobile live video streaming optimization via crowdsourcing brokerage," *IEEE Trans. Multimedia*, vol. 19, no. 10, pp. 2267–2281, Oct. 2017.

[7] Y. Zhou, L. Chen, M. Jing, Z. Ming, and Y. Xu, "Performance analysis of thunder crystal: A crowdsourcing-based video distribution platform," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 28, no. 4, pp. 997–1008, Apr. 2018.

[8] M. Goudarzi, Z. Movahedi, and M. Nazari, "Efficient multisite computation offloading for mobile cloud computing," in *Proc. IEEE Conf. Ubiquitous Intell. Comput.*, 2016, pp. 1131–1138.

[9] Y. Sun, S. Zhou, and J. Xu, "EMM: Energy-aware mobility management for mobile edge computing in ultra dense networks," *IEEE J. Sel. Areas Commun.*, vol. 35, no. 11, pp. 2637–2646, Nov. 2017.

[10] M. Hu, L. Zhuang, D. Wu, Y. Zhou, X. Chen, and L. Xiao, "Learning driven computation offloading for asymmetrically informed edge computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 8, pp. 1802–1815, Aug. 2019.

[11] X. Chen, "Decentralized computation offloading game for mobile cloud computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 4, pp. 974–983, Apr. 2015.

[12] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Trans. Netw.*, vol. 24, no. 5, pp. 2795–2808, Oct. 2016.

[13] D. Challet, M. Marsili, and Y. Zhang, *Minority Games: Interacting Agents in Financial Markets*. London, U.K.: Oxford Univ. Press, 2013.

[14] M. Hu, L. Zhuang, D. Wu, Z. Huang, and H. Hu, *Edge Computing for Real-Time Video Stream Analytics*. Cham, Switzerland: Springer, 2018, pp. 1–5.

[15] Z. Lu, K. S. Chan, and T. L. Porta, "A computing platform for video crowdprocessing using deep learning," in *Proc. IEEE Conf. Comput. Commun.*, 2018, pp. 1430–1438.

[16] Z. Lu, K. Chan, S. Pu, and T. L. Porta, "CrowdVision: A computing platform for video crowdprocessing using deep learning," *IEEE Trans. Mobile Comput.*, vol. 18, no. 7, pp. 1513–1526, Jul. 2019.

[17] T. S. Rappaport *et al.*, *Wireless Communications: Principles and Practice*, vol. 2, New Jersey, NJ, USA: Prentice Hall, 1996.

[18] W. Rhee and J. M. Cioffi, "On the capacity of multiuser wireless channels with multiple antennas," *IEEE Trans. Inf. Theory*, vol. 49, no. 10, pp. 2580–2595, Oct. 2003.

[19] D. Huang, P. Wang, and D. Niyato, "A dynamic offloading algorithm for mobile computing," *IEEE Trans. Wireless Commun.*, vol. 11, no. 6, pp. 1991–1995, Jun. 2012.

[20] Y. Mao, J. Zhang, and K. B. Letaief, "Dynamic computation offloading for mobile-edge computing with energy harvesting devices," *IEEE J. Sel. Areas Commun.*, vol. 34, no. 12, pp. 3590–3605, Dec. 2016.

[21] H. Guo and J. Liu, "Collaborative computation offloading for multiaccess edge computing over fiber090009wireless networks," *IEEE Trans. Veh. Technol.*, vol. 67, no. 5, pp. 4514–4526, May 2018.

[22] D. A. Popescu, N. Zilberman, and A. W. Moore, "Characterizing the impact of network latency on cloud-based applications' performance," Univ. Cambridge, Cambridge, U. K., Tech. Rep. UCAM-CL-TR-914, Nov. 2017.

[23] S. Ranadheera, S. Maghsudi, and E. Hossain, "Minority games with applications to distributed decision making and control in wireless networks," *IEEE Wireless Commun.*, vol. 24, no. 5, pp. 184–192, Oct. 2017.

[24] M. Marsili and Y.-C. Zhang, "Fluctuations around Nash equilibria in game theory," *Physica A-statistical Mechanics Appl.*, vol. 245, no. 1, pp. 181–188, 1997.

[25] M. C. Canellas, K. M. Feigh, and Z. K. Chua, "Accuracy and effort of decision-making strategies with incomplete information: Implications for decision support system design," *IEEE Trans. Human-Mach. Syst.*, vol. 45, no. 6, pp. 686–701, Dec. 2015.

[26] S. Ranadheera, S. Maghsudi, and E. Hossain, "Computation offloading and activation of mobile edge computing servers: A minority game," *IEEE Wireless Commun. Lett.*, vol. 7, no. 5, pp. 688–691, Oct. 2018.

[27] C. You, K. Huang, H. Chae, and B. Kim, "Energy-efficient resource allocation for mobile-edge computation offloading," *IEEE Trans. Wireless Commun.*, vol. 16, no. 3, pp. 1397–1411, Mar. 2017.

[28] X. Lyu, H. Tian, C. Sengul, and P. Zhang, "Multiuser joint task offloading and resource optimization in proximate clouds," *IEEE Trans. Veh. Technol.*, vol. 66, no. 4, pp. 3435–3447, Apr. 2017.

[29] P. Mertikopoulos and A. L. Moustakas, "The simplex game: Can selfish users learn to operate efficiently in wireless networks?" in *Proc. Int. Conf. Perform. Eval. Methodologies Tools*, 2007, pp. 1–10.

[30] Y. Tao, C. You, P. Zhang, and K. Huang, "Stochastic control of computation offloading to a helper with a dynamically loaded cpu," *IEEE Trans. Wireless Commun.*, vol. 18, no. 2, pp. 1247–1262, Feb. 2019.

[31] J. Ghaderi, "Randomized algorithms for scheduling VMs in the cloud," in *Proc. Annu. IEEE Int. Conf. Comput. Commun.*, 2016, pp. 1–9.

[32] S. Taherizadeh, V. Stankovski, and M. Grobelnik, "A capillary computing architecture for dynamic Internet of Things: Orchestration of microservices from edge devices to fog and cloud providers," *Sensors*, vol. 18, no. 9, 2018, Art. no. 2938.

[33] P. Stefanic *et al.*, "Switch workbench: A novel approach for the development and deployment of time-critical microservice-based cloud-native applications," *Future Gener. Comput. Syst.*, vol. 99, pp. 197–212, 2019.

[34] P. Kochovski, S. Gec, V. Stankovski, M. Bajec, and P. D. Drobintsev, "Trust management in a blockchain based fog computing platform with trustless smart oracles," *Future Gener. Comput. Syst.*, vol. 101, pp. 747–759, 2019.

[35] P. Kochovski, P. D. Drobintsev, and V. Stankovski, "Formal quality of service assurances, ranking and verification of cloud deployment options with a probabilistic model checking method," *Inf. Softw. Technol.*, vol. 109, pp. 14–25, 2019.

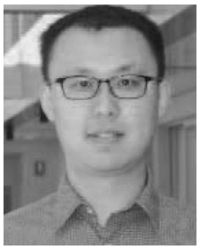
[36] P. Guan, X. Deng, Y. Liu, and H. Zhang, "Analysis of multiple clients behaviors in edge computing environment," *IEEE Trans. Veh. Technol.*, vol. 67, no. 9, pp. 9052–9055, Sep. 2018.

[37] Q. He *et al.*, "A game-theoretical approach for user allocation in edge computing environment," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 3, pp. 515–529, Mar. 2020.

Miao Hu (Member, IEEE) received the BS and PhD degrees in communication engineering from Beijing Jiaotong University, Beijing, China, in 2011 and 2017, respectively. He is currently an associate research fellow in computer science with the School of Data and Computer Science, Sun Yat-Sen University, Guangzhou, China. From 2014 to 2015, he was a visiting scholar with the Pennsylvania State University, PA. His research interests include edge computing and software defined networks.

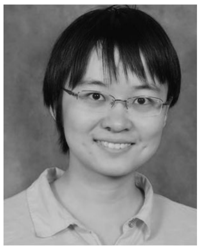
Zixuan Xie received the BE degree in software engineering from Northeastern University, Shenyang, China, in 2017. He is currently working toward the MS degree in the School of Data and Computer Science, Sun Yat-Sen University, Guangzhou, China. His research interests include edge computing and cloud computing.

Di Wu (Senior Member, IEEE) received the BS degree from the University of Science and Technology of China, Hefei, China, in 2000, the MS degree from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2003, and the PhD degree in computer science and engineering from the Chinese University of Hong Kong, Hong Kong, in 2007. He was a post-doctoral researcher with the Department of Computer Science and Engineering, Polytechnic Institute of New York University, Brooklyn, NY, from 2007 to 2009, advised by Prof. K. W. Ross. He is currently a professor and the assistant dean of the School of Data and Computer Science with Sun Yat-sen University, Guangzhou, China. His research interests include cloud computing, multimedia communication, Internet measurement, and network security. He was a co-recipient of the IEEE INFOCOM 2009 Best Paper Award. He has served as an editor of the *Journal of Telecommunication Systems* (Springer), the *Journal of Communications and Networks*, Peer-to-Peer Networking and Applications (Springer), *Security and Communication Networks* (Wiley), and the *KSII Transactions on Internet and Information Systems*, and a guest editor of the *IEEE Transactions on Circuits and Systems for Video Technology*. He has also served as the MSIG chair of the Multimedia Communications Technical Committee in the IEEE Communications Society from 2014 to 2016. He served as the TPC co-chair of the IEEE Global Communications Conference - Cloud Computing Systems, and Networks, and Applications, in 2014, the Chair of the CCF Young Computer Scientists and Engineers Forum - Guangzhou from 2014 to 2015, and a member of the Council of China Computer Federation.



Yipeng Zhou received the bachelor's degree in computer science from the University of Science and Technology of China (USTC). He is currently a lecturer in computer science with the Department of Computing at Macquarie University, and the recipient of ARC Discovery Early Career Research Award, 2018. From August 2016 to February 2018, he was a research fellow with the Institute for Telecommunications Research (ITR) with University of South Australia. From 2013 to 2016, He was a lecturer with the College of Computer Science and Software Engineering, Shenzhen University. He was a postdoctoral fellow with the Institute of Network Coding (INC) of The Chinese University of Hong Kong (CUHK) from 2012 to 2013. He won his PhD degree supervised by Prof. Dah Ming Chiu and Mphil degree supervised by Prof. Dah Ming Chiu and Prof. John C.S. Lui from Information Engineering (IE) Department of CUHK.

Xu Chen received the PhD degree in information engineering from the Chinese University of Hong Kong, in 2012. He is a full professor with Sun Yat-sen University, Guangzhou, China, and the vice director of the National and Local Joint Engineering Laboratory of Digital Home Interactive Applications. He was a post-doctoral research associate with Arizona State University, Tempe, from 2012 to 2014, and a Humboldt Scholar Fellow with the Institute of Computer Science, University of Goettingen, Germany, from 2014 to 2016. He was a recipient of the Prestigious Humboldt Research Fellowship awarded by Alexander von Humboldt Foundation of Germany, the 2014 Hong Kong Young Scientist Runner-Up Award, the 2016 Thousand Talents Plan Award for Young Professionals of China, the 2017 IEEE Communication Society AsiäCPacific Outstanding Young Researcher Award, the 2017 IEEE ComSoc Young Professional Best Paper Award, the Honorable Mention Award of 2010 IEEE international conference on Intelligence and Security Informatics, the Best Paper Runner-Up Award of 2014 IEEE International Conference on Computer Communications (INFOCOM), and the Best Paper Award of 2017 IEEE International Conference on Communications. He is currently an area editor of IEEE Open Journal of the Communications Society, an associate editor of the *IEEE Transactions Wireless Communications*, the *IEEE Internet of Things Journal* and the *IEEE Journal on Selected Areas in Communications* (JSAC) Series on Network Softwarization and Enablers.



Liang Xiao (Senior Member, IEEE) received the BS degree in communication engineering from the Nanjing University of Posts and Telecommunications, China, in 2000, the MS degree in electrical engineering from Tsinghua University, China, in 2003, and the PhD degree in electrical engineering from Rutgers University, NJ, in 2009. She is currently a professor with the Department of Communication Engineering, Xiamen University, Xiamen, China. She has served in several editorial roles, including an associate editor of the *IEEE Trans. Information Forensics & Security* and IET Communications. Her research interests include wireless security, smart grids, and wireless communications. She won the best paper award for 2016 IEEE INFOCOM Bigsecurity WS. She was a visiting professor with Princeton University, Virginia Tech, and University of Maryland, College Park.

Minority Disk Failure Prediction Based on Transfer Learning in Large Data Centers of Heterogeneous Disk Systems

Ji Zhang, Ke Zhou, *Member, IEEE*, Ping Huang, Xubin He, *Senior Member, IEEE*, Ming Xie, Bin Cheng, Yongguang Ji, and Yinhu Wang

Abstract—The storage system in large scale data centers is typically built upon thousands or even millions of disks, where disk failures constantly happen. A disk failure could lead to serious data loss and thus system unavailability or even catastrophic consequences if the lost data cannot be recovered. While replication and erasure coding techniques have been widely deployed to guarantee storage availability and reliability, disk failure prediction is gaining popularity as it has the potential to prevent disk failures from occurring in the first place. Recent trends have turned toward applying machine learning approaches based on disk SMART attributes for disk failure predictions. However, traditional machine learning (ML) approaches require a large set of training data in order to deliver good predictive performance. In large-scale storage systems, new disks enter gradually to augment the storage capacity or to replace failed disks, leading storage systems to consist of small amounts of new disks from different vendors and/or different models from the same vendor as time goes on. We refer to this relatively small amount of disks as minority disks. Due to the lack of sufficient training data, traditional ML approaches fail to deliver satisfactory predictive performance in evolving storage systems which consist of heterogeneous minority disks. To address this challenge and improve the predictive performance for minority disks in large data centers, we propose a minority disk failure prediction model named *TLDFP* based on a transfer learning approach. Our evaluation results in two realistic datasets have demonstrated that *TLDFP* can deliver much more precise results and lower additional maintenance cost, compared to four popular prediction models based on traditional ML algorithms and two state-of-the-art transfer learning methods.

Index Terms—Disk failure, machine learning, transfer learning, cloud computing, data center

1 Introduction

Hard disks are widely used as the common and primary storage devices for large-scale storage systems in modern data centers. In such data centers, it has been an extremely challenging undertake to ensure high availability and reliability for IT management, as various disk failures constantly occur in the field, whether being hard disks [2], [3], [4], flash-based SSDs [5], [6] or NVMes. Disk failures can lead to temporary data loss and thus system unavailability or even permanent data loss if the lost data cannot be recovered by existing data protection schemes, e.g., replication and erasure codes [7], [8] due to disk failures exceeding the designed correction capability. A disk is a rather complex system consisting of a variety of magnetic, mechanical, and electronic components, each of which could fail. As a result, disk failures show different manifestations and extents of severeness [9] for numerous reasons, which has been observed in data centers from major IT companies [10], [11]. Compared with the traditional passive fault tolerance techniques like Erasure Code (EC) and Redundant Arrays of Independent Disks (RAID) [12], proactive disk failure prediction tends to ensure the reliability and availability of large-scale storage systems in advance. Therefore, successful disk failure prediction not only reduces the risk of losing data but also reduces the data recovery cost (i.e., network bandwidth) associated with recovering the data residing on failed disks.

Disk manufacturers implement the self-monitoring, analysis and reporting technology (SMART) technology [13] in the disk firmware. Most of the SMART attributes contain information about gradual degradations and possible defects of disks. Internally, a disk uses the so-called *"threshold method"* [14] based on SMART values to claim its failure status, which means the hard disk would raise an alarm if the value of an SMART attribute crosses the corresponding predefined threshold. However, this *"threshold method"* only achieves a failure detection rate (FDR) of 3-10 percent with 0.1 percent false alarm rate (FAR) [14] (FDR and FAR see Section 5.1.2 in detail). In other words, these numbers highlight the

- *J. Zhang and K. Zhou are with the Wuhan National Laboratory for Optoelectronics (Huazhong University of Science and Technology), Key Laboratory of Information Storage System, Intelligent Cloud Storage Joint Research Center, Huazhong University of Science and Technology, Wuhan 430074, China, and also with Tencent, Shenzhen 518057, China. E-mail: {jizhang, k.zhou}@hust.edu.cn.*
- *P. Huang and X. He are with Temple University, Philadelphia, PA 19122. E-mail: {templestorager, xubin.he}@temple.edu.*
- *M. Xie, B. Cheng, Y. Ji, and Y. Wang are with Tencent Inc., Shenzhen 518057, China. E-mail: {reganxie, bencheng, raidmanji, yhwang}@tencent.com.*

Manuscript received 30 Aug. 2019; revised 13 Feb. 2020; accepted 21 Mar. 2020. Date of publication 6 Apr. 2020; date of current version 1 May 2020.
(Corresponding author: Ke Zhou.)
Recommended for acceptance by K. Mohror.
Digital Object Identifier no. 10.1109/TPDS.2020.2985346

conservative nature of this method, i.e., it would rather miss chances to detect more disk failures than report false alarms at a higher rate.

To improve the predictive performance, several machine learning (ML) algorithms based disk failure prediction models [15], [16], [17], [18] have been proposed, which leverage training SMART data to predict disk failures. Unfortunately, these works focused on a large number of homogeneous disks which have sufficient training data. In large-scale storage system scenarios, bunches of new disks enter gradually to replace failed disks, resulting in storage systems consisting of heterogeneous disks from different vendors and different models from the same vendor as time goes on. Heterogeneous disks with numerous disk models are common in data centers [19], [20], [21]. Moreover, in evolving storage systems, some disk models are dramatically fewer than others and we call this relatively small amount of disks *minority* disks (conversely the large amount of disks as *majority* disks) in large data centers of heterogeneous disk systems. We found that about 25 percent of disks with numerous models (more than 50) are minority disks in two real-world data centers as detailed in Section 3.1. Due to the small sample and insufficient training data of minority disks, traditional ML algorithms using the training data of minority disks would dramatically increase the risk of overfitting (Section 3.1) or poor generalization [22] which will weaken the performance of predictive models and seriously affect the reliability of the storage system. Therefore, we are poised to develop a disk failure prediction model *TLDFP* to predict failures for minority disks under the condition of having rich heterogeneous disk datasets. Our basic idea is to predict minority disk failures from the available majority disk datasets, which is an application of transfer learning.

In this paper, we aim to seek answers to the following problems: *(1) What* is the definition of a minority disk dataset as far as failure predication is concerned? *(2) Why* should we use transfer learning for minority disks failure prediction? *(3) How* to use transfer learning methods to predict minority disks failure? *(4) When* to use transfer learning for minority disks failure prediction? Besides, when applied to three real-world datasets from the public *Backblaze* and *Tencent* which is one of the largest social network companies in the world, our method *TLDFP* achieves on average 96 percent failure detection rate with 0.5 percent false alarm rate based on 5 disk manufacturers (Hitachi, Seagate, Hitachi Global Storage Technologies and Western Digital) in 3 different storage media (HDD, SSD and NVMe) when making cross-disk models failure prediction in addressing realistic system challenges.

2 Background and Related Work

2.1 SMART Technology

Almost all hard disk drives, flash-based SSDs and NVMes come with built-in Self-Monitoring, Analysis and Reporting Technology, which are indicators of disk health status. The specification of SMART technology contains up to 30 attributes, reporting various disk operating conditions. SMART data directly or indirectly reflects the health condition of disks and even contains some statistical information. SMART data can be obtained through specified disk protocols upon which the disk manufacturer reached agreement. The disk would raise an alarm if the value of an SMART attribute crosses the corresponding predefined threshold. Each SMART attribute entry consists of five elements described as a tuple *(ID, Normalized, Raw, Threshold, Worst)*.

- *ID:* The designated sequence number of the SMART attribute.
- *Normalized:* Current or last normalized value (most are normalized to a value between the best value 253 and the worst value 1 calculated by manufacturer-specific algorithms using its raw value).
- *Raw:* The original value corresponding to counts or physical states provided by a sensor and vendor-specific.
- *Threshold:* The threshold value beyond which a disk alarms a failure.
- *Worst:* The lowest or worst value for a given attribute.

Not all five elements in a tuple are used. In our paper, we focus on the first three elements $(ID, Normalized, and Raw)$ in our collected datasets. For convenience, we use *"smart_ID_Raw: V"* to denote the raw value of a SMART attribute whose ID is V. For example, *smart_1_Raw: 10* means that the raw value of the read error rate attribute (ID: 1) is 10 and *smart_5_Normalized: 56* means that the normalized value of the reallocated sectors count attribute (ID: 5) is 56. More specific information about the SMART attributes we use in our evaluation is given in Table 6.

2.2 Large Scale SMART Data Collection

We not only used the publicly available SMART datasets from *Backblaze*,[1] but also collected the real-world datasets from *Tencent* Inc.,. *Tencent* Inc., founded in November 1998, is currently one of the world's biggest internet companies. In China, *Tencent* has four major data centers located in Shenzhen, Tianjin, Shanghai and Chongqing, respectively. Among them, Tianjin data center is the largest in Asia. According to *Tencent*'s statistics, as of 2019, *Tencent* hosts a total of more than 700 thousand servers and 8 million disks. In the *Tencent* Cloud Foundation Department where we obtained our SMART dataset in this paper, there are about 400 thousand servers (5 million disks) supporting a variety of business applications, such as WeChat, Qcloud (Cloud of *Tencent*), TencentVideo, and the QQ photo store QQphoto.

We configure the collection interval as one hour. In practice, the size of one sampled SMART data per disk is about 2 KB, which results in 201.4 GB data per day. Moreover, suppose the disk life is about 4-5 years, then we need to store 350.1 TB SMART data for a period of five years. In each hour, nearly 400 thousand servers send their SMART data to one server, which parses, processes and stores massive amounts of data at the same time. It imposes a great challenge to collect and store such a large-scale set of SMART data. To well handle this situation, we have proposed a scalable framework for collecting large-scale SMART data and it has been deployed in the *Tencent*'s data centers in practice.

1. https://www.backblaze.com/b2/hard-drive-test-data.html

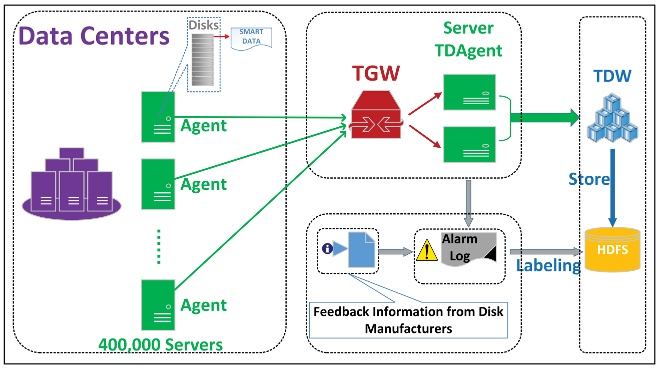


Fig. 1. The scalable SMART data collection framework deployed in the *Tencent* Inc. data center.

As it is shown in Fig. 1, the scalable SMART data collection framework mainly contains five components:

- *Agent*: An agent runs on the same server with multiple disks, typically 11 disks in *Tencent*'s servers. It periodically issues protocol-compliant requests to disks to obtain SMART data and sends SMART data files to two dedicated servers via *Tencent* Gate Way (*TGW*).
- *TGW*:[2] Due to all servers in *Tencent* data centers not locating in the same network and high concurrency easily happening, we leverage the open source technology Tencent Gate Way to forward data file send requests. *TGW* enables multi-network unified accesses and also supports automatic load balancing. Upon receiving SMART data files from agents, *TGW* forwards them to one of two dedicated servers in a random manner (For illustration purpose, we draw two servers in the figure. However, *TGW* can easily scale up to connect more servers as needed).
- *TDAgent*: A dedicated server which receives raw binary SMART data files from *TGW* and converts them to regular text files for data transformation (these files will remain for 3 days temporarily). After this processing, it then periodically forwards the resultant text files to *TDW* for further storage and analysis.
- *Alarm Log*: The extracted disk alarm events and the feedback information from disk manufacturers, which are used for disk labeling. This alarm log is important for us to redeploy the collected data in new settings and to label disks in supervised learning.
- *TDW*:[3] *Tencent* Distributed Data Warehouse *(TDW)*, is an open-source system of distributed data processing based on Hadoop, Hive and PostgreSQL which provides massive data storage and analysis functionality.

The framework of large-scale SMART data collection we proposed can effectively solve the problem of mass storage and difficult retrieval. Besides, these key technologies of our framework are all open source, thus more conducive to achieve and apply. Last but not the least, the high-quality SMART data is essential for a firm foundation for modeling of disk failure prediction.

2.3 Related Work

There have proposed a host of ML algorithms for disk failure prediction models based on SMART data. Hamerly and Elkan [23] employ two Bayesian methods to model disk failure based on SMART data from Quantum Inc., which consists of 1,927 good drives and 9 failed drives. They categorize the problem as anomaly detection and establish a mixture model called *NBEM*, short for Naive Bayes clusters trained using expectation-maximization and another method called naive Bayes classifier. They achieve failure detection rates of 35-40 percent for *NBEM* and 55 percent for naive Bayes classifier at about 1 percent *FAR*. Hughes *et al.* [24] explore two statistical methods to improve predictive performance. They explore the capabilities of statistical tests like the rank sum test and OP-ed single variate test, and test both methods with 7,744 drives data (out of which 36 are failures) from two different disk models spanning across a period of 3 months. They achieved a failure detection rate (*FDR*) of 60 and 0.5 percent false alarm rate (*FAR*). Murray *et al.* [25] compare the predictive performance of Support Vector Machine (SVM), rank-sum test, unsupervised clustering and reverse arrangements test.

Zhu *et al.* [15] explore the capability of a Backpropagation (BP) neural network and an improved SVM model to establish the prediction model based on SMART data. Many researchers use a SVM [26] because they claim SVM can efficiently perform a non-linear classification using the kernel trick, implicitly mapping their inputs into high-dimensional feature spaces [27], [28]. In order to improve the stability and interpretability of the disk failure prediction model, Li *et al.* [29] propose a new hard drive failure prediction model based on Classification And Regression Trees (CART). The Regression Tree can give the disk a heath assessment rather than a simple classification result. Gradient Boosted Regression Tree (GBRT [30]) has been proposed to model disk failure [31], [32], where GBRT is a gradient descent boosting technique based on tree averaging, and is an accurate and effective ML method that can used for both regression and classification problems. To avoid over-fitting, the GBRT algorithm trains many tree stumps as week learners, rather than full, high variance trees. Moreover, the Regularized Greedy Forests (RGF [33]) approach is a powerful, non-linear classification method. It is a variation of GBRT in which the structure search and optimization are decoupled and it utilizes the concept of structured sparsity to perform greedy search directly over the forest nodes based on the forest structure. Mirela Madalina Botezatu *et al.* employ this method to model disk failure and achieve good results [34]. Xu *et al.* [35] present a Recurrent Neural Networks (RNN [36], [37]) method to leverage sequential information for predicting hard disk failure. They use a dataset collected from a real-word data center containing 3 different disk models represented as W, S and M and establish the prediction model for those disk models, respectively. They model the long-term dependent sequential SMART data and demonstrate the capability of their predictive model. More recently, Mahdisoltani *et al.* [38] propose to use traditional ML algorithms to predict disk sector errors using SMART

2. http://wiki.open.qq.com/wiki/TGW
3. https://github.com/tonycody/tencent-tdw

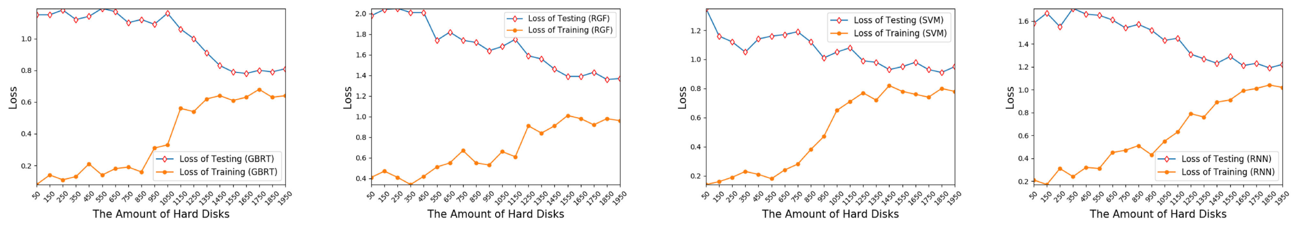


Fig. 2. The training loss and testing loss of four popular traditional ML algorithms. Note that the y-axis means loss values as the dataset size increases.

datasets. Our goals in this paper are to make whole disk failure predictions which require much higher accuracy due to cost consideration. *Note that all these studies focus only on HDDs and none of them has investigated NVMe SSDs.*

As previously mentioned, the need for transfer learning occurs when there is a limited availability of training data from a new disk model, which regularly happens to evolving storage systems. With big data repositories becoming more prevalent, using existing datasets that are related to, but not exactly the same as, a target domain of focus point or interest makes transfer learning solutions an attractive approach. There are various applications in which transfer learning has been successfully applied to, including multi-language text classification [39], [40], image classification [41], human activity classification [42], text sentiment classification [43], Web document classification [44] and so on. Unsurprisingly, in recent years, researchers have started to use transfer learning method to solve minority disks failure prediction problems [20], [34]. Mirela Madalina Botezatu *et al.* proposed the sample selection de-biasing method [34], which we denoted as *SSDB* in our paper. Its main idea is to train a classifier that can rank the observations linked to a specific disk model based on their similarity to samples pertaining to the target disk model. This method is also a single-source domain transfer learning method in spirit similar to *TLDFP* algorithm. FLF Pereira *et al.* proposed the multi-source domain transfer learning for Bayesian network [20], which we denoted as *TLBN* in our paper. It proposes a new source building method called clustering-based information source and groups several HDDs according to their similarity to build a novel information source for transfer learning. Although these methods also provide a solution to minority disks failure prediction, we have compared *TLDFP* with them and show our approach delivers better predictive performance. Besides, note that our work is the first to systematically (*What, Why, How and When*) propose using the transfer learning method to solve minority disks failure prediction based on SMART attributes for large-scale, active, evolving storage systems.

3 Preliminary Study and Motivation

In this section, we define minority disk datasets via experimental examination, investigate the distributions of SMART data, and justify *why* we use transfer learning for minority disk failure prediction.

3.1 Minority Disk Datasets

As mentioned previously, we aim to improve disk failure predictive performance for a disk dataset which has insufficient training data and where traditional ML algorithms deliver suboptimal performance. In this section, we give the definition of a *Minority Disk Dataset* and quantitatively evaluate them via experimentally showing the *training loss* and *testing loss* [45] of four popular ML algorithms (GBRT, RGF, SVM, and RNN) in disk failure prediction [28], [31], [34], [35]. A loss is a number indicating how bad a model's prediction. Note that we have added a regularization term to construct the loss function in all four methods which can effectively prevent over-fitting caused by the model having a very large number of parameters. Fig. 2 illustrates the results. As can be seen from the figure, with the dataset increasing, the loss of training set increases to a certain extent while the loss of testing set decreases because the increase in dataset leads to more complex situations where the training model needs to be fitted. More specifically, when the amount of disks is less than 1,500, the gap between the loss of training and testing decreases as the dataset enlarges, which is called *over-fitting* caused by minority disks. When the amount of disks goes over 1,500, the gap becomes smaller and stabilizes. Therefore, we can draw the following conclusions: (1) A disk dataset containing fewer than 1500 disks could lead to over-fitting which we name it as *Minority Disk Datasets*; (2) The four popular traditional ML algorithms cannot deliver satisfactory performance when the dataset contains fewer than 1,500 disks. As far as we know, we are the first to define minority disk datasets and quantitatively evaluate them through extensive data analysis and experiments. We studied two real data centers and categorized the disk quantities by the threshold of 1500. As shown in Table 1, in data center *BackBlaze*, *91* different disk models only account for less than *24 percent* of all disks while 12 models account for more than 76 percent. We call these 91 models minority disks. A similar observation has been found in data center of *Tencent*.

The above description and analysis implies that making disk failure prediction for minority disks is a realistic problem that needs to be resolved.

3.2 The Baseline Results of Using Traditional ML Only Trained on Minority Disk Datasets

In order to investigate the predictive results of using traditional ML methods *only* trained on minority disk datasets, we use 7 minority disk models from 5 disk manufacturers

TABLE 1
Characteristics of Disk Population

Data Center	Disk Number	Disk Models	Total Number	Percentage
Backblaze	$\geq$ 1500	12	114,570	76.61%
	$<$ 1500	**91**	34,978	**23.39%**
Tencent	$\geq$ 1500	8	52,235	73.32%
	$<$ 1500	**52**	18,996	**26.67%**

TABLE 2
The Results of Minority Disk Failure Prediction via Traditional ML on SATA SSDs From *STX* and *WDC* and NVMe SSDs From *SAMSUNG*

Type	Methodology	Manufacturer	FDR	FAR
HDD	GBRT	*HDS/STX/HGST/WDC*	27.3%/37.5%/31.6%/38.5%	29.0%/19.4%/17.6%/21.8%
	RGF	*HDS/STX/HGST/WDC*	36.4%/50.0%/47.4%/53.8%	44.3%/22.4%/ 53.4%/36.6%
	SVM	*HDS/STX/HGST/WDC*	50.0%/41.7%/57.9%/30.8%	20.7%/47.8%/ 24.0%/43.7%
	RNN	*HDS/STX/HGST/WDC*	40.9%/33.3%/36.8%/30.8%	28.3%/31.3%/ 39.7%/38.7%
SSD	GBRT	*STX/WDC/SAMSUNG*	23.7%/38.2%/52.8%	25.5%/27.2%/40.1%
	RGF	*STX/WDC/SAMSUNG*	35.6%/46.0%/60.4%	20.5%/32.7%/30.1%
	SVM	*STX/WDC/SAMSUNG*	15.6%/39.5%/47.2%	16.3%/41.4%/22.4%
	RNN	*STX/WDC/SAMSUNG*	40.7%/52.6%/62.3%	14.3%/22.5%/30.9%

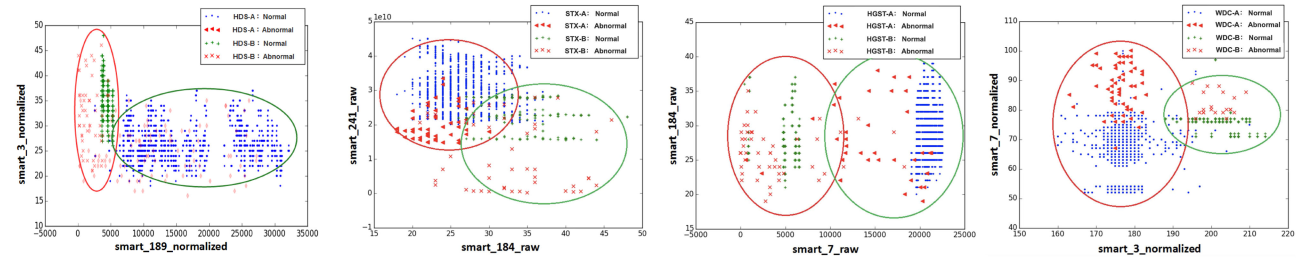
to conduct this experiment based on four popular traditional ML methods: GBRT, RGF, SVM and RNN which include the popular tree structure algorithms and deep learning algorithms and have been commonly used in disk failure prediction. Note that we only use 70 percent of the minority disk datasets as training sets and the remaining 30 percent as testing sets without any other datasets. Besides, we consistently use the following acronyms for these five vendors throughout the paper which are also used in their disk models: Hitachi (*HDS*) and Seagate (*STX*) from *Backblaze*, Hitachi Global Storage Technologies (*HGST*), Western Digital (*WDC*) and *SAMSUNG* come from *Tencent*. As can be seen from the Table 2, none of the four traditional ML methods can deliver a high *FDR* and low *FAR*. We know the poor predictive performance due to over-fitting caused by using small homogeneous datasets based on traditional ML.

We have also conducted experiments to directly use large datasets of the available majority disks to predict minority disk failures based on the four popular traditional ML techniques, the performance is not satisfactory either (details are shown in Fig. 6 in Section 5). To understand the reason, we analyze the SMART data distributions as below.

3.3 SMART Data Distributions

It is interesting to observe that the values of SMART attributes indicating disk health conditions of different disk models from the same manufacturer exhibit similar distribution patterns. We have analyzed both the publicly available SMART dataset from *Backblaze* and a dataset from the data center of *Tencent*. Fig. 3 shows the revealed SMART data distribution patterns. Each subfigure shows a pair of SMART attribute values' distribution pattern (we also investigate many other different SMART attributes, which lead to similar results) of two different disk models from the same manufacturer and each circle is used to highlight different disk models. As it is evidently shown, the relationship between *Abnormal* and *Normal* states indicated by the two SMART attributes of two disk models shows a similar pattern, with only the difference of SMART values being in different ranges. Figs. 3a, 3b, 3c and 3d respectively show that the *Abnormal* state is above, below, and to the left of the *Normal* state for the two disk models from the same manufacturer. Furthermore, the SMART values are distributed in different spectrums. Take Fig. 3b Seagate as an example, the distribution region of model *STX-B* is right-lower than that of model *STX-A*. Note that even if we ignore the y-axis (smart_241_Raw, LBAs written), there are still differences in the distribution of SMART data from these two different disk models (the distribution region of model *STX-B* is right to that of model *STX-A*). Traditional ML algorithms deliver good predictive performance only when both training and testing data are drawn from the same distribution [46]. Therefore, they fail to perform satisfactorily when it comes to cross-disk models failure prediction due to different distribution spectrums as revealed in Fig. 3.

In order to take an in-depth look at the distributions of SMART values of different disk models from the same manufacturer and further motivate the transfer learning from one disk model to a different model and explain why we use transfer learning for minority disks failure prediction, we investigate the differences in the distributions of SMART data and present a intuitive analysis comparing different disk models from the same manufacturer. Probability Density



(a) *HDS* (b) *STX* (c) *HGST* (d) *WDC*

Fig. 3. The distributions of two SMART attributes of two disk models from four manufacturers, i.e., *Hitachi*, *Seagate*, *HGST*, and *WDC*. Each subfigure shows the *Abnormal* and *Normal* states indicated by a randomly chosen pair of SMART attributes of two disk models. These four subfigures indicate that two disk models of each manufacturer exhibit similar failure patterns represented by the two SMART attributes distributions and the SMART data are distributed in different value ranges, which motivates us to apply transfer learning to make cross-model disk failure predictions.

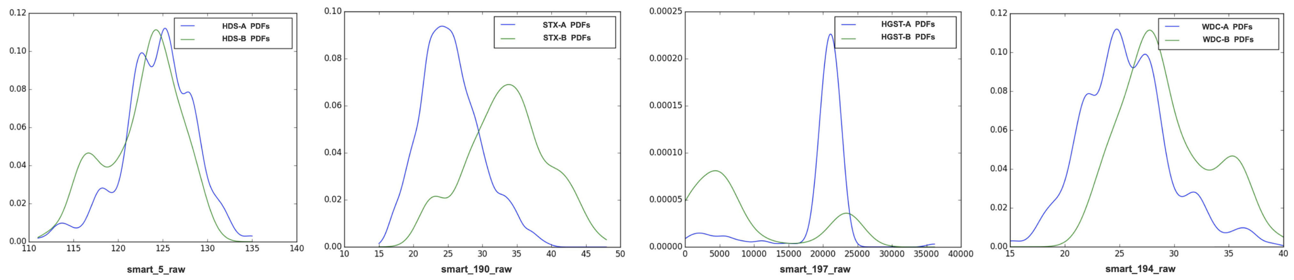


Fig. 4. PDFs of a SMART attribute value of two different disk models from four manufacturers.

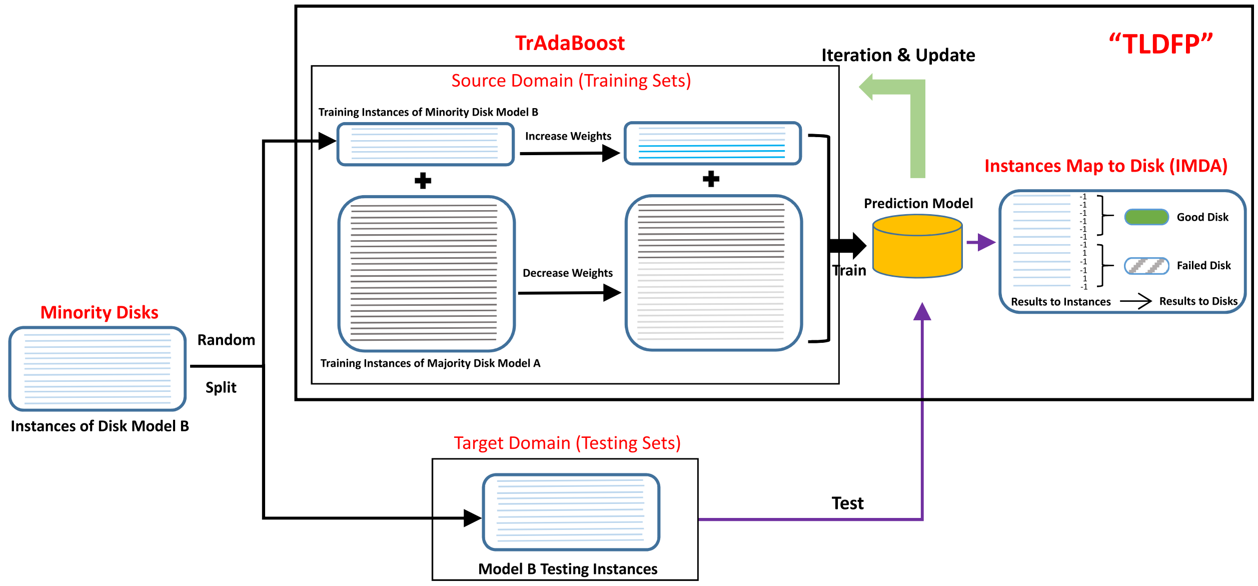
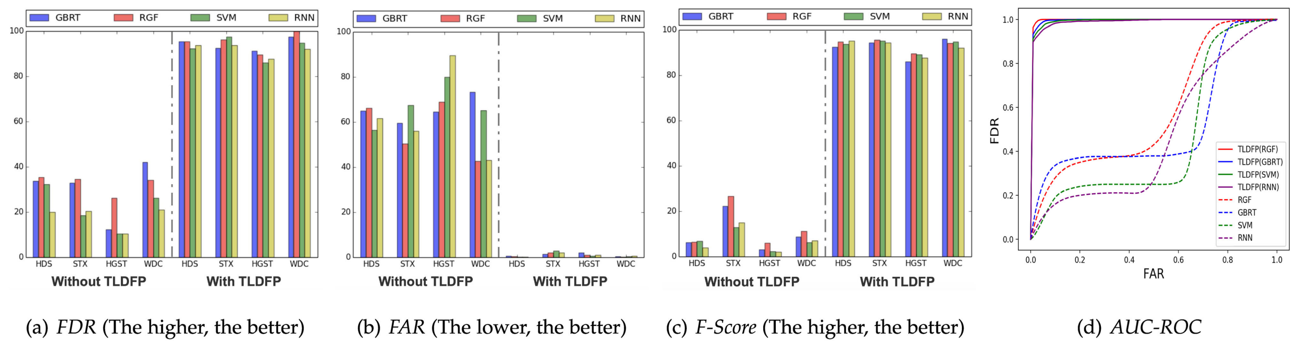

TrAdaBoost
"TLDFP"
Iteration & Update
Source Domain (Training Sets)
Training Instances of Minority Disk Model B
Increase Weights
Decrease Weights
Training Instances of Majority Disk Model A
Prediction Model
Train
Instances Map to Disk (IMDA)
Good Disk
Failed Disk
Results to Instances
Results to Disks
Minority Disks
Random
Split
Instances of Disk Model B
Target Domain (Testing Sets)
Test
Model B Testing Instances

Fig. 5. The overall structure of *TLDFP*, which contains the transfer learning algorithm *TrAdaBoost* and the *Instances Map to Disk Algorithm (IMDA)*. Source domain contains a fully labeled dataset of majority disk model A and a small portion of labeled dataset of minority disk model B. The testing data in the target domain is the remaining unlabeled dataset of minority disk model B.



(a) *FDR* (The higher, the better) (b) *FAR* (The lower, the better) (c) *F-Score* (The higher, the better) (d) *AUC-ROC*

Fig. 6. The results of *FDR*, *FAR*, *F-Score* and *AUC-ROC* using four disk models based on *TLDFP* compared to four traditional ML methods.

Function statistic (PDFs) is frequently used to describe the intensity of continuous random variable. For easy observations, we use the Gaussian Kernel Density Estimation (GKDE) as the kernel function, which results in smooth curves.

Fig. 4 shows the PDFs of the value of a SMART attribute of two models from four manufactures. The distributions of the SMART data of two disk models are different but similar in that they show similar spikes though at different points and magnitudes. We refer to this phenomenon as *covariate shift* [47] among relevant predictors between different models from the same manufacturer. Therefore, we conclude that different disk models from the same manufacturer exhibit varying SMART value distributions. For the problem of minority disks failure prediction, the implication is that a prediction model built upon traditional ML algorithms using training data from one disk model is not applicable to other different models even from the same manufacturer. Therefore, to leverage a prediction model for a disk model built on adequately sufficient SMART training data to build a predictive model for a different model for which there is only limited training data, we could employ transfer learning algorithm which is inherently suitable for transferring health state information from one disk model to another disk model from the same manufacturer.

Considering the above regularity of relationship between *Abnormal* and *Normal* disk states and varying SMART data distribution spectrums across different disk models, we are motivated to apply transfer learning to predict disk failures for minority disks using the knowledge from the majority disks, which we name as *TLDFP*.

4 Minority Disk Failure Prediction

In this section, we answer the questions of *how and when* to use transfer learning for minority disk failure prediction. Specifically we detail our transfer learning method for minority disks failure prediction *TLDFP*, followed by our method of selecting source domain based on *KLD*.

4.1 *TLDFP*: Transfer Learning for Minority Disk Failure Prediction

We elaborate on *how* to use transfer learning to predict minority disk failure in this section. Fig. 5 illustrates the overall structure of our proposed predictive method *TLDFP*. It main consists of two components: a transfer learning algorithm *TrAdaBoost* [48] and an instance map to disk algorithm *IMDA*. Note that we randomly divide the SMART data of minority disk model B into two parts. The first part includes a small portion (e.g., 10 percent) of the labeled target domain data, which is then put together with the data of majority disk model A as a combined source domain to establish the relationship between the two different disk models so as to reduce variation between their distributions. The other part contains the remaining unlabeled data as testing data. Then we use our *TLDFP* method to establish a predictive model and make failure prediction for the minority disk model B training data. With the above description, the problem we aim to solve in this paper can then be formally defined as: given enough labeled training data S_a, a small amount of labeled training S_b and unlabeled testing data T_b, the main objective is to leverage the useful portions of S_a and S_b and train a classifier C which achieves a good performance of classifying the unlabeled training set T_b. The *TrAdaBoost* is an extension of the traditional ML method *AdaBoost*. *AdaBoost* is an iterative algorithm and its key procedure includes training several different weak classifiers with different weights and then consolidating those weak classifiers to a strong classifier to boost predictive performance. According to the *AdaBoost* algorithm, it first gives an initial weight to all training instances. When an instance in the source domain is found to be misclassified, we consider this instance as difficult to classify and thus increase its weight. In this way, the significance of this instance will become greater in the next iteration. However, *AdaBoost* is a traditional ML method that can only build effective predictive model for testing data which has the same distribution as the training data. In the transfer learning algorithm *TrAdaBoost*, when an instance of disk model B from the combined source domain is misclassified, we increase the weight of this instance in the next iteration, which is similar to *AdaBoost*. However, when an instance of disk model A is mispredicted, the instance is assumed to be different from disk model B. Therefore, unlike *Adaboost*, it decreases the weight of that instance in the next iteration to reduce its influences on the target domain. The details of *TrAdaBoost* are showed in Algorithm 1.

Algorithm 1. TrAdaBoost Algorithm

```
Require:
    The source domain containing labeled disk model $A$ data $S_a = \{a_1, a_2, \dots, a_n\}$, a small amount of labeled disk model $B$ data $S_b = \{b_{n+1}, b_{n+2}, \dots, b_{n+m}\}$, the target domain containing all unlabeled disk model B data $T_b$, and the maximum iteration $T$.
    Note: $n$ is the number of disk model $A$ data and $m$ is the number of disk model $B$ data in the source domain
1:  Begin;
2:  Initialize weights distribution $W_j^t$, where $t$ is the sequence of iteration, i.e., $W^1 = \{w_1^1, w_2^1, \dots w_{n+m}^1\}$:
        $w_j^1 = \begin{cases} 1/n, & j = 1, 2, \dots, n \\ 1/m, & j = n, n+1, \dots, n+m \end{cases}$    (1)
3:  Set $\varphi = 1/(\sqrt{2 \ln n / T} + 1)$;
4:  for $t = [1, T]$ do
5:      Set the weights of instances as:
            $I^t = \frac{W^t}{\sum_{j=1}^{j=n+m} w_j^t}$    (2)
6:      Apply the basic learner to $S_a$ and $S_b$ with weights of instances $I^t$, and also to the unlabeled target domain disk model $B$ dataset $T_b$. We can achieve a classifier $h_t$;
7:      Calculate the error rate of $h_t$ on the labeled source domain disk model $B$ datasets $S_b$, note that $c(x_j)$ is the true label of the $j$th SMART sample $x_j$ :
            $\epsilon_t = \sum_{j=n+1}^{n+m} \frac{|c(x_j) - h_t(x_j)| w_j^t}{\sum_{j=n+1}^{n+m} w_j^t}$    (3)
8:      Set $\varphi_t = \epsilon_t/(1 - \epsilon_t)$. Note that if $\epsilon_t$ is greater than $1/2$, it needs to be reset to $1/2$;
9:      Update the weight distributions as below:
            $w_j^{t+1} = \begin{cases} w_j^t \varphi^{|c(x_j) - h_t(x_j)|}, & j = 1, \dots, n \\ w_j^t \varphi_t^{|c(x_j) - h_t(x_j)|}, & j = n, \dots, n+m \end{cases}$    (4)
10: end for
11: Map the instance to disk results using IMDA;
Ensure: Classify the disk as good(-1) or failed(1);
```

The input of the *TrAdaBoost* algorithm includes two disk models' training and testing data, and the maximum number of iterations T. It initializes the weights of training data and performs the iteration process. In each cycle, we use the basic learner, such as GBRT, RGF, SVM, RNN, and the weight distribution I^t to build a classifier h_b on testing data and calculate the error rate on the labeled source domain disk model B dataset S_b. Lastly, we set the new weights based on the previous iteration results and the error rate. Note that if majority disk model A instances of the source domain are misclassified, they are considered to be different from the minority disk model B. As a result, we reduce the weights of these instances in order to reduce their influences

TABLE 3
The *KLD* Values of the PDFs in Fig. 4

Source Domain	Target Domain	SMART Attribute	KLD
$HDS-A$	$HDS-B$	5_RAW	0.61
$STX-A$	$STX-B$	190_RAW	0.89
$HGST-A$	$HGST-B$	197_RAW	1.35
$WDC-A$	$WDC-B$	194_RAW	0.56

on the predictive model in the next iteration. Specifically, we multiply the instances by $w_j^t\varphi^{|c(x_j)-h_t(x_j)|}$, where φ ranges from 0 to 1 and $c(\cdot)$ is the true label of a SMART attribute. On the other hand, if the disk model B instances in the combined source domain are misclassified, we increase the weights of these instances to gain more attention in the next iteration through multiplying these instances by $w_j^t\varphi_t^{|c(x_j)-h_t(x_j)|}$, where φ_t is greater than 1. After several iterations (we will investigate the impact of this value on the predictive performance in Section 6.4), the instances of majority disk model A in the source domain that fit minority disk model B will gain greater weights and those different from disk B will have smaller weights.

Since the inputs of the failure prediction model include many SMART instances from a lot of disks at different moments, each output result indicates the prediction result for a particular instance rather than the disk health state. Therefore, we need to map the results of multiple SMART instances to the final disk state. To achieve that, we propose an *Instances Map to Disk Algorithm* (*IMDA*) by extensive experiments and analysis using the majority disk dataset in the training progress. *IMDA* determines the final health state of a minority disk in testing progress (practical use). Specifically, it performs a prediction for each minority disk once every day, if any instance of one disk is predicted as a failure, the corresponding disk will be considered as failure. Besides, we discuss other alternative options are discussed in Section 6.5 and our method outperforms all others in terms of predictive result.

4.2 Source Domain Selection Based on KLD

When can we use *TLDFP* for minority disks failure prediction? To answer this question, we use Kullback Leibler Divergence (*KLD*), which is a metric measuring the divergence degree of one probability distribution from another expected probability distribution [49]. *KLD* values indicate the disparities between two random variable distributions. A zero *KLD* value means that the two random distributions are the same, while the *KLD* value increases as the differences between two random distributions widen. In general, the bigger a *KLD* value is, the greater differences between two distributions will be and the more difficult the knowledge transfer between two distributions will be. Table 3 gives *KLD* values corresponding the PDFs showed in Fig. 4. Table 3 shows that all *KLD* values are not equal to zeros, confirming that the respective SMART data distributions are indeed not the same. Note that as indicated in Table 3, the *KLD* value trend, which is consistent with the PDF differences increase from *HDS*, to *STX*, to *HGST*, to *WDC* shown in Fig. 4. Considering that *TLDFP* is a method to decrease the data distribution differences between source domain and target domain, so we infer that the bigger *KLD* value between one disk model and another is, the harder *TLDFP* can transfer experience. The predictive results in Section 5.2 proves our conjecture and we also conduct detailed experiments and discussions in Section 6.1. Therefore, the value of *KLD* can guide us to select appropriate majority disk dataset (source domain) for training the minority disk failure predictive model. As far as we know, we are the first attempt to present a novel method based on *KLD* values as an effective indicator to select proper majority disk models and improve disk failure prediction. Note that we also use the Jensen-Shannon Divergence (*JSD*) and Wasserstein Distance (*WD*) but find *KLD* is the best metric to guide us to select an appropriate majority disk dataset and results in better prediction performance. More specifically, *JSD* (*WD*) only achieves on average 79.6 percent (83.1 percent) *FDR* and 3.2 percent (2.1 percent) *FAR*. Our evaluation results in Section 6.1 demonstrate that our approach of using *KLD* is very effective and practical.

5 Experimental Evaluation

In this section, we evaluate the predictive performance of *TLDFP*. We first describe the methodology, followed by the experimental results of comparing *TLDFP* against four ML algorithms and two state-of-the-art transfer learning methods according to the evaluation metrics.

5.1 Methodology

We describe the characteristics of three real-world SMART datasets in our experiments and SMART attributes selection. Then we introduce four evaluation metrics commonly used in ML and some testing methods we use to conduct all our experiments.

5.1.1 Datasets and Attribute Selections

Datasets. Defining a failed disk is a difficult task in varying deployment scenarios. Based on experience and post-mortem analysis in Tencent Inc., we define a disk that cannot function properly as failed if it was replaced as part for repair. Specifically, it includes three cases: the disk has a write operation error, the system loses connection to the disk and an operation (i.e., disk scrubbing, read and write calls) exceeds the timeout threshold. Note that not all samples of failed disks need to be used in the training set; otherwise, those good samples of failed disks which are far from the actual failure would disturb the training of the detection model. Therefore, only the last 14 continuous samples (our goal is to predict disk failure 14 days in advance) before the moment of failure of the training disks can be regarded as failed samples. We use three SMART datasets from real-world data centers for evaluations. Table 4 gives the overall characteristics of the two datasets. Every disk is classified either as *"Good"* or *"Failed"*. "Sample" indicates SMART records. Each good disk or failed disk has many SMART records. As the original dataset has more samples of good disks than failure disks, we use majority class undersampling to improve training in the case of imbalanced classes to create training datasets. We have chosen a 1:3 ratio of failure disk to good disk [38]. When performing the training in traditional ML method, we divide the data into *70 percent*

TABLE 4
SMART Datasets

Data center	Duration	Good	Good Sample	Failed	Failure Sample
BackBlaze	50 months	141,891	106,867,099	7,657	7,689
Tencent	26 months	68,436	774,994,430	2795	31,574,341

training and 30 percent testing data for our experiments which is in line with existing work [20]. Note that all the results of our experiments are obtained by *cross-validation* [50] in order to avoid fortuitous accident which is common used in ML. Table 5 lists our chosen disks for evaluations.

