Optimizing Streaming Parallelism on Heterogeneous Many-Core Architectures

Peng Zhang, Jianbin Fang, Canqun Yang, Chun Huang, Tao Tang, and Zheng Wang

Abstract—As many-core accelerators keep integrating more processing units, it becomes increasingly more difficult for a parallel application to make effective use of all available resources. An effective way of improving hardware utilization is to exploit spatial and temporal sharing of the heterogeneous processing units by multiplexing computation and communication tasks – a strategy known as heterogeneous streaming. Achieving effective heterogeneous streaming requires carefully partitioning hardware among tasks, and matching the granularity of task parallelism to the resource partition. However, finding the right resource partitioning and task granularity is extremely challenging, because there is a large number of possible solutions and the optimal solution varies across programs and datasets. This article presents an automatic approach to quickly derive a good solution for hardware resource partition and task granularity for task-based parallel applications on heterogeneous many-core architectures. Our approach employs a performance model to estimate the resulting performance of the target application under a given resource partition and task granularity configuration. The model is used as a utility to quickly search for a good configuration at runtime. Instead of hand-crafting an analytical model that requires expert insights into low-level hardware details, we employ machine learning techniques to automatically learn it. We achieve this by first learning a predictive model offline using training programs. The learned model can then be used to predict the performance of any unseen program at runtime. We apply our approach to 39 representative parallel applications and evaluate it on two representative heterogeneous many-core platforms: a CPU-XeonPhi platform and a CPU-GPU platform. Compared to the single-stream version, our approach achieves, on average, a 1.6x and 1.1x speedup on the XeonPhi and the GPU platform, respectively. These results translate to over 93 percent of the performance delivered by a theoretically perfect predictor.

Index Terms—Heterogeneous computing, parallelism, performance tuning, machine learning

1 Introduction

Heterogeneous many-cores, as representative by GPGPUs and Intel's XeonPhi, are widely used for accelerating parallel applications [1], [2], [3]. As users demand higher performance, many-core accelerators have become more powerful by providing more and more processing units. While the abundant computing resources offer the potential for higher performance, it becomes harder for a parallel application to utilize all the available computing resources [4], [5]. As a result, many parallel applications fail to fully unlock the performance potential of a many-core accelerator.

One way for improving heterogeneous many-core utilization is to exploit spatial and temporal sharing of processing resources. This strategy is also known as *heterogeneous streaming* [6]. The idea is to exploit the computation and communication independency of task parallelism to improve hardware utilization. It works by partitioning the processor cores to allow independent communication and computation tasks (i.e., streams) to run concurrently on different hardware resources, which effectively overlaps the concurrent kernel execution with data movements. Representative heterogeneous streaming implementations include CUDA Streams [7], OpenCL Command Queues [8], and Intel heterogeneous streams library (hStreams) [6], [9]. These implementations allow a parallel program to spawn more than one stream (or pipeline) so that the data movement stage of one pipeline overlaps the kernel execution stage of another.

Prior work on heterogeneous streaming mainly targets GPUs [10], [11], [12]. Compared to GPU implementations, OS-enabled coprocessors, like the Intel XeonPhi, provides some unique features that are currently unavailable on the GPU. For example, besides specifying the number of streams, developers can explicitly map streams to different groups of cores on XeonPhi to control the number of cores of each hardware partition. This parameter is not exposed to programmers on GPUs, making previous work on GPU-based parallel streaming optimizations infeasible to fully exploit XeonPhi-like many-core accelerators (see also Section 6.3). On the other hand, ample evidence is showing that choosing the right stream configuration, i.e., the number of processor core partitions and the number of concurrent tasks of a multi-stream application, values, has a significant impact the application's performance on many-core architectures [13], [14], [15]. However, attempting to find the optimal values through exhaustive profiling would be ineffective, because the range of the possible values for the two parameters is huge. What we need is a technique that automatically determines the optimal stream configuration for any streamed application in a fast manner.

- *P. Zhang, J. Fang, C. Yang, C. Huang, and T. Tang are with the National University of Defense Technology, Changsha, Hunan 410073, China. E-mail: {zhangpeng13a, j.fang, canqun, chunhuang, taotang84}@nudt.edu.cn.*
- *Z. Wang is with the University of Leeds, LS2 9JT Leeds, United Kingdom. E-mail: z.wang5@leeds.ac.uk.*

Manuscript received 24 Jan. 2019; revised 23 Feb. 2020; accepted 26 Feb. 2020. Date of publication 3 Mar. 2020; date of current version 1 Apr. 2020. (Corresponding author: Jianbin Fang.) Recommended for acceptance by P. Sadayappan. Digital Object Identifier no. 10.1109/TPDS.2020.2978045

1045-9219 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article presents a novel approach to determine the right number of processor core partitions and tasks for heterogeneous streams, targeting heterogeneous many-core architectures. Our key insight is to use a performance model to quickly search for the optimal stream configuration. The performance model estimates the resulting performance of the target streamed application when it runs under a given stream configuration. If the prediction can be performed quickly with low overhead, we can then quickly explore a large configuration space. Instead of hand-crafting the performance model that requires human modification whenever the architecture evolves (i.e., when the number and types of cores change), we employ machine learning techniques to automatically construct a predictive model. Our predictor is first trained *off-line*. Then, using code and dynamic runtime features of the program, the model predicts performance for a *new*, *unseen* program under a given stream configuration.

Our prior work [16] develops a machine learning based classifier to predict the optimal stream configuration. However, this approach can only choose from a limited set of configurations seen during the training phase. Unlike a classification-based approach, the approach presented in the article allows us to explore a larger number of stream configurations (including those that are not seen during the training phase) with negligible runtime overhead. This advantage significantly improves the generalization ability of the proposed approach (Section 3).

Due to the newness of heterogeneous streaming execution model, there are very few multi-stream benchmarks available. To evaluate our approach on a wide range of applications, we have developed a compiler-based tool to automatically translate standard OpenMP benchmarks into their streamed variants for the backends of XeonPhi and GPU architectures (Section 4). With the help of this code generator, we can apply our approach to 39 parallel benchmarks. We argue that this tool can help generate more streamed code and thus is an added value to the community.

We evaluate our approach on two representative heterogeneous many-core platforms: a 57-core Intel XeonPhi and an NVIDIA 1080Ti GPU platforms. We achieve, on average, a 1.6x and 1.1x speedup over the single-stream execution on the XeonPhi and the GPU platforms, respectively. This translates to over 93 percent of the best available performance.

The core contribution of this paper is a novel machine-learning-guided approach for automatically determining the optimal stream configuration on heterogeneous many-cores. We show that our approach delivers good performance across benchmarks and heterogeneous many-core platforms. While we do not seek to advance the machine learning algorithm itself, our work shows how machine learning can be used to address the challenging problem of tuning fine-grained streaming parallelism on heterogeneous many-core architectures. In this work, we demonstrate the usefulness of our approach on XeonPhi and an NVIDIA GPU, but our approach is equally applicable on other heterogeneous platforms like AMD GPUs.

2 Background and Overview

In this section, we first give a brief introduction of heterogeneous streaming; we then define the scope of this work, before motivating the need of our scheme and providing an overview of our approach.

```
1  //setting the partition-size and task granularity
   hStreams_app_init(partition_size,streams_p_part);
3
   //stream queue id
5  stream_id = 0;
   for(...){
7    //enquque host-device transfer to current stream
     hStreams_app_xfer_memory(,,, stream_id,
         ↪ HSTR_SRC_TO_SINK,...);
9    ...
     //enqueue computation to the current stream
11   hStreams_EnqueueCompute(stream_id, "kernel1", ...);
     ...
13   //move to the next stream
     stream_id = (stream_id++) % MAX_STR;
15 }
   //transfer data back to host
17 hStreams_app_xfer_memory(,,, HSTR_SINK_TO_SRC,...);
```

Fig. 1. Heterogeneous streaming using hStreams as an example.

2.1 Heterogeneous Streaming

The idea of heterogeneous streaming is to exploit spatial and temporal sharing of computing resources to utilize the hardware resources to improve application performance.

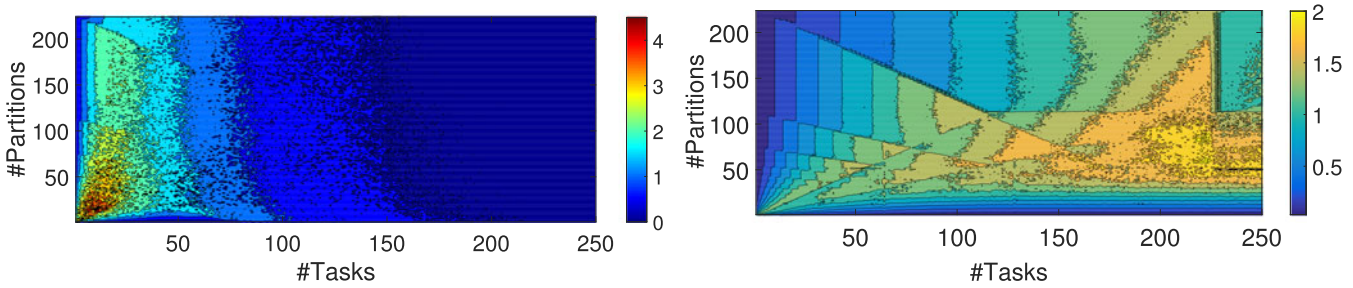
Spatial Sharing. Modern many-core accelerators offer a large number of processing units. Since many applications cannot fully utilize all the cores at a time, we can partition the computing units into multiple groups to concurrently execute multiple tasks. In this way, the computing resource is spatially shared across concurrently-running application tasks. The key to spatial sharing is to determine the right number of partitions, because over-provisioning of processing units would waste computing resources but under-provisioning would lead to slowed down performance.

Temporal Sharing. Code written for heterogeneous computing devices typically consists of several stages, such as host device communication and computation. Using temporal sharing, one can overlap some of these stages to exploit pipeline parallelism to improve performance by overlapping the host-device communication and kernel execution.

2.2 Problem Scope

Our work aims to improve the performance of a data parallel application by exploiting spatial and temporal sharing of heterogeneous streams. We do so by determining at runtime how many partitions should be used to group the cores (#*partitions*) and how many data parallel tasks (#*tasks*) should be used to run the application. Our current implementation is applicable to XeonPhi and GPUs by using different runtime back-ends (hStream for XeonPhi, and CUDA or OpenCL for GPUs).

Code Example. Fig. 1 gives a simplified code example written with Intel's hStreams APIs that can run on the XeonPhi many-core. At line 2 we initialize the stream execution by setting the number of partitions and tasks/streams per partition. This initialization process essentially creates multiple processor domains and determines how many logical streams can run on a partition. In the *for* loop (lines 7-14) we enqueue the communication and computation tasks to a number of streams identified by the `stream_id` variable. In this way, communication and computation of different streams can be



(a) `binomial` (b) `prefixsum`

Fig. 2. Heatmaps show the resultant speedup (over single-stream) of `binomial` and `prefixsum` under different stream configurations. The *#partitions* and *#tasks* have a significant impact on the resultant performance, and the sweet spots are sparse and vary across programs.

overlapped during execution (temporal sharing); and streams on different processor domains (or partitions) can run concurrently (spatial sharing). Our predictive model determines the *#partitions* and the *#tasks* before invoking the hStreams initialization routine, `hStreams_app_init()`.

2.3 Motivating Examples

Consider Fig. 2 which shows the resultant performance improvement given by multi-stream parallelism over the single-stream version of the code for two applications on a 57-core Intel XeonPhi system. We use two streamed programs from prior work [13]: `binomial` computes the price evolution over a given period and `prefixSum` calculates the prefix sum for a sequence of numbers.

It is observed from this example that not all multi-stream configurations give improved performance. As can be seen from the diagrams, the search space of multi-stream configurations is huge but good configurations are sparse. The performance varies significantly over stream configurations (*#partitions*, *#tasks*). The optimal *#tasks* for `binomial` ranges from 1 to 30, and the best *#partitions* is between 1 and 40. In contrast to `binomial`, `prefixsum` benefits from fine-grained parallelism when using a larger *#tasks* (220 to 224) and *#partitions* (60 to 80). However, the stream configurations that are effective for `prefixsum` give no speedup over the single-stream version for `binomial`.

Now consider Fig. 3 that shows the speedups of `dct` under 16 multi-stream configurations over the single-stream version, where each configuration is found to give the best-performance for one of the 16 inputs. In the color table, each cell shows the performance of a stream configuration ($C1, \ldots, C16$) on a specific input dataset ($D1, \ldots, D16$); and the values along the diagonal line represent the best-available performance (found through profiling) for an input. As can be seen from the figure, the best stream configuration can vary across inputs for the same benchmark. For example, while $C4$ gives a speedup of 1.33x over the baseline for dataset $D4$, it delivers a poor performance for dataset $D14$ by doubling the execution time over the single-stream version. This diagram also suggests that no single configuration can give improved performance for all inputs.

	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12	D13	D14	D15	D16
C1	**1.41**	1.33	1.10	1.17	1.16	1.15	1.07	1.07	1.01	0.86	0.98	0.96	1.04	1.05	0.86	0.99
C2	1.28	**1.46**	1.27	1.17	1.18	1.09	1.14	1.05	1.00	0.97	1.05	1.00	1.05	0.98	1.01	0.75
C3	1.28	1.18	**1.29**	1.22	1.14	1.00	1.05	1.00	1.04	0.95	1.03	0.96	0.99	1.01	0.99	1.03
C4	1.14	1.06	1.05	**1.33**	1.23	1.21	1.14	1.12	1.16	1.11	1.06	1.00	0.98	0.46	1.06	0.85
C5	1.16	1.15	1.20	1.12	**1.27**	1.18	1.15	1.13	1.08	1.08	1.00	0.98	0.96	0.99	1.06	1.03
C6	1.24	1.06	1.08	1.16	1.19	**1.24**	1.20	1.06	1.11	0.96	0.94	0.94	1.06	0.93	1.02	1.06
C7	1.16	1.10	1.13	1.21	1.20	1.10	**1.31**	1.09	1.13	0.97	0.99	0.95	0.99	0.56	0.95	1.03
C8	1.23	1.05	1.21	1.15	1.12	1.14	1.17	**1.14**	1.14	0.90	0.98	1.05	0.99	0.43	0.99	1.02
C9	1.19	1.07	1.10	1.17	1.22	1.19	1.20	1.11	**1.17**	0.99	1.04	0.96	1.00	0.63	0.97	1.06
C10	1.13	0.97	1.07	1.05	1.27	1.14	1.24	0.98	1.04	**1.15**	0.94	1.03	0.96	0.41	0.99	1.00
C11	1.15	1.13	1.08	1.19	1.10	1.08	1.15	0.97	0.97	1.01	**1.11**	0.80	1.02	0.96	1.02	1.00
C12	1.02	0.94	1.17	1.17	1.21	1.10	1.17	0.94	1.00	0.96	0.96	**1.06**	1.04	0.47	1.08	1.10
C13	0.94	0.96	0.96	1.22	0.96	1.05	1.00	0.86	0.95	0.95	1.00	1.04	**1.17**	0.49	0.98	0.99
C14	0.89	1.03	0.81	0.89	0.89	0.86	0.84	0.91	0.92	1.04	0.92	0.99	1.00	**1.07**	1.04	1.10
C15	1.01	1.10	1.08	1.11	1.05	1.01	0.98	0.88	0.88	0.90	0.95	0.95	1.04	0.95	**1.11**	0.96
C16	0.82	0.81	0.97	1.12	1.02	1.07	1.00	0.94	0.96	0.99	1.06	1.02	1.04	0.41	1.03	**1.14**

Fig. 3. Color table showing the speedups of best-performing configurations across inputs for `dct`. Each cell shows the performance for one of the 16 best-performing configurations, Cn, on a given input, Dn. The best configuration varies across inputs and a good configuration on one input can give poor performance on another dataset.

Lesson Learned. These two examples show that choosing the stream configuration has a great impact on performance and the best configuration must be determined on a per-program and per-dataset basis. Later, we will show that this observation is not unique to XeonPhi but also holds for GPUs. Attempting to find the optimal configuration through means of an exhaustive search would be ineffective, and the overhead involved would be far bigger than the potential benefits. Online search algorithms, while can speed up the search process, the overhead can still outweigh the benefit. For example, when applying simulated annealing to `binomial`, the best-found configuration only reaches 84 percent of the best-available performance after 310,728 iterations.[1] Classical hand-written heuristics are not ideal either, as they are not only complex to develop, but are likely to fail due to the variety of programs and the ever-changing hardware architecture. An alternate approach, and the one we chose to use, is to use machine learning to automatically construct a performance model to estimate the benefit of any candidate configuration, providing minimal runtime overhead for searching for a good configuration, and having little development cost when targeting new architectures.

2.4 Overview of Our Approach

Our library-based approach, depicted in Fig. 4, is completely automated. To determine the best streaming configuration, our approach follows a number of steps described as follows. We use a set of information or *features* to capture the characteristics of the program. We develop a LLVM [17] compiler pass to extract static code features at compile time, and a low-overhead profiling pass to collect runtime information at execution time (i.e., during the first few loop iterations). Because profiling also contributes to the final program output, no computation cycle

1. In Section 6.1, we show that our approach achieves 93 percent of the best-available performance for `binomial` on XeonPhi.

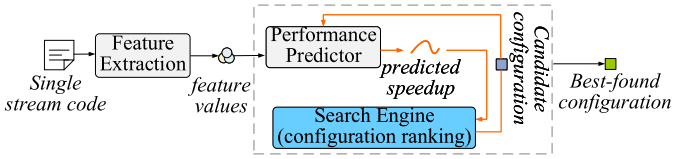


Fig. 4. Our machine learning based performance model (trained *offline*) predicts the speedup based on the extracted feature values of the code and a given stream configuration. We use the predictions to quickly rank candidate configurations at runtime to choose the one with the best predicted performance.

is wasted. At runtime, we search for a good configuration through an offline trained performance model to estimate the resulting performances for all candidate configurations. The performance model takes in the feature values, a given configuration of resource partition and task granularity and estimates the potential speedup for the given configuration over the single-stream version. The overhead of runtime feature collection and search is a few milliseconds, which is included in all our experimental results. Since our training process can be performed automatically, we can easily target our performance model for different architectures.

3 Performance Modeling

At the core of our approach is a machine learned performance model built upon the Multi-layer Perceptron (MLP) artificial neural network (ANN). Our prototype is implemented using the Python scikit-learn machine learning package [18]. It is to note that our prior work [16] uses a Support Vector Machine (SVM) based classifier. However, such an approach can only make predictions on a limited set of configurations seen at the training time. Unlike a classification-based approach, the new approach presented in this article is a *regression-based* model which can make predictions on any stream configuration. This new approach thus has a better generalization ability for various heterogeneous architectures. We have also evaluated a number of alternative modeling techniques, including MLP, SVM, and decision trees. We chose MLP because it gives the best performance and has modest training overhead (see Section 6.6.1).

Our performance model takes as input the feature values and a given configuration (e.g., #*partitions* and #*tasks* for XeonPhi and #*tasks* for GPUs). It predicts the speedup for the given configuration. Building and using such a model follows a 3-step process for supervised learning: (i) generate training data (ii) train a performance model (iii) use the performance model, described as follows.

3.1 Training the Performance Model

Our method for model training is shown in Fig. 5. To learn a regression model, we first need to profile the execution time (in order to calculate the speedup over the single-stream version) of all candidate configurations for each training program, and extract the feature values from the program. We then use the feature values, configuration settings and speedups to train a model.

3.1.1 Generating Training Data

To generate training data, we apply *cross-validation* to 39 benchmarks, i.e., by excluding the testing benchmarks from

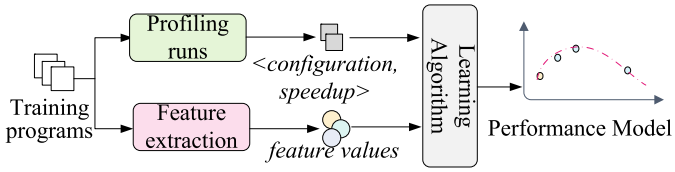


Fig. 5. The training process of our performance model.

the training dataset (see also Section 5.3.1). We execute each training program and benchmark a number of times until the gap of the upper and lower confidence bounds is smaller than 5 percent under a 95 percent confidence interval setting. We then calculate the average speedup for a given stream configuration over the single-stream version. We exhaustively execute each training program across a wide range of stream configurations, and record the performance of each. Next, we calculate the speedup for each configuration, program and dataset. Finally, we extract the values of our selected set of features from each program and dataset. We stress that the trained model can be applied to stream configurations that are not seen in the training phase.

3.1.2 Profiling Configurations

During the training phase, we exhaustively execute each training program across a set of streamed configurations. On XeonPhi, we profile each training program using the #*partitions* ranging from 1 to 224 (the maximum number of physical threads on XeonPhi) and the #*tasks* ranging from 1 to 256.[2] On GPUs, we cannot configure the number of partitions currently, we set the #*partitions* to the same as #*tasks* to be consistent with XenPhi. On this platform, we also set the #*tasks* to be range between 2^0 and 2^{10}, which is big enough to include the optimal values according to our experiments. Note that these parameter ranges can be configured by the user.

3.1.3 Building the Model

Each evaluated configuration is appended to the feature value vector of a training program to form a model input. The model inputs and the corresponding speedups (i.e., ground truths) for all training programs are passed to a learning algorithm. The algorithm finds a correlation between the input vector and the desired prediction. The output of our learning algorithm is an MLP model where the weights of the model are determined from the training data. Model parameter tuning is performed on the training dataset for each targeting hardware architecture, using cross-validation (see also Section 6.6.3). In our case, the overall training process for all the 39 training programs (which is dominated by training data generation) takes less than a week on a single machine. Since training is performed only once "at the factory", this is a *one-off* cost.

3.2 Features

Our performance models are based exclusively on code and dynamic features of the target programs. Code features are extracted from the program source code, and dynamic features are collected using hardware performance counters

2. We chose these values because configuration settings beyond these values give a poor performance during our initial evaluation.

TABLE 1
Chosen Features for XeonPhi Performance Model

Feature	Description
loop nest	at which level the outermost parallelizable loop lies on
loop count	# of the parallel loop iterations
#xfer_mem	# of host-device transfer API calls
dts	total host-device transfer size
redundant transfer size	host-device transfer size among overlapping tasks
max blocks	the maximum number of tasks of the application
min task unit	the minimum task granularity for a partition
# instructions	the total number of instructions of the kernel
branch miss	branch miss rate
L1 DCR	L1 Data cache miss rate

TABLE 2
Chosen Features for GPU Programs

Feature	Description
Access type 1	# array access, whose fastest varying index is an affine function of the block id
Access type 2	#array accesses, whose second or higher dimensional index is an affine function of the block id
#xfer_mem	# of host-device transfer API calls
host to device transfer size	total host to device transfer size
device to host transfer size	total device to host transfer size
redundant transfer size	host-device transfer size among overlapping tasks
max blocks	the maximum number of tasks
# instructions	the total number of instructions of the kernel
divergent branches	# divergent branches
L2 read miss rate	L2 cache read miss rate

during the initial profiling run of the target application. We restrict us in using hardware performance counters that are commonly available on modern processors such as the data cache misses to ensure that our approach can be applied to a wide range of many-core architectures.

We considered 38 candidate raw features in this work. Some features were chosen from our intuition based on factors that can affect the performance such as `dts` (host-device data transfer size) and `#xfer_mem`, while other features were chosen based on previous work [19], [20].

3.2.1 Feature Selection

To build an accurate model through supervised learning, the training sample size typically needs to be at least one order of magnitude greater than the number of features. In this work, we start from 311 training samples and 38 raw features, so we would like to reduce the number of features in use. Our process for feature selection is fully automatic, described as follows.

We first combine several raw features to form a set of combined normalized features, which are able to carry more information than the individual parts. For example, instead of reporting raw branch hit and miss counts, we use the branch miss rate. Next, we removed raw features that carried similar information which is already captured by chosen features. To find which features are closely correlated, we constructed a correlation coefficient matrix using the Pearson correlation coefficient [21]. The closer a coefficient between two features is to +/-1, the stronger the correlation between the two input features. We removed any feature which had a correlation coefficient (taking the absolute value) greater than 0.7. Similar features include the number of executed instructions and the number of E-stage cycles that were successfully completed.

Our feature selection process reduces the number of features to 10 for XeonPhi (see Table 1) and 10 for the NVIDIA Titan 1080Ti GPU (see Table 2), where some features are shared. Since our approach for feature selection is automatic, the approach can be applied to other sets of candidate features. It is to note that feature selection is also performed using cross-validation (see also Section 5.2).

3.2.2 Feature Standardization

Supervised learning typically requires the feature values to lie in a certain range. Therefore, we scaled the value for each of our features between the range of 0 and 1. We record the maximum and minimum value of each feature found at the training phase, and use these values to scale features extracted from a new application after deployment. We truncate a value during deployment if the value is outside the minimum/maximum value range seen during training. It is to note that we also use the same approach to normalize the model predictions (speedups) to the range of 0 and 1. In this work, we choose *Z-score* to standardize the training data, and the details of quantifying the impact of feature engineering methods can be found in Section 6.6.2.

3.2.3 Feature Importance

To understand the usefulness[3] of each feature, we apply a factor analysis technique called Varimax rotation [22] to the feature space transformed by the principal component analysis (`PCA`). This technique quantifies the contribution of each feature to the overall variance in each of the `PCA` dimensions. Intuitively, the more variances a feature brings to the space, the more useful information the feature carries.

As an example, Fig. 6 shows the top features chosen for XeonPhi and NVIDIA GPU architectures. For the XeonPhi platform, features that capture the parallelism degree (e.g., `max blocks`), host-device communication (e.g., `redundant transfer size`), and computation (e.g. `#instructions`) are found to be important. Other features such as `L1 DCR` and `loop nest` are useful, but are less important compared to others. On the NVIDIA GPU platform, we note that the parallelism degree is important, and the other features are equally useful (Fig. 6b). This figure shows that prediction can accurately draw upon a subset of aggregated feature values.

3.3 Runtime Deployment

Once we have built and trained our performance model as described above, we can use it as a cost function to search for the best stream configuration for any *new*, *unseen* program. Feature values are extracted from the single-stream version of the code. Static code features (such as `loop count`) are extracted from the program source at compile time. Dynamic

3. In Section 6.6.4, we give a further breakdown of the impact of individual feature to the model performance on a per benchmark basis.

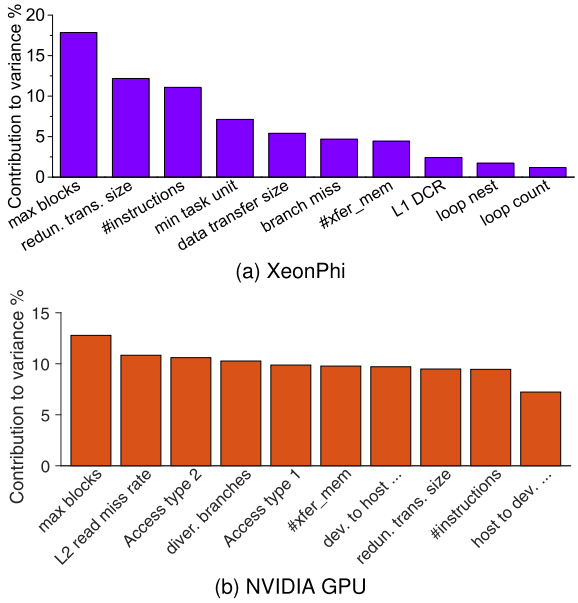


Fig. 6. Feature importance on (a) XeonPhi and (b) NVIDIA GPU.

features (such as `branch miss`) are extracted by profiling the program without partitioning for a few loop iterations (which typically translate to several microseconds). After feature collection, we feed the feature values to the search engine to rank all candidate configurations using the performance model. The top-ranked stream configuration is then used for the target program. In Section 4.4, we provide further details on how the performance model can be integrated with the host code generation process.

3.3.1 Adapt to Changing Program Phases

Our current implementation chooses a configuration for each kernel and does not change the configuration throughout the kernel execution. Therefore, it can adapt to different behaviors across kernels because predictions are performed on a per-kernel basis. We found that this strategy is sufficient for many data-parallel kernels targeted in this work.

Our approach can be extended to adapt phase or program behavior changes within a kernel. One way of doing this is to first partition the input data into groups and then perform configuration selection before launching the kernel that performs on an input data group. To reduce the prediction and configuration overhead, we can sample periodically to see if the performance counter readings are significantly different from the ones used for the current prediction to trigger reconfiguration. Dynamic re-configuration of a running kernel will require extending the underlying runtime (e.g., hStreams or CUDA) to adjust thread mapping and having hardware support to stop and resume the execution contexts. We leave this as future work.

4 OpenMP to Streamed Code Generator

Currently, there are very few publicly available benchmarks for utilizing the streaming capability of heterogeneous many-core architectures, in particular, XeonPhi. To evaluate our approach on a diverse set of benchmarks, we have developed a compiler-based code generator, autostreamer, to automatically translate OpenMP programs onto streamed

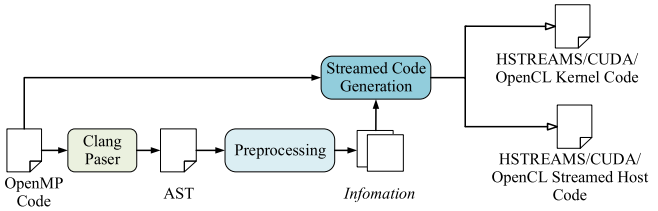


Fig. 7. Work flow for translating OpenMP programs to streamed programs using our automatic code generator.

code depending on the target architecture. Our code generator is open sourced.[4] Our implementation currently supports converting OpenMP code to hStreams, CUDA and OpenCL programs. While we do not claim novelty on this as several works on source-to-source translation from OpenMP to CUDA [23], [24], [25], [26] or OpenCL [20], [27] exist, we believe the tool could serve as a useful utility for translating OpenMP programs to exploit multi-stream performance on heterogeneous many-core architectures.

4.1 Code Generator Overview

Fig. 7 depicts our source to source code generator for translating OpenMP code to streamed programs. We use LLVM's Clang front-end to convert OpenMP code into the abstract syntax tree (AST). We then traverse the AST to obtain the information to generate candidate streamed kernels and host-device management code. The generated kernel and host code make use of exiting programming models for kernel launching and communication management. We use hStreams for XeonPhi and CUDA or OpenCL for GPUs.

Our current implementation supports the translation of OpenMP parallel loops, i.e., loops annotated with `omp for` or `omp for reduction` constructs. For each parallel loop, we outline the loop body and translate it into an individual kernel function. We then replace the original loop body with a function call (running on the host CPU) to launch the generated kernel. We also generate management code for streaming context initialization, data partitioning, data movements between the host and the accelerator, etc.

Our code generator relies on the native host/device compiler to optimize the generated code. We have also compared our automatically generated code against the manually translated code used in our prior work [16] and found that there is little difference in performance for the set of OpenMP benchmarks used in this work.

4.2 Preprocessing

As an example, Fig. 8 illustrates how an OpenMP parallel loop can be translated into hStreams code for XeonPhi. Note that a similar code generation process is implemented for GPUs, using CUDA for NVIDIA GPU architectures and OpenCL for other GPU platforms.

For each OpenMP parallel loop, we extract information of loop iterations from the loop head. In this work, partitioning is achieved by splitting the loop iteration space. Furthermore, we collect all the variables needed by the hStreams kernel. Because hStreams requires kernel parameters to be passed as the `uint64_t` (lines 1-2 of Fig. 8b), the kernel

4. Available at: https://github.com/wisdom-moon/autostreamer.

```
1  // An OpenMP C code for vector addition
   float * hostOutput = (float *) malloc(inputLength*sizeof
       ↪ (float));
3  ...
   #pragma omp parallel for
5  for(int i=0; i<inputLength; i++)
   {
7    hostOutput[i] = hostInput1[i] + hostInput2[i];
   }
9  ...
```

(a) OpenMP code.

```
1  COINATIVELIBEXPORT void kernel (uint64_t arg0,
         uint64_t arg1, ... uint64_t arg5)
3  {
     int _start = (int) arg0;
5    ...
     float *hostInput2 = (float *) arg5;
7
     #pragma omp parallel for
9    for(int i= _start; i< _end; i++)
       hostOutput[i] = hostInput1[i] + hostInput2[i];
11 }
```

(b) hSTREAMS kernel code.

```
1  //output buffer
   float * hostOutput = (float *) malloc(inputLength*sizeof
       ↪ (float));
3
   //Feature update and prediction
5  Stream config;

7  conf_search(&config, &kernel_1_features,
       ↪ kernel_1_profile_runs);
   int partitions = config.partitions;
9  int tasks = config.tasks;

11 //hStreams Initialization
   hStreams_app_init(partitions, 1); .
13 ...
   hStreams_app_create_buf((float *)hostInput1, ...);
15 ...

17 //Work partition
   int sub_blocks = inputLength / tasks;
19 int remain_index = inputLength % tasks;

21 //Initialize kernel arguments
   uint64_t args[6]; args[2] = (uint64_t) inputLength;
23 ...
   for (int idx = 0; idx < tasks; idx++) {
25   args[0] = (uint64_t) _start;
     _end = _start + sub_blocks;
27
     if (idx < remain_index)
29     _end ++;

31   args[1] = (uint64_t) _end;
     hStreams_app_xfer_memory(&hostInput1[_start], &
         ↪ hostInput1[_start], (_end-_start)*sizeof(
         ↪ float), idx % partitions, HSTR_SRC_TO_SINK,
         ↪ NULL);
33   hStreams_app_xfer_memory(&hostInput2[_start], ...);

35   //Kernel launch
     hStreams_EnqueueCompute(idx % partitions, "kernel_1",
         ↪ 3, 3, args, ...);
37
    //Read back results
39   hStreams_app_xfer_memory(&hostOutput[_start], ...);
     _start = _end;
41 }
   ...
43 //hStreams cleanup code
   hStreams_app_fini();
```

(c) hSTREAMS host code.

Fig. 8. A running example of translating (a) an OpenMP parallel loop to (b) hSTREAMS kernel and (c) host management code.

parameters will be cast into this type. The kernel parameters need to be packed into an array (line 21 in Fig. 8c). Then the hSTREAMS library will unpack kernel parameters from the array and pass the parameters to kernel function.

During the preprocessing stage, we also extract the static code feature values of each target parallel loop. The code feature values will be encoded into the source code during host code generation. It is to note that our approach can be easily applied to existing hSTREAMS programs—by first gathering feature values from an hSTREAMS kernel, and then storing the extracted information in an auxiliary file or source code through a compiler front-end pass.

4.3 Kernel Code Generation

Generating a streamed kernel function is straightforward as much of the OpenMP code can be re-used. Fig. 8b gives an example of the automatically generated kernel for the OpenMP loop given in Fig. 8a for hSTREAMS kernels.

For the example given in Fig. 8, an hSTREAMS kernel starts with a pre-processor macro `COINATIVELIBEXPORT` (lines 1-2 in Fig. 8b). The number and the type of the kernel parameters are loop-specific and are automatically determined by our code generator. Within the generated kernel, all the function parameters are cast from `uint64_t` into an appropriate type before they are used. Note that the OpenMP `parallel for` pragmas are kept in the generated kernel code per hSTREAMS requirement (line 8 in Fig. 8b).

With our code generator, the original outer-most loop iteration space will be partitioned among parallel streams. The amount of work given to a specific stream is determined by the `_start` and `_end` variables, which define which part of the loop iteration space a stream instance will work on. A similar kernel code generation approach is implemented for GPUs using CUDA or OpenCL.

4.4 Host Code Generation

To generate host code, we replace the original OpenMP parallel loop with a function call to invoke the generated kernel (e.g., `hStreams_EnqueueCompute` in Fig. 8c)) together with additional code to initialize the host context and to manage data transfer.

4.4.1 Feature Value Collection

Static code features, extracted by our code generator, will be encoded as a feature vector of real values. The feature vector will be passed to our configuration search engine to find the optimal stream configuration at runtime. Dynamic feature values are automatically collected by running the generated streamed kernel for 5 iterations under the single-stream configuration. As some loop bounds are dependent on the input, we might be unable to determine certain feature values at compile time. These features are represented as static symbolic pre-computation of loop bound variables, which will be updated using runtime values at runtime.

4.4.2 Setting Stream Configurations

To partition tasks among streams, we break the loop iterations into a number of chunks of an equal size of subtask. We then group the hardware processor cores into partitions, where each partition contains a fixed set of streams. Processor partitioning and streams creation are achieved by calling the `hStreams_app_init` (line 12 in Fig. 8c) function for XeonPhi (and `cudaStreamCreate` and `clCreateCommandQueue` for CUDA and OpenCL programs respectively)

by passing the stream configuration given by our search engine. To overlap host device communications, we further split the input/output data arrays to multiple data blocks (lines 32-39 in Fig. 8c) where each task operates on one block at a time while another data block is transferring between the host and the accelerator. The number of data blocks is determined by the stream configuration chosen at program runtime. The amount of work per task and the size of transferred data can be determined with kernel parameters. For example, in *for-loop* at line 24 of Fig. 8c, we calculate them with the starting position (`_start`) and the block size (`sub_block`). Thereafter, we schedule tasks and transfer the corresponding data blocks onto streams in a round-robin fashion.

4.4.3 Runtime Prediction

When a streamed (e.g., hStreams or CUDA) kernel is invoked, the configuration selection engine library will choose a stream configuration (line 7 in Fig. 8c) for the kernel. It uses the performance model to rank the candidate stream configurations and returns the optimal configuration (#*partitions* and #*tasks* for the example shown in Fig. 8). The returned values are then used to initialize the streamed context (lines 8-9 of Fig. 8c). The overhead of prediction is negligible (a few milliseconds) and is included in the results.

4.4.4 Supporting OpenMP Constructs

OpenMP variables may have additional type information specified by directives, including `default`, `share`, `private`, `firstprivate`, `lastprivate`, `copyin` and `threadprivate`. Our generator uses these directives to map data onto the accelerator memory space. Each variable with the `share` or `default` directive will be translated into a global variable shared by all parallel threads. Variables declared as `private` and `threadprivate` are translated such that there is a private copy for each streamed kernel; no memory transfer between the host and the accelerator is needed. For each variable specified as `copyin` or `first private`, we create a private copy for each streamed kernel but initialize each copy using explicit memory transfers before its first use. Similarly, we create a private copy of a `last private` variable and the original variable is updated by a stream that executes the last iteration.

Our implementation also supports a number of synchronization and thread constructs. Structured blocks identified with `master`, and `single` directives are executed by one thread on the host multi-core. `barrier` is implemented by splitting up the parallel loop into smaller tasks to create synchronization points among multiple streams. `critical` is implemented by using a mutex lock to restrict the execution of the associated structured blocks to a single thread at a time. The `atomic` and `flush` directives are already supported by hStreams, CUDA or OpenCL.

4.4.5 Host-Accelerator Communication Optimization

For each buffer that is used by both the host and the accelerator, we manage two copies: one on the host memory and the other on the accelerator memory. Our runtime records the status of each variable and checks whether the copy on a device memory space is valid or not. No memory transfer is needed as long as the copy in the target memory space is valid. We currently use a conservative approach: if an element of an buffer has been updated, the entire buffer needs to be synchronized before it can be used by threads running on a different device. We also avoid unnecessary device to host data transfer by tracking the data dependence between the kernel and the host program. For example, when there are data-dependencies between two kernels but the host does not access this data in between the two kernels, we directly pass the memory address of the buffer to the later kernel (without moving the data back to the host).

TABLE 3
Our Evaluation Platforms

	CPU-XeonPhi	CPU-GPU
CPU	8-core Xeon CPU @ 2.6 GHz	Core i7-8700K CPU @ 3.7 GHz
Accelerator	Intel Xeon 31SP Phi	NVIDIA GeForce GTX 1080 Ti GPU

5 Experimental Setup

5.1 Hardware, Systems Software, and Benchmarks

Platforms. We evaluate our approach on two heterogeneous many-core platforms: one is a CPU-XeonPhi platform and the other is a CPU-GPU platform. Table 3 gives details of our hardware platforms.

Systems Software. On the CPU-XeonPhi platform, the host CPU and the accelerator are connected through `PCIe`. The host runs Redhat Linux v7.0 (with kernel v3.10). The coprocessor runs a customized uOS (v2.6.38.8). We use Intel's MPSS (v3.6) to communicate between the host and the coprocessor. We use the Intel hStreams library (v3.6) and Intel ICC (v16.0.3) for compilation (with −O3 as the compiler option). The CPU-GPU platform runs Ubuntu 16.04 (with kernel v4.15). We use CUDA v10.0 and gcc v5.3 as the host compiler with option "-O3".