SMART Attribute Selections. Each SMART observation can contain up to 30 meaningful SMART attributes. However, some attributes are irrelevant to our disk failure predictive model because they are immutable or have not experienced noticeable abnormal changes. Therefore, we selectively keep those attributes that are relevant to the disk health state according to a feature selection process based on Principle Component Analysis (PCA) while ignoring other irrelevant attributes. The selected SMART attributes of HDDs, SATA SSDs and NVMe SSDs are listed in Table 6. For each SMART sample, we use the *Normalized Value* and *Raw Value*. The normalized value typically represents the current value of an attribute. However, certain normalized values lose accuracy when transformed from the raw value and some raw values are more sensitive to the predictive model. We also use other methods of feature selection like [20], [34], [38]. However, the impact on the experimental results is not significant, so we do not discuss further due to limited space.

In addition, different SMART attributes have different output ranges, which will lead to different impacts on the predictive model. In order to make a fair comparison among different SMART attributes in our disk failure predictive model, we normalize the range of all selected SMART attributes using the min-max scaling which is in line with existing work [38]

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}},$$

where x is the original value of a SMART attribute, x_{max} and x_{min} are the maximum and minimum value of the attribute in the training set, respectively.

5.1.2 Evaluation Metrics

Confusion matrix [51] is a tool of visualization for interpreting the performance of machine learning algorithms.

TABLE 5
Selected Disk Models Used in Evaluations

Data center	Manufacturer	Disk model	Good	Bad
Backblaze	Hitachi	HDS722020ALA330	4774	225
		HDS723030ALA640	1048	72
	STX	ST4000DM000	37006	3157
		ST4000DX000	222	81
Tencent	HGST	HGST-A	13367	451
		HGST-B	679	63
	WDC	WDC-A	6847	259
		WDC-B	472	42
Tencent	STX (SATA SSD)	SSD-S-A	8672	397
		SSD-S-B	922	135
	WDC (SATA SSD)	SSD-W-A	13972	489
		SSD-W-B	679	76
	SAMSUNG (NVMe SSD)	NVMe-A	2653	136
		NVMe-B	392	53

TABLE 6
SMART Attributes of HDD, SATA SSD, and NVMe SSD Selected for Our Evaluations

Type	#ID	SMART Attribute Name	Attribute type
HDD	001	Raw Read Error Rate	Normalized&Raw
	003	Spin-Up Time	Normalized
	005	Reallocated Sectors Count	Normalized&Raw
	007	Seek Error Rate	Normalized&Raw
	009	Power-On Hours	Normalized&Raw
	184	I/O Error Detection and Correction	Normalized&Raw
	187	Reported Uncorrectable Errors	Normalized&Raw
	188	Command Timeout	Raw
	189	High Fly Writes	Normalized&Raw
	190	Airflow Temperature	Normalized&Raw
	193	Load/Unload Cycle Count	Normalized&Raw
	194	Temperature	Normalized&Raw
	197	Current Pending Sector Count	Normalized&Raw
	240	Head Flying Hours	Raw
	198	Offline Uncorrectable Sector Count	Normalized&Raw
	241	Total LBAs Written	Raw
	242	Total LBAs Read	Raw
SATA SSD	001	Raw Read Error Rate	Normalized&Raw
	005	Retired Block Count	Normalized
	009	Power On Hours Count	Normalized&Raw
	012	Power Cycle Count	Normalized
	171	Program Fail Count	Normalized&Raw
	172	Erase Fail Count	Normalized&Raw
	174	Unexpected Power Loss Count	Normalized&Raw
	177	Wear-Range Data	Normalized&Raw
	187	Reported Uncorrectable Errors	Normalized&Raw
	188	Command Timeout	Normalized
	195	On the Fly Reported Uncorrectable Error Count	Raw
	196	Reallocated Event Count	Normalized&Raw
	197	Read Failure block Count	Normalized&Raw
	206	Write Error Rate	Normalized
	208	Erase Count Average	Normalized&Raw
NVMe SSD	001	Temperature	Normalized&Raw
	002	Available Spare	Normalized
	003	Available Spare Threshold	Normalized
	004	Percentage Used	Normalized
	005	Controller Busy Time	Normalized&Raw
	006	Power Cycles	Normalized&Raw
	007	Power On Hours	Normalized&Raw
	008	Unsafe Shutdowns	Normalized&Raw
	009	Media and Data Integrity Errors	Normalized&Raw
	010	Error Information Log Entries	Normalized&Raw

Each column in a confusion matrix denotes the classified true class and each row denotes the predictive class. Table 7 shows the confusion matrix used in disk failure prediction. We refer to a failed disk sample as a *Positive* instance (denoted as *"P"*), and a good disk sample as *Negative* instance (denoted as *"N"*). The prediction result takes on only two values: *True* (denoted as *"T"*) or *False* (denoted as *"F"*). Therefore, *"TP"* standing for *"True Positive"*, means that a failed disk is correctly predicted and *"FP"* standing for *"False Positive"*, means that a good disk is falsely predicted as a failed disk. By a similar reasoning, we can get the meanings of both *"FN"* and *"TN"*. Using the statistics information about the four metrics, we can construct many evaluation criteria which are often utilized in judging machine learning algorithms. In our evaluations, we use the following four metrics to report the results in our experiments which are commonly used for evaluating the capability of a classification model in ML [52].

TABLE 7
Confusion Matrix Used in Disk Failure Prediction

	True failed disk	True good disk
Predictive failed disk	*TP*	*FP*
Predictive good disk	*FN*	*TN*

FDR. Failure Detection Rate ($FDR = \frac{TP}{TP+FN}$) also called *recall rate*. It captures the proportion of true failed disks that are correctly predicted as failed. The higher the *FDR* is, the better the model is.

FAR. False Alarm Rate ($FAR = \frac{FP}{FP+TN}$), the proportion of good disks that are falsely predicted as failed. The lower the *FAR* is, the better the model is.

F-Score. F-Score is a balance between *FDR* and *Prediction Precision* ($PP = \frac{TP}{TP+FP}$). *PP* is the proportion of predictive failed disks that are correctly predicted as failed. Therefore, the specific calculation formula of *F-Score* is $\frac{2*FDR*PP}{FDR+PP}$. The higher the *F-Score* is, the better the model is.

AUC-ROC Curve. The *Area Under the Curve-Receiver Operating Characteristic* (*AUC-ROC*) curve is a performance measurement for classification problem at various threshold settings. *ROC* is a probability curve and *AUC* represents degree or measure of separability. It is plotted with *FDR* against the *FAR* where *FDR* is on y-axis and *FAR* is on the x-axis. In disk failure prediction, a higher the *AUC* means the model is better at distinguishing failed and good disks.

5.1.3 Testing Methods and Configurations

To verify the effectiveness of our proposed *TLDFP*, we conduct experiments in three scenarios: 1) to use traditional ML methods only trained on the minority disk datasets, 2) to compare *TLDFP* with traditional ML techniques (*baseline*), and 3) to compare *TLDFP* with other transfer learning approaches. The settings are described below.

1) Traditional ML methods only trained on minority disk:

The detailed description see Section 3.2.

2) TLDFP with traditional ML techniques:

In this scenario, we conduct experiments to investigate the performance of minority disks failure prediction based on four traditional ML algorithms using large heterogeneous datasets for training from both different disk models and the same disk manufacturer. Table 8 shows the training and testing datasets. Note that we randomly choose 10 percent testing datasets for training with all training datasets.

3)TLDFP with other transfer learning approaches:

In addition to traditional ML techniques, we also compare our *TLDFP* with two state-of-the-art transfer learning methods (*SSDB* and *TLBN*) of predicting minority disks failure. Note that we use the same datasets for the two methods in [34] and [20] for fair comparison in all our experiments.

5.2 Experimental Results

In this section, we show the HDD, SATA SSD and NVMe SSD results of the *TLDFP* compared to traditional ML methods and other transfer learning methods with four evaluation metrics mentioned in Section 5.1.2 respectively. *Note that we had showed the poor baseline results of using traditional ML methods only trained on the minority disk datasets in Section 3.2.*

5.2.1 Evaluations Compared to Traditional ML Approaches

- *FDR/Recall Rate:* We conduct experiments to investigate the *FDR* of *TLDFP* and four popular traditional ML methods using four HDD models from two real data centers. As can be seen from the Fig. 6a, none of the four traditional ML methods can deliver a high *FDR* using large heterogeneous datasets. However, the *TLDFP* use the above GBRT, RGF, SVM, RNN algorithms respectively as the basic learners all achieved higher *FDR*.
- *FAR:* Note that the goal of our *TLDFP* is not only to achieve high *FDR* but also low *FAR* for minority disk failure prediction. The results of *FAR* are showed in Fig. 6b. All other methodologies show higher *FAR* which is unacceptable in realistic data centers. Further, none of the four traditional ML methods can deliver both a high *FDR* and a low *FAR* on minority disks except for *TLDFP*. Based on the analysis in Section 3.3, we know the poor predictive performance caused by the traditional ML methods do not have the ability to reduce the distribution difference between the minority disk datasets in target domain and majority disk datasets in source domain.

TABLE 8
Datasets of Minority Disk Failure Prediction Using Large Heterogeneous Dataset Based on Traditional ML

Data Center	Manufacturer	Training Disk Model	Testing Disk Model	Training set	Testing set
Backblaze	*Hitachi*	*HDS-A*	*HDS-B*	4774 good *HDS-A* and 225 failed *HDS-A* 105 good *HDS-B* and 7 failed *HDS-B*	943 good *HDS-B* and 65 failed *HDS-B*
	STX	*STX-A*	*STX-B*	37006 good *STX-A* and 3157 failed *STX-A* 22 good *STX-B* and 8 failed *STX-B*	200 good *STX-B* and 73 failed *STX-B*
Tencent	*HGST*	*HGST-A*	*HGST-B*	13367 good *HGST-A* and 451 failed *HGST-A* 68 good *HGST-B* and 6 failed *HGST-B*	611 good *HGST-B* and 57 failed *HGST-B*
	WDC	*WDC-A*	*WDC-B*	6847 good *WDC-A* and 259 failed *WDC-A* 47 good *WDC-B* and 4 failed *WDC-B*	425 good *WDC-B* and 38 failed *WDC-B*
	STX	*SSD-S-A*	*SSD-S-B*	8672 good *SSD-S-A* and 397 failed *SSD-S-A* 92 good *SSD-S-B* and 14 failed *SSD-S-B*	830 good *SSD-S-B* and 121 failed *SSD-S-B*
	WDC	*SSD-W-A*	*SSD-W-B*	13972 good *SSD-W-A* and 489 failed *SSD-W-A* 68 good *SSD-W-B* and 8 failed *SSD-W-B*	611 good *SSD-W-B* and 68 failed *SSD-W-B*
	SAMSUNG	*NVMe-A*	*NVMe-B*	2653 good *NVMe-A* and 136 failed *NVMe-A* 39 good *NVMe-B* and 5 failed *NVMe-B*	353 good *NVMe-B* and 48 failed *NVMe-B*

TABLE 9
The *FDR*, *FAR*, *F-Score*, and *AUC* of the Comparisons Between *TLDFP* and *SSDB*, *TLBN*

Methods	Manufacturer	*FDR*	*FAR*	*F-Score*	*AUC*
TLDFP (RGF) VS SSDB	*STX*	94.9%/ 85.8%	1.6%/ 3.0%	92.6%/ 83.7%	0.93/ 0.86
	Hitachi	97.1%/ 70.8%	0.9%/ 5.4%	95.7%/ 69.0%	0.96/ 0.81
TLDFP (RGF) VS TLBN	*STX*	91.3%/ 73.1%	0.6%/ 2.6%	91.3%/ 70.4%	0.91/ 0.83

TABLE 10
KLD Values Between Each Disk Model in the Training Sets and the Minority Disk Model in the Testing Set Using Method *TLBN*

Training Disk Model	Testing Disk Model	SMART Attribute	*KLD*
ST320005XXXX	ST33000651AS	5_RAW	5.6
ST32000542AS		190_RAW	2.7
ST1500DL003		188_RAW	7.1
ST31500341AS		190_RAW	1.5
ST31500541AS		197_RAW	0.83

- *F-Score:* Fig. 6c compares the *F-Score* of the different prediction models on the two datasets using four disk models. As can be seen, *TLDFP* has much higher *F-Score* than other different traditional ML methods. As an example, the *F-Score* of *TLDFP* with RBF as its basic learner *TLDFP(RBF)* is almost 9 times as high as the algorithm RBF for *WDC* disks from data center of *Tencent*. As we recall in Section 4.2, the bigger *KLD* value of HGST dataset will lead to more difficult knowledge transfer. This conclusion is also confirmed by the *F-Score* observations given that *the F-Score of TLDFP for HGST are generally lower that other cases*. We will further discuss this issue in detail in Section 6.1.
- *AUC-ROC Curve:* We plot the *AUC-ROC* curve in Fig. 6d using *WDC* disk model in *Tencent*. As it is shown, the *AUC-ROC* curve of *TLDFP* are all close to the top left corner and *TLDFP(RGF)* achieving the higher *AUC* value compared to other two transfer learning methods. The four traditional ML methods achieved lower *AUC* values, reflecting their poor classification ability in performing cross-disk model failure predictions.

5.2.2 Evaluations Compared to Other Transfer Learning Approaches

As it is shown in Table 9, *TLDFP* shows higher *FDR*/*F-Score* and lower *FAR* than *SSDB* and *TLBN*. The reason is that although *SSDB* matches the distribution of the source domain with the target domain, it only ranks the observations in source domain, while *TLDFP* makes more effective weight adjustments to every observation. Considering the *TLBN* is a multi-source domain transfer learning method and *TLDFP* is a single-source domain transfer learning method, we analyze the *KLD* values between each disk model in the source domain and the minority disk model in the target domain. The results are showed in Table 10. We find the data in the disk model with a large *KLD* value in the source domain (such as ST320005XXXX and ST1500DL003). However, *TLDFP* only uses the disk model which has the smallest *KLD* value as source domain. A large *KLD* value leads to difficulties for *TLBN* in mitigating the distribution differences between the source and target domains. This result also shows that single-source domain transfer learning performs better than multi-source domain transfer learning and there is a good metric (e.g, *KLD*) for evaluating differences between different domains.

5.2.3 Evaluations on SATA SSD and NVMe SSD

In addition to the experimental results of HDDs failure prediction, we also conduct experiments on SATA SSD and NVMe SSD in the three scenarios mentioned in Section 5.1.3 and verify the good performance of our *TLDFP*. Table 2 illustrates the poor results of first scenarios (see Section 3.2). As can be seen from the Table 11, whether with traditional ML methods or other transfer learning approaches, our *TLDFP* can achieve higher *FDR*, *F-Score*, *AUC* and lower *FAR* at same time. That shows the effectiveness of our *TLDFP* on different storage media.

In summary, the results have demonstrated that *TLDFP* can effectively solve the problem of minority disk failure prediction with much better predictive performance than traditional ML methods and two other transfer learning methods for the same datasets. More specifically, *TLDFP* not only delivers high *FDR*, *F-Score and AUC*, but also shows a rather low *FAR* at the same time. The main reason for the comparison results is that our *TLDFP* algorithm is able to utilize the small number of labeled target disk data to establish the relationship between source and target disk models, which helps the large heterogeneous disk model to be well trained toward the characteristics of the minority target disk model. In other words, *TLDFP* reduces the difference in data distribution between the source and target domain, which we will further discuss in Section 6.2. *Note that we don't include the results of all methods or disk models due to space limit. From all our tests, TLDFP demonstrates the best performance.*

6 Observations and Sensitivity Study

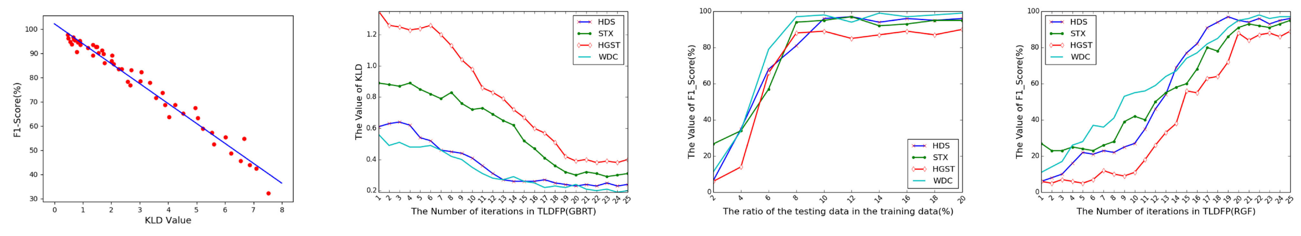
In this section, we provide several additional sensitivity studies from six aspects.

6.1 The Impact of KLD in Source Domain Selection

In order to verify our conjecture in Section 4.2 and further explain the results in Section 5.2.1 and Section 5.2.2, we analyze the relationship between *KLD* and *F-Score* in *TLDFP*

TABLE 11
The Results of Evaluations on SATA SSD and NVMe SSD

Model	Methodology	*FDR(%)*	*FAR(%)*	*F-Score(%)*	*AUC*
SSD-S-B	*TLDFP(GBRT)*	**92.6**	**0.8**	**93.6**	**0.88**
	GBRT/SSDB/ TLBN	18.5/68.9/ 65.2	34.2/12.3/ 13.6	10.6/54.5/ 50.6	0.39/0.71/ 0.68
SSD-W-B	*TLDFP(RGF)*	**93.4**	**1.2**	**91.6**	**0.90**
	RGF/SSDB/ TLBN	28.9/76.3/ 82.9	41.4/13.5/ 11.8	11.7/56.2/ 57.6	0.42/0.73/ 0.75
NVMe-B	*TLDFP(RNN)*	**94.3**	**1.0**	**93.4**	**0.91**
	RNN/SSDB/ TLBN	50.9/66.0/ 77.4	46.4/20.2/ 16.8	20.6/41.9/ 51.2	0.51/0.58/ 0.6



(a) (b) (c) (d)

Fig. 7. (a) The fitting curves of *KLD* and *F-Score*. (b) *KLD* value decrease in training. (c) The relationship between FDR and percentage of target domain data drawn into source domain. (d) The relationship between *FDR* and the number of iterations in the *TLDFP*.

using several minority disk models as testing disk model and large heterogeneous datasets of one disk model for training from the same manufacturer in two real data centers. The results are shown in Fig. 7a and Table 12. We observed that the value of *F-Score* keeps rising as the value of *KLD* decreases. In other words, it shows that the smaller the difference in SMART data distribution between the source and target domain, the easier knowledge transfer in *TLDFP* can be. Therefore, we use *KLD* as an effective indicator for source domain selection of large heterogeneous dataset and usually select the one which has the smallest *KLD* value.

6.2 The Change of KLD Value in TLDFP

In order to more intuitively observe how the *TLDFP* method reduces the difference in data distribution between the source domain and the target domain, we record the value of *KLD* between the training set and the dataset after each iteration of the model during the training process of *TLDFP(GBRT)*, as illustrated in Fig. 7b. We can see that as the number of iterations of the model increases, the *KLD* value between the source domain and the target domain decreases continuously, and stabilizes at a smaller value when the number of iterations is about 22. This shows that our *TLDFP* model continuously reduces the difference of the SMART data distribution between the two domains in the training process, enabling us to use the large heterogeneous disk data to predict the minority disk data and realize knowledge transfer.

6.3 Varying Samples from Target Domain

As discussed previously our prediction model *TLDFP* uses a portion of labeled dataset from target domain as part of its source domain dataset. In this section, we investigate how the percentage number affects predictive performance of *TLDFP*. We report the results of *TLDFP* with RGF as its basic learner and using the disk data of *WDC* model from *Tencent* data center, as the other three basic learners show similar results. Fig. 7c shows the relationship between the *FDR* and the percentage of target domain data put in the source domain. As it clearly shows, the *FDR* increases dramatically when the percentage increases from 2 to 10 percent. When the percentage goes beyond 10 percent, the *FDR* does not continue to increase but remains a relatively high level, meaning putting more target domain data to the source domain does not help further improving predictive performance. Consequently, we randomly choose 10 percent target domain data in our previous experiments.

TABLE 12
The *F-Score* Varies With the Value of *KLD*

Data Center	Method	Training Model	Testing Model	*KLD*	*F-Score*
BackBlaze	TLDFP (GBRT)	*HGST-K*	*HGST-L*	2.67	76.8%
			HGST-M	1.76↓	85.9%↑
			HGST-N	0.91↓	93.5%↑
			HGST-O	0.69↓	95.7%↑
			HGST-P	0.47↓	97.6%↑
Tencent	TLDFP (SVM)	*STX-K*	*STX-L*	3.57	71.6%
			STX-M	2.58↓	78.2%↑
			STX-N	2.26↓	83.4%↑
			STX-O	1.35↓	93.1%↑
			STX-P	1.17↓	92.2%↓
			STX-Q	0.71↓	95.3%↑
			STX-R	0.66↓	96.6%↑

6.4 The Impact of Iterations

The *TrAdaBoost* algorithm takes an input parameter to cap the iterations performed by the algorithm. In this section, we study how the iteration parameter affects predictive performance in terms of *F-Score*. We report the results of *TLDFP* with RGF as its basic learner. Fig. 7d shows how *F-Score* changes as the number of iterations varies for different datasets. From this figure, we can make similar observations as in the preceding subsection. As the number of iterations increases, the *F-Score* increases quickly and reaches a stable level at 22 iterations (fast convergence). It also shows that as the algorithm adjusts instance weights in each iteration, *TLDFP* gradually adjusts to converge toward the testing data. Since the number of 22 represents a inflection point, we adopt this iteration number in our previous experiments. Note that the number of iterations is not fixed according to the different datasets and model parameters. In general, the model would be stopped for training after the model loss bottoms.

6.5 The Sensitivity Study of IMDA

In Section 4.1, we introduced the algorithm *KLD*. Specifically, if any instance is classified as failure, the corresponding disk is considered as failure. Here we conduct experiments to evaluate the impact of different instances, specifically, 1 instance, 1/3 of all instances, 1/2 of all instances, 2/3 of all instances, and all instances being classified as failure.

The result of this experiment using *TLDFP(RNN)* based on the *WDC* disk model data in *Tencent* is showed in Table 13. It is clear that the option we use (one instance failure indicates the corresponding disk failure) achieves the

TABLE 13
The Sensitivity Study Results of *IMDA* in *TLDFP*

Methods	Metrics	1	1/3	1/2	2/3	All
TLDFP	*FDR*	95%	32%	24%	8%	3%
	FAR	0.7%	0.7%	0.5%	0.2%	0.2%

best performance. In practice, we perform a prediction for each minority disk once a day. If any instance of the disk is predicted as a failure, the corresponding disk will be considered as failure.

6.6 Cost Benefits

As we have demonstrated so far that *TLDFP* achieves both high *FDR* and low *FAR*. It is easily understandable that improving *FDR*, i.e., disk failures are correctly predicted, can reduce the probability of data loss occurring. In this section, we discuss the benefits brought by low *FAR* in ML methods. A *FAR* will trigger responsive procedures to be launched, e.g., data migration, disk replacement, etc., which incurs extra cost to the IT management. Assume that disk failures are independent events with a probability of ϵ_d and the number of disks in a data center is N. A disk will fail after a certain period of time, which is called disk lifespan. For instance, the *Tencent*'s data centers assume the lifespan T to be 4-5 years. The disk failure rate over a period of time is dependent on the time T and ϵ_d. Therefore, we use $f(T, \epsilon_d)$ to denote the failure rate of disks in a data center that are failed over a period of T. The total number of failed disks in the period T can be expressed as N*$f(T, \epsilon_d)$ and the total number of failed disks per unit time can be described as:

$$\frac{N * f(T, \epsilon_d)}{T}.$$

The total number of good disks is thus given by

$$N * (1 - \frac{f(T, \epsilon_d)}{T}).$$

Given a *FAR*, the number of all non-faulty disks which are wrongly predicted to be failed can be described as

$$FAR * N * (1 - \frac{f(T, \epsilon_d)}{T}).$$

Assume $c(\$)$ is the statistic average cost of replacing one disk (including many disks out of the warranty period), which contains the cost of manual replacement c_1 and the cost of a new disk c_2 as well as the cost c_3 caused by copying the data from the failed disk to a new disk. The total additional cost is

$$Cost = c * FAR * N * (1 - \frac{f(T, \epsilon_d)}{T}).$$

Therefore, if *FAR* = 0, the cost will be 0. On the other hand, if *FAR* = 1, meaning all good disks are wrongly predicted to be failed and replaced, the cost would be maximum. Last but not the least, the total additional cost caused to the data center is proportional to the *FAR* of the prediction model. The larger *FAR* is, the more additional cost will be. The traditional machine learning methods and other transfer learning methods, lead to an average of 60 and 4 percent *FAR*, when they are used to predict disk failures for minority disks, while our *TLDFP* achieves an average of 0.5 percent *FAR*, which can be translated to 120X and 8X reduction in additional cost. According to an average cost was estimated as c = \$426 in *Tencent* data centers in 2018, the *Tencent* company can save about \$105 to \$1,800 milions per year. In addition, we get a higher *FDR* with lower *FAR* which means *TLDFP* predicts true failed disks as a good disk rarely, reducing the inestimable cost due to disk failures.

7 Conclusion

In this paper, we develop a model called *TLDFP* to effectively predict minority disk failure leveraging the transfer learning based on different storage media, where traditional ML approaches perform poorly. Our main contributions include: (1) we are the first to define minority disk datasets and quantitatively evaluate them through extensive data analysis and experiments in data centers of heterogeneous disk models, (2) we are the first to present a novel method based on *KLD* values to select proper majority disk models, and (3) we develop a method of making crossing-disk model (HDD, SATA SSD and NVMe SSD) failure prediction, which has important practical applicability as different disk models are gradually placed into the realistic storage systems to replace failed disks. Our experiments with three datasets from real-world data centers have shown that *TLDFP* well outperforms representative traditional ML methods and existing transfer learning approaches in terms of failure detection rate and false alarm rate. *TLDFP* achieves on average 96 percent failure detection rate with only 0.5 percent false alarm rate in performing crossing-disk models failure prediction and reduces inestimable cost to data centers.

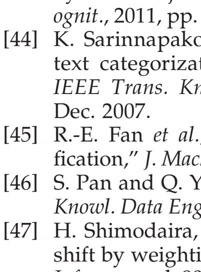
Acknowledgments

This work was supported by the Innovation Group Project of the National Natural Science Foundation of China No.61821003, the National Key Research and Development Program of China under Grant No.2016YFB0800402 and the U.S. National Science Foundation Grants CCF-1717660 and CCF-1813081. A preliminary version [1] appeared in the proceedings of the 48th International Conference on Parallel Processing (ICPP), 2019.

References

[1] Z. Ji *et al.*, "Transfer learning based failure prediction for minority disks in large data centers of heterogeneous disk systems," in *Proc. 48th Int. Conf. Parallel Process.*, 2019, pp. 1–10.
[2] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an MTTF of 1, 000, 000 hours mean to you?" in *Proc. 5th USENIX Conf. File Storage Technol.*, 2007, pp. 1–16.
[3] E. Pinheiro *et al.*, "Failure trends in a large disk drive population," in *Proc. 5th USENIX Conf. File Storage Technol.*, 2007, pp. 17–28.
[4] L. N. Bairavasundaram *et al.*, "An analysis of latent sector errors in disk drives," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst.*, 2007, pp. 289–300.
[5] J. Meza *et al.*, "A large-scale study of flash memory failures in the field," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst.*, 2015, pp. 177–190.

[6] B. Schroeder *et al.*, "Flash reliability in production: The expected and the unexpected," in *Proc. 14th USENIX Conf. File Storage Technol.*, 2016, pp. 67–80.
[7] B. Calder *et al.*, "Windows azure storage: A highly available cloud storage service with strong consistency," in *Proc. 23rd ACM Symp. Operating Syst. Princ.*, 2011, pp. 143–157.
[8] C. Huang *et al.*, "Erasure coding in windows azure storage," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2012, pp. 15–26.
[9] S. Huang and S. Fu, Q. Zhang, and W. Shi, "Characterizing disk failures with quantified disk degradation signatures: An early experience," in *Proc. IEEE Int. Symp. Workload Characterization*, 2015, pp. 150–159.
[10] Y. Xu *et al.*, "Improving service availability of cloud systems by predicting disk error," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2018, pp. 481–494.
[11] H. S. Gunawi *et al.*, "Fail-slow at scale: Evidence of hardware performance faults in large production systems," *ACM Trans. Storage*, vol. 14, no. 3, pp. 23:1–23:26, 2018.
[12] D. A. Patterson *et al.*, "A case for redundant arrays of inexpensive disks (RAID)," in *Proc. ACM Int. Conf. Manage. Data*, 1988, pp. 109–116.
[13] B. Allen, "Monitoring hard disks with SMART," *Linux J.*, vol. 117, pp. 60–65, 2004.
[14] J. F. Murray *et al.*, "Machine learning methods for predicting failures in hard drives: A multiple-instance application," *J. Mach. Learn. Res.*, vol. 6, pp. 783–816, 2005.
[15] B. Zhu, G. Wang, X. Liu, D. Hu, S. Lin, and J. Ma, "Proactive drive failure prediction for large scale storage systems," in *Proc. IEEE 29th Symp. Mass Storage Syst. Technol.*, 2013, pp. 1–5.
[16] W. Yang, D. Hu, Y. Liu, S. Wang, and T. Jiang, "Hard drive failure prediction using big data," in *Proc. IEEE 34th Symp. Reliable Distrib. Syst. Workshop*, 2015, pp. 13–18.
[17] T. Pitakrat *et al.*, "A comparison of machine learning algorithms for proactive hard disk drive failure detection," in *Proc. 4th Int. ACM Sigsoft Symp. Architecting Critical Syst.*, 2013, pp. 17–21.
[18] J. Zhang *et al.*, "Tier-scrubbing: An adaptive and tiered disk scrubbing scheme with improved MTTD and reduced cost," in *Proc. 57th ACM/EDAC/IEEE Des. Autom. Conf.*, 2020, pp. 1–6.
[19] W. Jiang *et al.*, "Are disks the dominant contributor for storage failures—A comprehensive study of storage subsystem failure characteristics," *ACM Trans. Storage*, vol. 4, no. 3, pp. 7:1–7:25, 2008.
[20] F. L. F. Pereira, F. D. dos Santos Lima, L. G. de Moura Leite, J. P. P. Gomes, and J. de Castro Machado, "Transfer learning for Bayesian networks with application on hard disk drives failure prediction," in *Proc. Brazilian Conf. Intell. Syst.*, 2017, pp. 228–233.
[21] J. Zhang *et al.*, "An end-to-end automatic cloud database tuning system using deep reinforcement learning," in *Proc. Int. Conf. Manage. Data*, 2019, pp. 415–432.
[22] I. V. Tetko *et al.*, "Neural network studies, 1. Comparison of overfitting and overtraining," *J. Chem. Inf. Comput. Sci.*, vol. 35, no. 5, pp. 826–833, 1995.
[23] G. Hamerly and C. Elkan, "Bayesian approaches to failure frediction for disk drives," in *Proc. 18th Int. Conf. Mach. Learn.*, 2001, pp. 202–209.
[24] G. F. Hughes, J. F. Murray, K. Kreutz-Delgado, and C. Elkan, "Improved disk-drive failure warnings," *IEEE Trans. Rel.*, vol. 51, no. 3, pp. 350–357, Sep. 2002.
[25] J. F. Murray *et al.*, "Hard drive failure prediction using nonparametric statistical methods," in *Proc. Int. Conf. Artif. Neural Netw.*, 2003, pp. 1–4.
[26] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, 1995.
[27] F. Salfner *et al.*, "A survey of online failure prediction methods," *ACM Comput. Surv.*, vol. 42, no. 3, pp. 10:1–10:42, 2010.
[28] S. E. A. Ganguly, "A practical approach to hard disk failure prediction in cloud platforms: Big data model for failure management in datacenters," in *Proc. IEEE 2nd Int. Conf. Big Data Comput. Service Appl.*, 2016, pp. 105–116.
[29] J. Li *et al.*, "Hard drive failure prediction using classification and regression trees," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2014, pp. 383–394.
[30] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *Ann. Statist.*, vol. 29, no. 5, pp. 1189–1232, 2001.
[31] J. Li, R. J. Stones, G. Wang, Z. Li, X. Liu, and K. Xiao, "Being accurate is not enough: New metrics for disk failure prediction," in *Proc. 35th IEEE Symp. Reliable Distrib. Syst.*, 2016, pp. 71–80.
[32] J. Li *et al.*, "Hard drive failure prediction using decision trees," *Rel. Eng. Sys. Saf.*, vol. 164, pp. 55–65, 2017.
[33] R. Johnson and T. Zhang, "Learning nonlinear functions using regularized greedy forest," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 36, no. 5, pp. 942–954, Mar. 2014.
[34] M. M. Botezatu *et al.*, "Predicting disk replacement towards reliable data centers," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2016, pp. 39–48.
[35] C. Xu, G. Wang, X. Liu, D. Guo, and T.-Y. Liu, "Health status assessment and failure prediction for hard drives with recurrent neural networks," *IEEE Trans. Comput.*, vol. 65, no. 11, pp. 3502–3508, Nov. 2016.
[36] T. Mikolov, S. Kombrink, L. Burget, J. Černocký, and S. Khudanpur, "Extensions of recurrent neural network language model," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2011, pp. 5528–5531.
[37] T. Mikolov, J. Kopecky, L. Burget, O. Glembek, and J. Černocký, "Neural network based language models for highly inflective languages," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2009, pp. 4725–4728.
[38] F. Mahdisoltani *et al.*, "Improving storage system reliability with proactive error prediction," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2017, pp. 391–402.
[39] J. T. Zhou *et al.*, "Hybrid heterogeneous transfer learning through deep learning," in *Proc. 28th AAAI Conf. Artif. Intell.*, 2014, pp. 2213–2220.
[40] P. Prettenhofer and B. Stein, "Cross-language text classification using structural correspondence learning," in *Proc. 48th Annu. Meeting Assoc. Comput. Linguistics*, 2010, pp. 1118–1127.
[41] B. Kulis, K. Saenko, and T. Darrell, "What you saw is not what you get: Domain adaptation using asymmetric kernel transforms," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2011, pp. 1785–1792.
[42] M. Harel and S. Mannor, "Learning from multiple outlooks," in *Proc. 28th Int. Conf. Mach. Learn.*, 2011, pp. 401–408.
[43] H. Wang, A. Kläser, C. Schmid, and C.-L. Liu, "Action recognition by dense trajectories," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2011, pp. 3169–3176.
[44] K. Sarinnapakorn and M. Kubat, "Combining subclassifiers in text categorization: A DST-based solution and a case study," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 12, pp. 1638–1651, Dec. 2007.
[45] R.-E. Fan *et al.*, "LIBLINEAR: A library for large linear classification," *J. Mach. Learn. Res.*, vol. 9, pp. 1871–1874, 2008.
[46] S. Pan and Q. Yang, "A survey on transfer learning," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 10, pp. 1345–1359, Oct. 2010.
[47] H. Shimodaira, "Improving predictive inference under covariate shift by weighting the log-likelihood function," *J. Statist. Planning Inference*, vol. 90, no. 2, pp. 227–244, 2000.
[48] W. Dai *et al.*, "Boosting for transfer learning," in *Proc. 24th Int. Conf. Mach. Learn.*, 2009, pp. 193–200.
[49] S. Kullback and R. A. Leibler, "On information and sufficiency," *The Ann. Math. Statist.*, vol. 22, pp. 79–86, 1951.
[50] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proc. 14th Int. Joint Conf. Artif. Intell.*, 1995, pp. 1137–1143.
[51] S. V. Stehman, "Selecting and interpreting measures of thematic classification accuracy," *Remote Sens. Environ.*, vol. 62, no. 1, pp. 77–89, 1997.
[52] D. Powers, "Evaluation: From precision, recall and F-measure to ROC, informedness, markedness and correlation," *J. Mach. Learn. Technol.*, vol. 2, pp. 37–63, Jan. 2007.



Ji Zhang is currently working toward the PhD degree in computer science and technology from the Huazhong University of Science and Technology (*HUST*), Wuhan, China and currently a visiting scholar with the Center for Data Science of New York University, New York. His major is computer system structure. He has been an internship at the Intelligent Cloud Storage Joint Research center of *HUST* and *Tencent*. Currently, his main research interests are using artificial intelligence (AI) technologies to optimize the system of data storage or data management. He has published papers in international conferences and journals including SIGMOD, ICPP, DAC, FAST, TPDS, and NEDB, etc.

Ke Zhou (Member, IEEE) received the BE, ME, and PhD degrees in computer science and technology from the Huazhong University of Science and Technology (*HUST*), China, in 1996, 1999, and 2003, respectively. He is currently a professor of the School of Computer Science and Technology and Wuhan National Laboratory for Optoelectronics, *HUST*. His main research interests include computer architecture, cloud storage, parallel I/O, and storage security. He has more than 50 publications in journals and international conferences, including ACM SIGMOD, ICCP, the *IEEE Transactions on Parallel and Distributed Systems*, the Performance Evaluation (PEVA), FAST, USENIX ATC, MSST, ACM MM, INFOCOM, SYSTOR, MASCOTS, ICCD, etc. He is a member of USENIX.

Ping Huang received the PhD degree from the Huazhong University of Science and Technology, China, in 2013. He is currently a research assistant with the Department of Computer and Information Sciences, Temple University, Philadelphia, Pennsylvania. His main research interest includes non-volatile memory, operating system, distributed systems, DRAM, GPU, Key-value systems, etc. He has published papers in various international conferences and journals, including SYSTOR, NAS, MSST, USENIX ATC, Eurosys, IFIP Performance, INFOCOM, SRDS, MASCOTS, ICCD, the *Journal of Systems Architecture (JSA)*, the Performance Evaluation (PEVA), the Sigmetrics, ICPP, the *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, the *ACM Transactions on Storage*, etc.

Xubin He (Senior Member, IEEE) received the BS and MS degrees in computer science from the Huazhong University of Science and Technology, China, in 1995 and 1997, respectively, and the PhD degree in electrical engineering from the University of Rhode Island, Kingston, Rhode Island, in 2002. He is currently a professor with the Department of Computer and Information Sciences, Temple University, Philadelphia, Pennsylvania. His research interests include computer architecture, data storage systems, virtualization, and high availability computing. He received the Ralph E. Powe Junior Faculty Enhancement Award, in 2004 and the Sigma Xi Research Award (TTU Chapter) in 2005 and 2010. He is a member of the IEEE Computer Society and USENIX.

Ming Xie is currently a general manager in Cloud Architecture Platform Department at *Tencent* Corporation. His main research includes cloud computing, massive data storage, and intelligent operation and maintenance.

Bin Cheng is currently a director in Cloud Architecture Platform Department at *Tencent* Corporation. His main research includes cloud computing, massive data storage, machine learning and database system, etc.

Yongguang Ji is currently a group leader in Cloud Architecture Platform Department at *Tencent* Corporation. His main research includes cloud computing, cloud block storage, I/O systems and performance optimization, etc.

Yinhu Wang is currently a senior staff engineer in Cloud Architecture Platform Department at *Tencent* Corporation. His main research includes distributed systems, massive data storage system, disk failure prediction, etc.

The Workflow Trace Archive: Open-Access Data From Public and Private Computing Infrastructures

Laurens Versluis, Roland Mathá, Sacheendra Talluri, Tim Hegeman, Radu Prodan, Ewa Deelman, and Alexandru Iosup

Abstract—Realistic, relevant, and reproducible experiments often need input traces collected from real-world environments. In this work, we focus on traces of workflows—common in datacenters, clouds, and HPC infrastructures. We show that the state-of-the-art in using workflow-traces raises important issues: (1) the use of realistic traces is infrequent and (2) the use of realistic, *open-access* traces even more so. Alleviating these issues, we introduce the Workflow Trace Archive (WTA), an open-access archive of workflow traces from diverse computing infrastructures and tooling to parse, validate, and analyze traces. The WTA includes >48 million workflows captured from >10 computing infrastructures, representing a broad diversity of trace domains and characteristics. To emphasize the importance of trace diversity, we characterize the WTA contents and analyze in simulation the impact of trace diversity on experiment results. Our results indicate significant differences in characteristics, properties, and workflow structures between workload sources, domains, and fields.

Index Terms—Workflow, open-source, open-access, traces, characterization, archive, survey, simulation

1 Introduction

Workflows are already a significant part of private datacenter and public cloud infrastructures [1], [2]. This trend is likely to intensify [3], [4], as organizations and companies transition from basic to increasingly more sophisticated cloud-based services. For example, 96 percent of companies responding to RightScale's 2018 survey are using the cloud [5], up from 86 percent in 2012 [6]; the average organization combines services across five *public and private* clouds. To maintain, tune, and develop the *computing infrastructures* for running workflows at the massive scale and with the diversity suggested by these trends, the systems community requires adequate capabilities for testing and experimentation. Although the community is aware that workload traces enable a broad class of realistic, relevant, and reproducible experiments, currently such traces are infrequently used, as we summarize in Fig. 1 (left) and quantify in Section 2. Toward addressing this problem, we focus on improving trace availability and understanding by proposing a new, free and open-access *Workflow Trace Archive (WTA)*, as detailed in Fig. 1 (right) and in the remainder of this work.

• L. Versluis, T. Hegeman, S. Talluri, and A. Iosup are with Computer Science, Vrije Universiteit Amsterdam, 1081 HV Amsterdam, Netherlands. E-mail: {l.f.d.versluis, A.Iosup}@vu.nl, {sacheendra.t, tim.m.hegeman}@gmail.com.
• R. Mathá is with the Institute of Computer Science, Universitat Innsbruck, 6020 Innsbruck, Tyrol, Austria. E-mail: roland@dps.uibk.ac.at.
• R. Prodan is with the Institute of Software Technology, University of Klagenfurt, 9020 Klagenfurt am, Austria. E-mail: radu@itec.aau.at.
• E. Deelman is with Information Sciences Institute, University of Southern California, Los Angeles, CA 90292. E-mail: deelman@isi.edu.

Manuscript received 8 July 2019; revised 5 Mar. 2020; accepted 22 Mar. 2020. Date of publication 14 Apr. 2020; date of current version 1 May 2020. (Corresponding author: Laurens Versluis.) Recommended for acceptance by T. Kosar. Digital Object Identifier no. 10.1109/TPDS.2020.2984821

The need for workflow traces is stringent [4], [7]. In this work, we adopt the workflow model of Coffman and Graham [8]. In this model, a workflow is considered a directed acyclic graph (DAG) where each vertex represents a task and an edge a computation/data constraint. As such, we do not consider workflow formalisms with iteration (loops) and human interaction, such as BPMN/BPEL [9] and Petri nets [10]. We consider as tasks a broad range of activities, that is, black boxes ranging from simple compute and data operations to entire workflows, recursively.

A workflow trace is a recording of useful, relevant information during the processing of the workflow. Traces can be used to create models with, or used in emulations and simulations to replay the execution of a workflow in a controlled environment, etc. Not only the sheer volume of workloads has increased significantly over time [2], but also the users of datacenters and cloud operations are expecting increasingly better Quality of Service (QoS) from the workflow-management systems, including elasticity, reliability, and low-cost, under strong assumptions of validation [4], [7] and reproducibility [3], [11]. Developing workflow management systems to meet these requirements requires considerable scientific and technical advances and, correspondingly, comprehensive trace-based experimentation and testing. This can be conducted (i) *in vivo*, i.e., experimenting in live/production settings, (ii) *in vitro*, i.e., experimenting using emulation, and (iii) *in silico* i.e., experimenting in simulation [12].

Testing such systems, especially at cluster and datacenter scale, often cannot be done in vivo, due to

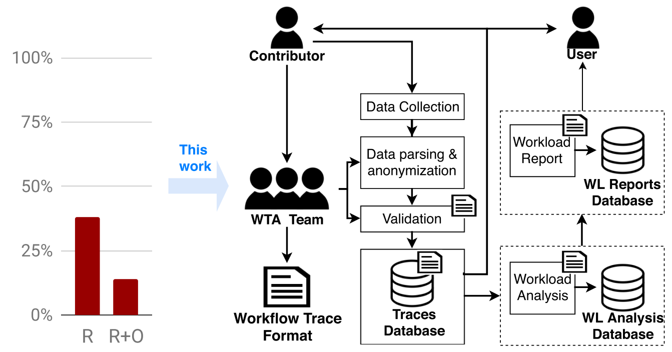


Fig. 1. A visual map to this work: (*left*) The problem: infrequent use of Realistic ($\approx 40\%$) and Open-source ($\approx 15\%$) workflow-traces in representative articles (see Section 2), which can affect the relevance and reproducibility of experiments for the entire community. (*right*) Toward an answer: the WTA stakeholders, process, and tools provide the community with open-source traces of relevant workflows running in public and private computing infrastructures.

downtime or the operational costs required. Instead, workflow traces can be replayed in silico, allowing multiple setups to run in parallel, testing individual components, etc. without the downtime nor costs. Although *realistic workflow traces* are key for testing, tuning, validating, and inspiring system designs, they are currently still scarce [13]. Prior work, such as WorkflowHub [14], has introduced numerous workflow traces, yet only from the science domain. As Fig. 1 (left) indicates, and Section 2 quantifies and explains, less than 40 percent of relevant articles focusing on workflow systems conduct experiments with *realistic* traces, and less than 15 percent conduct experiments with realistic and *open-source* traces.

The current scarcity of traces forces researchers to either use synthetically generated workloads or to use one of the few available traces. Synthetic traces may reduce the representatives and quality of experiments, if they do not match relevant real-world settings. Using realistic traces that correspond to a narrow application-domain may result in overfitting; Amvrosiadis *et al.* [15] demonstrate this for cluster-based infrastructures. Additionally, a lack of realistic traces may lead to limited or even wrong understanding of workflow characteristics, their performance, and their usage, which hampers the reuse of the systems tested with such (workloads of) workflows [16]. This gives rise to the research question *RQ-1: How diverse are the workflow traces currently used by the systems community?*

We identify the need to share workflow traces collected from relevant environments running relevant workloads under relevant constraints. Effective sharing requires unified trace formats, and also support for emerging and new features. For example, since the introduction of commercial clouds, clients have increasingly started to ask for better QoS, and in particular have started to increasingly express non-functional requirement (NFRs) such as availability, privacy, and security demands in traces [4], [17]. This leads us to research question *RQ-2: How to support sharing workflow traces in a common, unified format? How to support in it arbitrary NFRs?*

Persuading both academia and industry to release data is vital to address the problems stated prior. We tackle this issue with two main approaches. First, by offering tools to obscure sensitive information, while still retaining significant detail in shared traces. Second, by encouraging the same organization to share the data across its possibly multiple workflow management systems (*sources*), and by explicitly aiming to collect data across diverse application *domains* and *fields*. The availability of diverse data and tools stimulate the benefits of making available such traces, while simultaneously reducing the concerns of competitive disadvantage or of an (accidental) disclosure of sensitive information. The community is already helping with both approaches, by increasingly focusing on the problem of reproducibility. For example, ACM introduced artifact review and badges to stimulate the release of both software and data artifacts for reproducibility and verification purposes [18]. We add to this community-effort ours, which is scientific in nature: *RQ-3: What is the impact of the source and domain of a trace on the characteristics of workflows?*

Addressing research questions 1–3, our contribution is four-fold:

1) To answer RQ-1, we conduct the first comprehensive survey of how the systems community uses workflow traces (Section 2). We collect, select, and label articles from top conferences and journals covering workflow management. We analyze the types of traces used in the community, and the domains and fields covered in published studies. To improve reproducibility and promote extensions, we make public all (raw) data used for this survey.
2) To answer RQ-2, we design the WTA for open-access to traces of *workloads* of workflows (Section 3). We identify a comprehensive set of requirements for a workflow trace archive. A key conceptual contribution of the WTA is the design of a unified trace format for sharing workflows, the first to generalize NFRs support at both workflow- and task-levels. The WTA currently archives a diverse set of (1) real workflow traces collected from real-world environments, (2) realistic workflow traces used in peer-reviewed publications, and (3) workflow traces collected from simulated and emulated environments commonly used by the systems community. WTA also introduces tools to detail and compare its traces.
3) To address RQ-3, we compare key workload characteristics across traces, domains, and sources (Section 4). Our effort is the first to characterize the new trace from Alibaba, and the first to investigate the critical path task length, level of parallelism, and burstiness using the Hurst exponent on workloads of workflows. Overall, the archive comprises 96 traces, featuring more than 48 million workflows containing over 2 billion CPU core hours.
4) To validate our answers to RQs 1–3, we analyze various threats (Section 5). We conduct a trace-based simulation study and qualitative analysis. Our results for the former indicate systems should be tested with different traces to validate claims about the generality of the proposed approach.

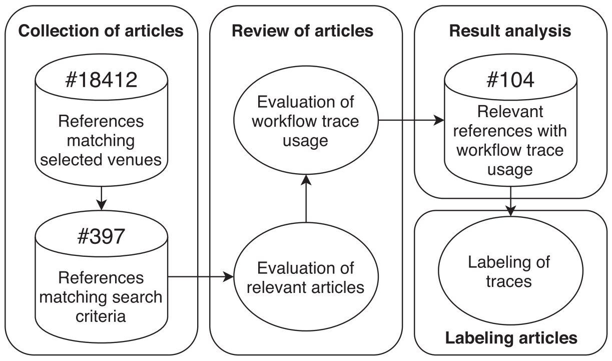

Collection of articles
#18412
References matching selected venues
#397
References matching search criteria
Review of articles
Evaluation of workflow trace usage
Evaluation of relevant articles
Result analysis
#104
Relevant references with workflow trace usage
Labeling of traces
Labeling articles

Fig. 2. The article selection process. Subsequent stages decrease the amount of articles: from a corpus of 18 412 articles, down to 104 relevant references.

2 A Survey of Workflow Trace Usage

To assess the current usage of workflow traces in the systems community and the need for a workflow archive, we systematically survey a large body of work published in top conferences and journals, and investigate articles that perform experiments using workflow traces, either through simulation or using a real-world setup. The process and outcome of this survey answer RQ-1.

2.1 Article Selection and Labeling

Selection. Fig. 2 displays our systematic approach to select articles relevant to this survey, based on [19]. First, we collect data from DBLP [20] and Semantic Scholar [21]. We filter them by venue, retaining only articles from the 10 key conferences and journals in distributed systems listed in the caption of Table 1, including TPDS. While not an exhaustive list, this covers a significant part of the systems community. This yields 18,412 articles. Next, we automatically select all articles from the last decade (2009–2018) containing the word "workflow" in either title or abstract, yielding 397 articles. This step provides articles that focus on all aspects of workflows, e.g., scheduling, analysis, and design. Finally, to obtain insights into workflow traces usage, we manually check the 397 articles. Overall, this systematic process yields 104 articles using workflow traces. To highlight the relevance of papers, we use Google Scholar to obtain citation counts. In total the 104 papers have been cited 3,965 times.

Labeling. We label for each of the 104 articles the type of trace usage. For articles explicitly describing their use, we use the label *realistic* for traces *collected* from real-world workflow executions. For all others, including workflows *extrapolated* from real-world data or *generated* from known statistical distributions, we use the label *synthetic*.

We further label traces as *open-access* (or open-source) if they are available online and to a broad audience, and *closed-access* (or closed-sources) otherwise. In our analysis, we include among the open-access traces only those that are also realistic.

We also label traces by *domain* and *field*. *Domains* are corresponding to the area of study of which the trace originates from. We label sub-domains within these domains as *fields*. We adopt the domains and fields reported by the respective authors, where mentioned. If the domain or field are not mentioned, yet the application appears in another article by name and with labels, we remain consistent in our labeling by adopting the domain/field from this prior article. We have not encountered cases where an application is labelled as belonging to multiple domains or fields. We identify in articles explicit use of traces from the domains "scientific", "engineering", "multimedia", "governmental", and "industry", and from fields such as "bioinformatics", "astronomy", "physics", etc. We further label a trace with *uncategorized* when its origin remains unexplained.

All data used in this survey is available as open-access data[1] and can be used to verify and extend this survey.

2.2 Types of Traces Used in the Community

We analyze here the types of traces used by the community, with the following Observations (Os):

O-1: Less than 40 percent of articles use realistic traces.
O-2: Only one-seventh of all articles use open-access traces.

Table 1 presents the types of traces used in the community, focusing on realistic (R) and open-access (R+O) traces. The community uses traces for experiments across both conference and journal articles, across various levels of (high) quality. In contrast to this positive finding, the results indicate that, from the total number of articles using traces at all, the fraction of articles using realistic and even open-access traces is relatively small. Across all venues, only 38 percent of the articles use at least one realistic trace, and only 13 percent of the articles use at least one open-access trace.

These findings match the perceived difficulty in reproducing studies in the field [11], [12], and may hint why so few of these seemingly successful designs are adopted for use in practice [22].

2.3 Workflow Domains and Fields

We analyze the domains and fields from which the community sources workflows, with as main observations:

O-3: The community sources workflows from 5+ domains and 25+ fields.
O-4: Traces containing scientific workflows are used significantly more (20x) than workflows from other domains, e.g., industry and engineering, in the surveyed articles.
O-5: Bioinformatics workflows are the most commonly used, but three other fields exhibit usage within a factor of 3.
O-6: Many traces have uncategorized domain (14 percent) and/or field (31 percent).

Overall, we find that the community uses diverse workflows, sourced from 5+ domains and 25+ fields.

We further investigate the distribution of use, per domain and per field. Fig. 3 (top) shows that the scientific domain is over-represented in the literature in the top-five trace domains encountered, due to the large number of available open-access traces and from their conventional use in prior work. In particular, a large portion of the articles use workflow traces from the Pegasus project, which covers the scientific domain. The number of traces in this domain exceeds 200, which is larger than the number of

1. https://github.com/atlarge-research/wta-analysis

TABLE 1
Workflow Trace Usage in Venues Having at Least One Paper Returned in the Initial Query

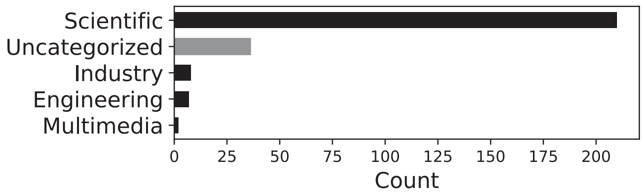
	Acronym	Total	FGCS	CCGrid	*TPDS*	Other
`T`	Articles using traces	104	37	17	17	33
`R`	Articles using *realistic* traces	40 (38%)	13 (35%)	8 (47%)	6 (35%)	13 (39%)
`R+O`	Articles using traces that are both *realistic* and *open-access*	14 (13%)	6 (16%)	2 (12%)	3 (18%)	3 (9%)

The venues with > 5 hits have their individual column. The column "Other" shows combined results for conferences with ≤ 5 hits: ATC, CLOUD, CLUSTER, e-Science, Euro-Par, GRID, HPDC, JSSPP, IC2E, ICDCS, ICPE, IPDPS, NSDI, OSDI, SC, SIGMETRICS, and WORKS. Percentages are computed from the total in the corresponding column, e.g., 13 out of 37 for the cell corresponding to row `R` and column `FGCS`.

articles in the study as each article uses multiple traces. In contrast, the next-largest domains are industry and engineering, each with less than 10 traces representing less than one-twentieth of the scientific domain.

We remark the positive diversity of the workflow domains, considering that the entire community is tempered by the extreme focus on scientific workflows. This confirms the bias demonstrated by Amvrosiadis *et al.* [23] with the popular Google-cluster traces. A similar situation appears for fields, but more tempered, as Fig. 3 (bottom) indicates. A large portion of the traces have their domains and fields as "uncategorized" (14 and 31 percent, respectively) which is unhelpful when determining if the proposed solution works in a certain environment.