Benchmarks. We use our code generator to translate 37 OpenMP applications from commonly used benchmark suites into hStreams and CUDA programs. We have excluded benchmarks where the data transfer cannot be overlapped with the kernel execution, which do not benefit from streamed parallelization. Table 4 gives the full list of these benchmarks. Among them, `convolutionFFT2d` and `convolutionSeparable` have algorithm-dependent parameters, which are regarded as different benchmarks in the experiments. This setting gives us a total of 39 programs. We run the majority of the programs using over 25 different datasets, except for some applications where we used around 10 datasets because the algorithmic constraints prevent us from using a larger number of inputs.

5.2 Competitive Approaches

We compare our regression-based approach against our preliminary work that employs an `SVM`-based classifier to predict the optimal stream configuration [16]. We denote our prior approach as `SVM-classifier`. We also compare our approach against two recent models for predicting the optimal stream configuration on GPUs. As it is currently not possible to configure the number of processor partitions on GPUs, the relevant GPU models can only predict the number of tasks.

TABLE 4
Streamed Benchmarks Used in Our Experiments

Suite	Name	Acronym	Name	Acronym
NVIDIA SDK	convol.Separable convolutionFFT2d MonteCarlo scalarProd vectorAdd	convsepr1(8) fftx1y1(4y3) montecarlo scalarprod vecadd	dotProduct fwt matVecMul transpose	dotprod fwt mvmult transpose
AMD SDK	binomial dct	binomial dct	BlackScholes prefixSum	blackscholes prefix
Parboil	bfs lbm mri-gridding sgemm	bfs lbm mri-gridding sgemm	histo mri-q sad spmv	histo mri-q sad spmv
POLY BENCH	2mm adi covariance gemm gesummv jacobi-1d mvt syrk	2mm adi covariance gemm gesummv jacobi-1d mvt syrk	3mm correlation deriche gemver heat-3d jacobi-2d syr2k	3mm correlation deriche gemver heat-3d jacobi-2d syr2k

Liu et al. In [12], Liu *et al.* use linear regression models to search for the optimal number of tasks for GPU programs [12]. The approach employs several analytic models, described as follows.

For a task with an input data size of m, the transferring time between the CPU and the accelerator, T_t, is determined as $T_t = \alpha \cdot m + \beta$, and the computation time, T_c, is calculated as: $T_c = \eta \cdot m + \gamma$ where the model coefficients, α, β, η and γ, are determined through empirical experiments. For a given kernel with N input data elements running using n streams, this approach partitions the computation into n tasks, where the data size for each task, m, is equal to N/n. For the programs which kernel dominated, the total execution time, T_{total}, can be determined by

$$T_{total} = T_t + nT_c = \alpha \cdot m + \frac{N\gamma}{m} + N\eta + \beta.$$

For the programs which data transfer dominated

$$T_{total} = \alpha \cdot N + 2\frac{N}{m}\beta.$$

By calculating the partial differential and second-order partial differential of T_{total} with respect to m, we can obtain the optimal task-granularity as $m = \sqrt{\frac{N\gamma}{\alpha}}$. Then we can calculate the number of tasks (n).

Note that $m = N/2$ is the optimal parameter for programs which data transfer dominated, i.e., the optimal number of tasks is 2. Another problem of this model is that it does not consider scenarios where communications in different direction (i.e., host to device and device to host) can overlap with each other. Note that we set the #*partitions* to be the same as n for XeonPhi.

Werkhoven et al. The work presented by Werkhoven *et al.* models the performance of data transfers between the CPU and the GPU [10]. They use the LogGP model to estimate the host-device data transfer time. Specifically, the model estimates the data transfer time using five parameters: the communication latency (L), overhead (o), the gap (g), the number of processors (P), and the `PCIe` bandwidth (G).

Let B_{hd} denotes the amount of data transferred from the host to the device and B_{dh} denotes vice versa, and T_{kernel} donates the kernel execution time. For the dominant transfer scenario, the optimal number of tasks (i.e., #*tasks*), N_s, can be estimated by solving the following equations:

$$B_{dh} * G_{dh} + g * (N_s - 1) = \begin{cases} \frac{T_{kernel}}{N_s} + \frac{B_{dh}}{N_s} * G_{dh}, & \text{if} B_{dh} > B_{hd} \\ \frac{B_{hd}}{N_s} * G_{hd} + \frac{T_{kernel}}{N_s}, & \text{otherwise} \end{cases}.$$

This model does not consider the dominant kernel scenario, as it assumes the kernel execution time will increase as the number of streams increases and can not model the kernel execution time. Here, we use the same equation to calculate the optimal number of tasks. For this model, we also set the #*partitions* to be equal to the optimal N_s value on XeonPhi.

5.3 Evaluation Methodology

5.3.1 Model Evaluation

We use cross-validation to evaluate our machine learning models. To test the portability of our approach, we apply *leave-one-out* cross-validation, described as follows. We exclude the target program for predictions from the training program set, and learn a model using the *remaining* programs. We then apply the learned model to the testing program. We repeat this process until each benchmark is tested once. This is a standard evaluation methodology, providing an estimate of the generalization ability of a machine learned model in predicting *unseen*data. Note that we exclude both `convolutionFFT2d` and `convolutionSeparable` from the training set when one of the two is evaluated, and we make sure all approaches are trained on the same benchmarks for fair comparisons.

5.3.2 Performance Report

We run each program under a stream configuration multiple times and report the *geometric mean* of the runtime. Compared to the arithmetic mean, the geometric mean is often considered as a more suitable metric for reporting program performance, as it can better minimize the impact of outliers [28]. To determine how many runs are needed, we calculated the confidence range using a 95 percent confidence interval and make sure that the difference between the upper and lower confidence bounds is smaller than 5 percent.

6 Experimental Results

In this section, we first present the overall performance of our approach on both platforms. We then compare our approach to that uses fixed stream configurations, two prior analytical models and our previous work. We futher discuss the benefit sources of the streaming parallelism and the working mechanism of our approach. At last, we demonstrate the tunning process of our model.

6.1 Overall Performance

In this experiment, we exhaustively profiled each application with all possible stream configurations and report the best-found performance as the *Oracle* performance. The Oracle gives an indication of how close our approach is to a *theoretically perfect* solution. The baseline used to calculate

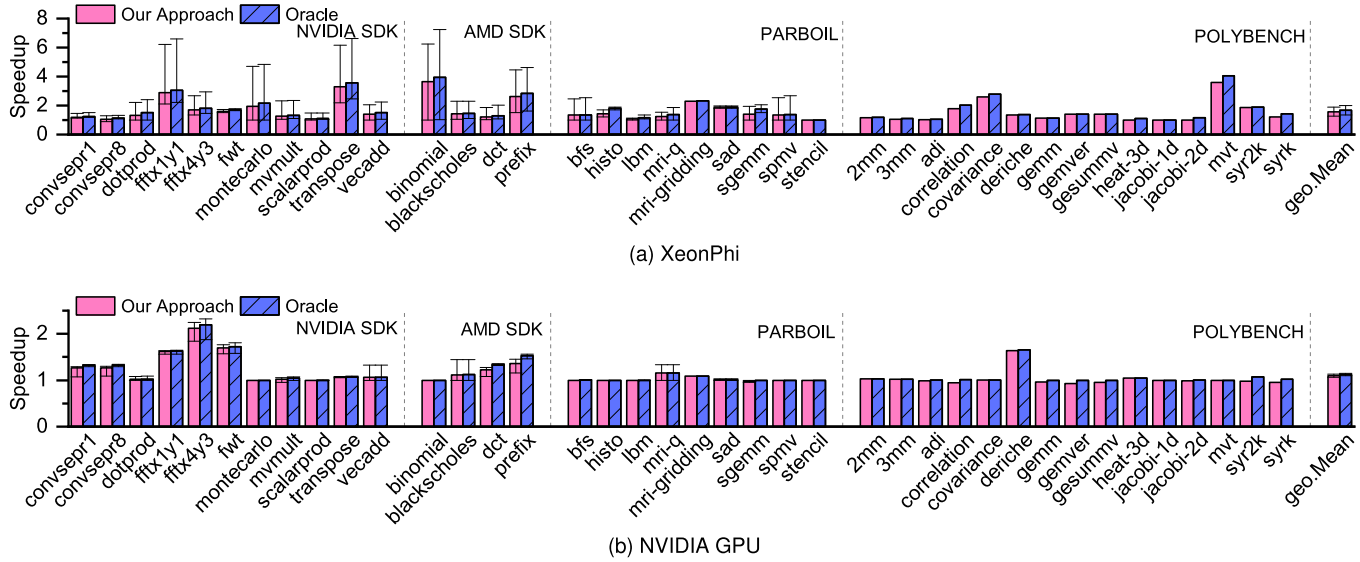


Fig. 9. Overall performance of our approach over a single-stream version on XeonPhi (a) and NVIDIA GPU (b). Our approach achieves, on average, 93.7 and 97.9 percent of the oracle performance on XeonPhi and NVIDIA GPU, respectively. The min-max bars show the range of performance achieved across different inputs.

the speedup is running the application using a single-stream without processor core or task partitioning.

The overall result is shown in Fig. 9. The min-max bar on the diagram shows the range of speedups per application across all evaluated inputs. Overall, our approach achieves an average speedup of 1.57× and 1.1× over the single-stream configuration on XeonPhi and the GPU respectively. This translates to 93.7 and 97.9 percent of the Oracle performance on XeonPhi and the GPU respectively.

On XeonPhi, the performance improvement of our approach comes from two factors. First, by predicting the right processor partition size, our approach allows effective overlapping of the host-device communication and computation. Second, by matching task parallelism to the number of available processor cores, our approach can reduce the overhead of thread management, compared to the single-stream execution. When the host-device communication time dominates the streaming process, the performance improvement mainly comes from computation-communication overlapping and the speedup from streaming is consistently less than 2×. When the kernel execution time dominates the stream process, the application can benefit from the overhead reduction of thread management. In this case, the speedup can be as large as 5×. We provide a further discussion on this later in Section 6.5.1.

On the GPU, we can exploit bidirectional data transfer between the host and the device by using pined memory which is not supported by hStreams. The support of bidirectional data transfer allows us to obtain further performance gains by overlapping host-device data transfer and computation. The theoretically up-bound speedup on the GPU platform is 3×, when data transfer is perfectly overlapped with computation. The representative sample is `fftx4y3` with the larges dataset, the data transfer time in the two directions is the same, and the kernel execution time is 1.5 times of the data transfer time. The oracle speedup is 2.3×, and our approach achieves a speedup of 2.2 ×. On the other hand, because the current GPU implementation does not support processor core partition, the kernel execution time benefits less from using multiple streams. Programs which the kernel execution time dominated have no speedup using multiple streams, such as `bfs`, `MonteCarlo`.

6.2 Comparison to Fixed Stream Configurations

Our approach predicts from a wide range of stream configurations, which configuration is likely to give the best performance for a given program and dataset. A natural question to ask is that: *is there a fixed stream configuration that gives reasonable good performance across benchmarks and datasets?* To answer this question, we compare our predictive modeling based approach to two specific configurations on each of our evaluation platforms. Our justification for why we selecting the fixed configurations are described as follows. On XeonPhi, our initial results in Section 2 indicate that using the stream configuration of (4, 16), i.e., partitioning the cores to 4 groups and running 4 tasks on each partition (16 tasks in total), gives good performance. The statistics obtained from the training data suggest that the configuration of (17, 85) give the best average performance across training samples. On the GPU, several programs support a maximum of 4 tasks. Thus we select the two configurations (2, 2) and (4, 4). The results are shown in Fig. 10.

6.2.1 XeonPhi

On XeonPhi, we observe improved performance for several benchmarks such as `mri-gridding`, `transpose`, `sad`, under both configurations, but slower performance for `dotprod`, `vecadd`, `blackscholes`, `lbm`, and `mir-q` (Fig. 10a). For `prefix`, configuration (17, 85) delivers improved performance while configuration (4, 16) leads to slowdown performance. Overall, none of the two fixed configurations give an improved performance on average. On average, our approach outperforms the two fixed configurations by a factor of 1.4, and delivers consistently improved performance across benchmarks and datasets.

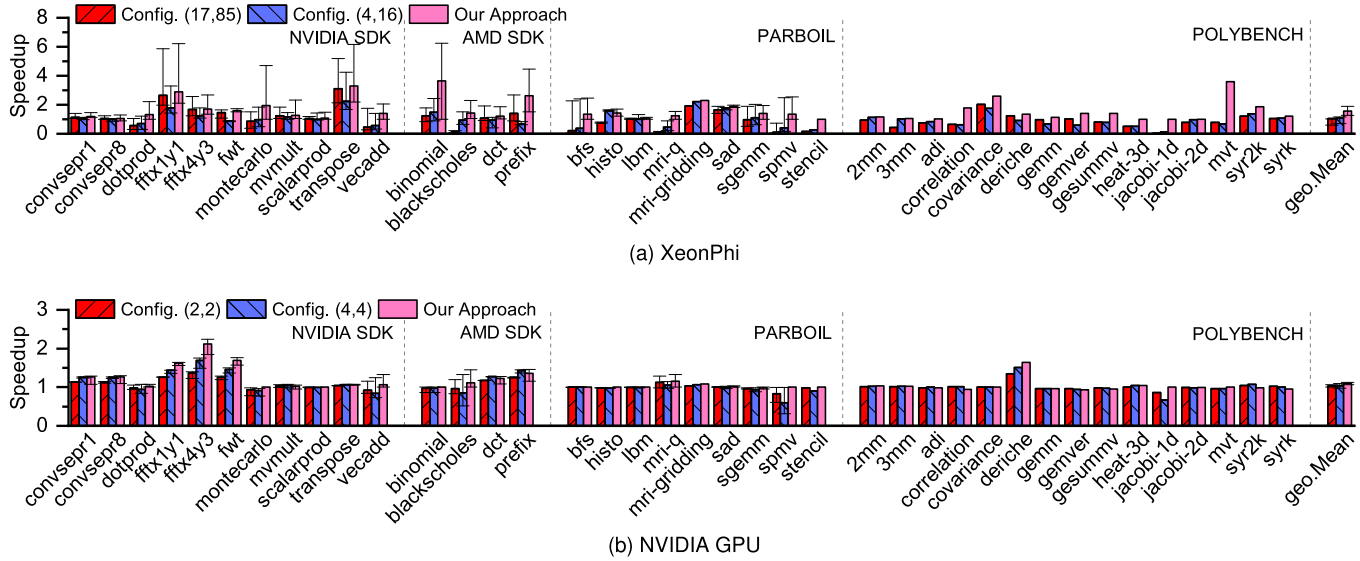


Fig. 10. Comparing the performance with two fixed configurations on XeonPhi (a) and NVIDIA GPU (b): config. (4,16) of 4 partitions and 4 tasks per partition, config. (17, 85) of 17 partitions and 5 tasks per partition, config. (2, 2) of 2 partitions and 1 tasks per partition, and config. (4, 4) of 4 partitions and 1 tasks per partition.

The violin plot in Fig. 11a shows how far is each of the three schemes to the Oracle performance across benchmarks and datasets. Our approach not only delivers the closest performance to the Oracle, but also has the largest number of samples whose performance is next to the Oracle. By contrast, the performance given by the fixed configurations for many samples is farther from the Oracle performance.

6.2.2 GPU

On the GPU, in most cases, the performance of configuration (2, 2) is moderate, not great, but not much worse than single-version, leading to an average speedup 1.03× (Fig. 10b). By contrast, although configuration (4, 4) performs poorly on two programs, it delivers a slightly larger averaged speedup of 1.04×. By choosing the stream configuration on a per-program basis, our approach outperforms the two fixed configurations, achieving an averaged speedup 1.10×. On only four programs, our approach delivers slightly worse performance with a small margin.

The violin plot in Fig. 11b also confirms the strengths of our approach by presenting the distribution of performance improvement. The results on the diagram are normalized to the Oracle (best-available) performance. For most of the programs, the two fixed configurations deliver 80 to 100 percent to the Oracle performance. However, configuration (4, 4) can lead to rather poor performance (less than 40 percent to the best available performance) on some programs. Compared to the fixed configurations, the performance distribution given by our approach is centralized on a range between 90 to 100 percent, where most programs are within this percentile range. Furthermore, compared to the fixed configurations, our approach has a fewer number of performance outliers, which have less serious performance slowdown. Therefore, our approach delivers consistently better performance compared with the fixed configurations.

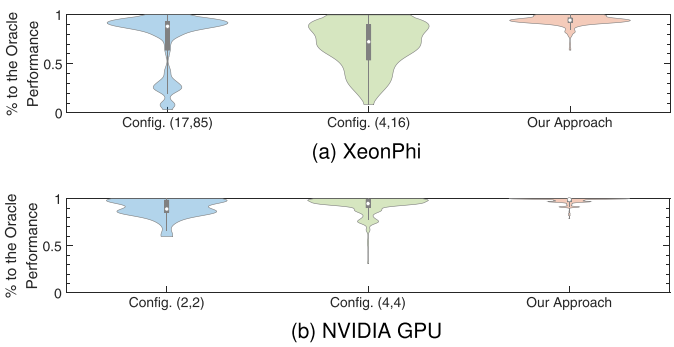


Fig. 11. Violin plot showing the distribution of speedups per scheme across benchmarks and datasets on XeonPhi (a) and GPU (b). The shape of the violin corresponds to the speedup distribution to the oracle performance. The thick black line shows where 50 percent of the data lies.

6.2.3 Summary

This experiment confirms that a fixed configuration fails to deliver improved performance across applications and datasets, and selecting a right stream configuration on a per program, per dataset basis is thus required.

6.3 Comparison to Analytical Models

In this experiment, we compare our approach to the two recent analytical models described in Section 5.2. The results are shown in Figs. 12 and 13. On XeonPhi, both competitive models prefer using 2 tasks across benchmarks and datasets. This is because that many programs are kernel dominated, the analytical models simply assume that task partition has no effect on kernel's performance, and do not consider the thread management overhead. On the GPU, the model proposed by *Liu et al.* tends to use 2 tasks across benchmarks and datasets. This is due to the fact that most programs are data transfer dominated and this model ignores the overlap of the bidirectional data transfers between the host and the device.

XeonPhi. Fig. 12a demonstrates that our approach gives better performance for nearly all programs on XeonPhi. For the remaining handful programs, all three approaches deliver comparable performance. Compared to the results Fig. 10, we

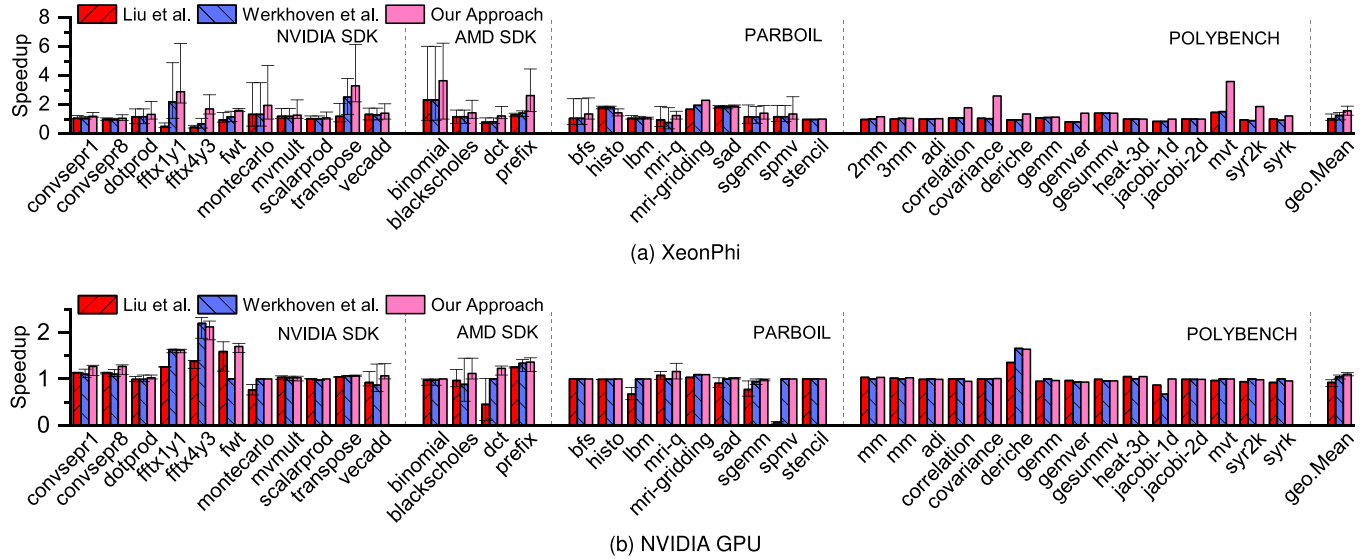


Fig. 12. Comparing against *Liu et al.* and *Werkhoven et al.* on XeonPhi (a) and NVIDIA GPU (b).

can find the performance of the analytical models is similar to fixed stream configurations. This is because the performance of the seven programs, such as `binomial`, changes dramatically with different stream configurations (see also Fig. 2). The performance of the remaining programs is not sensitive to the variation of stream configurations. From Fig. 13a, we can further see that *Liu et al.* and *Werkhoven et al.* deliver a speedup within a range on 20 to 80 percent, while the performance of our approach is centralized on a range between 80 to 100 percent. Thus, our approach delivers consistently better performance compared with the alternative models.

GPU. Fig. 12b shows that our approach delivers better performance for around 75 percent of the programs on the GPU. Since *Werkhoven et al.* and *Liu et al.* are manually tuned for the GPUs, they give better performance on some benchmarks over our approach. However, our approach has the advantages of being automatically learned from training data, with little expert involvement. The performance of our approach can be further improved by using more training examples to better cover the program space. Fig. 13b shows that *Liu et al.* and *Werkhoven et al.* delivers a speedup within a range of 5 to 80 percent, and 70 to 100 percent, respectively. By contrast, the performance of our approach is centralized within a range between 90 to 100 percent for more programs. Therefore, overall, our approach delivers better average performance compared with the alternative models.

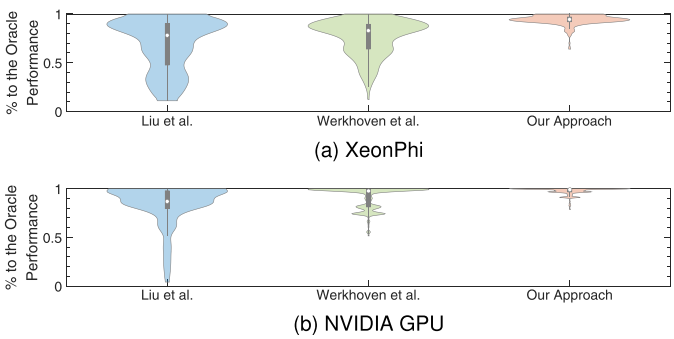


Fig. 13. Violin plots showing the distribution of speedups across benchmarks and datasets on XeonPhi (a) and GPU (b).

6.4 Comparison to Classification-Based Approach

Our prior work uses a `SVM` classifier to predict the configurations [16]. Compared with it, the regression-based model presented in this article has several advantages.

A classification model predicts which of a set of predefined labels the input belongs to. Using this strategy, we will need to label each unique stream configuration. This will lead to a total of 175 labels for 311 profiling samples on the XeonPhi, and 11 labels on the GPU. On the XeonPhi, the ratio of samples to labels is too small to build an accurate model. As a result, we have to merge labels in our prior work [16] at the cost of losing accuracy. Classification is a constraint optimization problem where the model has to know all the possible configurations during training. Our new regression-based approach avoids this pitfall by directly modeling the impact of the stream configuration; it thereby can be used on any stream configuration as the configuration is the model's input.

Fig. 14a presents results obtained on the XeonPhi. Our regression-based approach outperforms the `SVM-classifier` in 21 of the 39 programs and achieves over 5 percent performance improvement for 13 programs. It is to note that the overhead for ranking stream configurations is included in the experimental results. Overall, our regression-based approach improves the `SVM-classifier` by, on average, 3 percent (up to 46 percent). Unlike XeonPhi, we were able to obtain sufficient training samples per label (because the optimization space is smaller) on the GPU to build a more accurate classification model. As can be seen from Fig. 14b, the average speedup of `SVM-classifier` and the regression-based approach is comparable.

Compared to a classifier, our regression-based approach has the advantage of being able to be applied to configurations that were not seen during the training phase. Therefore, our approach has a better generalization ability.

6.5 Further Analysis of Performance Results

We now take a closer look into the performance results, using XeonPhi as a case study.

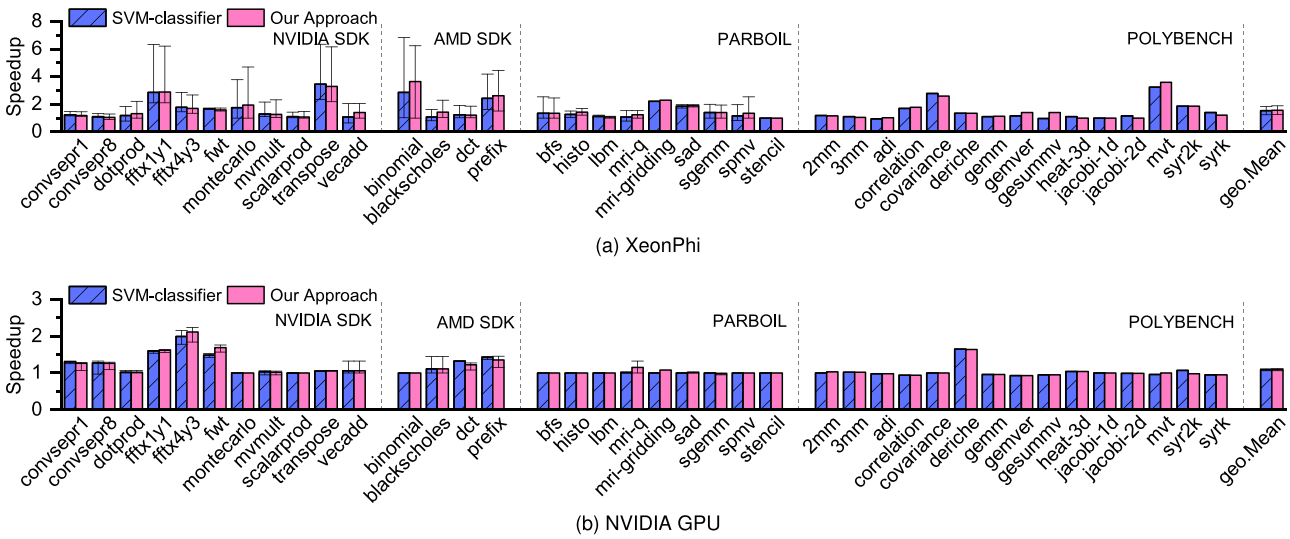


Fig. 14. Comparing against a classification based approach on XeonPhi (a) and NVIDIA GPU (b).

6.5.1 High Speedup Cases

On XeonPhi, bidirectional data transfer between the host and the accelerator cannot be overlapped, i.e., we can only issue data transfer from the host to the device or vice versa at once but not simultaneously. As a result, the theoretical up-bound speedup for overlapping computation and communication is 2×, when the computation is perfectly overlapped with the data transfer time. It is interesting to observe that several benchmarks achieve a speedup of over 2× on XeonPhi (see Fig. 9a). After having a closer investigation, we notice that such performance is attributed to the reduction in the kernel execution time in additional to the overlapping of communication and computation.

To quantify the benefit of kernel time reduction, we measure the kernel execution time with and without multiple streams and calculate the speedup between them. Note that we *exclude the host-device communication time in this case* to isolate the contributing factors. The kernel time improvement for `transpose`, `binomial`, and `fftx1y1` is shown in Fig. 15. As can be seen from the diagram, choosing a good stream configuration can lead to more than 4x reduction on the kernel execution time. This is because these benchmarks are implemented by parallelizing the inner loop within a nested loop. During runtime, the parallel threads working on the inner loop will be created, synchronized, or destroyed for each outer loop iteration. Such threading overhead could be significant when the outer loop iterates a large number of times. With multiple streams, we divide the whole outer loop iteration space into multiple smaller iterations. This allows multiple groups of threads to be managed simultaneously, leading to a significant decrease in threading overhead and faster kernel execution time. On the other hand, using too many streams and partitions will lead to a performance decrease. This is because stream management also comes at a cost, which increases as the number of partitions grows. Nonetheless, for applications where the kernel computation dominates the program execution time, by reducing the kernel time can lead to additional improvement, yielding more than 2x speedups.

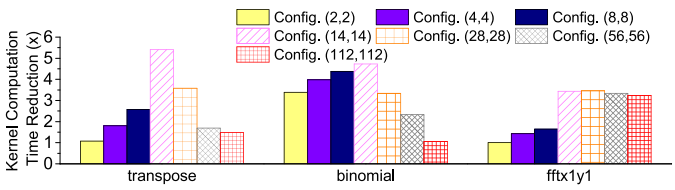


Fig. 15. Reduction of kernel computation time over a single-stream execution on XeonPhi. The performance improvement comes from the reduction of the threading overhead. A stream configuration is annotated as (*#partitions*, *#tasks*).

6.5.2 Speedup Distribution

Fig. 16 gives the speedup per benchmark across datasets on XeonPhi and the GPU. The shape of the violin plot corresponds to the speedup distribution.

On XeonPhi, we see that the speedups of `montecarlo` and `prefix` distribute fairly uniformly while the data distribution of `fftx1y1` and `fftx4y3` is multimodal (i.e., it has two peaks). Further, the input datasets have little impact on the behavior of `fwt` and `lbm`, so the speedups remain constant across datasets.

On the GPU, the speedups of `dotprod`, `vecadd`, `blackscholes` and `mri-q` distribute fairly uniformly while the data distribution of `convsepr1`, `convsepr8`, `fftx1y1`, `fftx4y3` and `dct` is unimodal (i.e., it has one peak). Furthermore, the input datasets have a very slight impact on the performance behaviors of `montecarlo`, `scalarprod`, `transpose` and `binomial`. Thus, their speedups remain constant across datasets.

To conclude, the streaming speedups of some applications are sensitive to their input datasets whereas the others are not. And the distribution of speedups on the GPU is more concentrated than XeonPhi. This is because the current GPU implementation does not support processor core partition, the kernel execution time benefits less from multiple streams than XeonPhi.

6.5.3 Correlation Analysis

Fig. 17 shows the relation between the computation-communication ratio and the achieved speedup when using heterogeneous streams across all benchmarks and datasets

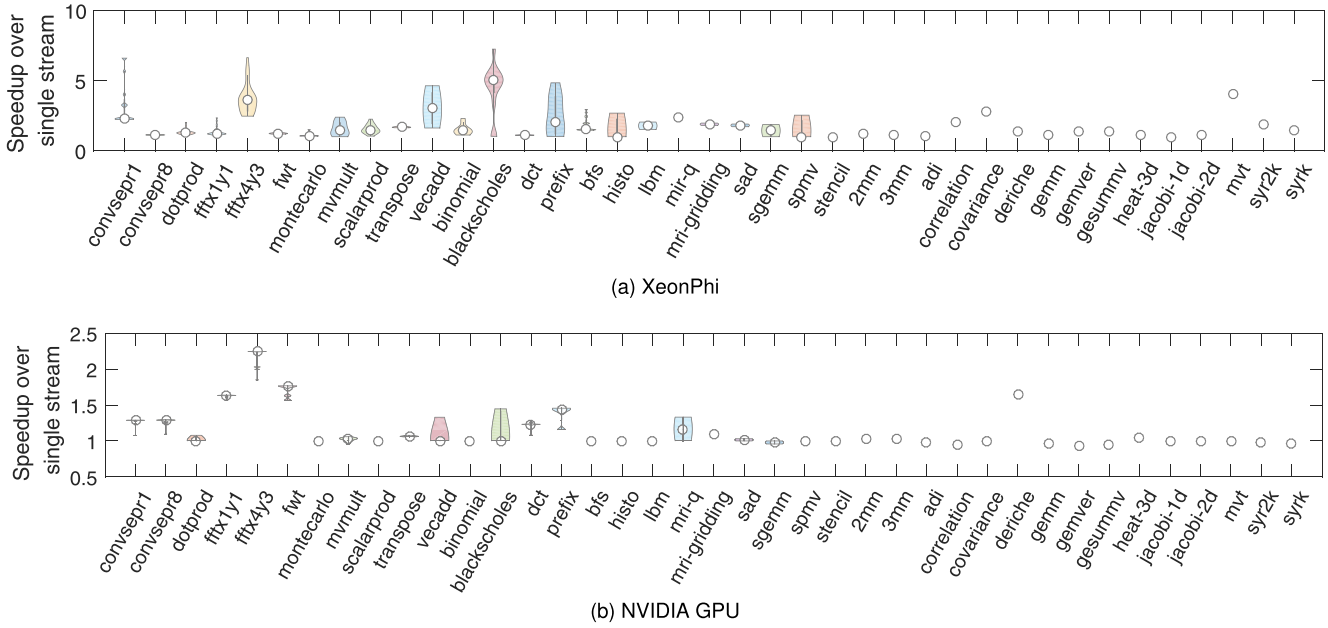


Fig. 16. Violin plot showing the distribution of speedups per benchmark across datasets on XeonPhi (a) and NVIDIA GPU. (b) The shape of the violin corresponds to the speedup distribution. The thick black line shows where 50 percent of the data lies.

on XeonPhi. We see that the computation-communication ratio varies over the benchmarks and the speedup changes accordingly, but in general, a higher computation-to-communication ratio leads to a greater speedup. As explained in Section 6.5.1, in addition to overlapping computation and communication, our approach can also reduce the kernel computation time by choosing the right stream configuration. Therefore, benchmarks with a high computation-communication ratio also benefit from a reduction in the kernel computation time.

To quantify the relation between the computation-communication ratio and the speedup, we calculate the Pearson correlation coefficient of the two variables. The calculation gives a correlation coefficient of 0.7, indicating that the two variables (the computation-communication ratio and the speedup) have a strong linear correlation. By carefully selecting the stream configuration, our approach tries to maximize the overlap between communication and computation, which thus leads to favourable performance.

6.5.4 *Impact of Streaming Parallelism*

Our earlier experiments show that by carefully exploiting streaming parallelism, we can significantly improve application performance. We now take a closer look at three representative benchmarks, `fftx1y1`, `fwt`and `gesummv`, to get a better understanding of streaming performance on XeonPhi. These benchmarks represent different degrees of benefits obtained from streamed parallelism (with a speedup of $2\times$, $1.5\times$ and $1\times$, respectively).

We use the following analytical model to breakdown the execution time of a multi-stream program

$$T = T_m + T_k + T_c - T_o, \tag{1}$$

where T_m is host-device data transfer time, T_k is kernel execution time, T_c is the overhead for initializing the context, and T_o is overlapping time between data transfer and kernel execution. We measure T, T_m, T_k, and T_c, and use the measurements to calculate T_o.

Fig. 18 gives the breakdown for the five components in Equation (1). For each testing program, we compare the single-stream configuration against the best-performing multi-stream configuration. The host-device data transfer time, T_m, is nearly constant among a single and a multiple stream configuration, but multi-streaming can reduce the kernel execution time, T_k, by exploiting the spatial sharing of processing resources among computation tasks. The overhead of initializing the hStreams context, T_c, depends on the kernel execution time. For `fftx1y1` and `fwt`, whose kernels run for a sufficiently long time, this one-off runtime overhead is negligible. However, for `gesummv`, this overhead cannot be

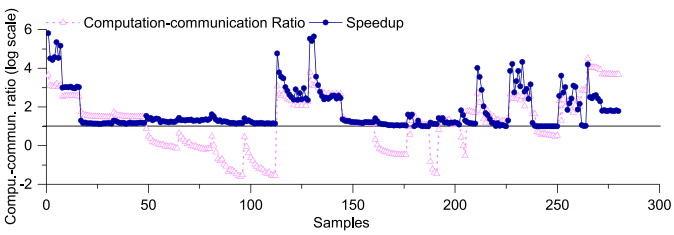


Fig. 17. The relation between computation-communication ratio and the speedup. The computation-communication ratio is normalized using the natural logarithm function. Thus, the kernel computation time equals the host-device communication time when $ratio = 0$. In general, a higher computation-communication ratio leads to a better speedup.

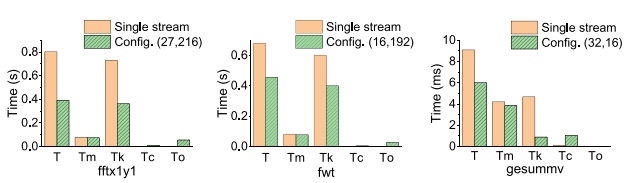


Fig. 18. Breakdown of program execution time (T), host-device data transfer time (T_m), kernel execution time (T_k), hStreams context initialization overhead (T_c) and communication-computation overlapping time (T_o) for single and best-performing multi-stream configurations.

TABLE 5
Comparison to Alternative Modeling Techniques

Technique	Training time	Avg. pred. time	Avg. speedup
SVM (regression)	100 hours	2280 ms	1.56
DCT (regression)	65.57 seconds	0.74 ms	1.51
RF (regression)	317.89 seconds	11.94 ms	1.51
XGB (regression)	28.46 seconds	0.74 ms	1.49
MLP (regression, ours)	245.8 seconds	0.76 ms	1.57
SVM (classifier)	1.28 seconds	0.10 ms	1.53
DCT (classifier)	0.79 seconds	0.05 ms	1.38
MLP(classifier)	46.45 seconds	0.05 ms	1.41
KNN (classifier)	0.22 seconds	0.23 ms	1.43

ignored due to the relatively short kernel running time. The contribution for overlapping host-device communications with kernel execution, T_o, varies across programs. For fftx1y1 and fwt, it accounts for around 50 percent of T_m, suggesting that by exploiting temporal sharing to overlap communication with kernel execution can amortize the host-device communication overhead. For gesummv, T_o is small due to little alignment between data transfer and kernel execution. As such, there is little benefit for exploiting temporal sharing for this program.

This experiment gives a more detailed analysis for the benefits of exploiting multiple streams. The results reinforce our claim that the benefit for streaming parallelism depends on the computation kernel and hence an adaptive scheme for choosing the optimal stream configuration is needed. Our work aims to offer such a capability.

6.6 Analysis of Predictive Modeling Techniques

In this section, we analyse the working mechanism of our predictive model, using XeonPhi as an evaluation platform.

6.6.1 Comparison to Alternative Modeling Techniques

We compare our MLP-based model against four widely used regression methods: the DCT (Decision Tree), the RF (Random Forest), the XGB (eXtreme Gradient Boosting) and SVM (Support Vector Machine) as well as four classification models: SVM, DCT, MLP and KNN (K-Nearest Neighbors). We use the Radial basis function kernel for the SVM models.

For each technique, we follow the same training methodology and use the same features and training examples to build a model. For classification models, we apply the label merging process described in our prior work [16] to improve the prediction accuracy. Table 5 compares the training overhead, average prediction time and achieved average speedup for each model. We note that training a regression-based SVM model has the largest overhead. Although training a DCT has less overhead over our MLP-based regression model, MLP gives better prediction performance. The RF and XGB models are based on DCT, but they do not yield a better performance. Compared to regression models, a classification model takes less time to train and make predictions. However, classification models give worse performance over regression models as they require more training data to cover the optimization space. Overall, we choose to use a regression-based approach and employ MLP because it gives the best overall prediction performance and has modest training overhead.

6.6.2 Feature Engineering

Feature engineering has a significant impact on the performance of a machine learning model (Section 3.2). Here we quantify the impact of feature engineering methods. In this work, we consider three standard feature engineering approaches including standardization, normalization and dimension reduction.

Standardization converts all features value to be in a common range, e.g., between 0 and 1. The idea is to prevent the feature value range to dominate the importance of that feature. In this work we apply a commonly used standardization method called *Z-score* [29] to standardize the raw feature values and the speedups (i.e., prediction targets) in the training data. We found that feature standardization improves the achieved speedup by 3 percent on average, and speedup standardization improves the achieved speedup by 5 percent on average.

Normalization scales the feature values to make them follow the normal distribution. We have tested a range of normalization methods including the square root, the reciprocal of square root and the natural logarithm transformation. However, we found that normalization does not improve our model prediction accuracy.

Dimension reduction reduces the number of features, which is often useful when the number of training examples is not proportional to the number of feature dimensions. In this work, we apply factor analysis (FA) [30] and principal component analysis (PCA) [31] to the raw features. Applying PCA and using 9 PCA components gives the best overall result, by improving the average speedup by 17 percent. PCA outperforms FA which gives an average 3 percent improvement on the achieved speedup.