Overall, the results reveal that the community has a strong bias for one domain (scientific) and favors scientific fields (especially bioinformatics). We conjecture the large amount of open-access data from these fields facilitates this bias. This is consistent with our findings **O-4** and **O-5**, and with the assumption of people selecting traces with equal probability. An alternative is that the domains and fields whose data are used more, share artifacts that are more easily reused and rerun. An example of a well-known initiative for reproducibility in the scientific domain is the MyExperiment repository [24]. To overcome such biases, and to further reduce the large fraction of uncategorized traces evident in both plots of Fig. 3, we posit the community should require that open-access and *diverse* traces be used in articles claiming the generality of their techniques and indicate the domains and fields of the workflows used.



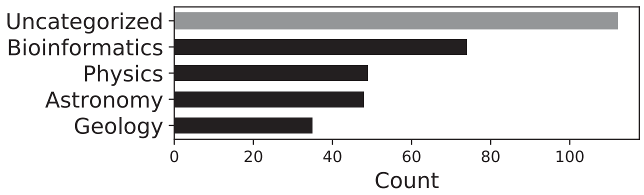


Fig. 3. (*top*) Top-5 (out of 6) domains and (*bottom*) Top-5 (out of 28) fields from which the community sources workflows. ("Uncategorized" for unclear domain or field.)

3 The Workflow Trace Archive

In this section, we outline the design of the WTA, the unified trace format used, tools to support consumers with the trace selection according to their use-case, and give a summarized overview of the current contents of the archive. Furthermore, that facilitates the continuous growth of the archive, we provide tools for trace anonymization and a collection of trace parse scripts for different trace sources.

Similar to how the design of experiments is now commonly described in publications in our field, as the setup leading to experimental results, we include an overview of the design process that led to the design presented in this section. Outlining the design and the process that led to the design is important for understanding how the final design came to be and how it fits the intended goal [25].

We started by listing initial requirements (see Section 3.1) that the WTA has to fulfill, and *co-evolved the requirements with the development of the solution* (the archive). For example, we added explicitly the requirement to provide scripts and datasets to aid users in building their own tools, as we discovered how difficult it was to engineer them from scratch (see Section 3.6). Next, we defined an initial format, centered around a number of unique features, such as non-functional requirements (NFRs) that are missing in other workflow trace formats. We improved this format iteratively, to meet the requirements and/or to pass various thought experiments. For the latter, whenever we encountered a new data-format that was not fully covered by our format, we discussed which properties and/or objects should be added to the format (see Section 3.4). We assessed the trade-off between format comprehensiveness (what to include?) and brevity (what is too much or too complex?) based on personal experience, on the perceived importance of data-fields in literature, and on their frequency of use in other archives. Finally, we designed the analysis tools iteratively, including in them initially our own ideas and then aspects highlighted by other archives, literature reports, and perceived shortcomings.

3.1 Use Cases and Requirements

We foresee four direct use cases for the WTA. First, trace characterization and workload analysis for understanding and tuning systems. As workloads evolve it is important to characterize the changes in, e.g., structure and resource consumption to see if schedulers require change, can be improved or if these changes can be exploited. Such characterizations can provide interesting insights (see Section 4).

Second, experimentation using emulation or simulation. As discussed in Section 1, emulations and simulations may be the only viable option for specific scenarios (e.g., what if?, long-term operational analysis). Having an archive that offers diverse, heterogeneous traces allows for more diverse testing scenarios. Especially when a new scheduler is developed for multiple domains or scenarios, it is important to experiment with diverse workloads covering the scenarios and domains targeted (see Section 5, **C-1**).

Third, workload and operational models can arise from the characterization and simulation results. In turn, these models can lead to new insights or to new variations to experiment with.

Last, such data can be used for education and training. As systems grow more complex, education and training becomes more important for both students and employees [26]. Models and heterogeneous traces are useful in education, to demonstrate scenarios and to provide hands-on experience.

To meet these use cases, we identify five key requirements for the structure, content, and operation of a useful archive for workflow traces.

R-1*: Diverse Traces for Academia, Industry, and Education.* Trace archives, such as Google's and Alibaba's, offer only workloads from a single domain, e.g., industrial workloads.

We identify as requirement that an archive must include a diverse set of traces to cover a broad spectrum of workflow sizes, structures, and other characteristics, including both general characteristics to many domains and fields, and idiosyncratic characteristics corresponding to only one domain or field. This requirement is based on the conjectures that different traces can have workflows with significantly different characteristics (tested in Section 4) and such differences impact system performance (tested in Section 5, **C-1**).

Addressing this requirement is important for academia to demonstrate the generality and applicability of a novel approach, for industry to test production-ready systems or to validate techniques proposed by academia [27], and for education to train employees on more complex systems.

R-2: A Unified Format for Workload of Workflows Traces.

To improve the reusability of diverse traces and to support the reproducibility of experimental results, long-term, we identify as a requirement the use of a unified trace format for workloads of workflows. The format must cover a broad set of data about the workloads and about the workflow management systems including: workload metadata; workflow-level data including NFRs; task-level data including per-task NFRs and operational metadata; inter-dependencies between tasks and other operational elements such as data transfers; system-level information including resource provisioning, allocation, and consumption; etc.

Addressing this requirement simplifies trace exchange and integration effort, prevents redundant work for other users, and supports the development of dataset independent tools (expressed as **R-3**).

R-3*: User level adapted insights into Trace Properties.*

To improve trace discovery, the archive must provide detailed trace insights adapted to the level of the broad audience, from beginner to expert, as implied by **R-1**. Broad insights include *extrinsic properties*, such the number of workflows and tasks, and *intrinsic properties*, such the workflow arrival patterns and the resource consumption per-task. In contrast, detailed expert-level insights include *analysis of single traces* at workload-, workflow-, and system-level; and *collective analysis* across all traces or traces filtered by a feature (e.g., all traces of a domain or field). These properties must be accessible through readily available tools (see **R-4**) and, possibly, through interactive online reports. Addressing this requirement helps to correlate information across different traces, resulting in better quantitative evidence, intuition about otherwise black-box applications, and understanding that helps avoiding common pitfalls [28].

R-4*: Tools for Trace Access, Parsing, Analysis, Validation.*

The most important tool is the online presence of the archive itself. The archive must further provide tools to parse traces from different sources to the unified format (see also **R-2**), to provide insight into traces (see also **R-3**), and to *validate* common properties (e.g., the presence of and correctness of properties). An absence of such tools would lead to users unable to select appropriate traces, validate their properties, and compare them.

The archive should further aid users in building more sophisticated tools. Newly built tools can then be added to the selection of tools so more parties can make use of them (contributing to **R-5**)

R-5*: Methods for Contribution.*

The archive must reflect the continuous evolution of workflow use in practice, by increasing the coverage of different scenarios. We make a distinction between two types of contribution: (1) traces from a new domain or application-field, and (2) traces, introducing new properties. To facilitate the former contribution, the archive must provide a method for the upload and (basic) automated traces verification. To facilitate the latter, the format must integrate specific provisions that enable upgrades and long-term maintainability, such as adding a version to each component of the format.

Addressing this requirement encourages new and existing contributors to submit new traces. In particular, tools to add new domains are of particular importance, to support emerging paradigms with realistic data.

3.2 Overview of the WTA

We design the WTA as a process and set of tools helping a diverse set of stakeholders. We consider three roles for the WTA community members, outlined in Fig. 1. The *contributor* supplies, as the legal owner or representative, one or more traces to the WTA. A workflow trace contains historical task execution data, resource usage, NFRs, resource description, inputs and outputs, etc. To fulfill **R-5**, the *WTA team* assists the contributor in parsing, anonymizing, and converting the traces into the unified format (Section 3.4), minimizing the risk of competitive disadvantage, and verifying their integrity. WTA fulfills **R-1** as it incrementally expands with contributors of traces from different domains with different properties.

The *user* represents non-expert or expert trace consumers. Non-expert users often need to rely on generic domain or trace properties, whereas the expert users have detailed knowledge of their system and require fine-grained details for selecting the correct trace. In addition, expert users may comment on (missing) properties and may develop new tools, models or other techniques to further compare and rank the traces. Both user types require assistance in selecting

TABLE 2
Overview of the Current WTA Content, Grouped by Source

Source ID. Name	#WL	D	DS	#PA	#PL	#S	#A	#WF	#T	#U	#G	Year(s)	Timespan	TCH
S1. Askalon Old	2	Eng	-	-	1	-	mixed	4,583	167,677	*7	*6	2007	19 months	4,685,300
S2. Askalon New	67	Sci	-	*2	2	67	*3	1,835	91,599	*67	*67	2016	47 days	193
S3. LANL	2	Sci	-	-	1	-	mixed	1,988,397	475,555,927	-	-	2011-2016	63 months	*9,625,431
S4. *Pegasus*	8	Sci	-	-	*6	-	8	56	10,573	9	-	2011	4 days	1,477
S5. Shell	1	Ind	-	-	1	-	mixed	3,403	10,208	-	-	2016	10 minutes	25
S6. SPEC	2	Sci	-	-	1	-	mixed	400	28,506	-	-	2017	-	1,231
S7. Two Sigma ‡	2	Ind	-	-	1	-	mixed	41,607,237	50,518,481	610	1	2016	16 months	69,992,196
S8. *WorkflowHub*	10	Sci	*5	*4	5	-	3	10	14,275	10	-	2017	-	52
S9. *Alibaba*	1	Ind	-	-	1	-	mixed	4,210,365	1,356,691,136	1	1	2018	8 days	1,526,925,484
S10. Google	1	Ind	-	1	1	-	mixed	494,179	17,810,002	430	1	2011	29 days	434,821,345
Total	96	-	*5	*7	*20	67	-	48,310,465	1,900,898,384	*1,134	*76	-	-	2,046,052,734

*Legend: D = Domain, DS = Datasets, PA = parameters, PL = Platform, S = Setup, A = Applications, WL = workload, WF=workflow, T = task, U = user, G = group, * = minimum, Eng = Engineering, Sci = Scientific, Ind = Industry, and TCH = Total Core Hours. Items in bold are workloads introduced by this work. Items where workflows are for the first time analyzed in this work are in* italics. *The symbol ‡ next to S7 indicates data with promise to release, but for which the legal forms have not been completed yet; WTA can already release all other workloads.*

the most suitable trace given a set of criteria (Section 3.5) as well as analysis and validation (Section 3.6) from the available set of traces (Section 3.7). To support both user types, the WTA discloses both high-level and low-level details.

3.3 Workflow Model

There are numerous types of workflow models used across different communities. A 2018 study by Versluis *et al.* finds DAGs are the most commonly used formalism in computer system conferences [29]. Popular formalisms such as CWL [30] and Condor DAG [31] are also DAG-based. Therefore, for the first design of this archive, we adopt DAGs as the workflow model.

A workflow constructed as a DAG in which nodes are computational tasks and directed edges depict the computational or data constraints between tasks. Entry tasks are tasks with no incoming dependencies and, once submitted to the system, immediately are eligible for execution. Similarly, end tasks are nodes that have no outgoing edges. A collection of workflows submitted to the same infrastructure over a certain period of time is considered a *workload*.

Although popular, we specifically do not focus in this work on BPMN and BPEL, Petri nets, hyper graphs, general undirected, or cyclic graphs. These formalisms either include business and human-in-the-loop elements [32] or add additional complexity due to having a large set of control structures such as loops, conditions, etc. [9] which we consider out of scope for this work.

Executable formalisms are meant to define what resources and software should be available *before* execution. Our formalism needs to capture the system state *during* execution. Both types of formalisms are needed, and are complementary to each other. For example, a person could use the CWL to define and run their workload, then turn to our formalism and tools to analyze its execution and subsequently improve various operational aspects.

Given the different nature of these formalisms, if we were to extend an existing executable workflow formalism, e.g., CWL, several elements would not be used. This would lead to *feature creep*. Conversely, the additions made by our formalism could be regarded as feature-creep by the CWL community. This is emphasized by the CWL community currently developing CWLProv [33]. This formalism aims at fully reproducible workflows, including re-execution which is not a goal of the WTA. While promising, CWLProv is still a work in progress; elements such as capturing resource usage (e.g., CPUs and power consumption) are still lacking.

3.4 Unified Trace Format

Creating a unified format (**R-2**) requires from the designer a careful balance between limiting the number of recorded fields while supporting a diverse set usage scenarios for all stakeholders in Section 3.2. Modern logging and tracing infrastructure can capture thousands of metrics for each machine and workflow-task involved [34], from which the designer must select. We specifically envision support for common system and workflow properties found in the typical scenarios considered in the top venues surveyed in Section 2, such as engineering a workflow engine [35], characterizing the properties of workloads of workflows [36], and designing and tuning new workflow schedulers [37].

Our unified format attempts to cover different trace *domains*, while preserving valuable information, such as resource consumption and NFRs, contributing to fulfilling **R-1 and 3**. The full technical description of the format can be found in our technical report [38] and on the WTA website.[2] By analyzing the raw data formats, we carefully selected useful properties to include in our unified format, omitting low-level details, such as cycles per instruction, page cache sizes, etc.

Answering RQ-2 and fulfilling **R-2**, our trace format is the first to support arbitrary NFRs both at task and workflow levels. For example, one of the LANL traces (introduced in Table 2) contains deadlines per workflow and the Google cluster data features task priorities, both are supported by the WTA unified format. Capturing these properties is important to test QoS-aware schedulers.

As depicted in Fig. 4, the WTA format includes seven objects: *Workload*, *Workflow*, *Task*, *TaskState Resource*, *ResourceState*, and *DataTransfer*. Each of these objects contains a version field, updated whenever the set of properties is altered (**R-5**).

2. https://wta.atlarge-research.com/traceformat.html

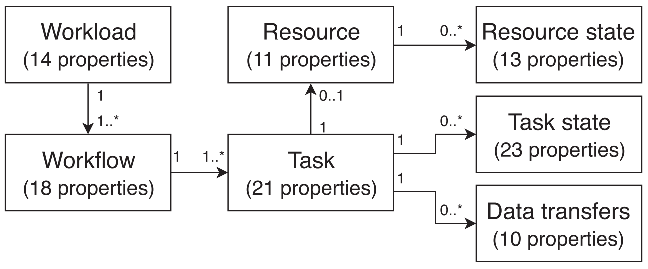


Fig. 4. The WTA trace format.

Each trace is a single *workload*, consisting of multiple workflows and their *arrival process*. Workload properties include the number of workflows, tasks, users, domain and field when available, authors list, and resource consumption statistics. Each workload belongs to one or more *domains*. and contains a *description* including its source, execution environment, etc.

Each *workflow* in the workload has a unique identifier, an arrival time, and contains a set of tasks and several properties, including scheduler used, number of tasks, critical path length, NFRs, and resource consumption. Each workflow also has the *name* of its *field* of study, when possible. Different related fields constitute a domain.

Each *Task* has a unique identifier and lists its submission and waiting time, runtime and resource requirements, including required (compute) resource type, memory, network, and energy usage. Additionally, each task provides optional dictionaries for task-specific execution parameters and NFRs. To model dependencies between tasks, the WTA format maintains for each workflow its topology by specifying parents and children per task. Similarly, data dependencies are recorded as a list of data transfers.

Resource objects cover various resource types, such as cloud instances, cluster nodes, and IoT devices. A resource has a unique identifier and contains several properties, such as resource type (e.g., CPU, GPU, threads), number, processor model, memory, disk space, and operating system. An optional dictionary provides further details, such as instance type or Cloud provider. The *ResourceState* event snapshots periodically the resource state, including availability and utilization. Analogous to the ResourceState, the *TaskState* records periodically the resource consumption of the task (the Task object records the resource demand).

Each *DataTransfer* describes a file transfer from a source to a destination task, which can be a local copy on the same resource or a network transfer from a remote source, etc. To support bandwidth analysis, a data transfer introduces submission time, transfer time, and data size. Each data transfer also provides an optional dictionary with detailed event timestamps (e.g., pause, retry).

3.5 Mechanisms for Trace Selection

We address **R-3** by assisting archive users in retrieving appropriate traces for their scenarios, using filter and selection mechanisms. The website is the most important such filter and mechanism, containing an overview of all traces in a general table with the number of workflows, tasks, users, etc. This table is sortable and searchable, allowing website users to interact with the more than 90 traces currently in the WTA (column "#WL", row "Total" in Table 2).

TABLE 3
Trace Anonymization Methods Used in WTA Tools

Obfuscation method	Description
IP	Encodes IPv4 addresses
Mail and host	Obfuscate mail and host names
File paths	Hide file paths in Linux and Windows format
Executable files	Encode executable file names, e.g., py, sh, exe, jar
All files	Hide all file names, ending with 2, 3, or 4 letters
Keywords	Anonymize a list of custom keywords
All	Apply all obfuscation methods listed above

We provide, online and as separate tools, a detailed report for each trace. Each report includes automatically generated statistics, such as the number of workflows and tasks, then resource properties such as compute, memory, and IO, and job and task arrival times and runtime distributions (see Section 4). The metrics featured in the report are reported as important by prior studies [39], [40] and enable developers to select traces appropriate for their intended use-case.

3.6 Tools for Analysis and Validation

We implement the unified trace format using the Parquet file format and the Snappy compression algorithm. Parquet is a binary file format that is supported by many big data tools such as Apache Spark, Flink, Druid, and Hadoop [41]. Many programming languages also have libraries to parse this format, such as PyArrow for Python and parquet-avro for Java. Snappy[3] compression reduces the size of the dataset significantly and has low CPU usage during extraction.

Beside trace selection support and to address **R-4**, the WTA offers several tools to facilitate and incentivize the continuous growth of the archive. Most of these tools required significant engineering effort to develop, due to the typical challenges of big data processing (high volume, noisy data, diverse input-formats, etc.). The WTA simplifies the upload of new traces by providing a set of parsing scripts for different trace sources, such as Google, Pegasus, and Alibaba. Parsing traces can become non-trivial, once they grow both in complexity and size. Such traces require big data tools, such as Apache Spark, and enough resources, a cluster, to compute. Noisy data raise another non-trivial issue: both Google's and Alibaba's cluster data contained either anomalous fields, undocumented attributes, and non-DAG workflows. Some of these issues were never discovered by their respective communities and were corrected in our parsing tools. Debugging, filtering, and correcting noisy big data requires significant compute power and detailed engineering.

Because traces may contain sensitive information, the WTA offers a *trace anonymization tool*, which supports users to automatically replace privacy and security-related information, to avoid an accidental reveal of proprietary information. Specifically, to remove sensitive information from trace files, we use two common techniques [42], *culling* and *transforming*. Culling is done during trace conversion, by omitting parts of the raw trace data which do not match our workflow trace format. For the transformation, as presented in Table 3, our anonymization tool automatically scans the

3. https://github.com/google/snappy

TABLE 4
Overview of Properties Available Per Source

Source ID	Task details	Task resource req.	Structural information	Disk	Memory	Network	Energy	NFRs
S1	✓	✓	✓					
S2	✓	✓	✓					
S3	~	✓	~					✓
S4	✓	✓	✓					
S5	✓	✓	✓					
S6	✓	✓	✓			✓		
S7	✓	✓	~		✓			
S8	✓	✓	✓		✓	✓		
S9	✓	✓	✓	✓	✓	✓		
S10		✓	~	✓	✓			✓

Legend: ✓ = available, ~ = partially available, blank = not available, and Task details = individual task information.

workflow trace file for sensitive data, such as IP addresses, file paths, names, etc., by string pattern matching. Beside these standard sensitive-data checks, the WTA offers the option to search for custom privacy-critical strings.

Finally, all matched strings are replaced by a salted SHA-256 hash key. This approach using cryptographic hash functions offers protection of sensitive data, while preserving the relationships between the matched values in the same trace file [42]. Additionally, our tool hides potential relations to other trace files by adding a salt of length 16 to the hash key generation, which is randomly generated on each tool run.

To validate traces, the WTA provides a *validation* script that checks the integrity and summarizes important characteristics of a trace. During trace conversion, using the validation script, we successfully identified several parse bugs and inconsistencies in the data that we subsequently corrected.

Specifically, because tasks build the base of each trace, our tool checks if all contained tasks are well defined. This, for example, means that all parsed control dependencies, such as children and parents, link only to existing tasks with valid properties. A task property is valid, if the parsed property type matches the property type definition, and the property value is allowed e.g., task runtime > 0. Based on and similar to this fundamental validation, our tool provides options to check the workflow and data transfer properties to identify inconsistencies, as well.

These tools help combating perceived barriers to share data described by Sayogo *et al.* [43]. Several technological barriers are addressed by using a unified format and validation (data architecture, quality, and standardization), Legal and policy barriers are more difficult to address. Our anonymization tool aids in overcoming the data protection barrier, yet legal and other enforced policies may require tailored solutions.

Besides offering these tools, the WTA also hosts the trace data, addressing logistic and economical barriers. The increasing focus on sharing data artifacts by the community, is lowering the barrier regarding competition for merit and reputation for quality and bolsters the culture of open sharing. Finally, each trace has its own DOI by also uploading it to Zenodo[4] which can be cited and thus provides authors with the appropriate credits (incentive barrier).

3.7 Current Content

Having a diverse set of traces available is necessary to use in experimentation. When using traces in experimentation, different traces should be used to prove generality of the proposed approach (see Section 5). Gathering and parsing raw logs and other traces requires significant computing effort. Using 16 nodes (32 eight-core Xeon E5-2630 v3 and 1TB RAM) from the Dutch DAS5 super computer [44], several traces require up to a day to compute using big data tools such as Apache Spark. In total, the WTA team spent more than two person months on converting traces to the unified trace format. By offering these parse scripts and the data, we contribute to **R-4**.

The WTA features currently 96 workloads from 10 different sources, with over 48 million workflows and 2 billion CPU core hours. All of them are available on the WTA archive website.[5] Each workload is uniquely identified by a combination of the following properties if available: source, runtime environment, application, and application parameters [45]. Tables 2 and 4 summarize these traces. From these tables we observe that WTA contains a vast amount of different traces, from different sources and domains, with various number of workflows, properties, number of tasks, timespans, and core hour counts. Although supported by our format, no trace currently has information on energy consumption, highlighting the need of such traces [14]. These traces are collected by combining open-access data (logs, traces, etc.) and closed-access data throughout the years in collaboration with both industry and academia. This contributes to **R-1**.

This diversity enables new workflow management techniques and systems to be thoroughly tested for their feasibility, strengths, and, equally important, weaknesses.

4 A Characterization of Workloads of Workflows

To answer RQ-3, we perform in this section a characterization of the workloads in the WTA. These workloads originate from publicly available archives combined with workloads we obtained from collaborations. As we expect these workloads to be heterogeneous in many dimensions, we characterize them using a variety of metrics and properties, including workflow size, resource usage, and structural

4. https://zenodo.org/

5. https://wta.atlarge-research.com/

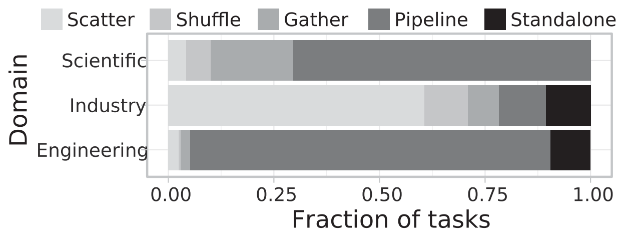


Fig. 5. Structural workflow patterns, per domain.

patterns. Our characterization reveals significant differences between workloads from different domains and sources. Such differences further support our claim that the community needs to look beyond just scientific workloads, and consider a wider range of domains and sources for experimental studies when developing workflow management systems aimed at multiple domains or for general applicability.

We present in this section only detailed insights that lead to new observations for the community. We include in our technical report other types of analysis, such as task and workflow inter-arrival times, task and workflow runtimes, and their breakdown per domain and source [38].

4.1 Structural Patterns

O-7: Scientific, industrial, and engineering workflows exhibit various structural patterns, but at least 60 percent of tasks in a domain match the dominant pattern of that domain.

O-8: Industry workflows stand out by exhibiting primarily scatter patterns, as opposed to pipeline operations.

This characterization quantifies five structural patterns in workflows often used by researchers [46]: scatter (data distribution), shuffle (data redistribution), gather (data aggregation), pipeline, and standalone (process). Investigating these structural patterns is important to understand the types of applications being executed and tune a system's performance. We exclude from this analysis the LANL, Two Sigma, and Google traces, which lack structural information, that is, task parent-child relationship information.

Fig. 5 depicts the structural patterns found per domain. From this figure, we observe that in each domain a dominant pattern emerges that accounts for 61–85 percent of tasks. In the scientific and engineering domains, the majority of tasks are simple pipelines. Interestingly, the industrial workflows include primarily scatter operations. This observation matches known properties of the Alibaba trace, which accounts for over 99 percent of tasks with structural information we analyzed in this domain. In particular, the Alibaba trace includes MapReduce jobs, each consisting of many "map" tasks (scatter operations) and a smaller number of "reduce" tasks (gather operations).

4.2 Arrival Patterns

O-9: From all domains, industrial traces show on average orders of magnitude higher rates of task arrival.

O-10: Scientific traces can show high variability in task arrival rates, unlike industrial and engineering traces.

O-11: Two Sigma shows a typical workday diurnal pattern.

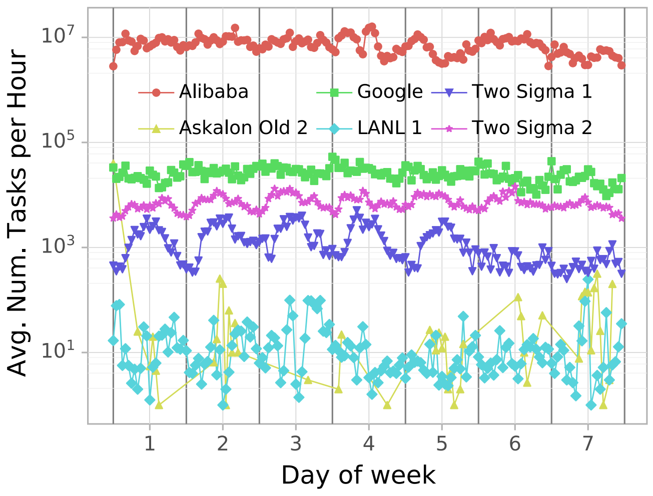


Fig. 6. Daily task-arrival trend, per source.

To investigate the weekly trends that may appear in workload traces, we depicts in Fig. 6 for several traces the average number of tasks that arrive per day of the week. We omit the Askalon new source from the hourly task-arrival plot as they contain 4 or 5 data points, which is too few to plot a trend. We observe that traces have significantly different arrival rates and patterns. The Alibaba trace features the highest task arrival rates, peaking at over 10,000,000 tasks per hour. Google and the Two Sigma workloads follow with 100-10,000 tasks per hour. This shows that industrial workloads included in this work have significantly more tasks per hour than the other compute environments, which agrees with companies such as Alibaba and Google operating at a global scale. The non-industrial traces show significant fluctuations throughout the week, whereas both Alibaba and Google do not. This might be due to the global, around-the-clock operation of Alibaba's and Google's services, which can lead to a more stable task arrival rate.

To observe differences in daily trends, we depict the average task rate per hour of day in Fig. 7. This figure reaffirms our observation that the two largest traces–Alibaba and Google–have a relatively stable arrival pattern throughout the day. In contrast, the Two Sigma 1 trace exhibits a

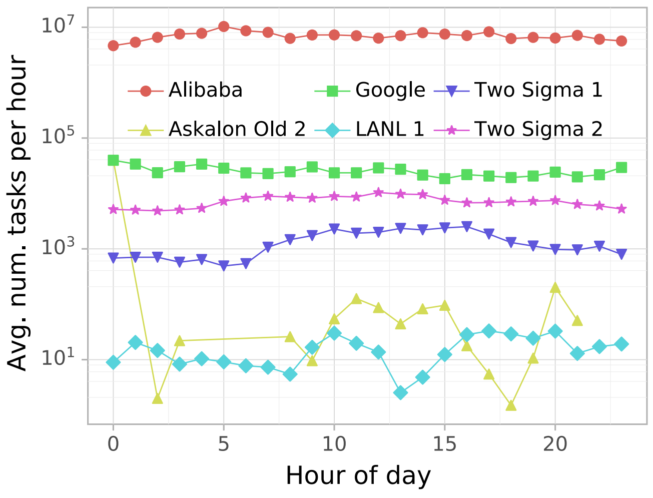


Fig. 7. Hourly task-arrival trend, per source.

TABLE 5
The Design and Setup of our Characterization

ID	Section	Description	Traces	Metric	Granularity
E1	4.1	Analyze structural patterns in workflows per domain	All but S3, 7, 10	Structural patterns	Workflow level
E2	4.2	Longitudinal analysis	S1, S3, S7, S9, S10	Tasks per day	Workload level
E3	4.3	Analysis of burstiness per trace	All but S4-8	Hurst exponent	Workload level
E4	4.4	Measure the level of parallelism per workflow	All but S3, 7, 10	Level of parallelism	Workflow level
E5	4.5	Analysis of critical path length	All but S3, 7, 10	Critical path length	Workflow level

typical office hours pattern; task arrival rates increase around hour 7 and start dropping around 17. The same pattern occurs to a lesser extent in the Two Sigma 2 trace. The highly variable arrival rates of tasks in the LANL traces, as observed in Fig. 6, are also evident in our analysis of daily trends. We study this in more depth in Section 4.3.

4.3 Burstiness

O-12: Most traces investigated exhibit bursty behavior within small window sizes.

O-13: The LANL trace exhibits maximum burstiness at medium window sizes.

O-14: The largest traces (Alibaba and Google) exhibit uniquely bursty behavior: low burstiness at small and high burstiness at large, window sizes.

To investigate if workloads expose bursty behavior, a special kind of arrival pattern, the Hurst exponent H is used. H quantifies the effect previous values have on the present value of the time series. A value of $H < 0.5$ indicates a tendency of a series moving in the opposite direction based on the previous values, and thus exhibit jittery behavior (sporadic burst). A value of $H > 0.5$ indicates a tendency to move in the same direction, and thus towards well defined peaks (sustained burst). When $H = 0.5$, the series behaves like a random Brownian motion.

In this experiment, we inspect busty behavior by computing the Hurst exponent for task arrivals. The results of this experiment are visible in Fig. 8. From this figure, we observe most traces depict bursty behavior at least for one of small, medium, and large window size. They are also not bursty for at least one window size. This is expected, as in most systems task arrivals vary at (sub-)second interval. Interestingly, LANL traces exhibit most bursty behavior at medium window sizes. This might be due to national laboratories workflows being submitted in batches. A batch of tasks is submitted all at once, leading to a burst. But, the batch itself is processed at a constant rate. The workload is also stable over longer time periods as evidenced by $H \approx 0.5$ for larger windows. Finally, the two largest traces in this work, Alibaba and Google, exhibit increasingly burst behavior for larger windows. This indicates that for larger arrival times, the workloads (in absolute numbers) vary more than for the other sources. This matches the observations in Section 4.2.

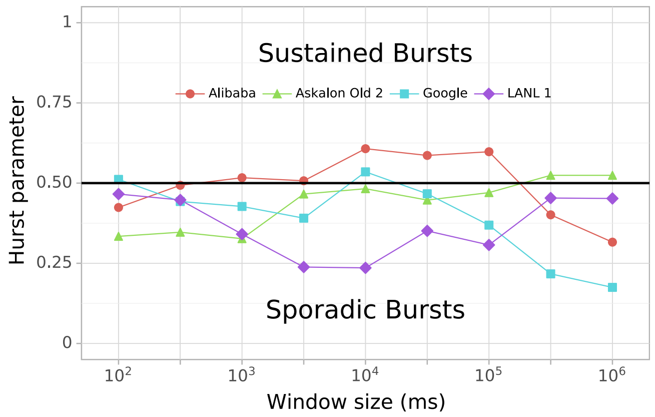


Fig. 8. Hurst exponent estimations for different time windows per trace. Horizontal axis does not start at zero.

4.4 Parallelism in Workflows

O-15: Task parallelism per workflow can differ significantly between workload domains and sources.

O-16: Industrial workflows exhibit the highest level of parallelism.

O-17: Out of all sources, Alibaba workflows have the highest level of parallelism, followed by Pegasus and WorkflowHub.

With the structural patterns observed, we investigate if the large occurrence of the pass-through patterns expresses in a high level of parallelism. The level of parallelism indicates how many tasks can maximally run in parallel for a given workflow, provided sufficient resources. Fig. 9 depicts the approximated level of parallelism per domain. The approximation algorithm used produces results very close to the true level of parallelism as demonstrated by Ilyushkin *et al.* [47]. From this figure, we observe the industrial domain exhibits the highest level of 99th percentile parallelism, up to hundreds of thousands of tasks. This is likely a consequence of the many MapReduce workflows, which are highly-parallel by nature, that are present in the Alibaba trace. Alibaba also contains bag of tasks workflows, which by nature have a high parallelism. Scientific workflows exhibit low median parallelism but high 99th percentile parallelism, featuring levels of parallelism up to thousands of tasks. Engineering traces exhibit a moderate amount of

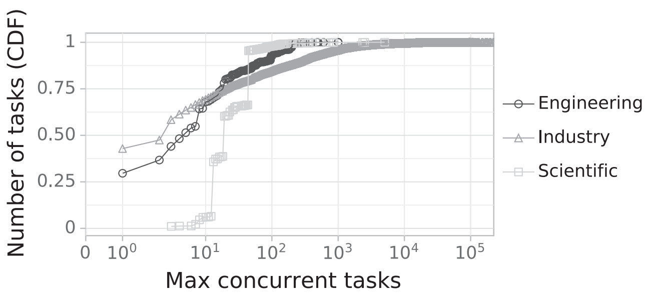


Fig. 9. Workflow level-of-parallelism, per domain.

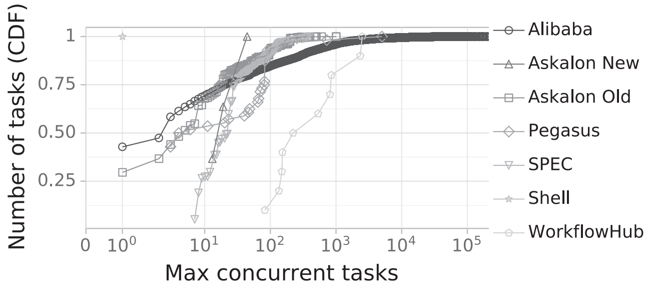


Fig. 10. Workflow level-of-parallelism, per source. Curves are shaded by domain, to further reveal patterns.

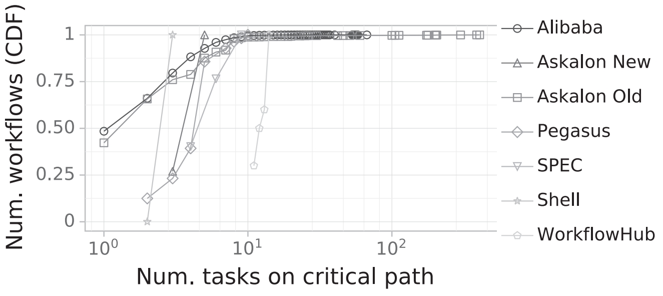


Fig. 12. Workflow level-of-parallelism, per source. Curves are shaded by domain, to further reveal patterns.

median parallelism, between industry and scientific, with at most 1000 concurrent running tasks.

Fig. 10 shows the level-of-parallelism per source. From this figure, we observe that Alibaba exhibits the highest levels of parallelism, as discussed previously. Second are the Pegasus and WorkflowHub workflows. These sources contain a variety of scientific applications, commonly known for their parallel structures, as observed in Section 4.1. Other traces demonstrate less parallelism, with up to 1000 concurrent running tasks. As Shell exist entirely of sequential pipelines, the source does not exhibit any variation.

4.5 Limits to Parallelism in Workflows

O-18: Workflows from the scientific domain have significantly different critical-path lengths.

O-19: The amount of tasks on the critical path is the highest for engineering workflows.

O-20: Although highly parallel, industrial workflows exhibit longer critical paths than scientific workflows.

The critical path (CP) refers to the longest sequence of dependent tasks in a workflow, from any entry task to any exit task. By quantifying the CP length, we investigate if workflow runtimes are primarily dominated by a few heavy tasks, or by many small tasks. Fig. 11 presents the results of this characterization per workload domain. From this figure we observe the CP length for engineering workflows is the highest. This matches with the parallelism observations in Sections 4.1 and 4.4. Interestingly, even though industrial workflows are often highly parallel, their critical paths are often longer than those of scientific workflows. This indicates that industrial workflows are bigger in size than scientific workflows, which our data supports.

Fig. 12 presents the results of CP characterization per workload source. From this figure we observe the CP length differs significantly per trace. Based on the prior findings, the engineering traces are expected to show longer critical paths. As we can observe, the Askalon old traces contains workflows with the longest critical path. Alibaba workflows also exhibit long critical paths, indicating their workflows next to being highly parallel, also contain a lot of tasks with stages. More concentrated, the other traces exhibit lower critical path lengths, yet the traces are still clearly distinct. As the Shell trace contains solely sequential workflows, the critical path length is one.

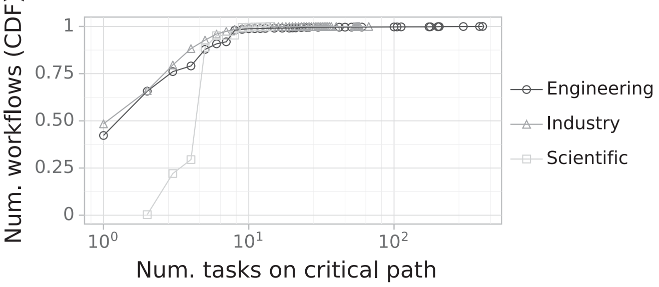


Fig. 11. Workflow level-of-parallelism, per domain.

5 Addressing Challenges of Validity

In this section, we discuss challenges to the validity of this work. We address the challenges through either trace-based simulation (the first) or argumentation (the others).

Challenge **C-1. Trace diversity does not impact the performance of workflow schedulers.** As outlined in Sections 3.7 and 4, the WTA traces are diverse. However, *what is the impact of trace diversity?*

To demonstrate the impact of trace diversity on scheduler performance, we conduct a trace-based simulation study. The simulator used is an optimized version of DGSim [48] which we publish as open-access artifact.[6] We simulate workloads from five sources using two scheduler configurations. We equip the simulated scheduler with either the first-come first-serve (FCFS) or the shortest job first (SJF) queue sorting policy. For both scheduler configurations, we further use a best-fit task placement policy. We do not use a fixed resource environment to prevent bias when sampling or scaling traces [28]. Instead, we tailor the amount of available resources for each trace to reach roughly a 70 percent resource utilization on average, based on the amount of CPU (core) seconds of trace and its length. Although ambitious, 70 percent resource utilization is achievable in parallel HPC environments [49] and can be seen as a target for cloud environments. To evaluate the performance of each scheduler, we use three metrics commonly used to assess schedulers' performance [50], [51]: task response time (ReT), bounded task slowdown (BSD, using a lower bound of 1 second), and normalized workflow schedule length (NSL, the ratio between a workflow's response time and its critical path). The entire experiment, including software and data, can be reproduced on Code Ocean.[7]

We report the performance of each simulated scheduler in Table 6 per source. From this table we observe significant differences between schedulers and trace sources. In particular,

6. Available at https://github.com/atlarge-research/wta-sim
7. https://doi.org/10.24433/CO.8484557.v1

TABLE 6
The Performance in Simulation of Two Schedulers for Traces From Different Sources

		Source of Trace				
Metric	Policy	Askalon Old	Askalon New	Pegasus	Shell	SPEC
Avg. ReT	FCFS	$2.02 \cdot 10^5$ s	167 s	$2.43 \cdot 10^4$ s	9.76 s	491 s
	SJF	$1.74 \cdot 10^5$ s	113 s	$2.12 \cdot 10^4$ s	9.52 s	248 s
Avg. BSD	FCFS	$1.53 \cdot 10^4$	65.1	$1.31 \cdot 10^3$	1.13	47.4
	SJF	$0.14 \cdot 10^4$	11.6	$0.10 \cdot 10^3$	1.06	2.2
Avg. NSL	FCFS	$1.05 \cdot 10^5$	2.50	$2.35 \cdot 10^3$	1.12	13.9
	SJF	$0.01 \cdot 10^5$	3.14	$0.06 \cdot 10^3$	1.07	1.78

Lower values are better.

we find that the relative performance of schedulers differs between trace sources. For example, SJF outperforms FCFS on the normalized schedule length metric by up to two orders of magnitude on traces from Askalon Old and Pegasus. In contrast, on traces from Askalon New and Shell, the scheduling policies perform similarly. For other metrics, these differences are present, but less pronounced. SJF performs better than FCFS on response time and slowdown for each trace source, but the differences in performance between the schedulers vary greatly across traces.

Overall, we kept the working environment fixed per trace, yet obtained significantly different results depending on the scheduler and input trace. Thus, our trace-based simulations give practical evidence that researchers require experimenting with different traces to claim generality and feasibility of their proposed approaches.

C-2. Limited venue selection in the survey. Besides omitting venues that yielded no results on our initial query, we made sure that journals, workshops, and conferences were covered at various levels in term of quality. The selected venues are highly ranked in several of the available rankings, including CORE,[8] Google Scholar.[9] and AMiner.[10] As these rankings use different metrics to define the top-ranking, we made a selection that covers different types of venues that also match our experience in terms of quality, see the list in Table 1. We believe this covers the field of systems community to a degree where conclusions can be drawn from. We specifically focus on articles published in the systems communities as specialized communities, e.g., bioinformatics, focus on systems that solve domain-specific problems, but rarely conduct in-depth experiments, including trace-based, to test the system-level capabilities and behavior.

C-3. Level of data anonymization. The Google team published interesting work data [42], but their anonymization approach, of normalizing values of both resource consumption and available resources, reduces significantly the usability of traces and the characterization details they provide. We argue this type of anonymization is not preferred. When available resources per machine, e.g., available disk space, memory, etc., and resource consumption numbers are normalized, reusing traces for different environments becomes difficult. Researchers then need to make assumption on what kind of hardware the workflows were executed as done in the work of Amvrosiadis *et al.* [15] or need to assume a homogeneous environment. Instead, obfuscation techniques, such as multiplying both consumption and resources by a certain factor, allow for relative comparisons and the possibility to replay scheduling the workload on the resources while still concealing the original data.

C-4. The Workflow Trace Format. A fourth challenge is the properties included in the workload trace format. For each encountered property in other formats, we carefully decided whether to include it or not. Low-level details such as page caches are omitted to not complicate unnecessarily the traces. If future work demands change, the versioning schema per object will allow for these additions. In defining the fields of our trace format, we also looked at a variety of workflow specification languages and formalisms, from the very generic (e.g., BPMN/BPEL and Petri net) to the executable workflow formalisms (e.g., CWL and DAX).

6 Related Work

We survey in this section the relevant body of work focusing on trace archives and on characterizing workloads. Differently from other archives, the WTA focuses on *workloads of workflows*, preserving workflow-level arrival patterns and task inter-dependencies not found in other archives. Differently from other characterization work, ours is the first to reveal and compare workflow characteristics across different domains and fields of application.

Open-Access Trace Archives. Closest to this work is WorkflowHub [14], which archives traces of workflows executed with the Pegasus workflow engine and offers them in a unifying format containing structural information. WorkflowHub also provides a tool to convert Pegasus execution logs to traces, similar to our parsing tools. Different from this work, WorkflowHub's traces include a single workflow and thus not a workload with a job-arrival pattern. WorkFlowHub also does not provide statistical insights per trace and thus, they do not meet requirements **R-1** and **R-3**, and only partially meet **R-4**.

Also relatively close to this work, the ATLAS repository maintained by the Carnegie Mellon University [15] contains two traces (the S3 traces in this work), with other two traces announced but not yet released (as announced, the S7 traces in this work). None of their published traces contains task-interdependency data, so, although overlapping with our S3 and S7, the ATLAS work is different in scope and in particular does not address workflows. Further, they do not consider different domains nor fields, and their archive lacks a unified format, statistical insights, selection mechanisms, and tooling—thus, they do not meet our requirements **R1–4**.

Other trace-archives with similarities to this work include the MyExperiment archive (ME) [24], the Parallel Workloads

Archive (PWA) [52], and the Grid Workloads Archive (GWA) [53]. ME stores workflow executables, and semantic and provenance-data, but not provide execution traces as WTA does and thus has different scope. The PWA includes

8. http://portal.core.edu.au/conf-ranks
9. https://scholar.google.com/citations?view_op=top_venues&hl=en&vq=eng_computingsystems
10. https://cn.aminer.org/ranks/conf

traces collected from parallel production environments, which are largely dominated by tightly-coupled parallel jobs and, more recently, by bag-of-tasks applications. The GWA includes traces collected from grid environments; differently from this work, these traces are dominated by bag-of-tasks applications and by virtual-machine lease-release data.

Workload Characterization, Definition, and Modeling. There is much related and relevant work in this area, from which we compare only with the closely related; other characterization work does not focus on comparing traces by domain and does not cover a set of characteristics as diverse as this work, leading to so many findings. Closest to this work, the Google cluster-traces have been analyzed from various points of view, e.g., [54], [55], [56]. Amvrosiadis *et al.* [15], [23] compare the Google cluster traces with three other cluster traces, of 0.3-3 times the size and 3-60 times the duration, and find key differences; our work adds new views and quantitative data on diversity, through both survey and characterization techniques. Bharathi *et al.* [46] provide a characterization on workflow structures and the effect of workflow input sizes on said structures. Five scientific workflows are used to explain in detail the compositions of their data and computational dependencies. Using the characterization, a workflow generator for parameterized workflows is developed. Juve *et al.* [36] provide a characterization of six scientific workflows using workflow profiling tools that investigate resource consumption and computational characteristics of tasks. The teams of Feitelson and Iosup have provided many characterization and modeling studies for parallel [57], grid [58], and hosted-business [59] workloads; and Feitelson has written a seminal book on workload modeling [60]. In contrast, this work addresses in-depth the topic of workloads of workflows.

7 Conclusion and Ongoing Work

Responding to the stringent need for diverse workflow traces, in this work we propose the Workflow Trace Archive (WTA), which is an open-access archive containing workflow traces.

We conduct a survey of how the systems community uses workflow traces, by systematically inspecting articles accepted in the last decade in peer-reviewed conferences and journals. We find that, from all articles that use traces, less than 40 percent use realistic traces, and less than 15 percent use any open-access trace. Additionally, the community focuses primarily on scientific workloads, possibly due to the scarcity of traces from other domains. These findings suggest existing limits to the relevance and reproducibility of workflow-based studies and designs.

We design and implement the WTA around five key requirements. At the core of the WTA is an unified trace format that, uniquely, supports both workflow- and task-level NFRs. The archive contains a large and diverse set of traces, collected from 10 sources and encompassing over 48 million workflows and 2 billion CPU core hours.

Finally, we provide deep insight into the WTA traces, through a statistical characterization revealing that: (1) there are large differences in workflow structures between scientific, industrial, and engineering workflows, (2) our two biggest traces– from Alibaba and Google—have the most stable arrival patterns in terms of tasks per hour, (3) industrial workflows tend to have the highest level of parallelism, (4) the level of parallelism per domain is clearly divided, (5) engineering workloads tend to have the most tasks on the critical path, (6) the three domains inspected in this work show distinct critical path curves, (7) in order to claim generality of an approach, one should test a system with a variety of traces with different properties, possibly from different domains.

In ongoing work, we aim to attract more organizations to contribute real-world traces to the WTA, and to encourage the use of the WTA content and tools in educational and production settings. One of our goals is to develop a library system administrators can integrate into their systems to generate traces in our format. Our preliminary experience with this learns that developing such a library, even for a single system, requires significant engineering effort and is thus left for future work. We aim to support other formalisms in the future, including directed graphs, BPMN workflows, etc. based on the community's needs. Investigating if formalisms such as CWLProv can be used to further enhance the archive's content, possibly by merging, is another interesting item for future work. Finally, we aim to improve the trace format and statistics we report for each trace, based on community feedback.

Reproducibility Statement

We support reproducible science. A full description on how to reproduce our findings can be found in our technical report [38]. The WTA datasets are available online on the archive's website https://wta.atlarge-research.com/. The WTA tools, simulator, and parse scripts and survey data are available as free open-source software at https://github.com/atlarge-research/wta-tools, https://github.com/atlarge-research/wta-sim, and https://github.com/atlarge-research/wta-analysis, respectively. The experiment conducted in Section 5, **C-1** can be reproduced using our Code Ocean capsule available at https://doi.org/10.24433/CO.8484557.v1.

Acknowledgments

This work was supported by the projects Vidi MagnaData, Commit, the European Union's Horizon 2020 Research and Innovation Programme, under Grant 801091 "ASPIDE", and the National Science Foundation award 1664162.

References

[1] A. Ilyushkin *et al.*, "An experimental performance evaluation of autoscaling policies for complex workflows," in *Proc. 8th ACM/SPEC Int. Conf. Perform. Eng.*, 2017, pp. 75–86.
[2] F. Wu *et al.*, "Workflow scheduling in cloud: A survey," *J. Supercomputing*, vol. 71, pp. 3373–3418, 2015.
[3] P. K. Isom *et al.*, *Is Your Company Ready for Cloud*. Indianapolis, IN, USA: IBM Press, 2012.
[4] E. Deelman *et al.*, "The future of scientific workflows," *Int. J. High Perform. Comput. Appl.*, vol. 32, pp. 159–175, 2018.
[5] K. Weins, "Cloud computing trends: 2018 state of the cloud survey. rightscale," 2018.
[6] M. Kelly, "86 percent of companies use multiple cloud services," 2012. [Online]. Available: https://venturebeat.com/2012/05/10/cloud-services-data/

[7] A. Iosup *et al.*, "Massivizing computer systems: A vision to understand, design, and engineer computer ecosystems through and beyond modern distributed systems," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 1224–1237.

[8] E. G. Coffman and R. L. Graham, "Optimal scheduling for two-processor systems," *Acta Inf.*, vol. 1, pp. 200–213, 1972.

[9] A. Slominski, "Adapting BPEL to scientific workflows," in *Proc. Workflows e-Sci.*, 2007, pp. 208–226.

[10] T. Murata, "Petri nets: Properties, analysis and applications," *Proc. IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989.

[11] Y. Gil *et al.*, "Examining the challenges of scientific workflows," *Computer*, vol. 40, no. 12, pp. 24–32, Dec. 2007.

[12] J. T. Dudley *et al.*, "In silico research in the era of cloud computing," *Nat. Biotechnol.*, vol. 28, no. 11, pp. 1181–1185, 2010.

[13] S. Cohen-Boulakia *et al.*, "Search, adapt, and reuse: the future of scientific workflows," *ACM SIGMOD Record*, vol. 40, pp. 6–16, 2011.

[14] R. F. da Silva *et al.*, "Community resources for enabling research in distributed scientific workflows," in *Proc. IEEE 10th Int. Conf. e-Sci.*, 2014, pp. 177–184.

[15] G. Amvrosiadis *et al.*, "On the diversity of cluster workloads and its impact on research results," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2018, pp. 533–546.

[16] L. Ramakrishnan *et al.*, "A multi-dimensional classification model for scientific workflow characteristics," in *Proc. 1st Int. Workflow Approaches New Data-Centric Sci.*, 2010, pp. 1–12.

[17] J. Cardoso *et al.*, "Workflow quality of service," in *Proc. Int. Conf. Enterprise Integration Modeling Technol.*, 2002, pp. 303–311.

[18] ACM, "Artifact review and badging," 2017. [Online]. Available: https://www.acm.org/publications/policies/artifact-review-badging

[19] B. Kitchenham *et al.*, "Systematic literature reviews in software engineering–a systematic literature review," *Inf. Softw. Technol.*, vol. 51, pp. 7–15, 2009.

[20] M. Ley, "The DBLP computer science bibliography: Evolution, research issues, perspectives," in *Proc. Int. Symp. String Process. Inf. Retrieval*, 2002, pp. 1–10.

[21] W. Ammar *et al.*, "Construction of the literature graph in semantic scholar," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Human Lang. Technol.*, 2018, pp. 84–91.

[22] W. Schwiegelshohn, "How to design a job scheduling algorithm," in *Proc. Workshop Job Scheduling Strategies Parallel Process.*, 2014, pp. 147–167.

[23] G. Amvrosiadis *et al.*, "Bigger, longer, fewer: What do cluster jobs look like outside google?" Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-PDL-17–104, 2017.

[24] C. Goble *et al.*, "myExperiment: Social networking for workflow-using e-scientists," in *Proc. 2nd Workshop Workflows Support Large-Scale Sci.*, 2007, pp. 1–2.

[25] A. Iosup *et al.*, "The atLarge vision on the design of distributed systems and ecosystems," in *Proc. Int. Conf. Distrib. Comput. Syst.*, 2019. [Online] Available as *arXiv e-print*, https://arxiv.org/pdf/1906.07471.pdf

[26] A. Iosup *et al.*, "The openDC vision: Towards collaborative datacenter simulation and exploration for everybody," in *Proc. 16th Int. Symp. Parallel Distrib. Comput.*, 2017, pp. 85–94.

[27] D. Klusáček *et al.*, "On interactions among scheduling policies: Finding efficient queue setup using high-resolution simulations," in *Proc. Eur. Conf. Parallel Process.*, 2014, pp. 138–149.

[28] E. Frachtenberg and D. G. Feitelson, "Pitfalls in parallel job scheduling evaluation," in *Proc. Workshop Job Scheduling Strategies Parallel Process.*, 2005, pp. 257–282.

[29] L. Versluis *et al.*, "An analysis of workflow formalisms for workflows with complex non-functional requirements," in *Proc. ACM/SPEC Int. Conf. Perform. Eng.*, 2018, pp. 107–112.

[30] Amstutz *et al.*, "Common workflow language, v1. 0," Figshare, 2016.

[31] P. Couvares *et al.*, "Workflow management in condor," in *Proc. Workflows e-Science*, 2007, pp. 357–375.

[32] W. V. der Aalst *et al.*, "Advanced topics in workflow management: Issues, requirements, and solutions," *J. Integr. Des. Process Sci.*, vol. 7, 2003.

[33] F. Z. Khan *et al.*, "Sharing interoperable workflow provenance: A review of best practices and their practical application in CWLProv," *GigaScience*, vol. 8, pp. 1–27, 2019.

[34] P. Xiong *et al.*, "vPerfGuard: An automated model-driven framework for application performance diagnosis in consolidated cloud environments," in *Proc. 4th ACM/SPEC Int. Conf. Perform. Eng.*, 2013, pp. 271–282.

[35] A. C. Zhou *et al.*, "A declarative optimization engine for resource provisioning of scientific workflows in IaaS clouds," in *Proc. 24th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2015, pp. 223–234.

[36] G. Juve *et al.*, "Characterizing and profiling scientific workflows," *Future Gener. Comput. Syst.*, vol. 29, pp. 682–692, 2013.

[37] Q. Sun *et al.*, "Adaptive data placement for staging-based coupled scientific workflows," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2015, Art. no. 65.

[38] L. Versluis *et al.*, "The workflow trace archive: Open-access data from public and private computing infrastructures – technical report," 2019, *arXiv:1906.07471*[cs.DC]. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2019arXiv190607471V

[39] Y. L. Simmhan *et al.*, "A framework for collecting provenance in data-centric scientific workflows," in *Proc. IEEE Int. Conf. Web Services*, 2006, pp. 427–436.

[40] W. Chen *et al.*, "Using imbalance metrics to optimize task clustering in scientific workflow executions," *Future Gener. Comput. Syst.*, vol. 46, pp. 69–84, 2015.

[41] S. Talluri *et al.*, "Characterization of a big data storage workload in the cloud," in *Proc. ACM/SPEC Int. Conf. Perform. Eng.*, 2019, pp. 33–44.

[42] C. Reiss *et al.*, "Obfuscatory obscanturism: making workload traces of commercially-sensitive systems safe to release," in *Proc. IEEE Netw. Operations Manage. Symp.*, 2012, pp. 1279–1286.

[43] D. S. Sayogo and T. A. Pardo, "Exploring the determinants of scientific data sharing: Understanding the motivation to publish research data," *Government Inf. Quart.*, vol. 30, pp. S19–S31, 2013.