6.6.3 MLP Parameters Tuning

We now discuss the impact of the MLP parameter choices. There are four configurable parameters for an MLP model: the activation function, the number of hidden layers, the number of neurons, and the learning algorithm (i.e., the solver). For activation functions, we consider identity, logistic, tanh and relu. For hidden layers and neurons, we vary the number of hidden layers from 1 to 5 and the number of neurons per layer from 3 to 100. For the solver, we consider three commonly used weight optimizers: lbfgs, sgd and and adam. We use scikit-learn implementations for the activation function and the solver. Our experimental results suggest that the best-performing activation function and solver are tanh and adam respectively, and using three hidden layers with 9 neurons per layers gives the best overall results on our training data. Overall, tuning MLP model parameter improves the average speedup by 5 percent over the default parameter setting.

6.6.4 Impact of Individual Feature

In this experiment, we consider the impact of a specific feature to the resultant performance. Fig. 19 presents a Hinton diagram illustrating how important a feature contribution to the performance model prediction accuracy (which in turns affects the resulting application performance). The larger the box, the more significant a feature for a given program's performance. Here, the x-axis denotes the programs,

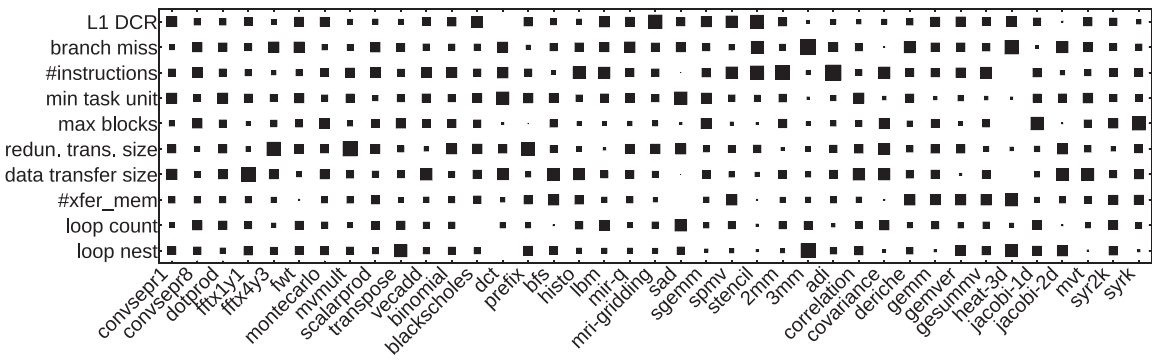


Fig. 19. A Hinton diagram showing the impact of each feature used by the performance model to the resultant application performance. The larger the box, the more likely a feature has a greater impact on the performance of the respective benchmark.

and the y-axis denotes the features used by our performance model. The impact of a feature is quantified by measuring how much speedup improvement can be obtained if that feature is used by the performance model. Note that this is a post-hoc analysis and, in general, we cannot know in advance the importance of a feature on *unseen* programs. Fig. 19 shows that all the features are important for the set of benchmarks targeted in the work, but the importance of features varies across programs. This diagram illustrates how hard it is to develop an analytical model to capture the diverse behaviors and characteristics of streaming programs.

7 Related Work

Our work builds upon the following past foundation, while qualitatively differing from each.

Task Scheduling. There is considerable work on distributing work across heterogeneous processors to improve application performance [32], [33], [34]. Prior work in the area typically assumes that the processor configuration is fixed and relies on the operating system to schedule parallel tasks across processing units. Recent studies show that by partitioning the processing units into groups it is possible to significantly improve the application performance by overlapping the host-device communication and computation on coprocessors like Intel XeonPhi [6], [14]. However, existing approaches rely on static tuning to find the processor partition and the best number of streams to run within a partition. As a result, previous approaches cannot adapt to the change of program inputs. As a departure from prior work, we develop an automatic approach to dynamically adjust the processor partition and task-granularity during runtime, considering the characteristics of applications and input datasets; our approach thereby can adapt to the change of program inputs.

Domain-Specific Optimizations. There is considerable work on domain-specific optimization on Intel XeonPhi. Cheng *et al.* [35] and Jha *et al.* [36] show that in-memory database applications suffer from under-utilization of processor resources and hence a fine-grained tuning approach is required. Mrphi is a framework for optimizing MapReduce workload on the XeonPhi [37]. It employs a set of techniques to improve the resource utilization to obtain higher application performance. Other works look at performance optimization for numerical solvers [38], sparse matrix vector multiplication [39], [40], and dynamic stochastic economic models [39]. Ferrão *et al.* [41] and Memeti *et al.* [42] develop a stream processing framework for XeonPhi to increase the programming productivity. The runtime can automatically distribute workloads across CPUs and accelerating devices. These approaches improve the processor utilization by adjusting the algorithmic design, which are complementary to our work on tuning multi-streaming parallelism for data parallel applications.

Multiple Streams Modeling. Gomez-Luna *et al.* [11] develop a set of models to estimate the asynchronous data transfer overhead on different GPU architectures. The models can be used to estimate the optimal number of streams to use on a given GPU platform. Werkhoven *et al.* [10] present an analytical model to determine when to apply an overlapping method on GPUs. Liu *et al.* [12] also develop an analytical based approach to determine the optimal number of streams to use on GPUs. However, none of these approaches considers the processor partition. As we have shown in Section 6.3, ignoring the processor partitioning parameter can lead to poor performance on Intel XeonPhi. Furthermore, these hand-crafted models have the drawback of being not portable across architectures as the model is tightly coupled to a specific GPU architecture. Our work advances prior work by employing machine learning to automatically learn the optimal processor partition and the number of streams/tasks to use. Since our models are automatically learned from empirical observations, one can easily re-learn a model for a new architecture.

Predictive Modeling. Recent studies have shown that machine learning based predictive modeling is effective in code optimization [43], [44], performance predicting [45], [46], parallelism mapping [20], [47], [48], [49], [50], and task scheduling [51], [52], [53], [54], [55]. Its great advantage is its ability to adapt to the ever-changing platforms as it has no prior assumption about their behavior. The work presented by Wen *et al.* [56] employs `SVMs` to develop a binary classifier to predict that if a given OpenCL kernel can achieve a large speedup or not. Our work differs from [56] in that it targets a different architecture and programming model, and it predicts from a larger number of configurations instead of making a binary prediction. Our prior work developed an `SVM` based classifier to predict the optimal stream configuration for Intel XeonPhi [16]. However, it requires having sufficient training data samples to cover all possible stream configurations. Our approach improves the prior work by directly modeling the impact of the stream configuration. As a result, our approach can make predictions for any stream configuration (even those are not seen in the training data).

Autotuning Parallel Programs. Our approach is closely related to autotuning that searches for the best-performing optimization configuration [57], [58]. This technique is demonstrated to be effective for choosing algorithmic choices [59], tuning GPU code [60], [61], [62], optimizing structured parallel programs [63], [64] and non-uniform memory access (NUMA) architectures [65], and more recently for deep neural networks [66]. Many of the prior works in this area employ an evolutionary-based approach by applying and profiling candidate optimization options to choose a good option to use. One of the key changes of autotuning is how to avoid the profiling overhead which could be prohibitively expensive. We do so by using a performance model to quickly evaluate the profitability of a candidate optimization option. We show that our approach has low runtime overhead, which thus permits us to apply it at runtime to best match the optimization strategy to the program input. Furthermore, our work is the first for tuning heterogeneous streaming parallelism on heterogeneous many-cores (XeonPhis and GPUs).

Automatic Generation of Parallel Programs. The OpenMPC compiler [67] translates OpenMP to CUDA programs. Wang *et al.* [20], [24], [68] translates OpenMP to OpenCL programs and use machine learning to select the most suitable device from the host CPU and the GPU to run the code. Rawat *et al.* presents an automatic approach to generate GPU code from a domain-specific language (DSL) for stencil programs [69]. All of the above approaches target GPUs, and do not utilize the multi-streaming strategy.

8 Conclusion

This article has presented an automatic approach to exploit streaming parallelism on heterogeneous many-cores. Central to our approach is a machine learning-based model that predicts the resulting performance when running the target application under a given streamed configuration. The performance predictor is then used as a cost function to quickly rank candidate configurations at runtime, to determine which stream configuration should be used on a per-program per-dataset basis. We have evaluated our approach on an Intel XeonPhi and an NVIDIA GTX 1080 Ti GPU, with 39 representative benchmarks. Experimental results show that our approach delivers an average speedup of 1.6x and 1.1x on XeonPhi and the GPU, respectively. These results translate to over 93 percent of the best-available performance.

Acknowledgments

This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB0204301, the National Natural Science Foundation of China under Grants 61972408, 61602501, and 61872294.

References

[1] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proc. IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[2] A. Li *et al.*, "Exploring and analyzing the real impact of modern on-package memory on HPC scientific kernels," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2017, Art. no. 26.

[3] C. Chen *et al.*, "LU factorization on heterogeneous systems: An energy-efficient approach towards high performance," *Computing*, vol. 99, pp. 791–811, 2017.

[4] M. R. Meswani *et al.*, "Modeling and predicting performance of high performance computing applications on hardware accelerators," *Int. J. High Perform. Comput. Appl.*, vol. 27, pp. 89–108, 2013.

[5] J. Fang, A. L. Varbanescu, and H. Sips, "A comprehensive performance comparison of CUDA and OpenCL," in *Proc. Int. Conf. Parallel Process.*, 2011, pp. 216–225.

[6] C. J. Newburn *et al.*, "Heterogeneous streaming," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2016, pp. 611–620.

[7] *CUDA C Best Practices Guide Version 7.0*, NVIDIA Inc., 2015.

[8] The Khronos OpenCL Working Group, "OpenCL - The open standard for parallel programming of heterogeneous systems," 2016. [Online]. Available: http://www.khronos.org/opencl/

[9] *hStreams Architecture for MPSS 3.5*, Intel Inc., 2015.

[10] B. Van Werkhoven, J. Maassen, F. J. Seinstra, and H. E. Bal, "Performance models for CPU-GPU data transfers," in *Proc. 14th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2014, pp. 11–20.

[11] J. GóMez-Luna *et al.*, "Performance models for asynchronous data transfers on consumer graphics processing units," *J. Parallel Distrib. Comput.*, vol. 72, pp. 1117–1126, 2012.

[12] B. Liu *et al.*, "Software pipelining for graphic processing unit acceleration: Partition, scheduling and granularity," *Int. J. High Perform. Comput. Appl.*, vol. 30, pp. 169–185, 2016.

[13] Z. Li *et al.*, "Streaming applications on heterogeneous platforms," in *Proc. IFIP Int. Conf. Netw. Parallel Comput.*, 2016, pp. 116–129.

[14] J. Fang *et al.*, "Evaluating multiple streams on heterogeneous platforms," *Parallel Process. Lett.*, vol. 26, 2016, Art. no. 1640002.

[15] Z. Li, J. Fang, T. Tang, X. Chen, C. Chen, and C. Yang, "Evaluating the performance impact of multiple streams on the MIC-based heterogeneous platform," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2016, pp. 1341–1350.

[16] P. Zhang, J. Fang, T. Tang, C. Yang, and Z. Wang, "Auto-tuning streamed applications on Intel Xeon Phi," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2018, pp. 515–525.

[17] 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.

[18] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.

[19] G. Fursin *et al.*, "MILEPOST GCC: Machine learning based research compiler," in *GCC Summit*, 2008.

[20] Z. Wang *et al.*, "Automatic and portable mapping of data parallel programs to OpenCL for GPU-based heterogeneous systems," *ACM Trans. Archit. Code Optim.*, vol. 11, 2015, Art. no. 42.

[21] S. Boslaugh, *Statistics in a Nutshell*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2012.

[22] B. F. Manly, *Multivariate Statistical Methods: A Primer*. Boca Raton, FL, USA: CRC Press, 2004.

[23] S. Lee *et al.*, "OpenMP to GPGPU: A compiler framework for automatic translation and optimization," *ACM SIGPLAN Notices*, vol. 44, pp. 101–110, 2009.

[24] D. Grewe, Z. Wang, and M. F. P. O'Boyle, "Portable mapping of data parallel programs to OpenCL for heterogeneous systems," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2013, pp. 1–10.

[25] D. Mikushin, N. Likhogrud, E. Z. Zhang, and C. Bergström, "KernelGen–the design and implementation of a next generation compiler platform for accelerating numerical models on GPUs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2014, pp. 1011–1020.

[26] T. Grosser and T. Hoefler, "Polly-ACC transparent compilation to heterogeneous hardware," in *Proc. Int. Conf. Supercomput.*, 2016, pp. 1–13.

[27] R. Sotomayor *et al.*, "Automatic CPU/GPU generation of multi-versioned OpenCL kernels for C++ scientific applications," *Int. J. Parallel Program.*, vol. 45, pp. 262–282, 2017.

[28] W. Ertel, "On the definition of speedup," in *Proc. Int. Conf. Parallel Archit. Lang. Europe*, 1994, pp. 289–300.

[29] E. Kreyszig, *Advanced Engineering Mathematics*, 10th ed. Hoboken, NJ, USA: Wiley, 2009.

[30] R. L. Gorsuch, *Factor Analysis*, 2nd ed. Evanston, IL, USA: Routledge, 2014.

[31] H. Hotelling, "Analysis of a complex of statistical variables into principal components," *J. Educ. Psychol.*, vol. 24, pp. 417–441, 1933.

[32] S. Mittal and J. S. Vetter, "A survey of CPU-GPU heterogeneous computing techniques," *ACM Comput. Surv.*, vol. 47, no. 4, 2015, Art. no. 69.

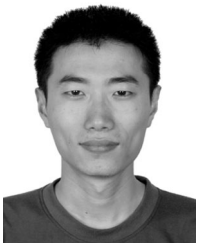
[33] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2009, pp. 45–55.
[34] J. Shen, A. L. Varbanescu, Y. Lu, P. Zou, and H. Sips, "Workload partitioning for accelerating applications on heterogeneous platforms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 9, pp. 2766–2780, Sep. 2016.
[35] X. Cheng *et al.*, "Many-core needs fine-grained scheduling: A case study of query processing on Intel Xeon Phi processors," *J. Parallel Distrib. Comput.*, vol. 120, pp. 395–404, 2018.
[36] S. Jha *et al.*, "Improving main memory hash joins on Intel Xeon Phi processors: An experimental approach," *Proc. VLDB Endowment*, vol. 8, pp. 642–653, 2015.
[37] M. Lu, Y. Liang, H. P. Huynh, Z. Ong, B. He, and R. S. M. Goh, "MrPhi: An optimized MapReduce framework on Intel Xeon Phi coprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 11, pp. 3066–3078, Nov. 2015.
[38] A. Lastovetsky, L. Szustak, and R. Wyrzykowski, "Model-based optimization of EULAG kernel on Intel Xeon Phi through load imbalancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 3, pp. 787–797, Mar. 2017.
[39] W. T. Tang *et al.*, "Optimizing and auto-tuning scale-free sparse matrix-vector multiplication on Intel Xeon Phi," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2015, pp. 136–145.
[40] M. E. Guney *et al.*, "Optimizing matrix multiplication on Intel Xeon PhiTH x200 architecture," in *Proc. IEEE 24th Symp. Comput. Arithmetic*, 2017, pp. 144–145.
[41] P. Ferrão *et al.*, "Stream processing on hybrid CPU/Intel Xeon Phi systems," in *Proc. Eur. Conf. Parallel Process.*, 2018, pp. 796–810.
[42] S. Memeti and S. Pllana, "HSTREAM: A directive-based language extension for heterogeneous stream computing," in *Proc. IEEE Int. Conf. Comput. Sci. Eng.*, 2018, pp. 138–145.
[43] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "End-to-end deep learning of optimization heuristics," in *Proc. 26th Int. Conf. Parallel Archit. Compilation Techn.*, 2017, pp. 219–232.
[44] Z. Wang and M. O'Boyle, "Machine learning in compiler optimisation," *Proc. IEEE*, vol. 106, no. 11, pp. 1879–1901, Nov. 2018.
[45] J. Zhao, H. Cui, J. Xue, and X. Feng, "Predicting cross-core performance interference on multicore processors with regression analysis," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 5, pp. 1443–1456, May 2016.
[46] Z. Wang and M. F. O'boyle, "Using machine learning to partition streaming programs," *ACM Trans. Archit. Code Optim.*, vol. 10, 2013, Art. no. 20.
[47] G. Tournavitis *et al.*, "Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping," *ACM SIGPLAN Notices*, vol. 44, pp. 177–187, 2009.
[48] Z. Wang and M. F. O'Boyle, "Partitioning streaming parallelism for multi-cores: A machine learning based approach," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Techn.*, 2010, pp. 307–318.
[49] Z. Wang *et al.*, "Integrating profile-driven parallelism detection and machine-learning-based mapping," *ACM Trans. Archit. Code Optim.*, vol. 11, 2014, Art. no. 2.
[50] B. Taylor *et al.*, "Adaptive optimization for OpenCL programs on embedded heterogeneous systems," in *Proc. 18th ACM SIGPLAN/SIGBED Conf. Lang. Compilers Tools Embedded Syst.*, 2017, pp. 11–20.
[51] M. K. Emani, Z. Wang, and M. F. P. O'Boyle, "Smart, adaptive mapping of parallelism in the presence of external workload," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2013, pp. 1–10.
[52] V. S. Marco *et al.*, "Improving spark application throughput via memory aware task co-location: A mixture of experts approach," in *Proc. 18th ACM/IFIP/USENIX Middleware Conf.*, 2017, pp. 95–108.
[53] J. Ren, L. Gao, H. Wang, and Z. Wang, "Optimise web browsing on heterogeneous mobile platforms: A machine learning based approach," in *Proc. IEEE INFOCOM*, 2017, pp. 1–9.
[54] J. Ren, L. Gao, H. Wang, and Z. Wang, "Proteus: Network-aware web browsing on heterogeneous mobile systems," in *Proc. 14th Int. Conf. Emerg. Netw. Experiments Technol.*, 2018, pp. 379–392.
[55] L. Yuan, J. Ren, L. Gao, Z. Tang, and Z. Wang, "Using machine learning to optimize web interactions on heterogeneous mobile systems," *IEEE Access*, vol. 7, pp. 139394–139408, 2019.
[56] Y. Wen, Z. Wang, and M. F. P. O'Boyle, "Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms," in *Proc. 21st Int. Conf. High Perform. Comput.*, 2014, pp. 1–10.
[57] K. Datta *et al.*, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proc. ACM/IEEE Conf. Supercomput.*, 2008, pp. 1–12.
[58] J. Ansel *et al.*, "OpenTuner: An extensible framework for program autotuning," in *Proc. 23rd Int. Conf. Parallel Archit. Compilation Techn.*, 2014, pp. 303–315.
[59] J. Ragan-Kelley *et al.*, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2013, pp. 519–530.
[60] A. Nukada and S. Matsuoka, "Auto-tuning 3-D FFT library for CUDA GPUs," in *Proc. Conf. High Perform. Comput. Netw. Storage Anal.*, 2009, pp. 1–10.
[61] P. Tillet and D. Cox, "Input-aware auto-tuning of compute-bound HPC kernels," in *Proc. Conf. High Perform. Comput. Netw. Storage Anal.*, 2017, pp. 1–12.
[62] T. T. Dao and J. Lee, "An auto-tuner for OpenCL work-group size on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 2, pp. 283–296, Feb. 2018.
[63] U. Dastgeer *et al.*, "Auto-tuning SkePU: A multi-backend skeleton programming framework for multi-GPU systems," in *Proc. 4th Int. Workshop Multicore Softw. Eng.*, 2011, pp. 25–32.
[64] J. J. Thiagarajan *et al.*, "Bootstrapping parameter space exploration for fast tuning," in *Proc. Int. Conf. Supercomput.*, 2018, pp. 385–395.
[65] T. Katagiri, S. Ohshima, and M. Matsumoto, "Auto-tuning on NUMA and many-core environments with an FDM code," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2017, pp. 1399–1407.
[66] L. Liao *et al.*, "UHCL-darknet: An OpenCL-based deep neural network framework for heterogeneous multi-/many-core clusters," in *Proc. 47th Int. Conf. Parallel Process.*, 2018, Art. no. 44.
[67] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP programming and tuning for GPUs," in *Proc. ACM/IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2010, pp. 1–11.
[68] Z. Wang *et al.*, "Exploitation of GPUs for the parallelisation of probably parallel legacy code," in *Proc. Int. Conf. Compiler Construction*, 2014, pp. 154–173.
[69] P. S. Rawat *et al.*, "Domain-specific optimization and generation of high-performance GPU code for stencil computations," *Proc. IEEE*, vol. 106, no. 11, pp. 1902–1920, Nov. 2018.

Peng Zhang received the master's degree in computer science from the National University of Defense Technology (NUDT), Changsha, China, in 2016, where is currently working toward the PhD degree. His research interests include heterogeneous programming, performance optimization of parallel programs, and source-to-source compilers.

Jianbin Fang received the PhD degree from the Parallel and Distributed System Group, Delft University of Technology, Delft, The Netherlands. He is currently an assistant professor in computer science with NUDT. His research interests include parallel programming for many-cores, parallel compilers, performance modeling, and scalable algorithms.

Canqun Yang is currently a full processor in computer science with NUDT. His research interests include performance analysis of high-performance computing systems, parallel compilers, parallel programming, and high-performance computing applications.

Chun Huang is currently a full processor in computer science with NUDT. Her research interests include high-performance computing systems, parallel compilers, parallel programming, and high-performance math libraries.



Tao Tang received the BSc, MSc, and PhD degrees from the National University of Defense Technology (NUDT), Changsha, China. He is currently an associate processor in computer science with NUDT. His research interests include compilers, parallel programming, and high- performance computing.

Zheng Wang is currently an associate professor with the University of Leeds. His research cut across the boundaries of parallel program optimisation, systems security, and applied machine learning. He received four best paper awards for his work on machine learning-based compiler optimisation (PACT '10, CGO '17, PACT '17, and CGO '19).

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

CROCUS: Enabling Computing Resource Orchestration for Inline Cluster-Wide Deduplication on Scalable Storage Systems

Prince Hamandawana, Awais Khan, *Member, IEEE*, Chang-Gyu Lee, Sungyong Park, and Youngjae Kim, *Member, IEEE*

Abstract—Inline deduplication dramatically improves storage space utilization. However, it degrades I/O throughput due to compute-intensive deduplication operations such as chunking, fingerprinting or hashing of chunk content, and redundant lookup I/Os over the network in the I/O path. In particular, the fingerprint or hash generation of content contributes largely to the degraded I/O throughput and is computationally expensive. In this article, we propose CROCUS, a framework that enables compute resource orchestration to enhance cluster-wide deduplication performance. In particular, CROCUS takes into account all compute resources such as local and remote *{CPU, GPU}* by managing decentralized compute pools. An opportunistic Load-Aware Fingerprint Scheduler (LAFS), distributes and offloads compute-intensive deduplication operations in a load-aware fashion to compute pools. CROCUS is highly generic and can be adopted in both inline and offline deduplication with different storage tier configurations. We implemented CROCUS in Ceph scale-out storage system. Our extensive evaluation shows that CROCUS reduces the fingerprinting overhead by 86 percent with 4KB chunk size compared to Ceph with baseline deduplication while maintaining high disk-space savings. Our proposed LAFS scheduler, when tested in different internal and external contention scenarios also showed 54 percent improvement over a fixed or static scheduling approach.

Index Terms—Distributed file systems, scheduling, storage management

1 INTRODUCTION

THE explosive increase in data has made the problems of data storage space worse in the cloud [1], [2]. Cloud providers employ shared-nothing scale-out storage systems (SN-SS) such as Ceph [3] and GlusterFS [4] for high performance, scalability, and availability and to seamlessly suffice storage demands. In recent years, data deduplication [5], [6], [7], [8], a capacity optimization technique that is being used to dramatically improve storage efficiency, has emerged as an effective alternative to compression in backup storage for reducing storage demands. With the emergence of high-speed SSDs, the trend towards deploying tiered hybrid architecture is gaining more importance [9]. Configuring such hybrid systems (SSD and HDD) to achieve performance goals at a minimum cost remains a challenge [10]. Data deduplication on the SSDs would be at the forefront of the tiered architectures to minimize storage costs and increase SSD endurance without impairing storage performance.

- *P. Hamandawana is with the Department of Computer Engineering, Ajou University, Suwon 16499, Republic of Korea. E-mail: phamandawana@ajou.ac.kr.*
- *A. Khan, C. Lee, S. Park, and Y. Kim are with the Department of Computer Science and Engineering, Sogang University, Seoul 04107, Republic of Korea. E-mail: {awais, changgyu, parksy, youkim}@sogang.ac.kr.*

Manuscript received 28 July 2019; revised 28 Jan. 2020; accepted 31 Jan. 2020. Date of publication 11 Feb. 2020; date of current version 18 Mar. 2020. (Corresponding author: Youngjae Kim.) Recommended for acceptance by D. Talia. Digital Object Identifier no. 10.1109/TPDS.2020.2972882

Deduplication can be applied to the SSD tier either inline or offline. Inline deduplication introduces additional latency and performance degradation in the I/O path, whereas, offline deduplication requires high storage capacity in order to stage data temporarily. In particular, additional latency and performance degradation in inline deduplication are mainly attributed to compute-intensive operations of deduplication. This includes dividing an object into small fixed or variable-size chunks, computing fingerprint (FP) against each chunk using hash functions such as SHA1 or SHA512 and fingerprint indexing. Recently, there have been studies that have implemented cluster-wide inline deduplication in Ceph [11], [12]. These studies implemented cluster-wide deduplication following shared-nothing constraints, without impairing the scalability and fault tolerance of scale-out storage systems [11], [12]. However, in both studies, fingerprinting is computationally intensive, and in the case of small chunks, the performance degradation becomes critical due to frequent fingerprint operations.

Due to the many-core architectures in modern servers, there exist opportunities to overcome such latency and performance degradation problems in the current state of the art deduplication systems [13], [14]. By leveraging such powerful architectures, we can utilize the multi-threaded invocation of fingerprint operations to enhance the deduplication performance. On the other end, the CPU usage trend of the scale-out storage system suffocates the multi-thread deduplication performance [13]. For example in Ceph, the

1045-9219 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

frequent Object Storage Daemon (OSD) operations and internal cluster traffic such as replication, heartbeat messages, and storage rebalancing generates high CPU load, giving little room for performance improvement from the multi-threaded invocation of FP operations [15]. Nevertheless, the availability of idle CPU resources across the cluster creates opportunities to offload FP operations. Moreover, most cloud providers are now investing heavily in GPU equipped servers to handle compute-intensive workloads [16]. In the presence of GPUs in the cluster, the overhead of FP operations can be minimized, because of their high computing power, which is derived from their multi-threaded and many core parallel architecture [17], [18].

On the flip side, the current deduplication systems does not have a framework to utilize the computational power of idle CPUs or GPUs in neighboring servers across the cluster. Consequently, there is a need for an efficient communication framework and a dynamic resource scheduler that maximizes the use of available computational resources to enhance the deduplication performance. To build such a framework, there exist several challenges which need to be addressed. For example, (i) there is a need for a framework that provides a global view of the compute resources, that is, the distributed CPU and GPU resources in the cluster. However, the framework for providing a global view, when implemented in a centralized manner, not only raises scalability issues but also violates the design properties (shared-nothing) of the target architecture. (ii) The framework should be adaptive to dynamic load changing scenarios, to subdue under or over-utilization of compute resources. In this framework, reducing unnecessary data transfer and communication overhead is paramount.

To address such challenges, we propose CROCUS, a high-performance deduplication framework equipped with effective compute resource management, which provides a global view of the compute resources (CPU and GPU) available in the cluster. Crocus offloads deduplication jobs to each of the pool based on load-awareness using a multi-stream communication model to enhance data transfer and to minimize communication overhead. The proposed framework is highly generic and can be adopted in both inline and offline deduplication with different storage architectures such as single and multi-tier storage setups.

This paper makes the following specific contributions.

- We build Crocus, a cluster-wide deduplication framework for multi-tier hybrid storage architectures, capable of eliminating duplicates with negligible performance degradation. Crocus provides an up-to-date view of available compute resources (CPU/GPU), without violating the design properties of shared-nothing storage systems.
- We design a simple yet dynamic Load-Aware Fingerprint Scheduler (LAFS), which is an efficient compute-resource orchestrator. LAFS parallelizes inline deduplication operations on available computing resources such as local or remote CPU and GPU across the cluster. It then distributes the compute-intensive deduplication operations to idle CPU and GPU resources in a load-aware fashion.
- We implement Crocus in Ceph [3]. Our real testbed evaluation shows that CROCUS with LAFS scheduler improves the fingerprinting performance by 86 percent with 4 KB small chunks compared to a LAFS-free system. In addition, we achieve high-disk space savings using 4 KB small chunks with low performance degradation of 6 percent compared to Ceph without deduplication.

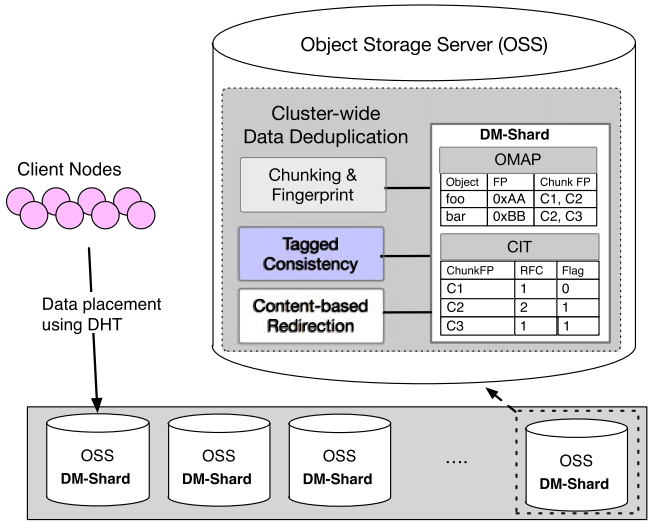


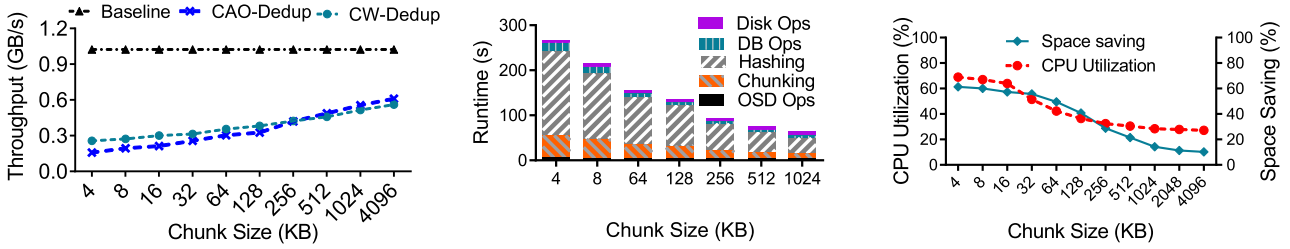
Fig. 1. Overview of inline cluster-wide data deduplication [11].

2 Background and Motivation

2.1 Cluster-Wide Deduplication

Cluster-wide deduplication has been widely explored to improve space efficiency in storage sytems. In general, deduplication is adopted either inline or offline. However, this design decision highly depends on the underlying storage cluster, as it directly impacts its performance. For instance, inline deduplication offers instant space savings but at the cost of performance degradation as it includes deduplication operations such as chunking, hashing and duplicate lookup I/Os in the critical I/O path. On the contrary, offline deduplication also referred to as lazy deduplication requires a temporary data staging buffer space to store data. Note that, the offline approach does not include deduplication operation in the I/O critical path but incurs a double write problem, that is, i) all incoming data is written to the temporary staging buffer space and ii) then after applying deduplication, unique data is written to permanent storage again. This double write problem is harmful when the storage used is SSDs. Moreover, offline deduplication requires certain controls, e.g., a size threshold on staging buffer space to trigger deduplication, which often conflicts with incoming write I/Os. As our target system is a shared-nothing tiered hybrid architecture, where SSD servers overlay slow HDD based storage for higher performance, deploying inline deduplication provides the best choice due to limited SSD storage endurance and higher cost ($/GB) issues.

Although the cluster wide deduplication has been explored in several studies [11], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], these contemporary deduplication techniques cannot be directly adopted because of two major reasons; (i) They do not comply with the shared-nothing



(a) Performance comparison (b) Time breakdown analysis (c) CPU utilization and space savings

Fig. 2. Performance analysis of cluster-wide deduplication when employed on hybrid Ceph cluster. We used Testbed-1 with SSDs, HDDs, and inactive GPUs.

design property of the target architecture. (ii) They were not designed to work effectively on scalable tiered hybrid data storage model. There exist two most recent studies which proposed cluster-wide data deduplication without violating the design properties of shared-nothing storage systems, [11], [12]. The work in [11] manages the deduplication metadata using the partitioned database approach (DM-Shard), where each horizontally partitioned database instance (also called DB-Shard) is tightly integrated to each object storage node in the cluster as shown in Fig. 1. Each DM-Shard stores two persistent data structures, i.e., Object Map (OMAP) containing layout information of objects and Chunk Information Table (CIT) containing sensitive deduplication metadata such as chunk fingerprint, reference count and consistency flag. On the other hand, [12] proposed to handle the deduplication metadata in the form of a content addressable object (CAO), which is stored just like any other object in the underlying storage. In order to access the metadata information, the use of the extended attribute (xattr) is adopted. Both approaches assured conformity to the shared-nothing design constraint.

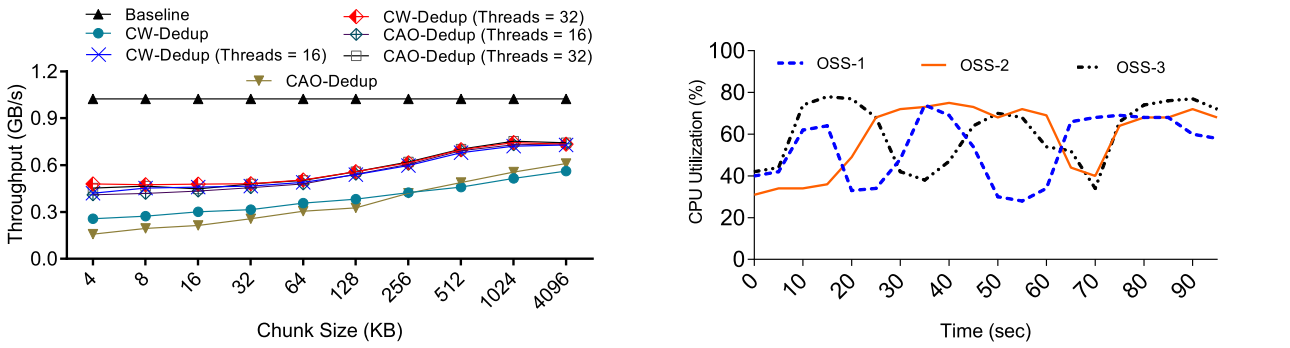
2.2 Motivation

To motivate our research, we first investigated the adoption of the two recent inline cluster-wide deduplication techniques, compatible with the shared nothing design storage systems (CW-Dedup implementation [11] and CAO-Dedup implementation [12]), on top of a hybrid tiered Ceph cluster. We configured the Ceph cluster by using Testbed-I with no active GPUs as shown in Table 1 (Section 5). We used the fixed-size chunking algorithm and SHA-1 hash function to compute the hash fingerprints for each data chunk. Note that other content defined chunking methods such as FASTCDC [14] can be applied to CROCUS. However, this work focuses on overcoming the shortcomings of existing deduplication frameworks. As workloads for evaluation, we generated a total of 2 TB writes per client via the SPEC SFS benchmark [29]. For this particular experiment, we used 5 client nodes each hosting 16 threads issuing write IOs. Note that we disabled GPUs on all the nodes of Testbed-I to clearly investigate the influence of deduplication on performance. We issued random write requests to the Ceph cluster and on each successive experiment, object size was fixed to 4 MB but doubled the chunk size by 2. The results in Fig. 2 show the outcome of our investigation on the existing inline cluster-wide deduplication techniques.

Fig. 2a, shows that adopting the current deduplication implementations [11], [12], introduces a huge performance degradation as compared to the baseline without deduplication. This performance penalty is worse with small-sized chunks. To investigate the cause of the performance overhead, we carried out a time breakdown analysis on CW-Dedup [11], for a single dedup IO. Fig. 2b shows the results. CAO-Dedup [12] shows high performance penalty as compared to CW-Dedup [11] due to the additional metadata checks of CAO objects on storage. So, we focused our investigation of the performance penalty on CW-Dedup. Fig. 2b shows the aggregated runtime for each deduplication operation for the whole dataset. We observed that the overhead of computing fingerprints is considerably large. It contributes to more than 65 percent of the aggregated runtime. Because the current implementation uses a single thread to invoke hash operations, the overhead of fingerprinting is quite high and a more robust approach is required to reduce this performance penalty. Fig. 2c shows the CPU overhead and storage space gain in the object storage server depending on the chunk size. The smaller chunk size benefits with high space savings. However, small chunk sizes show high CPU utilization which averages 65 percent across the storage servers. On the contrary, the large chunk sizes result in lower CPU occupancy but at the cost of low disk space savings. Even though the current deduplication implementation is single threaded, but fingerprinting operation shows quite a high CPU utilization especially with the smaller chunks which achieves more deduplication benefits.

2.3 Deduplication Improvement Opportunities

Since the current cluster-wide deduplication implementations uses a single thread fingerprinting approach, opportunities to improve the performance can arise with multi-threaded fingerprinting invocations. Fig. 3a shows the results of employing multi-thread invocations of the hash function to compute fingerprints on CW-Dedup and CAO-Dedup. We observe that, for both implementations, using 16 threads will increase the throughput, but the performance is still degraded as compared to baseline with no deduplication. Nevertheless, increasing to 32 threads does not improve any performance. This is because fingerprint computation is a CPU bound operation. In scale-out storage systems, there are frequent object storage daemon operations and continuous internal communication which enforces replication, load re-balancing and node heartbeat messages. This generates high amounts of CPU load and suppresses the performance improvement from the multi-threaded dedup approach. In our case, the performance saturates at 16 threads. However, in distributed clusters, there exist pockets of CPU idle times or low CPU



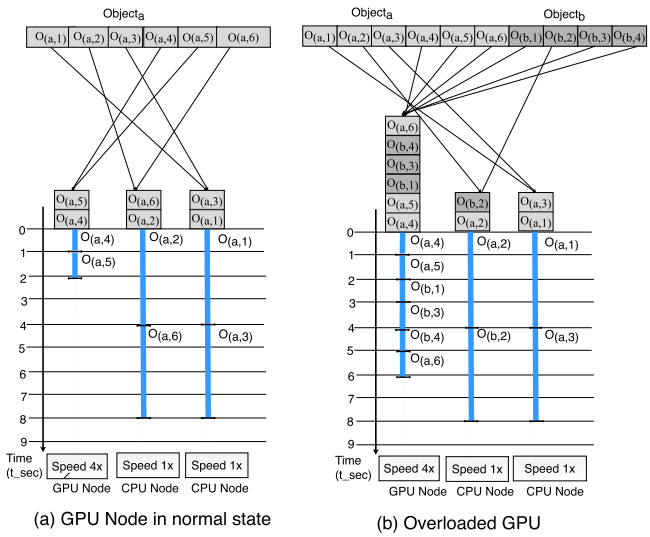
(a) Multi-threaded deduplication analysis (b) CPU utilization snapshot on SSD tier Object Storage Servers (OSSs)

Fig. 3. Improving the deduplication performance via multi-threaded deduplication.

utilization on object storage servers (OSSs). Fig. 3b shows a snapshot of the CPU utilization of the SSD tier nodes on our Ceph cluster, using 4 KB chunking, over a period of 90 seconds during a write request to the cluster. These pockets of low CPU utilization shown in the snapshot provide opportunities to use computing resources on neighboring storage servers when a server has no available computational resources.