[44] H. Bal *et al.*, "A medium-scale distributed system for computer science research: Infrastructure for the long term," *Computer*, vol. 49, no. 5, pp. 54–63, May 2016.

[45] D. Król *et al.*, "Workflow performance profiles: Development and analysis," in *Proc. Eur. Conf. Parallel Process.*, 2016, pp. 108–120.

[46] S. Bharathi *et al.*, "Characterization of scientific workflows," *Proc. 3rd Workshop Workflows Support Large-Scale Sci.*, 2008, pp. 1–10.

[47] A. Ilyushkin *et al.*, "Scheduling workloads of workflows with unknown task runtimes," in *Proc. 15th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2015, pp. 606–616.

[48] A. Iosup *et al.*, "DGSim: Comparing grid resource management architectures through trace-based simulation," in *Proc. Eur. Conf. Parallel Process.*, 2008, pp. 13–25.

[49] J. P. Jones and B. Nitzberg, "Scheduling for parallel supercomputing: A historical perspective of achievable utilization," in *Proc. Workshop Job Scheduling Strategies Parallel Process.*, 1999, pp. 1–16.

[50] Y. K. Kwok and I. Ahmad, "Benchmarking and comparison of the task graph scheduling algorithms," in *Proc. 1st Merged Int. Parallel Process. Symp. Parallel and Distrib. Process.*, vol. 59, pp. 531–537, 1999.

[51] D. G. Feitelson *et al.*, "Theory and practice in parallel job scheduling," in *Proc. Workshop Job Scheduling Strategies Parallel Process.*, 1997, pp. 1–34.

[52] D. G. Feitelson, "Parallel workload archive," 2007. [Online]. Available: http://www. cs. huji. ac. il/labs/parallel/workload

[53] A. Iosup *et al.*, "The grid workloads archive," *Future Gener. Comput. Syst.*, vol. 24, pp. 381–422, 2008.

[54] C. Reiss *et al.*, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proc. 3rd ACM Symp. Cloud Comput.*, 2012, pp. 1–13.

[55] Y. Chen *et al.*, "Analysis and lessons from a publicly available google cluster trace," Univ. California, Berkeley, CA, Tech. Rep. UCB/EECS-2010–95, 2010.

[56] A.K. Mishra *et al.*, "Towards characterizing cloud backend workloads: insights from google compute clusters," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 37, pp. 34–41, 2010.

[57] D. G. Feitelson *et al.*, "Experience with using the parallel workloads archive," *J. Parallel Distrib. Comput.*, vol. 74, pp. 2967–2982, 2014.

[58] A. Iosup and D. Epema, "Grid computing workloads," *IEEE Internet Comput.*, vol. 15, no. 2, pp. 19–26, Mar./Apr. 2011.

[59] S. Shen *et al.*, "Statistical characterization of business-critical workloads hosted in cloud datacenters," in *Proc. 15th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2015, pp. 465–474.

[60] D. G. Feitelson, *Workload Modeling for Computer Systems Performance Evaluation*. Cambridge, U.K.: Cambridge Univ. Press, 2015.

Laurens Versluis received the BSc and MSc degrees in computer science from the Delft University of technology, The Netherlands. Currently, he is working toward the PhD degree in the Massiving Computer Systems Group, Department of Computer Science, Faculty of Sciences, VU Amsterdam, The Netherlands. His research interests include cloud computing, distributed systems, scheduling, complex workflows, image processing, and privacy enhancing technologies.

Roland Mathá received the BSc degree in computer science and the MSc degree from the University of Innsbruck, Austria, in 2011 and 2014, respectively. Since January 2015, he is working toward the PhD degree under the guidance of prof. Radu Prodan. His research interests include Cloud simulations, workflow applications, and multi-objective optimisations.

Sacheendra Talluri received the MSc degree from the Delft University of Technology, The Netherlands. In the spring of 2018, he was a research intern at big data company Databricks, working on resource management and scheduling across the memory-storage stack.

Tim Hegeman received the BSc and MSc degrees in computer science from the Delft University of Technology, The Netherlands. He is currently working toward the PhD degree at Vrije Universiteit Amsterdam, The Netherlands under guidance of prof. Alexandru Iosup. His research interests include distributed systems, big data, and performance engineering.

Radu Prodan received the PhD degree from the Vienna University of Technology, in 2004. He was a associate professor until 2018 at the University of Innsbruck, Austria. He is currently a professor in distributed systems at the Institute of Software Technology, University of Klagenfurt. He has participated in numerous national and European projects, as is currently principal coordinator of the H2020-ICT project ARTICONF (smART socIal media eCOsystem in a blockchaiN Federated environment). He is the author of one book, more than 100 journal and conference publications, and is the recipient of two IEEE best paper awards.

Ewa Deelman received the PhD degree in computer science from Rensselaer Polytechnic Institute. She is currently a research professor with the Computer Science Department and the assistant director for the Science Automation Technologies group at the University of Southern California Information Sciences Institute. Her research focuses on distributed computing, in particular regarding how to best support complex scientific applications on a variety of computational environments, including campus clusters, grids, and clouds.

Alexandru Iosup received the PhD degree in computer science from TU Delft, The Netherlands, in 2009. He is currently a tenured full professor and University Research chair with the Vrije Universiteit Amsterdam, The Netherlands. He is also chair of the SPEC Research Cloud Group. He was awarded the yearly Netherlands Prize for Research in Computer Science (2016), the yearly Netherlands Teacher of the Year (2015), and several SPECtacular awards (2012–2017). His research interests include massivizing computer systems, that is, making computer systems combine desirable properties such as elasticity, performance, and availability, yet maintain their ability to operate efficiently in controlled ecosystems. Topics include cloud computing and big data, with applications in big science, big business, online gaming, and (upcoming) massivized education.

Boosting the Performance of SSDs via Fully Exploiting the Plane Level Parallelism

Congming Gao, Liang Shi, Kai Liu, Chun Jason Xue, Jun Yang, and Youtao Zhang, *Member, IEEE*

Abstract—Solid state drives (SSDs) are constructed with multiple level parallel organization, including channels, chips, dies, and planes. Among these parallel levels, plane level parallelism, which is the last level parallelism of SSDs, has the most strict restrictions. Only the same type of operations that access the same address in different planes can be processed in parallel. In order to maximize the access performance, several previous works have been proposed to exploit the plane level parallelism for host accesses and internal operations of SSDs. However, our preliminary studies show that the plane level parallelism is far from well utilized and should be further improved. The reason is that the strict restrictions of plane level parallelism are hard to be satisfied. In this article, a *from plane to die* parallel optimization framework is proposed to exploit the plane level parallelism through smartly satisfying the strict restrictions all the time. In order to achieve the objective, there are at least two challenges. First, due to that host access patterns are always complex, receiving multiple same-type requests to different planes at the same time is uncommon. Second, there are many internal activities, such as garbage collection (GC), which may destroy the restrictions. In order to solve above challenges, two schemes are proposed in the SSD controller: First, a die level write construction scheme is designed to make sure there are always N pages of data written by each write operation. Second, in a further step, a die level GC scheme is proposed to activate GC in the unit of all planes in the same die. Combing the die level write and die level GC, write accesses from both host write operations and GC induced valid page movements can be processed in parallel at all time. To further improve the performance of SSDs, host write operations blocked by GCs are suggested to be processed in parallel with GC induced valid page movements, bringing lesser waiting time cost of host write operations. As a result, the GC cost and average write latency can be significantly reduced. Experiment results show that the proposed framework is able to significantly improve the write performance without read performance impact.

Index Terms—SSD, parallelism, storage, scheduling, performance improvement

1 Introduction

Solid state drives (SSDs) are widely adopted in modern computer systems, ranging from embedded systems, personal computers, to large servers in data centers. SSDs have many advantages, such as shock resistance, high random access performance, and low power consumption [1]. An SSD usually consists of multiple channels with each channel having multiple chips, each chip having multiple dies, and each die having multiple planes [2], [3]. To achieve high performance, the prior studies strive to exploit the parallelism at channel/chip/die/plane levels so that multiple accesses, such as reads, writes, and erases, can be processed in different parallel units simultaneously [4], [5], [6].

- C. Gao is with the College of Computer Science, Chongqing University, Chongqing 400044, P.R. China, and also with the University of Pittsburgh, Pittsburgh, PA 15260. E-mail: albertgaocm@gmail.com.
- K. Liu is with the College of Computer Science, Chongqing University, Chongqing 400044, P.R. China. E-mail: liukai0807@gmail.com.
- J. Yang and Y. Zhang are with the University of Pittsburgh, Pittsburgh, PA 15260. E-mail: juy9@pitt.edu, zhangyt@cs.pitt.edu.
- L. Shi is with the School of Computer Science and Technology, East China Normal University, Shanghai 200241, P.R. China. E-mail: shi.liang.hk@gmail.com.
- C. Xue is with the Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong. E-mail: jasonxue@cityu.edu.hk.

Manuscript received 9 July 2019; revised 18 Feb. 2020; accepted 8 Apr. 2020. Date of publication 15 Apr. 2020; date of current version 5 May 2020. (Corresponding author: Liang Shi.) Recommended for acceptance by C. Ding. Digital Object Identifier no. 10.1109/TPDS.2020.2987894

However, the parallelism at the last level, referred to as plane level parallelism, exhibits strict restrictions – for two operations that can be issued simultaneously to two different planes, they not only need to be of the same type (i.e., read or write) but also need to have the same in-plane address (i.e., the same offset within each plane), making it challenging to explore as shown in recent studies [7], [8], [9], [10], [11], [12]. For example, to concurrently write two planes, their write points need to be aligned. Unfortunately, a host often sends uneven numbers of write requests to different planes [9] and the activities originated from SSDs (e.g., garbage collection operations) are often imbalanced across different planes [9], [13]. Such asynchronicity leads to sub-optimal exploration of plane level parallelism and prevents modern SSDs from achieving further performance improvement.

To exploit plane level parallelism, Tavakkol *et al.* proposed *TwinBlk* to write data to the different planes in a die in a round-robin fashion [11] such that concurrent writes can be issued to different planes at the same time. However, the write points from different planes may be mis-aligned due to (1) single-page write operations; (2) GC or wear leveling activities originated inside the SSD [9], [13], [14], [15], disabling the concurrent writes in these cases. To reduce GC-induced write point mis-alignment, Shahidi *et al.* proposed *ParaGC* to activate GC from all planes of the same die at the same time [9], which opportunistically exploits the plane level parallelism when all the pages at the same address of different planes are valid. *TwinBlk* can concurrently activate multiple GCs by choosing multiple blocks with the same offset in different planes. It cannot process all valid page movements in parallel when not all paired pages are valid. Superpage enabled SSDs [1], [16], [17] strip requests to all planes in a die, which increases the number of sequential writes as well as concurrent write opportunities. However, activating GC in one plane introduces mis-aligned free blocks so that subsequent requests can not be processed in parallel. In summary, a major limitation of existing studies is that they explore plane level parallelism passively, making it difficult to satisfy the access restrictions all the time. In particular, it is challenging to construct *multi-plane command* after GC and/or wear leveling mis-align the write points in different planes.

In this paper, we propose SPD, an SSD *from plane to die* parallel optimization framework, to fully exploit the plane level parallelism of SSDs for performance improvement. We summarize our contributions as follows.

- We propose SPD to treat all planes (e.g., N planes) in a die as a single unit so that a die write results in N page writes while a die read fetches N or fewer pages. Similarly, internal activities, e.g., GC, get triggered for N blocks from different planes that have the same in-plane block address. To our best knowledge, this is the first work on actively maintaining aligned write points for multiple planes in a die combining writes from both host and internal activities for all the time;
- We then propose die level write construction and die level GC schemes to fully exploit the plane level parallelism enabled by SPD. The write construction scheme is to construct write operation with N pages of data and issue them to a die at once; The die level GC scheme is to process valid page movements, aligning the write points of all planes in the same die.
- To further improve the write performance, we propose SPD+, which is designed to process host write operations blocked by die level GCs in parallel with GC induced valid page movements. Therefore, host write operations are able to be completed with lesser waiting time.
- We evaluate the proposed approach using a significantly extended SSDSim [10] and compare it to the state-of-the-arts. The experimental results show that proposed approach is able to significantly improve write performance of SSDs without read performance impact.

The rest of this paper is organized as follows: In Section 2, the background is presented. In Section 3, the problem statement is presented. In Section 4, the SPD framework is presented. In Sections 5 and 6, the experiment setup and evaluations are presented. In Section 7, related works are discussed. Finally, the work is concluded in Section 8.

2 Background

In this section, we briefly discuss the background, including SSD organization, advanced SSD commands, parallelism, and garbage collection (GC).

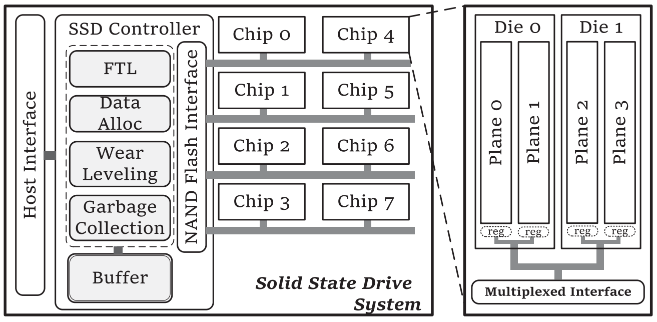


Fig. 1. The organization of SSDs.

2.1 SSD Organization

A modern SSD usually consists of multiple channels with each channel containing multiple flash chips. Within each flash chip, there are multiple dies with each die containing multiple planes. Fig. 1 illustrates the organization of a typical SSD that has 4 channels, 2 chips per channel, 2 dies per chip, and 2 planes per die. The SSD parallelism can be exploited at channel/chip/die/plane levels, which have one major focus of previous studies for performance improvement [2], [13], [18]. To manage the flash memory as well as to explore the parallelism, an SSD controller comprises several components, including flash translation layer (FTL), data allocation (DA), wear leveling (WL), garbage collection (GC).

The `FTL` is to manage the mapping between logical addresses and physical addresses. Based on the operation granularity, there are three types of mapping schemes, i.e., page mapping [4], block mapping [19], and hybrid mapping [20] [21], [22]. In this work, we assume the widely adopted page mapping as it tends to have its better performance. The `data allocation` is to determine the allocations of channel, chip, die and plane for write operations. The `wear leveling` is to distribute written data evenly to flash pages for prolonging the SSD lifetime [23], [24]. Since flash memory cannot reprogram a programmed flash page before executing an erase operation to reclaim the whole block, modern SSDs widely adopt out-of-place-update scheme for data updating. To reclaim invalid pages, `GC` is activated while the number of free pages drops below a predefined threshold.

In addition, modern SSDs widely equip a built-in Random Access Memory (RAM), referred to as the *SSD buffer*, within SSD controller for temporarily storing hot data and metadata. Since the access latency of RAM is much smaller than that of flash memory, buffer-equipped-SSDs can provide much better performance for data hit in the buffer [25], [26], [27], [28].

2.2 Parallelism and Advanced Commands

The hierarchical SSD architecture provides four level parallelism, from channel, chip, die to plane. For channel and chip level parallelism, data can be processed in different chips in parallel. The parallelism of these two levels is naturally supported by SSDs while that of the rest two levels are supported by advanced commands [9], [10], [12], [18], [29]. The die and plane level parallelism is also referred to as internal parallelism [3].

For die level parallelism, operations issuing to the same chip but different dies can be processed in parallel with *interleaving command* [9], [10]. There is no restriction on when to use the interleaving command. For the last level parallelism, plane level parallelism may be exploited to further improve performance through processing operations concurrently on different planes of the same die. Due to circuit restrictions [7], as shown in the open NAND flash interface (ONFI) standard specification [8], the plane level parallelism can be exploited when satisfying the two operation type and in-plane address restrictions of *multi-plane command*. A *multi-plane command* improves plane utilization as it operates multiple planes within the same die in parallel and only takes the time to finish one operation. However, when the restrictions can not be met, it processes different planes sequentially to the requested operation. In particular, an operation processed on one plane blocks other planes of the same die from servicing other operations.

2.3 Garbage Collection

Within flash memory, pages can not be updated in place [7]. In order to solve this issue, data is always updated out of place by programming updated data in another block and invalidating original version. Invalid pages can not be reused until they are erased. With the increasing of invalid pages' number and reduction of free pages' number, garbage collection (GC) is triggered for reclaiming invalid pages [1], [13]. The process of GC can be described as follows: First, a victim block is selected; Second, valid pages in the victim block are read and wrote to free pages in other blocks; Third, the victim block is erased. During this process, valid page movement is performed page by page, which is able to introduce significant time cost of SSD system while the incoming host requests are blocked and delayed [13].

To solve such a problem, a simple and effective approach, termed greedy GC, has been proposed and widely used to reduce the time cost of valid page movement [22], [30]. Greedy GC is designed to minimize the cost of valid page movement by selecting block with minimum number of valid pages. Thus, the total GC induced time cost can be minimized so that fewer host requests are blocked and delayed, increasing the performance of SSDs. In this work, greedy GC also is considered as a typical GC algorithm while other GC algorithms [31], [32], [33] also can be applied in proposed approach without loss of generality.

3 Problem Statement

In this section, we study the challenges in exploiting the plane level parallelism, which comes mainly from the restrictions of the *multi-plane command* [9], [18]. For clarity, we focus on write operations as they are much slower than read operations and thus have larger impact on the overall performance.

We next conduct a study on the operations to an SSD with each die consisting of two planes. Without losing generality, the non-GC operations that access the same die may be categorized to the following four cases.

`Case 1:` The operations are issued to one plane only (*Single Write*). Such write operations introduce unaligned write points across different planes;

`Case 2:` Two different types of operations are issued to the two planes of the same die. Due to the operation type restriction, the operations are not allowed to be processed in parallel;

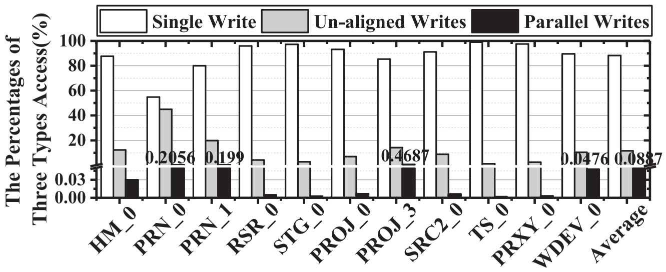


Fig. 2. The percentages of write operations in three cases.

`Case 3:` Two same type operations with unaligned in-plane addresses are issued to two planes of the die (*Unaligned Writes*). Due to the address restriction, the operations cannot be processed in parallel either;

`Case 4:` Two same type operations with aligned in-plane addresses are issued to two planes (*Parallel Writes*), which can be processed in parallel with the support of *multi-plane command*.

Among these four cases, Case 2 can not be avoided due to the circuit restriction of *multi-plane command* while mixed types of operations being issued to different planes of the same die. Then, the numbers of operations falling in Case 1, Case 3 and Case 4 are collected and reported in Fig. 2. The experiment setting details can be found in the experiment section. We have two observations from the results: (i) plane level parallelism is far from well utilized; (ii) a large percentage of write operations issued to the die are unaligned write operations, which can be exploited for performance improvement.

To solve the issue of unaligned write points across planes, a naive solution is to write data at the aligned points greedily [10]. However, if the current write points are unaligned, writing data at the aligned points lead to wasted space. For example, we assume there are two planes per die, one block per plane, and six pages per block, as shown in Fig. 3a. In Fig. 3a-(1), the current write points are unaligned. Traditionally, if two write operations, W1 and W2, are issued to the two planes in the same die, they will be processed sequentially. If they are written to the aligned pages, a free page in Plane 1 would be wasted, as shown in Fig. 3a-(2). *In this work, we strive to design a write construction scheme to align the write points in each die.*

Apart from host requests, internal SSD activities, e.g., GC, also introduce non-negligible performance impact from the unaligned write points across planes [13], [14]. Given a die with multiple planes, if one plane activates GC, the other planes cannot be accessed before this GC finishes. To solve this problem, Shahidi *et al.* proposed to activate GCs in all planes at a time so that GC induced time cost can be overlapped [9]. To avoid significant parallel GC induced write amplification, ParaGC proposed to select a block containing most invalid pages in a plane first. Then, if the number of invalid pages in the paired block resided in the paired plane is large too, these two blocks can be reclaimed by GCs simultaneously. Otherwise, only one block is processed by GC. However, such a solution faces two issues: `First`, since the

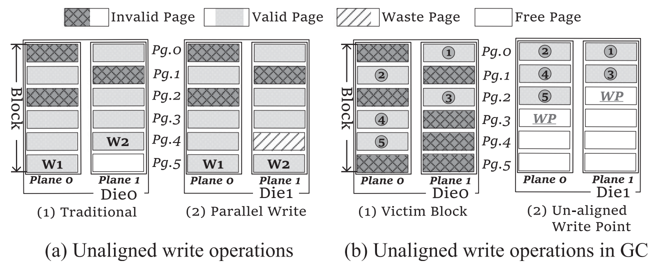


Fig. 3. The problems of unaligned write operations.

number of valid pages in paired blocks are different, ParaGC may lead to unaligned write points across different planes after valid page movements. For example, in Fig. 3b, after moving valid pages in each plane in Fig. 3b-(1), new write points (*WP* in the figure) become unaligned, as shown in Fig. 3b-(2). Second, if only one block is occupied by GC, write points will be unaligned while this block is erased and switched as a free block and its paired block still has not been reclaimed. That is, to maintain aligned write points at all time, we need to construct *multi-plane* oriented writes for both of host requests and GC induced operations.

4 SPD: From Plane to Die Parallelism Exploration

4.1 Overview

To maximize plane level parallelism, the access addresses of writes on all planes in the same die should be aligned at all time. In this work, we propose SPD, an SSD *from plane to die* framework, to exploit the plane level parallelism for performance improvement by smartly maintaining aligned write points across multi-planes in each die at all time.

Basically, SPD takes the following strategies to achieve the objective, as shown in Fig. 4. The basic SPD design adds three new components — a die level write construction, a die level GC and a combination scheme. The die level write construction is designed to maintain aligned write points for host writes. The die level GC is designed to maintain aligned write points for GC induced page movements. Note that for other activities, such as WL, they also can adopt the same design principle of GC. For simplicity, only GC is taken as an example in this paper due to its non-negligible performance impact on SSDs. The combination scheme is proposed to process write operations and GC related valid page movements in parallel so that write operations from host system can be processed with less waiting time.

For die level write construction, SPD exploits the SSD buffer to choose N dirty pages and writes them back to one die simultaneously. This helps to convert one die access to N page writes at the aligned in-plane address. This is referred to as `Die-Write`. Similarly, the read access to the die is referred to as `Die-Read`. Note that `Die-Read` only needs to read required number of pages, which does not introduce any read amplification. For die level GC, it is activated at the multiple planes of the same die at the same time. During the process of die level GC, all writes induced from the valid page movements also is processed in the unit of N page writes to maintain the aligned write points. After moving all valid pages to new free pages, erase operations are executed in parallel to reclaim victim blocks with same

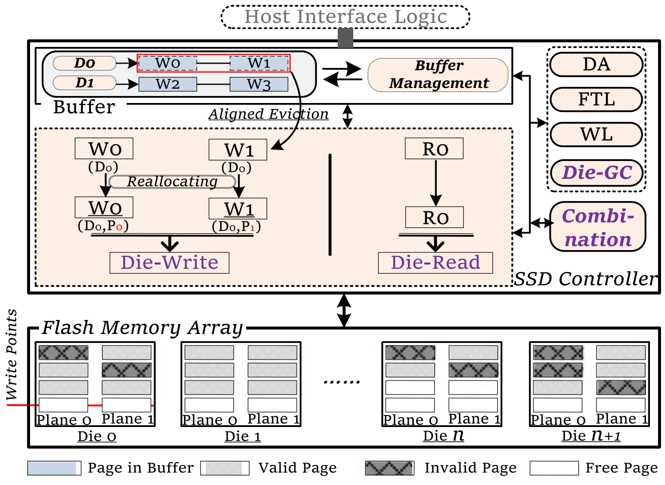


Fig. 4. The Overview of the *from plane to die* framework.

in-plane address. This is referred to as `Die-GC`. N is set to two in the following discussion while we evaluate different N values in the experiments. Since GC process is time-consuming, host write operations blocked by `Die-GC` (called as access conflict in this paper) are able to be significantly delayed, causing write performance degradation. To minimize `Die-GC` induced access conflict, we combine `Die-Write` and `Die-GC` together for processing host write operations and GC induced valid page movements in parallel. Therefore, host write operations can be processed with lesser waiting time, bringing better write performance. We will elaborate the details of these three components in following sections.

4.2 Die Level Write Construction

Given that *multi-plane commands* would be disabled if the in-plane addresses are mis-aligned, the basic idea of die level write construction is to maintain aligned write points all the time by write the same amount of data synchronously to all planes in the same die. That is, (1) the amount of data issued to a die should be a multiple of N pages, assuming there are N planes in a die; and (2) the starting locations of data should be aligned for all the planes in the same die. With this scheme, whenever there are multiple write operations issued to a die, they can be processed in parallel.

SPD exploits SSD buffer to assist die level write construction. An SSD buffer evicts a multiple of N dirty pages from one die at a time such that these pages can be written using `Die-Write`. For data allocation, we adopt a round-robin plane allocation scheme within a die [11], which evenly distributes N dirty pages to different planes at each cycle. The data allocation at higher levels can either be static or dynamic, as discussed in Section 2.1. In the following discussion, we assume static allocation at the channel, chip, and die levels, which is simple and has been widely equipped in real SSD devices.

4.2.1 Buffer Supported Die-Write

Fig. 5 illustrates how the SSD buffer assisted `Die-Write` works. Fig. 5a shows how the SSD buffer is organized. It maintains a die queue that keeps a list of dirty pages for each die in the system. The pages in each list are linked together

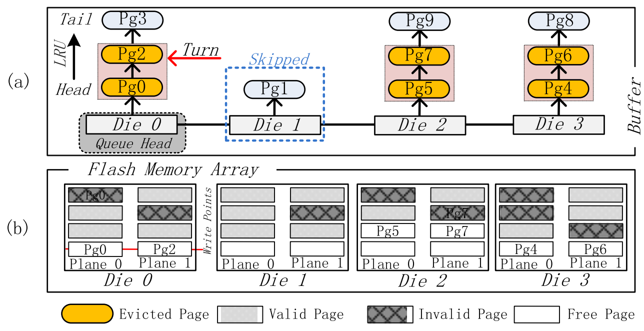


Fig. 5. Organization of write buffer and the die level write construction.

using LRU algorithm. The data evicted from the buffer are written to their corresponding dies. To balance the number of writes sent to different dies, SPD adopts round-robin to choose the next die from which its LRU pages are evicted.

For the example, in Fig. 5b, the SSD has four dies, each die has two planes, and the current turn is Die 0. When the SSD buffer is full and there is a host requirement for inserting five dirty pages to the buffer, SPD chooses the victim dies with at least two dirty pages (i.e., two is the number of planes in a die) and evicts the dirty pages from each selected die. In the example, it first chooses Die 0 and then skips Die 1 as the latter does not have enough dirty pages. It continuously chooses Die 2 and Die 3 and then evicts two pages from Die 0, 2, and 3, respectively.

From this example, the write points of all planes are effectively aligned. The proposed scheme may evict one more dirty page than the number of dirty pages from the host. Since one `Die-Write` takes the same amount of time as one page write, the scheme is able to speed up the storage access if there exist several dirty pages evicted to the same die. But if only one dirty page from the host, evicting one more dirty page can align the write points without introducing additional time cost. In addition, since all `Die-Writes` operations can be scheduled in parallel by leveraging the parallel architecture of SSDs, SPD avoids the access conflicts on the same die [3], [18]. Due to that we always evict the pages at LRU positions, the write amplification also can be minimized.

Since the addresses of requested data are fixed, die level read operations cannot be constructed the same way at that for `Die-Write`. In this work, `Die-Read` only read the requested data, i.e., if there exist read operations with aligned access locations, they can be issued to the die in parallel; otherwise, only single page read gets processed next. The goal of `Die-Read` is to maximize the number of *multi-plane command* supported read operations without introducing read amplification.

4.2.2 *Implementation and Analysis*

To assist die level write constructions, SPD enhances the SSD buffer management to expose more parallel processing opportunities. Different from traditional buffer management scheme, SPD needs to evict a multiple of N dirty pages from one die queue. In this work, the N pages of dirty data at the head of LRU are selected for eviction. SPD does not require an extra built-in buffer and thus does not introduce extra space demand. However, SPD requires a minimal of $M * N * Size\_of\_Page$-byte buffer for smooth buffer management where M is the number of dies in an SSD, and each die has N planes. For example, for a 512 GB SSD, with 32 dies, two planes in each die and 4 KB page size, the minimal size of buffer should be 256 KB, which can be satisfied by most existing SSD products.

In addition, to maintain such die queues, more space overhead is introduced. First, total 32 queue heads are required and each queue head points to the the LPN of a page. Each LPN requires 4 bytes and the total space cost of all queue heads is 128 bytes. Second, another pointer is set to locate current active die queue, where pages are going to be evicted in the next eviction innovation. Since there at most are 32 die queues, this pointers needs 5 bits space cost. Third, each die queue should set one counter to record the number of pages linked in this queue. In worst case, all pages in the buffer are linked in one die queue. That is, each counter requires 17 bits and total space cost of counters is 68 bytes. In summary, less than 197 bytes are needed to maintain these die queues.

Another issue that SPD needs to consider is the power interruption induced data loss, which is often mitigated by integrating a super capacitor [34], [35], [36], [37], [38].

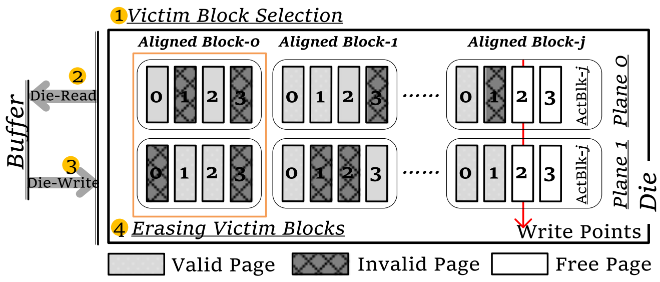


Fig. 6. The Process of Die-GC.

4.3 Die Level GC

A GC process includes three steps: victim block selection [1], [14]; valid page movement; and victim block erase. The dominate cost of a GC comes from valid page movement [13]. The design goal of `Die-GC` is to speed up the GC process with minimal GC cost. For this purpose, SPD activates GC at all planes in the same die at the same time with carefully selected victim blocks. By adopting `Die-Write` instead of sequential page writes, SPD improves reclaim effectiveness by reducing the most timing cost. We elaborate the details as follows.

4.3.1 *GC Process*

Fig. 6 shows an example for `Die-GC`. Within each die, N stripped blocks from N planes — one from each plane and all the selected blocks share the same in-plane address, are grouped together as aligned block, which is set as the minimal granularity of `Die-GC`. That is, different to the traditional GC process, `Die-GC` should be modified while aligned block is taken as the minimal granularity. Totally, there exist four steps: First, we adopts the greedy based victim block selection [1], [13], where the aligned block with maximal number of invalid pages is selected. With this scheme, the time cost of valid page movement will be minimized. Such scheme can be realized without any modification. Since traditional GC process scans the states (valid/invalid/free) of all pages in a block and finds a victim block which contains maximal number of invalid pages, proposed Die-GC is realized by summing the number of valid pages of blocks in each aligned block. Based on the summed number, victim aligned block is selected. Second, SPD uses Die-Read (in Section 4.2.1) to read the valid pages to the SSD buffer, where N page slots are required to store valid pages from victim blocks. Third, after reading N pages of valid data, SPD groups the N pages of data to construct a Die-Write operation and then writes the valid data back to the die. Finally, when all the valid pages are written back, the N aligned blocks can be erased in parallel. Given SPD reclaims N blocks from one GC invocation, the GC gets triggered less frequently than that of the traditional one. However, since each GC erases and reclaims two blocks after one invocation, the total number of erase operations during the whole lifetime of SSDs can be increased while the frequency of triggering GC can not be significantly reduced, causing lifetime degradation of SSDs. In the experiment, the impact on lifetime is going to be evaluated and presented.

For the example shown in Fig. 6, let us assume the two aligned blocks 0 from two planes are selected as the victim blocks. According to Die-GC, the valid pages in these two blocks are read and written with Die-Read and Die-Write, respectively.

Step1: Read page 0 from plane 0 and page 1 from plane 1 to the SSD buffer. Since they are not aligned, they are read sequentially.

Step2: Group the two valid pages together to construct a Die-Write operation and written them back to the current aligned write point of both planes at block j. The current write points are marked using red arrows in the figure.

Step3: Then, read page 2 from plane 0 and plane 1 to the SSD buffer. These two pages are read in parallel as they have aligned addresses.

Step4: Repeat step (2) for the last two valid pages.

Step5: Then, erase the two victim blocks in parallel. From the above discussion, Die-GC significantly reduces GC cost because it maintains aligned write points in the die such that many strip reads and writes can operate in parallel.

An exception for the above scheme happens when the total number of valid pages in the victim aligned blocks is odd. In this case, the last Die-Write cannot be constructed due to the lack of one more valid page, causing the write points of different planes misaligned after GC while the last Die-Write is carried out with only one valid page inside. To address this issue, the last Die-Write operation is constructed by the remaining valid page in victim block and one dirty page in the write buffer (as discussed in Section 4.2).

4.3.2 *Implementation, Analysis, and Discussion*

We next elaborate the implementation overhead of SPD. We identify the construction of Die-Write as the most critical component in SPD. Given that SPD transfers more data to the write buffer in the controller, it demands larger data storage. Considering the worst that all dies are activated with the die level GC, each die needs at least N pages in the write buffer. For a typical SSD setting as presented in Section 4.2, the required buffer size for Die-GC is 256 KB for a 512 GB two-plane SSD and 512 KB for a 512 GB four-plane SSD at least. In summary, the storage requirement is modest for modern SSDs.

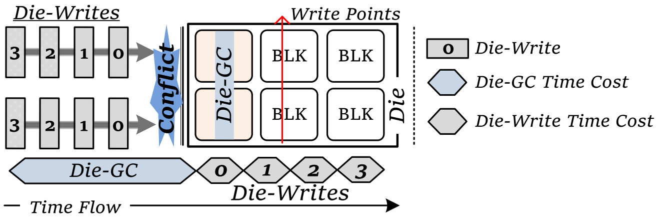


Fig. 7. Access Conflict between Die-Writes and Die-GC.

4.4 Combining Die Level Writes and GC

Both of Die-Write and Die-GC can be used to improve the write performance by fully exploiting the plane level parallelism. However, if there is a die being occupied by Die-GC, constructed Die-Writes towards this die will be delayed and wait for the completion of Die-GC. As shown in Fig. 7, there are four Die-Writes accessing current die, where a Die-GC is being processed in two blocks. In this case, these four Die-Writes have to be delayed until the completion of Die-GC. The bottom of Fig. 7 shows the time flow of Die-GC and Die-Writes. All these four Die-Writes suffer from a long waiting time caused by Die-GC. Such a GC induced conflict between Die-GC and Die-Write is regarded as access conflict in this work. In order to minimize the impact from access conflict between Die-Write and Die-GC, the most straightforward method is to process Die-Write with higher priority and postpone Die-GC. However, such a method may make Die-GC induced writes starve. Therefore, to avoid the starvation of Die-GC induced writes, we propose a combination scheme, which aims to reduce the waiting time of Die-Write without destroying the aligned write points.

4.4.1 *Combination Scheme*

As shown in top part of Fig. 8, there are eight valid pages in victim blocks (light blue boxes), and eight dirty page (gray boxes) generated write operations, which can be processed in parallel by constructing Die-GC and Die-Writes, respectively. Totally there are three steps (Step 0 to Step 2), which are processed in sequence order. For Die-Writes, which arrive during the process of Die-GC, they have to be delayed before the completion of Die-GC, causing write performance degradation.

To minimize the waiting time of Die-Writes, Die-GC is divided into two parts, including valid page movements and erase operations, where valid page movements are processed by constructing new Die-Writes. To solve this problem, in the bottom of Fig. 8, valid pages in victim blocks and dirty pages from buffer are combined to construct new Die-Writes. In this case, we assume there are three and five valid pages in two victim blocks, respectively. The whole combination scheme can be divided into three steps (Step 3 to Step 5 in Fig. 8). For Step 3, one victim block is selected to execute valid page movements while the paired

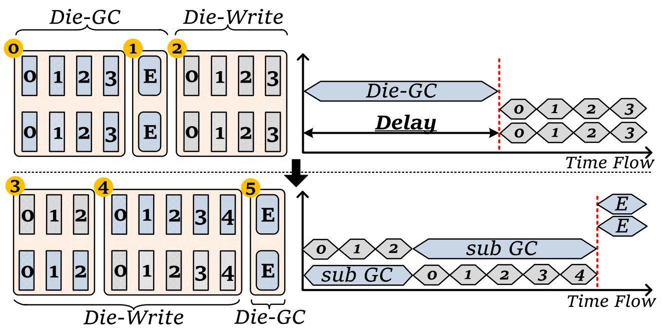


Fig. 8. Combining Die-Writes and Die-GC for Minimizing the Impact from Access Conflict (light blue box indicates GC related operations and gray box indicates dirty page generated write operations).

block is used to service write operations. Totally, three write operations from the buffer can be written in parallel with valid page movements of GC as three `Die-Writes`. Similarly, in Step 4, valid page movements in another victim block are realized while additional five dirty pages from the buffer are being wrote to the paired block. Since write points of all planes in the same die are aligned, the new `Die-Writes` are constructed and processed in parallel via accessing aligned write points. Lastly, in Step 5, two erase operations are executed in parallel as a `Die-GC` that does not contain valid page movements. For the time cost of this combination scheme, although the total time cost is the same as the original scheduling case (the top part in Fig. 8), the time cost of write operations from host system can be reduced while they are being processed in parallel with GC process.

4.4.2 *Pseudocode Analysis*

To explain more details, a combination scheme based algorithm is presented in Algorithm 1. Initially, we assume there are two planes in a die. Before evicting dirty pages to a corresponding die, the following algorithm is used to check whether these evicted dirty pages can be processed in parallel with valid page movements in GCs. If there is a `Die-GC` being processed in a die and some dirty pages are going to be evicted from buffer (Line 1), the combination scheme is activated. Otherwise, proposed `Die-GC` and `Die-Write` are processed as presented in above sections (Line 11-14). For the combination scheme, the numbers of valid pages and evicted pages should be larger than 0 (Line 2, 6) so that at least one valid page in victim block can be read out and constructed as a new `Die-Write` with an evicted dirty page from the buffer (Line 3, 7). After that, this new `Die-Write` containing valid page and dirty page is written back to the aligned write points (Line 4, 8). This process is repeatedly executed until all valid pages have been moved out or there is no more dirty pages being evicted from buffer. After that, `Die-GC` is resumed (Line 10). If there still exist some valid pages in victim blocks, paired valid pages are read out and written back in parallel based on the design of `Die-GC`. Otherwise, two erase operations are processed in parallel, such as the Step 5 in Fig. 8.

4.4.3 *Implementation and Analysis*

To implement proposed combination scheme, the `Die-GC` required buffer space is used to maintain one dirty page in the buffer, and then another valid page from victim block is transferred into this buffer space (two page space is required while a die contains two planes), where two pages can be constructed as a `Die-Writes`. Based on the implementation of `Die-Write` and `Die-GC`, proposed combination scheme can be realized without introducing additional implementation overhead.

Algorithm 1. Optimizing Access Conflict between Die-GC and Die-Writes

Input:
Assume that there are two victim blocks in a die, *Blk_0* and *Blk_1*;
Die_GC: indicates a die is occupied by Die-GC;
Die_Write: indicates a write operation distributed to a die with Die-GC;
Blk1_VP and *Blk2_VP*: indicates the number of valid pages in two blocks.
Output:

```
1:  if $Die\_GC \neq NULL$ and $Die\_Write \neq NULL$ then
2:    while $Blk0\_VP \neq 0$ and $Die\_Write \neq NULL$ do
3:      $DieWrite\_Generation(Valid\ Page, Dirty\ Page)$;
4:      $Processing\_New\_DieWrite$;
5:    end while
6:    while $Blk1\_VP \neq 0$ and $Die\_Write \neq NULL$ do
7:      $DieWrite\_Generation(Valid\ Page, Dirty\ Page)$;
8:      $Processing\_New\_DieWrite$;
9:    end while
10:   $Processing\_DieGC$;
11: else if $Die\_GC \neq NULL$ then
12:   $Processing\_DieGC$;
13: else if $Die\_Write \neq NULL$ then
14:   $Processing\_DieWrite$;
15: end if
```

5 Experiment Setup

5.1 Simulated SSD Devices

Due to that the proposed scheme needs firmware support of SSDs, in this work, we use a popular trace driven simulator, SSDsim [10], to evaluate the effectiveness of the proposed framework. In order to simulate a state-of-the-art SSD, SSDsim is significantly extended based on ONFI [8]. During the evaluation, a 512 GB SSD is simulated, and page mapping and greedy based GC scheme are adopted [3], [10]. The threshold value for GC activation is set to 7 percent [9]. To triggering GC process, SSD is warmed up by filling SSD with valid and invalid data ahead. The warming up process contains two steps: first, each plane of the SSD is randomly filled with data from 93 to 95 percent to trigger GC immediately, of which 80 percent are valid; second, the evaluated workload is pre-processed in the SSD to validate read data [13]. The over-provisioning ratio is set to 25 percent, which complies with the setting in previous work [9]. For the data allocation scheme, the most widely used Channel-Chip-Die-Plane scheme is adopted. The experiment settings represent an aged state-of-the-art SSD. Other details are presented in Table 1.

During the evaluation, a DRAM buffer is configured in the SSD. We set the buffer size to be 1‰ of the footprint of the evaluated workload [25], [40], which helps to prevent setting a large buffer from generating biased results in evaluation.

TABLE 1
Parameters of the Simulated SSD [9], [39]

SSD Configuration	512 GB;16 Channels; 8 Chips/Channel; 1 Die/Chip; 2 Planes/Die;2048 Blocks/Plane; 256 Pages/Block; 4 KB Page;
Timing Parameters	0.075 ms for page read; 1.5 ms for page write; 3.8 ms for block erase; 25 ns for byte transfer.

TABLE 2
The Characteristics of Evaluated Workloads

Workloads	W/R Ratio§	FP§	R_V§	W_V§	R_S§	W_S§
HM_0	67.9%	1.35	6.9	15.2	11.2	11.6
PRN_0	93.7%	2.93	3.0	20.5	24.8	11.6
PRN_1	32.1%	5.16	31.4	10.9	24.2	11.4
RSR_0	90.7%	0.31	1.8	14.6	15.0	12.6
STG_0	76.9%	0.28	7.4	9.3	33.6	12.6
PROJ_0	82.9%	1.58	7.2	56.5	21.9	35.7
PROJ_3	4.89%	1.86	21.6	2.8	11.9	29.9
SRC2_0	88.6%	0.52	1.9	13.6	12.2	11.0
TS_0	82.6%	0.57	4.9	15.9	17.5	11.8
PRXY_0	97.06%	0.17	0.27	5.8	9.6	6.2
WDEV_0	79.9%	0.34	3.2	9.2	16.5	12.1

§ *W/R Ratio: Write and Read Requests Ratio.*
FP: FootPrint (GB).
R_V/W_V: Read/Write Data Volume (GB).
R_S/W_S: Average Read/Write Request Size (KB).

The default data organization of die lists in the buffer is designed based on the scheme of the Element-Level Parallelism Optimization (EPO) [41]. EPO evicts dirty pages from buffer based on its die location so that the utilization of die level parallelism can be maximized. The data are organized in LRU for each die list of the buffer.

5.2 Evaluated Workloads

The workloads studied in this work include a subset of MSR Cambridge Workloads from servers [42]. These workloads are widely used in previous works for studying SSD performance [9], [14], [18], [43]. The characteristics of workloads are presented in Table 2. Each workload is characterized by three metrics: W/R *Ratio*, *FP*, *R_V*, *W_V*, *R_S* and *W_S*. W/R *Ratio* represents the write and read operation ratios, *FP* is the footprints of each workload, *R_V* is the total amount of read data, *W_V* represents the total amount of written data, *R_S* represents the average size of read requests, and *W_S* is the average size of write requests.

5.3 Evaluated Schemes:

Seven schemes are implemented to show the effectiveness of SPD.

Baseline-D. This scheme is implemented to represent the traditional SSD design [10]. The buffer management of Baseline-D adopts EPO to exploit die level parallelism through adding dirty pages to different die lists based on their die locations [41]. With this organization, dirty data evicted from write buffer can be distributed to different dies so that die level parallelism can be exploited;

Baseline-P. This scheme is similar to Baseline-D. The difference is that Baseline-P evicts dirty data based on their plane locations to further exploit plane level parallelism. In this case, dirty pages accessing different planes within the same die are evicted at a time. Baseline-P evenly distributes dirty pages to different planes to better exploit plane level parallelism, which is similar to the previous studies [18], [44];

TwinBlk. This scheme is designed based on the work proposed by Tavakkol *et al.* [11], which aims to align write points of all planes in a die via round-robin policy. In this case, several host requests can be processed in parallel when write points are aligned. During GC process, the adopted round-robin policy is designed to align write points of active blocks in victim blocks as well, aiming to move valid pages with the support of *multi-plane command*;

SuperPage. This scheme is implemented based on [17], [45], which groups pages into one super page that is set as the smallest access granularity of flash memory. Such a large-granularity accessing approach can fully exploit plane level parallelism by writing pages to all planes in a die at any time. Differing from our proposed work, if there exist an update operation, the paired data in other planes should be read out first if they are valid. Later, the read data and updated data are constructed as one new super page, and then are written to the die with the support of *multi-plane command*.

ParaGC. This scheme is designed by Shahidi *et al.* [9], which aims to align valid page movement during GC to minimize the GC cost. Differing from TwinBlk, ParaGC aligns write points of active blocks through sequentially moving valid pages to one active block until write points of all planes are aligned. After that, with cache assistance, all valid pages can be written back to active blocks with the support of *multi-plane command*;

SPD. This is the proposed framework, which includes `Die-Write` and `Die-GC`.

SPD+. This is the proposed framework, which includes `Die-Write`, `Die-GC` and the combination scheme.

6 Experiment Results and Analysis

In this section, basic SPD is evaluated with two scenarios based on whether GC is triggered. For the first scenario without triggering GC, it is evaluated to show the advantages of the proposed `Die-Write` scheme. For the second scenario with triggering GC, it is evaluated to show the effectiveness of SPD, including `Die-Write` and `Die-GC`. In addition, the `Die-GC` is also evaluated in term of its cost and lifetime impact. Then, proposed SPD+ is evaluated and compared with basic SPD scheme to identify its effectiveness. Since SPD+ is designed to solve the access conflict between `Die-Write` and `Die-GC`, there is only one scenario with triggering GC. Finally, the impact of different buffer sizes and results on SSD with 4 planes per die are presented.

6.1 Experiment Results Without GC

1) Write Latency Evaluation: Fig. 9 shows the results of write latency for the six schemes. Note that, since ParaGC is designed to optimize GC process, the results of ParaGC in this part are same to that of Baseline-D. The results show that SPD achieves write latency reduction for all evaluated workloads. For example, for HM_0, PRN_0, PROJ_3, SRC2_0 and PRXY_0, the write latency is reduced by more than 15 percent

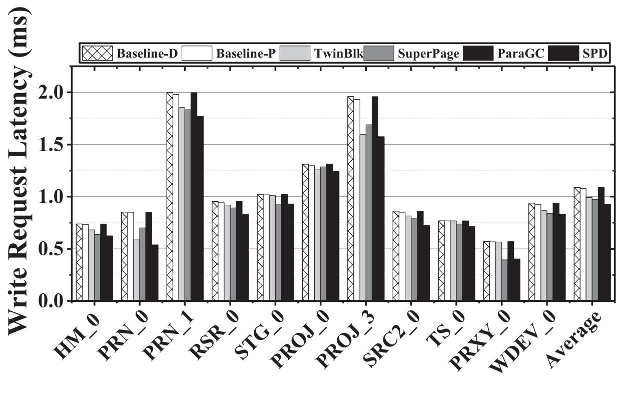


Fig. 9. Write latency reduction.

compared with Baseline-D. These results show that deploying `Die-Write` to maintain aligned write points for the multiple planes in a die is important in improving the access performance. In Fig. 10, we collected the percentages of write operations processed by *multi-plane command*. The results show that the proposed `Die-Write` is able to maintain aligned write points for all write operations. However, this is not a promise for the other schemes.

To obtain more details, we compare SPD with other three schemes, Baseline-P, TwinBlk and SuperPage. Two observations can be concluded from the results: *First*, compared with these three schemes, SPD achieves the best write performance. Baseline-P is proposed to distribute the same type requests to all planes evenly. However, the address restriction is not taken into consideration. As a result, Baseline-P only achieves little write latency reduction, which is only up to 1.4 percent. TwinBlk aims to align write points of all planes in the same die as well. However, the write points still may be unaligned due to the unaligned accesses on planes of the same die. On average, TwinBlk achieves 7.8 percent write latency reduction compared with Baseline-D. As shown in Fig. 10, the percentages of write operations processed by *multi-plane command* for Baseline-P is similar to that of Baseline-D. For TwinBlk, the percentage is largely increased compared with Baseline-D. SuperPage also can fully exploit plane level parallelism at any time by constructing super page on a die. However, such a super page would introduce write amplification and cause write traffic. Before constructing an update operation related super page, read operations are required to read out all valid pages resided in original super page. Therefore, compared with

TABLE 3
Read Latency Results Without GC

	Baseline-D	Baseline-P	TwinBlk	SuperPage	ParaGC	SPD
Reduction	0	0.049%	0.011%	0.304%	0%	0.096%

Baseline-D, although SuperPage can achieve 10.4 percent write latency reduction, SPD still outperforms SuperPage at the performance improvement.

Second, for several workloads, TwiBlk only achieves similar performance improvement to that of Baseline-D, such as RSR_0, STG_0, TS_0 and PRXY_0. This can be explained from the results in Fig. 10, where the percentage of write operations supported by *multi-plane command* is limited. The reason is that TwinBlk cannot guarantee aligned write points for all planes all the time.

For read latency, the average read latency improvement compared with Baseline-D is presented in Table 3. The results show that read latency is similar among the six schemes. The key reasons are from two aspects: first, read requests of all evaluated schemes are processed with highest priority [3], [31], [46], [47]; second, `Die-Read` is designed to only read requested pages, which are barely located in the same in-plane addresses. In conclusion, the proposed `Die-Read` is same to that of normal read operations without introducing read amplification. Note that, in SuperPage, read operations always are executed with the support of *multi-plane command*, but only required data is transferred out for time cost saving.

2) Plane Utilization: Plane Utilization is defined to present the average number of planes being occupied in parallel. In order to obtain plane utilization, the number of planes being accessed is counted when each buffer eviction process is completed. Fig. 11 shows the plane utilization (Bars) and the maximal number of planes being accessed in parallel (Dots +Line) for the six schemes. The results have a matching pattern with the write performance improvement in Fig. 9. SPD can significantly increase the plane utilization through doubling the number of parallel planes with satisfying the restrictions of *multi-plane command*. On average, the plane utilization is increased by 34.5 percent compared with Baseline-D. For the maximal number of planes accessed in parallel, all planes of the SSD can be accessed in parallel for most workloads. Similarly, SuperPage also can achieve the maximal plane level parallelism, whose plane utilization is averagely increased by 32.9 percent compared with Baseline-D. However, for Baseline-D, Baseline-P and TwinBlk, there still

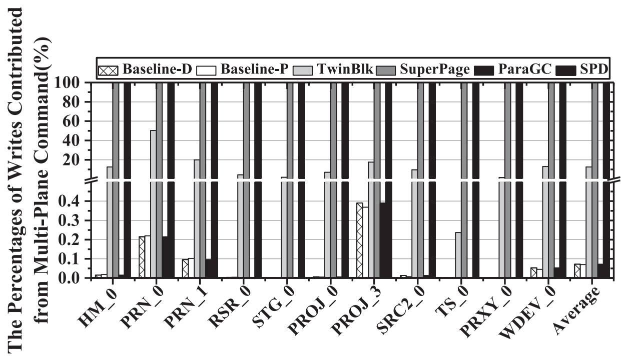


Fig. 10. Percentages of write operations processed by *multi-plane command*.

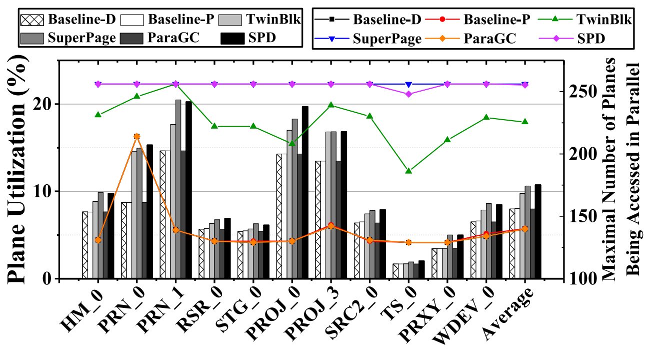


Fig. 11. The plane utilization and maximal number of planes being accessed in parallel.

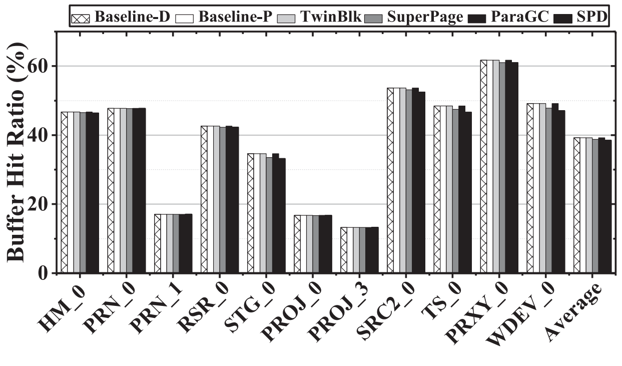


Fig. 12. The buffer hit ratios of evaluated schemes.

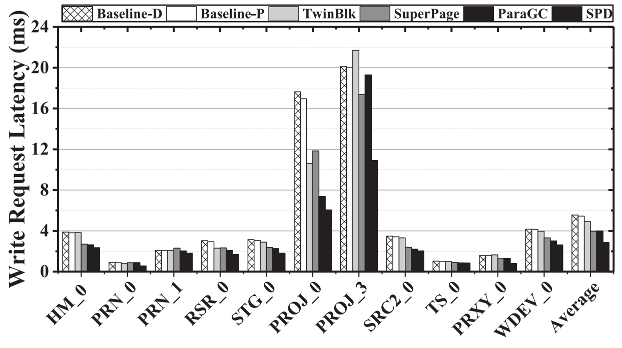


Fig. 13. The write latencies of evaluated schemes.

exists a large gap compared with SPD. In conclusion, `Die-Write` is not only able to increase plane utilization, but also can make a full use of all planes of the SSD.

3) Buffer Hit Ratio: Differently from previous work, `Die-Write` may need to evict more data from the buffer to align the write points. In this case, it may have impact to the hit ratio of buffer. Fig. 12 presents the results of buffer hit ratios for the six schemes. The results show that SPD has little impact to the hit ratio of buffer. The average buffer hit ratio is reduced by only 1.92 percent, which is negligible. The reason for the slight reduction is that `Die-Write` is designed with following principles: first, it always only need to evict one more dirty page, which is critical in aligning write points; second, the buffer is designed to only evict the cold dirty data from the LRU position.

6.2 Experiment Results With GC

Fig. 13 shows the results of write latency with GC triggered. The results show that SPD is able to significantly reduce the write latency for all workloads. The write latency is reduced by 48.61, 47.65, 42.05, 28.19, and 28.58 percent compared with Baseline-D, Baseline-P, TwinBlk, SuperPage, and ParaGC, on average. The significant improvement comes from three aspects: *First*, SPD constructs aligned write access to reduce write latency, which has been verified in Section 6.1. *Second*, the GC cost is further reduced through moving all valid pages with the support of `Die-Write`. *Third*, total GC count is noticeably reduced by reclaiming two planes at once time, and write amplification is avoided compared with SuperPage.

To understand more details, the total GC costs are presented in Fig. 14. The results show that first, TwinBlk generally has much higher cost than SuperPage and ParaGC. On average, compared with Baseline-D, total GC costs of

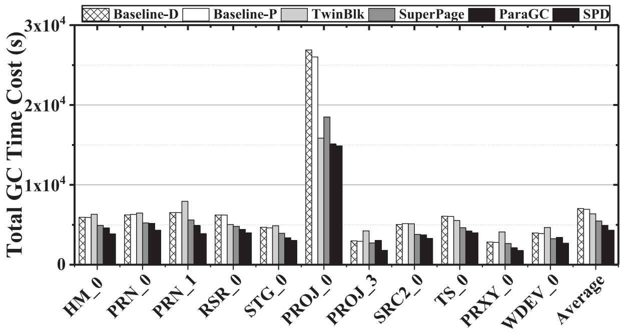


Fig. 14. Total GC cost of evaluated schemes.

TABLE 4
Read Latency Results With GC

	Baseline-D	Baseline-P	TwinBlk	SuperPage	ParaGC	SPD
Reduction	0	0.052%	-0.042%	4.173%	1.144%	1.203%

SuperPage and ParaGC are reduced by 22.4 and 30.8 percent while TwinBlk only reduces the total GC cost by 6.9 percent. For SuperPage, it reclaims two blocks at once time to reduce GC cost, but the total GC count can be increased when update operation would read other valid pages for constructing a new super page, consuming free space more rapidly. More details about GC cost is presented in Section 6.3. For ParaGC, it activates GCs in paired planes only when the number of free pages in the other plane is smaller than 7 percent. In this case, it can avoid introducing high GC cost while moving valid pages. In addition, ParaGC proposed to align write points during the process of valid page movement so that valid pages in the same position of paired planes can be read and written in parallel. However, for TwinBlk, it activates paired GCs without considering the number of valid pages in the paired planes. In this case, more valid pages from paired planes may be moved during GC process. In addition, TwinBlk adopted round-robin policy. If current write points are not aligned, valid pages having same position in different planes still can not be read and written in parallel. Therefore, for some workloads, the total GC cost of TwinBlk is larger than Baseline-D. Second, even though SPD also activates GC at the all planes at the same time, it is proposed to regard the whole die as the smallest access unit and all the write operations during GC are processed via `Die-Write`. Moreover, parallel access in SPD is constructed without introducing write amplification, avoiding triggering more GCs. As a result, the total GC cost is reduced by 36.4 percent, on average. In conclusion, SPD achieves the best write performance compared with all other related works.

For read latency, results of read latency improvement with considering GC are presented in Table 4. Similarly, the read latency is similar among each scheme. The reason also comes from the highest priority of read request, which can be processed without delay [31], [46]. The results show that SPD has no impact to read access with significant write performance improvement.

6.3 GC Evaluation

In this part, `Die-GC` is evaluated. First, the average GC cost and the number of triggered GC in different schemes are

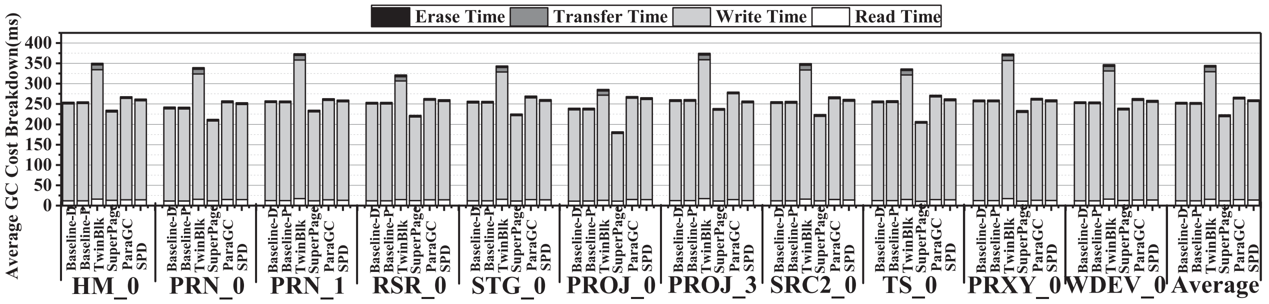


Fig. 15. Average GC cost breakdown of evaluated schemes.

evaluated. Second, the number of erase operations induced by GC is collected to show its impact on the lifetime of SSDs.

1) Average GC Cost: Average GC costs are collected in Fig. 15. In the Figure, the average GC cost is broken into four parts: read cost, write cost, transfer cost and erase cost. Read cost is the cost in reading valid pages from the victim block; write cost is the cost in writing the valid data to free pages; transfer cost is the cost in transferring the valid data among planes or between controller and chips; and erase cost is the time cost in erasing the victim block. The results show that the write cost takes the dominate part of the total cost [18]. This is because write latency of flash memory is several times of read latency. In addition, there are always a large number of valid page movement during GC. There are three observations from the results: *First*, SPD has the less GC cost compared with TwinBlk and ParaGC. Clearly, the reduced GC cost is from the `Die-Write` used in `Die-GC`, which is triggered to write dirty pages back to the multiple planes in parallel. For TwinBlk, it also trigged GC in the paired planes. However, TwinBlk adopted round-robin policy for write operations among planes, which is not able to always align the write points. In this case, many valid pages written back may be processed sequentially. *Second*, the GC cost of SPD is similar to that of Baseline-D and Baseline-P. As presented in the technique part, `Die-GC` is designed to reclaim several blocks in one GC. Several block reclaiming costs are similar with single block reclaiming cost in Baseline-D and Baseline-P due to that we carefully select victim blocks among planes as a single unit and use `Die-Write` to speed up the process. *Third*, SuperPage can achieve the minimal GC cost compared with other schemes. The reason is that, SuperPage can aggressively invalidate pages in a block, reducing the total number of valid pages in a block. This is because each update operation can invalidate all paired pages for maintaining consistent states (valid or invalid) of all pages in a super page.

2) GC Count: Fig. 16 shows the total number of triggered GCs during runtime. We can find that `Die-GC` highly reduces the number of GCs. Therefore, the frequency of triggering GC is reduced. The results show that GC count is reduced in the range of 32.9 to 50.1 percent, compared with Baseline-D. As a result, the total GC cost during whole runtime can be highly reduced as well so that the performance of SSDs can be improved. For related works, the number of triggered GCs in Baseline-P is similar to Baseline-D. Both TwinBlk and ParaGC can reduce the number of triggered GCs as well. This is because that TwinBlk and ParaGC erase more blocks in each GC process as well. But for TwinBlk, it selects victim blocks inefficiently so that its GC counts are slightly higher in most cases. For a exception, PROJ_0, since SPD may slightly increase write operations, the total triggered GC count of SPD may be slightly increased. For SuperPage, since it can cause significant write amplification, total number of GCs is increased as well.

3) GC Induced Erases: Fig. 17 shows the number of erase operations for the six schemes. Since TwinBlk, SuperPage, ParaGC and `Die-GC` are designed to erase more blocks in each GC process, the number of erase operations are larger than that of Baseline for most workloads. The reason is that, reclaiming blocks from different planes at once time may trigger premature GCs [48], [49]. However, the results show that the number of erase operations of `Die-GC` is much smaller than TwinBlk, SuperPage and ParaGC. For example, TwinBlk, in the worst case, introduces more than 102.2 percent erase operations for PRXY_0, compared with Baseline-D. ParaGC, introduces more than 65.8 percent erase operations compared with Baseline-D. Worst of all, SuperPage triggers 177.7 percent erase operations compared with Baseline-D. Compared with these three related works, SPD introduces fewer erase operations in most cases. On average, the number of erase operations is reduced by 13.43, 34.23 and

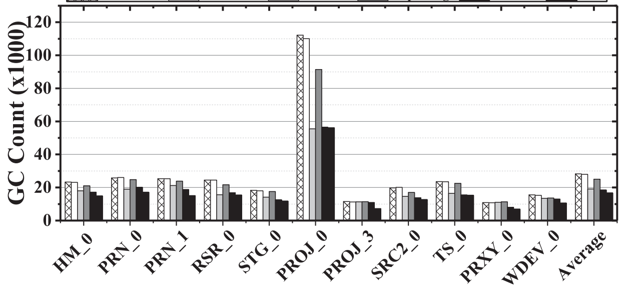


Fig. 16. The total number of triggered GC.

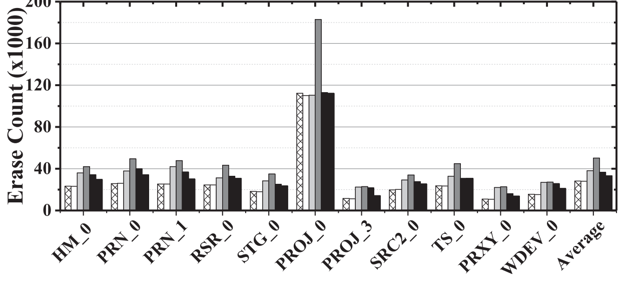


Fig. 17. The total number of erase operations.

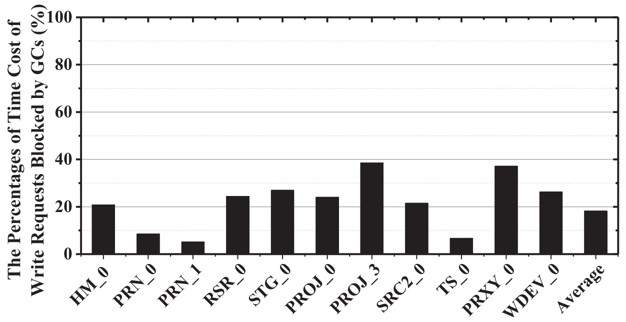


Fig. 18. The impact of access conflict between Die-write and Die-GC. The percentages of total time cost of write requests blocked by GCs are evaluated.

10.04 percent compared with TwinBlk, SuperPage and ParaGC. The reason comes from that `Die-GC` is triggered with regarding the whole die as the smallest unit without introducing additional valid page movements. In addition, the write amplification is also avoided to reduce total GC count.

6.4 Experiment Results of SPD+

In order to evaluate the effectiveness of proposed SPD+, the impact of access conflict between `Die-Write` and `Die-GC` is measured first, which is the potential of SPD+. In SPD scheme, write requests in the constructed `Die-Write` are identified as blocked write requests while there is a `Die-GC` in the accessing die. To indicate the impact of access conflict, the percentages of blocked write requests' time cost are evaluated and presented in Fig. 18. On average, total time cost of write requests blocked by GCs accounts for 18.2 percent. The results presented in Fig. 18 show a matching pattern with Fig. 13, where the write performance improvements of PRN_0, PRN_1 and TS_0 are slight while the GC impacts on PRN_0, PRN_1 and TS_0 presented in Fig. 18 is weak. Take PRN_1 as an example. In Fig. 18, the total time cost of write requests blocked by GC of PRN_1 only accounts for 5.1 percent while the achieved write performance improvement of SPD presented in Fig. 13 is 13.7 percent, which is the minimal write performance improvement among all workloads.

By adopting the proposed combination scheme, which constructs `Die-Write` by grouping evicted page from the cache and valid page from victim block, the waiting time of evicted page generated write requests can be significantly reduced. The results of average write latency of SPD and SPD+ are evaluated and presented in Fig. 19. On average, SPD+ can further reduce the average write latency by 23.2

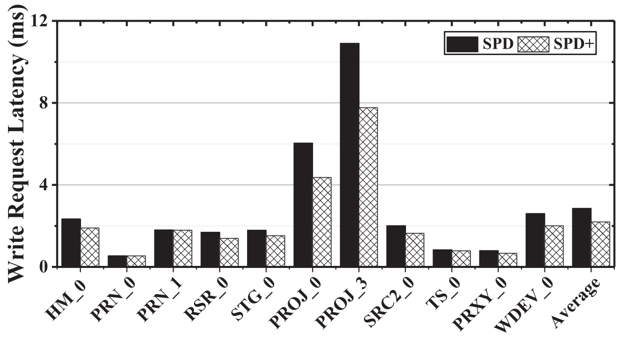


Fig. 19. The write latencies of SPD and SPD+.

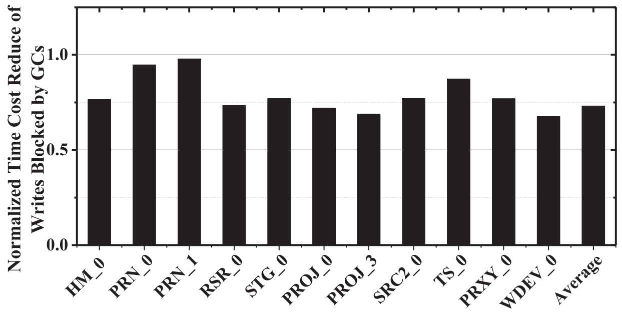


Fig. 20. Normalized time cost reduce of write requests blocked by GCs.

percent compared with basic SPD scheme. For these workloads little affected by access conflict (PRN_0, PRN_1 and TS_0), the achieved write latency decreases also are slight, accounting for 0.5, 0.2 and 5.1 percent, respectively. On the contrary, for PROJ_3, of which the total time cost of write requests is highly affected by GCs, the achieved write latency decrease reaches 28.7 percent.

To elaborate the reason of achieved significantly write performance improvement, the results of average time cost of write requests blocked by GCs are evaluated and presented in Fig. 20, where the results of SPD+ are normalized to SPD. For SPD+, while write requests can be processed in parallel with valid page movements during GC process, the waiting time of write requests can be reduced, causing the average time cost drop. In this figure, one can see that the average time cost of write requests blocked by GCs is reduced by 27.1 percent. Similarly, for PRN_1, which is least affected by GC induced access conflict, the reduced average time cost of write requests blocked by GCs also is minimal, only being reduced by 2.2 percent. On the other hand, for PROJ_3, the achieved average time cost reduction of write requests blocked by GCs is the maximal one, reaching 31.2 percent. In conclusion, the proposed combination scheme can be applied to significantly reduce waiting time of write requests blocked by GCs, making write performance to be further improved.

For GC evaluation, the total GC time cost, average GC time cost and GC count of SPD+ are similar to SPD, which are increased by 0.012, -0.015 and 0.027 percent, respectively. Since proposed SPD+ is designed to schedule write requests for reducing waiting time, the GC process will not be affected, and then the metrics of evaluated GC process are similar to SPD.

For read latency, SPD+ still executes read requests with highest priority, making the achieved read latency be similar to SPD. On average, the read latency of SPD+ is reduced by 0.33 percent compared with SPD.

6.5 Sensitive Studies

1) Buffer Size Impact: In this part, the write intensive workload, RSR_0, is selected for buffer size sensitivity study. Buffer size is different within different devices. Its impact on SPD is presented. Fig. 21 shows the results of the normalized write latencies of the six schemes by varying buffer size from 256 KB to 16 MB. During the evaluation, GC is not triggered to only understand the impact from different buffer sizes. Two observations can be concluded from the results. *First*, with larger buffer size, the write latencies of all schemes can

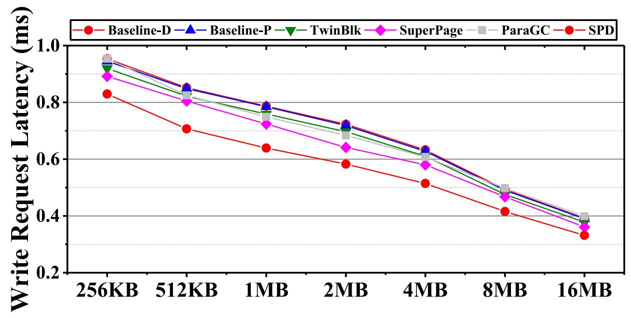


Fig. 21. Write latency with different buffer sizes.

be further reduced. This is because that more dirty pages can be stored and higher hit ratio can be achieved. *Second*, compared with other schemes, stable write latency reduction is achieved by SPD with different buffer sizes. The proposed framework is designed to align the write point of planes all the time. It has benefit once there are multiple write operations issued to a die.

2) Four-Plane SSD Evaluation: In this part, SSD with four planes per die is evaluated for SPD. For the four planes of a die, each paired planes can be accessed in parallel with the support of *multi-plane command* [1], [10]. The results of write latencies for Baseline-D, Baseline-P, and SPD are presented in Fig. 22, where GC is not triggered to evaluate the single influence from more planes per die. *First*, Baseline-P has similar write latency to that of Baseline-D. Four-plane SSD requires that only paired plane 0&1 or 2&3 can be processed in parallel. Only a few write operations can be processed with the support of *multi-plane command*. *Second*, for SPD with four-plane SSD, the write latency is further reduced. This is because that all four planes are regarded as one unit in *Die-Write*. Therefore, more dirty pages can evicted and written back as `Die-Write` at the cost of one write operation when the number of planes in a die increases. On average, compared with Baseline-D, SPD achieves 43.9 percent write latency reduction, on average.

3) Various Mapping Schemes based SSDs: In this part, demand-based selective caching of page-level address mappings, termed DFTL [4], and hybrid mapping scheme are implemented to show the effectiveness of proposed SPD. Similar to the settings of above evaluations, GC is not triggered in this evaluation. For DFTL based SPD, mapping entries evicted from RAM buffer are organized as *Die-Write* as well. When mapping entries are supposed to be read from flash memory, *Die-Read* is generated to read required

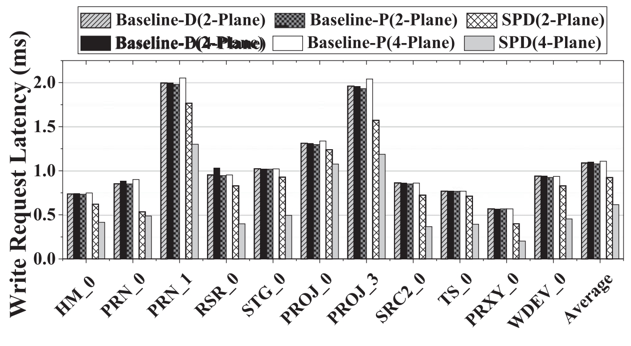


Fig. 22. Write latency with 4 planes per die.

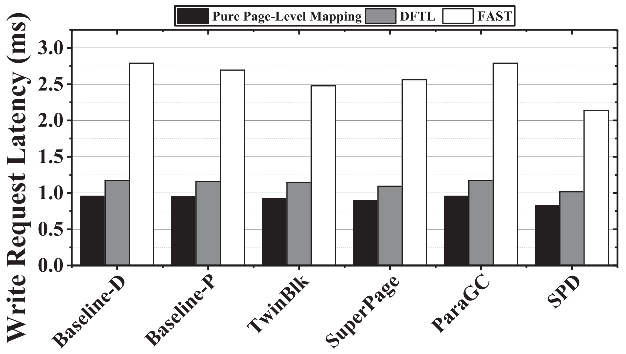


Fig. 23. Write latencies of pure page-level mapping, DFTL and FAST based evaluated schemes.

mapping entries as well. For hybrid mapping, FAST is implemented as an example [50]. In FAST based SPD, paired log blocks resided in paired planes are reserved. That is, write requests can be written into paired log blocks in the form of `Die-Write` as well. While merge operation is required, valid pages in selected paired log blocks and corresponding data blocks are read out and written to new paired data blocks by `Die-Write`.

Take RSR_0 as an example, Fig. 23 shows the write latencies of pure page-level mapping, DFTL and FAST based evaluated schemes. Herein, the write latencies of DFTL based schemes are worse than that of pure page-level mapping based schemes. For SPD, the write latency of DFTL based SPD is increased by 22.47 percent compared with pure page-level mapping based SPD. For FAST based schemes, since there are additional merge operations caused by FAST mapping, the write latencies of FAST based schemes are highly larger than that of other mappings based schemes respectively. But for FAST based SPD, it still can achieve substantial write performance improvement compared with other schemes equipped with FAST. This is because that, not only host write requests can be processed in parallel, but also each merge operation can reclaim two log blocks at the time cost of one merge operation. On average, compared with Baseline-D equipped with FAST mapping, FAST based SPD can reduce write latency by 23.43 percent.

4) Hot/Cold Separation based GC: To further reduce GC time cost, a widely adopted method is to distribute data to different blocks according to their hotness. Inside SSD, valid pages in victim block are cold enough, since only hot data are evicted from buffer and only relatively hot data in flash blocks tend to be invalidated. To further improve the efficiency of GC process, cold and relatively hot valid pages in victim blocks are written to different blocks according to their hotness as well in the revision [51], [52]. To realize SPD with considering hot/cold separation in GC process, each plane maintains two active blocks, cold block and hot block. To separate hot and cold data, the update count within a fixed time interval is recorded, which is set to 120 minutes according to [53]. Within 120 minutes, most data in evaluated workloads are updated. Except for valid pages from GC, host data evicted from buffer are considered as hot data and written into hot block as well.

The writ latencies of evaluated schemes with and without hot/cold data separation are collected and presented in Fig. 24. In this figure, two observations can be concluded.

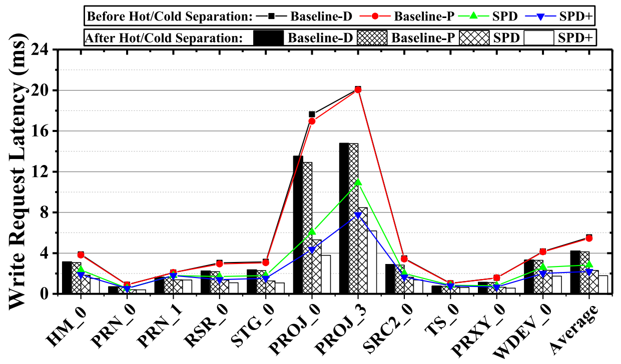


Fig. 24. Write latencies of evaluated schemes with and without hot/cold separation.

First, after applying hot/cold data separation, proposed SPD and SPD+ still can noticeably reduce write latency. On average, compared with Baseline-D, SPD+ can reduce write latency by 57.64 percent. Second, hot/cold data separation can significantly reduce write latency. Averagely, take Baseline-D as an example, scheme applied with hot/cold data separation can reduce write latency by 23.83 percent. The reasons come from two aspects: first, GC efficiency is improved while fewer valid pages are moved during GC process; second, total GC count can be reduced while cold pages are barely moved.

7 Related Works

In this section, related works on improving the plane level parallelism and reducing GC impact on performance are presented, respectively.

(1) Plane Level Parallelism Exploration: In order to improve plane level parallelism, several previous works have been proposed. Gao *et al.* [18] and Jung *et al.* [44] proposed to increase the potential of using *multi-plane command* through distributing requests belonging to different planes at one time. Similarly, Abdurrab *et al.* [29] proposed DLOOP to modify mapping policy to evenly distribute data across planes based on a fixed location calculation. However, the achieved performance is limited since they highly depend on the access patterns of workloads to match the limitations of *multi-plane command*. On the other hand, Tavakkol *et al.* [11] and Hu *et al.* [10] proposed to align writing points of planes. Tavakkol *et al.* [11] proposed to maintain the write points to distribute writes among planes in round-robin fashion. However, due to the above mentioned unaligned access problem, plane level parallelism still can not be fully exploited. Hu *et al.* [10] proposed a greedy *multi-plane command*. They proposed to allocate new writing points in the same position. However, this will waste space. Caulfield and Seong *et al.* [17] and [45] proposed a new design principle for fully exploiting plane level parallelism, which groups all pages residing in the same in-chip location as a large logical page, termed super page. Such design can boost the performance and bandwidth of SSDs at the penalty of lifetime.

Different from all these works, SPD is the first on proposing to align the write points in an active way. `Die-Write` is designed to align the write point all the time. In this case, all write operations issued to multiple planes in a die can be processed in parallel. Also, SPD can minimize the impact on the lifetime of SSDs by grouping pages together more flexible.

(2) Garbage Collection Impact Minimization: Previous works aiming at reducing GC impact on performance can be classified into two groups: The first group proposed to reduce the time cost of GC activity [13], [54]; For example, Gao *et al.* [13] proposed to reduce the time cost of valid page movement through migrating valid pages to idle chips. Park *et al.* [54] proposed a new hotness identification method for accurately capturing the recency and frequency of data. The second group proposed to schedule requests or GCs to reduce the impact on performance of SSDs [12], [14], [55]. For example, Wu *et al.* [14] used cache to store requests conflicted by GC. Jung *et al.* [55] proposed to advance or delay GC through moving the time-consuming activity from busy period to idle period. Choi *et al.* [12] proposed to combine host I/O operations with valid pages migration. However, the aforementioned GC optimization methods still have not taken unaligned access problem of plane level parallelism into consideration.

There are two works proposed to reduce GC impact resulted from unaligned access problem. Shahidi *et al.* [9] proposed ParaGC to select paired planes, where GC activities can be processed in parallel. However, if the paired planes can not be found, unaligned access problem still exist. Tavakkol *et al.* [11] proposed TwinBlk, which can minimize the unaligned access induced impact on GC. TwinBlk is designed to trigger GCs on all planes of the same die simultaneously so that symmetric victim blocks on planes can be reclaimed in parallel. During this process, valid pages are evenly moved to all planes in round robin policy for aligning write points of all planes.

Different from these works, SPD uses `Die-GC` to speed up the GC process and reduce the GC cost. `Die-GC` is designed to select multiple blocks in the unit of die and adopt `Die-Write` to speed up the GC process.

8 Conclusion

In this work, a *from plane to die* optimization framework is proposed to exploit the plane level parallelism, which is the last level parallelism of SSDs. Three components are designed in the framework: die level write construction, die level GC and combination scheme. Different from previous work, this work is the first which is able to maintain the aligned write points for the multiple planes for each die at the time. There are two components designed to align the write points of all planes in the same die all the time. In this case, the last level parallelism, plane level parallelism, is fully exploited to improve the performance of write requests and internal activities. In addition, the combination scheme is used to construct new die level write containing dirty page evicted from cache and valid page in victim block. The combination scheme can largely reduce the waiting time of write requests blocked by GCs, bringing write latency decrease. Experiment results show that SPD and SPD+ achieve significant write performance improvement and much smaller lifetime impact compared with state-of-the-art works.

Acknowledgments

This work is supported in part by NSFC 61772092 and 61872049, NSF 1910413,1617071,1725657 and 1718080, the

Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CityU 11219319), the Frontier Interdisciplinary Research Funds for the Central Universities (Project No. 2018CDQYJSJ0034). This work was carried out while Congming Gao was visiting the University of Pittsburgh on a CSC scholarship.

REFERENCES

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proc. Annu. Tech. Conf.*, 2008, pp. 57–70.

[2] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, 2011, pp. 266–277.

[3] C. Gao, L. Shi, M. Zhao, C. J. Xue, K. Wu, and E. H.-M. Sha, "Exploiting parallelism in I/O scheduling for access conflict minimization in flash-based solid state drives," in *Proc. 30th Symp. Mass Storage Syst. Technol.*, 2014, pp. 1–11.

[4] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. 14th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2009, pp. 229–240.

[5] F. Chen, T. Luo, and X. Zhang, "CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives," in *Proc. 9th USENIX Conf. File Stroage Technol.*, 2011, pp. 77–90.

[6] M. Jung and M. T. Kandemir, "An evaluation of different page allocation strategies on high-speed SSDs," in *Proc. 4th USENIX Conf. Hot Topics Storage File Syst.*, 2012, pp. 9–13.

[7] R. Micheloni, A. Marelli, and S. Commodaro, "NAND overview: From memory to systems," *Inside NAND Flash Memories*. Berlin, Germany: Springer, 2010.

[8] ONFI, "Open NAND flash interface specification 4.1. website," 2017. [Online]. Available: http://www.onfi.org/ /media/onfi/specs/onfi_4_1_gold.pdf?la=en

[9] N. Shahidi, M. Arjomand, M. Jung, M. T. Kandemir, C. R. Das, and A. Sivasubramaniam, "Exploring the potentials of parallel garbage collection in SSDs for enterprise storage systems," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 561–572.

[10] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity," in *Proc. Int. Conf. Supercomputing*, 2011, pp. 96–107.

[11] A. Tavakkol, P. Mehrvarzy, and H. S. -Azad,"TBM: Twin block management policy to enhance the utilization of plane-level parallelism in SSDs," *IEEE Comput. Archit. Lett.*, vol. 15, no. 2, pp. 121–124, Jul.-Dec. 2016.

[12] W. Choi, M. Jung, M. Kandemir, and C. Das, "Parallelizing garbage collection with I/O to improve flash resource utilization," in *Proc. 27th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2018, pp. 243–254.

[13] C. Gao *et al.*, "Exploiting chip idleness for minimizing garbage collection induced chip access conflict on SSDs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 23, pp. 1–29, 2017.

[14] S. Wu, B. Mao, Y. Lin, and H. Jiang, "Improving performance for flash-based storage systems through GC-aware cache management," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 10, pp. 2852–2865, Oct. 2017.

[15] S. Wu, Y. Lin, B. Mao, and H. Jiang, "GCaR: Garbage collection aware cache management with improved performance for flash-based SSDs," in *Proc. Int. Conf. Supercomputing*, 2016, pp. 1–12.

[16] M. Jung, W. Choi, S. Srikantaiah, J. Yoo, and M. T. Kandemir, "HIOS: A host interface I/O scheduler for solid state disks," *ACM SIGARCH Comput. Archit. News*, vol. 42, pp. 289–300, 2014.

[17] A. M. Caulfield, L. M. Grupp, and S. Swanson, "Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications," *ACM SIGARCH Comput. Archit. News*, vol. 37, pp. 217–228, 2009.

[18] C. Gao *et al.*, "Exploiting parallelism for access conflict minimization in flash-based solid state drives," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 37, no. 1, pp. 168–181, Jan. 2018.

[19] S. Choudhuri and T. Givargis, "Performance improvement of block based NAND flash translation layer," in *Proc. 5th IEEE/ACM Int. Conf. Hardware/Softw. Codesign Syst. Synthesis*, 2011, pp. 257–262.

[20] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Trans. Embedded Comput. Syst.*, vol. 6, pp. 18–44, 2007.

[21] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "System software for flash memory: A survey," in *Proc. Int. Conf. Embedded Ubiquitous Comput.*, 2006, pp. 394–404.

[22] E. Gal and S. Toledo, "Algorithms and data structures for flash memories," *ACM Comput. Surv.*, vol. 37, pp. 138–163, 2005.

[23] Y. Pan, G. Dong, and T. Zhang, "Exploiting memory device wear-out dynamics to improve NAND flash memory system performance," in *Proc. 9th USENIX Conf. File Stroage Technol.*, 2011, pp. 18–31.

[24] Y.-J. Woo and J.-S. Kim, "Diversifying wear index for MLC NAND flash memory to extend the lifetime of SSDs," in *Proc. Int. Conf. Embedded Softw.*, 2013, Art. no. 6.

[25] H. Kim and S. Ahn, "BPLRU: A buffer management scheme for improving random writes in flash storage," in *Proc. 6th USENIX Conf. File Storage Technol.*, 2008, pp. 1–14.

[26] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proc. 2nd USENIX Conf. File Storage Technol.*, 2003, pp. 115–130.

[27] L. Shi, J. Li, C. J. Xue, C. Yang, and X. Zhou, "ExLRU: A unified write buffer cache management for flash memory," in *Proc. 9th ACM Int. Conf. Embedded Softw.*, 2011, pp. 339–348.

[28] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *IEEE Comput. Archit. Lett.*, vol. 10, no. 1, pp. 16–19, Jan.-Jun. 2011.

[29] A. R. Abdurrab, T. Xie, and W. Wang, "DLOOP: A flash translation layer exploiting plane-level parallelism," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 908–918.

[30] P. Desnoyers, "Analytic models of SSD write performance," *ACM Trans. Storage*, vol. 10, 2014, Art. no. 8.

[31] J. Lee, Y. Kim, G. M. Shipman, S. Oral, F. Wang, and J. Kim, "A semi-preemptive garbage collector for solid state drives," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2011, pp. 12–21.

[32] M.-L. Chiang, P. C. H. Lee, and R.-C. Chang, "Using data clustering to improve cleaning performance for flash memory," in, *Software: Practice and Experience*. Hoboken, NJ, USA: Wiley, 1999.

[33] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "SFS: Random write considered harmful in solid state drives," in *Proc. USENIX Conf. File Storage Technol.*, 2012, pp. 1–16.

[34] M. Huang, Y. Wang, L. Qiao, D. Liu, and Z. Shao, "SmartBackup: An efficient and reliable backup strategy for solid state drives with backup capacitors," in *Proc. IEEE 17th Int. Conf. High Perform. Comput. Commun.*, 2015, pp. 746–751.

[35] J. Guo, J. Yang, Y. Zhang, and Y. Chen, "Low cost power failure protection for MLC NAND flash storage systems with PRAM/DRAM hybrid buffer," in *Proc. Des. Autom. Test Eur. Conf. Exhibit.*, 2013, pp. 859–864.

[36] W. H. Kang, S. W. Lee, B. Moon, Y. S. Kee, and M. Oh, "Durable write cache in flash memory SSD for relational and NoSQL databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 529–540.

[37] C. Gao, S. Liang, Y. Di, Q. Li, C. Xue, and H.M. E. Sha, "An efficient cache management scheme for capacitor equipped solid state drives," in *Proc. Great Lakes Symp. VLSI*, 2018, pp. 463–466.

[38] Micron, 7100 M.2 NVMe PCIe SSD, 2016. [Online]. Available: https://www.micron.com/ /media/documents/products/data-sheet/ssd/7100_m2_pcie_ssd.pdf

[39] C. GaoL. Shi, C. J. Xue, C. Ji, J. Yang, and Y. Zhang, "Parallel all the time: Plane level parallelism exploration for high performance SSDs," in *Proc. 35th Symp. Mass Storage Syst. Technol.*, 2019, pp. 172–184.

[40] S.-W. Lee, B. Moon, and C. Park, "Advances in flash memory SSD technology for enterprise database applications," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2009, pp. 863–870.

[41] T. Xie and J. Koshia, "Boosting random write performance for enterprise flash storage systems," in *Proc. IEEE 27th Symp. Mass Storage Syst. Technol.*, 2011, pp. 1–10.

[42] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to SSDs: Analysis of trade-offs," in *Proc. 4th ACM Eur. Conf. Comput. Syst.*, 2009, pp. 145–158.

[43] C. Gao *et al.*, "Constructing large, durable and fast SSD system via reprogramming 3D TLC flash memory," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 493–505.

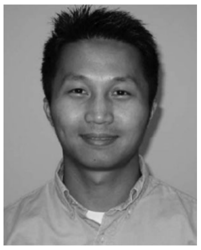
[44] M. Jung, E. H. Wilson III, and M. Kandemir, "Physically addressed queueing (PAQ): Improving parallelism in solid state disks," in *Proc. 39th Annu. Int. Symp. Comput. Archit.*, 2012, pp. 404–415.

[45] Y. J. Seong *et al.*, "Hydra: A block-mapped parallel flash memory solid-state disk architecture," *IEEE Trans. Comput.*, vol. 59, no. 7, pp. 905–921, Jul. 2010.
[46] T. Brokhman, "ROW scheduling algorithm in block layer. Website," 2012. [Online]. Available: https://lwn.net/Articles/509829/
[47] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Disk schedulers for solid state drivers," in *Proc. 7th ACM Int. Conf. Embedded Softw.*, 2009, pp. 295–304.
[48] L.-P. Chang, T.-W. Kuo, and S.-W. Lo, "Real-time garbage collection for flash-memory storage systems of real-time embedded systems," *ACM Trans. Embedded Comput. Syst.*, vol. 3, pp. 837–863, 2004.
[49] L.-P. Chang and C.-Y. Wen, "Reducing asynchrony in channel garbage-collection for improving internal parallelism of multichannel solid-state disks," in *ACM Trans. Embedded Comput. Syst.*, vol. 13, 2014, Art. no. 63.
[50] S.-W. Lee, W.-K. Choi, and D.-J. Park, "FAST: An efficient flash translation layer for flash memory," in *Proc. Int. Conf. Embedded Ubiquitous Comput.*, 2006, pp. 879–887.
[51] H.-J. Kim and S.-G. Lee, "A new flash memory management for flash storage system," in *Proc. Int. Comput. Softw. Appl. Conf.*, 1999, pp. 284–289.
[52] O. Kwon, J. Lee, and K. Koh, "EF-greedy: A novel garbage collection policy for flash memory based embedded systems," in *Proc. Int. Conf. Comput. Sci.*, 2007, pp. 913–920.
[53] W. Choi, M. Arjomand, M. Jung, and M. Kandemir, "Exploiting data longevity for enhancing the lifetime of flash-based storage class memory," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 1, 2017, Art. no. 21.
[54] D. Park and D. H. C. Du, "Hot data identification for flash-based storage systems using multiple bloom filters," in *Proc. IEEE 27th Symp. Mass Storage Syst. Technol.*, 2011, pp. 1–11.
[55] M. Jung, R. Prabhakar, and M. T. Kandemir, "Taking garbage collection overheads off the critical path in SSDs," in *Proc. 13th Int. Middleware Conf.*, 2012, pp. 164–186.

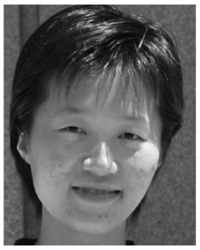
Congming Gao received the BS degree in computer science and technology from Chongqing University, China, in 2014. He is working toward the PhD degree in the College of Computer Science, Chongqing University. currently, he is a visiting scholar with the Department of Electrical and Computer Engineering, University of Pittsburgh, Pittsburgh, PA. His research interests include flash memory, non-volatile memory and architecture optimizations.

Liang Shi received the BS degree in computer science from the Xi'an University of Post & Telecommunication, Xi'an, Shanxi, China, in 2008, and the PhD degree from the University of Science and Technology of China, Hefei, China, in 2013. He is currently a full-time professor with the School of Computer Science and Technology, East China Normal University. His research interests include flash memory, embedded systems, and emerging non-volatile memory technology.

Kai Liu received the PhD degree in computer science from the City University of Hong Kong in 2011. From December 2010 to May 2011, he was a visiting scholar with the Department of Computer Science, University of Virginia, USA. From 2011 to 2014, he was a postdoctoral fellow at Singapore Nanyang Technological University, City University of Hong Kong, and Hong Kong Baptist University. He is currently a professor with the College of Computer Science, Chongqing University, China. His research interests include Internet of Vehicles, Mobile Computing and Pervasive Computing.



Chun Jason Xue received the BS degree in computer science and engineering from the University of Texas at Arlington, in May 1997, and the MS and PhD degrees in computer science from the University of Texas at Dallas, in 2002 and 2007, respectively. He is currently an associate professor with the Department of Computer Science at the City University of Hong Kong. His research interests include memory and parallelism optimization for embedded systems, software/hardware co-design, real time systems, and computer security.



Jun Yang received the BS degree in computer science from Nanjing University, China, in 1995, the PhD degree in computer science from the University of Arizona, in 2002. She is a professor with the Electrical and Computer Engineering Department, University Pittsburgh, Pittsburgh, PA. She is the recipient of US NSF Career Award in 2008. She has won best paper awards from ICCD 2007 and ISLPED 2013. Her research interests include GPU architecture, secure processor architecture, emerging non-volatile memory technologies, performance, and reliability of memories.

Youtao Zhang (Member, IEEE) received the PhD degree in computer science from the University of Arizona, Tucson, AZ, in 2002. He is currently an associate professor of Computer Science, University of Pittsburgh, Pittsburgh, PA. His current research interests include computer architecture, program analysis, and optimization. He was the recipient of the U.S. National Science Foundation Career Award, in 2005. He is also the co-author of several papers that received paper awards.

Towards Higher Performance and Robust Compilation for CGRA Modulo Scheduling

Zhongyuan Zhao, Weiguang Sheng, Qin Wang, Wenzhi Yin, Pengfei Ye, Jinchao Li, and Zhigang Mao

Abstract—Coarse-Grained Reconfigurable Architectures (CGRA) is a promising solution for accelerating computation intensive tasks due to its good trade-off in energy efficiency and flexibility. One of the challenging research topic is how to effectively deploy loops onto CGRAs within acceptable compilation time. Modulo scheduling (MS) has shown to be efficient on deploying loops onto CGRAs. Existing CGRA MS algorithms still suffer from the challenge of mapping loop with higher performance under acceptable compilation time, especially mapping large and irregular loops onto CGRAs with limited computational and routing resources. This is mainly due to the under utilization of the available buffer resources on CGRA, unawareness of critical mapping constraints and time consuming method of solving temporal and spatial mapping. This article focus on improving the performance and compilation robustness of the modulo scheduling mapping algorithm for CGRAs. We decomposes the CGRA MS problem into the temporal and spatial mapping problem and reorganize the processes inside these two problems. For the temporal mapping problem, we provide a comprehensive and systematic mapping flow that includes a powerful buffer allocation algorithm, and efficient interconnection & computational constraints solving algorithms. For the spatial mapping problem, we develop a fast and stable spatial mapping algorithm with backtracking and reordering mechanism. Our MS mapping algorithm is able to map loops onto CGRA with higher performance and faster compilation time. Experiment results show that given the same compilation time budget, our mapping algorithm generates higher compilation success rate. Among the successfully compiled loops, our approach can improve 5.4 to 14.2 percent performance and takes x24 to x1099 less compilation time in average comparing with state-of-the-art CGRA mapping algorithms.

Index Terms—CGRA, modulo scheduling, temporal mapping, spatial mapping

1 Introduction

Recent years, Coarse-Grained Reconfigurable Architectures (CGRAs) has gained many attentions from both academia and industry. Based on our knowledge, at least 40 CGRAs have been developed to adapt diverse applications in the past decades of years. These CGRAs are either positioned as accelerators or standalone processing units, and target on improving the efficiency of running applications that cover mobile computing [1], [2], [3], media processing [4], [5], [6], [7], [8], image processing [9], [10], digital signal processing (DSP) [11], [12], [13], [14], [15], [16], [17], [18], ultra-low power processing [19], [20], [21], machine learning [22], [23], [24], data or computational intensive domains [25], [26], [27], [28], [29], [30], [31], [32], and even general purpose computing [33], [34], [35], [36], [37], [38]. However, building the software ecosystem around CGRAs is challenging due to the diverse CGRA hardware design flavors and application purposes. Many compilers for CGRAs are hardware specific (works such as [28], [39], [40], [41], [42], [43]).

Therefore, in terms of generality and programmability, a generalized CGRA execution model is important for the compilers to formulate a uniform compilation problem. ADRES [44], CRC [27] and CGRA-ME [45] are representative works which provide generalized CGRA template of modeling CGRA loop accelerators. These templates define the basic components inside CGRA computing system, which includes reconfigurable computing unit (RPU) that consists of one or multiple arrays of processing elements (PE), hierarchical on-chip buffers, and interconnect networks. The template model is not specific to single CGRA design flavor and the components inside the CGRA template can be parameterized. In this way, the compiler is able to formulate a uniform problem of mapping the dataflow graph (DFG) extracted from the application program onto this generalized CGRA template, and the goal of the mapping problem depends on the CGRA execution schemes.

One of the popular execution scheme is to software pipelining the innermost loops on CGRA by overlapping the execution of successive loop iterations [46]. Operations within the same and different iterations are executed in parallel and both instruction-level parallelism and loop-level parallelism can be exploited. The interval between two consecutive iterations of the loop is referred as Initiation Interval (II), which is the most important metric to evaluate the performance of mapping, the smaller the better. In general, compilers use modulo scheduling [46] based mapping algorithm to deploy loops onto CGRA and this mapping problem has been proven to be NP-Complete [47], [48].

Earlier work [49] has formulated the CGRA module scheduling (MS) problem as mapping the DFG onto an abstracted modulo extended 3D-CGRA (CGRA is extended

• *Z. Zhao, W. Sheng, Q. Wang, P. Ye, J. Li, and Z. Mao are with the Department of Micro/Nano Electronics, Shanghai Jiao Tong University, Shanghai 200240, China. E-mail: zyzhao.sjtu@gmail.com, {wgshenghit, qinqinwang, teddyye, littledanni, maozhigang}@sjtu.edu.cn.*
• *W. Yin are with Nvidia, China. E-mail: wgshenghit@sjtu.edu.cn.*

Manuscript received 8 Oct. 2019; revised 5 Apr. 2020; accepted 16 Apr. 2020. Date of publication 21 Apr. 2020; date of current version 8 May 2020. (Corresponding author: Weiguang Sheng.) Recommended for acceptance by M. D. Santambrogio. Digital Object Identifier no. 10.1109/TPDS.2020.2989149

II in the third dimension), a space-time graph that models the computational resources and routing resources of CGRA, and their goal is to minimize the II. The mapping problem consists multiple processes such as assigning time, PE placement, PE routing and allocating buffers (registers). Hamzeh [50] classified the patterns of organizing these processes into the integrated policy or decomposed policy. With the integrated mapping policy, the process of assigning time, buffer allocating, and PE placement and routing are taken place at once in a node-by-node fashion [50]. Whereas the decomposed mapping policy handles these processes separately, each process can be formulated to a problem which is independent with others with a well-defined objective. Based on the decomposing style of existing mappings [47], [51], [52], [53], [54], [55], [56] and our mapping in this paper, we conceptually name two basic decomposed processes as the *temporal mapping* and *spatial mapping* for all the mappings using the decomposed mapping policy.

Existing CGRA MS algorithms using different mapping policies may suffer from different or same challenges:

1. *Underutilizing the buffer resources in both policies:* Most of the works do not explicitly provide optimization algorithm using all types of the available buffer resources in CGRA. Some works use single buffer types, for example, REGIMap [52] use local register buffer (LRB) distributed in PEs, MA-Map [54] leverages on-chip memory buffer (OMB). Even though RAMP [55] use both LRB and OMB, it only use them separately. The combinational utilization of PE, global register buffer (GRB), LRB and OMB is important to keep the performance competitive when there are insufficient buffer resources on CGRA.
2. *Good performance but long compilation time in algorithms using integrated mapping policy:* Even though classic optimization heuristics such as simulated annealing [57], particle swarm optimization [58] or linear integer programming (ILP) [59] are able to generate competitive mapping performance within a reasonable time budget. They still take relative long time to converge comparing with other heuristics, especially for large and irregular DFGs [52], [57], [60].
3. *Fast but sacrifice performance in algorithms using integrated mapping policy:* Other heuristics such as EMS [61], Resource-Aware [62] and GraphMinor [48] focus on some specific aspects during mapping, but lacks generality. This makes their approach faster but lower mapping efficiency over some irregularities that they do not cover.
4. *Unawareness of the interconnection and computational resources constraints in algorithms using decomposed mapping policy:* Decomposing the temporal and spatial mapping has been shown to be effective [43], [50]. However, if the temporal mapping cannot realize the insufficiency of the interconnection and routing resources in CGRA which leads to the failure of spatial mapping, the spatial mapping algorithm will perform redundant and useless searching.
5. *Time consuming spatial mapping algorithm in decomposed mapping policy:* Existing mappings such as EPIMap [47], REGIMap [52], Conflict-free [53], Mem-Aware [54] and RAMP [55] integrate buffer allocating and PE placement and routing processes into spatial mapping (Fig. 2a). They formulate spatial mapping problem of finding maximum clique, which is also an NP-Complete problem. They select a time consuming way to perform spatial mapping, which incurs the waste of time if problem 4 appears.

These motivations push us to think about a new mapping methodology of solving the CGRA MS problem towards higher performance and more robust compilation. The contributions of this paper are summarized as follows:

1. We give a comprehensive analysis on existing CGRA mapping algorithms, propose a mapping design philosophy of decomposing the temporal mapping and spatial mapping. We argue that the temporal mapping should be paid more attention than spatial mapping for the purpose of higher performance and robust compilation (Fig. 2b). Therefore, we develop a systematic temporal mapping flow which comprehensively analyzes and reschedules the DFG to help improving the successful rate of spatial mapping and the performance of the generated code.
2. In the temporal mapping flow, we formulate a buffer allocating problem by building the constraints and rules for different buffer resources and assigning the routing paths to these buffer resources according to their corresponding constraints and rules in order to maximize the released computational resources. We solve the problem by proposing a buffer allocating heuristic which is capable of using both same and hybrid buffer resources to buffer value during their lifetime. This makes our algorithm have strong adaptability over CGRAs with limited buffer resources.
3. Also in the temporal mapping flow, we propose the interconnection and computational constraints solving heuristics to foresee the incapability of the spatial mapping, and reschedule the operations in DFG to guarantee the successful rate of the following spatial mapping process.
4. We propose a fast and efficient spatial mapping heuristic which combines backtracking and reordering mechanism to guarantee the success rate of the spatial mapping. In the forward process, we use the greedy based algorithm to fast generate the optimal mapping with minimal II. If it fails, the backtracking and reordering algorithm will guarantee the successful mapping.

Experiment results show that our mapping strategy guarantees the stability of the compile time, all the selected loop kernels are able to be mapped within certain time budget. Among the successfully mapped loops, our mapping improves the performance from 5.4 to 14.2 percent and compilation time from x24 to x1099 faster in average comparing with state-of-the-art competitive CGRA mapping algorithms.

2 Architecture Overview of CGRA

CGRA is a kind of spatial architecture which is able to explicitly execute dataflow graphs. CGRA contains computational resources and routing resources. The computational

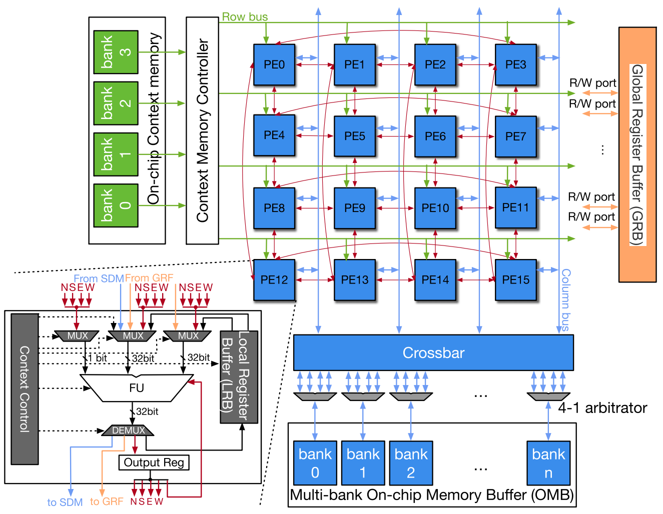


Fig. 1. Typical system architecture of CGRA which contains context memory, array of processing elements, on-chip memory buffer and global register buffer. Inside PE, there is context control unit, function unit, output register, and local register buffer.

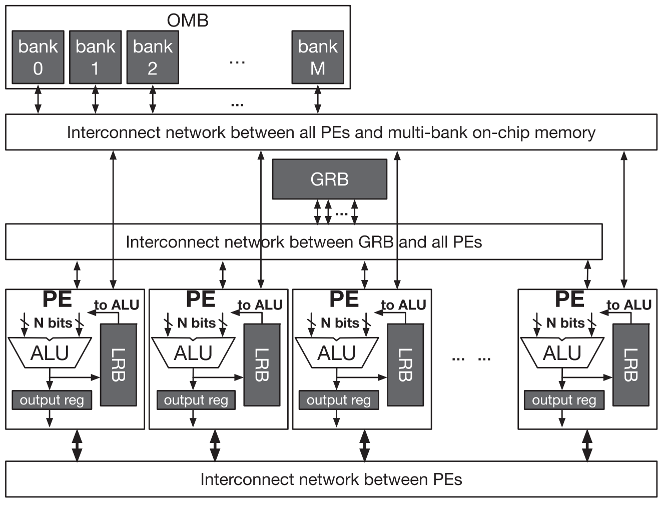

OMB
bank 0
bank 1
bank 2
bank M
Interconnect network between all PEs and multi-bank on-chip memory
GRB
Interconnect network between GRB and all PEs
PE
N bits
to ALU
ALU
LRB
output reg
Interconnect network between PEs

Fig. 3. The hierarchy of different buffer types which specified with solid gray color.

resources are referred as processing elements (PEs). The architecture of PE is shown in Fig. 1. The PE is able to receive input data from neighbor PEs (including itself), local register buffer (LRB), global register buffer (GRB) and on-chip memory buffer (OMB). The function unit (FU) inside PE performs fixed-point arithmetic and logic operations and write the computation result to the output register, LRB, GRB or OMB.

In CGRA, data is communicated through the routing resources. The routing resources includes the interconnections between PEs which we refer as the *interconnection resources*, and all the available on-chip *buffer resources* including PE, LRB, GRB and OMB. The produced data can be directly consumed through the interconnections between PEs. However, there exists situation that the produced data can not be consumed immediately due to the different latency between input values. For example, in Fig. 4a, the data produced by operation (or node) A is able to be consumed immediately by operation B, but not by node E and F. This forms a *routing path* from A to E and A to F. The *routing path* inside the DFG is a path from producer to one or multiple consumers where the producer can not be immediately consumed by the consumers.

In real world, there are many large and irregular DFGs that contain many routing paths. In this way, the compiler must use additional routing resources to route the value along the routing path. CGRA provides four buffer resources for routing, which are PE, LRB, GRB and OMB (highlighted with grey color in Fig. 3).

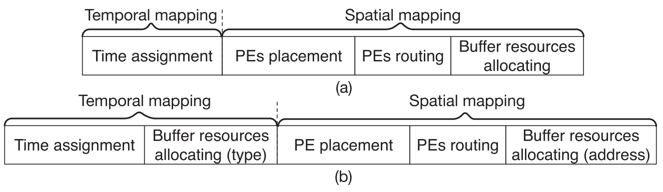


Fig. 2. (a) Process distribution over temporal and spatial mapping of the existing decomposing policy, (b) Process distribution over temporal and spatial mapping of our mapping.

1. *PE:* When we use PE to be the routing media, data is buffered in its output register (OReg). However, when the output register is used to buffer the data in a certain time, the PE can not perform any other operation during the same time. In this way, the routing node inside the routing path will be assigned to PEs and the compiler use PEs and interconnections between PEs to route value. Fig. 4b shows the example of using PE to route data produced by node A to node E and F.
2. *Local Register Buffer (LRB):* Unlike the previous buffering choice, when the value is buffered in LRB of a PE, PE can perform the other operations. However, any operation who consumes the data in LRB must be mapped to the producer PE, which brings additional mapping constraints onto the compiler, we called it the *LRB constraints*. All the routing paths inside the DFG must satisfy LRB constraints if they use LRB to perform routing. The detail of the LRB constraints will be illustrated in Section 4. Fig. 4c is an example of using LRB to route value A. One of the LRB constraints is that operation A, E, F must be executed by the same PE (same color means same PE).
3. *Global Register Buffer (GRB):* GRB can be accessed by any PEs. Any operation who consumes the data in GRB can be mapped to any PEs. However, as the overhead of the GRB scales super-linearly with its number of ports [49], the GRB resource is usually limited. Thus, GRB must be carefully used for the optimization purpose. Fig. 4d shows the example of using GRB to route value.
4. *On-chip Memory Buffer (OMB):* At last, data can be buffered into on-chip memory. The OMB can be accessed by all PEs and has relative more read/write ports than GRB because of its multi-bank designing style. However, when the compiler choose to buffer the value into OMB, the additional store/load operations will be added (Fig. 4e), and both the accessing latency and energy are larger than all the previous buffers.