Futher, if cluster nodes are GPU equipped, then we can leverage the massive parallelism provided by GPUs to accelerate the overhead of fingerprint computations. There are two scenarios to adopt when setting up GPU nodes on the faster SSD tier. One case is a homogeneous GPU-equipped SSD tier and the other case is when not every server on the SSD tier is GPU-equipped. However, the second case creates computational heterogeneity on the SSD tier. This heterogeneity leads to some bottleneck, since the CPU based fingerprinting of non-GPU servers slows down the I/O performance. We explain this problem using Fig. 4, where object_a is striped into multiple smaller objects $O_{(a,1)}$ to $O_{(a,6)}$ and written to different SSD tier nodes for deduplication. In Fig. 4a, we use an example where deduplication on GPU node is 4x faster as compared to CPU node. Let t be the time unit to compute one object chunk. The time to compute deduplication of object $O_{(a,4)}$ and object $O_{(a,5)}$ placed on the GPU node will be $2t$. However, the time to compute the same number of objects placed on the other 2 CPU nodes will be 8t. This entails that the overall deduplication time for Object_a is determined by the tail dedup operations on slower CPU nodes. Assuming negligible network overhead, offloading objects $O_{(a,1)}$, $O_{(a,2)}$, $O_{(a,3)}$ and $O_{(a,6)}$ onto GPU node will reduce the overall deduplication time from $8t$ to $6t$. This shows the need to offload slower deduplication processes from CPU to GPU node.

However, there are situations where the GPU node is overloaded (depicted by Fig. 4). In this case, blindly offloading deduplication computations to the GPU may result in worse performance. For example, In Fig. 4, where two files, Object_a and Object_b, from two different client nodes, are written to the SSD tier nodes. We see that when normal scheduling is done for the two objects, a and b, the processing time will complete in time 8t. However, if we consider offloading all the objects ($O_{a,1}$, $O_{a,2}$, $O_{a,3}$, and $O_{b,2}$) from CPU nodes to the GPU node, just because of its higher computational power, each object ($O_{a,1}$, $O_{a,2}$, $O_{a,3}$, and $O_{b,2}$) will add a processing time of 1t to the GPU node, hence increasing the total processing time from 8t to 10t. Therefore, there is a need for CPU nodes to dynamically schedule the amount of data to offload to GPU nodes based on the workload intensity of GPU nodes.



(a) GPU Node in normal state (b) Overloaded GPU

Fig. 4. Illustration of performance degradation due to redirection without consideration of GPU load.

To this end, we are motivated to propose CROCUS, a holistic approach which orchestrates the distribution of deduplication processes to all participating compute resources (local/remote CPU and GPU nodes).

3 CROCUS ARCHITECTURAL OVERVIEW

The primary design goal of CROCUS is to build a high performance inline cluster-wide deduplication framework, capable of minimizing the performance degradation introduced by deduplication operations. These operations include chunking and fingerprinting at the SSD tier of the tiered storage system. Fig. 5 presents the architectural overview of CROCUS. The SSD and HDD tiers depict the hybrid nature of scale-out storage system. Importantly, only SSD tier nodes are equipped with CROCUS as shown in Fig. 5 whereas, HDD tier is simply responsible for storing the unique data.

The client nodes perform object name hashing via DHT-based algorithm such as CRUSH in Ceph [3] to place or retrieve objects in the cluster. Due to the tiered architecture, the object placement location is computed to one of the SSD tier nodes. Once the I/O lands on an SSD tier node, CROCUS

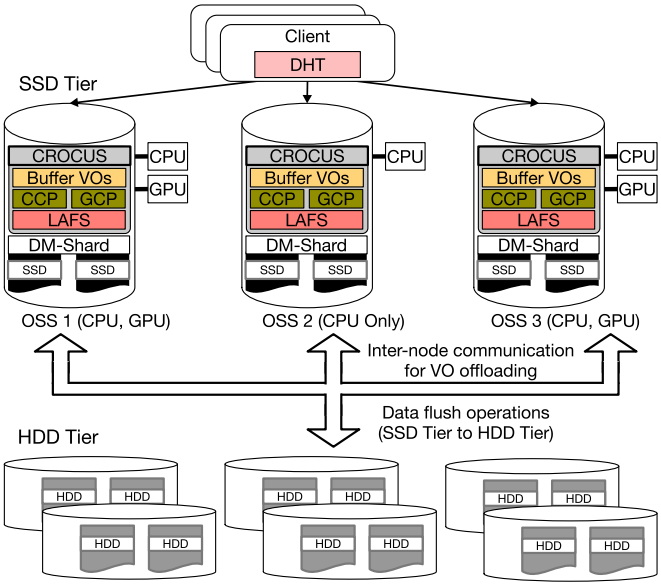


Fig. 5. Overview of Crocus architecture.

divides the objects into smaller fixed size chunks and is required to compute the fingerprint using a cryptographic hash function such as SHA1 or SHA512. In this study, we chose fixed size chunking to illustrate the benefits of Crocus. However, Crocus is also applicable for variable size chunking. The direct fingerprint computation using local CPU is not optimal and might degrade the deduplication performance. Crocus is equipped with an intelligent, load-aware fingerprint scheduler, which offloads the fingerprint operations to compute resources such as local and remote CPU or GPU based on the availability of each resource. In order to use remote resources while avoiding network overhead, Crocus batches several objects into a single large object unit, which we call virtual object (VO) and offloads it onto available CPU or GPU resource for effective fingerprint computation.

Crocus manages resource availability through maintaining two data structures globally across each SSD tier node, i.e., CPU Computation Pool (CCP) consisting of all SSD tier nodes with CPU (as it is obvious that all nodes are equipped with CPU) and GPU Computation Pool (GCP) consisting of SSD tier nodes equipped with GPU accelerator units. These data structures not only assist in optimal fingerprint computation but also empower Crocus to handle resource heterogeneity, if any, among SSD tier nodes. In cases of this resource heterogeneity, where not every SSD tier node is equipped with GPU accelerators, client nodes are unaware of such GPU availability on SSD tier nodes. Worse off, clients can issue write I/Os directly to SSD tier nodes without GPU hardware. The details of VO, LAFS, and data structures are provided in Section 4.

Once fingerprints are computed against the VOs then, deduplication related metadata is updated in the local Deduplication Metadata Shard (DM-Shard). After deduplication metadata operations, unique data chunks are redirected for real-time storage and I/O transaction finishes. Note that client immediately receives an ack message when dedup metadata is successfully added in DM-Shard. This reduces the additional latency in I/O path. In order to ensure data and metadata consistency, we rely on the asynchronous tagged-consistency provided by [11], which provides for eventual consistency for the chunks with updated metadata but not written on disk.

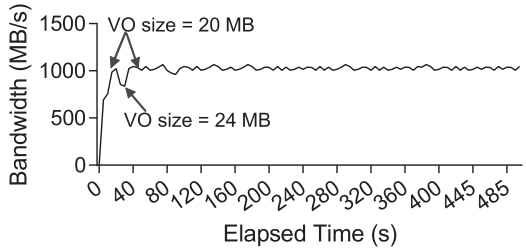


Fig. 6. Automated VO re-sizing.

4 Design and Implementation

4.1 Compute Resource Pooling

To devise a resource and load-aware fingerprint scheduler in Crocus, we formulate our model as a compute-resource mapper, which takes into account several thresholds and constraints. The threshold includes buffering of objects up to a certain limit at each SSD tier node, and constraints such as inline deduplication and minimizing additional latency in fingerprint computation are considered. We achieve this by characterizing the available compute resources into a vector pair consisting of two pools, i.e., CCP and GCP. All the SSD tier nodes along with their CPU utilization ratios are managed by the CCP, whereas the GCP is responsible for maintaining the GPU utilization ratios of GPU equipped SSD tier nodes.

In GPU heterogeneous SSD tiers, it is possible that the number of nodes in CCP can be a magnitude higher than the number of nodes in GCP, which motivates us to consider neighbor CPU/GPU resources in our model formulation. We observed from Fig. 3b, that, there exist opportunities to exploit idle CPU resources residing on the SSD tier nodes to amortize the fingerprint computation cost. Both CCP and GCP maintain a list of SSD tier nodes sorted based on each resource availability with respect to the number of admissible VOs for service.

4.2 Virtual Object and Partition Creation

Our model uses the Virtual Object as a basic unit to offload or schedule a fingerprint operation on remote SSD tier nodes. Each VO consists of multiple objects batched together. We set the VO size as a tunable parameter for any cluster setup. However, a challenge arises on how to determine the optimal VO size for any given cluster configuration. In order to overcome this problem we designed an automated VO resizing strategy that optimizes the best VO size for a given cluster hardware configuration.

To choose the optimum VO size, Crocus first initiates the VO size equal to the original object size, i.e., 4 MB, and then increases the VO size by a single object after every 5 seconds. Fig. 6 shows how the automated VO resizing works. After 20 seconds, when the VO size would have adjusted to 20 MB, Crocus reaches an optimal performance. Increasing the VO size after that degrades the performance. This degradation is reflected in Fig. 6 after 25 seconds when the VO size is adjusted to 24 MB, in which the performance drops by approximately 12 percent.

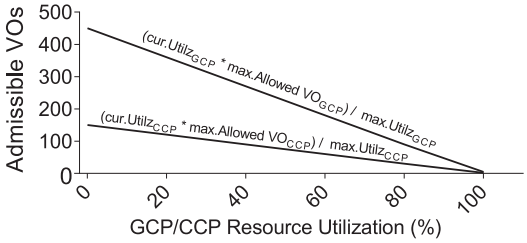


Fig. 7. Admissible number of VOs in GCP and CCP with changing utilization ratio of each compute pool.

When there is a performance drop of more than 5 percent, CROCUS reverts the VO size to the previously allocated VO size with best performance. We set this 5 percent value to minimize the number of VO size adjustments. After the adjustment point CROCUS maintains the VO size as optimum. The CCP or GCP then permits scheduling of VOs as per available compute resource bandwidth.

For example, every GCP node computes the admissible VOs for scheduling using the following equation;

$$(cur.utilz_{GCP} * max.allowedVO_{GCP})/max.utilz_{GCP}. \quad (1)$$

Similarly, all CCP nodes compute the number of VOs to be scheduled using the following equation;

$$(cur.utilz_{CCP} * max.allowedVO_{CCP}) \ /max.utilz_{CCP}. \quad (2)$$

Cur. utilz in equations denotes the currently available GPU and CPU nodes utilization, which is computed simply as 100 percent - CPU/GPU utilization. *Max. allowed VO* is the maximum number of VOs admissible to CPU or GPU resources (This is the case when the GPU or CPU resources are free) and *Max.utilz* is maximum utilization ratio of GPU or CPU resources. The most important parameter is the *max.allowed VO* which denotes the maximum number of VOs admissible to the GCP and CCP nodes when their utilization is close to zero. This number is specified in Section 5.1 under system parameters. In the above modeling, the max. utilz and max.allowed VO are constants and the number of admissible VOs are therefore determined by the cur.utilz value which represents the varying resource utilization ratio at any given time.

We observe that the number of admissible VOs at any given instant is directly proportional to the available GCP/CCP node compute resource utilization ratios. Fig. 7 reflects the results of our VO admission model, which shows a linear decrease in the VO intake of both the GCP and CCP nodes with increasing resource utilization of the nodes. The results of this model motivates us to prioritize the GCP pool nodes due to their higher VO admission rates. Finally, after computing the permissible number of VOs, each compute pool node, i.e., CCP or GCP nodes periodically broadcast its remaining number of VO intake or current utilization ratio. However, with the co-location of GPUs and CPUs in the storage nodes, CROCUS prioritizes scheduling of local VO buffers to local GPU over remote VO buffers as they execute faster. CROCUS differentiates the local and remote buffers due to local DMA tagging and remote RDMA tagging.

We allocate a certain memory buffer area on each SSD tier node, which we called a Partition. All incoming data objects from clients are staged for fingerprinting scheduling in this

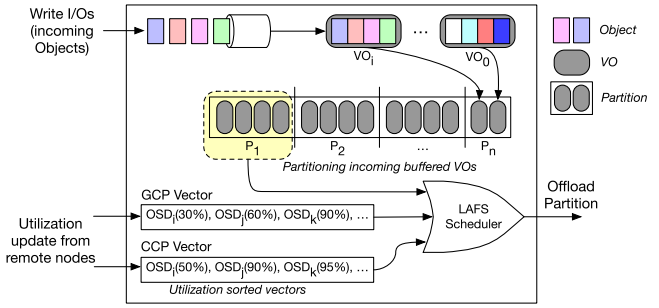


Fig. 8. Partition scheduling in CROCUS LAFS scheduler.

partitioned area as shown in Fig. 8. Each partition consists of VO_1 up to VO_n. We avoid scheduling based on individual objects as it increases the node to node data transfer and introduces communication overheads. Once a VO is scheduled to a remote node, it will be treated as another incoming object on the remote node by default. Hence, the scheduled VO can be rescheduled again to another node. In this way, the VO can continue to be rescheduled to other nodes. This phenomenon is called a circular scheduling problem. We avoid this circular scheduling problem by separating scheduling/partition buffers from the deduplication process buffers. Consequently, the scheduled VO only land in the deduplication process buffers.

To this end, we have identified a few system tunable parameters on which the performance of LAFS can vary. These parameters include, (i) maximum admisable VOs for CPU/GPU pools of each SSD tier node and (ii) resource utilization vector update interval. These parameter values depend on workload and hardware characteristics. These values are set as configuration parameters and can be tuned to the required default settings. Therefore, finding the optimal values for these tunable parameters is not the scope of this study.

4.3 Dynamic Load-Aware Fingerprint Scheduler

The design motivation behind dynamic load-aware fingerprint scheduler includes the efficient distribution of deduplication operations, such as chunking and fingerprinting, across all available compute resources on the SSD tier nodes. This eradicates the idle times of compute resources in all nodes on the SSD tier, hence achieving maximum resource utilization. LAFS is the core scheduling engine embedded on each SSD tier node, responsible for cluster-wide data deduplication. LAFS makes scheduling decisions on incoming VOs as shown in Fig. 8. Then, with the utilization ratios provided by the GCP and CCP pools, the LAFS scheduler assign the VOs in the current scheduling partition, by referencing the current utilization ratios of compute resources in GCP and CCP. This mapping prioritizes GCP over CCP due to the higher throughput and performance of GPUs. If no GPU-equiped nodes are present in the cluster, LAFS directly schedules the VOs onto the CCP pool nodes. Recalling, both GCP and CCP maintain a vector of weighted nodes in a sorted manner of respective compute pools.

When two remote SSD tier nodes have an equal current utilization ratio, LAFS scheduler randomly selects the node to schedule among the two. In case the list of GCP pool is fully exhausted, LAFS shifts the distribution of VOs to the CCP pool nodes. Note that if remote nodes fail whilst

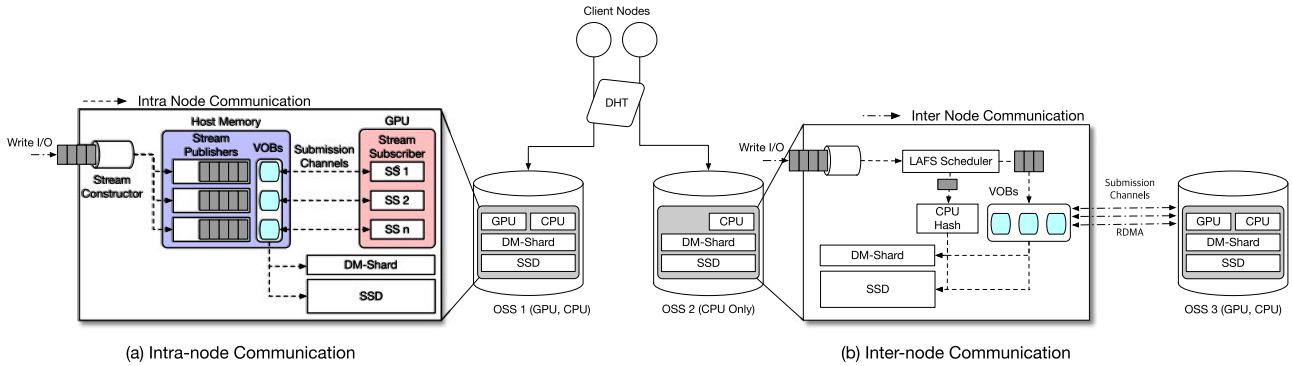


Fig. 9. a) Crocus intra-node communication optimization in GPU-equipped nodes and (b) inter-node scheduling mechanism in CPU nodes.

executing fingerprint operations, metadata would not have been updated. This makes our system free from such failures which might cause any inconsistencies, since we push metadata to persistent storage only when transaction completes. Another critical but rare case which is highly dependent and affected by vector update interval is that, two or more nodes can simultaneously offload VOs to a remote node which cannot facilitate both VOs. This results in the over-scheduling of VOs to that remote node. To cushion such bursty VO offloading cases, we set an abstracted over-provisioned buffer area in all nodes, which provides some elasticity in these over-scheduling scenarios.

4.4 GPU and Communication Optimization

To maximize inline deduplication throughput on GPU equipped nodes, i.e., GCP pool, efficient data transfer techniques and effective memory management approaches must be employed on both CPU host and GPU acceleration unit. In particular, when using local/remote GPU acceleration units, we aim to eradicate the serialization dependency between data transfers and the GPU computations that occur in the default GPU computational model. We implement multi-stream communication channels between CPU memory and local/remote GPU memory as depicted by Fig. 9. The communication protocol is composed of a Stream Constructor (SC) which constructs stream channels from incoming data. All GPU threads executing in a specific stream are called Stream Subscribers (SS) and the corresponding CPU mapped memory per stream is called the Stream Publisher (SP.) This multi-stream design enables interleaving of deduplication tasks running on GPUs as much as possible, thus maximizing the high parallelism offered by GPUs.

We implemented a sliding window chunking and fingerprinting kernel, which computes fingerprints from the beginning of GPU memory address space of each stream. Depending on chunk size specification, SS threads slide by offset positions equal to the chunk size whilst calculating the fingerprint of each chunk. Once FPs for all the chunks in the stream are computed, the FP values are transferred back to host memory and stream is then destroyed clearing up buffer area for new streams to be created. Fingerprint lookup is then issued to respective DM-Shard to validate if the data chunks existence in storage or not.

Intra-Node Multi-Stream Offloading of VOs to GPU via DMA Operations. This occurs when data is placed on a GPU-equipped node and LAFS schedules to use its node local GPU, as depicted by Fig. 9a. When objects are ingested into GPU equipped node, the stream constructor initiates equal sized streams. Each stream is created one after the other with a fixed-size equal to the configured size of the partition. The stream constructor forwards them to a temporary fixed buffer area of host memory called the stream publisher. Upon data request by stream subscriber threads, VOs are then transferred to GPU memory. The SS threads always fetches data from the corresponding SP and this mapping is leveraged by the GPU driver.

Inter-Node Multi-Stream Re-Routing of VOs to Remote GPU Node. This case is triggered when a CCP node schedules chunking and fingerprinting operations on a remote GCP node, as depicted by Fig. 9b. There are two possible ways of inter-node data transfers. The default CUDA inter-node protocol follows a pipelined approach that first copies data between host memories of the two nodes (H2H) over the network, then once data is on the GPU equipped node, it is offloaded to GPU device memory via additional cudaMemcpy operations for accelerated deduplication processes. The second option, which we used for our implementation is for the CPU node to directly communicate with the remote node GPU via the network using RDMA transfers. Direct $\text{H2D}_{\text{remote}}$ and $\text{D2H}_{\text{remote}}$ reduce the additional latencies caused by proxy processes.

Inter-Node Multi-Stream Re-Routing of VOs to Remote CPU Node. This is when a node is not GPU-equipped, and all remote GPU nodes are overloaded and the LAFS scheduler instructs to use a remote CPU node. In this case, using the MPI-based communication API allows two CPU nodes to exchange VOs. In addition, if the RDMA network is available, it performs RDMA-based MPI communication.

4.5 Crocus Overhead

Recall that Crocus achieves maximum deduplication throughput by utilizing idle compute resources of either Local GPU or neighboring GPU/CPU nodes. However, offloading objects to other nodes in the cluster may incur additional network overheads thus increasing the I/O latency. However, executing deduplication with CPU nodes cause more performance bottleneck due to the limitations of CPUs described in Section 2.2, which restrains maximum deduplication throughput. In our proposed approach, we argue that although there is a small increase in latency per I/O, there is a significant throughput

TABLE 1
Experiment Testbed

Testbed	SSD Tier	HDD Tier
Testbed-I	3 Nodes, 2 SSDs Per Node, 1 GPU Per Node	3 Nodes, 2x1 TB ATA HDD Per Node
Testbed-II	7 Nodes (2 GPU Nodes, 5 CPU Nodes), 2 SSDs Per Node	5 Nodes, 2x1 TB ATA HDD Per Node

performance boost achieved from offloading deduplication tasks to faster GPU nodes and idle CPU node resources. We also amortize the network overheads through a batching operation that groups multiple objects into a single larger VO size that will be used as a unit for data transfer to neighboring nodes. We discuss the network performance trade-off in Section 5.4.

5 Evaluation

5.1 Experimental Setup

Our experimental setup consists of a Ceph Cluster with two tiers to depict a hybrid tiered storage model. The testbeds used for evaluation are shown in Table 1. In Testbed-I, all SSD tier nodes have equal resource configurations, whereas in Testbed-II only 2 nodes are equipped with GPU hardware while others are not GPU equipped to depict compute resource heterogeneity on the SSD tier. Each node runs Linux CentOS v7.3 and equipped with 10 cores Intel Xeon (R) CPU running at 2.40 GHz with a 32 GB DRAM. The nodes with GPU on SSD tier are provisioned with Nvidia Tesla K80, 24 GB GPU. On the storage side, each SSD tier node has two 256 GB Intel 850 PRO SSDs and HDD tier nodes have two 1 TB Hard disk drives, all connected via a 10 Gbps network. To flush the unique chunks from the SSD to the HDD tier, we set the cache-min-evict-age to 1800 ms, which evicts the chunks every 30 seconds and provided for the optimum flushing performance.

Workloads. We used both synthetic and realistic benchmarks to evaluate Crocus. For synthetic workloads, we used FIO benchmark [30] and for realistic workloads, we used SPEC SFS 2014 database benchmark [29]. For SPEC SFS workload, we mounted a block device on each client node and generated a total of 2 TB files per client node using a total of 5 clients. We then issued random write requests to the Ceph cluster. To fairly evaluate the effectiveness of LAFS in high contention scenarios, we run the SRAD application from the Rodinia Benchmark suite [31] to vary the load in GPU nodes. The LAFS scheduler in such cases is forced to redistribute the fingerprint requests between CPU and GPU pool nodes in an adaptive manner based on the available compute resources in the nodes. All the results in our evaluation show an end to end aggregate time or throughput for all deduplication operations, for the whole dataset and for all approaches. Our results show an average of three experimental runs. We compare the following systems:

- No-Dedup: Original Ceph with no deduplication. We use baseline and No-Dedup interchangeably in the rest of section.
- Dedup-CPU: Ceph with inline cluster-wide deduplication on SSD tier using CPU with single-thread implementation.
- Dedup-CPU(m): Ceph with inline cluster-wide deduplication on SSD tier using CPU invoking multi-threaded (16 Threads) hash computations.
- Dedup-GPU: Ceph with inline cluster-wide deduplication [11], on SSD tier using GPU with no optimizations.
- Crocus: Ceph with inline cluster-wide deduplication on SSD tier using GPU with intra/inter-node optimizations.

System Parameters. We set the object size to 4 MB, which is the default in Ceph and the VO size is determined by the automated VO resizing approach (in our case 20 MB). Note that the performance varies with chunk and VO size but not object size. Irrespective of object size, the chunking is conducted and VOs are the scheduling units. Smaller VO size will require more aggressive scheduling and communications. The maximum allowed VOs on GCP are 440 and 150 VOs for CCP. We extract these numbers after performing a set of experiments whilst monitoring congestion levels in GCP and CCP. We set the periodic broadcast time of the GCP and CCP pool vectors to every 5 seconds. We obtain this broadcast time after running the experiment shown in Section 5.3.1. We use same system parameter values for all experiments unless specified.

5.2 Crocus in Homogeneous Cluster Settings

To show the effectiveness of offloading deduplication operations such as chunking and fingerprinting, we first analyze the deduplication performance with CPU only nodes and later with all GPU-equipped nodes. We perform the experiments with SPEC SFS workload and instrument a trace for each deduplication sub-operation that measures the time of each sub-operation separately. Fig. 10 shows that the Dedup-CPU (DC) constitutes an average of 75 percent runtime only for deduplication operations, i.e., chunking, fingerprinting and fingerprint lookup (DB Ops). This huge overhead is mainly attributed to fingerprinting and it gets worse with small chunk size.

On the other hand, Dedup-GPU (DG) reduces the deduplication overhead by 86 percent on the total runtime for smaller chunk sizes such as 4 KB. It is due to offloading the fingerprint computations onto the GPU. Transferring data across the GPU PCI-e bus in the unit of a chunk causes a significant overhead when the chunk size becomes small. However, CROCUS design avoids this by transferring batches of data objects (VOs) and chunking initiates once the VOs land on the GPU global memory. Thus, the chunk size does not have an effect on the transfer time over the GPU PCI-e bus. However, the important thing to note here is that Dedup-GPU slightly increases the fingerprint runtime with increasing chunk size. It is because when the chunk size is small, it precisely fits in the fast GPU shared memory and hence the sliding window of the fingerprint kernel can quickly shift through the offset positions of smaller chunk boundaries. Whereas, for the large chunk size, the fingerprint kernel threads shift many times, fetching data between the small but fast GPU shared memory and the larger GPU global memory before shifting to another chunk boundary. This result shows that the use of GPU can greatly reduce the

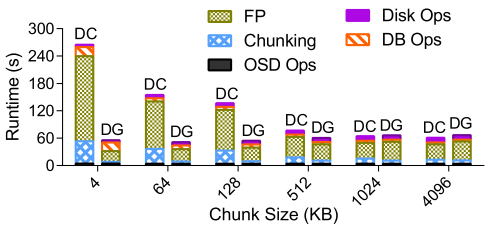


Fig. 10. Time breakdown analysis. Dedup-GPU (DG) versus Dedup-CPU (DC).

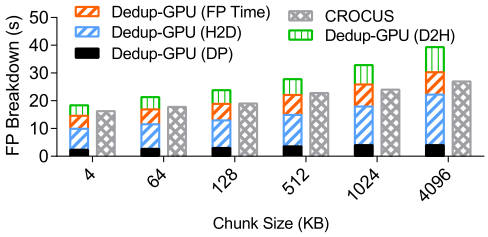


Fig. 11. Effects of multistream interleaving using per stream VO offloading.

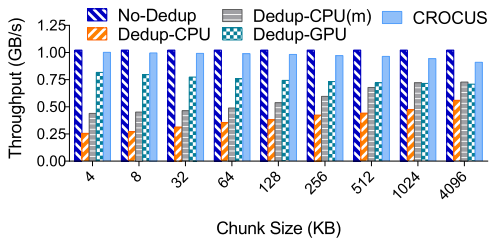


Fig. 12. Performance evaluation with Realistic SPEC SFS workload.

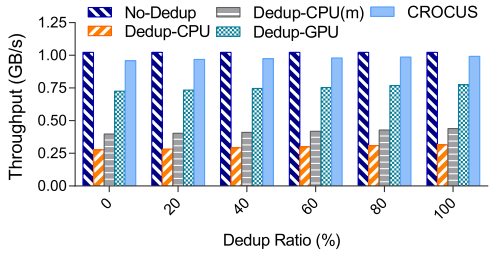


Fig. 13. Performance evaluation with synthetic FIO workload.

overheads of deduplication. However, in a heterogeneous setting it might not be the case as some SSD tier nodes will not be equipped with GPU. We evaluate how Crocus deals with such heterogeneity in Section 5.3.

To achieve the best deduplication performance with GPU in an all GPU SSD tier, we proposed some inter-node multi-stream VO offloading. In this section, we show the significance of our proposed intra-node optimizations with respect to varying chunk size. We compare the proposed intra-node optimizations (Crocus) against Dedup-GPU provisioned with no GPU optimizations. Fig. 11 compares the results of the optimizations versus non optimized GPU-Dedup. We trace the overall time for data preparation (DP), data transfers (Host-to-Device and Device-to-Host), and fingerprint kernel execution time in Dedup-GPU. However, the overall fingerprint computation time for Crocus consists of an overlap of data transfers and FP executions between the multiple stream submission channels, so we only collect the overall runtime. The results show that the multi-stream processing of Crocus improves the fingerprint runtime by 33 percent compared to Dedup-GPU. This improvement is attributed to (i) overlapping data transfers and fingerprint kernel executions and (ii) interleaving multiple fingerprint computations from different stream submission channels.

5.2.1 Crocus for Various Workload Characteristics

To evaluate Crocus with different workloads in a homogeneous SSD tier setup, we used synthetic and realistic workloads. We tried to mimic different deduplication attributes such as varying chunk size and duplication ratio. For realistic workloads, we employ SPEC SFS [29] and use FIO [30] to generate synthetic workload with controlled deduplication ratio. We compare Crocus with different variants of deduplication implementations. Dedup-CPU(m) in Figs. 12 and 13 employs multiple threads instead of a single thread. Fig. 12 shows the performance trend with varying chunk size using SPEC SFS workload. The results shows that both Dedup-CPU versions degrade performance specifically with small chunks. On the contrary, both Dedup-GPU implementations outperform the CPU implementations because of offloading compute-intensive operations on the powerful GPUs. With small 4KB chunk size, Dedup-GPU reduces the overall performance degradation from a high of 76 percent (in Dedup-CPU) to a significant low of 32 percent and Crocus further minimizes to the best low of 6 percent. Crocus performs better than Dedup-GPU due to the less synchronous communication between host and device, through the parallel transfers and fingerprint computations on different stream channels. However, we observe a minor performance degradation while traversing from smaller to bigger chunk sizes in GPU implementations. This is because, an increase in chunk size increases the number of shift cycles between the global and shared memory of the GPU.

Next, we show the effect of deduplication ratio using FIO generated workload in Fig. 13. For this experiment, we fix the chunk size to 32 KB (less than shared memory size of 48 KB on the Tesla K80 GPU) and vary the deduplication ratio. We observe that with increasing deduplication ratio from 0 to 100 percent, performance also slightly improves in all implementations, i.e., Dedup-CPU, Dedup-CPU(m), Dedup-GPU and Crocus. It is because the number of duplicate fingerprints increases with increasing dedup ratio, consequently, the number of I/O requests to the underlying storage reduces. However, the performance of Crocus still outperforms the Dedup-GPU, Dedup-CPU and Dedup-CPU (m) implementations. This improvement is attributed to proposed multi-stream communications in Crocus.

5.3 LAFS in Heterogeneous Cluster Settings

In this section, we perform a set of experiments to show the effectiveness of the proposed LAFS scheduler in Crocus. We use Testbed-II to incorporate resource heterogeneity on SSD tier nodes as shown in Table 1. To the best of our knowledge, no prior work has been done to explore deduplication scheduling orchestrations across the entire cluster. In this regard, we compare our dynamic LAFS of Crocus with a static scheduling approach. The static scheduling

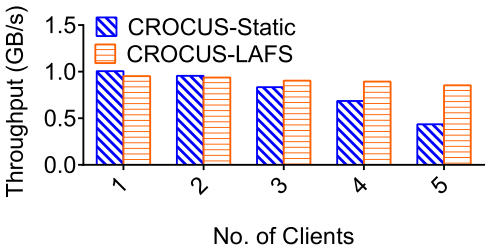


Fig. 14. Static versus dynamic scheduling.

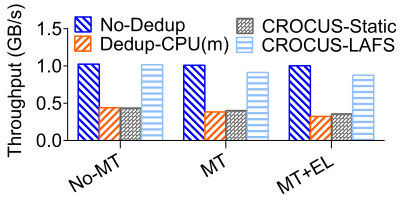


Fig. 15. Load-aware LAFS.

approach schedules all fingerprint operations to GPU nodes without the knowledge of their load or contention. It performs well under normal circumstances, but, degrades performance when the load on GPU nodes increases. However, the proposed dynamic LAFS scheduler adapts well to dynamically changing workloads. We perform the experiment with a single client and then, add more clients on each experimental run till we reach five clients whilst observing the impact of increasing load on LAFS scheduling. Fig. 14 shows a nearly equal performance for both LAFS and static scheduling with a few number of clients due to less I/O contention on GPU nodes. However, the LAFS scheduler significantly outperforms the static scheduler with increased number of clients. The reason of the low performance of static scheduling is the high number of scheduling operations leading to contention on GPUs. Whereas, the LAFS scheduler is aware of workload changes on GPU and can dynamically adjust the workload distribution between CPU and GPU nodes, thus outperforming the static scheduling approach.

For the next experiment, we mimic the realistic scenarios and evaluate the performance of LAFS scheduler by increasing the number of threads on each client node (total of five physical client nodes), to the storage cluster. Also as cloud environments hosts several VMs running multiple applications on a single node, some VM applications make heavy use of GPUs than others. To mimic this scenario, we further stress GPU nodes by generating additional load to GPUs via Rodinia SRAD Benchmark application [31], whilst monitoring the LAFS performance. Fig. 15 shows this experimental results. The No-MT stands for 5 client nodes with no multi-threading, MT stands for 5 client nodes each running 32 threads to continuously issue I/O requests, and MT+EL stands for 5 client nodes, each running 32 threads with external load (EL) generated via SRAD application. The results show that when there is no contention (No-MT), Crocus-LAFS performance is nearly at equal to No-Dedup. Whereas, Crocus performance drops sharply when integrated with static scheduling approach. This drop is worse off when the external load on GPU nodes increase as depicted in MT+EL case. However, the point here is that

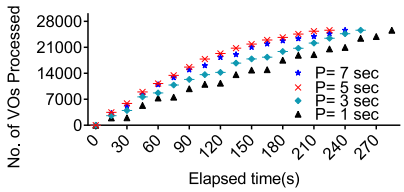


Fig. 16. Sensitivity test of vector updates.

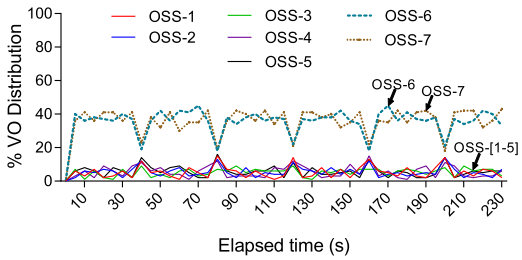
Crocus-LAFS shows a nearly consistent performance for all cases, even in cases of high contention on GPU nodes by distributing the workload on CPU and GPU nodes proportionally to available resource bandwidths.

5.3.1 Sensitivity Test and VO Distribution Pattern

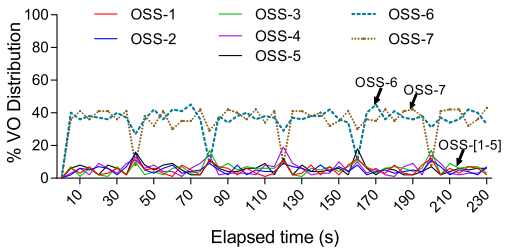
The LAFS scheduler highly depends on utilization ratios of compute resources to schedule or offload fingerprint operations on SSD tier nodes. Thus, the utilization vector update interval is very critical. We perform a sensitivity test to analyze how this update interval impacts the number of VOs processed per unit time. We performed this test under Testbed-II setting with SPEC SFS workload.

CROCUS design tries as much as possible to provide an up-to-date GCP pool and CCP pool vector information for the purpose of accurate scheduling processes. However, the frequency of the update intervals is an important factor in the scheduling performance. As depicted in Fig. 16, which shows the sensitivity analysis with respect to varying the utilization vector update interval, lower vector update period (1 second - 3 seconds), results in lower scheduling performance as VOs scheduled per second are fewer. This is mainly caused by the excessive internal communication interference overheads. We clearly observe that the optimal number of VOs processed is achieved at periodic update time equal to 5 seconds. Finally, we observe that increasing the periodic update time to higher than 5 seconds results in a decreased number of VOs scheduled per second. This result shows that choosing the periodic update time value is vital as far as scheduling performance in CROCUS is concerned. We believe that the utilization vector update period highly depends on the workload pattern.

One of the most important functionalities of CROCUS is to identify free slots with minimal overhead. Crocus achieves this through the use of the readily available up-to-date GCP and CCP vectors. An evaluation of how the system changes the scheduling distribution with varying workload patterns on each node is therefore important. Fig. 17 captures this important distribution phenomenon. In order to conduct a detailed analysis of the VO distribution pattern of Crocus-LAFS in different contention cases, we create two scenarios. Fig. 17a shows the case when all GPU equipped nodes are equally congested, whereas, Fig. 17b depicts a scenario where GPU nodes are overloaded one at a time after every 40 seconds, in a round-robin manner. The OSS-1 to OSS-7 are CPU computation pool nodes whereas, OSS-6 and OSS-7 both have GPUs hence, they are part of GPU computation pool as well. We create contention by running SRAD application [31] for 5 seconds after every 40 seconds interval. Figs. 17a & 17b both show the highly balanced VO distribution pattern,



(a) Overloading both OSS-6 and 7 GPU nodes



(b) Overloading OSS-6 and OSS-7 in Round-robin fashion

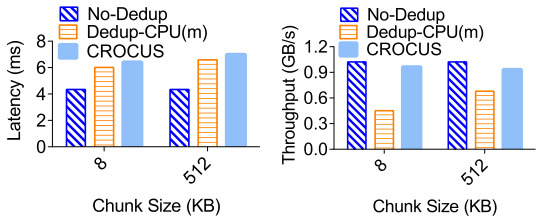
Fig. 17. Virtual object distribution pattern analysis with varying loads on Testbed-II using SPEC SFS workload.

thereby avoiding additional contention on already contented nodes. And when required, LAFS also distributes VOs to CPU nodes as shown in Fig. 17b. In particular, in Fig. 17a, we observe OSS-[6-7] shows the same trend of VO distribution, however, in Fig. 17b, we see OSS-7 drops at 40 seconds, but OSS-6 does not drop because OSS-7 is only overloaded in this experiment.

5.4 Network Overhead Analysis

In this section, we analyze the impact on performance caused by offloading tasks onto neighboring nodes. The two major overheads of CROCUS on top of the existing works are (i) memory overhead caused by buffering incoming objects before they are scheduled for deduplication processes, and (ii) network overhead, which is a result of offloading VOs to neighbor nodes. Both overheads add extra I/O time in the deduplication process. To evaluate the effects of such overheads on CROCUS perfomance, we use Testbed-II of Table 1 with the SPEC SFS database workload.

Fig. 18 shows the results of this experiment. First of all, the results in Fig. 18a, shows that enabling the multi-threaded CPU based deduplication, Dedup(m), incurs additional latency per I/O as compared to the baseline with no deduplication. This is due to the additional deduplication I/Os introduced on the I/O path. We further observe that latency is already dominated by software overhead such as fingerprinting on the baseline. The introduction of high speed networks such as Cisco Nexus technology [32], which offers cluster network speeds of up to 100 Gbps, does not eliminate this performance overhead. We also observe our approach slightly increases the latency while improving the performance overhead of the baseline in Fig. 18. However, the huge trade-off between slightly degraded I/O latency and high throughput performance shows that the network overhead of CROCUS is



(a) Latency (b) Throughput

Fig. 18. An Evaluation of CROCUS-LAFS latency versus bandwidth on Testbed-II.

insignificant as compared to the throughput gain that comes with the CROCUS implementation.

6 Related Works

Many works have been done to employ cluster-wide deduplication in distributed storage systems [11], [19], [20], [21], [23], [24], [26], [33], [34], [35]. However, many of them do not adhere to our target shared nothing architecture and none of them investigated the possibility of utilizing neighbor compute resources to accelerate the overheads of deduplication. Our work differs from the prior works in the sense that the proposed framework orchestrates the use of all cluster compute resources (CPU/GPU) and also adheres to the shared nothing design architecture. Furthermore, CROCUS does the offloading of dedup computations in a load-aware fashion, by first taking advantage of powerful GPU accelerators (if available), then distributes the fingerprinting load on the rest of the CPU resources if GPU is overloaded.

Recently, software based data deduplication products such as Intel's ISA-L [36], [37] has been introduced in the storage industry to accelerate the CPU based data reduction performance. However, CPU based data reduction methods in large scale data clusters requires an inhibitive amount of CPUs to achieve high data reduction throughput. Previous works [27], proved that, saturating a high end Suwon201622-core CPU with optimized data reduction processes using high performance Intel's ISA-L hashing, can only result in very low local data reduction throughput. Worse off, this data reduction throughput can further decrease when applied on a cluster-wide scale.