One distinguished feature of our mapping over existing mappings is the combinational usage of different buffer

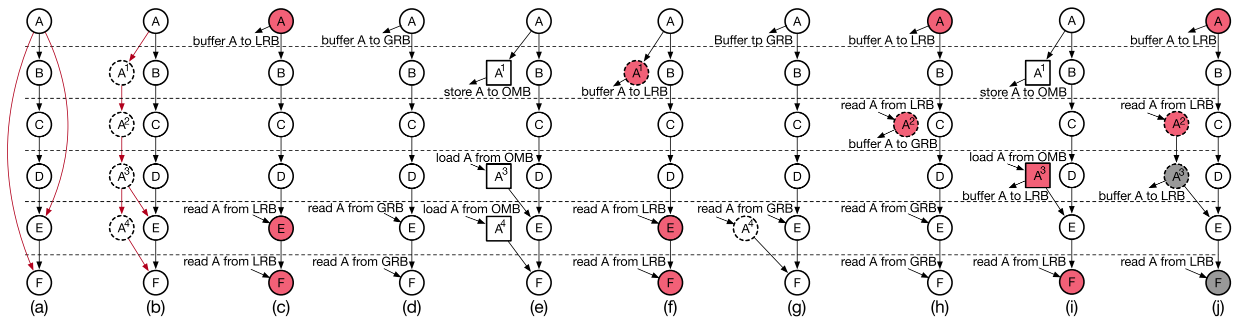


Fig. 4. Different routing methods for single routing path: (a) the original DFG, (b) only use PEs, the dash circle represent an routing operation in PE, (c) only use LRB, the colored circles must be mapped onto the same PE, (d) only use GRB, (e) only use OMB, (f) use PE and LRB, (g) use PE and GRB, (h) use LRB and GRB, (i) use OMB and LRB, (j) use LRBs of different PEs.

resources for single routing path. Fig. 4c to (i) plots the combinations of using different buffers for the shared routing path from A to E and F (red edges specified in Fig. 4b). The combinational usage of buffer resources makes our algorithm able to keep the competitive performance when certain type of buffer resource is limited in some buffer-hungry CGRAs for the purpose of power or area saving.

These examples shows that, the effective utilization of different buffer resources releases PEs from routing data so that these PEs are able to perform other valid computations. We formulate this buffer allocating problem and provide a heuristic to solve it in Section 4.2

3 CGRA Compilation Techniques

The compilation flow of CGRA starts with extracting the loops from the application and transform the corresponding code regions to the data flow graph (DFG) structure. The DFG, $D(V, E)$, is consisted of set of nodes V and edges E, where each node $v \in V$ represents a valid operation such as arithmetic, logic or even memory access, and each edge $e \in E$ represents the data dependency between the producer and consumer. The CGRA modulo scheduling technique is to find out a mapping relationship between DFG, $D(V, E)$ and time-space CGRA resources graph, $R_{II}(V_R, E_R)$.

Since 2002, there have been many CGRA modulo scheduling heuristics. We summarize these heuristics and our mapping heuristic in Table 1 from five prospectives including patterns of organizing the mapping processes, buffer resources that are explicitly leveraged for the optimization purpose during mapping, each mapping distinguished features, their placement and routing method and the 3D-CGRA model they select to use. We analysis these mappings according to the aforementioned integrated and decomposed mapping policy they use and analysis their characteristics as follows.

3.1 Mappings Using the Integrated Mapping Policy

There are pioneers who solve the CGRA MS mapping using classic optimization heuristics such as simulated annealing, particle swarm optimization and integer linear programming solving. For example, DRESC [57] use simulated annealing (SA) based mapping algorithm. PSOMap [58] use particle swarm optimization which mimics social behavior of the bird flocks. These works first formulate the problem

TABLE 1
Comparison Between Mappings (P&R: Placement and Routing; MRRG: Modulo Routing Resource Graph; MRT: Modulo Reservation Table; TEC: Time Extended CGRA)

Mapping	Pattern	Leveraged buffers	Distinguished features	P&R method	3D-CGRA format
DRESC [60]	Integrated	PE+LRB	Simulated annealing (SA)	Cost Function	MRRG
PSOMap [58]	Integrated	PE+LRB	Particle swarm optimization	Cost Function	MRRG
AA-ILP [59]	Integrated	PE+LRB	Integer Linear Programming (ILP)	ILP Solver	MRRG
MGE [63]	Integrated	PE+LRB	Skewing the scheduling space	Cost Function	MRT
EMS [61]	Integrated	PE+LRB	Edge-centric	Cost Function	MRT
RA [62]	Integrated	PE+LRB	Backtracking	Cost Function	MRRG
GM [48]	Integrated	PE	Graph-minor+Routing paths sharing	Cost Function	TEC
Bimodal [64]	Integrated	PE+LRB+CU	Adaptive cost function	Cost Function	MRRG
SPR [51]	Decomposed	PE+LRB	Simulated annealing (SA)	Cost Function	MRRG
EPIMap [47]	Decomposed	PE	Epimorphism+Re-compute	MaxClique Finding	TEC
REGIMap [52]	Decomposed	PE+LRB	LRB aware+Reordering	MaxClique Finding	TEC
MA [54]	Decomposed	PE+LRB+OMB	OMB aware	MaxClique Finding	TEC
RAMP [55]	Decomposed	PE+LRB+OMB	Re-schedule	MaxClique Finding	TEC
Ours	Decomposed	PE+LRB+GRB+OMB	All buffers aware	Cost Function	TEC

of mapping DFG onto the Modulo Routing Resource Graph (MRRG), which is one of the representation form of 3D-CGRA. Each operations in DFG are randomly assigned with time and resource, and resource conflict are checked. The resource conflict can be regarded as different operations executed on the same PE and in the same cycle. Their compilers iteratively do this until they either find a valid mapping and the performance can not be further improved (converge), or the mappings reach the compilation time budget and return sub-optimal solution.

DRESC and PSOMap integrate timing, placement and routing during mapping, and the searching space for operations covers both time and space dimensions. All these mappings perform buffer allocating in the last priority, which incurs large rescheduling overhead when the local register buffer spills.

The mappings such as [59] and [65] use 0-1 integer linear programming to solve the mapping problem between DFG and MRRG. Their goal is to find out the set of values of all the variables for the objective function. The ILP based mappings also suffers from relative long compilation time of mapping large DFG onto CGRA with limited computational and routing resources. This is because the time of finding the optimal solution critically hinges upon the power of linear programming solver. The time complexity of the ILP solver increases exponentially with the number of variables and constraints. Thus, ILP based mapping is also challenging for large loops. For example, based on the data in [59], there exists 2 kernels whose operation number in DFG is no more than 30, but the compilation time is larger than 24 hours. Whereas in our test bench, the operation number in DFG can reach 154 at most, using ILP may suffer from extremely long compilation time.

EMS [61] foresees the importance of the routing dependencies between operations. Instead of using the node-centric approach which takes the PE placement as its first priority, EMS takes an edge-centric approach which focus on the routing problem as its primary objective. The EMS lacks the backtracking or remapping mechanism when the mapping is failed, it gains faster compilation speed at the cost of performance. Bimodal [64] scheduler improve EMS by performing back-tracking and using adaptive priority functions and cost functions over different loops to make a better trade-off between performance and compilation time.

Resource-aware [62] mapping use node-centric approach to map each operation according to cost function. When mapping fails, it uses the backtracking method to remap the previous operations under the same II in order to prevent the performance from reduction.

GraphMinor [48] formalizes the mapping problem into finding a subgraph of MRRG whose minor graph is isomorphic to DFG. GraphMinor proposes a path sharing technique to keep data in the shared PE during its life time from producer to multiple consumers. This mechanism can significantly result in better PE utilization. However, GraphMinor lacks the optimization of using buffer resources in CGRA.

In summary, DRESC, PSOMap and AA-ILP are relative generalized heuristics of solving CGRA mapping for any DFG formats. Theoretically, given enough time, they are able to converge to the optimal solution. However, this time is unpredictable and heavily relies on the tunable parameters within the algorithm. In terms of compilation time, they have to adjust the tunable parameters to compromise on the performance, for example, set time budget parameter. Otherwise, the compilation time may become unacceptable, especially when compiling large and irregular loops.

Whereas heuristics such as EMS, Resource-Aware and GraphMinor are dedicated optimization heuristics which capture some specific problems during mapping process, and provide one or multiple dedicated optimizations to help fast search for the optimal solution. Their approach is fast comparing with mappings using classic heuristics, but sacrifice the performance and generality.

3.2 Mappings Using the Decomposed Mapping Policy

SPR [51] decomposes mapping into time scheduling process and placement & routing process. SPR permits the operations to be rescheduled beyond slack windows to meet data movement latency, but its mechanism is nearly the same with DRESC [57].

EPIMap [47] formulates the graph epimorphism problem with the additional feature of re-computations. A systematic approach is used to schedule (perform timing) each operation and transform the DFG into a modified graph that satisfies several constraints. It converts the placement and routing process to finding a maximum common subgraph of a time-extended CGRA (TEC) graph which is isomorphic to the modified DFG. To find the subgraph, the mapper build a compatible graph between the modified DFG and TEC where each node represents a possible mapping pair (between operation and PE in certain time slot), and each edge reflects a possible valid mapping. Some edges will be eliminated due to the mapping constraints. Finally, the compiler needs to find a maximum clique whose number of nodes equals to the number of the operations in the modified DFG. For values with long life time, EPIMap prefers to use PE as the routing engine, which occupies the computational resources. Whereas REGIMap [52] explicitly utilizes the LRB to buffer the value. It also combines buffer allocating with placement and routing, which is able to effectively avoid the large remapping overhead due to register spilling. Memory-aware [54] is an algorithm that explicitly use OMB during mapping process. Instead of using OMB as the final option that tackling the register spilling, memory-aware mapping put OMB allocating into temporal mapping process. RAMP [55] is an extension of the REGIMap, it provided a stronger re-scheduling heuristic to tackle the mapping failure which aims to prevent the compiler from large searching time. RAMP provides a comprehensive choices to solve the routing problem. Right now, it is a very competitive mapping in both performance and compilation time comparing with other mappings. However, the robustness of the compilation time is still questionable in EPIMap, REGIMap and RAMP, as the compatible graph is a large graph with dense connectivity, finding maximum clique is also an NP-complete problem, the time complexity of the heuristic they use in their paper is $O(N^8)$ [55]. The critical problem is that these mappings can not judge whether the maximum clique exist and trapped into time consuming searching. Although RAMP provides some strong heuristics of rescheduling, the fetal problem still remains.

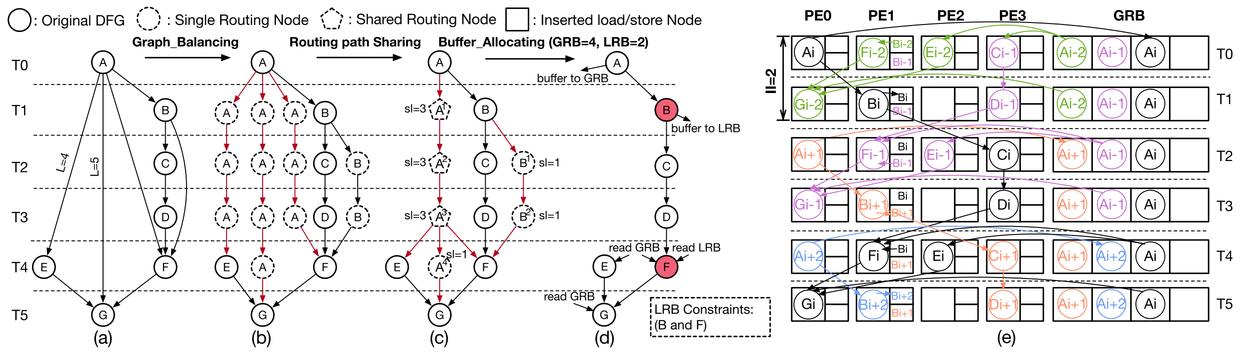


Fig. 5. (a) The original DFG, (b) DFG after graph balancing, (c) DFG after routing paths sharing, there are two routing paths highlighted by red edges, (d) the final modified DFG after our temporal mapping over CGRA with LRB=2, GRB=4, (e) final mapping after our spatial mapping.

In conclusion, for mappings which decompose the temporal and spatial mapping process. The key is to build a reliable mechanism of organizing temporal and spatial mapping phases. It should be able to foresee the impossibility of performing spatial mapping over DFG after temporal mapping and the compiler is able to realize this impossibility without taking too much time.

Thus, our mapping use a totally different philosophy of designing the temporal and spatial mapping. We think temporal mapping should perform a efficient optimization and more comprehensive analysis over DFG to guarantee the success of spatial mapping. It should reach the following goals:

- The temporal mapping should decide the schedule time for each operation in DFG and provide the routing solution for all the routing paths inside DFG according to the available routing resources inside CGRA.
- All the buffer resources including PE, LRB, GRB and OMB should be leveraged during temporal mapping for the optimization purpose.
- A powerful routing constraint solving algorithm should be used to check whether the interconnections in CGRA are enough to perform spatial mapping over DFG after temporal mapping.
- A powerful computational constraint solving algorithm should be used to check whether the PEs in CGRA are enough to support the parallelism generated in the temporal mapping phase and re-schedule the time for some operations.

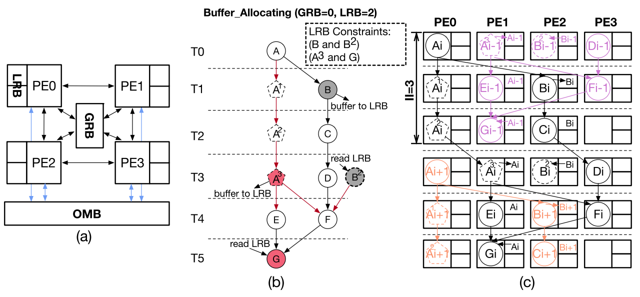

LRB
PE0
PE1
GRB
PE2
PE3
OMB
(a)
Buffer_Allocating (GRB=0, LRB=2)
LRB Constraints: (B and B2) (A3 and G)
buffer to LRB
read LRB
(b)
PE0
PE1
PE2
PE3
II=3
(c)

Fig. 6. (a) 2x2 CGRA with LRB, GRB and OMB, we use two configurations of CGRA, one is LRB=2, GRB=4, another one LRB=2, GRB=0, (b) the modified DFG generated through our temporal mapping flow over 2X2 CGRA with LRB=2 and GRB=0 from the original DFG in Fig. 5(a), (c) The final mapping generated from our spatial mapping algorithm.

On the other hand, spatial mapping algorithm should be fast and efficient instead of time consuming.

The details of the temporal mapping flow is introduced in Section 4, and the spatial mapping algorithm is introduced in Section 5

3.3 Other Related CGRA Compiling Techniques

There is also CGRA mapping work [66] uses dynamic compilation techniques to generate efficient code with lower compilation time. RL-Map [67] leverages the deep reinforcement learning model to build methods that learn to map DFGs onto spatially programmed CGRAs directly from experiences. This approach currently only covers the 2D spatial mapping. The mapping approach like conflict-free [53] combines modulo scheduling and memory conflict optimization to reduce the conflicts when multiple PEs access concurrently access shared data memory. In this way, the synchronization overhead inside CGRA can be efficiently reduced.

Despite the novelty of these works, their execution models and optimization goals are not purely the modulo scheduling problem, therefore, are different with the work in this paper.

4 Temporal Mapping Flow

The temporal mapping flow gets the original DFG as the input and outputs a modified DFG that is ready for spatial mapping. In the modified DFG, all the nodes will have a certain schedule time and their data will also be assigned to buffer into a specific buffer resource. If the data is buffered into LRB, there will be additional LRB constraints. Fig. 5d is one example of the modified DFG. It specifies the routing solution, that is, data produced by operation A is buffered in GRB and then consumed by E, F and G. The data produced by B is buffered in LRB and consumed by F. B and F must be mapped onto the same PE.

Fig. 7 shows the organization of the temporal mapping flow. There are six basic algorithms inside our temporal

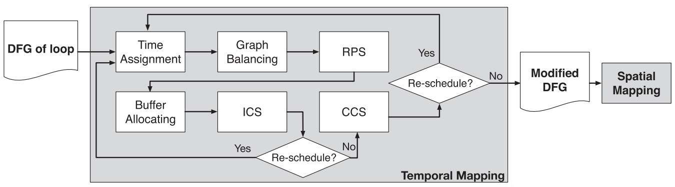


Fig. 7. Temporal mapping flow overview.

TABLE 2
List of Symbols

Symbol	Description
N_{pe}	Number of PEs in CGRA
N_l	Size of local register buffer (LRB)
N_g	Size of global register buffer (GRB)
$D(V,E)$	The original DFG
$D_m(V_m,E_m)$	The Modified DFG during scheduling
$R_{II}(V_R,E_R)$	The time-extended CGRA (TEC)
$Pre(v)/Suc(v)$	Node set which are the predecessor/ successor of node v
$I_{pre}(S)$	If $S=\{v_1,\dots,v_n\}$, $I_{pre}(S)=Pre(v_1)\cap\dots\cap Pre(v_n)$
$I_{suc}(S)$	If $S=\{v_1,\dots,v_n\}$, $I_{suc}(S)=Suc(v_1)\cap\dots\cap Suc(v_n)$
$U_{pre}(S)$	If $S=\{v_1,\dots,v_n\}$, $U_{pre}(S)=Pre(v_1)\cup\dots\cup Pre(v_n)$
$U_{suc}(S)$	If $S=\{v_1,\dots,v_n\}$, $U_{suc}(S)=Suc(v_1)\cup\dots\cup Suc(v_n)$
$T_i(S)$	Node set scheduled at time i in node set S, $S\subseteq V_m$
$M_j(S)$	Node set in set S scheduled at modulo time j, $S\subseteq V_m$
$R_j(S)$	Set of all PEs in set S of time slot j in V_R, for example, $\{PE_0^j, PE_1^j,\dots,PE_{N_{pe}-1}^j\}$
$C(S,n)$	The set of combination of selecting n elements from set S

mapping flow, which are time assignment, graph balancing, routing paths sharing (RPS), buffer allocating, interconnection constraints solving (ICS) and computational constraints solving (CCS). The flow iteratively goes through these algorithms until it thinks the modified DFG is ready for spatial mapping. We will briefly introduce the former three algorithms which are mainly based on existing works [47], [48], [68] and emphasis on introducing the latter three novel algorithms. Before going deeper into each algorithm, we list all the symbols that will appear in the following texts and their descriptions in Table 2.

4.1 Time Assignment, Graph Balancing, and Routing Path Sharing

Time assignment: The time assignment algorithm statically calculates the schedule time for all the nodes in DFG. The goal of the time assignment is to keep the life time of every variable as short as possible while maintaining the data dependency between operations. The insight of minimizing the lifetime of value is to release the routing resources for the other valid operations as much as possible. Therefore, we use life time sensitive scheduling algorithm [68] to decide the schedule time for each node. The details of the scheduling algorithm can be referred in paper [68]

Graph Balancing. The concept of graph balancing is first proposed in EPIMap [47]. Essentially, graph balancing is a technique to guarantee every node gets all of their input data at the same time. When there exists $v_i, v_j \in V$ and $(v_i, v_j)\in E$, if $t(v_i)-t(v_j)>1$, where $t(v_i)$ denotes the schedule time of node v_i, the DFG is not balanced. In this way, a routing node should be inserted between v_i and v_j. The number of the routing node added into the path is equal to $t(v_i)-t(v_j)-1$. Routing node can be regarded as virtually assign a PE to buffer the value for one cycle. The PE keep the value until it is consumed by the successor node. Figs. 5a and 5b reflects the graph balancing example. The dash circles in Fig. 5b represent the routing nodes which keep the result of operation A and B. Actually, recall Section 2, all the routing nodes are added into the *routing paths*.

Routing Paths Sharing (RPS). The RPS optimization is first proposed in Graph Minor [48]. The motivation of RPS is to save PEs for routing data. If there exists routing nodes scheduled at the same time and keep the same data, they can share the same predecessor to be a shared routing node. Formally, the RPS is to merge all the routing paths who share the same producer into a single routing path. For example, in Fig. 5b, there are three routing nodes keeping the same data of operation A in time $T1$, $T2$ and $T3$. Three routing paths from A to E, F and G can share the same data during its lifetime by combining their routing nodes in $T1$, $T2$ and $T3$ into a single routing node A^1, A^2, A^3 and forms a shared routing path (Fig. 5c). In this way, the DFG after RPS saves six routing nodes and edges. We use sl to represent the *shared_level* of each routing node. The *shared_level* of the routing node reflects how many successors will consume the same data.

4.2 Buffer Allocating

The buffer allocating algorithm is an optimization leveraging PE, GRB, LRB and OMB to buffer values during their lifetime so that the PE is able to be saved for the other valid computations.

4.2.1 Problem Formulation

Definition 4.1 (Routing path). *Let RPs be the set of all the routing paths in DFG, $D_m(V_m,E_m)$, after routing paths sharing. Let $P_i(V_p^i,E_p^i)$ be the routing path ($0\le i<|RPs|$) inside RPs, where V_p^i and E_p^i are the node and edge set inside the routing path P_i. $P_i(V_p^i,E_p^i)$ is a acyclic weakly connected graph whose V_g^i must contains one unique producer $u_{P_i}^0\in V_p^i$, J_i routing nodes $u_{P_i}^1, u_{P_i}^2,\dots,u_{P_i}^{J_i}$ ($J_i\ge 1$), and K_i consumers $v_{P_i}^0, v_{P_i}^1,\dots,v_{P_i}^{K_i-1}$ ($K_i\ge 1$). Let $u_{P_i}^j\in V_p^i$ ($1\le j\le J_i$) be the* routing node *which is j cycles latency relative to the producer ($t(u_{P_i}^j)-t(u_{P_i}^0)=j$).*

The routing paths in DFG after routing paths sharing have 4 features:

1. Every routing paths has its unique producer and routing nodes.
2. The routing node may have multiple successors, but have only one predecessor.
3. Except for the routing node $u_{P_i}^1$, all the successors of the producer $u_{P_i}^0$ do not belong to routing path.
4. The life time of routing path P_i, $LT(P_i)=J_i+1=\max_{\forall v_{P_i}^k\in V_p^i}(t(v_{P_i}^k)-t(u_{P_i}^0))$, where $0\le k<K_i$.

Essentially, assigning a routing path to GRB, LRB or OMB is to assign the routing nodes along the path to these buffers. Different buffer assignment over the routing path will lead to different DFG modification rules and constraints.

DFG modification rules of buffer allocating over P_i:

1. *GRB assignment rule*: eliminate all the routing nodes in V_p^i and all the incoming and outgoing edges of these eliminated routing nodes.

2. *LRB assignment rule*: same as (1), additionally, same PE constraints that the producer $u^0_{P_i}$ and all the K_i consumers of P_i should be mapped onto the same PE.
3. *OMB assignment rule*: same as (1), additionally, add store node $u^1_{P_i}$ and edge $e(u^0_{P_i}, u^1_{P_i})$, if there exists consumer $v^k_{P_i}$ $(0 \leq k < K_i)$, where $t(v^k_{P_i}) - t(u^0_{P_i}) > 2$, add load node $u^{j-1}_{P_i}$ and edge $e(u^{j-1}_{P_i}, v^k_{P_i})$.

In order to trigger the rules of assigning the routing path to GRB, LRB and OMB, their corresponding constraints must be satisfied.

GRB constraints:

1. The GRB must be available to buffer all the routing paths that is assigned to GRB under II according:

$$N^i_{req} = \left\lfloor \frac{LT(G_i)}{II} \right\rfloor + 1 \tag{1}$$

$$\sum_{i=0}^{|GRPs|-1} N^i_{req} \leq N_g, \tag{2}$$

where $GRPs$ denotes the set of routing paths which are assigned to GRB, $G_i(V_g, E_g)$ is the GRB routing path i and N^i_{req} denotes the GRB requirement for routing path G_i.
2. In order to get benefits from GRB assignment, there should be at least one routing node eliminated from G_i according to GRB assignment rule, that is, $LT(G_i) > 2$.

LRB constraints:

We use $LRPs$ to denote the set of routing paths assigned to LRB. For each routing path $L_i(V^i_l, E^i_l)$ $(0 \leq i < |LRPs|)$ in $LRPs$, let $S^i_l \subset V^i_l$ be the set of nodes which contains only producer $u^0_{P_i}$ and consumers $v^0_{P_i}, v^1_{P_i} \ldots, v^{K_i-1}_{P_i}$. For each routing path L_i, the LRB constraints is performing over its corresponding S^i_l.

1. The performance should not be worse after LRB assignment, that is, $|S^i_l| \leq II$.
2. All the nodes inside S^i_l can be assigned to the same PE without conflict, they should be distributed in different modulo time, that is, $\forall\ j \in [0, II)$, $|M_j(S^i_l)| \leq 1$.
3. There are enough interconnection resources to support the routing between nodes outside V^i_l and same PE nodes in S^i_l. Let V_q contains all such node $v \in V_m$ and $v \notin V^i_l$, $\exists u \in S_i$, $e(u,v) \in E_m$ but $e(u,v) \notin E^i_l$ or $e(v,u) \in E_m$ but $e(v,u) \notin E^i_p$, Eq. (3) must be satisfied.

$$\forall\ j \in [0, II), \left|M_j(V_q)\right| \leq \max_{0 \leq i < N_{pe}} \left|Succ(PE^j_i)\right|, \tag{3}$$

where $|Succ(PE^j_i)|$ is the number of the interconnections of PE_i.

4. LRB has enough space to buffer the value for L_i according to:

$$\left\lfloor \frac{LT(L_i)}{II} \right\rfloor + 1 \leq N_l. \tag{4}$$

5. Same as GRB constraint 2, $LT(L_i) > 2$.

OMB constraints:

We use $ORPs$ to represent the set of routing paths assigned to OMB. In order to benefit from OMB assignment, the latency of any routing path $O_i(V^i_o, E^i_o) \in ORPs$ $(0 \leq i < |OPRs|)$, $LT(O_i) > 4$.

Definition 4.2 (Sub-routing path). *Routing path $G_i(V^i_g, E^i_g)$ is the Sub-routing path of $P_j(V^j_p, E^j_p)$ iff: 1. $u^0_{G_i} \in V^j_p$ is the producer of routing path G_i, for $\forall v \in V^j_p$, if $1 < t(v) - t(u^0_{G_i}) \leq LT(G_i)$, $v \in V^i_g$; 2. if $v \in V^j_p$ and $t(v) - t(u^0_{G_i}) = 1$, $v \in V^i_g$ iff v is the routing node in P_j; 3. for $\forall u, v \in V^i_g$, if $e(u,v) \in E^j_p$, $e(u,v) \in E^i_g$. The relationship between G_i and P_j can be represented by $G_i \subseteq P_j$*

Given the routing path set RPs, the buffer allocating problem is to decide the $GRPs$, $LRPs$ and $ORPs$ such that: 1. $\forall G_i(V^i_g, E^i_g) \in GRPs, \exists G_i \subseteq P_j$ and $G_i; \forall L_i(V^i_l, E^i_l) \in LRPs$, $\exists L_i \subseteq P_k$; $\forall O_i(V^i_o, E^i_o) \in ORPs$, $\exists O_i \subseteq P_m$ $(0 \leq j, k, m < |RPs|)$. 2. All the routing paths in $GRPs$ satisfy GRB constraints, every routing path in $LRPs$ satisfies the LRB constraints and every routing path in $OPRs$ satisfies the OMB constraints. $D_b(V_b, E_b)$ is the DFG generated by modifying D_m according to the corresponding rules for every routing path in $GRPs$, $LRPs$ and $ORPs$. The goal is to maximize the number of eliminated routing nodes, $|V_m - V_b|$.

Algorithm 1. Buffer Allocating

```
Input: $D_m(V_m, E_m)$, $II$ and $BI$
Output: $D_b(V_b, E_b)$, $SPT$
1  $FinishAllocation \leftarrow Failed$;
2  $GRPs, LRPs, ORPs \leftarrow \emptyset$;
3  while $FinishAllocation = Failed$ do
4    $D_b \leftarrow D_m$;
5    $RPs \leftarrow GenerateRoutingPaths(D_b)$;
6    for each $P_i \in RPs$ of $D_b, 0 \leq i < |RPs|$ do
7        if $LRB\_constraints\_checking(S^i_p, II, BI) = True$ then
8            $LRPs \leftarrow LRPs \cup P_i$;
9            $RPs \leftarrow RemovePath(RPs)$;
10       end
11   end
12   $GRPs, LRPs, RPs \leftarrow AssignGRB(RPs, BI, GRPs, LRPs)$;
13   for each $P_i \in RPs$ of $D_b, 0 \leq i < |RPs|$ do
14       $LRPs, ORPs \leftarrow CombAssign(P_i, II, BI)$;
15   end
16   $LRPs \leftarrow MergeLRBPaths(D_b)$;
17   if $\lceil |V_b|/N_{pe} \rceil > II$ then
18       $II \leftarrow II + 1$;
19       $continue$;
20   else
21       $D_b \leftarrow GRBAssignRule(GRPs)$;
22       $D_b, SPT \leftarrow LRBAssignRule(LRPs)$;
23       $D_b \leftarrow OMBAssignRule(ORPs)$;
24       $FinishAllocation = Success$;
25   end
26 end
```

4.2.2 Buffer Allocating Overview

The buffer allocating heuristic is shown in Algorithm 1, the input of the algorithm is the modified DFG D_m after routing paths sharing, II, and buffer information BI such as N_g, N_l

(meaning can be referred from Table 2). The output is the modified DFG D_b after buffer allocating and a same PE table (SPT) structure which specifies the operations that must be mapped to the same PE due to the LRB constraints. The SPT is one of the input to the spatial mapping process which will be illustrated in Section 5.

The algorithm first extracts all the routing paths RPs from DFG (line 5). Before assigning paths to GRB, the $LRPs$ collects all the LRB routing paths from RPs by performing $LRB\_constraints\_checking$ over each routing path in RPs, if P_i satisfies the LRB constraints, it will be added into $LRPs$ and removed from RPs (line 6 to 11). The $LRB\_constraints\_checking$ method is according to LRB constraints in Section 4.2.1, problem formulation. The $GRPs$ then collects all the GRB routing paths through $AssignGRB$ function and update RPs and $LRPs$ (line 12). The $LRPs$ again collects the LRB routing paths from remaining paths in RPs. If any $P_i \in RPs$ passes the LRB constraints checking (line 7), it will be put into $LRPs$. Otherwise the algorithm finds the best combinational buffer assignment scheme (line 14) and update both $LRPs$ and $ORPs$. All the routing paths in $LRPs$ must be merged or decomposed if some of them shares the common node or nodes (line 16). If buffer allocating is not able to release enough PEs so that the theoretical performance can reach II, II will be increased by 1 and the allocating process is repeated until the performance reaches II (line 17 to 19). Otherwise the algorithm finally assign paths in $GRPs$, $LRPs$ and $ORPs$ to GRB, LRB and OMB according to their corresponding DFG modification rules (line 21 to 24).

Algorithm 2. AssignGRB

```
    Input: $RPs, BI, GRPs, LRPs$
    Output: $GRPs, LRPs$
1   $RPs \leftarrow SortRoutingPaths(RPs)$;
2   $GRPs \leftarrow InitializeGRPs(RPs)$;
3   $max\_sl \leftarrow getMaxSharedLevel(RPs)$;
4   for $max\_sl \geq$ shared_level $> 0$ do
5       for each routing path $P_i(V_p^i, E_p^i)$ in ordered $RPs$ do
6           for each $u_j^i$ in $V_p^i$, $j$ from 1 to $J^i$ do
7               if $u_j^i.sl =$ shared_level then
8                   $G_i \leftarrow extendPath(G_i, u_j^i)$;
9                   $L_i \leftarrow OCRP(G_i, P_i)$;
10                  if $GRB\_constraints(GRPs, II) = False$ then
11                      $G_i \leftarrow remove(G_i, u_j^i)$;
12                      $RPs \leftarrow UpdateRPs(GRPs)$;
13                      return $GRPs, RPs$;
14                  end
15                  if $LRB\_constraint\_checking(S_l^i) = True$ then
16                      $LRPs \leftarrow LRPs \cup L_i$;
17                      $RPs \leftarrow RemovePath(P_i)$;
18                  end
19              end
20          end
21      end
22  end
23  $RPs \leftarrow updateRPs(GRPs)$;
24  return $GRPs, RPs$;
```

4.2.3 GRB Assignment

Our GRB assignment approach is shown in Algorithm 2. All the routing paths in RPs are sorted in decreasing order

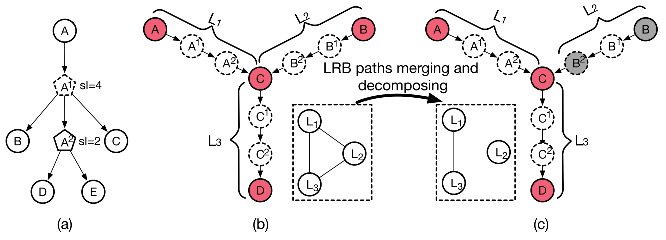


Fig. 8. (a) Example of highly shared routing path, value A is consumed by four operations (b) three LRB paths that have common node C, (c) L_1 and L_3 are merged, L_2 removes C from itself.

according to the maximum *shared_level* of their routing node (line 1). At first, $GRPs$ is initialized by setting each path $G_i(V_g^i, E_g^i)$ where $V_g^i = \{u_{P_i}^0, u_{P_i}^1\}$ and $E_g^i = \{e(u_{P_i}^0, u_{P_i}^1)\}$ (line 2). We gradually extend the path G_i by inserting one routing node in P_i to be the routing node in G_i according to the decreasing order of their *shared_level* (line 8) and checks the GRB constraints over $GRPs$ until GRB do not have enough space (line 10 to line 12). The algorithm will stop the extension of G_i when its *overlapped complemented routing path* (OCRP) L_i satisfies the LRB constraints.

Definition 4.3 (Overlapped complemented routing path). *Given routing path* $P_i^1(V_{p^1}^i, E_{p^1}^i) \subseteq P_i(V_p^i, E_p^i)$ *and* $u_{P_i^1}^0 = u_{P_i}^0$, $P_i^2(V_{p^2}^i, E_{p^2}^i)$ *is the overlapped complemented routing path of* P_i^1, *if and only if* $P_i^2 \subseteq P_i$, $E_{p^1}^i \cap E_{p^2}^i = \emptyset$, $E_{p^1}^i \cup E_{p^2}^i = E_p^i$, $V_{p^1}^i \cup V_{p^2}^i = V_p^i$ *and* $V_{p^1}^i \cap V_{p^2}^i = u_{P_i}^j$ *where* $j = LT(P_i^1)$.

The insight of this heuristic has two prospectives:

1. Assigning routing node with higher *shared_level* to GRB in priority is because we want to assign the highly shared routing paths to GRB. Usually these paths are difficult to be efficiently assigned by LRB and OMB such as Fig. 8a.
2. Using the fine-grained approach which assign one routing node each time instead of one routing path is because routing paths with same shared level should have equal opportunity to utilize the GRB resource.

4.2.4 Combinational Buffers Assignment

Definition 4.4 (Flattened complemented routing path). *Given routing path* $P_i^1(V_{p^1}^i, E_{p^1}^i) \subseteq P_i(V_p^i, E_p^i)$ *and* $u_{P_i^1}^0 = u_{P_i}^0$, $P_i^2(V_{p^2}^i, E_{p^2}^i)$ *is the flattened complemented routing path (FCRP) of* P_i^1, *if and only if* $P_i^2 \subseteq P_i$, $V_{p^1}^i \cup V_{p^2}^i = V_p^i$, $V_{p^1}^i \cap V_{p^2}^i = \emptyset$, $E_{p^1}^i \cap E_{p^2}^i = \emptyset$ *and* $E_{p^1}^i \cup E_{p^2}^i = E_p^i - e(u_{P_i}^j, u_{P_i}^{j+1})$, *where* $j = LT(P_i^1)$.

The algorithm of the combinational buffers assignment is shown in Algorithm 3. For each remaining routing path P_i in RPs, we perform the exhausted searching of all the combinations of assigning P_i to LRB and OMB. The routing path P_i is decomposed to two complemented routing paths P_i^1 and P_i^2. And we evaluate the total saving number of the routing nodes under different combinational buffer assignment schemes and select the one which can save maximum number of the routing nodes (line 9). It should be noticed that if routing path P_i is assigned to two LRBs from different PEs, it is decomposed to two LRB routing paths P_i^1 and P_i^3 (line 7), and P_i^3 is the flattened complemented routing path of P_i^1. This is because if P_i can not be assigned using LRB from single PE, at least two LRBs from different PEs should be leveraged, which needs an routing node from one PE to another. Recall example in Fig. 4j, there is additional routing from A^2 to A^3.

Algorithm 3. CombAssign

```
   Input: $P_i, BI, II$
   Output: $LRPs, ORPs$
1  for each $P_i \in RPs\ of\ D_b, 0 \le i < |RPs|$ do
2      for $LT(P_i^1) = 0\ to\ LT(P_i), P_i^1 \subseteq P_i$ and $u^{P_i^1} = u^{P_i}$ do
3          $P_i^2 \leftarrow OCRP(P_i^1, P_i)$;
4          $P_i^3 \leftarrow FCRP(P_i^1, P_i)$;
5          $save[] \leftarrow calculateLRB(P_i^1) + calculateOMB(P_i^2)$;
6          $save[] \leftarrow calculateOMB(P_i^1) + calculateLRB(P_i^2)$;
7          $save[] \leftarrow calculateLRB(P_i^1) + calculateLRB(P_i^3)$;
8      end
9      $LRPs, ORPs \leftarrow SelMaxSave(save[])$;
10 end
11 return $LRPs, ORPs$;
```

Algorithm 4. LRB Paths Merging and Decomposing

```
   Input: $LRPs, BI, II$
   Output: $LRPs$
1  $CCs \leftarrow BuildConnectedComponents(LRPs)$;
2  for each $CC_i \in CCs, 0 \le i < |CCs|$ do
3      while $LRB\_constraint\_checking(S_{merge}^{CC_i}) = False$ do
4          $CC_i' \leftarrow MaxMerge(CC_i)$;
5          $CC_i, LRPs \leftarrow Decompose(CC_i', CC_i)$;
6      end
7  end
8  return $LRPs$;
```

4.2.5 LRB Paths Merging and Decomposing

Recall the LRB constraints, for each LRB routing path $L_i \in LRPs$ $(0 \le i < |LRPs|)$, the LRB constraints are performed over S_l^i. We construct an undirected graph $C(V_c, E_c)$, where each routing path L_i is abstracted to a vertex $v_{L_i} \in V_c$, and for any two LRB routing paths L_i, L_j $(i \ne j)$, the undirected edge $e(v_{L_i}, v_{L_j}) \in E_c$ if and only if $S_l^i \cap S_l^j \ne \emptyset$. This abstracted graph C contains a set of multiple weak connected components CCs (line 1). For each weak connected component $CC_i(V_{CC}^i, E_{CC}^i) \in CCs$, $(0 \le i < |CCs|)$ the LRB constraints performs over the merged set $S_{merge}^{CC_i}$, where $S_{merge}^{CC_i}$ is the union set of all the S_l^i corresponding to L_i, such that $v_{L_i} \in V_{CC}^i$.

For example in Fig. 8b, three LRB paths, L_1, L_2 and L_3 share the common node C, the connected component is shown in dash frame, and the LRB constraints should perform over $S_{merge} = S_1 \cup S_2 \cup S_3$. When S_{merge} can not pass the LRB constraints, strategy should be used to decide which paths can be merged to pass the LRB constraints and which path should be decomposed with the other paths. We refer these two steps as LRB paths merging and decomposing. The $MaxMerge$ function performs the combinational search for the maximum sub-connected component CC_i' from CC_i so that: 1. All the routing paths within CC_i' can be merged so that the $S_{merge}^{CC'}$ satisfies the LRB constraints. 2. After CC_i' is cut from CC_i, the remaining graph CC_i'' is also a connected component. The $Decompose$ function implements the process of cutting CC_i' from CC_i. Every routing path in CC_i'' removes the node, which is the common with the merged paths in CC_I', from its own path. We iteratively performs merging and decomposing until all the merged routing paths are able to pass the LRB constraints. In Fig. 8c, after LRB path merging and decomposing, P_1 and P_3 are merged and P_2 is decomposed from these paths.

Algorithm 5. Interconnection_Constraints_Solving

```
   Input: $D_m$, $R_{II}$ and $II$
   Output: $D_m$
1  for $L - 1 \ge i \ge 0$ then
2      for $1 \le n \le Thredshold$ do
3          for every $S(i) \in C(T_i(V_m), n)$ do
4              while $IC\_checking(S(i), II) = Failed$ do
5                  if $n = 1$ then
6                      if $v \in S(i)$ is OriginalNode then
7                          $D_m \leftarrow RescheduleNode(D_m)$;
8                      end
9                      if $v \in S(i)$ is RoutingNode then
10                         $D_m \leftarrow SplitSRNode(D_m)$;
11                     end
12                 else
13                     if $I_{pre}(S(i))$ has routing node then
14                         $D_m \leftarrow SplitSRNode(D_m)$;
15                     end
16                     if $I_{pre}(S(i))$ doesn't have routing node then
17                         $D_m \leftarrow RescheduleNode(D_m)$;
18                     end
19                 end
20             end
21         end
22     end
23 end
24 $II_{new} \leftarrow \lceil |D_m|/N_{pe} \rceil$;
25 return $II_{new}, D_m$
```

4.3 Interconnection Constraints Solving (ICS)

The ICS checks whether there is enough interconnection resources to support the spatial mapping from D_m to TEC and then reschedule operations if necessary. Performing ICS can effectively prevent compiler from trapping into time consuming spatial mapping process caused by unawareness of the interconnection constraints.

Our ICS not only checks the interconnection constraints of every single node, for example, the fanout constraints evaluated in most existing mappings [47], [61]. It also checks the interconnection constraints between multiple nodes scheduled at the same time. The algorithm of ICS is shown in Algorithm 5. We start with checking the interconnection constraint for single node (n = 1), then for any two nodes (n = 2) scheduled at the same time, then for three and so on. Due to the interconnection limitation of the hardware model, the threshold of n is no larger than 5 under mesh or torus topology. For any combination of selecting n nodes from all the nodes that are scheduled at time i ($S(i)$ in line 3). The $IC\_checking$ function checks the interconnection constraints over S_i according to Eq. (5). The meaning of each symbol can be referred from Table 2. The routing constraints can be illustrated in this way, $\forall\ i, m = (i \bmod II)$ and n $(1 \le n \le |T(i)|)$,

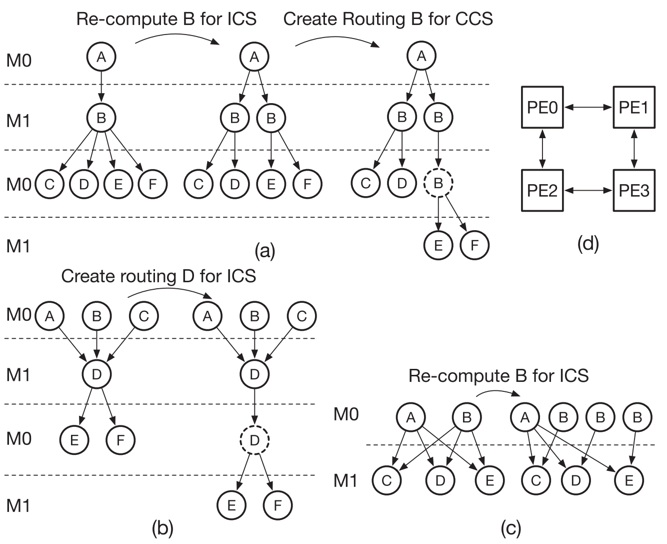


Fig. 9. (a) Re-compute operation B for single node ICS and create routing operation of value B in the critical path for CCS in example DFG of [47], (b) Create routing operation of value D for ICS, (c) Re-compute operation B for multiple nodes ICS, (d) 2x2 PE interconnection topology.

$\forall S(i) \in C(T(i), n)$ must satisfies Eq. (5). Furthermore, when $II \leq 2$, Eq. (6) must be satisfied. The meaning of the symbols in the equation can be referred from Table 2.

For any operation does not pass the interconnection constraints checking, we perform *RescheduleNode* on the original node or *SplitSRNode* on the routing node. If there is routing node added in the critical path, the schedule time of all the node must be re-calculated and our compiler will go back to the start of the temporal mapping flow and go through these sub-algorithms again (Fig. 7).

There are two ways of rescheduling in *ReschduleNode* function, one is re-compute [47], another one is creating a routing operation. For example, in Fig. 9a, operation B has four fanout nodes. Existing work such as EPIMap [47] is able to perform ICS over single node. According to Fig. 9 the maximum number of PEs that have interconnection with single common PE is 3. Whereas operation B has four fan-outs, which means the interconnection resources in CGRA is not enough to directly map this DFG and B is re-computed.

Our work is able to find out the interconnection constrains between multiple nodes, in Fig. 9c A and B are scheduled at the same modulo time, and they have three common successors. Accordingly, in CGRA, there should exist two PEs which have interconnection with three common PEs. However, the maximum number of PEs that connects to any two common PEs is 2. The ICS will solve this problem by re-computing B 3 times.

Re-computing is not always the correct option for ICS, if a node has multiple inputs, re-computing will lead to double scale of the input edges. We provide another way by creating routing node for this operation. An example is shown in Fig. 9b, as operation D has three inputs, routing operation D (dash circle) is created.

$$\begin{aligned} |I_{pred}(S(i))| &\leq \max_{\forall S_R(m) \in C(R_m(V_R),n)} |I_{pred}(S_R(m))| \\ |I_{succ}(S(i))| &\leq \max\nolimits_{\forall S_R(m) \in C(R_m(V_R),n)} |I_{succ}(S_R(m))| \end{aligned} \tag{5}$$

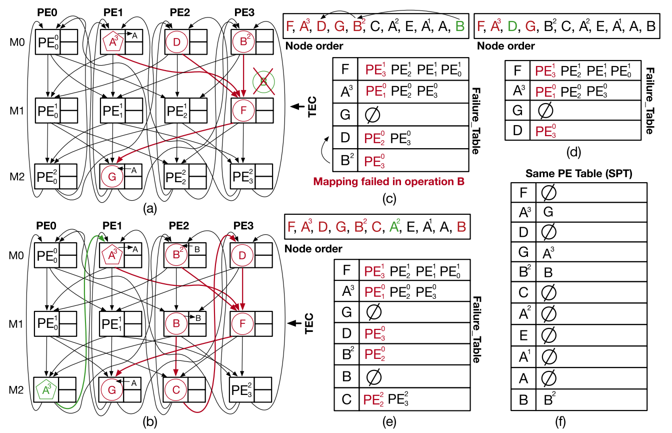


Fig. 10. (a) Status of TEC when mapping B; (b) status of TEC when mapping A^3; (c) status of nodes in order list and Failure Table (FT) when mapping B; (d) status of nodes in order list and FT when mapping D; (e) status of nodes in order list and FT when mapping A^3. (f) status of SPT.

$$|I_{pred}(S(i)) \cup I_{succ}(S(i))| \leq \max_{\forall S_R(m) \in C(R_m(V_R),n)} |I_{pred}(S_R(m))|. \tag{6}$$

It should be noticed that in *LRB_constraint_checking* method, we also check the interconnection constraint for LRB constraint, this is not included in the ICS phase.

4.4 Computational Constraints Solving (CCS)

Finally, the CCS algorithm checks whether there are enough PEs to support the mapping of operations scheduled at the same time. Under the modulo scheduling scheme, the way of checking computational constraints should be $\forall j \in [0, II), M_j(V_m) \leq N_{pe}$. The number of operations that is scheduled in a specific modulo time should not be larger than the available computational resources. When there exists modulo time j, where $M_j(V_m) > N_{pe}$ we select node with higher schedule time mobility and re-assign their schedule time from modulo time j to i, where $i \neq j$ and $M_i(V_m) < N_{pe}$. If there is no node with schedule mobility larger than one, additional routing node will be added to the critical path. In this way, the schedule time of all the operations should be analyzed again, our compiler will go back to the start point of the temporal mapping flow and go through these sub-algorithms again. For example, in the middle DFG of Fig. 9a, there are five operations in M0 (A, C, D, E, F), but there are only 4 PEs. The CCS add routing node B so that E and F are scheduled one cycle later from modulo time 0 to 1, and both the number of operations in M0 and M1 are 4 (right most DFG in Fig. 9a).

5 Spatial Mapping

5.1 Mapping Overview

The problem of spatial mapping can be formulated as finding the subgraph of II extended TEC, $R_{II}(V_R, E_R)$, which is isomorphic to $D_m(V_m, E_m)$, so that each node $v \in V_m$ scheduled at time $t(v)$ must be mapped on $PE^{m(v)}$ in V_R.

Algorithm 6 shows the complete mapping process. The input of the spatial mapping is the modified DFG D_m generated through the temporal mapping flow, II, and same PE table (SPT). The SPT is a table structure recording the set of operations that must be mapped to same PE according to LRB constraints generated from temporal mapping. Fig. 10f is an example of SPT generated from the modified DFG in Fig. 6b. It shows operation A^3 and G, B and B^2 must be mapped to same PE. If the operation do not have same PE mapping constraint, its same PE set is empty in SPT.

Our mapping algorithm starts with calculating a minimal II (line 1). Then, it iteratively performs the temporal mapping (line 7) and spatial mapping (line 12 to line 57) until the mapping success. When the mapper is not able to map all the nodes of D_m, it repeats the temporal and spatial mapping process with a compromised II (line 51 to 54).

5.2 The Spatial Mapping Algorithm

The spatial mapping algorithm places and routes each operation from $D_m(V_m, E_m)$ to $R_{II}(V_r, E_r)$ according to the topology node order list (line 10). When the algorithm is to find a PE slot that is able to be mapped for a node, it first find out all the PE slot candidates and sort them according to the priority (line 25). The principle of selecting the candidate is that, if the mapping node is v in V_m, and for any node u in the set of operations that have already been mapped to $PE(u) \in V_r$, the candidate PE slot of node v, $PE(v)$, must have interconnection with $PE(u)$:

If e(u, v) $\in E_m$, then e($PE(u)$, $PE(v)$) $\in E_r$; if e(v, u) $\in E_m$, then e($PE(v)$, $PE(u)$) $\in E_r$.

From the candidates, we select the PE slots with the highest priority, create a mapping pair between this PE and node v, and save all the candidates into the failure table (line 27, 28). After the node v is successfully mapped, it then check SPT to see wether there exist the other node that must be mapped onto the same PE of node v according to LRB constraint, and map them onto the same PE if $SPT[v] \neq \emptyset$ (line 29).

The priority function is based on Eq. (7), where N_r denotes the number of PE nodes in TEC, R_{II}, that the candidate PE node v_r can reach through the interconnection between v_r and the unplaced PE nodes, and N_d is number of the unplaced predecessors and successors of the mapping node v. If N_r is smaller than N_d, then placing v on v_r will cause some of v's unplaced predecessors or successors failing to place, due to the lack of interconnection resources. If N_r is equal to or larger than N_d, then we choose slot v_r whose $\frac{N_d}{N_r}$ is the largest of all. The insight behind this selection is to leave resources as many as possible for the rest unmapped nodes.

$$Priority\ of\ candidate\ PE = \begin{cases} 0 & \text{if } N_r < N_d \\ \frac{N_d}{N_r} & \text{if } N_r \geq N_d \end{cases} \tag{7}$$

Backtracking. When mapping fails on a certain node v, our algorithm is able to backtrack to the previous node u that may influence the mapping of the failed node (line 31). It erases the saved information in FT and mapping pair (MP) of all the operations after u in V_{order} and select another PE candidates in FT for operation u (line 35 to 39). When there is no node to backtrack, the mapping under V_{order} fails. To further improve the mapping successful rate, we also develop the second insurance plan, which is to remap the DFG according to another topology node order list (line 48, 49).

Algorithm 6. Mapping Algorithm

```
   Input: Original data flow graph $D$, and CGRA size.
   Output: Mapping pairs MP
1  $MII \leftarrow Calculate\_MII(D, CGRASize)$;
2  $II \leftarrow MII$;
3  $FT \leftarrow Initialize\_FailureTable()$;
4  $MP \leftarrow Initialize\_MappingPairs()$;
5  $V_{order} \leftarrow \emptyset$;
6  repeat
7     $D_m, II, SPT \leftarrow Temporal\_Mapping(D, II)$;
8     $R_{II} \leftarrow Construct\_TEC(CGRASize, II)$;
9     if $V_{order} = \emptyset$ then
10       $V_{order} \leftarrow OrderNode(D_m)$;
11    end
12    while $|MP| \neq |V_m|$ do
13       $pointer \leftarrow V_{order}.begin()$;
14       while $pointer \neq V_{order}.end()$ do
15          $v \leftarrow pointer$;
16          if $MP.already\_mapped(v) = True$ then
17             $pointer \leftarrow to\_next\_node()$;
18             $continue$;
19          end
20          if $FT[v] \neq \emptyset$ then
21             $MP \leftarrow Insert\_MP(v, FT[v][0])$;
22             $pointer \leftarrow pointer.forward\_to\_next\_node()$;
23             $continue$;
24          end
25          $v.candidates \leftarrow OrderSlotsByPriority(R_{II}, v)$;
26          if $v.candidates \neq \emptyset$ then
27             $MP \leftarrow Insert\_MP(v, v.candidates[0])$;
28             $FT[v] \leftarrow v.candidates$;
29             $MP \leftarrow Map\_nodes\_in\_SPT(SPT[v])$;
30          else
31             $v_{bk} \leftarrow Node\_To\_Backtrack(v, V_{order}, FT)$;
32             if $v_{bk} = NULL$ then
33                $break$;
34             else
35                $FT \leftarrow Erase\_FT\_Behind\_Node(v_{bk}, FT)$;
36                $FT \leftarrow Erase\_MP\_Behind\_Node(v_{bk}, MP)$;
37                $FT[v_{bk}] \leftarrow FT[v_{bk}] - FT[v_{bk}][0]$;
38                $pointer \leftarrow pointer.backward\_to(v_{bk})$;
39                $continue$;
40             end
41          end
42          $pointer \leftarrow pointer.forward\_to\_next\_node(V_{order})$;
43       end
44       if $|MP| = |V_m|$ then
45          return $MP$;
46       else
47          if $reorder\_time \leq |V_{order}|$ then
48             $V_{order} \leftarrow Update\_Node\_Order(D_m)$;
49             $reorder\_time \leftarrow reorder\_time + 1$;
50          else
51             $II \leftarrow II + 1$;
52             $Clear\_FailureTable()$;
53             $Clear\_MappingPairs()$;
54             $break$;
55          end
56       end
57    end
58 until $|MP| = |V_m|$ or $II > L$
```

Reordering. The order of mapping each node is an important factor that influence the successful rate of spatial mapping. This is because the modified DFG may contain some special nodes that is hard to be mapped. For example, the node with high degree. These nodes have heavy requirement on the interconnection resources. Another special set of nodes are those mapped onto the same PE, because these nodes have special requirements on the modulo time of PE slot. If we map these special nodes in the latter order, there will be high possibility that the mapping fails at these nodes. In this way, when backtracking also fails to generate the valid mapping, we will generate a new topology order which is different with previous orders and remap the DFG under the same II. We generate the different topology orders of nodes by performing BFS over different source nodes inside DFGs. If all the node order is tried and there is no valid mapping, we increase II by 1, clear the previous mapping status and restart the temporal and spatial mapping process (line 51 to 54).

5.3 Spatial Mapping Example

We use an example of mapping the modified DFG in Fig. 6b onto TEC with II=3, (Fig. 10a). In this TEC, we use PE_i^m to represent the PE slot of PE_i in modulo time m. The node order list is generated from the BFS topology order starts from operation F, which is the same as GraphMinor [48]. In Figs. 10c, 10d, and 10e, in node order, red color denotes the nodes that have already been mapped, green color is the mapping node, and black color ones are the unmapped nodes. The mapping fails at operation B which has the same PE mapping constraint with B^2, after B^2 is mapped to PE_3^0, B should be mapped onto the same PE, PE_3^1. However, it is conflict with the already mapped operation F. According to the backtracking algorithm, it backtrack to operation D, all the saved candidates of the operations after D in the node order list are erased from the failure table. The status of the failure table changes from Figs. 10c and 10d. The operation D is remapped to another candidate PE_3^0 and then, operation B is successfully mapped onto PE2. The final mapping result is shown in Fig. 6c.

In conclusion, we use a fast greedy based approach to map each operation in forward order of the order list. If mapping fails, we use backtracking and reordering to guarantee the mapping successful rate.

6 Evaluation

6.1 Methodology

To evaluate the performance and compilation time of our mapping algorithm, we implement a full stack compiler framework based on LLVM [69] platform. The compiler extracts the loops from the program of different applications and compiles them into the context of CGRA. A runtime simulator is build to evaluate the efficiency of the compiler. We select two state-of-the-art CGRA mapping algorithms as the baseline, REGIMap [52] and RAMP [55]. These two mapping algorithms are two representative ones which decomposes the temporal and spatial process, and have been proven to be competitive in both performance and compilation time over the other CGRA MS algorithms, especially the representative simulated annealing based algorithm DRESC [60]. Our compilation flow is able to deploy the extended C language based loop kernel onto the real energy-efficiency dynamic reconfigurable fabric [70] which target on Berkeley 13 drawfs [71].

Compilation. We modify the Clang (front-end of LLVM) by adding keywords to specify the loop kernels in the grogram. Then, we split the loop kernel from the main program in LLVM IR level and the DFG of the extracted loop is generated from LLVM IR. Our mapping algorithm is implemented and integrated into LLVM backend as passes. We also port the open-source code of REGIMap and RAMP (https://github.com/cmlasu/ccf) into the same LLVM framework with our mapping algorithm so that they are able to map the same DFG and their performance and compilation time can be compared fairly. Experiments are conducted on CGRAs with 4x4 PEs and different GRB and LRB sizes. Since PEs are connected in a 2D torus network, all PEs can route data to and access data from neighboring PEs or themselves. The first PE and last PE at each column or row are also neighbors. All experiments are carried out on the same Intel Core i5 machine with CPU frequency of 3.60 GHz.

Test Suite. We take 28 computational intensive loop kernels from different benchmark suites and applications. Some of loops are directly extracted from existing benchmark suite such as EEMBC, MediaBench [72], MiBench [73], MachSuit [74] and Polybench [75]. Others are modified from different application domains, including digital signal processing (DSP), graph searching (Graph), dynamic programming (DP) and computer vision (CV). The size of their loop body varies from 17 to 154 operations. In order to reflect the regularity of the data flow graph, we calculate the number of routing paths (RPs) and number of high degree nodes inside DFG. The RPs shows the number of producer-consumer relationship whose lifetime is larger than 1. More RPs exists in DFG means higher pressure for the compiler to manage the buffer resources. The degree of each operation means the input edge and output edge number of a node. Typically, the high degree nodes are those who have three inputs (sel instruction) or multiple outputs (larger than 1). More high degree nodes inside DFG means higher for the compiler to manage the interconnection resources. Table 3 shows the features of all the loops we select.

6.2 Performance

We evaluate the performance of mapping algorithms over CGRA with different LRB sizes from 2 to 8, and GRB size is 4. For the compilation time, we set one week as the upper bound. If the mapper cannot finish mapping within one week, it will stop the mapping even though the mapper may find the valid mapping without time limitation. The reason of selecting 2, 4 and 8 as the LRB size is because for most of the loop kernels we select, there is almost no difference in performance when LRB is increased from 4 to 8 when using our baseline mapping. GRB size is set to 4 is because we mainly want to test the performance of our mapping when GRB buffer resource is limited.

6.2.1 LRB Size Influence

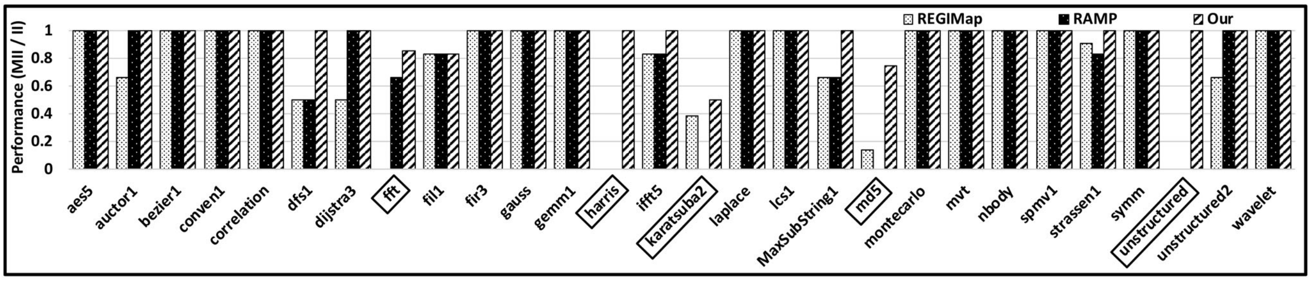
Fig. 11 shows the performance of REGIMap, RAMP and our mapping approach. The Y axis is the ratio between

TABLE 3
Loop Kernel Features

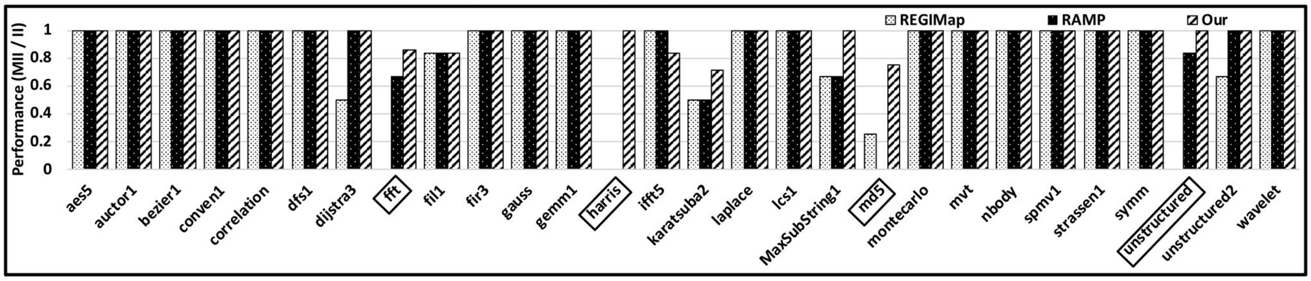
Kernel	OPs	RPs	Degree≥4	Benchmark	Kernel	OPs	RPs	Degree≥4	Benchmark
aes5	78	0	0	EEMBC	karatsuba2	78	14	10	DSP
autocr1	32	1	0	EEMBC	laplace	32	0	0	DSP
bezier1	41	2	4	EEMBC	lcs	48	0	2	DP
conven1	41	3	3	EEMBC	MaxSubString1	18	3	2	DP
correlation	17	1	4	PolyBench	md5	47	11	4	MediaBench
dfs1	40	4	7	Graph	montecarlo1	41	1	2	EEMBC
dijstra	33	5	5	Mibench	mvt	146	0	0	PolyBench
fft	92	8	2	EEMBC	nbody	54	3	1	MachSuit
fil	79	0	0	EEMBC	spmv	52	0	0	MachSuit
fir	18	0	0	EEMBC	strassen	154	0	0	DSP
gauss	20	1	0	DSP	symm	26	0	0	PolyBench
gemm1	22	0	0	PolyBench	unstructured	77	6	1	EEMBC
harris	116	7	10	CV	unstructured2	23	2	1	EEMBC
ifft5	70	4	4	EEMBC	wavelet	68	0	0	DSP

minimum II and actual II, which reflects the percentage of the theoretical best performance. Kernels specified with red rectangle frame are those can not be compiled within one week time budget.

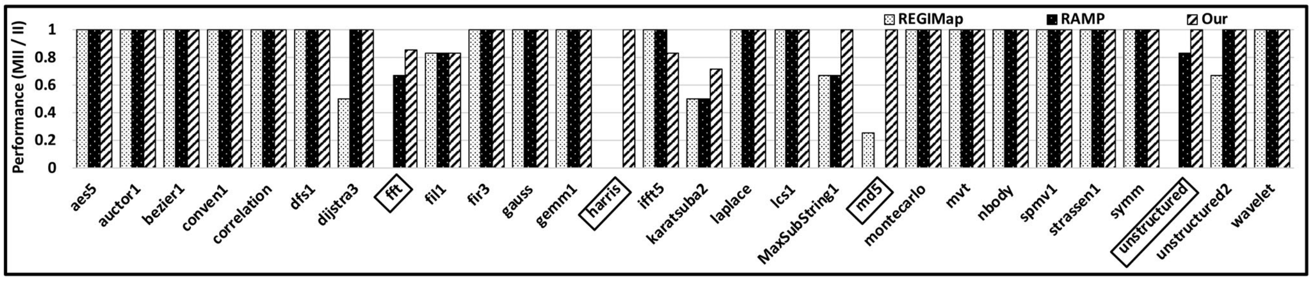
REGIMap versus Our. We can observe from Table 4 that when LRB=2, REGIMap gets 0.84 normalized performance among the successfully compiled loops in average, whereas our mapping (Our_complete) is able to reach 0.95 in average when GRB=0, and 0.96 when GRB=4. When LRB grows from 2 to 4 and 8, the average performance of REGIMap increase from 0.84 to 0.89, and our approach increase to 0.97 and 0.98 respectively. When LRB grows from 2 to 4 and 8, the average performance of REGIMap increase from 0.84 to 0.89. Out of 28 loop kernels, REGIMap is able to map 15 kernels with optimal performance. 13 kernels can not be mapped with optimal performance, and 3 of them can not be mapped within the given compilation time budget. As LRB size grows, the performance of 6 out of 10 kernels are improved. This means LRB resource is important for these 6 loop kernels, but is not useful for the rest of 4 kernels



(a) Performance of compiled loops using different compile techniques for 4x4 CGRA with LRB=2 at each PE and GRB=4



(b) Performance of compiled loops using different compile techniques for 4x4 CGRA with LRB=4 at each PE and GRB=4



(c) Performance of compiled loops using different compile techniques for 4x4 CGRA with LRB=8 at each PE and GRB=4

Fig. 11. The performance comparison between different mapping algorithms over 4x4 CGRA with different LRB sizes.

TABLE 4
The Performance, Compilation Successful Rate and Compilation Time Comparison Between Our, REGIMap and RAMP

	LRB=2, GRB=0		LRB=2, GRB=4		LRB=4, GRB=4		LRB=8, GRB=4	
Average performance of the successfully compiled loops (MII/II)	REGIMap	Our	REGIMap	Our	REGIMap	Our	REGIMap	Our
	0.84	0.95	0.84	0.96	0.89	0.97	0.89	0.98
	RAMP	Our	RAMP	Our	RAMP	Our	RAMP	Our
	0.93	0.95	0.93	0.99	0.94	0.98	0.94	0.98
Compilation successful rate	REGIMap	89.3%	REGIMap	89.3%	REGIMap	89.3%	REGIMap	89.3%
	RAMP	85.7%	RAMP	85.7%	RAMP	92.9%	RAMP	92.9%
	Our	100%	Our	100%	Our	100%	Our	100%
Compilation time of the successfully compiled loops (sec)	REGIMap	Our	REGIMap	Our	REGIMap	Our	REGIMap	Our
	3062	11.29	3061	6.10	230	9.02	220	9.04
	RAMP	Our	RAMP	Our	RAMP	Our	RAMP	Our
	5540	14.15	5536	5.85	9563	8.69	8165	8.71

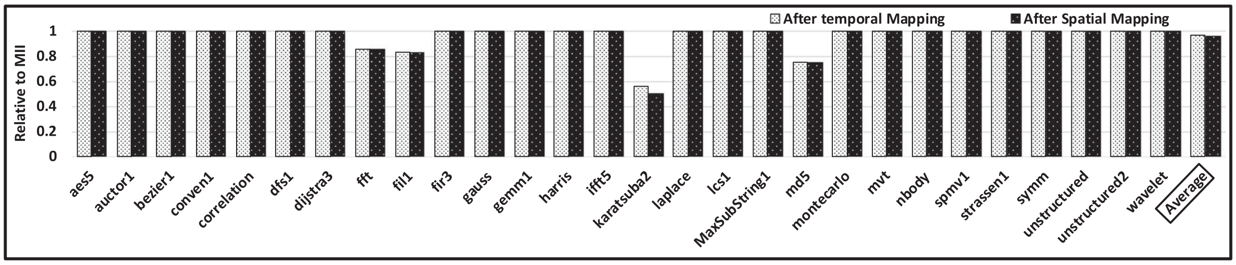


Fig. 12. The theoretical performance after temporal mapping and the final performance after spatial mapping relative to MII.

dijstra3, *fil1*, *Maxsubstring* and *unstructured2*. These kernels contains irregular routing paths which can not be routed using LRB. REGIMap choose PE as the routing media, this is why the performance of these 4 kernels do not increase as the LRB grows. Another reason is REGIMap does not use GRB as the first routing choice but position it and OMB as the final solution for LRB spilling. This is also one of the reason why REGIMap fails the compilation for 3 kernels no matter how large LRB is.

RAMP versus Our. We then observe from Table 4 that when LRB=2, RAMP fails map four loop kernels. Among all the successfully mapped loops, RAMP is 0.93 of optimal performance whereas our mapping algorithm is 0.95 when GRB = 0 and 0.99 when GRB = 4. When LRB size grows, RAMP is able to get two more mapped loops *karatsuba* and *unstructured*, but still not able to get the mapping for *harris* and *MD5*. The average performance among all the mapped kernels is also increase slightly. Comparing with REGIMap, RAMP shows stronger mapping ability, if we compare them with the successfully mapped kernels, RAMP is better than REGIMap. This make sense because RAMP improves REGIMap by providing more choices of re-scheduling when map fails, and OMB is also be explicitly used during the temporal mapping phase. However, RAMP only select one strategy when it tries to assign buffer resources for routing paths. It does not consider the hybrid use of these routing choices, especially the rational utilization of the valuable GRB resource.

6.2.2 GRB Size Influence

Fig. 13 shows the average performance and compilation time of our mapping over 4x4 CGRA with same LRB size (LRB=2) in each PE and different GRB size (GRB=0, 4, 8, 16). From Fig. 13a we can observe the performance is improved slightly when GRB size increases. This data shows the strong adaptability of our mapping, because even when there is no GRB in CGRA, the performance can still reach 94 percent of the theoretical best performance. However, the shortage of GRB will lead to the increase of compilation time. This is because more routing paths will be assigned to LRB and OMB. Assigning routing paths to LRB brings same PE mapping constraint. This mapping constraint may increase the probability of failure in the forward mapping process and force the spatial mapping to perform backtracking and even reordering in order to keep the performance from decreasing.

6.3 Performance Degradation Analysis

Fig. 12 shows the performance degradation after temporal and spatial mapping process. We can observe that performance is mainly degrade during the temporal mapping process but not spatial mapping process. Except for the loop kernel *karatsuba2*, the spatial mapping process is able to maintain the performance after temporal mapping. This means as long as all the constraints are find out during our temporal mapping process, our spatial mapping algorithm is able to place and route the modified DFG without performance degradation for most of the loop kernels. This data

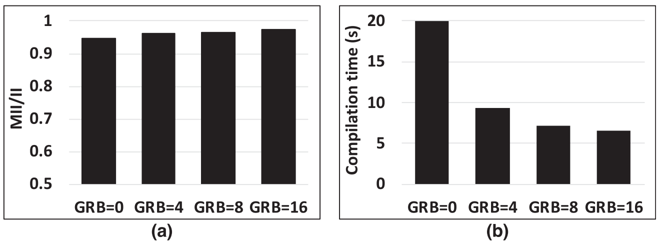
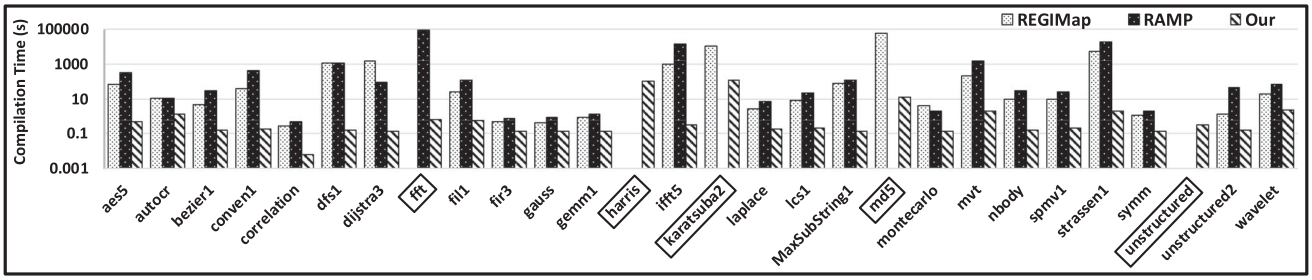
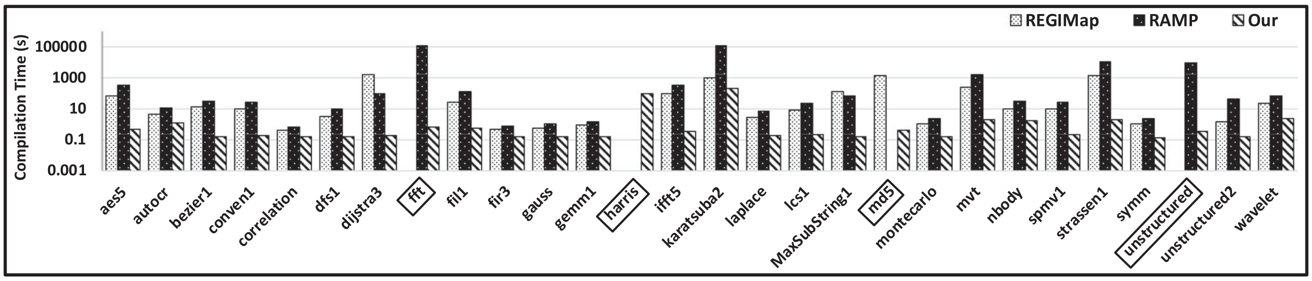


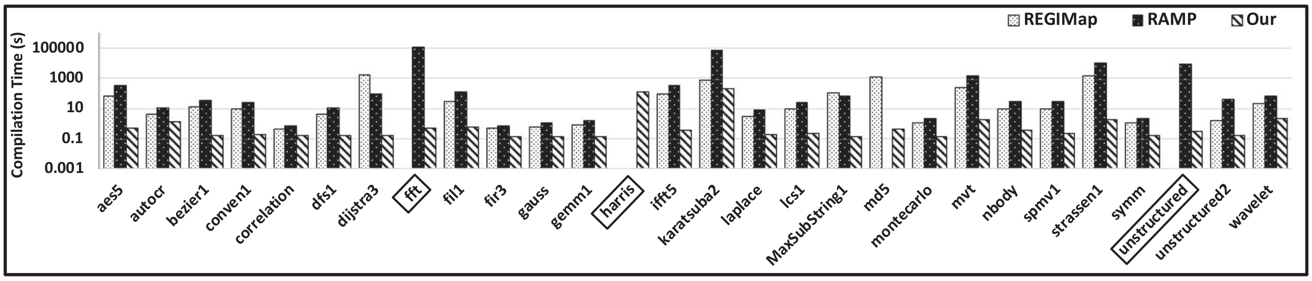
Fig. 13. (a) Performance of our mapping over 4X4 CGRA with LRB=2 at each PE and GRB=0, 4, 8 and 16, (b) compilation time of our mapping over 4x4 CGRA with LRB=2 and GRB=0, 4, 8, 16.



(a) Compilation time of compiled loops using different mapping techniques for 4x4 CGRA with LRB=2 at each PE and GRB=4



(b) Compilation time of compiled loops using different mapping techniques for 4x4 CGRA with LRB=4 at each PE and GRB=4



(c) Compilation time of compiled loops using different mapping techniques for 4x4 CGRA with LRB=8 at each PE and GRB=4

Fig. 14. The compile time of different mapping algorithms over 4x4 CGRA with different LRB size.

also prove our strategy that paying more effort on the temporal mapping phase and using fast spatial mapping algorithm is effective.

6.4 Compilation Robustness Analysis

Even though our algorithm and two baseline algorithms are all decomposing the temporal and spatial mapping process, but the strategy we use is totally different with theirs. REGIMap and RAMP lacks the process of checking important constraints before spatial mapping, and at the same time, they select a very time consuming spatial mapping heuristic (which is O($(V_m \times N_{pe})^8$) according to their paper). On the other side, we think the importance of these two process process should be reversed. The temporal mapping process should be performed more carefully. More precisely, it should make a precise prediction on whether the temporal mapped DFG and be successfully spatially mapped onto CGRA and then make adjustment in time. And for spatial mapping, we do not need to select time consuming heuristic but fast and effective one instead.

Table 4 reflects the compilation successful rate under specific time budget over 4x4 CGRA with different LRB sizes. Our mapping is able to generate all the mappings within time budget. Whereas REGIMap gets 89.3 percent compilation successful rate when LRB = 2, 4 and 8, RAMP gets 85.7 percent when LRB = 2, 92.9 percent when LRB = 4 and 8.

The reason of the failure are different among these kernels. For example, REGIMap fails in *fft*, *harris* and *unstructured*, The reason that REGIMap fails to map is because the DFG of these kernels contains irregular routing paths which is one-producer to multiple consumers. These RPs do not satisfy the LRB constraint. As REGIMap lacks the LRB constraint checking process, it is able to generate the competable graph but the spatial mapping process can not realize that it can not find the maximum clique no matter how many compilation time is given. RAMP has the same problem as REGIMap, even though it has more choices of selecting the buffer resource. However it only use OMB for long routing path when LRB size is not able to buffer the value during its life time. This makes RAMP still traps into infinite iteratively rescheduling and spatial mapping process for *harris* kernel. Fortunately, RAMP is still able to map *MD5*, *karatsuba2* and *unstructured2* when RAMP is given more LRB or more compilation time budget (larger than one week).

6.5 Compilation Time Analysis

Fig. 14 plots the compilation time of loops over 4x4 CGRA with different LRB size. Same as the way of evaluating performance, when comparing the compilation time with every baseline, we only evaluate the compilation time for those loop kernels that are able to be successfully mapped within given time budget (one week). Table 4 summarizes the average data of the compilation time. When LRB = 2, REGIMap takes 3061 seconds and our mapping takes 14 seconds among 25 successfully compiled loop kernels in average. Whereas RAMP takes 5536 and our mapping takes 14 seconds among

24 successfully compiled loops. Our mapping is x219 and x595 times faster than REGIMap and RAMP respectively.

For each loop, when LRB size grows, the compilation time will decrease. However, it makes sense that both of the average compilation time of the successfully compiled loops using RAMP when LRB=4 and 8 is shorter than that when LRB=2. This is because when LRB=2 RMAP fails in two more loop kernels and their compilation time is very large.

Another observation is our mapping approach is able to map most of the kernels within 1 seconds. However, when compiling large and irregular loops or the node number of the modified DFG is very close to the PE resource in time extended CGRA, our spatial mapping may take longer time because we try different topology orders to get the best mapping result.

7 Conclusion

In this paper, we provide a novel methodology of solving the CGRA modulo scheduling problem. We leverage the decomposed mapping policy and argue that temporal mapping flow is far more important than we expect for mappings using decomposed policy. We stress on developing the systematic temporal mapping flow which contains comprehensive buffer allocating heuristic and interconnection and computational resources constraint solving algorithms to improve the mapping performance and guarantee the success rate of the following spatial mapping. Together with the lightweight, fast and efficient spatial mapping algorithm, our approach is able to generate high mapping success rate within certain compilation time budget. Furthermore, the experiment results shows that our mapping is able to generate higher performance and lower compilation time.

Acknowledgments

The authors would like to thank professor Tushar Krishna from Georgia Institute of Technology for the constructive comments on this article.

References

[1] C. Kim, M. Chung, Y. Cho, M. Konijnenburg, S. Ryu, and J. Kim, "ULP-SRP: Ultra low power samsung reconfigurable processor for biomedical applications," in *Proc. Int. Conf. Field-Programmable Technol.*, 2012, pp. 329–334.

[2] M. Karunaratne, A. K. Mohite, T. Mitra, and L. Peh, "HyCUBE: A CGRA with reconfigurable single-cycle multi-hop interconnect," in *Proc. 54th ACM/EDAC/IEEE Des. Autom. Conf.*, 2017, pp. 1–6.

[3] A. Alsolaim, J. Becker, M. Glesner, and J. Starzyk, "Architecture and application of a dynamically reconfigurable hardware array for future mobile communication systems," in *Proc. IEEE Symp. Field-Programmable Custom Comput. Mach.*, 2000, pp. 205–214.

[4] S. C. Goldstein *et al.*, "PipeRench: A coprocessor for streaming multimedia acceleration," in *Proc. 26th Annu. Int. Symp. Comput. Archit.*, 1999, pp. 28–39.

[5] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings, "A reconfigurable arithmetic array for multimedia applications," in *Proc. ACM/SIGDA 7th Int. Symp. Field Programmable Gate Arrays*, 1999, pp. 135–143.

[6] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Trans. Comput.*, vol. 49, no. 5, pp. 465–481, May 2000.

[7] H. Park, Y. Park, and S. Mahlke, "Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2009, pp. 370–380.

[8] L. Liu *et al.*, "An energy-efficient coarse-grained reconfigurable processing unit for multiple-standard video decoding," *IEEE Trans. Multimedia*, vol. 17, no. 10, pp. 1706–1720, Oct. 2015.

[9] R. W. Hartenstein, A. G. Hirschbiel, M. Riedmuller, K. Schmidt, and M. Weber, "A novel ASIC design approach based on a new machine paradigm," *IEEE J. Solid-State Circuits*, vol. 26, no. 7, pp. 975–989, Jul. 1991.

[10] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou, "A fully pipelined and dynamically composable architecture of CGRA," in *Proc. IEEE 22nd Annu. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2014, pp. 9–16.

[11] D. C. Chen and J. M. Rabaey, "A reconfigurable multiprocessor IC for rapid prototyping of algorithmic-specific high-speed DSP data paths," *IEEE J. Solid-State Circuits*, vol. 27, no. 12, pp. 1895–1904, Dec. 1992.

[12] A. K. Yeung and J. M. Rabaey, "A reconfigurable data-driven multiprocessor architecture for rapid prototyping of high throughput dsp algorithms," in *Proc. 26th Hawaii Int. Conf. Syst. Sci.*, 1993, pp. 169–178.

[13] C. Ebeling, D. C. Cronquist, and P. Franklin, "Configurable computing: The catalyst for high-performance architectures," in *Proc. IEEE Int. Conf. Appl.-Specific Syst. Archit. Processors*, 1997, pp. 364–372.

[14] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a reconfigurable coprocessor," in *Proc. 5th Annu. IEEE Symp. Field-Programmable Custom Comput. Mach.*, 1997, pp. 12–21.

[15] T. Miyamori and U. Olukotun, "A quantitative analysis of reconfigurable coprocessors for multimedia applications," in *Proc. IEEE Symp. FPGAs Custom Comput. Mach.*, 1998, pp. 2–11.

[16] H. Zhang *et al.*, "A 1-V heterogeneous reconfigurable DSP IC for wireless baseband digital signal processing," *IEEE J. Solid-State Circuits*, vol. 35, no. 11, pp. 1697–1704, Nov. 2000.

[17] J. Becker and M. Vorbach, "Architecture, memory and interface technology integration of an industrial/ academic configurable system-on-chip (CSoC)," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, 2003, pp. 107–112.

[18] D. Rossi, F. Campi, S. Spolzino, S. Pucillo, and R. Guerrieri, "A heterogeneous digital signal processor for dynamically reconfigurable computing," *IEEE J. Solid-State Circuits*, vol. 45, no. 8, pp. 1615–1626, Aug. 2010.

[19] L. Duch, S. Basu, R. Braojos, G. Ansaloni, L. Pozzi, and D. Atienza, "HEAL-WEAR: An ultra-low power heterogeneous system for bio-signal analysis," *IEEE Trans. Circuits Syst. I: Regular Papers*, vol. 64, no. 9, pp. 2448–2461, Sep. 2017.

[20] J. Lee, Y. Shin, W. Lee, S. Ryu, and J. Kim, "Real-time ray tracing on coarse-grained reconfigurable processor," in *Proc. Int. Conf. Field-Programmable Technol.*, 2013, pp. 192–197.

[21] T. Toi, N. Nakamura, T. Fujii, T. Kitaoka, K. Togawa, K. Furuta, and T. Awashima, "Optimizing time and space multiplexed computation in a dynamically reconfigurable processor," in *Proc. Int. Conf. Field-Programmable Technol.*, 2013, pp. 106–111.

[22] Y. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit.*, 2016, pp. 367–379.

[23] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *Proc. 22nd Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2017, pp. 751–764.

[24] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects," in *Proc. 23rd Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2018, pp. 461–475.

[25] R. W. Hartenstein and R. Kress, "A datapath synthesis system for the reconfigurable datapath architecture," in *Proc. ASP-DAC'95/CHDL'95/VLSI'95 EDA Technofair*, 1995, pp. 479–484.

[26] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix," in *Field Programmable Logic and Application*. Berlin, Germany: Springer, 2003, pp. 61–70.

[27] T. Oppold, T. Schweizer, J. Oliveira Filho, S. Eisenhardt, and W. Rosenstiel, "CRC–concepts and evaluation of processor-like reconfigurable architectures," *IT-Inf. Technol.*, vol. 49, no. 3, pp. 157–164, 2007.

[28] Y. Park, H. Park, and S. Mahlke, "CGRA express: Accelerating execution using dynamic operation fusion," in *Proc. Int. Conf. Compilers Archit. Synthesis Embedded Syst.*, 2009, pp. 271–280.

[29] G. Ansaloni, P. Bonzini, and L. Pozzi, "EGRA: A coarse grained reconfigurable architectural template," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 19, no. 6, pp. 1062–1074, Jun. 2011.

[30] D. Voitsechov and Y. Etsion, "Single-graph multiple flows: Energy efficient design alternative for GPGPUs," in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit.*, 2014, pp. 205–216.

[31] Z. Zhao, W. Sheng, W. He, Z. Mao, and Z. Li, "A static-placement, dynamic-issue framework for CGRA loop accelerator," in *Proc. Des. Autom. Test Eur. Conf. Exhibit.*, 2017, pp. 1348–1353.

[32] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 416–429.

[33] Mirsky and DeHon, "MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *Proc. IEEE Symp. FPGAs Custom Comput. Mach.*, 1996, pp. 157–166.

[34] E. Waingold *et al.*, "Baring it all to software: Raw machines," *Computer*, vol. 30, no. 9, pp. 86–93, 1997.

[35] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *Proc. 36th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2003, pp. 291–302.

[36] K. Sankaralingam *et al.*, "Exploiting ILP, TLP, and DLP with the polymorphous trips architecture," in *Proc. 30th Annu. Int. Symp. Comput. Archit.*, 2003, pp. 422–433.

[37] V. Govindaraju *et al.*, "DySER: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, Sep./Oct. 2012.

[38] J. D. Souza, L. Carro, M. B. Rutzig, and A. C. S. Beck, "A reconfigurable heterogeneous multicore with a homogeneous ISA," in *Proc. Des. Autom. Test Europe Conf. Exhibit.*, 2016, pp. 1598–1603.

[39] C. Selvidge, A. Agarwal, M. Dahl, and J. Babb, "TIERS: Topology independent pipelined routing and scheduling for virtualwire amp compilation," in *Proc. 3rd Int. ACM Symp. Field-Programmable Gate Arrays*, 1995, pp. 25–31.

[40] D. C. Cronquist, P. Franklin, S. G. Berg, and C. Ebeling, "Specifying and compiling applications for rapid," in *Proc. IEEE Symp. FPGAs Custom Comput. Mach.*, 1998, pp. 116–125.

[41] M. Wan *et al.*, "Design methodology of a low-energy reconfigurable single-chip DSP system," *J. VLSI Signal Process. Syst.*, vol. 28, no. 1–2, pp. 47–61, 2001.

[42] N. Bansal, S. Gupta, N. Dutt, A. Nicolau, and R. Gupta, "Network topology exploration of mesh-based coarse-grain reconfigurable architectures," in *Proc. Des. Autom. Test Europe Conf. Exhibit.*, 2004, pp. 474–479.

[43] K. E. Coons, X. Chen, D. Burger, K. S. McKinley, and S. K. Kushwaha, "A spatial path scheduling algorithm for edge architectures," in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2006, pp. 129–140.

[44] F. Bouwens, M. Berekovic, A. Kanstein, and G. Gaydadjiev, "Architectural exploration of the ADRES coarse-grained reconfigurable array," in *Reconfigurable Computing: Architectures, Tools and Applications*. Berlin, Germany: Springer, 2007, pp. 1–13.

[45] S. A. Chin *et al.*, "CGRA-ME: A unified framework for CGRA modelling and exploration," in *Proc. Int. Conf. Appl.-Specific Syst. Archit.s Processors*, 2017, pp. 184–189.

[46] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proc. 27th Annu. Int. Symp. Microarchit.*, 1994, pp. 63–74.

[47] H. Mahdi, A. Shrivastava, and S. Vrudhula, "EPIMap: Using epimorphism to map applications on CGRAs," in *Proc. Des. Autom. Conf.*, 2012, pp. 1280–1287.

[48] L. Chen and T. Mitra, "Graph minor approach for application mapping on CGRAs," *Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 3, 2014, Art. no. 21.

[49] B. De Sutter, P. Coene, T. Vander Aa, and B. Mei, "Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays," in *Proc. ACM SIGPLAN-SIGBED Conf. Lang. Compilers, Tools Embedded Syst.*, 2008, pp. 151–160.

[50] M. Hamzeh, *Compiler and Architecture Design for Coarse-Grained Programmable Accelerators*. Phoenix, AZ, USA: Arizona State University, 2015.

[51] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck, "SPR: An architecture-adaptive CGRA mapping tool," in *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, 2009, pp. 191–200.

[52] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "REGIMap: Register-aware application mapping on coarse-grained reconfigurable architectures (CGRAs)," in *Proc. ACM/EDAC/IEEE Des. Autom. Conf.*, 2013, pp. 1–10.

[53] S. Yin *et al.*, "Conflict-free loop mapping for coarse-grained reconfigurable architecture with multi-bank memory," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 9, pp. 2471–2485, 2017.

[54] S. Yin, X. Yao, D. Liu, L. Liu, and S. Wei, "Memory-aware loop mapping on coarse-grained reconfigurable architectures," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 5, pp. 1895–1908, May 2016.

[55] S. Dave, M. Balasubramanian, and A. Shrivastava, "RAMP: Resource-aware mapping for CGRAs," in *Proc. 55th Annu. Des. Autom. Conf.*, 2018, pp. 127:1–127:6.

[56] Z. Zhao, W. Sheng, N. Jing, W. He, and Z. Mao, "Resource-saving compile flow for coarse-grained reconfigurable architectures," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs*, 2015, pp. 1–8.

[57] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," in *Proc. Des. Autom. Test Eur. Conf. Exhibit.*, 2003, pp. 296–301.

[58] R. Gnanaolivu, T. S. Norvell, and R. Venkatesan, "Mapping loops onto coarse-grained reconfigurable architectures using particle swarm optimization," in *Proc. Int. Conf. Soft Comput. Pattern Recognit.*, 2010, pp. 145–151.

[59] S. A. Chin and J. H. Anderson, "An architecture-agnostic integer linear programming approach to CGRA mapping," in *Proc. 55th ACM/ESDA/IEEE Des. Autom. Conf.*, 2018, pp. 1–6.

[60] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "DRESC: A retargetable compiler for coarse-grained reconfigurable architectures," in *Proc. IEEE Int. Conf. Field-Programmable Technol.*, 2002, pp. 166–173.

[61] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-S. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proc. 17th Int. Conf. Parallel Archit.s Compilation Techn.*, 2008, pp. 166–176.

[62] G. Dimitroulakos, S. Georgiopoulos, M. D. Galanis, and C. E. Goutis, "Resource aware mapping on coarse grained reconfigurable arrays," *Microprocessors Microsystems*, vol. 33, no. 2, pp. 91–105, 2009.

[63] H. Park, K. Fan, M. Kudlur, and S. Mahlke, "Modulo graph embedding: Mapping applications onto coarse-grained reconfigurable architectures," in *Proc. Int. Conf. Compilers Archit. Synthesis Embedded Syst.*, 2006, pp. 136–146.

[64] P. Theocharis and B. D. Sutter, "A bimodal scheduler for coarse-grained reconfigurable arrays," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 2, Jun. 2016. [Online]. Available: https://doi.org/10.1145/2893475

[65] T. Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili, "A general constraint-centric scheduling framework for spatial architectures," in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2013, pp. 495–506.

[66] X. Man, L. Liu, J. Zhu, and S. Wei, "A general pattern-based dynamic compilation framework for coarse-grained reconfigurable architectures," in *Proc. 56th ACM/IEEE Des. Autom. Conf.*, 2019, pp. 1–6.

[67] D. Liu *et al.*, "Data-flow graph mapping optimization for CGRA with deep reinforcement learning," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 38, no. 12, pp. 2271–2283, Dec. 2019.

[68] J. Llosa, E. Ayguade, A. Gonzalez, M. Valero, and J. Eckhardt, "Lifetime-sensitive modulo scheduling in a production environment," *IEEE Trans. Comput.*, vol. 50, no. 3, pp. 234–249, Mar. 2001.

[69] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim.*, 2004, pp. 75–86.

[70] L. Liu, Z. Li, C. Yang, C. Deng, S. Yin, and S. Wei, "HReA: An energy-efficient embedded dynamically reconfigurable fabric for 13-dwarfs processing," *IEEE Trans. Circuits Syst. II: Express Briefs*, vol. 65, no. 3, pp. 381–385, Mar. 2018.

[71] K. Asanovic *et al.*, "The landscape of parallel computing research: A view from berkeley," EECS Dept., Univ. California, Berkeley, Tech. Rep. UCB/EECS-2006–183, 2006.

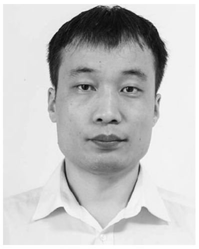
[72] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems," in *Proc. 30th Annu. ACM/IEEE Int. Symp. Microarchit.*, 1997, pp. 330–335.

[73] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. 4th Annu. IEEE Int. Workshop Workload Characterization*, 2001, pp. 3–14.

[74] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "MachSuite: Benchmarks for accelerator design and customized architectures," in *Proc. IEEE Int. Symp. Workload Characterization*, 2014, pp. 110–119.

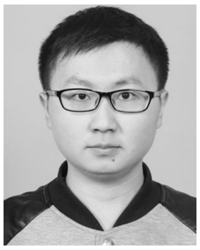
[75] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," 2012. [Online]. Available: http://www.cs.ucla.edu/pouchet/software/polybench

Zhongyuan Zhao received the BS degree from the School of Electronics and Information, Harbin Institute of Technology, Harbin, China, in 2012. Currently, he is currently working toward the PhD degree with the Department of Nano/Micro Electronics, Shanghai Jiao Tong University, Shanghai, China. His research interests include compiler and architecture optimization for reconfigurable computing platform and deep learning accelerators.



Weiguang Sheng received the bachelor's, master's, and PhD degrees from the Harbin Institute of Technology, Harbin, China, in 1999, 2004, and 2009, respectively. He is currently an research assistant professor with the Department of Micro/Nano Electronics, Shanghai Jiao Tong University. His research interest includes Reconfigurable Architectures and Compiling Techniques, Soft Error Analysis and Optimization.

Qin Wang received the BS degree from the University of Electronics Science and Technology of China, ChenDu, China, in 1997, and the PhD degree from Shanghai Jiao Tong University, in 2004. She has been an associate professor with the Department of Microelectronics and Nanoscience, Shanghai Jiao Tong University. Her research interests include deep learning, inmemory computing and 3D IC.



Wenzhi Yin received the MS degree from the Institute of Microelectronics, Shanghai Jiao Tong University, China, in 2019. His research in university includes reconfigurable computing and compiler optimization for CGRAs. Currently he is working on deep learning acceleration with GPU in Nvidia.

Pengfei Ye received the BS degree from the School of Information and Science Technology, East China Normal University, Shanghai, China, in 2017. Currently he is working toward the MS degree with the Institute of Microelectronics, Shanghai Jiao Tong University, Shanghai, China. His research interests include reconfigurable computing and optimization of compiler for reconfigurable computing.

Jinchao Li received the BS degree from the School of Electronic Science and Engineering, University of Electronic Science and Technology of China, Chengdu, China, in 2017. Currently, he is working toward the MS degree with the Department of Nano/Micro Electronics, Shanghai Jiao Tong University, Shanghai, China. His research interests include reconfigurable computing and architecture optimization of reconfigurable computing platform.

Zhigang Mao received the BS degree from Tsinghua University, China, in 1986, and the PhD degree from the University of Rennes 1, Rennes, France, in 1992. From 1992 to 2006, he was with the Microelectronics Center, Harbin Institute of Technology, Harbin, China. In 2006, he joined the Department of Micro–Nano Electronics, Shanghai Jiao Tong University, Shanghai, China, where he is currently a professor. His current research interests include DSP architecture design, video processor design, and reconfigurable processor architecture.

Lock-Free Parallelization for Variance-Reduced Stochastic Gradient Descent on Streaming Data

Yaqiong Peng, *Member, IEEE*, Zhiyu Hao, and Xiaochun Yun

Abstract—Stochastic Gradient Descent (SGD) is an iterative algorithm for fitting a model to the training dataset in machine learning problems. With low computation cost, SGD is especially suited for learning from large datasets. However, the variance of SGD tends to be high because it uses only a single data point to determine the update direction at each iteration of gradient descent, rather than all available training data points. Recent research has proposed variance-reduced variants of SGD by incorporating a correction term to approximate full-data gradients. However, it is difficult to parallelize such variants with high performance and accuracy, especially on streaming data. As parallelization is a crucial requirement for large-scale applications, this article focuses on the parallel setting in a multicore machine and presents LFS-STRSAGA, a lock-free approach to parallelizing variance-reduced SGD on streaming data. LFS-STRSAGA embraces a lock-free data structure to process the arrival of streaming data in parallel, and asynchronously maintains the essential information to approximate full-data gradients with low cost. Both our theoretical and empirical results show that LFS-STRSAGA matches the accuracy of the state-of-the-art variance-reduced SGD on streaming data under *sparsity assumption* (common in machine learning problems), and that LFS-STRSAGA reduces the model update time by over 98 percent.

Index Terms—Gradient descent methods, machine learning, parallel computing, multicore, streaming data

1 Introduction

1.1 Motivation

The core of machine learning problems is to learn a mathematical model from training datasets, in order to make predictions or decisions. The model is typically characterized by a decision variable, namely a vector of weights $\mathbf{w} \in \mathbb{R}^d$. Let us consider a set of n training data points $\mathcal{S}$. Fitting a model to $\mathcal{S}$ is usually cast as an optimization problem of minimizing the finite sum form: $\mathcal{R}_{\mathcal{S}}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{w})$, where $f_i(\mathbf{w})$ is the loss of $\mathbf{w}$ on the ith data point in $\mathcal{S}$. $\mathcal{R}_{\mathcal{S}}(\mathbf{w})$ is often assumed to be convex. The goal of machine learning problems is typically to find the decision variable $\mathbf{w}^*$ that minimizes $\mathcal{R}_{\mathcal{S}}(\mathbf{w})$. Gradient descent methods are widely used to pursue $\mathbf{w}^*$.

There are two common gradient descent methods: *Batch Gradient Descent* (BGD) and *Stochastic Gradient Descent* (SGD) [1], [2]. BGD uses all training data points to compute the full gradient at each iteration of gradient descent. In BGD, the cost of a gradient computation increases with the size of the training dataset. In contrast, SGD selects a single data point randomly from the training dataset to compute the gradient at each iteration of gradient descent, and thus SGD is especially suited for a large dataset. While an iteration of SGD is cheaper than that of BGD, its variance tends to be high. Remarkable recent progress has been made to propose *Variance-Reduced Variants of SGD* (VR SGD) by incorporating a correction term to approximate a full-data gradient generally in two ways. The first variance-reduced method stores the last gradient computed at each data point and uses their average at each iteration of gradient descent [3], [4], [5]. Another method periodically computes a gradient based on a batch of data points [6], [7], [8], requiring more computation than the first method.

Meanwhile, parallelization is a crucial requirement for large-scale applications. In parallel setting, many asynchronous variants of SGD are proposed on both a single multicore machine and distributed systems. Although the variance-reduced improvements yield a more accurate model in general, it is more challenging to parallelize such variants with high performance and accuracy, especially on streaming data arriving as an endless sequence of data points, which is a common scenario in computer systems. The first variance-reduced method mentioned above requires using the average of historical gradients. In parallel setting, it is difficult to maintain the average on streaming data with low cost, because the size of training dataset changes over time. Section 2.3 analyzes the problem in detail. In addition, the second method is also not suitable for this scenario, because maintaining a model on streaming data is limited in processing time.

In this paper, we focus on the parallel setting in a multicore machine with shared memory, and explore how to parallelize VR SGD on streaming data with high performance and accuracy. Although large datasets are generally suggested to be processed on a cluster of machines, the size of the data necessary for statistical analysis may be a few terabytes or less after appropriate preprocessing in many problems [9]. Such problems can be processed on a single multicore machine, rather than an expensive cluster. In addition, multicore systems have significant performance advantages due to low latency and high throughput of shared main memory [9].

• *The authors are with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China. E-mail: {pengyaqiong, haozhiyu, yunxiaochun}@iie.ac.cn.*

Manuscript received 6 Aug. 2019; revised 29 Feb. 2020; accepted 9 Apr. 2020. Date of publication 15 Apr. 2020; date of current version 8 May 2020. (Corresponding author: Zhiyu Hao.) Recommended for acceptance by H. Huang. Digital Object Identifier no. 10.1109/TPDS.2020.2987867

1.2 Key Ideas

Overcoming the challenges of parallelizing VR SGD on streaming data with high performance and accuracy, we present an algorithm named LFS-STRSAGA based on the state-of-the-art serial streaming data version of VR SGD named STRSAGA [10]. STRSAGA belongs to the first variance-reduced method mentioned in Section 1.1. It stores the last gradient computed for each data point in a global vector. To parallelize STRSAGA, LFS-STRSAGA embraces a lock-free concurrent data structure to process the arrival of streaming data. In addition, LFS-STRSAGA replaces the global vector of historical gradients with a local one for each thread to asynchronously maintain, and the model update rule based on the thread-local vector of historical gradients is sparse. To pursue high performance, LFS-STRSAGA updates the decision variable without using any locking like HOGWILD! [9], which is the foundation for the most parallel implementations of SGD and its variants. Our theoretical analysis shows that LFS-STRSAGA retains the linear convergence rate of STRSAGA to reach statistical accuracy in sparsity assumption. The sparsity assumption means that most update steps only modify small and disjoint parts of the decision variable. This phenomenon exists in most machine learning problems.

Like STRSAGA, LFS-STRSAGA is the choice in the settings where memory is not limited. Nowadays, the settings given the abundance of memory are common in many practical situations [10]. As mentioned in Section 7, we will explore how to restrict LFS-STRSAGA to use limited memory in the future work.

1.3 Experimental Evaluation

To show the competitiveness of LFS-STRSAGA in practice, we compare it to the original version of STRSAGA [10] and *the natural streaming data version of HOGWILD!* [9] (referred to as S-HOGWILD! in this paper) in terms of model update time and *sub-optimality* (defined in Section 4.2). We consider the support of all the evaluated algorithms for two popular machine learning problems, *Logistic Regression* and *Matrix Factorization*, on real-world datasets under various arrival distributions. The experimental results show that LFS-STRSAGA matches the accuracy of STRSAGA, and that it reduces the model update time by over 98 percent. In addition, LFS-STRSAGA is more accurate than S-HOGWILD!.

1.4 Contributions

In summary, this paper makes the following contributions:

- The presentation of LFS-STRSAGA: a lock-free approach to parallelizing VR SGD algorithm on streaming data.
- A theoretical analysis showing that LFS-STRSAGA matches the accuracy of STRSAGA in sparsity assumption.
- An experimental evaluation demonstrating the competitiveness of LFS-STRSAGA in practice.

1.5 Outline

The rest of the paper is organized as follows. Section 2 provides a foundation for LFS-STRSAGA. Section 3 presents the details of LFS-STRSAGA, followed by a theoretical analysis in Section 4. Section 5 provides a comprehensive evaluation of LFS-STRSAGA, and Section 6 overviews the most related work. Section 7 concludes the paper and discusses our future work.

2 Preliminaries

In this section, we first discuss the basics with respect to LFS-STRSAGA in more detail, and then look into the key ideas of typical SGD algorithms and the variance-reduced improvements. Finally, we discuss the challenges for parallelizing VR SGD on streaming data.

2.1 Basics

Empirical Risk Minimizer. The core of machine learning problems is to learn a mathematical model, which is drawn from a class of functions $\mathcal{F}$ and parameterized by a vector of weights $\mathbf{w} \in \mathbb{R}^d$ called decision variable. The expected risk of a function in $\mathcal{F}$ is defined as $\mathcal{R}(\mathbf{w}) = \mathbb{E}[f_{\mathbf{x}}(\mathbf{w})]$, where $f_{\mathbf{x}}(\mathbf{w})$ denotes the loss of $\mathbf{w}$ on input $\mathbf{x}$ and the expectation is taken over all $\mathbf{x}$ drawn from some underlying probability distribution $\mathcal{P}$ (unknown to the algorithm). For example, given a data point (x, y), the corresponding loss in L2-regularized logistic regression problem is: $f_{(x,y)}(\mathbf{w}) = log(1 + exp(-y\mathbf{w}^T x)) + \frac{\mu}{2}||\mathbf{w}||_2^2$. Let $\mathbf{w}^*$ = arg $\min_{\mathbf{w}\in\mathcal{F}}\mathcal{R}(\mathbf{w})$ denote the decision variable minimizing $\mathcal{R}(\mathbf{w})$. Then, $\mathcal{R}(\mathbf{w}^*)$ denotes the minimum expected risk over $\mathcal{F}$. Given a sample $\mathcal{S}$ consisting of n training data points drawn from $\mathcal{P}$, the empirical risk function measuring the average loss of $\mathbf{w}$ over $\mathcal{S}$ is defined as: $\mathcal{R}_{\mathcal{S}}(\mathbf{w}) = \frac{1}{n}\sum_{\mathbf{x}\in\mathcal{S}} f_{\mathbf{x}}(\mathbf{w})$. The best we can do is to find the *Empirical Risk Minimizer* (**ERM**), namely the weights $\mathbf{w}_{\mathcal{S}}^*$ = arg $\min_{\mathbf{w}\in\mathcal{F}}\mathcal{R}_{\mathcal{S}}(\mathbf{w})$.

Statistical Efficiency. Suppose that an algorithm produces an approximate solution $\mathbf{w}_{\mathcal{S}}$ over $\mathcal{S}$. $\mathbb{E}\left[\mathcal{R}_{\mathcal{S}}(\mathbf{w}_{\mathcal{S}}^*) - \mathcal{R}(\mathbf{w}^*)\right]$ is called statistical error, where the expectation is over the randomness of n-sample $\mathcal{S}$. The statistical error is usually bounded by $\mathcal{H}(n) = cn^{-\alpha}$, where c is a constant and $1/2 \leq \alpha \leq 1$. $\mathbb{E}\left[\mathcal{R}_{\mathcal{S}}(\mathbf{w}_{\mathcal{S}}) - \mathcal{R}_{\mathcal{S}}(\mathbf{w}_{\mathcal{S}}^*)\right]$ is called optimization error. Let $\epsilon(n)$ be an upper bound on the optimization error, then the total error is bounded by $\mathcal{H}(n) + \epsilon(n)$. Given a computational budget, $\epsilon(n)$ is typically increasing with n, whereas $\mathcal{H}(n)$ is always decreasing. To minimize the total error, the literature on learning theory suggests reducing $\epsilon(n)$ to asymptotically balance with $\mathcal{H}(n)$ [11].

Streaming Data Scenario. This paper considers the maintenance of a model in the same streaming data scenario as mentioned in [10]: Let $\mathbf{X}_i$ denote the training data points (zero or more) arriving at time step i, and each data point is drawn from some underlying probability distribution $\mathcal{P}$ (unknown to the algorithm). Let $\mathcal{S}_i$ denote the set of data points arriving in time steps 1 to i ($\mathcal{S}_i = \cup_{j=1}^{i}\mathbf{X}_j$). The optimal solution over $\mathcal{S}_i$ is denoted as $\mathbf{w}_i^* = \mathbf{w}_{\mathcal{S}_i}^*$ = arg $\min_{\mathbf{w}\in\mathcal{F}}\mathcal{R}_{\mathcal{S}i}(\mathbf{w})$. Our goal is to produce a solution $\mathbf{w}_i$, of which the empirical risk approximates $\mathcal{R}_{\mathcal{S}i}(\mathbf{w}_i^*)$ to the best. We focus on using SGD-style algorithms to achieve this goal.

2.2 SGD and its Variance-Reduced Variants

SGD is an iterative algorithm widely used to optimize problems of the finite-sum form. Generally, SGD starts with a known initial vector of weights $\mathbf{w}^0$, and updates the vector repeatedly. In every iteration, SGD samples a data point uniformly from the training dataset, and computes the gradient at this point based on the current state of decision variable. Then, it uses the gradient to determine the update direction of the decision variable. Given the value of $\mathbf{w}$ at the end of iteration t (referred to as $\mathbf{w}^t$), the updates for iteration $t+1$ in SGD are generally as follows:

1. Sample a data point i uniformly from $\mathcal{S}$ at random.
2. Compute the gradient $\nabla f_i(\mathbf{w}^t)$ at data point i.
3. Update $\mathbf{w}$: $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \eta\nabla f_i(\mathbf{w}^t)$, where η is the step size.

Because $\mathbf{w}$ is updated through a gradient computed at a single data point in the standard SGD, the variance of the update direction tends to be high. To ensure convergence, decreasing step sizes are generally chosen for SGD, resulting in a slower sub-linear convergence rate than using constant step sizes. The variance-reduced variants of SGD are proposed to solve this problem. Here, we focus on the update scheme of SAGA, which is a state-of-the-art variance-reduced method. SAGA starts with a known initial vector of weights $\mathbf{w}^0$, and uses a table α to store historical gradients. The length of α is equal to the number of training data points. Each element $\alpha(p)$ in α is the last gradient computed for each data point p (referred to as $\alpha(p) \in \mathbb{R}^d$), initialized to $\nabla f_p(\mathbf{w}^0)$. Given the value of $\mathbf{w}$ and each $\alpha(p)$ at the end of iteration t, the updates for iteration $t+1$ in SAGA are as follows:

1. Sample a data point i uniformly from $\mathcal{S}$ at random.
2. Compute the gradient $\nabla f_i(\mathbf{w}^t)$ at data point i.
3. Update $\mathbf{w}$ using $\nabla f_i(\mathbf{w}^t)$, $\alpha(i)$ and the historical gradient average: $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \eta(\nabla f_i(\mathbf{w}^t) - \alpha(i) + \frac{1}{n}\sum_{i=1}^{n}\alpha(i))$, where n is the number of training data points.
4. Update $\alpha(i)$, and other historical gradients remain unchanged: $\alpha(i) = \nabla f_i(\mathbf{w}^t)$

To control the computation cost, SAGA computes a gradient at a single data point like SGD. To approximate a full-data gradient, SAGA stores the last gradient computed at each data point and uses their average at each iteration of gradient descent. In this way, SAGA yields a more accurate model than SGD. Note that SAGA is an offline algorithm, which assumes the entire dataset is available beforehand. To the best of our knowledge, STRSAGA is the best performing VR SGD algorithm for handling streaming data, which is designed based on SAGA. As mentioned in Section 2.1, to guarantee the model accuracy, it is suggested to reduce the optimization error to asymptotically balance with the statistical error. Intuitively, it means that it is better to concentrate the computational budget on fewer data. Following this principle, STRSAGA increases the size of the sample set in a controlled manner. It does not add new data points into the effective sample set until a sufficient number of SAGA steps have been performed on the current sample set.

2.3 Challenges for Parallelizing Variance-Reduced SGD on Streaming Data

As parallelization is a crucial requirement for large-scale applications, this paper discusses the parallelization of VR SGD algorithm on streaming data with high performance and accuracy, and presents LFS-STRSAGA based on STRSAGA. To implement LFS-STRSAGA, we face the following challenges.

Challenge 1. How to sample from dynamically changing dataset in parallel?

In streaming data scenario, with the arrival of new data points, the training dataset changes over time. In parallel setting, concurrent threads may contend for changing the state of training dataset at the same time. Following STRSAGA, we add new training data points to the sample set in a controlled manner, and sampling threads only pick data points from the effective sample set for gradient computation. Similar to changing the training dataset, concurrent threads may contend for adding one training data point. In addition, changing the state of training dataset must be notified to sampling threads without preventing them from making progress. Therefore, we need to carefully design the lock-free synchronization scheme to coordinate threads with changing the state of training dataset and sample set in parallel.