Recently, hardware accelerators such as GPUs and FPGAs have been adopted to mitigate the overheads from compute-intensive components of storage such as compression, encryption and hashing [18], [27], [38]. StoreGPU [18], introduces a general framework for hashing to provide enhanced parallelism to storage systems. However, it follows the pipeline approach for host/device communications, in contrary to the concept of stream overlapping, thus inhibiting maximum deduplication performance. Shredder [38] uses GPU for chunking and hashing in incremental deduplication-enabled storage. It improves the performance by using a stream based approach, which enables to overlap multiple data transfers and hashing computations. Shredder [38] has a limited deduplication performance improvement, because its orchestration is not cluster-wide.

One of the recent works for accelerating data deduplication in storage is CIDR [27] and [39], which uses FPGA

hardware accelerators to speed up the compute-intensive data reduction operations (deduplication + compression). However, our work differs from [27] and [39] due to their centralized metadata management which does not conform to our targeted shared-nothing storage architecture. Another drawback of [27] and [39] is that they perform localized deduplication on a single SSD array node, which does not achieve maximum deduplication opportunities across the cluster. However, Crocus achieves this deduplication improvement by not only utilizing hardware accelerators but also considers the software based CPU acceleration with all possible compute resources across the storage cluster.

Several works such as [40], [41], [42], [43], [44] developed optimal resource scheduling in VM placement and cloud environments. However, the majority of resource scheduling for VM placement is controlled centrally, which violates the design principles of our target architecture. In [45], [46], all servers are assumed to have the same computational capabilities, whereas, our target architecture considers resource heterogeneity in the hybrid cluster environment. In [47], a novel resource provisioning approach was studied for shared nothing storage systems, using queuing network for analysis of both service performance and cost estimation of storage nodes. However, it uses a separate set of dedicated metadata servers and storage servers. In our case, we minimize the storage costs by embedding metadata management in each SSD tier node. Each SSD tier node handles the scheduling functionality. Kubernetes [43] and Slurm [44], dynamically construct mortal containers, that are temporarily created to allocate resources that runs a specific set of application workloads. As soon as the application completes its execution, these containers are terminated. This creates some problems when this approach is used to orchestrate resources in deduplication related frameworks. i) The creation of containerized resources whenever deduplication related processes are launched creates additional bottleneck. ii) Furthermore, when these containers are created on neighboring nodes, the batching on local node plus the creation of the containers at the neighbor nodes will add more overheads, thus degrading the deduplication performance.

However, none of the existing studies target dynamic and optimal resource scheduling in cluster-scale inline data deduplication. We propose to build an inline deduplication framework using compute resource orchestrations, which not only speed-up the system performance but also fully utilize the compute resources.

7 Conclusion

This paper presents Crocus, a high performance cluster-wide deduplication approach for multi-tier shared-nothing storage systems. Crocus improves the inline deduplication performance at the SSD tier with small chunks while achieving high disk space savings. Specifically, our compute-resource orchestrations exploits the available GPU resources on the SSD tier to accelerate the compute-intensive deduplication operations such as chunking and fingerprinting. Furthermore, considering the GPU heterogeneity on the SSD tier and fluctuating load on each SSD node, we design an opportunistic load-aware fingerprint scheduling algorithm, LAFS, which dynamically distributes the deduplication operations among GPU and CPU nodes. To maximize the throughput of GPU nodes, we proposed optimizations for both intra- and inter-node communications using the multi-stream based communication model. We implemented Crocus in Ceph. The experimental results confirm the effectiveness of Crocus and LAFS in both homogeneous and heterogeneous cluster configurations.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) Grant funded by the Korea Government (Ministry of Science and ICT) under Grant NRF-2018R1A1A1A05079398.

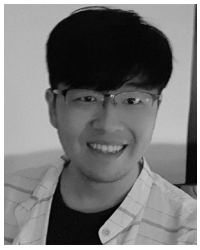
References

[1] T. Economist, "The data deluge: Five years on," 2016. [Online]. Available: https://www.slideshare.net/economistintelligenceunit/the-data-deluge-five-years-on
[2] H. Zou, Y. Yu, W. Tang, and H. M. Chen, "Improving I/O performance with adaptive data compression for big data applications," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2014, pp. 1228–1237.
[3] 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.
[4] Gluster, 2019. [Online]. Available: http://www.gluster.org
[5] 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.
[6] W. Xia *et al.*,"A comprehensive study of the past, present, and future of data deduplication," *Proc. IEEE*, vol. 104, no. 9, pp. 1681–1710, Sep. 2016.
[7] R. Kaur, I. Chana, and J. Bhattacharya, "Data deduplication techniques for efficient cloud storage management: A systematic review," *J. Supercomput.*, vol. 74, no. 5, pp. 2035–2085, May 2018.
[8] J. Paulo and J. Pereira, "A survey and classification of storage deduplication systems," *ACM Comput. Surv.*, vol. 47, no. 1, pp. 11:1–11:30, Jun. 2014.
[9] S. Lee, S. Cho, H. Kim, and Y. Won, "Performance analysis of SSD/HDD hybrid storage manager," in *Proc. 16th North-East Asia Symp. Nano Inf. Technol. Rel.*, 2011, pp. 136–139.
[10] J. Guerra, H. Pucha, J. Glider, W. Belluomini, and R. Rangaswami, "Cost effective storage using extent based dynamic tiering," in *Proc. 9th USENIX Conf. File Stroage Technol.*, 2011, pp. 20–20.
[11] A. Khan, C. Lee, P. Hamandawana, S. Park, and Y. Kim, "A robust fault-tolerant and scalable cluster-wide deduplication for shared-nothing storage systems," in *Proc. Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2018, pp. 1–7.
[12] M. Oh *et al.*, "Design of global data deduplication for a scale-out distributed storage system," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 1063–1073.
[13] R. Zhu, L.-H. Qin, J.-L. Zhou, and H. Zheng, "Using multi-threads to hide deduplication I/O latency with low synchronization overhead," *J. Central South Univ.*, vol. 20, no. 6, pp. 1582–1591, 2013.
[14] W. Xia *et al.*, "FastCDC: A fast and efficient content-defined chunking approach for data deduplication," in *Proc. USENIX Annu. Tech. Conf.*, 2016, pp. 101–114.
[15] D.-Y. Lee, K. Jeong, S.-H. Han, J.-S. Kim, J.-Y. Hwang, and S. Cho, "Understanding write behaviors of storage backends in ceph object store," in *Proc. IEEE Int. Conf. Massive Storage Syst. Technol.*, vol. 10, 2017.
[16] Forbes, "GPUs in the cloud, the largest providers are investing heavily today," 2017. [Online]. Available: https://www.forbes.com/sites/benjaminboxer/2017/04/12/gpus-in-the-cloud-the-largest-providers-are-investing-heavily-today
[17] K. Jang, S. Han, S. Han, S. B. Moon, and K. Park, "SSLShader: Cheap SSL acceleration with commodity processors," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 1–14.

[18] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu, "StoreGPU: Exploiting graphics processing units to accelerate distributed storage systems," in *Proc. 17th Int. Symp. High Perform. Distrib. Comput.*, 2008, pp. 165–174.
[19] C. Dubnicki *et al.*, "HYDRAstor: A scalable secondary storage," in *Proc. 7th USENIX Conf. File Storage Technol.*, 2009, pp. 197–210.
[20] Y. Fu, H. Jiang, and N. Xiao, "A scalable inline cluster deduplication framework for big data protection," in *Proc. 13th Int. Middleware Conf.*, 2012, pp. 354–373.
[21] D. Bhagwat, K. Eshghi, D. D. Long, and M. Lillibridge, "Extreme binning: Scalable, parallel deduplication for chunk-based file backup," in *Proc. IEEE Int. Symp. Conf. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2009, pp. 1–9.
[22] D. Frey, A.-M. Kermarrec, and K. Kloudas, "Probabilistic deduplication for cluster-based storage systems," in *Proc. 3rd ACM Symp. Cloud Comput.*, 2012, pp. 1–17.
[23] A. T. Clements *et al.*, "Decentralized deduplication in SAN cluster file systems," in *Proc. USENIX Annu. Tech. Conf.*, 2009, pp. 101–114.
[24] J. Kaiser, D. Meister, A. Brinkmann, and S. Effert, "Design of an exact data deduplication cluster," in *Proc. 28th IEEE Symp. Mass Storage Syst. Technol.*, 2012, pp. 1–12.
[25] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proc. 6th USENIX Conf. File Storage Technol.*, 2008, pp. 269–282.
[26] W. Xia, H. Jiang, D. Feng, and Y. Hua, "SiLo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput," in *Proc. USENIX Annu. Tech. Conf.*, 2011, pp. 26–30.
[27] M. Ajdari, P. Park, J. Kim, D. Kwon, and J. Kim, "CIDR: A cost-effective in-line data reduction system for terabit-per-second scale SSD arrays," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2019, pp. 28–41.
[28] Z. Han *et al.*, "DEC: An efficient deduplication-enhanced compression approach," in *Proc. IEEE 22nd Int. Conf. Parallel Distrib. Syst.*, 2016, pp. 519–526.
[29] SPEC, "SPEC SFS 2014," 2017. [Online]. Available: https://www.spec.org/sfs2014/
[30] Jens Axboe, "Flexible I/O Tester," 2017. [Online]. Available: https://github.com/axboe/fio
[31] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization*, 2009, pp. 44–54.
[32] Cisco, "Cisco nexus 7700 F3-series 12-Port 100 Gigabit ethernet module data sheet," 2019. [Online]. Available: https://www.cisco.com/c/en/us/products/collateral/switches/nexus-7000-series-switches/data_sheet_c78-728423.html
[33] C. Lin, Q. Cao, J. Huang, J. Yao, X. Li, and C. Xie, "HPDV: A highly parallel deduplication cluster for virtual machine images," in *Proc. 18th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2018, pp. 472–481.
[34] W. Xia, D. Feng, H. Jiang, Y. Zhang, V. I. Chang, and X. Zou, "Accelerating content-defined-chunking based data deduplication by exploiting parallelism," *Future Gener. Comput. Syst.*, vol. 98, pp. 406–418, 2019.
[35] J. Ma and C. Park, "Parallelizing inline data reduction operations for primary storage systems," in *Proc. Parallel Comput. Technol.*, 2017, pp. 301–307.
[36] Intel, "Reposity of Intel storage acceleration library," 2017. [Online]. Available: https://https://github.com/01org/isa-l
[37] Intel®, "Intel® storage acceleration library (open source version)," 2019. [Online]. Available: https://01.org/intel%C2%AE-storage-acceleration-library-open-source-version
[38] P. Bhatotia, R. Rodrigues, and A. Verma, "Shredder: GPU-accelerated incremental storage and computation," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, pp. 1–14.
[39] M. Ajdari, P. Park, D. Kwon, J. Kim, and J. Kim, "A scalable HW-based inline deduplication for SSD arrays," *IEEE Comput. Archit. Lett.*, vol. 17, no. 1, pp. 47–50, Jan. 2018.
[40] M. Mechtri, M. Hadji, and D. Zeghlache, "Exact and heuristic resource mapping algorithms for distributed and hybrid clouds," *IEEE Trans. Cloud Comput.*, vol. 5, no. 4, pp. 681–696, Oct.-Dec. 2017.
[41] B. Palanisamy, A. Singh, and L. Liu, "Cost-effective resource provisioning for MapReduce in a cloud," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 5, pp. 1265–1279, May 2015.
[42] Q. Zhang, M. F. Zhani, R. Boutaba, and J. L. Hellerstein, "Dynamic heterogeneity-aware resource provisioning in the cloud," *IEEE Trans. Cloud Comput.*, vol. 2, no. 1, pp. 14–28, Jan.-Mar. 2014.
[43] Kubernetes, "Production-grade container orchestration," 2019. [Online]. Available: https://kubernetes.io/
[44] SchedMD, "Slurm worload manager," 2019. [Online]. Available: https://slurm.schedmd.com/documentation.html
[45] J. Bi, Z. Zhu, R. Tian, and Q. Wang, "Dynamic provisioning modeling for virtualized multi-tier applications in cloud data center," in *Proc. IEEE 3rd Int. Conf. Cloud Comput.*, 2010, pp. 370–377.
[46] X. Zhou and P. Lama, "Efficient server provisioning with control for end-to-end response time guarantee on multitier clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 1, pp. 78–86, Jan. 2012.
[47] J. Zhou, J. Fan, and J. Jia, "A cost-efficient resource provisioning algorithm for DHT-based cloud storage systems," *Concurrency Comput. Pract. Experience*, vol. 28, no. 18, pp. 4485–4506, 2016.

Prince Hamandawana received the BSc honors degree in computer science from the National University of Science and Technology (NUST), Bulawayo, Zimbabwe. He is working toward the MS leading to PhD integrated program degree at Ajou University, Suwon, South Korea. He had the privilege to work for Econet Wireless, 2008-2011, and Liquid Telecom, 2011-2016. Currently, he is a member of the Database and Dependable Computing (DBDC) Lab, Ajou University with the Department of Computer Engineering. His research interests include distributed and parallel storage systems, and GPU assisted cluster-wide data deduplication.

Awais Khan (Member, IEEE) received the BS degree in bioinformatics from Mohammad Ali Jinnah University, Islamabad, Pakistan. He is working toward the MS leading to PhD integrated program degree with the Department of Computer Science and Engineering, Sogang University, Seoul, South Korea. He worked in Digital Research Laboratories as software engineer from 2012 to 2015. Currently, he is a member in Laboratory for Advanced System Software in the Department of Computer Science and Engineering, Sogang University, Seoul, South Korea. His research interests include cloud computing, cluster-scale deduplication, parallel and distributed file systems.



Chang-Gyu Lee received the BS degree in computer science from Ajou University, Suwon, South Korea. He is working toward the MS leading to PhD integrated program degree in the Department of Computer Science and Engineering, Sogang University, Seoul, South Korea. Currently, he is a member in Laboratory for Advanced System Software with the Department of Computer Science and Engineering, Sogang University, Seoul, South Korea. His research interests include operating systems, file and storage systems, key-value stores, and parallel and distributed systems.

Sungyong Park received the BS degree in computer science from Sogang University, Seoul, South Korea, and both the MS and PhD degrees in computer science from Syracuse University, Syracuse, New York. He is currently a professor with the Department of Computer Science and Engineering, Sogang University, Seoul, South Korea. From 1987 to 1992, he worked for LG Electronics, South Korea, as a research engineer. From 1998 to 1999, he was a research scientist at Telcordia Technologies (formerly Bellcore), where he developed network management software for optical switches. His research interests include cloud computing and systems, virtualization technologies, high performance I/O and storage systems, and embedded system software.

Youngjae Kim (Member, IEEE) received the BS degree in computer science from Sogang University, South Korea, in 2001, the MS degree in computer science from KAIST, in 2003, and the PhD degree in computer science and engineering from Pennsylvania State University, University Park, Pennsylvania, in 2009. He is currently an associate professor with the Department of Computer Science and Engineering, Sogang University, Seoul, South Korea. Before joining Sogang University, Seoul, South Korea, he was a R&D staff scientist with the US Department of Energy's Oak Ridge National Laboratory (2009-2015) and as an assistant professor at Ajou University, Suwon, South Korea (2015-2016). His research interests include operating systems, file and storage systems, parallel and distributed systems, computer systems security, and performance evaluation.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Accelerating Federated Learning via Momentum Gradient Descent

Wei Liu, Li Chen, Yunfei Chen, *Senior Member, IEEE*, and Wenyi Zhang, *Senior Member, IEEE*

Abstract—Federated learning (FL) provides a communication-efficient approach to solve machine learning problems concerning distributed data, without sending raw data to a central server. However, existing works on FL only utilize first-order gradient descent (GD) and do not consider the preceding iterations to gradient update which can potentially accelerate convergence. In this article, we consider momentum term which relates to the last iteration. The proposed momentum federated learning (MFL) uses momentum gradient descent (MGD) in the local update step of FL system. We establish global convergence properties of MFL and derive an upper bound on MFL convergence rate. Comparing the upper bounds on MFL and FL convergence rates, we provide conditions in which MFL accelerates the convergence. For different machine learning models, the convergence performance of MFL is evaluated based on experiments with MNIST and CIFAR-10 datasets. Simulation results confirm that MFL is globally convergent and further reveal significant convergence improvement over FL.

Index Terms—Accelerating convergence, distributed machine learning, federated learning, momentum gradient descent

1 Introduction

Recently, data-intensive machine learning has been applied in various fields, such as autonomous driving [1], speech recognition [2], image classification [3] and disease detection [4] since this technique provides beneficial solutions to extract the useful information hidden in data. It now becomes a common tendency that machine-learning systems are deployed in architectures that include tens of thousands of processors [5]. Great amount of data is generated by various parallel and distributed physical objects.

Collecting data from edge devices to the central server is necessary for distributed machine learning scenarios. In the process of distributed data collection, there exist significant challenges such as energy efficiency problems and system latency problems. The energy efficiency of distributed data collection was considered in wireless sensor networks (WSNs) due to limited battery capacity of sensors [6]; In fifth-generation (5G) cellular networks, a round-trip delay from terminals through the network back to terminals demands much lower latencies, potentially down to 1 ms, to facilitate human tactile to visual feedback control [7]. Thus, the challenges of data aggregation in distributed system urgently require communication-efficient solutions.

- *W. Liu, L. Chen, and W. Zhang are with the Department of Electronic Engineering and Information Science, University of Science and Technology of China, Hefei, Anhui 230052, China. E-mail: liuwei93@mail.ustc.edu.cn, {chenli87, wenyizha}@ustc.edu.cn.*
- *Y. Chen is with the School of Engineering, University of Warwick, CV4 7AL Coventry, United Kingdom. E-mail: Yunfei.Chen@warwick.ac.uk.*

Manuscript received 6 Oct. 2019; revised 14 Jan. 2020; accepted 15 Feb. 2020. Date of publication 19 Feb. 2020; date of current version 23 Mar. 2020. (Corresponding author: Li Chen.) Recommended for acceptance by J. Zhai. Digital Object Identifier no. 10.1109/TPDS.2020.2975189

In order to overcome these challenges, cutting down transmission distance and reducing the amount of uploaded data from edge devices to the network center are two effective ways. To reduce transmission distance, mobile edge computing (MEC) in [8] is an emerging technique where the computation and storage resources are pushed to proximity of edge devices where the local task and data offloaded by users can be processed. In this way, the distance of large-scale data transmission is greatly shortened and the latency has a significant reduction [9]. Using machine learning for the prediction of uploaded task execution time achieves a shorter processing delay [10], and dynamic resource scheduling was studied to optimize resources allocation of MEC system in [11]. To reduce the uploaded data size, model-based compression approaches, where raw data are compressed and represented by well-established model parameters, demonstrate significant compression performance [12]. Lossy compression is also an effective strategy to decrease the uploaded data size [13], [14]. Compressed sensing, where the sparse data of the edge can be efficiently sampled and reconstructed with transmitting a much smaller data size, was applied to data acquisition of Internet of Things (IoT) network [15]. All the aforementioned works need to collect raw data from individual device.

To avoid collecting raw data for machine learning in distributed scenarios, a novel approach named *Federated Learning* (FL) has emerged as a promising solution [16]. The work in [17] provided a fundamental architecture design of FL. Considering the growing computation capability of edge nodes (devices), FL decentralizes the centralized machine learning task and assigns the decomposed computing tasks to the edge nodes where the raw data are stored and learned at the edge nodes. After a fixed iteration interval, each edge node transmits its learned model parameters to the central

1045-9219 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

server. This strategy can substantially decrease consumption of communication resources and improve communication efficiency. To further improve the energy efficiency of FL, an adaptive FL approach was proposed in [17], where the aggregation frequency can be adjusted adaptively to minimize the loss function under a fixed resource budget. To reduce the uplink communication costs, the work in [18] proposed structured and sketched updates method, and compression techniques were adopted to reduce parameter dimension in this work. In [19], gradient selection and adaptive adjustment of learning rate were used for efficient compression. For security aggregation of high-dimensional data, the work in [20] provided a communication-efficient approach, where the server can compute the sum of model parameters from edge nodes without knowing the contribution of each individual node. In [21], under unbalanced resource distribution in network edge, FL with client (edge node) selection was proposed for actively managing the clients aggregation according to their resources condition. In [22], non-i.i.d data distribution was studied.

However, existing FL solutions generally use gradient descent (GD) for loss function minimization. GD is a one-step method where the next iteration depends only on the current gradient. Convergence rate of GD can be improved by accounting for more preceding iterations [23]. Thus, by introducing the last iteration, which is named the momentum term, momentum gradient descent (MGD) can accelerate the convergence [24], [25]. Due to the improved convergence of gradient methods brought by momentum, there are several works which apply stochastic gradient descent (SGD) with momentum in the field of distributed machine learning. In [26], momentum is applied to the update at each aggregation round for improving both optimization and generalization. In [27], the linear convergence of distributed SGD with momentum is proven. All these works with momentum are based on stochastic GD generally. Compared with SGD, deterministic gradient descent (DGD) can realize more precise training results with improved generalization and fast convergence under convex optimization [28].

Motivated by the above observations, we propose a new federated learning design of *Momentum Federated Learning* (MFL) in this paper. In the proposed MFL design, we introduce momentum term in FL local update and leverage MGD (in our paper, MGD means DGD with momentum) to perform local iterations. Furthermore, the global convergence of the proposed MFL is proven, and we derive the theoretical convergence bound of MFL. Compared with FL [17], the proposed MFL has an accelerated convergence rate under certain conditions. On the basis of MNIST and CIFAR-10 datasets, we numerically study the proposed MFL and obtain its loss function curve. The experimental results show that MFL converges faster than FL for different machine learning models. The contributions of this paper are summarized as follows:

- *MFL design*: According to the characteristic that MGD facilitates machine learning convergence in the centralized situation, we propose MFL design where MGD is adopted to optimize loss function in local updates. The proposed MFL can improve the convergence rate of distributed learning problem significantly.

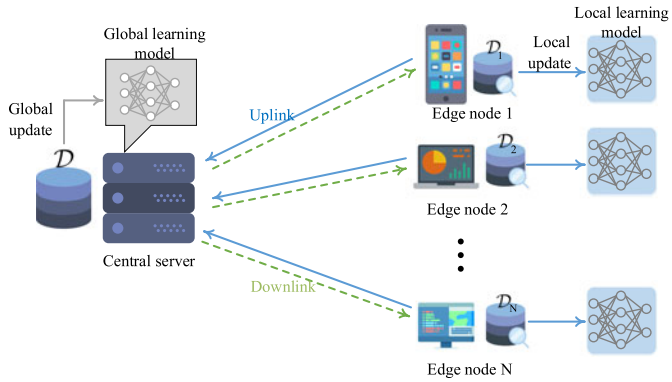


Fig. 1. The simplified structure of learning system for distributed user data.

- *Convergence analysis for MFL*: We prove that the proposed MFL is globally convergent on convex optimization problems, and derive its theoretical upper bound on convergence rate. We conduct a comparative analysis of convergence performance between the proposed MFL and FL. It is proven that MFL improves convergence rate of FL under certain conditions.
- *Evaluation based on MNIST and CIFAR-10 datasets*: We evaluate the proposed MFL's convergence performance via simulation based on MNIST and CIFAR-10 datasets with different machine learning models such as support vector machine (SVM), linear regression, logistic regression and convolutional neural network (CNN). Then an experimental comparison is made between FL and the proposed MFL. The simulation results show that MFL is convergent and confirm that MFL provides a significant improvement of convergence rate.

The remaining part of this paper is organized as follows. We introduce the system model to solve the learning problem in distributed scenarios in Section 2 and subsequently elaborate the existing solutions in Section 3. In Section 4, we describe the design of MFL in detail. Then in Sections 5 and 6, we present the convergence analysis of MFL and the comparison between FL and MFL, respectively. Finally, we show experimental results in Section 7 and draw a conclusion in Section 8.

2 System Model

In this paper, considering a simplified system model, we discuss the distributed network as shown in Fig. 1. This model has N edge nodes and a central server. These N edge nodes, which have limited communication and computation resources, contain local datasets $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_i, \ldots, \mathcal{D}_N$, respectively. So the global dataset is $\mathcal{D} \triangleq \mathcal{D}_1 \cup \mathcal{D}_2 \cup \cdots \cup \mathcal{D}_N$. Assume that $\mathcal{D}_i \cap \mathcal{D}_j = \emptyset$ for $i \neq j$. We define the number of samples in node i as $|\mathcal{D}_i|$ where $|\cdot|$ denotes the size of the set. The total number of all nodes' samples is $|\mathcal{D}|$, and $|\mathcal{D}| = \sum_{i=1}^{N} |\mathcal{D}_i|$. The central server connects all the edge nodes for information transmission.

We define *the global loss function* at the central server as $F(\mathbf{w})$, where $\mathbf{w}$ denotes the model parameter. Different machine learning models correspond to different $F(\cdot)$ and $\mathbf{w}$. We use $\mathbf{w}^*$ to represent the optimal parameter for minimizing the value of $F(\mathbf{w})$. Based on the presented model,

the learning problem is to minimize $F(\mathbf{w})$ and it can be formulated as follows:

$$\mathbf{w}^* \triangleq \arg\min F(\mathbf{w}). \tag{1}$$

Because of the complexity of machine learning model and original dataset, finding a closed-form solution of the above optimization problem is usually intractable. So algorithms based on gradient iterations are used to solve (1). If raw user data are collected and stored in the central server, we can use centralized learning solutions to (1) while if raw user data are distributed over the edge nodes, FL and the proposed MFL can be applied to solve this learning problem.

Under the situation where FL or MFL solutions are used, the local loss function of node i is denoted by $F_i(\mathbf{w})$ which is defined merely on $\mathcal{D}_i$. Then we define the global loss function $F(\mathbf{w})$ on $\mathcal{D}$ as follows:

Definition 1 (Global loss function). *Given the loss function $F_i(\mathbf{w})$ of edge node i, we define the global loss function on all the distributed datasets as*

$$F(\mathbf{w}) \triangleq \frac{\sum_{i=1}^{N} |\mathcal{D}_i| F_i(\mathbf{w})}{|\mathcal{D}|}. \tag{2}$$

3 Existing Solutions

In this section, we introduce two existing solutions to solve the learning problem expressed by (1). These two solutions are centralized learning solution and FL solution, respectively.

3.1 Centralized Learning Solution

Centralized machine learning is for machine learning model embedded in the central server and each edge node needs to send its raw data to the central sever. In this situation, edge nodes will consume communication resources for data transmission, but without incurring computation resources consumption.

After the central server has collected all datasets from the edge nodes, a usual way to solve the learning problem expressed by (1) is GD as a basic gradient method. Further, MGD is an improved gradient method with adding a momentum term to speed up learning process [24].

3.1.1 GD

The update rule for GD is as follows:

$$\mathbf{w}(t) = \mathbf{w}(t-1) - \eta \nabla F(\mathbf{w}(t-1)). \tag{3}$$

In (3), t denotes the iteration index and $\eta > 0$ is the learning step size. The model parameter $\mathbf{w}$ is updated along the direction of negative gradient. Using the above update rule, GD can solve the learning problem with continuous iterations.

3.1.2 MGD

As an improvement of GD, MGD introduces the momentum term and we present its update rules as follows:

$$\mathbf{d}(t) = \gamma \mathbf{d}(t-1) + \nabla F(\mathbf{w}(t-1)) \tag{4}$$

$$\mathbf{w}(t) = \mathbf{w}(t-1) - \eta \mathbf{d}(t), \tag{5}$$

where $\mathbf{d}(t)$ is the momentum term which has the same dimension as $\mathbf{w}(t)$, γ is the momentum attenuation factor, η is the learning step size and t is the iteration index. By iterations of (4) and (5) with t, $F(\mathbf{w})$ can potentially converge to the minimum faster compared with GD. The convergence range of MGD is $-1 < \gamma < 1$ with a bounded η and if $0 < \gamma < 1$, MGD has an accelerated convergence rate than GD under a small η typically used in simulations [29, Result 3].

3.2 FL Solution

In contrast with centralized learning solutions, FL avoids collecting and uploading the distributed data because of the limited communication resources at edge nodes and privacy protection for local data. It decouples the machine learning task from the central server to each edge node to avoid storing user data in the server and reduce the communication resources consumption. All of edge nodes make up a federation in coordination with the central server.

The FL design and convergence analysis are presented in [17] where FL network is studied thoroughly. In an FL system, each edge node uses the same machine learning model. We use τ to denote the global aggregation frequency, i.e., the update interval. Each node i has its local model parameter $\widetilde{\mathbf{w}}_i(t)$, where the iteration index is denoted by $t = 0, 1, 2, \ldots$ (in this paper, an iteration means a local update). We use $[k]$ to denote the aggregation interval $[(k-1)\tau, k\tau]$ for $k = 1, 2, 3, \ldots$. At $t = 0$, local model parameters of all nodes are initialized to the same value. When $t > 0$, $\widetilde{\mathbf{w}}_i(t)$ is updated locally based on GD, which is the *local update*. After τ local updates, *global aggregation* is performed and all edge nodes send the updated model parameters to the centralized server synchronously.

The learning process of FL is described as follows.

3.2.1 Local Update

When $t \in [k]$, local updates are performed in each edge node by

$$\widetilde{\mathbf{w}}_i(t) = \widetilde{\mathbf{w}}_i(t-1) - \eta \nabla F_i(\widetilde{\mathbf{w}}_i(t-1)),$$

which follows GD exactly.

3.2.2 Global Aggregation

When $t = k\tau$, global aggregation is performed. Each node sends $\widetilde{\mathbf{w}}_i(k\tau)$ to the central server synchronously. The central server takes a weighted average of the received parameters from N nodes to obtain the globally updated parameter $\mathbf{w}(k\tau)$ by

$$\mathbf{w}(k\tau) = \frac{\sum_{i=1}^{N} |\mathcal{D}_i| \widetilde{\mathbf{w}}_i(k\tau)}{|\mathcal{D}|}.$$

Then $\mathbf{w}(k\tau)$ is sent back to all edge nodes as their new parameters and edge nodes perform local updates for the next iteration interval.

In [17, Lemma 2], the FL solution has been proven to be globally convergent for convex optimization problems and exhibit good convergence performance. So FL is an effective solution to the distributed learning problem presented in (1).

TABLE 1
MFL Notation Summary

Notation	Definition
T; K; N	number of total local iterations; number of global aggregations/number of intervals; number of edge nodes
t; k; τ; $[k]$	iteration index; interval index; aggregation frequency with $\tau = T/K$; the interval $[(k-1)\tau, k\tau]$
$\mathbf{w}^*$; $\mathbf{w}^{\mathrm{f}}$	globally optimal parameter of $F(\cdot)$; the optimal parameter that MFL can obtain in Algorithm 1
η; β; ρ; γ	the learning step size of MGD or GD; the β-smooth parameter of $F_i(\cdot)$; the ρ-Lipschitz parameter of $F_i(\cdot)$; the momentum attenuation factor which decides the proportion of momentum term in MGD
$\mathcal{D}_i$; $\mathcal{D}$	the local dataset of node i; the global dataset
δ_i; δ	the upper bound between $\nabla F(\mathbf{w})$ and $\nabla F_i(\mathbf{w})$; the average of δ_i over all nodes
$F_i(\cdot)$; $F(\cdot)$	the loss function of node i; the global loss function
$\mathbf{d}(t)$; $\mathbf{w}(t)$	the global momentum parameter at iteration round t; the global model parameter at iteration round t
$\widetilde{\mathbf{d}}_i(t)$; $\widetilde{\mathbf{w}}_i(t)$	the local momentum parameter of node i at iteration round t; the local model parameter at iteration round t
$\mathbf{d}_{[k]}(t)$; $\mathbf{w}_{[k]}(t)$	the momentum parameter of centralized MGD at iteration round t in $[k]$; the model parameter of centralized MGD at iteration round t in $[k]$
$\theta_{[k]}(t)$; θ; p	the angle between vector $\nabla F(\mathbf{w}_{[k]}(t))$ and $\mathbf{d}_{[k]}(t)$; θ is the maximum of $\theta_{[k]}(t)$ for $1 \le k \le K$ with $t \in [k]$; p is the maximum ratio of $\|\mathbf{d}_{[k]}(t)\|$ and $\|\nabla F(\mathbf{w}_{[k]}(t))\|$ for $1 \le k \le K$ with $t \in [k]$

4 Design of MFL

In this section, we introduce the design of MFL to solve the distributed learning problem shown in (1). We first discuss the motivation of our work. Then we present the design of MFL in detail and the learning problem based on federated system. The main notations of MFL design and analysis are summarized in Table 1.

4.1 Motivation

Since MGD improves the convergence rate of GD [24], we want to apply MGD to local update steps of FL and hope that the proposed MFL will accelerate the convergence rate for federated networks.

First, we illustrate the intuitive influence on optimization problem after introducing the momentum term into gradient updating methods. Considering GD, the update reduction of the parameter is $\eta\nabla F(\mathbf{w}(t-1))$ which is only proportional to the gradient of $\mathbf{w}(t-1)$. The update direction of GD is always along gradient descent so that an oscillating update path could be caused, as shown by the GD update path in Fig. 2. However, the update reduction of parameter for MGD is a superposition of $\eta\nabla F(\mathbf{w}(t-1))$ and $\gamma(\mathbf{w}(t-2)-\mathbf{w}(t-1))$ which is the momentum term. As shown by the MGD update path in Fig. 2, utilizing the momentum term can deviate the direction of parameter update to the optimal decline significantly and mitigate the oscillation caused by GD. In Fig. 2, GD has an oscillating update path and costs seven iterations to reach the optimal point while MGD only needs three iterations to do that, which demonstrates that mitigating the oscillation by MGD leads to a faster convergence rate.

Because edge nodes of distributed networks are usually resource-constrained, solutions to convergence acceleration can attain higher resources utilization efficiency. Thus, motivated by the property that MGD improves convergence rate, we use MGD to perform local updates of FL and this approach is named MFL.

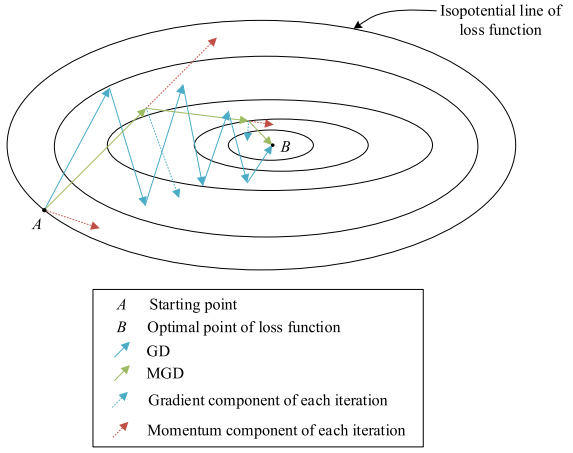


Fig. 2. Comparison of MGD and GD.

In the following subsection, we design the MFL learning paradigm and propose the learning problem based on the MFL design.

4.2 MFL

In the MFL design, we use $\widetilde{\mathbf{d}}_i(t)$ and $\widetilde{\mathbf{w}}_i(t)$ to denote momentum parameter and model parameter for node i, respectively. All edge nodes are set to embed the same machine learning model. So the local loss functions $F_i(\mathbf{w})$ are the same for all nodes, and the dimensions of both the model parameters and the momentum parameters are consistent. The parameters setup of MFL is similar to that of FL. We use t to denote the local iteration index for $t = 0, 1, \ldots, \tau$ to denote the aggregation frequency and $[k]$ to denote the interval $[(k-1)\tau, k\tau]$ where k denotes the interval index for $k = 1, 2, \ldots$. At $t = 0$, the momentum parameters and the model parameters of all nodes are initialized to the same values, respectively. When $t \in [k]$, $\widetilde{\mathbf{d}}_i(t)$ and $\widetilde{\mathbf{w}}_i(t)$ are updated based on MGD, called *local update steps*. When $t = k\tau$, MFL performs *global aggregation steps* where $\widetilde{\mathbf{d}}_i(t)$ and $\widetilde{\mathbf{w}}_i(t)$ are sent to the central server synchronously. Then in the central server, the global momentum parameter $\mathbf{d}(t)$ and the global model parameter $\mathbf{w}(t)$ are obtained by taking a weighted average of the received parameters, respectively, and are sent back to all edge nodes for the next interval.

The learning rules of MFL include the local update and the global aggregation steps. By continuous alternations of local update and global aggregation, MFL can perform its learning process to minimize the global loss function $F(\mathbf{w})$. We describe the MFL learning process as follows.

First of all, we set initial values for $\widetilde{\mathbf{d}}_i(0)$ and $\widetilde{\mathbf{w}}_i(0)$. Then

1) Local Update: When $t \in [k]$, local update is performed at each edge node by

$$\widetilde{\mathbf{d}}_i(t) = \gamma\widetilde{\mathbf{d}}_i(t-1) + \nabla F_i(\widetilde{\mathbf{w}}_i(t-1)) \tag{6}$$

$$\widetilde{\mathbf{w}}_i(t) = \widetilde{\mathbf{w}}_i(t-1) - \eta\widetilde{\mathbf{d}}_i(t). \tag{7}$$

According to (6) and (7), node i performs MGD to optimize the loss function $F_i(\cdot)$ defined on its own dataset.

2) Global Aggregation: When $t = k\tau$, node i transmits $\widetilde{\mathbf{w}}_i(k\tau)$ and $\widetilde{\mathbf{d}}_i(k\tau)$ to the central server which takes weighted

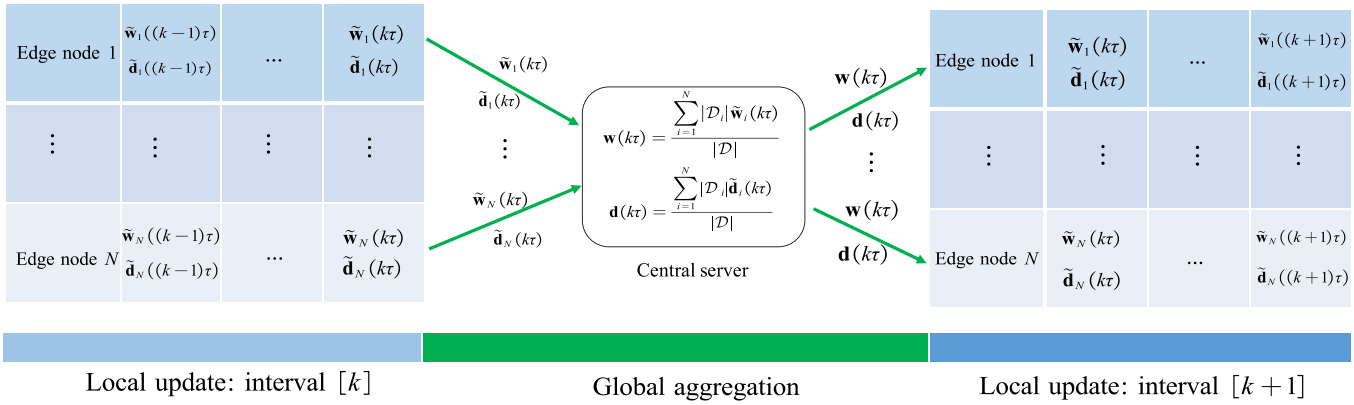


Fig. 3. Illustration of MFL local update and global aggregation steps from interval $[k]$ to $[k+1]$.

averages of the received parameters from N nodes to obtain the global parameters $\mathbf{w}(k\tau)$ and $\mathbf{d}(k\tau)$, respectively. The aggregation rules are presented as follows:

$$\mathbf{d}(t) = \frac{\sum_{i=1}^{N} |\mathcal{D}_i| \widetilde{\mathbf{d}}_i(t)}{|\mathcal{D}|} \tag{8}$$

$$\mathbf{w}(t) = \frac{\sum_{i=1}^{N} |\mathcal{D}_i| \widetilde{\mathbf{w}}_i(t)}{|\mathcal{D}|}. \tag{9}$$

Then the central server sends $\mathbf{d}(k\tau)$ and $\mathbf{w}(k\tau)$ back to all edge nodes where $\widetilde{\mathbf{d}}_i(k\tau) = \mathbf{d}(k\tau)$ and $\widetilde{\mathbf{w}}_i(k\tau) = \mathbf{w}(k\tau)$ are set to enable the local update in the next interval $[k+1]$. Note that only if $t = k\tau$, the value of the global parameters $\mathbf{w}(t)$ and $\mathbf{d}(t)$ can be observed. But we define $\mathbf{d}(t)$ and $\mathbf{w}(t)$ for all t to facilitate the following analysis. A typical alternation is shown in Fig. 3 which illustrates the learning steps of MFL in interval $[k]$ and $[k+1]$.

Algorithm 1. *MFL* The Dataset in Each Node Has Been Set, and the Machine Learning Model Embedded in Edge Nodes has Been Chosen. We Have Set Appropriate Model Parameters η and γ.