Challenge 2. How to maintain the average of last gradients computed at different data points with low cost?

Note that computing the average of historical gradients from scratch at each update step incurs non-trivial computation cost. To solve this problem, STRSAGA *updates the sum of last gradients computed at different data points incrementally*[1] and computes the average. To adopt this method on streaming data at an update step, our approach needs to get the size of the dynamic sample set. In parallel setting, suppose that a thread first reads the sum of historical gradients and then the size of sample set, and that some new data points are added into the sample set in the time interval between the two read operations. In this case, the sum of historical gradients and the size of sample set read by this thread does not correspond to the same state of the effective sample set. In addition, different threads may contend for updating the historical gradient associated with a data point at the same time. Therefore, in parallel setting, it is difficult to maintain the average of historical gradients with low cost in a lock-free manner.

3 LFS-STRSAGA: A Lock-Free Parallel Variant of Variance-Reduced SGD on Streaming Data

In this section, we present the details of LFS-STRSAGA. We begin with the high-level design of LFS-STRSAGA, followed by presenting a lock-free data structure for storing the streaming data. Next, we discuss how to process the arrival of new training data points based on the lock-free data structure. Finally, we introduce the details of update scheme used in LFS-STRSAGA.

3.1 High-Level Design of LFS-STRSAGA

Computation Model. Like most concurrent algorithms, LFS-STRSAGA is designed for asynchronous shared memory systems [12], where an application runs with n deterministic threads. In LFS-STRSAGA, threads are split between dedicated producers and consumers. Fig. 1 shows a diagram for this computation model. As shown in the diagram, between any two time steps i and $i+1$ ($i > 0$), producer and consumer threads work in parallel. To be specific, producer

1. Assume STRSAGA computes a gradient g at a data point p at an update step. The current sum of historical gradients is s, and the last gradient computed at p is g'. Then, STRSAGA updates the sum to $s + g - g'$.

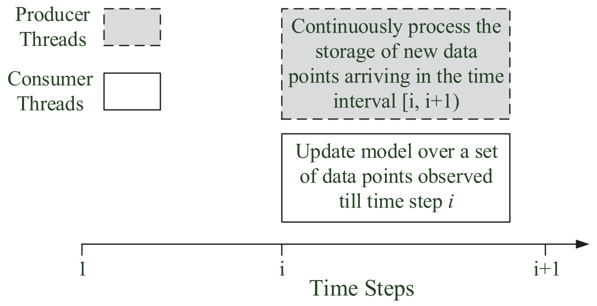

Producer Threads
Consumer Threads
Continuously process the storage of new data points arriving in the time interval [i, i+1)
Update model over a set of data points observed till time step i
1
i
i+1
Time Steps

Fig. 1. Computation model.

threads continuously process the storage of new data points arriving in the time interval $[i, i+1)$, and consumer threads update the model over a set of data points observed till the time step i. The time step 1 is the initial time for LFS-STRSAGA, at which no data points have arrived. Because LFS-STRSAGA is time efficient, consumer threads will complete the model update steps over the data points observed till the time step i before the next time step $i+1$.

Following STRSAGA, LFS-STRSAGA adds the new data points into the effective sample set after performing a sufficient number of update steps on the current sample set. As mentioned in Section 2.3, we face two challenges to achieve this goal in parallel. For challenge 1, we carefully design an infinite array. Fig. 2 provides a diagram for the array.

As shown in Fig. 2, the infinite array is a 4-tuple ($Dataset$, $\widetilde{T}$, $\widetilde{S}$, N). $Dataset$ is used to store the observed data points. $\widetilde{T}$, $\widetilde{S}$, and N are three indexes dividing the data points of $Dataset$ into three classes.

$Dataset[0,...,\min(\widetilde{T}, \widetilde{S}))$ acts as the effective sample set in LFS-STRSAGA. $\widetilde{T}$ is only updated atomically by consumer threads for adding data points into the effective sample set. $\widetilde{S}$ denotes the number of data points that have been deposited into $Dataset$, which is only updated by producer threads. $\widetilde{S}$ marks the upper boundary for the effective sample set. In other words, consumer threads extend the effective sample set at most to $Dataset[0, \ldots, \widetilde{S})$. The storage of a new data point consists of two atomic operations performed by a producer thread that first gets an index and then deposits the data point into the corresponding location of $Dataset$. If $\widetilde{S}$ is used to distribute the indexes of new data points, consumer threads may sample a data point that only gets an index but has not been deposited into $Dataset$. Therefore, we set another global variable N for distributing the index of every new data point processed by producer threads. Like $\widetilde{S}$, N is only updated atomically by producer threads. At each time step i ($i > 0$), a producer thread (referred to as main producer) is responsible for updating $\widetilde{S}$ to let consumer threads see all the stored data points arriving till the time step i. All the initial values of $\widetilde{T}$, $\widetilde{S}$, and N are 0.

Like the work on concurrent array-based data structures [13], [14], the infinite array can be implemented by a singly-linked list of equal-sized array segments. Section 3.2 will discuss how to operate on the array without using any locking in parallel. In addition, we present a parallel sample strategy based on this data structure. Section 3.3 will describe the details.

For challenge 2, we present an asynchronous strategy for approximating full-data gradients. Each thread maintains a local vector of historical gradients to approximate full-data gradients in LFS-STRSAGA, and uses the sparse version of their average with low computation expenses at an update step. Section 3.3 will describe the details.

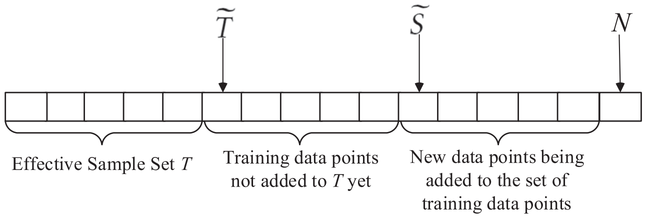

$\widetilde{T}$
$\widetilde{S}$
N
Effective Sample Set T
Training data points not added to T yet
New data points being added to the set of training data points

Fig. 2. Infinite array for storing data points.

Algorithm 1. Storing Training Data Points by Individual Producer Threads

Global shared variables: $Dataset[1, \ldots]$: infinite array for storing observed data points, which can be implemented by a singly-linked list of equal-sized array segments ; $\widetilde{S}$: number of data points that have been deposited into $Dataset$, initialized to 0; N: number of data points that have arrived now, initialized to 0.

```
1:  for i = 1, 2, 3, ... do
2:      for every time receive a data point $P$ arriving between
        time step $i$ and $i+1$ do
3:          // store the observed data point
4:          $j \leftarrow$ FAA($\&N$, 1)
5:          $Datasets[j] \leftarrow P$
6:      end for
7:      Wait for other producer threads
8:      if i am the main producer then
9:          // let consumer threads see the new data points
10:         $\widetilde{S} \leftarrow N$
11:     end if
12:     Wait for other producer threads
13: end for
```

3.2 Storing New Training Data Points

Algorithm 1 presents how to store training data points by individual producer threads. Once a new data point arrives, LFS-STRSAGA distributes it to a thread for storing it into the array named $Dataset$. As different threads may process the storage of new data points at the same time, we need to consider the problem that threads could overwrite each other's progress by depositing different data points into the same location of the array. LFS-STRSAGA uses the *Fetch-and-Add* (FAA)[2] atomic operation to solve this problem. A thread first obtains the index of the location for storing the data point by performing a FAA on the index N (line 4), and then stores the data point in the associated location of the array (line 5). The atomic feature of FAA ensures each data point gets a unique index. Among the producer threads, a thread acts as a main producer. At time step $i+1$ ($i > 0$), before all the producer threads continue to store the training data points arriving between the time steps $i+1$ and $i+2$, the main producer updates $\widetilde{S}$ to the current value of N in order to let consumer threads see all the stored data

2. FAA(*addr*, v): assuming the content stored in *addr* are a, this operation returns a and stores $a+v$ at *addr*.

points arriving till the time step $i+1$ (lines 8-11). Then, consumer threads start to update the model over the stored data points, and producer threads continue to process data points arriving between the time steps $i+1$ and $i+2$.

Algorithm 2. Model Updated Over a Set of Training Data Points S_i That Arrived in Time Steps 1 to i by Individual Consumer Threads, $i > 0$

Global shared variables: **w**: decision variable; $Datasets[1, \ldots]$: storing observed data points; $\widetilde{S}$: number of data points that have been deposited into $Dataset$, initialized to 0; $\widetilde{T}$: index of next data point added to the effective sample set, initialized to 0; t: iteration count, initialized to 0 at each time step.

Thread-local variables: α: a table for storing local historical gradients, initialized to $\emptyset$; $next\_p$: recording the index of the next data point added by the current thread to the effective sample set, initialized to -1; sum: the sum of all entries in α, initialized to 0^d and maintained incrementally (d is the number of features in a gradient);

```
1:  for ($t \leftarrow$ FAA(&$t$, 1); $t < \rho$; $t \leftarrow$ FAA(&$t$, 1)) do
2:    $effective\_sample\_size \leftarrow 0$
3:    if $t$ is even then // try to extend the effective sample set in the even iteration number
4:      if $next\_p \neq -1$ and $next\_p < \widetilde{S}$ then // add the data point reserved in the previous even iteration
5:        $p \leftarrow next\_p$, $next\_p \leftarrow -1$
6:        $effective\_sample\_size \leftarrow p+1$
7:      else
8:        if $\widetilde{T} < \widetilde{S}$ then // try to add a data point to the effective sample set
9:          $p \leftarrow$ FAA(&$\widetilde{T}$, 1)
10:         if $p < \widetilde{S}$ then // if successful, pick the added data point
11:           $effective\_sample\_size \leftarrow p+1$
12:         else // otherwise, reserve the index returned with FAA
13:           $next\_p \leftarrow p$
14:         end if
15:       end if
16:     end if
17:   end if
18:   if $effective\_sample\_size = 0$ then // if $t$ is odd or fail to extend the effective sample set in the even iteration number, directly sample a data point from the effective sample set at random
19:     $effective\_sample\_size \leftarrow \min(\widetilde{T}, \widetilde{S})$
20:     sample $p \sim$ Uniform[0, $effective\_sample\_size$)
21:   end if
22:   pick the data point $Datasets[p]$ and compute its gradient $\nabla f_p(\mathbf{w})$ on the current state of $\mathbf{w}$
23:   $g \leftarrow \nabla f_p(\mathbf{w})$
24:   for $v \in e(p)$ do // update the decision variable
25:     $\mathbf{w}_v \leftarrow \mathbf{w}_v - \eta(g_v - \alpha_v(p) + \frac{sum_v}{e_v \cdot effective\_sample\_size})$
26:   end for
27:   $sum \leftarrow sum + g - \alpha(p)$ // update the sum of local historical gradients
28:   $\alpha(p) \leftarrow g$ // update the last gradient computed at the data point $Datasets[p]$
29: end for
30: // The current effective sample set is $T_i$.
```

TABLE 1
Symbols Used in Algorithm 2

$\lvert\alpha\rvert$	number of gradients contained in α
$e(p)$	indexes of the coordinates on which the values of the data point indexed by p are non-zero
η	step size
$\alpha(p)$	if the local thread has not ever computed any gradient at the data point indexed by p, return 0; otherwise, return the last gradient computed at the data point indexed by p
x_v	value of the vector x (e.g., $\mathbf{w}$, g, and $\alpha(p)$) on the coordinate indexed by v
e_v	among the data points that have been picked by the local thread to update the decision variable, fraction of the data points that are non-zero on the vth coordinate

3.3 Update Scheme

Algorithm 2 shows the details of updating the model over a set of training data points S_i that arrived in time steps 1 to i by individual consumer threads. The algorithm mainly repeats three procedures: 1) pick a data point for gradient computation (lines 3-21); 2) update the decision variable by using the picked data point and local historical gradients to approximate the full-data gradients (lines 22-26); 3) update the local historical gradients and their sum (lines 27-28). Table 1 describes the symbols used in Algorithm 2.

Before presenting our sample strategy, let us first recall the sample strategy adopted by the latest version of STRSAGA. At each time step, STRSAGA starts to perform ρ iterations of gradient descent. To control the expansion of the effective sample set, STRSAGA first adds the new data points to a buffer. Then, it chooses a new data point from the buffer every two iterations, and adds the data point to the effective sample set followed by computing a gradient at this data point. In other iterations, STRSAGA directly samples a data point from the effective sample set for gradient computation without expanding the effective sample set. In LFS-STRSAGA, $Dataset[\widetilde{T}, \ldots, \widetilde{S})$ and $Dataset$[0,...,min($\widetilde{T}, \widetilde{S}$)) are equivalent to the buffer and effective sample set in STRSAGA, respectively.

Parallel Sample Strategy. LFS-STRSAGA distributes the ρ iterations of gradient descent starting at each time step to consumer threads dynamically, and coordinates these threads to expand the effective sample set in a controlled manner by performing FAA operations on $\widetilde{T}$ every two iterations (line 10). Note that consumer threads extend the effective sample set at most to $Dataset[0, \ldots, \widetilde{S})$. When $\widetilde{T}$ is updated to a value larger than $\widetilde{S}$, $Dataset[0, \ldots, \widetilde{S})$ denotes the effective sample set. Therefore, before performing a FAA in the even iterations, a consumer thread first judges whether $\widetilde{T}$ is currently less than $\widetilde{S}$ (line 8). If yes, the thread first performs a FAA on $\widetilde{T}$ (line 9), and then judges whether the value returned with FAA is less than $\widetilde{S}$ (line 10). If yes, it means that the value returned with FAA is equal to the index of a data point that has been stored into the array $Datasets$, and the thread selects the data point for gradient computation afterwards. Otherwise, because $\widetilde{T}$ is monotonically increased by FAA operations, LFS-STRSAGA may not ensure a sufficient number of gradient computation is performed on data points stored afterwards. To solve the problem, when the value returned with FAA is not less than $\widetilde{S}$, the thread records the value in $next\_p$ (line 13) as the candidate index considered in the next even iteration (lines 4-6). Therefore, at the start of every even iteration, a consumer thread first checks whether an index is reserved in $next\_p$ and the reserved index is less than $\widetilde{S}$ (line 4). If yes ($next\_p \neq -1$ and $next\_p < \widetilde{S}$), the thread directly selects the reserved index as the target index, and resets $next\_p$ to -1 for the next reservation (line 5). If the current iteration number is odd or the consumer thread fails to extend the effective sample set in the even iteration number, the thread directly samples a target index uniformly from [0, min($\widetilde{T}$, $\widetilde{S}$)) (lines 19-20). Once the target index is determined, the consumer thread computes a gradient at the indexed data point on the current state of $\mathbf{w}$ (lines 22).

Asynchronous Strategy for Approximating Full-Data Gradients. Different from STRSAGA, each consumer thread maintains a local vector of historical gradients to approximate full-data gradients in LFS-STRSAGA, and incrementally updates the sum of the local historical gradients. Because the local vector of historical gradients is only updated by its owner thread, LFS-STRSAGA avoids the inconsistent read problem mentioned in Section 2.3. Under this scheme, each consumer thread can easily maintain the average of local historical gradients with low cost. As mentioned in Section 2.2, SAGA-style algorithms use the gradient computed at a random data point and the average of historical gradients to update $\mathbf{w}$. To reduce the computation expenses, we borrow an idea from ASAGA [15]. When computing the average of the local historical gradients, each thread only considers the coordinates on which the values of the picked data point are non-zero in LFS-STRSAGA (lines 24-26). To make the update unbiased, assuming the picked data point is indexed by p, we multiply the value of the sum of the local historical gradients on the vth coordinate ($v \in e(p)$) by a coefficient $\frac{1}{e_v}$ (line 25) for approximating the full sum. The definition of e_v is given in Table 1. Then, we get the average of local historical gradients on the vth coordinate by the equation $\frac{sum_v}{e_v \cdot effective\_sample\_size}$.

At the end of each iteration, we update α with the last gradient computed at the data point indexed by p (line 28), and the sum of local historical gradients incrementally (line 27). When all the consumer threads complete the dispatched iterations of gradient descent, they wait for the update phase at the next time step, and the current effective sample set is referred to as T_i.

4 Theoretical Analysis

In this section, we analyze the accuracy of LFS-STRSAGA in terms of sub-optimality. According to the proof technique in previous work [10], the accuracy of a streaming data VR SGD algorithm depends on the convergence rate of its offline version. Therefore, we first present the offline version of LFS-STRSAGA, and then analyze its convergence rate in Section 4.1. Finally, we show that LFS-STRSAGA retains the linear convergence rate of STRSAGA to reach statistical accuracy in sparsity assumption in Section 4.2.

Following the assumptions in previous work [5], [9], [10], [16], we assume that all f_x are convex and their gradients are L-Lipschitz continuous. In addition, $\mathcal{R}_\mathcal{S}$ is μ-strongly convex for the training dataset $\mathcal{S}$.

4.1 Convergence Analysis for Offline Version of LFS-STRSAGA

The proof technique of STRSAGA relies on the convergence rate of SAGA, which can be seen as the offline version of STRSAGA. The convergence rate of SAGA is 1-min($\frac{1}{n}, \frac{\mu}{L}$). Following this proof technique, we first define the offline version of LFS-STRSAGA as follows:

Offline Version of LFS-STRSAGA. For the offline version of LFS-STRSAGA, the entire training dataset $\mathcal{S}$ is available beforehand. Thus, the effective sample set is fixed with size n. The offline version of LFS-STRSAGA starts with a known initial vector of weights $\mathbf{w}^0$, and each processor core c uses a table $\alpha(c)$ to store local historical gradients. The length of each $\alpha(c)$ is equal to the number of training data points. Each element $\alpha(c, i)$ in $\alpha(c)$ is initialized to 0. Following Recht *et al.* [9], we define $\mathbf{w}^t$ to be the state of $\mathbf{w}$ after t updates have been performed. Intuitively, t refers to the global iteration counts. As $\mathbf{w}^t$ is generally updated with a stale gradient based on the state of $\mathbf{w}$ read many clock cycles earlier, we use $\mathbf{w}^{k(t)}$ to denote the state of $\mathbf{w}$ when the update to $\mathbf{w}^t$ was read. Assuming the update step is run on the c^tth core in the global tth iteration, the updates for iteration $t+1$ in the offline version of LFS-STRSAGA are as follows:

1. Sample a point i uniformly from $\mathcal{S}$ at random.
2. Compute the gradient $\nabla f_i(\mathbf{w}^{k(t)})$ at point i.
3. Update $\mathbf{w}$ using $\nabla f_i(\mathbf{w}^{k(t)})$, $\alpha(c^t, i)$ and the average of local historical gradients: $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \eta g(\mathbf{w}^{k(t)}, c^t, i)$, where $g(\mathbf{w}^{k(t)}, c^t, i) \leftarrow \nabla f_i(\mathbf{w}^{k(t)}) - \alpha(c^t, i) + D_i \overline{\alpha}(c^t)$, $\alpha(c^t, i)$ denotes the last gradient computed locally on the c^tth core at the data point indexed by i, $\overline{\alpha}(c^t) := \frac{sum}{n}$ denotes the average of historical gradients maintained on the c^tth core, and D_i is a diagonal matrix equal to $\frac{1}{e_v}$ on the vth diagonal and zeros elsewhere ($v \in e(i)$). The definitions of e_v and $e(i)$ are given in Table 1.
4. Update $\alpha(c^t, i)$, and other local historical gradients remain unchanged: $\alpha(c^t, i) \leftarrow \nabla f_i(\mathbf{w}^{k(t)})$

There are three differences between the offline version of LFS-STRSAGA and SAGA: (1) the offline version of LFS-STRSAGA uses the local historical gradients and their sparse average at each iteration of gradient descent, rather than the full average of historical gradients like SAGA; (2) co-running threads in the offline version of LFS-STRSAGA lead to that $\mathbf{w}^t$ is generally updated with a stale gradient based on the state of $\mathbf{w}$ read many clock cycles earlier; (3) threads could overwrite each other's progress of updating the decision variable, which cannot occur in the serial SAGA. In the following, we will prove that the offline version of LFS-STRSAGA has the same convergence rate as SAGA in the big data regime under sparsity assumption.

Unbiased Condition for an Update Scheme. The unbiased condition is the heart of most convergence proofs for the asynchronous parallel SGD and its variants. To prove that the update schemes adopted in the offline version of LFS-STRSAGA are unbiased, we need to show $\mathbb{E}g(\mathbf{w}^{k(t)}, c^t, i) = \nabla f(\mathbf{w}^{k(t)})$, where $\nabla f(\mathbf{w}^{k(t)})$ is a full-data gradient: $\nabla f(\mathbf{w}^{k(t)}) = \frac{1}{n}\sum_{k=1}^{n} \nabla f_k(\mathbf{w}^{k(t)})$.

Lemma 1: $\mathbb{E}g(\mathbf{w}^{k(t)}$, c^t, i) = $\nabla f(\mathbf{w}^{k(t)})$ is satisfied in LFS-STRSAGA.

Proof: Suppose the total number of cores is c. For LFS-STRSAGA, we have: $\mathbb{E}D_i\overline{\alpha}(c^t) = \mathbb{E}\{\mathbb{E}[D_i\overline{\alpha}(c^t)|(e_1, e_2, \ldots, e_d)]\} = \mathbb{E}\{[\frac{1}{n}\sum_{k=1}^{n} D_k\overline{\alpha}(c^t)]|(e_1, e_2, \ldots, e_d)\} = \frac{1}{n}\sum_{k=1}^{n}\sum_{v\in e(k)} \frac{\overline{\alpha}_v(c^t)}{\mathbb{E}e_v} x_v = \sum_{v=1}^{d} \frac{d_v\overline{\alpha}_v(c^t)x_v}{n\mathbb{E}e_v}$. Note that d_v denotes the number of training data points (in $\mathcal{S}$) equal to non-zero on the vth coordinate. $x_v \in \mathcal{R}^d$ is equal to 1 on the vth coordinate and zeros elsewhere. According to the definition given in Table 1, the values of $e_1, e_2, \ldots, e_d$ depend on the data points picked for gradient computation in the previous iterations. Because these data points are picked uniformly from the sample set, we have: $\mathbb{E}e_v = \frac{d_v}{n}$ $(v = 1, 2, \ldots, d)$. Thus, $\mathbb{E}D_i\overline{\alpha}(c^t) = \overline{\alpha}(c^t)$. For any c^t, $\mathbb{E}\alpha(c^t, i) = \frac{1}{n}\sum_{k=1}^{n} \alpha(c^t, k) = \overline{\alpha}(c^t)$. Then, $\mathbb{E}g(\mathbf{w}^{k(t)}, c^t, i) = \mathbb{E}\nabla f_i(\mathbf{w}^{k(t)})$ - $\mathbb{E}\alpha(c^t, i) + \mathbb{E}D_i\overline{\alpha}(c^t) = \mathbb{E}\nabla f_i(\mathbf{w}^{k(t)}) = \frac{1}{n}\sum_{k=1}^{n}\nabla f_k(\mathbf{w}^{k(t)}) = \nabla f(\mathbf{w}^{k(t)})$. This completes the proof. □

Following Recht *et al.* [9], for any $\mathbf{w}^t$ and $\mathbf{w}^{k(t)}$, we define the bound of $t - k(t)$ as τ. In addition, we define Δ as: $\Delta = \frac{max_{1\leq v\leq d}|\{i:v\in e(i)\}|}{n}$, i.e., the maximum frequency that any feature appears in a training data point. With Lemma 1 holding, we reach the following lemma.

Lemma 2: *(Combining Theorem 2 and Corollary 3 in [15])* Suppose $\tau \leq O(n)$ and $\tau \leq O(\frac{1}{\sqrt{\Delta}} max\{1, \frac{n}{\kappa}\})$. Let $a^*(\tau) = \frac{1}{32(1+\tau\sqrt{\Delta})\xi(\kappa,\Delta,\tau)}$, where $\xi(\kappa, \Delta, \tau) = \sqrt{1 + \frac{1}{8\kappa} min\{\frac{1}{\sqrt{\Delta}}, \tau\}}$ and $\kappa = \frac{L}{\mu}$. Then using the step size $\eta = \frac{a^*(\tau)}{L}$, The offline version of LFS-STRSAGA has the same convergence rate as SAGA: $\boldsymbol{E}\big[f(w^{k(t)}) - f(w^*)\big] \leq \rho^t C_0$, where $\rho = 1 - min\{\frac{1}{n}, \frac{1}{\kappa}\}$, and C_0 is a constant independent of t.

When a machine learning problem is very sparse, the precondition of Lemma 2 is easily satisfied because Δ is a very small value in this case. Therefore, the offline version of LFS-STRSAGA has the same convergence rate as SAGA in sparsity assumption, which is a common phenomenon in machine learning problems.

4.2 Sub-Optimality of LFS-STRSAGA

Now, with sparsity assumption, we analyze the accuracy of LFS-STRSAGA in terms of the expected sub-optimality.

Metric for Sub-Optimality. Following Jothimurugesan *et al.* [10], we define the sub-optimality of an algorithm A over the training dataset $\mathcal{S}$ as follows:

$$SUBOPT_{\mathcal{S}}(A) = \mathcal{R}_{\mathcal{S}}(\mathbf{w}) - \mathcal{R}_{\mathcal{S}}(\mathbf{w}^*_{\mathcal{S}}), \tag{1}$$

where $\mathbf{w}$ is the solution produced by A and $\mathbf{w}^*_{\mathcal{S}}$ is the optimal solution over $\mathcal{S}$.

To prove the sub-optimality of LFS-STRSAGA, we use the upper bound function $\mathbf{U}$ defined in [5]

$$\mathbf{U}(t, n) = min\begin{cases} \rho_n\mathbf{U}(t-1, n) \\ min_{m<n}[\mathbf{U}(t, m) + \frac{n-m}{n}\mathcal{H}(m)] \end{cases}. \tag{2}$$

For a given function $\mathbf{U}(x, y)$, x denotes the total number of iterations that have been performed, and y denotes the effective sample size at the end of iteration x. When using the $\mathbf{U}$ function to analyze the sub-optimality of LFS-STRSAGA, ρ_n is the convergence rate of the offline version of LFS-STRSAGA for the effective sample set with size n.

Let T_i denote the effective sample set at the end of all update steps started at time step i in LFS-STRSAGA, and let t_i^L denote the size of T_i. We reach the following lemma.

Lemma 3: *Suppose the condition number $\frac{L}{\mu}$ and the initial sub-optimality of LFS-STRSAGA is bounded by a constant.* At the end of each time step i, the expected sub-optimality of LFS-STRSAGA over T_i is: $\mathbb{E}\big[SUBOPT_{T_i}(LFS-STRSAGA)\big] \leq (1 + O(1))\mathcal{H}(t_i^L)$.

Proof: In LFS-STRSAGA, each thread adds a new data point into the effective sample set every two iterations by a FAA. Suppose the total number of iterations performed by all the consumer threads is t_i at the end of each time step i. The atomic feature of FAA ensures that $t_i \geq 2t_i^L$. According to Equation (2), we have $\mathbb{E}\big[SUBOPT_{T_i}\ (LFS - STRSAGA)\big] \leq \mathbf{U}(t_i, t_i^L) \leq \rho^{t_i - 2t_i^L}\mathbf{U}(2t_i^L, t_i^L)$. According to Lemma 2, $0 < \rho < 1$. Therefore, we have $\mathbb{E}\big[SUBOPT_{T_i}\ (LFS - STRSAGA)\big] \leq \mathbf{U}(2t_i^L, t_i^L)$. Similar to the proof of Lemma 3 in [10], we have $\mathbb{E}\big[SUBOPT_{T_i}(LFS - STRSAGA)\big] \leq \mathcal{H}(t_i^L) + \frac{C_0}{2}(\frac{L}{\mu})^2(\frac{1}{t_i^L})^2 = (1 + O(1))\mathcal{H}(t_i^L)$, where C_0 denotes the initial sub-optimality of LFS-STRSAGA. This completes the proof. □

Lemma 3 bounds the expected sub-optimality of LFS-STRSAGA over its effective sample set. However, our goal is to bound the expected sub-optimality of LFS-STRSAGA over $\mathcal{S}_i$, where $\mathcal{S}_i$ denotes the data points have been observed till the time step i. Let $T_i^{'}$ denote the effective sample set at the end of all update steps started at time step i in STRSAGA, and let t_i^S denote the size of $T_i^{'}$. Lemma 3 in [10] shows that $\mathbb{E}[SUBOPT_{T_i^{'}}(STRSAGA)] \leq (1 + O(1))\mathcal{H}(t_i^S)$. Both LFS-STRSAGA and STRSAGA add a new data point into the effective sample set every two iterations if the buffer is not empty. Thus, when LFS-STRSAGA and STRSAGA produce solutions over the same $\mathcal{S}_i$, $t_i^L \approx t_i^S$. The proof used to bound the expected sub-optimality of STRSAGA over $\mathcal{S}_i$ also applies to LFS-STRSAGA. Therefore, LFS-STRSAGA retains the linear convergence rate of STRSAGA to reach the statistical accuracy over $\mathcal{S}_i$ in sparsity assumption.

5 Experimental Results

In this section, we evaluate the competitiveness of LFS-STRSAGA in terms of model update time and sub-optimality, and compare it to the state-of-the-art SGD algorithms and variance-reduced improvements. All the algorithms are implemented in C++, and the experiments are run on a machine consisting of four eight-core 2.0 GHz Intel Xeon E7-4809 v3 processors. Each core supports two hyper-threads.

5.1 Experimental Methodology

We compare LFS-STRSAGA to the following algorithms:

- STRSAGA: A state-of-the-art serial VR SGD algorithm.
- S-HOGWILD!: A natural streaming data version of HOGWILD!. To make the comparison fair, S-HOGWILD! also performs ρ gradient computations to update the model at each time step i, and adopts the same sample scheme as LFS-STRSAGA. Note that our sample scheme is better than the original sample scheme used in HOGWILD!.

TABLE 2
Datasets for Logistic Regression and Step Sizes Used in Experiments

Dataset	Number of Training Data Points	Number of Features	Density	Steps Size η
RCV1	677399	47236	0.15%	0.5
URL	2396130	3231961	0.004%	0.005

- LF-STRSAGA: An algorithm that can be seen as a middle ground between LFS-STRSAGA and S-HOGWILD!. The only difference between LF-STRSAGA and LFS-STRSAGA is that a thread uses the full average of local historical gradients in LF-STRSAGA at an update step.

We take STRSAGA as the performance and accuracy benchmarking because it is the original building blocks of LFS-STRSAGA. S-HOGWILD! is selected as a competitor because HOGWILD! is the foundation for the most parallel implementations of SGD and its variants. To show the impact of the sparse processing for the average of local historical gradients, we implement LF-STRSAGA and compare it to LFS-STRSAGA.

Benchmarks. We evaluate all the algorithms on two widely used machine learning problems: *Logistic Regression* and *Matrix Factorization*. For a data point in the logistic regression problem, the L2-regularized logistic loss function is: $f_{(x,y)}(\mathbf{w}) = log(1 + exp(-y\mathbf{w}^T x)) + \frac{\mu}{2}||\mathbf{w}||_2^2$. We consider the matrix factorization problem of finding two rank-10 matrices. For a data point M_{ij} in the matrix factorization problem, the regularized loss function is: $f_{(i,j)}(\mathbf{w}) = ((LR^T)_{ij} - M_{ij})^2 + \frac{\mu}{2}(||L||_F^2 + ||R||_F^2)$. For the logistic regression problem, we use 2 real-world datasets (RCV1 [17] and URL [18]) to evaluate all the algorithms, while using 1 real-world dataset (MovieLens1M [19]) for the matrix factorization problem. Tables 2 and 3 describe the details of the datasets and the step sizes used in experiments. We obtain the optimal step size using grid search on single-threaded SGD after a single pass of each dataset. In the following, we refer to the logistic regression problem using RCV1 and URL datasets as RCV1 and URL benchmarks, respectively. Similarly, the matrix factorization problem using MovieLens1M dataset is referred to as MovieLens1M benchmark. In all the benchmarks, threads are evenly divided into dedicated producers and consumers. If a thread acts as a dedicated producer, it only receives and stores new data points at different time steps. Otherwise, the thread only updates the model decision variables at different time steps.

Arrival Distributions. Note that the datasets given in Tables 2 and 3 are static. Following Jothimurugesan *et al.* [10], we convert these static training data points into streams. For each dataset in our experiments, all the training data points arrive over the course of 100 time steps, and the number of data points arriving in each time step i is a random variable x_i. Our experiments are conducted under two arrival distributions: *skewed arrivals* and *Poisson arrivals*. We choose the skewed arrivals and Poisson arrivals because they represent two commonly arrival patterns. The skewed arrival models bursty arrivals, which combines a number of time steps without arrivals of new data points and occasional bursts of new data points, reflecting the worst case for streaming data algorithms to deal with at limited processing rates. Meanwhile, a large number of arrival events follow the Poisson distribution in real scenarios. In skewed arrivals, $x_i = M$ with a probability $\frac{\lambda}{M}$, and $x_i = 0$ with a probability $1 - \frac{\lambda}{M}$, where $M > 0$ is an integer. In Poisson arrivals, x_i follows a Poisson distribution. In both arrivals, $\mathbb{E}[x_i] = \lambda$, where $\lambda = \frac{Number\ of\ Training\ Data\ Points}{Number\ of\ Time\ Steps}$.

5.2 Model Update Time

In this subsection, we present our experimental results for the model update time of different algorithms over the course of 100 time steps in all the benchmarks. For a given benchmark, the model update time of each algorithm running with different number of consumer threads is normalized to the model update time of STRSAGA. To avoid the scalability problem of parallel SGD-style algorithms on modern multicore machine with *Non-Uniform Memory Access* (NUMA) architecture [20], we allocate all the consumer threads to one processor and all the producer threads to another processor. In all the benchmarks, the number of consumer threads is up to 16, which is the maximum parallelism of one processor on our testbed like [9], [21]. Fig. 3 shows the normalized model update time of different algorithms for all the benchmarks under Poisson arrivals. The lower update time indicates the better performance.

As shown in Fig. 3, with the number of consumer threads increasing, the model update time of LF-STRSAGA is significantly reduced for all the benchmarks. Benefit from the sparse processing on the average of historical gradients, LFS-STRSAGA further reduces the model update time of LF-STRSAGA by over 98 percent. This is because the tested datasets are sparse and LFS-STRSAGA only considers the average of historical gradients on a small fraction of the coordinates when updating the decision variable, yielding less computation expenses than LF-STRSAGA. The time complexity of LFS-STRSAGA and LF-STRSAGA depends on the machine learning problem and the characteristics of datasets. In logistic regression problem, assuming d and *density* are the number of features and the average sparse density of a given dataset, the time complexity of updating the decision variables w is O(*density*) in LFS-STRSAGA and

TABLE 3
Dataset for Matrix Factorization and Step Sizes Used in Experiments

Dataset	Users	Movies	Date Range	Rating Scale	Density	Steps Size η
MovieLens1M	6040	3952	4/2000-2/2003	1-5, stars	4.47%	0.005

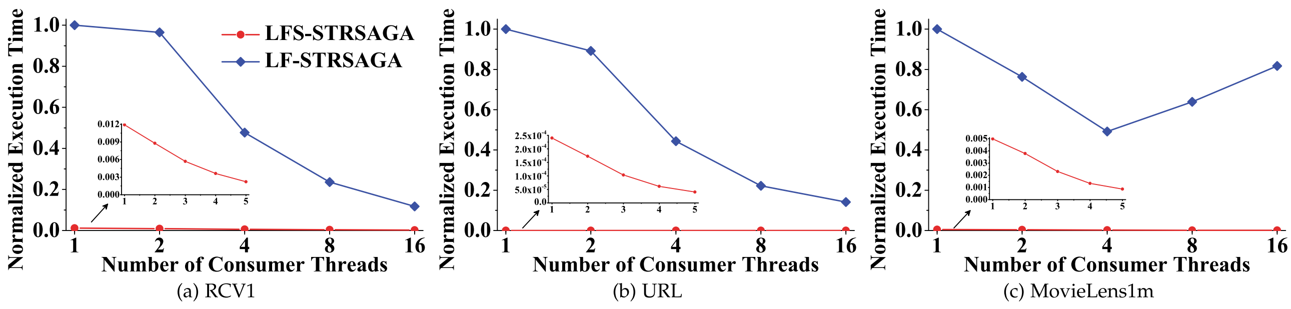


Fig. 3. Normalized update time of different algorithms under Poisson arrivals for (a) RCV1, (b) URL, and (c) MovieLens1m.

$O(d)$ in LF-STRSAGA. In matrix factorization problem, assuming *row*, *col*, *rank* are the row, column, rank numbers of a given data matrix, the time complexity of updating the decision variables w is $O(rank)$ in LFS-STRSAGA and $O(row*rank+col*rank)$ in LF-STRSAGA. The experimental results under skewed arrivals have the similar figures.

5.3 Sub-Optimality

In this subsection, we evaluate the accuracy of different algorithms in terms of their sub-optimality over the data points that have been observed till each time step i. The metric for sub-optimality is defined in Equation (1) (as shown in Section 4.2). Lower sub-optimality indicates higher accuracy.

Figs. 4 and 5 show the sub-optimality of different algorithms for all the benchmarks under skewed arrivals and Poisson arrivals, respectively. In both figures, the top row indicates the case in which all the algorithms (except STRSAGA) run with 1 consumer thread, and the bottom row indicates the case in which they run with 16 consumer threads. Because LF-STRSAGA with 1 consumer thread is equivalent to STRSAGA, we only take STRSAGA as the accuracy benchmarking in the case that other algorithms run with 16 consumer threads. We can observe that both LFS-STRSAGA and LF-STRSAGA significantly outperform S-HOGWILD! in most cases because they benefit from the faster convergence rate by using variance-reduced method. At some time steps, the sub-optimality spikes because many new data points arrive now and the number of data points that have yet to be processed bursts. Because the sparse processing on the average of historical gradients loses some information from other coordinates, LFS-STRSAGA is slightly less accurate than LF-STRSAGA in some cases. When the number of consumer threads reaches to 16, the sub-optimality of LFS-STRSAGA almost matches the sub-optimality of STRSAGA for RCV1 and URL in most cases, but is relatively bigger for movieLens1m. This is because the movielens1m dataset is more dense than another two datasets. More dense the dataset is, more update conflicts among threads will occur.

Summary. LFS-STRSAGA almost matches the accuracy of STRSAGA. Although LFS-STRSAGA is less accurate than LF-STRSAGA, it has a big advantage in the model update time. Because maintaining an accurate model on streaming data is limited in processing time, LFS-STRSAGA is a better alternative than LF-STRSAGA.

6 Related Work

Our work is related to the research on gradient descent methods and their parallel implementations. In this section, we briefly discuss the most related work in turn.

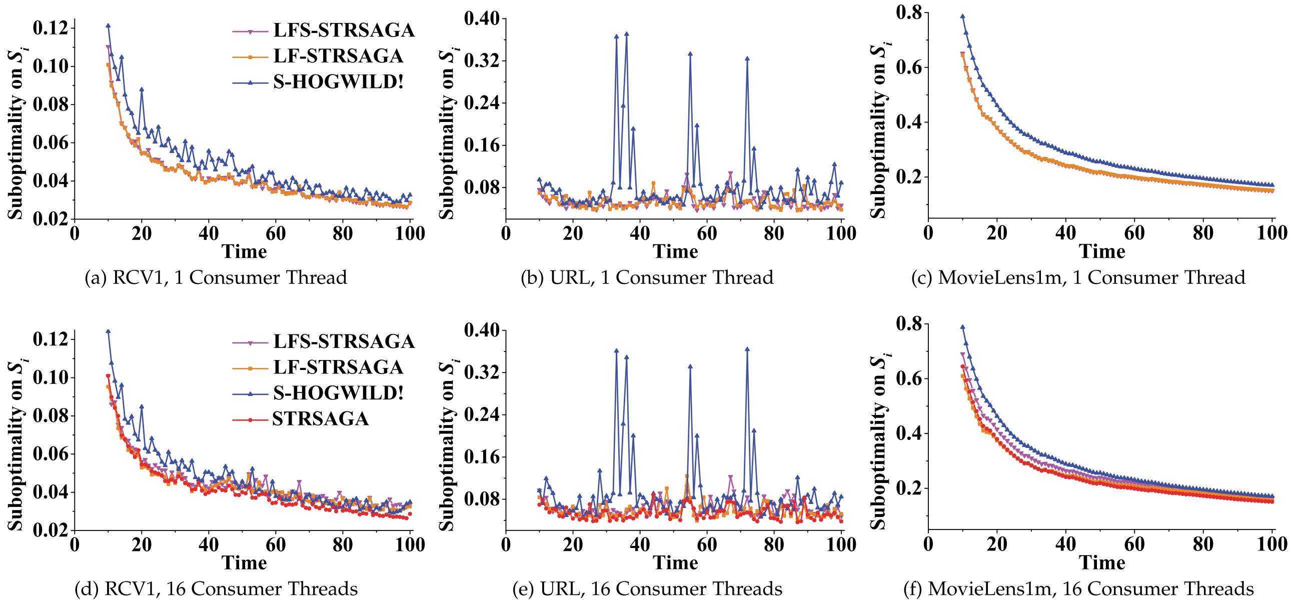
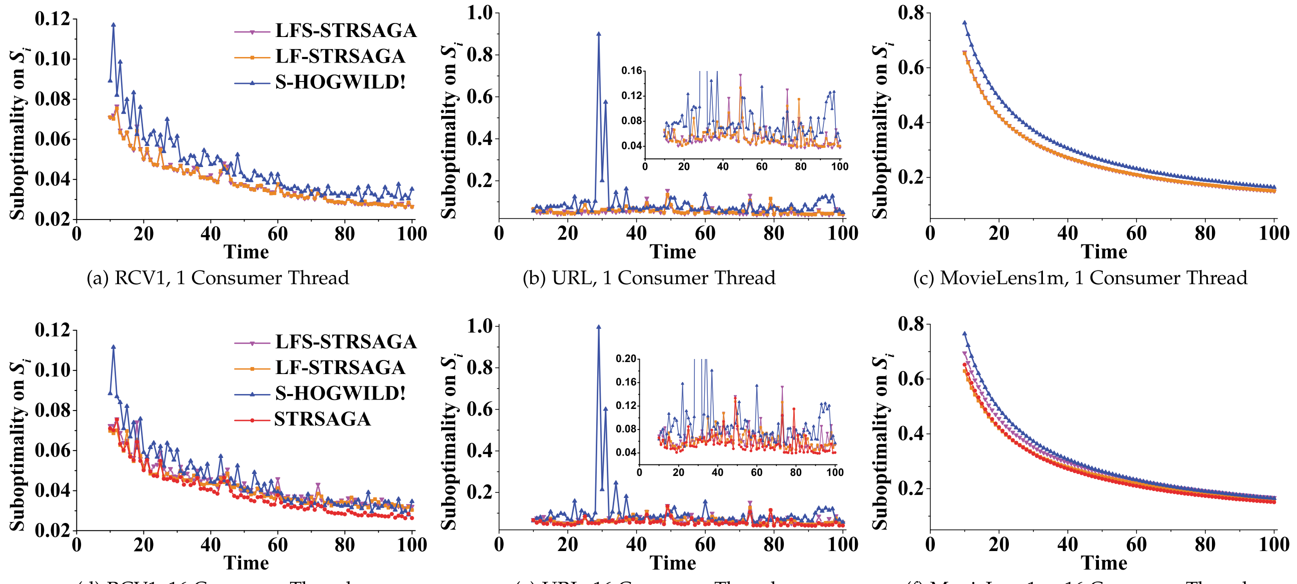


Fig. 4. Sub-optimality under skewed arrivals with $M = 8\lambda$.



(a) RCV1, 1 Consumer Thread (b) URL, 1 Consumer Thread (c) MovieLens1m, 1 Consumer Thread

(d) RCV1, 16 Consumer Threads (e) URL, 16 Consumer Threads (f) MovieLens1m, 16 Consumer Threads

Fig. 5. Sub-optimality under Poisson arrivals with mean λ.

6.1 Gradient Descent Methods

Gradient descent methods are widely used to pursue model parameters in the training phase of machine learning problems [22], [23]. According to the amount of data points used at each update step, gradient descent methods are generally divided into two classes: *Batching Gradient Descent* (BGD) and *Stochastic Gradient Descent* (SGD) [1]. BGD computes the full-data gradient to determine the update direction at each iteration of gradient descent. In contrast, SGD selects a single data point randomly from the training dataset to compute the gradient, requiring much less computation than BGD but missing the information from other data points. Thus, the variance of SGD can be high. To solve the problem, variance-reduced variants of SGD are proposed generally by incorporating a correction term that approximates a full-data gradient at each iteration of gradient descent [3], [4], [6], [7], [8], [24].

Roux *et al.* propose SAG [3], a representative variance-reduced method that stores the last gradient computed for each data point and uses their average at each iteration of gradient descent. Inspired from SAG, Defazio *et al.* propose SAGA [4], which optimizes SAG by eliminating a bias in the update step. Johnson *et al.* propose SVRG [6] that periodically computes a full-data gradient without the storage of historical gradients. Obviously, SVRG requires more computation than SAG and SAGA. To reduce the computation cost, Konečný *et al.* propose Semi-Stochastic Gradient Descent (S2GD) [7] based on SVRG. It uses a random-data-gradient and a full-data gradient at each iteration occasionally. Shah *et al.* propose another variant of SVRG, named CHEAPSVRG [8], which computes the gradient on a subset of training data points.

To improve the convergence rate of variance-reduced methods, Daneshmand *et al.* suggest to gradually increase the size of effective sample set [5]. They apply this idea to SAGA and propose DYNASAGA [5]. As a result, DYNASAGA achieves the statistical accuracy in $O(n)$ iterations, while all of the above methods require $O(nlogn)$ iterations. Jothimurugesan *et al.* extend DYNASAGA to maintain a model over streaming data, and propose STRSAGA [10]. STRSAGA significantly outperforms another state-of-the-art streaming version of VR SGD named Streaming SVRG (SSVRG) [25]. Because fitting an accurate model over streaming data requires the optimization algorithm to be time-efficient, we present LFS-STRSAGA, which can be seen as a lock-free parallelization of STRSAGA. Compared to STRSAGA, LFS-STRSAGA almost matches the accuracy of STRSAGA under sparsity assumption, and LFS-STRSAGA reduces the model update time by over an order of magnitude.

6.2 Parallelization of SGD and its Variants

Parallelization is a crucial requirement for large-scale applications. In parallel setting, many asynchronous variants of SGD are proposed on both a single multicore machine and distributed systems. The seminal text presented by Bertsekas and Tsitsiklis [26] is the foundation for the most work on parallelizing SGD and its variants. The authors present ideas that using the stale gradient computed across many computers in a master-worker setting at each update step and different processors only access particular components of the decision variable. Inspired from these ideas, Niu *et al.* propose a lock-free approach to parallelizing stochastic gradient descent named HOGWILD! [9]. The update scheme of HOGWILD! allows processors to directly modify the decision variable without using any locking at each iterations of gradient descent. When most updates only modify small and disjoint parts of the decision variable, HOGWILD! achieves a nearly optimal rate of convergence. Nguyen *et al.* further proves the convergence of HOGWILD! without the bounded gradients assumption [27], and Alistarh *et al.* proves the convergence of parallel SGD in asynchronous shared memory [28]. The key idea of HOGWILD! is extended to the most work on lock-free parallelization of VR SGD including the asynchronous variants of coordinate descent [29], SVRG [21], and SAGA [15]. So far, all the parallel implementations of SGD-like methods focus on the offline algorithms, in which the entire training dataset is assumed available beforehand. In this paper, we focus on streaming data arriving as an endless sequence of data points. In this setting, because the size of training dataset changes over time, it is more challenging to parallelize VR SGD with high performance and accuracy. The main challenge of parallelizing VR SGD on streaming data is how to store new data points and sample from dynamically changing dataset in parallel. The research work on non-blocking data structures [13], [14], [30] inspires the design of the data structures adopted in LFS-STRSAGA.

7 Conclusion and Future Work

In this paper, we present a lock-free approach to parallelizing the VR SGD algorithm on streaming data. Specifically, we have focused on a modification of STRSAGA by presenting a lock-free data structure to process the arrival of streaming data in parallel, and asynchronously maintain the essential information to approximate the full-data gradients with low cost. Both our theoretical and empirical results show that LF-STRSAGA matches the accuracy of the state-of-the-art VR SGD algorithm on streaming data in sparse setting that is common to machine learning problems, and that it reduces the model update time by over an order of magnitude.

This paper focuses on the design of LFS-STRSAGA for the settings where memory is not limited. We will explore how to restrict LFS-STRSAGA to use limited memory in our future work. For example, we can set a maximum size for the effective sample set. When the number of the data points added to the effective sample set reaches to the maximum size, some data points will be removed from the effective sample set. One problem is how to choose the data points for removal, and an alternative scheme is to remove most sampled data points because a relatively sufficient number of SAGA steps have been performed on these data points. Another problem is how to remove any data point from the effective sample set without using any locking. In addition, we also consider to enhance the scalability of LFS-STRSAGA on NUMA systems.

Acknowledgments

This work was supported by National Natural Science Foundation of China under Grant 61702499. The authors would like to thank the anonymous reviewers for their useful suggestions which helped to improve the quality of the article.

References

[1] H. Robbins and S. Monro, “A stochastic approximation method,” *Ann. Math. Statist.*, vol. 22, no. 3, pp. 400–407, 1951.
[2] L. Bottou and Y. L. Cun, “Large scale online learning,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2004, pp. 217–224.
[3] N. L. Roux, M. Schmidt, and F. Bach, “A stochastic gradient method with an exponential convergence rate for finite training sets,” in *Proc. 25th Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 2663–2671.
[4] A. Defazio, F. Bach, and S. Lacoste-Julien , “SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives,” in *Proc. 27th Int. Conf. Neural Inf. Process. Syst.*, 2014, pp. 1646–1654.
[5] H. Daneshm, and A. Lucchi, and T. Hofmann, “Starting small: Learning with adaptive sample sizes,” in *Proc. 33rd Int. Conf. Int. Conf. Mach. Learn.*, 2016, pp. 1463–1471.
[6] R. Johnson and T. Zhang, “Accelerating stochastic gradient descent using predictive variance reduction,” in *Proc. 26th Int. Conf. Neural Inf. Process. Syst.*, 2013, pp. 315–323.
[7] J. Konečný and P. Richtárik, “Semi-stochastic gradient descent methods,” 2013, *arXiv:1312.1666*.
[8] V. Shah, M. Asteris, A. Kyrillidis, and S. Sanghavi, “Trading-off variance and complexity in stochastic gradient descent,” 2016, *arXiv:1603.06861*.
[9] B. Recht, C. Re, S. Wright, and F. Niu, “HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2011, pp. 693–701.
[10] E. Jothimurugesan, A. Tahmasbi, P. Gibbons, and S. Tirthapura, “Variance-reduced stochastic gradient descent on streaming data,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2018, pp. 9906–9915.
[11] L. Bottou and O. Bousquet, “The tradeoffs of large scale learning,” in *Proc. 20th Int. Conf. Neural Inf. Process. Syst.*, 2007, pp. 161–168.
[12] M. P. Herlihy and J. M. Wing, “Linearizability: A correctness condition for concurrent objects,” *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.
[13] C. Yang and J. Mellor-Crummey , “A wait-free queue as fast as fetch-and-add,” in *Proc. 21st ACM SIGPLAN Symp. Princ. Practice Parallel Program.*, 2016, pp. 16:1–16:13.
[14] Y. Peng and Z. Hao, “FA-Stack: A fast array-based stack with wait-free progress guarantee,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 4, pp. 843–857, Apr. 2018.
[15] R. Leblond, F. Pedregosa, and S. Lacoste-Julien , “ASAGA: Asynchronous parallel SAGA,” in *Proc. 20th Int. Conf. Artif. Intell. Statist.*, 2017, pp. 46–54.
[16] A. Nemirovski, A. Juditsky, G. Lan, and A. A Shapiro, “Robust stochastic approximation approach to stochastic programming,” *Soc. Ind. Appl. Math.*, vol. 19, no. 4, pp. 1574–1609, 2009.
[17] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li, “RCV1: A new benchmark collection for text categorization research,” *J. Mach. Learn. Res.*, vol. 5, pp. 361–397, Dec. 2004.
[18] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker, “Identifying suspicious URLs: An application of large-scale online learning,” in *Proc. 26th Annu. Int. Conf. Mach. Learn.*, 2009, pp. 681–688.
[19] F. M. Harper and J. A. Konstan, “The MovieLens datasets: History and context,” *ACM Trans. Interactive Intell. Syst.*, vol. 5, no. 4, pp. 19:1–19:19, Dec. 2015.
[20] H. Zhang, C. Hsieh, and V. Akella, “HogWild++: A new mechanism for decentralized asynchronous stochastic gradient descent,” in *Proc. 16th Int. Conf. Data Mining*, 2016, pp. 629–638.
[21] S. J. Reddi, A. Hefny, S. Sra, B. Poczos, and A. J. Smola, “On variance reduction in stochastic gradient descent and its asynchronous variants,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2015, pp. 2647–2655.
[22] J. Lee *et al.*, “Wide neural networks of any depth evolve as linear models under gradient descent,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 8572–8583.
[23] S. Goldt, M. Advani, A. M. Saxe, F. Krzakala, and L. Zdeborová, “Dynamics of stochastic gradient descent for two-layer neural networks in the teacher-student setup,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 6981–6991.
[24] T. Hofmann, A. Lucchi, S. Lacoste-Julien , and B. McWilliams , “Variance reduced stochastic gradient descent with neighbors,” in *Proc. 28th Int. Conf. Neural Inf. Process. Syst.*, 2015, pp. 2305–2313.
[25] R. Frostig, G. Rong, S. M. Kakade, and A. Sidford, “Competing with the empirical risk minimizer in a single pass,” in *Proc. 28th Annu. Conf. Learn. Theory*, 2015, pp. 728–763.
[26] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*. Upper Saddle River, NJ, USA: Prentice-Hall, 1989.
[27] L. Nguyen, P. H. Nguyen, M. van Dijk, P. Richtarik, K. Scheinberg, and M. Takac, “SGD and HogWild! Convergence without the bounded gradients assumption,” in *Proc. 35th Int. Conf. Mach. Learn.*, 2018, pp. 3750–3758.
[28] D. Alistarh, C. De Sa , and N. Konstantinov, “The convergence of stochastic gradient descent in asynchronous shared memory,” in *Proc. ACM Symp. Princ. Distrib. Comput.*, 2018, pp. 169–178.
[29] J. Liu, S. J. Wright, C. Ré, V. Bittorf, and S. Sridhar, “An asynchronous parallel stochastic coordinate descent algorithm,” in *Proc. 31st Int. Conf. Int. Conf. Mach. Learn.*, 2014, pp. 469–477.
[30] A. Morrison and Y. Afek, “Fast concurrent queues for x86 processors,” in *Proc. 18th ACM SIGPLAN Symp. Princ. Practice Parallel Program.*, 2013, pp. 103–112.

Yaqiong Peng (Member, IEEE) received the PhD degree from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2016. Currently, she is an assistant professor with the Institute of Information Engineering, Chinese Academy of Sciences. Her current research interests include parallel algorithms, operating systems, and virtualization. She has published ten papers in journals and conferences including the *IEEE Transactions on Parallel and Distributed Systems*, the *Future Generation Computing Systems*, CCGRID, IPCCC *et al.*

Zhiyu Hao received the PhD degree in computer system architecture from the Harbin Institute of Technology, Harbin, China, in 2007. He is currently a professor with the Institute of Information Engineering, Chinese Academy of Sciences. His research interests include network security, system virtualization, and network emulation.

Xiaochun Yun received the PhD degree from the Harbin Institute of Technology, Harbin, China, in 1998. He is currently a full professor with the Institute of Information Engineering, Chinese Academy of Sciences, China. He also works with the National Computer Network Emergency Response Technical Team/Coordination Center of China. His research interests include network and information security. He has authored more than 200 papers in refereed journals and conference proceedings.