Input:
The limited number of local updates in each node T
A given aggregation frequency τ
Output:
The final global model weight vector $\mathbf{w}^{\mathrm{f}}$
1: Set the initial values of $\mathbf{w}^{\mathrm{f}}$, $\widetilde{\mathbf{w}}_i(0)$ and $\widetilde{\mathbf{d}}_i(0)$.
2: **for** $t = 1, 2, \ldots, T$ **do**
3: Each node i performs local update in parallel according to (6) and (7).//*Local update*
4: **if** $t == k\tau$ where k is a positive integer **then**
5: Set $\widetilde{\mathbf{d}}_i(t) \leftarrow \mathbf{d}(t)$ and $\widetilde{\mathbf{w}}_i(t) \leftarrow \mathbf{w}(t)$ for all nodes where $\mathbf{d}(t)$ and $\mathbf{w}(t)$ are obtained by (8) and (9) respectively. //*Global aggregation*
Update $\mathbf{w}^{\mathrm{f}} \leftarrow \arg\min_{\mathbf{w} \in \{\mathbf{w}^{\mathrm{f}}, \mathbf{w}(k\tau)\}} F(\mathbf{w})$
6: **end if**
7: **end for**

The learning problem of MFL to attain the optimal model parameter is presented as (1). However, the edge nodes have limited computation resources with a finite number of local iterations. We assume that T is the number of local updates and K is the corresponding number of global aggregations. Thus, we have $t \leq T$ and $k \leq K$ with $T = K\tau$. Considering that $\mathbf{w}(t)$ is unobservable for $t \neq k\tau$, we use $\mathbf{w}^{\mathrm{f}}$ to denote the achievable optimal model parameter defined on resource-constrained MFL network. Hence, the learning problem is to obtain $\mathbf{w}^{\mathrm{f}}$ within K global aggregations particularly, i.e.,

$$\mathbf{w}^{\mathrm{f}} \triangleq \underset{\mathbf{w} \in \{\mathbf{w}(k\tau): k=1,2,\ldots,K\}}{\arg\min} F(\mathbf{w}). \tag{10}$$

The optimization algorithm of MFL is explained in Algorithm 1.

5 Convergence Analysis

In this section, we first introduce some definitions and assumptions for MFL convergence analysis. Then based on these preliminaries, global convergence properties of MFL following Algorithm 1 are established and an upper bound on MFL convergence rate is derived. Also MFL convergence performance with related parameters is analyzed.

5.1 Preliminaries

First of all, to facilitate the analysis, we assume that $F_i(\mathbf{w})$ satisfies the following conditions:

Assumption 1. *For $F_i(\mathbf{w})$ in node i, we assume the following conditions:*

1) $F_i(\mathbf{w})$ *is convex*
2) $F_i(\mathbf{w})$ *is ρ-Lipschitz, i.e.,* $|F_i(\mathbf{w}_1) - F_i(\mathbf{w}_2)| \leq \rho \|\mathbf{w}_1 - \mathbf{w}_2\|$ *for some* $\rho > 0$ *and any* $\mathbf{w}_1, \mathbf{w}_2$
3) $F_i(\mathbf{w})$ *is β-smooth, i.e.,* $\|\nabla F_i(\mathbf{w}_1) - \nabla F_i(\mathbf{w}_2)\| \leq \beta \|\mathbf{w}_1 - \mathbf{w}_2\|$ *for some* $\beta > 0$ *and any* $\mathbf{w}_1, \mathbf{w}_2$
4) $F_i(\mathbf{w})$ *is μ-strong, i.e.,* $aF_i(\mathbf{w}_1) + (1-a)F_i(\mathbf{w}_2) \geq F_i(a\mathbf{w}_1 + (1-a)\mathbf{w}_2) + \frac{a(1-a)\mu}{2}\|\mathbf{w}_1 - \mathbf{w}_2\|^2$, $a \in [0,1]$ *for some* $\mu > 0$ *and any* $\mathbf{w}_1, \mathbf{w}_2$ *[30, Theorem 2.1.9]*

Because guaranteeing the global convergence of centralized MGD requires that the objective function is strongly convex [24], it is necessary to assume condition 4. Assumption 1 is satisfied for some learning models such as SVM, linear regression and logistic regression whose loss functions are presented in Table 2. Experimental results as presented in Section 7.2.1 show that for non-convex models such as CNN whose loss function does not satisfy Assumption 1, MFL also performs well. From Assumption 1, we can obtain the following lemma:

TABLE 2
Loss Function of Three Machine Learning Models

Model	Loss function
SVM	$\frac{\lambda}{2}\|\mathbf{w}\|^2 + \frac{1}{2\|\mathcal{D}_i\|}\sum_j \max\{0; 1 - y_j\mathbf{w}^{\mathrm{T}}\mathbf{x}_j\}$
Linear regression	$\frac{1}{2\|\mathcal{D}_i\|}\sum_j \|y_j - \mathbf{w}^{\mathrm{T}}\mathbf{x}_j\|^2$
Logistic regression	$-\frac{1}{\|\mathcal{D}_i\|}\sum_j \|y_j \log\sigma(\mathbf{w},\mathbf{x}_j) + (1-y_j)\log(1-\sigma(\mathbf{w},\mathbf{x}_j))\|$ where $\sigma(\mathbf{w},\mathbf{x}_j)$ is given as (23)

Lemma 1. *$F(\mathbf{w})$ is convex, ρ-Lipschitz, β-smooth and μ-strong.*

Proof. According to the definition of $F(\mathbf{w})$ from (2), triangle inequality and the definition of ρ-Lipschitz, β-smooth and μ-strong, we can derive that $F(\mathbf{w})$ is convex, ρ-Lipschitz, β-smooth and μ-strong directly. □

Then we introduce the gradient divergence between $\nabla F(\mathbf{w})$ and $\nabla F_i(\mathbf{w})$ for any node i. It comes from the nature of the difference in datasets distribution.

Definition 2 (Gradient divergence). *We define δ_i as the upper bound between $\nabla F(\mathbf{w})$ and $\nabla F_i(\mathbf{w})$ for any node i, i.e.,*

$$\|\nabla F(\mathbf{w}) - \nabla F_i(\mathbf{w})\| \leq \delta_i. \tag{11}$$

Also, we define the average gradient divergence

$$\delta \triangleq \frac{\sum_i |\mathcal{D}_i|\delta_i}{|\mathcal{D}|}. \tag{12}$$

Boundedness of δ_i and δ. Based on condition 3 of Assumption 1, we let $\mathbf{w}_2 = \mathbf{w}_i^*$ where $\mathbf{w}_i^*$ is the optimal value for minimizing $F_i(\mathbf{w})$. Because $F_i(\mathbf{w})$ is convex, we have $\|\nabla F_i(\mathbf{w}_1)\| \leq \beta\|\mathbf{w}_1 - \mathbf{w}_i^*\|$ for any $\mathbf{w}_1$, which means $\|\nabla F_i(\mathbf{w})\|$ is finite for any $\mathbf{w}$. According to Definition 1 and the linearity of gradient operator, global loss function $\nabla F(\mathbf{w})$ is obtained by taking a weighted average of $\nabla F_i(\mathbf{w})$. Therefore, $\|\nabla F(\mathbf{w})\|$ is finite, and $\|\nabla F(\mathbf{w}) - \nabla F_i(\mathbf{w})\|$ has an upper bound, i.e., δ_i is bounded. Further, δ is still bounded from the linearity in (12).

Since local update steps of MFL perform MGD, the upper bounds of MFL and MGD convergence rates exhibit certain connections in the same interval. For the convenience of analysis, we use variables $\mathbf{d}_{[k]}(t)$ and $\mathbf{w}_{[k]}(t)$ to denote the momentum parameter and the model parameter of *centralized MGD* in each interval $[k]$, respectively. This centralized MGD is defined on global dataset and updated based on global loss function $F(\mathbf{w})$. In interval $[k]$, the update rules of centralized MGD follow:

$$\mathbf{d}_{[k]}(t) = \gamma\mathbf{d}_{[k]}(t-1) + \nabla F(\mathbf{w}_{[k]}(t-1)) \tag{13}$$

$$\mathbf{w}_{[k]}(t) = \mathbf{w}_{[k]}(t-1) - \eta\mathbf{d}_{[k]}(t). \tag{14}$$

At the beginning of interval $[k]$, the momentum parameter $\mathbf{d}_{[k]}(t)$ and the model parameter $\mathbf{w}_{[k]}(t)$ of centralized MGD are synchronized with the corresponding parameters of MFL, i.e.,

$$\mathbf{d}_{[k]}((k-1)\tau) \triangleq \mathbf{d}((k-1)\tau)$$
$$\mathbf{w}_{[k]}((k-1)\tau) \triangleq \mathbf{w}((k-1)\tau).$$

For each interval $[k]$, the centralized MGD is performed by iterations of (13) and (14). In Fig. 4, we illustrate the distinctions between $F(\mathbf{w}(t))$ and $F(\mathbf{w}_{[k]}(t))$ intuitively.

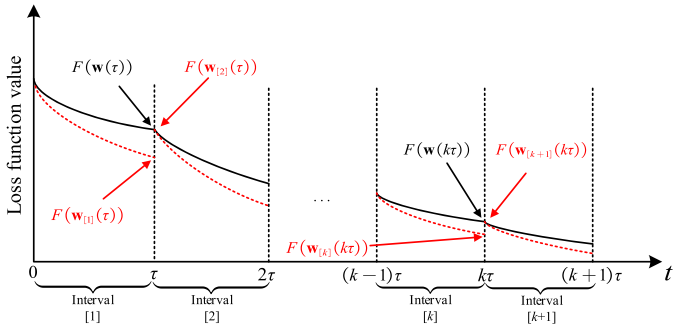


Fig. 4. Illustration of the difference between MGD and MFL in intervals.

Compared with centralized MGD, MFL aggregation interval with $\tau > 1$ brings global update delay because of the fact that centralized MGD performs global update on every iteration while MFL is allowed to spread its global parameter to edge nodes after τ local updates. Therefore, the convergence performance of MFL is worse than that of MGD, which is essentially from the imbalance between several computation rounds and one communication round in MFL design. The following subsection provides the resulting convergence performance gap between these two approaches.

5.2 Gap between MFL and Centralized MGD in Interval $[k]$

First, considering a special case, we consider the gap between MFL and centralized MGD for $\tau = 1$. From intuitive perspective, MFL performs global aggregation after every local update and there does not exist global parameter update delay, i.e, the performance gap is zero. In Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/TPDS.2020.2975189., we prove that MFL is equivalent to MGD for $\tau = 1$ theoretically.

Now considering the general case for any $\tau \geq 1$, the upper bound of gap between $\mathbf{w}(t)$ and $\mathbf{w}_{[k]}(t)$ can be derived as follows.

Proposition 1 (Gap between MFL and centralized MGD in intervals). *Given $t \in [k]$, the gap between $\mathbf{w}(t)$ and $\mathbf{w}_{[k]}(t)$ can be expressed by*

$$\|\mathbf{w}(t) - \mathbf{w}_{[k]}(t)\| \leq h(t - (k-1)\tau), \tag{15}$$

where we define

$$A \triangleq \frac{(1+\gamma+\eta\beta) + \sqrt{(1+\gamma+\eta\beta)^2 - 4\gamma}}{2\gamma},$$
$$B \triangleq \frac{(1+\gamma+\eta\beta) - \sqrt{(1+\gamma+\eta\beta)^2 - 4\gamma}}{2\gamma},$$

$$E \triangleq \frac{A}{(A-B)(\gamma A - 1)},$$
$$F \triangleq \frac{B}{(A-B)(1-\gamma B)}$$

and $h(x)$ yields

$$h(x)=\eta\delta\left[E(\gamma A)^x+F(\gamma B)^x-\frac{1}{\eta\beta}-\frac{\gamma(\gamma^x-1)-(\gamma-1)x}{(\gamma-1)^2}\right] \tag{16}$$

for $0<\gamma<1$ and any $x=0,1,2,\ldots$.

Because $F(\mathbf{w})$ is ρ-Lipschitz from Lemma 1, it holds that

$$F(\mathbf{w}(t))-F(\mathbf{w}_{[k]}(t))\leq\rho h(t-(k-1)\tau). \tag{17}$$

Proof. First, we derive an upper bound of $\|\widetilde{\mathbf{w}}_i(t)-\mathbf{w}_{[k]}(t)\|$ for node i. On the basis of this bound, we extend this result from the local cases to the global one to obtain the final result. The detailed proving process is presented in Appendix B, available in the online supplemental material. □

Because $h(1)=h(0)=0$ and $h(x)$ increases with x for $x\geq 1$, which are proven in Appendix C, available in the online supplemental material, we always have $h(x)\geq 0$ for $x=0,1,2,\ldots$.

From Proposition 1, in any interval $[k]$, we have $h(0)=0$ for $t=(k-1)\tau$, which fits the definition $\mathbf{w}_{[k]}((k-1)\tau)=\mathbf{w}((k-1)\tau)$. We still have $h(1)=0$ for $t=(k-1)\tau+1$. This means that there is no gap between MFL and centralized MGD when local update is only performed once after the global aggregation.

It is easy to find that if $\tau=1$, $t-(k-1)\tau$ is either 0 or 1. Because $h(1)=h(0)=0$, the upper bound in (15) is zero, and there is no gap between $F(\mathbf{w}(t))$ and $F(\mathbf{w}_{[k]}(t))$ from (17). This is consistent with Appendix A, available in the online supplemental material, where MFL yields centralized MGD for $\tau=1$. In any interval $[k]$, we have $t-(k-1)\tau\in[0,\tau]$. If $\tau>1$, $t-(k-1)\tau$ can be larger than 1. When $x>1$, we know that $h(x)$ increases with x. According to the definition of A, B, E and F, we can obtain $\gamma A>1$, $\gamma B<1$ and $E,F>0$ easily. Because $0<\gamma<1$, the last term will linearly decrease with x when x is large. Therefore, the first exponential term $E(\gamma A)^x$ in (16) will be dominant when x is large and the gap between $\mathbf{w}(t)$ and $\mathbf{w}_{[k]}(t)$ increases exponentially with t.

Also we find $h(x)$ is proportional to the average gradient gap δ. It is because the greater the local gradient divergences at different nodes are, the larger the gap will be. So considering the extreme situation where all nodes have the same data samples ($\delta=0$ because the local loss functions are the same), the gap between $\mathbf{w}(t)$ and $\mathbf{w}_{[k]}(t)$ is zero and MFL is equivalent to centralized MGD.

5.3 Global Convergence

We have derived an upper bound between $F(\mathbf{w}(t))$ and $F(\mathbf{w}_{[k]}(t))$ for $t\in[k]$. According to the definition of MFL, in the beginning of each interval $[k]$, we set $\mathbf{d}_{[k]}((k-1)\tau)=\mathbf{d}((k-1)\tau)$ and $\mathbf{w}_{[k]}((k-1)\tau)=\mathbf{w}((k-1)\tau)$. The global upper bound on the convergence rate of MFL can be derived based on Proposition 1.

The following definitions are made to facilitate analysis. First, we use $\theta_{[k]}(t)$ to denote the angle between vectors $\nabla F(\mathbf{w}_{[k]}(t)$ and $\mathbf{d}_{[k]}(t)$ for $t\in[k]$, i.e.,

$$\cos\theta_{[k]}(t)\triangleq\frac{\nabla F(\mathbf{w}_{[k]}(t))^{\mathrm{T}}\mathbf{d}_{[k]}(t)}{\|\nabla F(\mathbf{w}_{[k]}(t)\|\|\mathbf{d}_{[k]}(t)\|},$$

and θ is defined as the maximum value of $\theta_{[k]}(t)$ for $1\leq k\leq K$ with $t\in[k]$, i.e.,

$$\theta\triangleq\max_{1\leq k\leq K,t\in[k]}\theta_{[k]}(t).$$

Then we define

$$p\triangleq\max_{1\leq k\leq K,t\in[k]}\frac{\|\mathbf{d}_{[k]}(t)\|}{\|\nabla F(\mathbf{w}_{[k]}(t))\|},$$

and

$$\omega\triangleq\min_k\frac{1}{\|\mathbf{w}((k-1)\tau)-\mathbf{w}^*\|^2}.$$

Based on Proposition 1 which gives an upper bound of loss function difference between MFL and centralized MGD, global convergence rate of MFL can be derived as follows.

Lemma 2. *If the following conditions are satisfied:*

1) $\cos\theta\geq 0$, $0<\eta\beta<1$ *and* $0\leq\gamma<1$; *There exists $\varepsilon>0$ which makes*
2) $F(\mathbf{w}_{[k]}(k\tau))-F(\mathbf{w}^*)\geq\varepsilon$ *for all k;*
3) $F(\mathbf{w}(T))-F(\mathbf{w}^*)\geq\varepsilon$;
4) $\omega\alpha-\frac{\rho h(\tau)}{\tau\varepsilon^2}>0$ *hold,*

then we have

$$F(\mathbf{w}(T))-F(\mathbf{w}^*)\leq\frac{1}{T\left(\omega\alpha-\frac{\rho h(\tau)}{\tau\varepsilon^2}\right)}, \tag{18}$$

where we defined

$$\alpha\triangleq\eta\left(1-\frac{\beta\eta}{2}\right)+\eta\gamma(1-\beta\eta)\cos\theta-\frac{\beta\eta^2\gamma^2p^2}{2}.$$

Proof. The proof is presented in Appendix D, available in the online supplemental material. □

On the basis of Lemma 2, we further derive the following proposition which demonstrates the global convergence of MFL and gives its upper bound on convergence rate.

Proposition 2 (MFL global convergence). *Given* $\cos\theta\geq 0$, $0<\eta\beta<1$, $0\leq\gamma<1$ *and* $\alpha>0$, *we have*

$$F(\mathbf{w}^{\mathrm{f}})-F(\mathbf{w}^*)\leq\frac{1}{2T\omega\alpha}+\sqrt{\frac{1}{4T^2\omega^2\alpha^2}+\frac{\rho h(\tau)}{\omega\alpha\tau}}+\rho h(\tau). \tag{19}$$

Proof. The specific proving process is shown in Appendix E, available in the online supplemental material. □

According to the above Proposition 2, we get an upper bound of $F(\mathbf{w}^{\mathrm{f}})-F(\mathbf{w}^*)$ which is a function of T and τ. From inequality (19), we can find that MFL linearly converges to a lower bound $\sqrt{\frac{\rho h(\tau)}{\omega\alpha\tau}}+\rho h(\tau)$. Because $h(\tau)$ is related to τ and δ, aggregation intervals ($\tau>1$) and different data distribution collectively lead to that MFL does not converge to the optimum.

In the following, we discuss the influence of τ on the convergence bound. If $\tau = 1$, we have $\rho h(\tau) = 0$ so that $F(\mathbf{w}^{\mathrm{f}}) - F(\mathbf{w}^*)$ linearly converges to zero as $T \to \infty$, and the convergence rate yields $\frac{1}{T\omega\alpha}$. Noting $h(\tau) > 0$ if $\tau > 1$, we can find that in this case, $F(\mathbf{w}^{\mathrm{f}}) - F(\mathbf{w}^*)$ converges to a non-zero bound $\sqrt{\frac{\rho h(\tau)}{\omega\alpha\tau}} + \rho h(\tau)$ as $T \to \infty$. On the one hand, if there does not exist communication resources limit, setting aggregation frequency $\tau = 1$ and performing global aggregation after each local update can reach the optimal convergence performance of MFL. On the other hand, aggregation interval ($\tau > 1$) can let MFL effectively utilize the communication resources of each node, but bring about a decline of convergence performance.

6 Comparison Between FL and MFL

In this section, we make a comparison of convergence performance between MFL and FL.

A closed-form solution of the upper bound on FL convergence rate has been derived in [17, Theorem 2]. It is presented as follows.

$$F(\mathbf{w}_{FL}^{\mathrm{f}}) - F(\mathbf{w}^*) \leq \frac{1}{2\eta\varphi T} + \sqrt{\frac{1}{4\eta^2\varphi^2 T^2} + \frac{\rho h_{FL}(\tau)}{\eta\varphi\tau}} + \rho h_{FL}(\tau). \tag{20}$$

According to [17],

$$h_{FL}(\tau) = \frac{\delta}{\beta}((\eta\beta + 1)^{\tau} - 1) - \eta\delta\tau,$$

and $\varphi = \omega_{FL}(1 - \frac{\eta\beta}{2})$ where the expression of ω_{FL} is consistent with ω. Differing from that of ω, $\mathbf{w}((k-1)\tau)$ in the definition of ω_{FL} is the global model parameter of FL.

We assume that both MFL and FL solutions are applied in the system model proposed in Fig. 1. They are trained based on the same training dataset with the same machine learning model. The loss functions $F_i(\cdot)$ and global loss functions $F(\cdot)$ of MFL and FL are the same, respectively. The corresponding parameters of MFL and FL are equal including τ, η, ρ, δ and β. We set the same initial value $\mathbf{w}(0)$ of MFL and FL. Because both MFL and FL are convergent, we have $\omega = \frac{1}{\|\mathbf{w}(0) - \mathbf{w}^*\|^2}$. Then according to the definitions of ω and ω_{FL}, we have $w = w_{FL}$. Therefore, the corresponding parameters of MFL and FL are the same and we can compare the convergence rates between FL and MFL conveniently.

For convenience, we use $f_1(T)$ and $f_2(T)$ to denote the upper bounds on convergence rates of MFL and FL, respectively. Then we have

$$f_1(T) \triangleq \frac{1}{2T\omega\alpha} + \sqrt{\frac{1}{4T^2\omega^2\alpha^2} + \frac{\rho h(\tau)}{\omega\alpha\tau}} + \rho h(\tau) \tag{21}$$

and

$$f_2(T) \triangleq \frac{1}{2\eta\varphi T} + \sqrt{\frac{1}{4\eta^2\varphi^2 T^2} + \frac{\rho h_{FL}(\tau)}{\eta\varphi\tau}} + \rho h_{FL}(\tau). \tag{22}$$

We consider the special case of $\gamma \to 0$. For $\omega\alpha$ and $\eta\varphi$, we can obtain $\omega\alpha \to \omega\eta(1 - \frac{\beta\eta}{2}) = \eta\varphi$ from the definition of α. Then for $h(\tau)$ and $h_{FL}(\tau)$, we have $\gamma A \to \eta\beta + 1$ and $\gamma B \to 0$. Because $\frac{A}{A-B} \to 1$ and $\frac{B}{A-B} \to 0$, we can further get $E \to \frac{1}{\eta\beta}$ and $F \to 0$ from the definitions of E and F. So, according to (16), we have

$$\begin{aligned}\lim_{\gamma\to 0} h(\tau) &= \eta\delta\left[\frac{1}{\eta\beta}(\eta\beta + 1)^{\tau} - \frac{1}{\eta\beta} - \tau\right] \\ &= \frac{\delta}{\beta}((1 + \eta\beta)^{\tau} - 1) - \eta\delta\tau = h_{FL}(\tau).\end{aligned}$$

Hence, by the above analysis under $\gamma \to 0$, we can find that MFL and FL have the same upper bound on convergence rate. This fact is consistent with the property that if $\gamma = 0$, MFL degenerates into FL and has the same convergence rate with FL.

To avoid complicated calculations over the expressions of $f_1(T)$ and $f_2(T)$, we have the following lemma.

Lemma 3. *If there exists $T_1 \geq 1$ which satisfies that $\frac{1}{2T\omega\alpha}$ dominates in $f_1(T)$ and $\frac{1}{2\eta\varphi T}$ dominates in $f_2(T)$ for $T < T_1$, i.e.,*

$$\frac{1}{2T\omega\alpha} \gg \max\left\{\rho h(\tau), \sqrt{\frac{\rho h(\tau)}{\omega\alpha\tau}}\right\}$$

and

$$\frac{1}{2\eta\varphi T} \gg \max\left\{\rho h_{FL}(\tau), \sqrt{\frac{\rho h_{FL}(\tau)}{\eta\varphi\tau}}\right\},$$

then we have

$$f_1(T) \approx \frac{1}{T\omega\alpha}$$

and

$$f_2(T) \approx \frac{1}{T\eta\varphi}$$

for $T < T_1$.

Proof. We can find such a T_1. For example, considering (21), if $\eta \to 0$, we have $\alpha \to 0$ from the definition of α and $h(\tau) \to 0$ from Appendix F, available in the online supplemental material. So we can easily derive $\omega\alpha\rho h(\tau) \to 0$ and $\sqrt{\frac{\omega\alpha\rho h(\tau)}{\tau}} \to 0$. Then we can find $T_1 \geq 1$ which satisfies $\frac{1}{2T} \gg \omega\alpha\rho h(\tau)$ and $\frac{1}{2T} \gg \sqrt{\frac{\omega\alpha\rho h(\tau)}{\tau}}$ for $T < T_1$. Hence, we have $\frac{1}{2T\omega\alpha}$ dominates in $f_1(T)$ and $f_1(T) \approx \frac{1}{T\omega\alpha}$. For the same reason, considering (22), if $\eta \to 0$, we have $\eta\varphi \to 0$ from the definition of φ and $h_{FL}(\tau) \to 0$ from its definition. So we can easily derive $\eta\varphi\rho h_{FL}(\tau) \to 0$ and $\sqrt{\frac{\eta\varphi\rho h_{FL}(\tau)}{\tau}} \to 0$. Then for $T < T_1$, $\frac{1}{2T} \gg \eta\varphi\rho h_{FL}(\tau)$ and $\frac{1}{2T} \gg \sqrt{\frac{\eta\varphi\rho h_{FL}(\tau)}{\tau}}$. Hence, we have $\frac{1}{2\eta\varphi T}$ dominates in $f_2(T)$ and $f_2(T) \approx \frac{1}{T\eta\varphi}$. □

Based on Lemma 3, we have the following proposition.

Proposition 3 (Accelerated convergence of MFL). *If the following conditions are satisfied:*

1) $\eta\beta \leq 1$;
2) $T < T_1$;
3) $0 < \gamma < \min\{\frac{2(1-\eta\beta)\cos\theta}{\beta\eta p^2}, 1\}$,

MFL converges faster than FL, i.e.,

$$f_1(T) < f_2(T).$$

Proof. From condition 1 and condition 2, we have $f_1(T) \approx \frac{1}{T\omega\alpha}$ and $f_2(T) \approx \frac{1}{T\eta\varphi}$. Due to the definitions of α and φ, inequality $0 < \gamma < \frac{2(1-\beta\eta)\cos\theta}{\beta\eta p^2}$ is equivalent to $\omega\alpha > \eta\varphi$. So if $\omega\alpha > \eta\varphi$, it is obvious that $\frac{1}{T\omega\alpha} < \frac{1}{T\eta\varphi}$, i.e., $f_1(T) < f_2(T)$. However, $0 < \gamma < 1$ is the condition of MFL convergence. Hence, condition 3 is the range of MFL convergence acceleration after combining with MFL convergence guarantee $0 < \gamma < 1$. □

7 Simulation and Discussion

In this section, we build and evaluate MFL systems based on MNIST and CIFAR-10 datasets. We first describe the simulation environment and the relevant setups of parameters. Second, we present and evaluate the comparative simulation results of MFL, FL and MGD under different machine learning models, which include SVM, linear regression, logistic regression and CNN. Finally, extensive experiments are implemented to explore the impacts of γ, τ and non-i.i.d data distribution on MFL convergence performance, and to investigate the communication efficiency of MFL compared with that of FL.

7.1 Simulation Setup

Using the Python, we build a federated network framework where distributed edge nodes coordinate with the central server. In our network, the number of edge nodes can be chosen arbitrarily. SVM, linear regression, logistic regression and CNN are applied to model training. Loss functions of the first three models at node i are presented as in Table 2 [31], and the loss function of CNN is cross-entropy (see [32] for details). Note that $|\mathcal{D}_i|$ is the number of training samples in node i and the loss function of logistic regression is cross-entropy. For logistic regression, model output $\sigma(\mathbf{w}, \mathbf{x}_j)$ is sigmoid function for non-linear transform. It is defined by

$$\sigma(\mathbf{w}, \mathbf{x}_j) \triangleq \frac{1}{1 + e^{-\mathbf{w}^{\mathrm{T}}\mathbf{x}_j}}. \tag{23}$$

In our experiments, training and testing samples are randomly allocated to each node, which means that the information of each node is uniform and the data distribution at edge nodes is i.i.d. (Only in Section 7.2.2, non-i.i.d data distribution is used and the rest of experiments use i.i.d data distribution). We use FL and centralized MGD as benchmarks for comparison of MFL. If based on SVM, linear and logistic regression models, the deterministic gradient methods are performed for MFL, FL and centralized MGD. However, if based on CNN model, the stochastic gradient methods are used for MFL, FL and centralized MGD due to the large training data size.

SVM, linear and logistic regression are trained and tested on MNIST dataset [33] which contains 50,000 training handwritten digits and 10,000 testing handwritten digits. In our experiments, we only utilize 5,000 training samples and 5,000 testing samples because of the limited processing capacities of GD and MGD. In this dataset, the jth sample $\mathbf{x}_j$ is a 784-dimensional input vector which is vectorized from 28×28 pixel matrix and y_j is the scalar label corresponding to $\mathbf{x}_j$. SVM, linear and logistic regression are used to classify whether the digit is even or odd. If the image of $\mathbf{x}_j$ represents an even number, then we set $y_j = 1$. Otherwise, $y_j = -1$. But for logistic regression, we set $y_j = 1$ for the even number and $y_j = 0$ for the odd.

CNN is trained based on MNIST and CIFAR-10 datasets. The CIFAR-10 dataset includes 50,000 color images for training and 10,000 color images for testing, and has 10 different types of objects [34]. We use CNN to perform the classification among the 10 different labels under MNIST and CIFAR-10 datasets, respectively.

For experimental setups, we set 4 edge nodes in FL and MFL and the training models are distributed into all the edge nodes. The same initializations of model parameters are performed and the same data distributions are set for MFL and FL. Also $\mathbf{d}_i(0) = \mathbf{0}$ is set for node i. We set the learning step size $\eta = 0.002$ which is sufficiently small, SVM parameter $\lambda = 0.3$ and the total number of local iterations $T = 1,000$ for the following simulations.

7.2 Simulation Evaluation

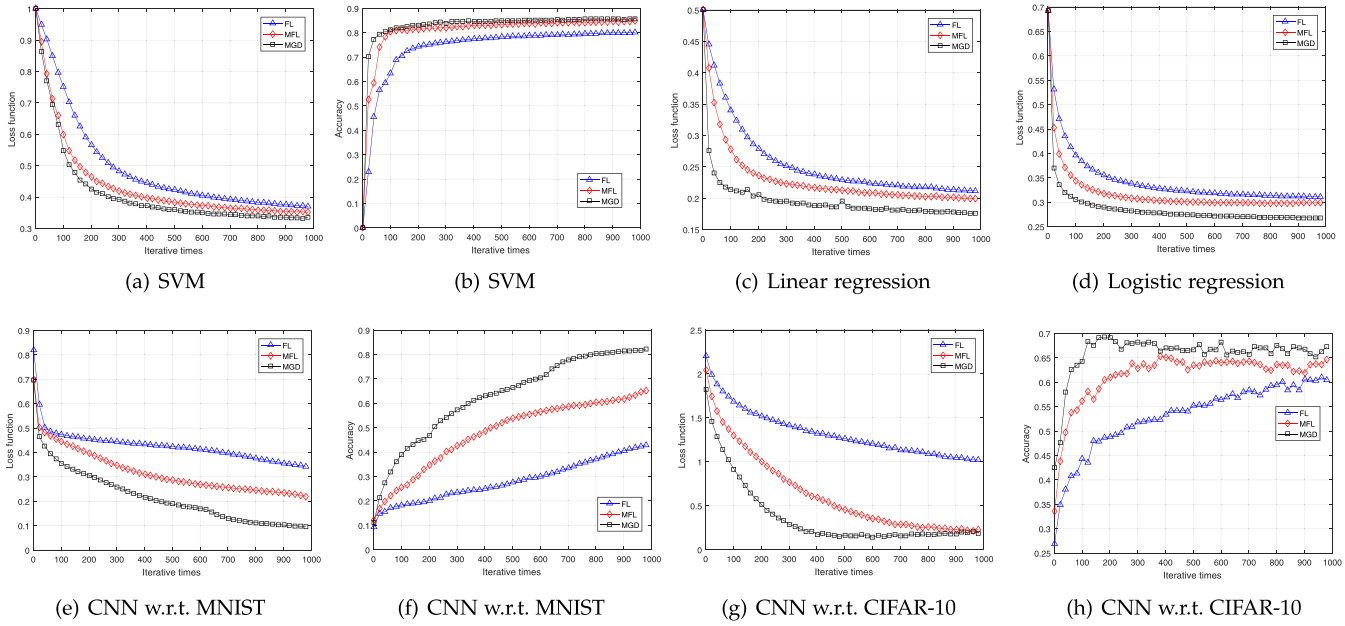
In this subsection, we verify the convergence acceleration of MFL and explore the effects of non-i.i.d data distribution, γ and τ on MFL convergence by simulation evaluation. We further investigate the communication efficiency of MFL compared with that of FL.

7.2.1 Convergence

In our first simulation, the models of SVM, linear regression, logistic regression and CNN are trained and we verify the accelerated convergence of MFL. We set aggregation frequency $\tau = 4$ and momentum attenuation factor $\gamma = 0.5$. MFL, FL and MGD are performed based on the four machine learning models. MGD is implemented based on the global dataset which is obtained by gathering the distributed data on all nodes. The global loss functions of the three solutions are defined based on the same global training and testing data.

The curves of loss function values and accuracy with iterative times are presented in Fig. 5. We can see that the loss function curves for all the learning models are gradually convergent with iterative times. Similarly, the test accuracy curves for SVM and CNN gradually rise until convergence with iterative times. Therefore, convergence of MFL is verified. We also see that the descent speeds of MFL loss function curves on the four learning models are always faster than those of FL while the centralized MGD convergence speeds are the fastest. So compared with FL, MFL provides a significant improvement on convergence rate. MGD converges with the fastest speed because MFL and FL suffer the delay in global gradient update for $\tau = 4$. Finally, comparing the results of CNN and SVM, we can conclude that based on CNN model, MFL still shows similar convergence performance compared with what MFL shows in convex model training. So the proposed MFL can perform well in neural networks with non-convex loss functions.

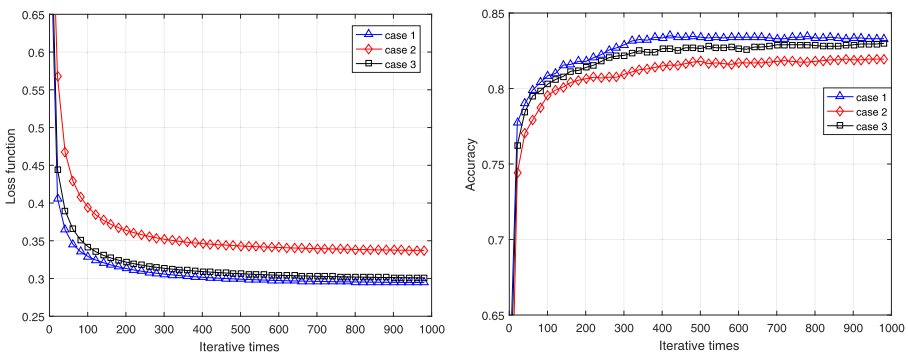
Because linear and logistic regression can not provide the testing accuracy curves, we focus on the SVM model in the



(a) SVM (b) SVM (c) Linear regression (d) Logistic regression

(e) CNN w.r.t. MNIST (f) CNN w.r.t. MNIST (g) CNN w.r.t. CIFAR-10 (h) CNN w.r.t. CIFAR-10

Fig. 5. Loss function values and testing accuracy under FL, MFL and MGD. (a) and (b) are the loss function and test accuracy curves of SVM, respectively; (c) and (d) are the loss function curves of linear regression and logistic regression, respectively. (e) and (f) are the loss function and test accuracy curves of CNN trained on MNIST, respectively. (g) and (h) are the loss function and test accuracy curves of CNN trained on CIFAR-10, respectively.



(a) (b)

Fig. 6. (a) and (b) are loss function and testing accuracy curves under different data distribution cases, respectively.

following experiments and further explore the impact of MFL parameters on convergence rate.

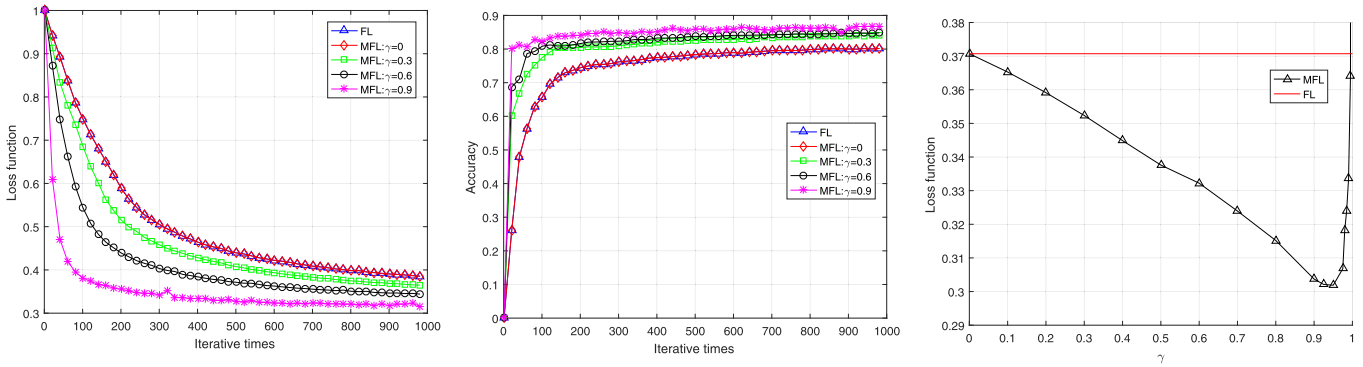
7.2.2 *Effect of Non-i.i.d. Data Distribution*

In the experiment, we consider three cases to distribute the data samples into different nodes. The three data distribution cases at edge nodes are representative for uniform, totally non-uniform information and the mixture of the previous two cases, respectively. The specific settings of the three cases are as follows:

- Case 1: For the uniform information distribution, each data sample is randomly allocated to a node. In this case, we think that the data on each node have uniform characteristics. Therefore, this case satisfies i.i.d data distribution as a benchmark.
- Case 2: All data samples at an individual node have the same label (If there are more labels than nodes, each node could have samples with more than one label but not the total number of labels). Because the global dataset has multiple labels, this case will lead to a non-uniform information distribution, which means characteristics brought by each node are not uniform. Thus, this case corresponds to totally non-i.i.d data distribution.
- Case 3: In this case, the first half of N nodes perform random allocation rule of Case 1 to obtain uniform information and the second half of the nodes perform allocation rule of Case 2. This case is a combination of uniform and non-uniform information. We use this case to explore the effect of the mixture of i.i.d and non-i.i.d data distribution.

In the experiment, SVM is used for the training of MFL network under the above data distribution cases. We set aggregation frequency $\tau = 4$ and momentum attenuation factor $\gamma = 0.5$ for general MFL algorithm.

The experimental results are shown in Fig. 6. The two subfigures show the MFL convergence influence made by different data distributions. We see that the loss function and testing accuracy curves of MFL are always convergent whether data distribution at edge nodes is i.i.d or non-i.i.d, which means even though under non-uniform information



(a) (b) (c)

Fig. 7. The influence of γ on MFL convergence. (a) Loss function values with iterative times under different γ; (b) Testing accuracy with iterative times under different γ; (c) Loss function values with γ when $T = 1000$.

distribution, MFL training still achieves expected convergence and shows its robustness. We also see that Case 2 and Case 3 have worse performance than Case 1, because each node in Case 2 and Case 3 has totally or partially non-uniform information. Further, Case 2 shows the worst convergence performance. The comparison results illustrate that non-i.i.d data distribution still remains MFL convergence but decreases MFL convergence performance.

7.2.3 Effect of γ

We evaluate the impact of γ on the convergence rate of loss function. In this simulation, we still set aggregation frequency $\tau = 4$.

The experimental results are shown in Fig. 7. Subfigure (a) and (b) show that how different values of γ affect the convergence curves of loss function and testing accuracy, respectively. We can see that if $\gamma = 0$, the loss function and accuracy curves of MFL overlap with the corresponding ones of FL because MFL is equivalent to FL for $\gamma = 0$. When γ increases from 0 to 0.9, we can see the convergence rates on both loss function curves and accuracy curves also gradually increase. Subfigure (c) shows the change of final loss function value ($T = 1000$) with $0 < \gamma < 1$. From this subfigure, we can find that the final loss function values of MFL are always smaller than those of FL with $0 < \gamma < 1$. Compared with FL, convergence performance of MFL is improved. This is because $\frac{2(1-\beta\eta)\cos\theta}{\beta\eta\rho^2} > 1$ and according to Proposition 3, the accelerated convergence range of MFL is $0 < \gamma < 1$. We can see that when $0 < \gamma < 0.95$, the loss function values decrease monotonically with γ so the convergence rate of MFL increases with γ. While $\gamma > 0.95$, the loss function values of MFL start to increase with a gradual deterioration of MFL convergence performance, and in this situation, MFL can not remain convergence. If the γ values are chosen to be close to 1, best around 0.9, MFL reaches the optimal convergence rate.

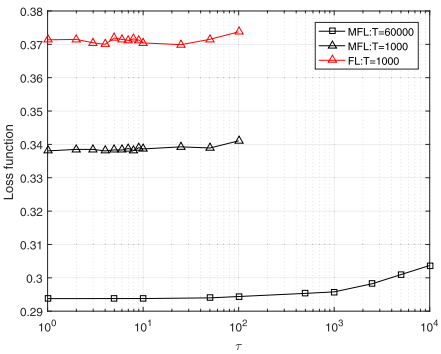


Fig. 8. Loss function values with τ.

7.2.4 Effect of τ

Finally, we evaluate the effect of different τ on loss function of MFL. We record the final loss function values with τ based on the three cases of $T = 1,000$ for FL, $T = 1,000$ for MFL and $T = 60,000$ for MFL. We set $\gamma = 0.5$ for MFL. The curves for the three cases are presented in Fig. 8. Comparing FL with MFL for $T = 1,000$, we see that the final loss function values of MFL are smaller than those of FL for any τ. As declared in Proposition 3, under a small magnitude of T and $\eta = 0.002$ which is close to 0, MFL always converges much faster than FL. Further, for $T = 1,000$, the effect of τ on convergence is slight because the curves of FL and MFL are relatively plain. This can be explained by Lemma 3, where $\frac{1}{2\eta\varphi T}$ and $\frac{1}{2\omega\alpha T}$ dominate the convergence upper-bound when the magnitude of T is small. While $T = 60,000$, change of τ affects convergence significantly and the final loss function values gradually increase with τ. As the cases of $T = 1,000$ for MFL and FL, the case of $T = 60,000$ for MFL has a slight effect on convergence if $\tau < 100$. But if $\tau > 100$, MFL convergence performance is getting worse with τ. According to the above analysis of τ, setting an appropriate aggregation frequency will reduce convergence performance slightly with a decline of communication cost (in our cases, $\tau = 100$).

7.2.5 MFL Communication Efficiency

In the experiment, we evaluate the communication efficiency of MFL compared with FL under different values of γ. Because the momentum and weight are transmitted between the edge nodes and the central server, we can simply assume that the communication size of MFL is twice that of FL due to the additional momentum parameters and the same dimension of momentum and weight. Therefore, we set that communication budget of one global aggregation for MFL is 1, so communication budget of one global aggregation for FL is 0.5. The experiment is based on SVM and we set aggregation frequency $\tau = 4$. The experiment consumes 125 communication

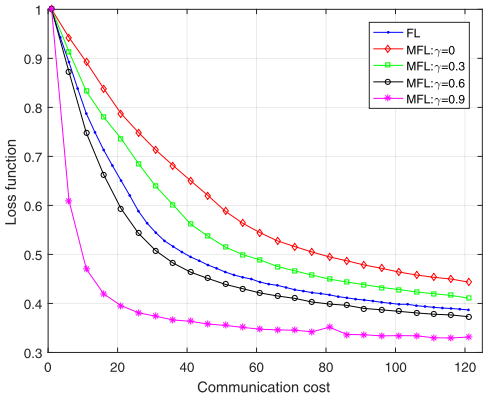


Fig. 9. Comparison of communication efficiency between MFL and FL.

budgets for MFL or FL, so MFL performs 125 global aggregations and FL performs 250 global aggregations.

The experimental results are shown in Fig. 9. We see that the communication efficiency is improved with increasing γ, because γ affects the convergence rate of MFL significantly as presented in Section 7.2.3. We can see that MFL with a large value of γ performs better than FL and a small γ results to worse performance based on the same communication cost. For example, if $\gamma = 0.6$, MFL has a better convergence with respect to communication cost than FL. Thus, MFL shows higher communication efficiency compared with FL for $\gamma = 0.6$.

8 Conclusion

In this paper, we have proposed MFL which performs MGD in local update steps to solve the distributed machine learning problem. First, we have established global convergence properties of MFL and derived an upper bound on MFL convergence rate. This theoretical upper bound shows that the sequence generated by MFL linearly converges to the global optimum point under certain conditions. Then, compared with FL, MFL provides accelerated convergence performance under the given conditions as presented in Proposition 3. Finally, based on MNIST and CIFAR-10 datasets, our simulation results have verified the MFL convergence and confirmed the accelerated convergence of MFL.

Acknowledgments

This work was supported by the National Key Research and Development Program of China under Grant 2018YFA0701603, the National Natural Science Foundation of China under Grant 61722114, and USTC Research Funds of the Double First-Class Initiative (No. YD3500002001).

References

[1] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2015, pp. 2722–2730.

[2] L. Deng *et al.*, "Recent advances in deep learning for speech research at microsoft," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2013, pp. 8604–8608.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.

[4] A. Esteva *et al.*, "Dermatologist-level classification of skin cancer with deep neural networks," *Nature*, vol. 542, no. 7639, 2017, Art. no. 115.

[5] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015.

[6] R. Subramanian and F. Fekri, "Sleep scheduling and lifetime maximization in sensor networks: Fundamental limits and optimal solutions," in *Proc. Int. Conf. Inf. Process. Sensor Netw.*, 2006, pp. 218–225.

[7] G. P. Fettweis, "The tactile internet: Applications and challenges," *IEEE Veh. Technol. Magazine*, vol. 9, no. 1, pp. 64–70, Mar. 2014.

[8] M. Patel *et al.*, "Mobile-edge computing introductory technical white paper," *White Paper, Mobile-Edge Computing (MEC) Industry Initiative*, pp. 1089–7801, 2014.

[9] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 4, pp. 2322–2358, Fourth Quarter 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. Wang *et al.*, "Dynamic resource scheduling in mobile edge cloud with cloud radio access network," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 11, pp. 2429–2445, Nov. 2018.

[12] N. Q. Viet Hung, H. Jeung, and K. Aberer, "An evaluation of model-based approaches to sensor data compression," *IEEE rans. Knowl. Data Eng.*, vol. 25, no. 11, pp. 2434–2447, Nov. 2013.

[13] S. Di, D. Tao, X. Liang, and F. Cappello, "Efficient lossy compression for scientific data based on pointwise relative error bound," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 2, pp. 331–345, Feb. 2019.

[14] S. Di and F. Cappello, "Optimization of error-bounded lossy compression for hard-to-compress HPC data," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 1, pp. 129–143, Jan. 2018.

[15] G. Yang, V. Y. F. Tan, C. K. Ho, S. H. Ting, and Y. L. Guan, "Wireless compressive sensing for energy harvesting sensor nodes," *IEEE Trans. Signal Process.*, vol. 61, no. 18, pp. 4491–4505, Sep. 2013.

[16] H. B. McMahan, B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. Y. Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Proc. 20th Int. Conf. Artif. Intell. Stat.*, vol. 54, pp. 1273–1282, Apr. 2017.

[17] S. Wang *et al.*, "Adaptive federated learning in resource constrained edge computing systems," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 6, pp. 1205–1221, Jun. 2019.

[18] J. Konečnỳ, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," in *Proc. NIPS Workshop Private Multi-Party Mach. Learn.*, 2016. [Online]. Available: https://arxiv.org/abs/1610.05492

[19] C. Hardy, E. Le Merrer, and B. Sericola, "Distributed deep learning on edge-devices: feasibility via adaptive compression," in *Proc. 16th IEEE Int. Symp. Netw. Comput. Appl.*, 2017, pp. 1–8.

[20] K. Bonawitz *et al.*, "Practical secure aggregation for privacy-preserving machine learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 1175–1191.

[21] T. Nishio and R. Yonetani, "Client selection for federated learning with heterogeneous resources in mobile edge," in *Proc. IEEE Int. Conf. Commun. (ICC)*, pp. 1–7, May 2019. [Online]. Available: https://arxiv.org/abs/1804.08333

[22] Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra, "Federated learning with non-IID data," 2018. [Online]. Available: https://arxiv.org/abs/1806.00582

[23] A. S. Nemirovsky and D. B. Yudin, "Problem complexity and method efficiency in optimization," New York: Wiley, 1983.

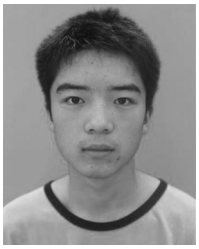
[24] B. T. Polyak, "Some methods of speeding up the convergence of iteration methods," *USSR Comput. Math. Math. Phys.*, vol. 4, no. 5, pp. 1–17, 1964.

[25] E. Ghadimi, H. R. Feyzmahdavian, and M. Johansson, "Global convergence of the heavy-ball method for convex optimization," in *Proc. Eur. Control Conf.*, 2015, pp. 310–315.

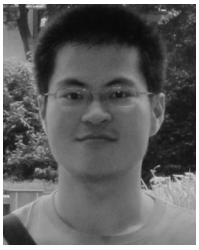
[26] J. Wang, V. Tantia, N. Ballas, and M. Rabbat, "SlowMo: Improving communication-efficient distributed SGD with slow momentum," 2019. [Online]. Available: https://arxiv.org/abs/1910.00643

[27] H. Yu, R. Jin, and S. Yang, "On the linear speedup analysis of communication efficient momentum SGD for distributed non-convex optimization," in *ICML*, 2019, pp. 7184–7193.

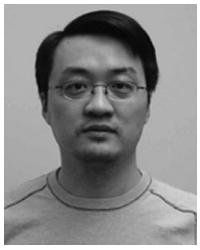
[28] R. Johnson and T. Zhang, "Accelerating stochastic gradient descent using predictive variance reduction," in *Proc. 26th Int. Conf. Neural Inf. Process. Syst.*, 2013, pp. 315–323.
[29] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural Netw.*, vol. 12, no. 1, pp. 145–151, 1999.
[30] Y. Nesterov, *Lectures on Convex Optimization*, vol. 137. Berlin, Germany: Springer, 2018.
[31] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, 2014.
[32] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.
[33] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[34] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., University of Toronto, 2009.



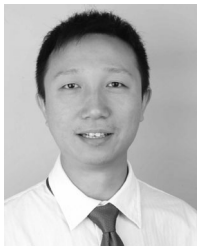
Wei Liu received the BE degree in electronic information engineering from the University of Science and Technology of China, Hefei, China, in 2018. He is currently working toward the ME degree with the Department of Electronic Engineering and Information Science, University of Science and Technology of China. His research interests include distributed machine learning and accelerated computation.



Li Chen received the BE degree in electrical and information engineering from the Harbin Institute of Technology, Harbin, China, in 2009, and the PhD degree in electrical engineering from the University of Science and Technology of China, Hefei, China, in 2014. He is currently a faculty member with the Department of Electronic Engineering and Information Science, University of Science and Technology of China. His research interests include wireless IoT communications and wireless optical communications.



Yunfei Chen (Senior Member, IEEE) received the BE and ME degrees in electronics engineering from Shanghai Jiaotong University, Shanghai, P.R.China, in 1998 and 2001, respectively, and the PhD degree from the University of Alberta, in 2006. He is currently working as an associate professor with the University of Warwick, United Kingdom. His research interests include wireless communications, cognitive radios, wireless relaying and energy harvesting.



Wenyi Zhang (Senior Member, IEEE) received the bachelor's degree in automation from Tsinghua University, in 2001, and the master's and PhD degrees in electrical engineering from the University of Notre Dame, in 2003 and 2006, respectively. He is currently a professor with the Department of Electronic Engineering and Information Science, University of Science and Technology of China. He was affiliated with the Communication Science Institute, University of Southern California, as a postdoctoral research associate, and with Qualcomm Incorporated, Corporate Research and Development. His research interest includes wireless communications and networking, information theory, and statistical signal processing. He was an editor for the *IEEE Communications Letters*, and is currently an editor for the *IEEE Transactions on Wireless Communications*.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

A Hybrid Update Strategy for I/O-Efficient Out-of-Core Graph Processing

Xianghao Xu, Fang Wang, Hong Jiang, *Fellow, IEEE*, Yongli Cheng, Dan Feng, *Member, IEEE*, and Yongxuan Zhang

Abstract—In recent years, a number of out-of-core graph processing systems have been proposed to process graphs with billions of edges on just one commodity computer, due to their high cost efficiency. To obtain a better performance, these systems adopt a full I/O model that scans all edges during the computation to avoid the inefficiency of random I/Os. Although this model ensures good I/O access locality, it leads to a large number of useless edges to be loaded when running graph algorithms that only access a small portion of edges in each iteration. An intuitive method to solve this I/O inefficiency problem is the on-demand I/O model that only accesses the active edges. However, this method only works well for the graph algorithms with very few active edges, since the I/O cost will grow rapidly as the number of active edges increases due to the increasing amount of random I/Os. In this article, we present HUS-Graph, an efficient out-of-core graph processing system to address the above I/O issues and achieve a good balance between I/O traffic and I/O access locality. HUS-Graph adopts a hybrid update strategy including two update models, Row-oriented Push (ROP) and Column-oriented Pull (COP). It supports switching between ROP and COP adaptively, for the graph algorithms that have different computation and I/O features. For traversal-based algorithms, HUS-Graph also provides an immediate propagation-based vertex update scheme to accelerate the vertex state propagation and convergence speed. Furthermore, HUS-Graph adopts a locality-optimized dual-block representation to organize graph data and an I/O-based performance prediction method to enable the system to dynamically select the optimal update model between ROP and COP. To save the disk space and further reduce I/O traffic, HUS-Graph implements a space-efficient storage format by combining several graph compression methods. Extensive experimental results show that HUS-Graph outperforms two existing out-of-core systems GraphChi and GridGraph by 1.2x-52.8x.

Index Terms—Graph processing, out-of-core, I/O, hybrid update strategy

1 Introduction

Graph data has been widely used to model and solve many problems in application areas ranging from social networks to web graphs, from chemical compounds to biological structures, etc. However, with the real-world graphs growing in size and complexity, processing these large and complex graphs in a scalable way has become increasingly more challenging. While a distributed system (e.g., Pregel [1], PowerGraph [2], GraphX [3] and Gemini [4]) is a natural choice for handling these large graphs, a recent trend initiated by GraphChi [5] advocates developing out-of-core support to process large graphs on a single commodity PC.

Out-of-core graph processing systems (e.g., GraphChi [5], X-Stream [6] and GridGraph [7]) utilize secondary storage to process very large graphs and achieve scalability without massive, costly hardware. Furthermore, they overcome the challenges faced by distributed systems, such as load imbalance and significant communication overhead. When processing an input graph, out-of-core systems divide the vertices into disjoint intervals and break the large edge list into smaller shards containing edges with source or destination vertices in corresponding vertex intervals. They process one vertex interval and its associated edge shard at a time. To obtain a better performance, these systems adopt a full I/O model to utilize the sequential bandwidth of disk and minimize the random I/Os. In this way, each edge shard is loaded entirely into memory, even though a large number of edges in the shard may not be needed.

Fig. 1 shows the percentage of active edges (the edges that have active sources vertices and are accessed in current iteration) per iteration for different iterative graph algorithms on LiveJournal graph [8]. For PageRank,[1] all edges are always active as all vertices compute their PR values in each iteration. For Breadth-first Search (BFS) and Weakly Connected Components (WCC), the number of active edges is small in most iterations. In this case, the pursuit of high I/O bandwidth overshadows the usefulness of data accesses. Repeatedly

- *X. Xu, F. Wang, D. Feng, and Y. Zhang are with the Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei 430074, China. E-mail: {xianghao, wangfang, dfeng, zyx}@hust.edu.cn.*
- *H. Jiang is with the Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX 76019. E-mail: hong.jiang@uta.edu.*
- *Y. Cheng is with the College of Mathematics and Computer Science, Fuzhou University, Fuzhou, Fujian 350116, China, and also with the Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei 430074, China. E-mail: chengyongli@fzu.edu.cn.*

Manuscript received 28 Dec. 2018; revised 1 Feb. 2020; accepted 5 Feb. 2020. Date of publication 11 Feb. 2020; date of current version 23 Mar. 2020. (Corresponding author: Fang Wang.) Recommended for acceptance by P. Sadayappan. Digital Object Identifier no. 10.1109/TPDS.2020.2973143

1. This is a standard implementation of PageRank. There is another implementation of PageRank (PageRank-Delata) where vertices are active in an iteration only if they have accumulated enough change in their PR value.

1045-9219 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

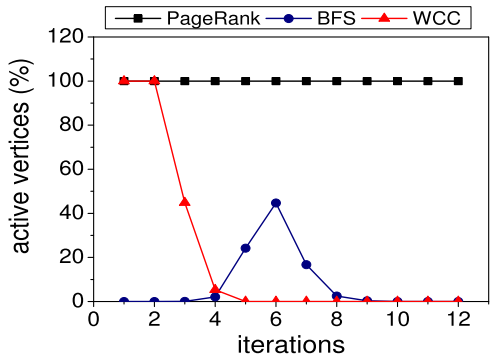


Fig. 1. The percentage of active edges per iteration.

loading the useless edges incurs significant I/O inefficiencies that degrade the overall performance. An intuitive method to solve this problem is the on-demand I/O model that only accesses the active edges. However, this method only works well for the graph algorithms with very few active edges. It incurs a large amount of small random disk accesses when the number of active edges is large due to the discontinuous distribution of active edges on disk. Dynamic Shards proposed in [9] eliminate this issue of random disk accesses by developing dynamic partitions whose layouts change during the runtime to compactly capture the set of active edges. It achieves this in light-weight manner by marking the active edges during compute phase and writing those marked edges back in separate partitions on disk during write-back phase. While [9] significantly eliminates loading of inactive edges, it requires writing back active edges. This dilemma motivates us to propose a new out-of-core system that achieves a good balance between I/O traffic and I/O access locality.

In this paper, we present HUS-Graph, an I/O-efficient out-of-core graph processing system with a hybrid update strategy. Our work is inspired by state-of-art shared-memory graph processing systems [10], [11] that utilize an adaptive push-pull model to handle graphs with different densities of active edges (percentage of active edges in all edges). When the density of active edges is sparse, these systems adopt a push-style model to only traverse the active edges and push updates to their destination vertices, which skips the processing of useless edges. When the density of active edges is dense, these systems adopt a pull-style model where each vertex collects data from its neighbors through its incoming edges and then updates its own value with the collected data, which eliminates atomic operations and enables full parallelism. By adaptively switching between these two update models, the system can handle different active edges densities with optimized performance. We extend this hybrid solution to the disk-based scenario. However, this is non-trivial work due to the following reasons. First, in both push and pull models, the vertices access their neighbors to update or obtain data. This can cause a large number of random and frequent disk accesses in out-of-core systems when the vertex values are too large to be cached in memory, which impacts the overall performance. Second, in order to gain optimal performance, it requires us to explore an effective performance prediction mechanism that guides the system to adaptively switch between push and pull models. HUS-Graph solves these problems by adopting a locality-optimized graph representation and an I/O-based performance prediction method.

The main contributions of our work are summarized as follows.

- *Locality-optimized graph representation*. HUS-Graph proposes a dual-block representation to organize the graph data. It divides the vertices into several disjoint intervals and groups the outgoing and incoming edges of a vertex interval in out-blocks and in-blocks according to the source and destination vertices respectively. By restricting data access to each out-block or in-block and corresponding source and destination vertices, access locality can be ensured under the dual-block representation.
- *Hybrid update strategy*. HUS-Graph proposes two update models, Row-oriented Push (ROP) and Column-oriented Pull (COP), to accommodate different computation and I/O loads. When running algorithms with sparse active edge sets, ROP only traverses the active edges and pushes updates to vertices, which avoids the loading of useless data. When running algorithms with dense active edge sets, COP streams all edges and updates vertices by pulling updates from neighbors, which overcomes the challenges of random disk accesses. For traversal-based algorithms, HUS-Graph also provides an immediate propagation-based vertex update scheme to accelerate the vertex states propagation and convergence speed. By utilizing an I/O-based performance prediction method, HUS-Graph can dynamically select the optimal update model based on the I/O load of the current iteration.
- *Space-efficient storage format*. To save the disk space and further reduce I/O traffic, HUS-Graph implements a space-efficient storage format by combining several graph compression methods. Compared with our previous work [12], the space-efficient storage format can reduce disk consumption by up to 71 percent, which further significantly reduces I/O traffic and improves system performance with very little extra preprocessing and decompression time.
- *Extensive experiments*. We evaluate HUS-Graph on several real-world graphs with different algorithms. Extensive evaluation results show that HUS-Graph outperforms GraphChi and GridGraph significantly, from 4.1x to 58.2x and from 1.2x to 29.3x respectively due to its efficient hybrid update strategy that brings a great improvement of I/O performance.

The rest of the paper is organized as follows. Section 2 presents the background and motivation. The system design is detailed in Section 3. Section 4 presents an extensive performance evaluation. We discuss the related works in Section 5 and conclude this paper in Section 6.

2 Background and Motivation

In this section, we first introduce the I/O issues of out-of-core graph processing systems. Then we present the prominent features of the adaptive push-pull model and its applying in shared-memory and distributed systems. This helps motivate us to propose a new out-of-core system that utilizes the hybrid solutions to solve the I/O issues of current out-of-core systems.

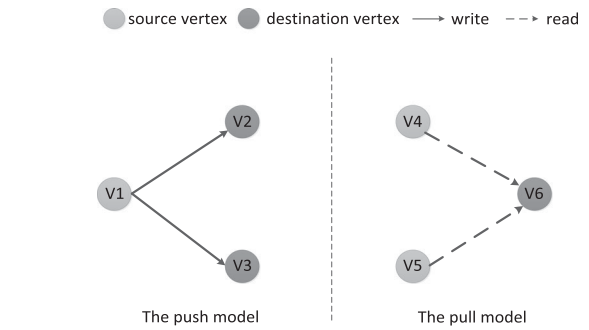


Fig. 2. The push and pull models.

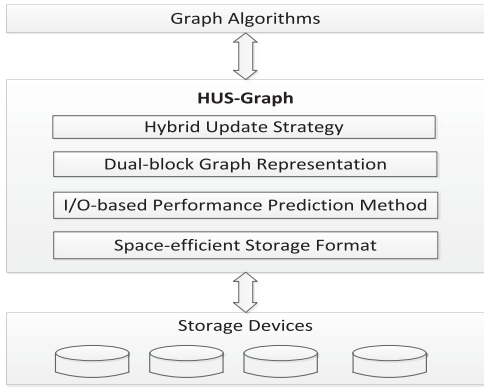


Fig. 3. The HUS-Graph architecture.

2.1 I/O Issues in Out-of-Core Systems

As many works [5], [6], [7] have shown, out-of-core graph processing systems can efficiently process billion-scale graphs on a PC. They process large graphs by efficiently using the disk drives. As the major performance bottleneck is disk I/O overhead [9], these systems are usually optimized for the sequential performance of disk drives and eliminate random I/Os by scanning the entire graph data in all iterations of graph algorithms. This full I/O model can be wasteful for algorithms that access only small portions of data during each iteration, such as BFS in Fig. 1. On the other hand, the on-demand I/O model that is based on the active edges can avoid loading the useless data. Unfortunately, it incurs a large amount of small random disk accesses due to the randomness of the active vertices. As we know, random access to disk drives delivers much less bandwidth than sequential access. Therefore, only accessing the useful data for out-of-core graph processing is an overkill when the number of active vertices is large. [9] solves this problem by employing dynamic partitions whose layouts are dynamically adjustable in each iteration. It achieves this in light-weight manner by marking the active edges during compute phase and writing those marked edges back in separate partitions on disk during write-back phase. Although [9] can eliminate random disk accesses, it requires writing back active edges.

2.2 Adaptive Push-Pull Model

The push and pull update models are extensively used in graph processing [4], [10], [11]. As shown in Fig. 2, in the push model, each vertex passes the updates to its neighbors through its outgoing edges. In the pull model, each vertex collects data from its neighbors through its incoming edges, and then updates its own value with the collected data. Obviously, push and pull are suitable for different scenarios, which is determined by the number of active edges. Specifically, sparse active edge set prefers the push model, since the system only traverses the outgoing edges of the active vertices where new updates are made. Contrarily, dense active edge set prefers the pull model, since it significantly reduces the contention in updating vertex states via locks or atomic operations. Inspired by this principle, several shared-memory systems [10], [11] and distributed systems [4], [13] adopt an adaptive update model during graph processing. For example, Ligra [11] proposes a lightweight graph processing framework that adaptively switches between the push and pull models according to the densities of active edge set in a shared-memory machine. Gemini [4] extends such adaptive design to distributed systems and proposes a sparse-dense signal-slot model.

However, current out-of-core systems either use the push model [6], [7] or the pull model [5], [14]. Furthermore, the push model for these systems is based on all vertices rather than the active vertices, since they put a higher priority on I/O access locality rather than I/O traffic. Actually, the adaptive push-pull model works well for out-of-core systems as well. When running algorithms with sparse active edge sets, the push model enables selective data access that only traverses the active edges, which fully avoids the loading of useless data. When running algorithms with dense active edge sets, the pull model sequentially accesses the edges of all vertices, which overcomes the challenges of random accesses and enables full parallelism. This motivates us to extend this hybrid solution to disk-based scenario to solve the I/O issues of current out-of-core systems.

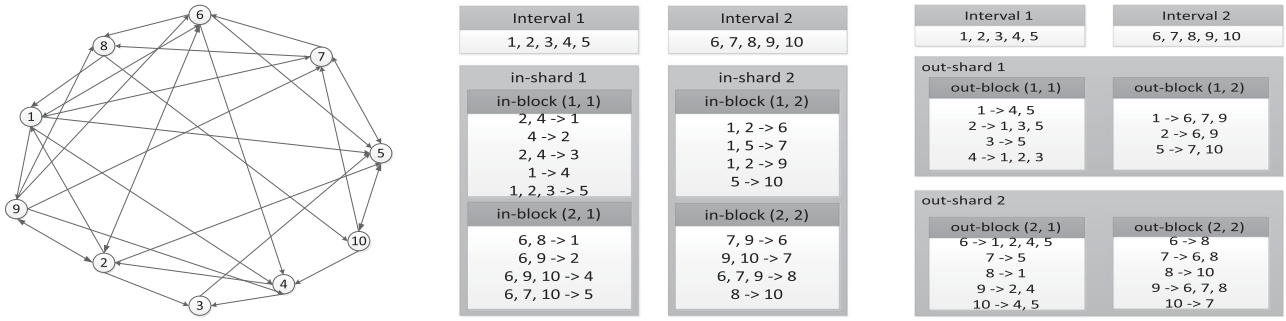
3 System Design

In this section, we first introduce the system overview of HUS-Graph. Then, we present the detail designs including the graph representation, space-efficient storage format, hybrid update strategy and the I/O-based performance prediction method.

3.1 System Overview

A graph $G = (V, E)$ is composed of its vertices V and edges E. For a directed edge $e = (u, v)$, we refer to e as v's *in-edge*, and u's *out-edge*. Additionally, u is an *in-neighbor* of v, v is an *out-neighbor* of u. The computation of a graph G is usually organized in several iterations where V and E are read and updated. Updating messages are propagated from source vertices to destination vertices through the edges. The computation terminates after a given number of iterations or when it converges.

Like previous out-of-core graph processing systems, HUS-Graph focuses on maximizing the I/O performance as well. It improves the I/O performance by achieving a good balance between I/O traffic and I/O access locality. To achieve this, it adopts a hybrid update strategy including Row-oriented Push (ROP) and Column-oriented Pull (COP), inspired by shared-memory systems. Fig. 3 presents the system architecture of HUS-Graph. To efficiently support the hybrid update strategy, HUS-Graph adopts a dual-block representation to organize the graph data, which provides



(a) Example graph (b) In-blocks (c) Out-blocks

Fig. 4. Illustration of the dual-block representation.

fast loading of out-edges (in ROP) and scanning of in-edges (in COP). By restricting data access to each out-block or in-block and corresponding source and destination vertices, locality can be ensured under the dual-block representation. In addition, HUS-Graph implements an I/O-based performance prediction method that enables the system to dynamically select the optimal update model based on the I/O load of the current iteration.

Compared with current graph processing systems, HUS-Graph is different in the following two aspects.

Compared with current out-of-core systems, HUS-Graph handles graph algorithms that have different computation and I/O loads (number of active edges) with hybrid update strategy to achieve an optimal performance. While current out-of-core systems either adopt a push-style update [6], [7], [15] or a pull-style update [5], [14]. Second, HUS-Graph achieves a good balance between I/O traffic and I/O access locality, while current out-of-core systems improve the I/O access locality at the expense of larger amount of disk I/O, which degrades the overall performance especially when the number of active edges is small.

Compared with shared memory or distributed systems that use the adaptive push/pull model, HUS-Graph makes great efforts and proposes different techniques to overcome several new challenges when extending adaptive push/pull model to out-of-core systems. First, in both push and pull models, the vertices frequently access their neighbors to update the latter or obtain data for their own update. This can cause a large number of random disk accesses in out-of-core systems when the vertex values are too large to be cached in memory, notably reducing system performance. Second, in order to gain optimal performance, some important issues such as the proper switching time and effective performance prediction model must be explored, which is not easy in an out-of-core system whose performance depends highly on disk I/O overhead. To address these challenges, HUS-Graph adopts a locality-optimized graph representation and an I/O-based performance prediction method.

3.2 Graph Representation

The hybrid update strategy requires the system to store both out-edges and in-edges of vertices to support the adaptive processing. Unlike previous systems [4], [11] that use 1-dimensional partitioning to store the out-edges/in-edges, HUS-Graph adopts a 2-dimensional partitioning method and implements a dual-block representation to improve the locality of vertex access.

Like many out-of-core systems, HUS-Graph first splits the vertices V of graph G into P disjoint intervals. Each interval associates two edge shards, *in-shard* and *out-shard*, to respectively store the in-edges and out-edges of the vertices within the interval. Moreover, each in-shard is further partitioned into P *in-blocks* according to their source vertices. Similarly, each out-shard is partitioned into P *out-blocks* according to their destination vertices. Edges inside each in-block and out-block are respectively sorted by the destination and source vertices. In this way, both in-edges and out-edges are partitioned into $P \times P$ edge blocks. Each in-block (i, j) or out-block (i, j) contains edges that start from vertices in interval i and end in vertices in interval j. By selecting P such that each edge-block and the corresponding vertices can fit in memory, HUS-Graph can ensure good locality when processing each edge-block.

Fig. 4 shows the dual-block representation of an example graph. The vertices are divided into two equal intervals (1, 5) and (6, 10), the in-edges and out-edges are respectively partitioned into four in-blocks and four out-blocks according to the two intervals. For example, the out-edge (1, 6) is partitioned into out-blocks (1, 2) since vertex 1 belongs to interval 1 and vertex 6 belongs to interval 2.

In addition, we also maintain the index to the edges for each vertex. We refer to in-index (i, j) as the vertex index of in-block (i, j) and out-index (i, j) as the vertex index of out-block (i, j). This enables selective loading of the active edges in ROP and parallel update in COP.

3.3 Space-Efficient Storage Format

Since the dual-block representation stores both in-edges and out-edges plus the vertex index of each in-block and out-block, the storage size of of a graph is $2(|E| + P|V|)$ where P is the number of vertex intervals. This consumes more disk space than the existing graph representations such as Compressed Sparse Row (CSR) [5] whose storage size of a graph is $|E| + |V|$. To solve this problem, we implement a more space-efficient storage format by combining several graph compression methods, i.e., compression of undirected graph and ID compression.

Compression of Undirected Graph. For undirected graphs, HUS-Graph stores each edge twice, one for each of the two directions. Assuming that the example graph in Fig. 4a is undirected, its dual-block representation is shown in Fig. 5. Actually, for an undirected edge $e = (u, v)$, e can be regarded as the in-edge and out-edge of u and v simultaneously. Therefore, in-block (1, 1) and out-block (1, 1) are a duplicate

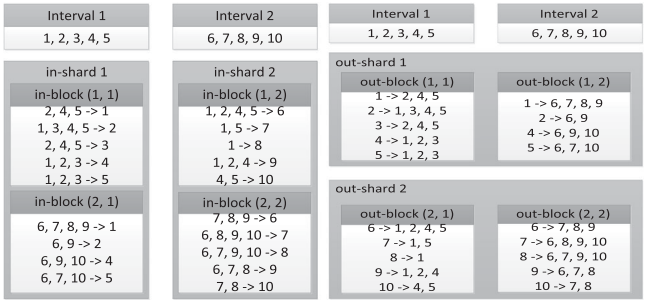


Fig. 5. Illustration of the dual-block representation when storing an undirected graph.

of each other as shown in Fig. 5, since the in-edges of an interval can also work as the out-edges. Similarly, in-block (1, 2) and out-block (2, 1), in-block (2, 1) and out-block (1, 2), in-block (2, 2) and out-block (2, 2) duplicate each other as well. Generally, for any two vertex intervals i and j in an undirected graph G with dual-block representation, if $i = j$, in-block (i, j) is a duplicate of out-block (i, j). Otherwise, in-block (i, j) is a duplicate of out-block ($P - i$, $P - j$), where $1 \leq i \leq P, 1 \leq j \leq P$.

To avoid the above redundant storage, HUS-Graph only maintains one copy of edges for undirected graphs, i.e., only storing in-edges or out-edges of an interval. For directed graphs, it still stores both in-edges and out-edges to enable different update models.

ID Compression. We further compress the ID of each vertex. To achieve the compression of vertex ID, we combine two methods as follows.

1)*Delta-based compression*. In fact, each edge block in the dual-block representation consists of all adjacency lists of the vertices in an interval. The adjacency list of a vertex consecutively stores the vertex IDs of the vertex's neighbors. We can compress the adjacency lists by utilizing the delta values of vertex IDs. This is motivated by the locality and similarity in web graphs [16] where most links contained in a page lead the user to some other pages within the same host. In this case, the neighbors of many vertices may have similar vertex IDs. Instead of storing all vertex IDs in an adjacency list, HUS-Graph stores the vertex ID of the first neighbors and the delta values of the vertex IDs of remaining neighbors.

2)*Variable-length ID encoding*. Current systems always store the ID as an integer of four-byte or eight-byte length. However, this can be wasteful if the IDs are of small values. HUS-Graph adopts a variable-length integer [17] to encode each vertex ID (including the delta values). Thus, a minimum number of bytes are used to encode a given integer. Furthermore, the most significant bit of each compressed byte is used to indicate different IDs and the remaining seven bits are used to store the value. For example, considering an adjacency list of vertex $v1$, $adj(v1) = \{v2, v3, v4\}$. Supposing that the IDs of $v2$, $v3$ and $v4$ are 2, 5, and 300, the adjacency list of vertex $v1$ is stored as "00000010 10000011 00000010 00100111". The first byte is the id of 2, and the second byte is the delta value between 2 and 5 (removing the most significant bit). The third byte and the fourth byte have the same most significant bit, which means that they are used to encode the same ID. 00000100100111 (after removing the most significant bit of the third and fourth byte) is the delta value between 300 and 5.

By combining these compression techniques, our space-efficient storage format can reduce disk consumption by up to 71 percent, which further significantly reduces I/O traffic and improves system performance with very little extra preprocessing and decompression time, as shown in the evaluation results in Section 4.4.

3.4 Hybrid Update Strategy

Like many out-of-core systems, HUS-Graph processes the input graph one vertex interval at a time. Algorithm 1 shows the computation procedure of HUS-Graph in one iteration. For the processing of each vertex interval, HUS-Graph proposes two update models, Row-oriented Push (ROP) and Column-oriented Pull (COP), to accommodate different I/O and computation loads of graph algorithms. The selection between ROP and COP is based on the number of active vertices with the I/O-based performance prediction method that is further discussed in Section 3.7. For both ROP and COP, we maintain two copies of vertex values for each interval i, source interval S_i and destination interval D_i. S_i stores the vertex values of previous iteration, serving as the source vertices. D_i stores the vertex values of current iteration, serving as the destination vertices.

Algorithm 1. Pseudo Code of HUS-Graph Execution

```
1: for each interval i do
2:    Out ← NewActiveVerticesSet
3:    /* Identify the active vertices in interval i*/
4:    V_active ← GetActiveVertices(i)
5:    /* Select the update strategy, using the I/O-based performance prediction method*/
6:    model ← UpdateModelSelection(V_active)
7:    if model = ROP then
8:       /* Implement ROP model, using Alg. 2*/
9:       RowOrientedPush(i, Out, V_active)
10:   else
11:      /* Implement COP model, using Alg. 3*/
12:      ColumnOrientedPull(i, Out)
13:   end if
14: end for
```

1)*Row-oriented Push. Execution*. ROP processes the input graph by pushing updates through the out-edges with the row-oriented process order. Algorithm 2 shows the procedure of ROP to execute a vertex interval i. ROP successively processes the out-blocks in a row from out-block (i, 1) to out-block (i, P) (Lines 2 $\sim$ 16). For the processing of each out-block (i , j) ($1 \leq j \leq P$), ROP first loads vertex values of S_i and D_j as well as the corresponding out-index. Then, it locates the out-edges of the active vertices in the out-block based on the corresponding out-index and loads them into memory (Line 8). As soon as the edges are loaded, ROP traverses the loaded edges and pushes the updates to their out-neighbors with a user-defined update function (Lines 9 $\sim$ 14). In this process, the vertex values in S_i are read-only while the vertex values in D_j are write-only. If there is new vertex activated, it is added to the new active vertex set and will be scheduled in the next iteration (Lines 11 $\sim$ 12). After the active edges in all out-blocks of interval i (in the ith row) are processed, ROP synchronizes the vertex values by replacing the values of source intervals with the values of destination intervals for

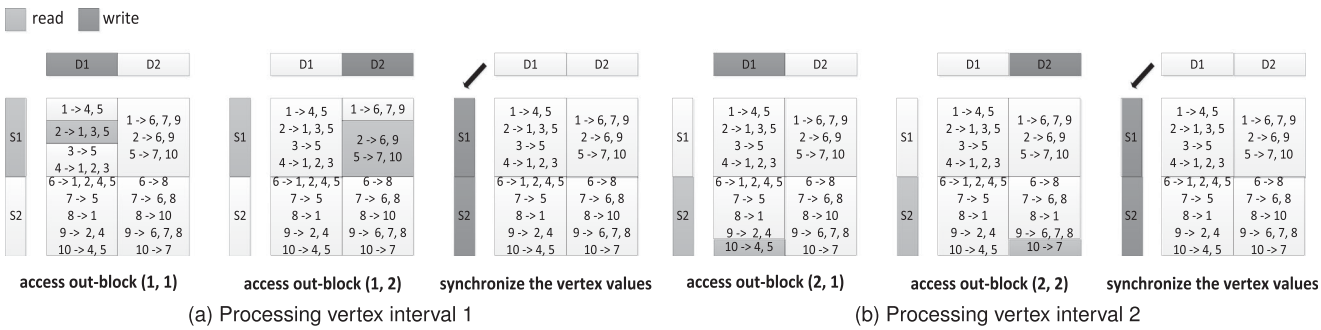


Fig. 6. Execution procedure of ROP.

subsequent computation (Lines 17 $\sim$ 19). Fig. 6 illustrates the execution procedure of ROP with the example graph in Fig. 4.

Algorithm 2. Pseudo Code of *RowOrientedPush* Function

```
1: LoadfromDisk(S_i)
2: /*parallel loops that overlap the processing of out-blocks of
   one row*/
3: for j from 0 to P-1 do
4:    LoadfromDisk(D_j)
5:    OutIndex ← out − index(i, j)
6:    for each active vertex v in V_active do
7:       out − degree ← OutIndex(v + 1) − OutIndex(v)
8:       edges ← LoadOutEdges(OutIndex(v), out − degree,
         out − block(i, j))
9:       for each edge e in edges do
10:         neighbor ← e.dst
11:         if UserUpdateFunction(v, neighbor) then
12:            Out.add(neighbor)
13:         end if
14:      end for
15:   end for
16: end for
17: for j from 0 to P-1 then
18:    S_j ← D_j
19: end for
```

Example. In the example of Fig. 6, ROP iterates over out-block (1, 1) to out-block (1, 2) when processing interval 1. Supposing that the active vertices of interval 1 in current iteration are vertex 2, 5 and 10. ROP successively loads and processes the out-edges of vertex 2 and 5 and updates their destinations when executing each out-block. Furthermore, processing of each out-block does not conflict with others due to the disjoint destination intervals. ROP can overlap the processing of out-blocks to fully exploit the multi-threading. When the processing of the first row of out-blocks is finished, ROP synchronizes the vertex values of all source intervals and destination intervals and moves to the next row to process out-block (2, 1) and out-block (2, 2).

I/O Cost Analysis. The I/O cost can be calculated by the total size of data accessed divided by the random/sequential throughput of disk access. Let M be the average size of the compressed adjacency list of a vertex and N be the size of a vertex value record. The active vertex set for each vertex interval i is A_i. In addition, T_{rr}, T_{rw}, T_{sr} and T_{sw} respectively represent random read, random write, sequential read and sequential write throughput (MB/s). For easy reference, we list the notations in Table 1.

For ROP, when processing vertex interval i, HUS-Graph only loads the out-edges of the active vertices, so the size of the active edges is $|A_i| \times M$ since the edge blocks consist of the compressed adjacency lists of vertices, as mentioned in Section 3.3. In addition, the vertex values of source interval (S_i) and destination intervals ($D_1 \sim D_P$) as well as the vertex indices are also loaded into memory. For the data writes, since the mutable attributes are stored in vertices, only vertex values of the destination intervals are written back to disk. For the ease of expression, we assume that the number of vertices in each interval is equal to $|V|/P$. Therefore, the I/O cost of processing vertex interval i, C_{rop}, can be expressed as

$$C_{rop} = \frac{|A_i| \times M}{T_{rr}} + \frac{(\frac{|V|}{P} + 2|V|) \times N}{T_{sr}} + \frac{|V| \times N}{T_{sw}}.$$

Summary. ROP puts a higher priority on I/O traffic rather than I/O access locality when processing a graph. It avoids the loading of useless data by only loading the active edges. Furthermore, with the row-oriented process order, ROP can overlap the processing of out-blocks to fully exploit the multi-threading when executing a vertex interval (Lines 2 $\sim$ 3), since the destination intervals of different out-blocks in a row are disjoint.

2) *Column-oriented Pull*. *Execution*. COP processes the input graph by pulling updates through the in-edges and updating the vertices with the column-oriented process order. Algorithm 3 shows the procedure of COP when executing a vertex interval i. COP successively processes the in-blocks in a column from in-block (1, i) to in-block (P, i) (Lines 2 $\sim$ 19). For the processing of each in-block (j, i) ($1 \leq j \leq P$), COP streams all in-edges of the in-block and loads the vertex

TABLE 1
Notations

Notation	Definition
G	the graph $G = (V, E)$
V	vertices in G
E	edges in G
P	number of intervals
A_i	active vertex set of interval i
M	average size of a compressed adjacency list
N	size of a vertex value
T_{rr}	random read throughput
T_{rw}	random write throughput
T_{sr}	sequential read throughput
T_{sw}	sequential write throughput

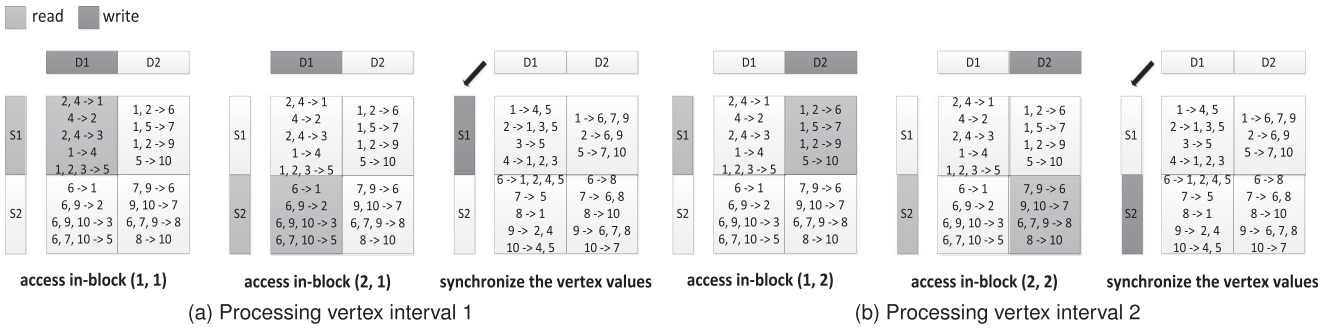


Fig. 7. Execution procedure of COP.

values of S_j and D_i. For each vertex in interval i , it locates its own in-edges and accesses its in-neighbors based on the corresponding in-index of the in-block (Line 9). Then, it collects data from the in-neighbors by reading S_j and updates its own value with a user-defined update function (Lines 9 $\sim$ 16). If there is new vertex activated, it is added to the new active vertex set. After all in-blocks of the interval i (in the ith column) are processed, COP replaces the S_i with D_i to synchronize the vertex values (Line 20). Fig. 7 illustrates the execution procedure of COP with the example graph in Fig. 4.

Algorithm 3. Pseudo Code of *ColumnOrientedPush* Function

```
1: LoadfromDisk(D_i)
2: for j from 0 to P-1 do
3:    LoadfromDisk(S_j)
4:    InIndex ← in − index(j, i)
5:    edges ← LoadInEdges(in − block(j, i))
6:    /*each vertex pulls updates in parallel*/
7:    for each vertex v in D_i do
8:       in − degree ← InIndex(v + 1) − InIndex(v)
9:       inedges ← edges.locate(InIndex(v), in − degree)
10:      for each edge e in inedges do
11:         neighbor ← e.src
12:         if IsActive(neighbor) then
13:            if UserUpdateFunction(neighbor, v) then
14:               Out.add(v)
15:            end if
16:         end if
17:      end for
18:   end for
19: end for
20: S_i ← D_i
```

Example. In the example of Fig. 7, COP iterates over in-block (1, 1) to in-block (2, 1) when processing interval 1. During the computation, each vertex in interval 1 can pull updates from their in-neighbors through the in-edges to update its own value in parallel. After the processing of the first column of in-blocks, COP replaces S_1 with D_1 and moves to process the next column.

I/O Cost Analysis. For COP, when processing vertex interval i, HUS-Graph loads all in-edges within an interval. Furthermore, the vertex values of source intervals ($S_1 \sim S_P$) and destination interval (D_i) as well as the vertex indices are also loaded into memory. For the data writes, only vertex values of the destination interval are written back to disk. Thus C_{cop} of vertex interval i is constant and can be expressed as

$$C_{cop} = \frac{\frac{|V|}{P} \times M + (\frac{|V|}{P} + 2|V|) \times N}{T_{sr}} + \frac{\frac{|V|}{P} \times N}{T_{sw}}.$$

Summary. Contrary to ROP, COP improves the I/O access locality at the expense of larger amount of disk I/O. Although it can not overlap the processing of the in-blocks in a column due to the write conflicts, the parallelism within each in-block can be achieved (Lines 6 $\sim$ 7) since the edges are sorted by the destination vertices. Note that ROP only accesses the out-edges stored in out-blocks while COP only accesses the in-edges stored in in-blocks. By restricting data access to each out-block or in-block and corresponding source and destination vertices, both ROP and COP ensure the locality when accessing vertices.

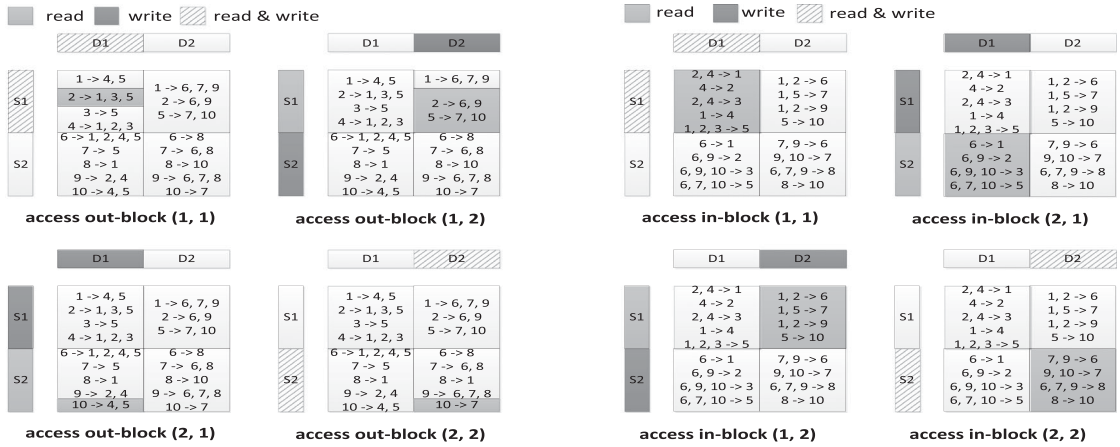
Note that our programming model is similar to Ligra's dense/sparse model [11]. In our programming model (Algorithms 1, 2 and 3), only function UserUpdateFunction is user-defined, while the others are provided by runtime. UserUpdateFunction is applied to the edges to conduct the user-defined programs, and identify if the destination vertex is activated. Algorithm 4 shows the implementation of UserUpdateFunction of Connected Components (CC).

Algorithm 4. *UserUpdateFunction (s, d)*: CC

```
1: Procedure CC
2: if s.label < d.label then
3:    d.label = s.label
4:    Return 1
5: else
6:    Return 0
7: end if
8: End Procedure
```

3.5 Comparison With GridGraph-Like Systems

Note that, some works like GridGraph [7] also use a 2-dimensional partitioning method and present a grid-like format to improve the I/O performance, which is similar to the dual-block representation of HUS-Graph. Moreover, GridGraph also supports column-oriented and row-oriented processing as well as selective scheduling. However, HUS-Graph adopts some new designs in both graph representation and graph processing execution model.



(a) Execution procedure of ROP (b) Execution procedure of COP

Fig. 8. Execution procedure of ROP and COP when using immediate propagation-based vertex update scheme.

In the graph representation, first, HUS-Graph stores both in-edges and out-edges to enable hybrid processing. Second, unlike GridGraph, HUS-Graph sorts the in-edges and out-edges by the destination and source vertices respectively, so that in-edges with the same destination vertex and out-edges with the same source vertex are stored contiguously. This is beneficial for the compression of edges and efficient parallel processing. Third, HUS-Graph creates a vertex index structure to enable the selective loading of edges.

In the graph processing execution model, HUS-Graph differs from GridGraph by adopting the following new designs based on the graph representation.

Hybrid Processing. Although GridGraph can also support column-oriented and row-oriented processing, it processes each edge in one direction, i.e., top-down update (push-style update) [11], shown in GridGraph's programming model. For HUS-Graph, it provides different update models for different edges and processing orders. For out-edges (ROP), HUS-Graph pushes updates to the destination vertices, which is a top-down update. For in-edges (COP), HUS-Graph adopts a pull-style model where each vertex pulls updates from the source vertices, which is a bottom-up update. By combining the two update models, HUS-Graph can enable selective loading of edges when using the push-style update and avoid the atomic operations when using the pull-style update, as described in Section 2.2.

Efficient Edge Storing and Parallel Processing. Due to the sorting of in-edges and out-edges, in-edges with the same destination vertex and out-edges with the same source vertex are stored contiguously. Therefore, HUS-Graph can enable edge compression and space-efficient storage format to reduce disk I/O. Moreover, fine-grained parallel processing can be achieved as shown in Section 3.4. While for GridGraph, it can not enable edges compression and fully utilize the parallelism without sorting edges [18].

Fine-Grained Selective Scheduling. By efficiently using the vertex index structure, HUS-Graph enables the selective loading of the active edges and avoids the loading of useless data. While for GridGraph, it can only skip processing the sub-blocks without any active-edges. This means that it loads and processes a sub-block even though there is only one active edge, which is a much more coarse-grained selective scheduling.

3.6 Immediate Propagation-Based Vertex Update

In both ROP and COP, the vertex update messages are propagated by synchronizing the vertex values of source and destination intervals. Concretely, HUS-Graph will replace the values of source intervals with the values of destination intervals until all out-blocks of one row are processed in ROP or until all in-blocks of one column are processed in COP. We call this an accumulation-based vertex update scheme. This scheme is particularly suitable for sparse matrix multiplication algorithms such as PageRank. These algorithms will not propagate update messages to vertices until all the update messages from their source vertices are accumulated. For traversal-based algorithms, they need not to accumulate all update messages of source vertices to update the states of vertices. For example, when running BFS, any active in-neighbor of a vertex can send an update message to it and make it active. Therefore, the accumulation-based vertex update scheme is suboptimal for these algorithms since the updated messages can not be applied instantly, which slows down the speed of vertex state propagation.

Based on this observation, HUS-Graph provides an immediate propagation-based vertex update scheme to accelerate the vertex state propagation and convergence speed of traversal-based algorithms. In this model, HUS-Graph applies the vertex updates immediately after receiving the update messages. Fig. 8 shows the executions of ROP and COP in the immediate propagation-based vertex update scheme with the example graph in Fig. 4. Here, we only maintain one copy of vertex values for each interval, which means that the source intervals work as the destination intervals at the same time. Therefore, the update messages are propagated to the destination vertices instantly with no need for vertex values synchronization, and the newly updated vertices can propagate update messages to their destination vertices in the subsequent processing in the same iteration. For example, when processing in-block(1, 1) in COP, as shown in Fig. 8b, vertex 1 that is newly updated by vertex 2 and vertex 4 is treated as a newly activated vertex from which vertex 4 and vertex 5 pull updates in the subsequent processing. While in the accumulation-based vertex update scheme as shown in Fig. 7a, vertex 4 and vertex 5 can only read vertex 1's value of the last iteration, which slows down the vertex state propagation and convergence speed of algorithms. The first source interval S1 and the first

destination interval D1 are identical physically and will be read and updated simultaneously, although they are logically separated.

3.7 I/O-Based Performance Prediction Method

The key to gain optimal performance is to select the fastest update model between ROP and COP. Moreover, unlike shared-memory systems [10] or distributed systems [19] whose performance depend on CPU performance or communication cost, the major performance bottleneck of out-of-core systems is I/O cost. HUS-Graph proposes a simple I/O-based performance prediction method that enables the system to dynamically select the optimal update model based on the I/O load of the current iteration.

The I/O-based Performance Prediction Method estimates and compares the I/O cost of ROP and COP when executing a vertex interval to guide the system to select the optimal update model. According to the I/O cost analysis in Section 3.4, the I/O traffics of vertex values and vertex indices for both ROP and COP are almost identical, except for the vertex values writes that have a slight impact on the I/O performance. Therefore, if $C_{rop} \leq C_{cop}$, we have

$$\frac{|A_i| \times M}{T_{rr}} \leq \frac{\frac{|V|}{P} \times M}{T_{sr}}.$$

It can be seen that

$$\frac{|A_i| \times P}{|V|} \leq \frac{T_{rr}}{T_{sr}},$$

The parameters A_i, $|V|$ and P can all be collected and computed in the runtime. Furthermore, the disk access throughput T_{rr} and T_{sr} can be measured by using several measurement tools such as fio [6] before we conduct the experiments. This provides an accurate performance estimate that enables the system to select the proper update model.

To reduce the extra computational overhead, HUS-Graph calculates and compares C_{rop} and C_{cop} just when the number of active vertices is less than a user-modified threshold α. In our experiments, α is empirically set to 5 percent of all vertices. If the number of active vertices is larger than α, HUS-Graph selects COP model regardless of the current active vertices.

4 Evaluation

In this section, we present experimental evaluation of our system HUS-Graph in comparison with state-of-the-art out-of-core graph processing systems.

4.1 Experiment Setup

Platform and Benchmarks. The hardware platform used in our experiments is a 16-core commodity machine equipped with 16 GB main memory and 500 GB 7200 RPM HDD, running Ubuntu 16.04 LTS. In addition, a 128 GB SATA2 SSD is installed to evaluate the scalability. The program is compiled with gcc version 5.4.0.

The datasets for our evaluation are all real-world graphs with power-law degree distributions, summarized in Table 2. LiveJournal, Twitter2010, Friendster and SK2005 are social graphs, showing the relationship between users within each online social network. UK2007 and UKunion are web graphs that consist of hyperlink relationships between web pages, with larger diameters than social graphs. The in-memory graph LiveJournal is chosen to evaluate the scalability of HUS-Graph. The other five graphs are respectively 1.5x, 1.9x, 2.1x, 3.9x and 6.1x larger than available memory.

TABLE 2
Datasets Used in Evaluation

Dataset	Vertices	Edges	Type
LiveJournal [8]	4.8 million	69 million	Social Graphs
Twitter2010 [20]	42 million	1.5 billion	Social Graphs
Friendster [16]	66 million	1.8 billion	Social Graphs
SK2005 [16]	51 million	1.9 billion	Social Graphs
UK2007 [21]	106 million	3.7 billion	Web Graphs
UKunion [21]	133 million	5.5 bilion	Web Graphs

The benchmarks algorithms used in our evaluation include three traversal-based graph algorithms, Breadth-first search (BFS), Weak Connected Components (WCC), and Single Source Shortest Path (SSSP), and a representative sparse matrix multiplication algorithm known as PageRank. BFS, WCC and SSSP vary the number of active vertices in different iterations and can effectively evaluate the hybrid update strategy of HUS-Graph. We run these algorithms until convergence. For PageRank, all vertices are always active as each vertex receives messages from its neighbors to compute the new rank value in all iteration. We run five iterations of PageRank on each dataset. In addition, we use the accumulation-based vertex update scheme for PageRank and use the immediate propagation-based vertex update scheme for other three traversal-based graph algorithms.

Systems for Comparison. We compare HUS-Graph with three baseline out-of-core graph processing systems. One is the first version of HUS-Graph (we name it as HUS-Graph-v1) [12]. We compare HUS-Graph against HUS-Graph-v1 to evaluate the effects of the newly proposed optimizations including the space-efficient storage format and immediate propagation-based vertex update scheme. The other two systems for comparison are GraphChi [5] and GridGraph [7]. GraphChi is an extensively-used out-of-core graph processing system that supports vertex-centric scatter-gather computation model. It exploits a novel parallel sliding windows (PSW) method to minimize random disk accesses. GridGraph uses a 2-Level hierarchical partition and a streaming-apply model to reduce the amount of data transfer, enable streamlined disk access, and maintain locality. For all compared systems, we provide 16 execution threads for the executions of all algorithms.

Evaluation Methodology. We first explore the effects of our system designs including hybrid update strategy, I/O-based performance prediction method, space-efficient storage format and immediate propagation-based vertex update scheme. Then, we compare HUS-Graph against other systems on runtime of algorithms and I/O traffic. Finally, we evaluate the scalability of HUS-Graph by observing the improvement when more hardware resource is added.

4.2 Effect of Hybrid Update Strategy

Fig. 9 shows the effect of the hybrid update strategy of HUS-Graph, by comparing the Hybrid model that adaptively

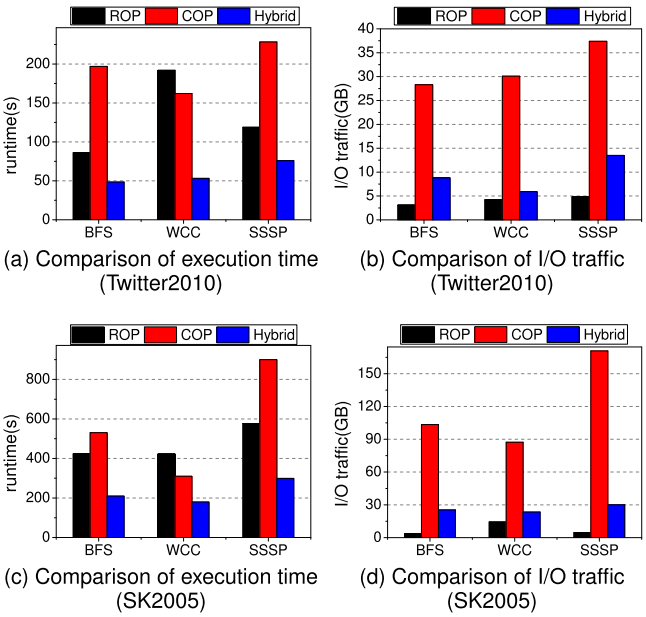


Fig. 9. Comparison of different update strategies.

switches between ROP and COP with two baseline approaches that respectively implement ROP and COP in each iteration. Figs. 9a and 9c shows the comparisons of runtime with different models on Twitter2010 and SK2005. Figs. 9b and 9d shows the corresponding comparisons of I/O traffic. As we see from the results, the Hybrid model always achieves the best performance, as it selects the optimal I/O model and update model for each vertex interval in each iteration. For BFS and SSSP where the number of active vertices is small in most iteration, COP has the worst performance as it loads the entire edges in each iteration. For WCC where most vertices are active in the first few iterations, ROP has the worst performance due to the significant overheads of random disk accesses.

As to I/O traffic, ROP enables selective data access based on the active vertices and accesses the least amount of data for all algorithms. COP streams the whole of data to achieve good I/O access locality. Thus, it accesses the most amount of data. Based on the I/O-based performance prediction method, the Hybrid model dynamically selects between ROP and COP. Therefore, the I/O traffic for the Hybrid model is moderate.

Theoretically, only using ROP can also accommodate different I/O and computation loads of graph algorithms. For sparse active edge sets, ROP only loads the active edges to skip useless data. For dense active edge sets, ROP streams

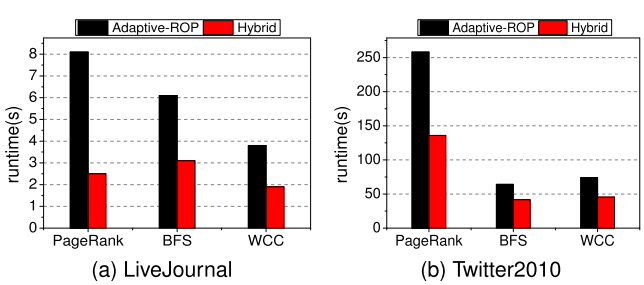


Fig. 10. Adaptive-ROP vs. hybrid.

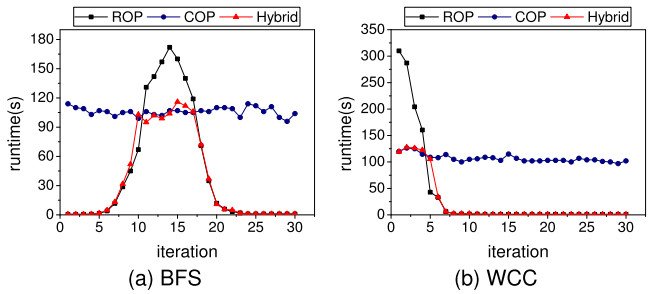


Fig. 11. Effect of the I/O-based performance prediction model.

all edges to avoid random disk accesses. The switch can be easily achieved by using our I/O-based performance prediction method (We referred this execution model as Adaptive-ROP). However, we take COP to handle dense active edge sets for better computational efficiency as follows. First, COP adopts a pull-style update model that can avoid the expensive atomic operations and make better use of parallelism especially when the number of active edges is large, as described in Section 2.2. Second, as many studies [10], [11] have shown, the pull-style update (bottom-up approach) can significantly reduce accesses to vertices that have already been visited and improve the performance of traversal-based algorithms. To further demonstrate the superiority of our Hybrid model (ROP+COP), we compare the performance of these two methods in Fig. 10. Thanks to the better computational efficiency, Hybrid model outperforms Adaptive-ROP in all cases. Specifically, for PageRank, BFS and WCC, Hybrid model respectively outperforms Adaptive-ROP by 2.6x, 1.6x and 1.8x. Actually, the more dense the active edge sets (like PageRank), the more superiority Hybrid has, due to the data contention when updating vertices faced by ROP. On the other hand, when the graph becomes larger, the superiority of Hybrid model becomes smaller due to the increasing disk I/O costs. Even though, Hybrid model still outperforms Adaptive-ROP by up to 1.9x for the Twitter graph.

4.3 Effect of I/O-Based Performance Prediction Method

To evaluate the effectiveness of the I/O-based performance prediction method, we run two algorithms (BFS and WCC) that exhibit different I/O features on Ukunion with three different update models, ROP, COP and Hybrid, and report the runtime of different models in each iteration (30 iterations) in Fig. 11. The two algorithms produce different numbers of active vertices in each iteration, which is appropriate to evaluate the accuracy of the the I/O-based performance prediction method.

As shown in Fig. 11, the performance gap between ROP and COP is quite significant. The performance of COP is relatively constant since it loads all graph data in per iteration, while the performance of ROP depends on the number active vertices. For BFS, ROP outperforms COP in most iterations, except in few iterations (iteration 11 $\sim$ 17) where there are a large number of active vertices that cause frequent random disk accesses. For WCC, COP performs better in the first few iterations when most vertices remain active, while ROP performs better when many vertices reach convergence. HUS-Graph is able to select the optimal update model based on the I/O-based performance prediction method in most iterations,

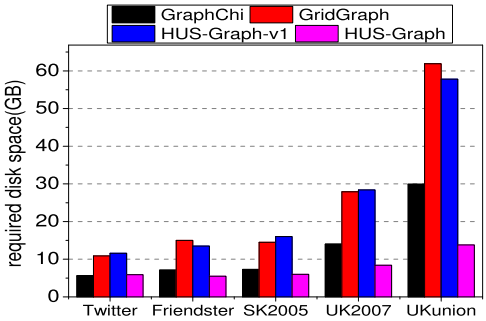


Fig. 12. Comparison of disk space consumption.

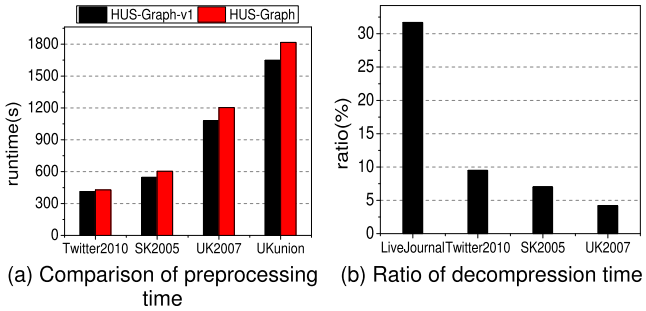


Fig. 14. Evaluating the overheads of space-efficient storage format.

except iteration 10 of BFS and iteration 5 of WCC. These wrong predictions are usually around the intersection of the ROP′ and COP′ performance curves. This indicates that we can implement a more accurate and fine-grained performance evaluation and prediction method to find the critical point where the update strategy switches between ROP and COP.

4.4 Effect of Space-Efficient Storage Format

Fig. 12 compares the required disk space of HUS-Graph with GraphChi, GridGraph and HUS-Graph-v1. Note that we only report the storage size of edges as the storage sizes of vertices and degrees are negligible. Compared with HUS-Graph-v1, HUS-Graph that incorporates the space-efficient storage format can reduce disk consumption by up to 71 percent. For GraphChi and GridGraph, they respectively store the edges in CSR and edge list format. Although they only maintain one copy of edges, the storage usages of them are 1.5x and 2.8x larger than that of HUS-Graph on average. This further shows the efficiency of the space-efficient storage format that adopts several graph compression methods.

Fig. 13 shows the required disk space and runtime when we store all the graph data without any compression (Base), using compression of undirected graph (Undirected), and when using all compression methods (All). We choose the undirected Friendster graph to conduct this experiment since it can efficiently evaluate the the effects of different compression methods. We can see that the Undirected compression can halve the storage size but has no impact on performance. This is because HUS-Graph loads either the in-edges or the out-edges for each vertex interval, and only reducing the redundant storage of edges can not reduce the I/O traffic and improve performance. When using all compression methods, the storage sizes of both in-edges and out-edges reduce, leading to reduced I/O traffic and improved algorithmic performance.

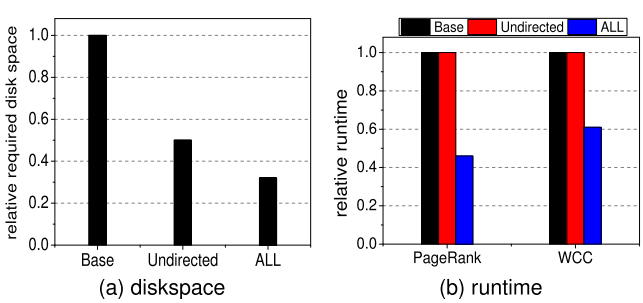


Fig. 13. Evaluating the effects of different compression methods.

We also evaluate the overheads of space-efficient storage format as it involves compression and decompression of graphs. The overheads include two aspects: more preprocessing (compression) time and extra decompression time. The decompression time refers to the time spent on transforming the compressed adjacency lists into initial adjacency lists that the system can directly process. Fig. 14 shows the evaluation results. In Fig. 14a, we compare the preprocessing time of HUS-Graph and HUS-Graph-v1 to evaluate the compression overheads. Due to the using of graph compression techniques, the preprocessing time of HUS-Graph is longer than that of HUS-Graph-v1 by 4.1-11.6 percent. Considering the benefits that the space-efficient storage format brings, these extra overheads in preprocessing are acceptable. Moreover, the graphs can be reused for many times after preprocessing. Fig. 14b shows the percentage of decompression time in overall runtime when running PageRank. The ratio of decompression time ranges from 4.2 to 31.7 percent. We can see that when the graph is larger, the ratio of decompression time is lower, since the disk I/O time dominates the overall runtime. Therefore, the space-efficient storage format may not work well for small graphs but can bring significant benefits for large graphs. In addition, the decompression procedure can be overlapped with vertex updating to further reduce the extra overheads.

4.5 Effect of Immediate Propagation-Based Vertex Update Scheme

We conduct our evaluation on the traversal-based algorithms (BFS, WCC and SSSP) with two vertex update schemes, accumulation-based vertex update (AVU) and immediate propagation-based vertex update (IPVU) schemes. Fig. 15 shows the evaluation results. With the use of the IPVU model, the performance of algorithms can improve by up to 39 percent,

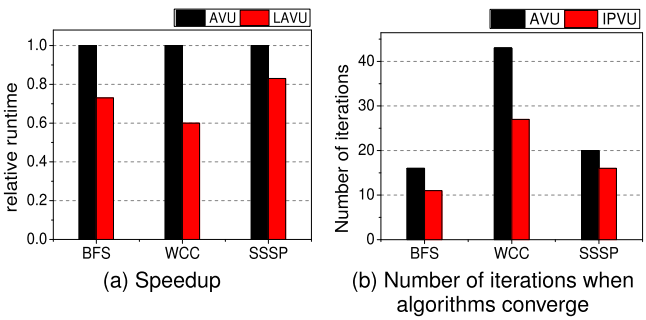


Fig. 15. Evaluating the benefits of immediate propagation-based vertex update scheme.

TABLE 3
Execution Time (in Seconds)

	Data sets	LiveJournal	Twitter2010	Friendster	SK2005	UK2007	UKunion
BFS	GraphChi	20.9	1624.3	2294.5	4973.6	7154.5	24062.3
	GridGraph	5.2	598.9	578.6	4066.3	6025.2	18929.2
	HUS-Graph-v1	3.9	70.6	133.9	424.5	1278.2	1897.9
	HUS-Graph	3.1	41.6	54.8	172.9	737.5	874.2
WCC	GraphChi	24.4	913.7	2612.3	2769.1	6862.8	15665.8
	GridGraph	5.1	522.5	526.8	3338.7	4783.8	13265.1
	HUS-Graph-v1	3.5	74.8	103.5	289.2	1068.3	1223.6
	HUS-Graph	1.9	45.6	44.9	113.9	674.1	734.2
SSSP	GraphChi	21.4	1913.9	1802.4	5415.8	11495.8	56650.9
	GridGraph	6.1	660.4	708.6	4180.7	7029.4	25554.2
	HUS-Graph-v1	4.5	106.8	164.8	605.3	1750.7	2997.9
	HUS-Graph	5.3	56.9	75.1	319.1	924.8	1461.4
PageRank	GraphChi	16.6	928.6	2562.8	970.3	2774.8	3376.6
	GridGraph	10.9	451.9	1009.4	669.1	1242.2	1829.3
	HUS-Graph-v1	2.2	230.3	349.3	291.3	600.5	922.9
	HUS-Graph	2.5	135.8	143.5	148.7	398.3	668.6

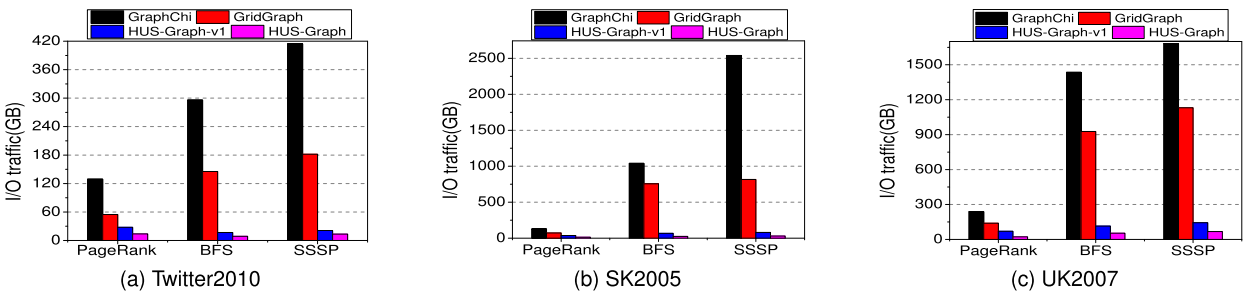


Fig. 16. I/O traffic comparison.

as shown in Fig. 15a. Fig. 15b explains why the IPVU model can improve the performance. It accelerates the vertex states propagation and convergence speed of traversal-based algorithms, reducing the number of iterations. The less iterations mean less computation overheads and unnecessary I/Os when running the algorithms.

4.6 Comparison to Other Systems

We report the execution time of the chosen algorithms on different datasets and systems in Table 3. The execution time includes the time of graph loading and vertices updating. We can see that HUS-Graph achieves a significant speedup over GraphChi and GridGraph. Specifically, HUS-Graph outperforms GraphChi by 4.1x-58.2x and GridGraph by 1.2x-29.3x.

GraphChi utilizes the vertex-centric scatter-gather processing to maximize sequential disk access. However, it writes a large amount of intermediate updates to disk, which incurs great I/O overheads. In addition, it needs a subgraph construction phase to construct the in-memory vertex-centric data structure, which is a time-consuming process [6]. GridGraph combines the scatter and gather phases into one streaming-apply phase to avoid writing the intermediate results to disk. While both GraphChi and GridGraph support the selective scheduling to reduce the loading of inactive edges, it is very coarse-grained since it can only skip processing the edge blocks without any active edges. This means it loads and processes an edge-block even though there is only one active edge. Therefore, these systems still load many useless data, which is wasteful for the traversal-based algorithms that only have a small number of active vertices in most iterations. And that is just the greatest strength for HUS-Graph that enables selective data access to avoid loading useless data. For the three traversal-based algorithms BFS, WCC, and SSSP, HUS-Graph respectively outperforms GraphChi and GridGraph by 23.9x and 12.3x on average. For the PageRank algorithm where all vertices are always active, HUS-Graph implements COP model and loads the whole of data in each iteration like other systems. Thanks to more compact storage that leads to less amount of I/O, HUS-Graph remains outperforming GraphChi and GridGraph by 8.5x and 3.9x respectively.

When compared with HUS-Graph-v1, the use of the space-efficient storage format and immediate propagation-based vertex update scheme can improve the performance by up to 2.5x. In addition, we can observe that these newly proposed optimizations bring few benefits when processing the small graph LiveJournal. This is attributed to the high decompression overhead as shown in Section 4.4.

We compare the amount of I/O traffic of HUS-Graph versus Graphchi, GridGraph and HUS-Graph-v1 in Fig. 16. For PageRank, the I/O traffic of HUS-Graph is respectively 9.8x and 4.2x smaller than that of GraphChi and GridGraph. This is mainly attributed to HUS-Graph's more compact

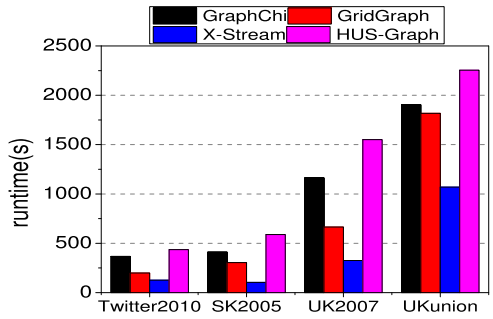


Fig. 17. Preprocessing time for different systems.

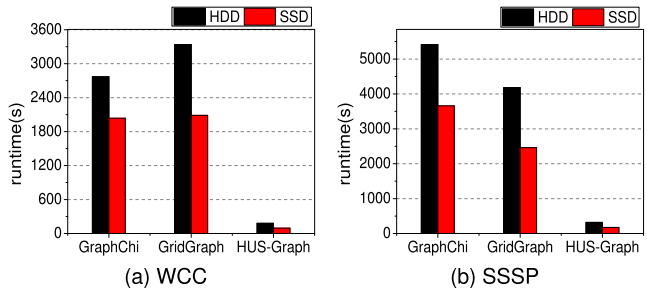


Fig. 19. Effect of I/O devices on performance.

storage format. In addition, GraphChi has to write a large amount of intermediate data (edge values) to disk for subsequent computation, while GridGraph and HUS-Graph only writes vertex values back to disk during the computation. For the two traversal-based algorithms BFS and SSSP, the I/O traffic of HUS-Graph is respectively 39.2x and 20.1x smaller than that of GraphChi and GridGraph, thanks to the combined effort of selective data access and space-efficient storage format. Compared with HUS-Graph-v1, the use of the space-efficient storage format and immediate propagation-based vertex update scheme can reduce I/O traffic by up to 68 percent, since the more compact storage and fewer number of iterations can lead to more smaller amount of I/O traffic.

We also evaluate the processing overhead (preprocessing time) of different systems. The preprocessing procedure converts an input graph into an internal form on the disk, which consists of loading raw data into memory, partitioning and building the graph. In addition to comparing with GraphChi and GridGraph, we also include the preprocessing time of X-Stream [6] that adopts an edge-centric model. As shown in Fig. 17, X-Stream takes the least preprocessing time since it does not need to sort edge lists during preprocessing [6]. It only shuffles the original edge lists to several files according to the streaming partitions [7], [22]. However, this partitioning strategy makes it inefficient for selective scheduling, which largely affects its performance on many algorithms where only a portion of the vertices are used in some iterations [7]. HUS-Graph takes more preprocessing time than other systems, since it needs to build two copies of edges and implement the space-efficient storage format. However, the overhead of the extra preprocessing is more than offset by the significant performance improvement it brings. For example, the space-efficient storage format can greatly reduce I/O traffic, leading to improved algorithmic performance. Moreover, the graphs can be reused for many times after preprocessing, and the preprocessing overheads can be significantly amortized.

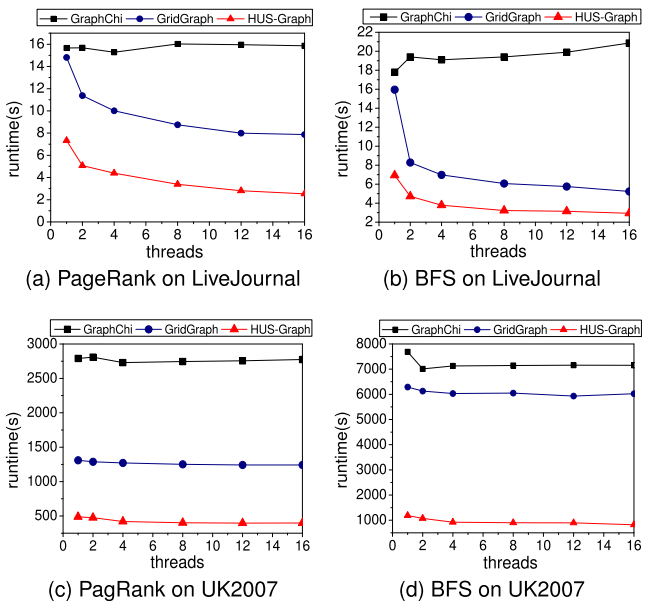


Fig. 18. Effect of the number of thread on performance.

4.7 Scalability

We evaluate the scalability of HUS-Graph by observing the improvement when more hardware resource is added. Fig. 18 shows the effect of the number of threads on system performance. To access the effect of the parallel processing on different algorithms and graphs, we choose two graph algorithms with different computation and I/O features to run on two graphs with different sizes. For the relatively small graph LiveJournal whose data can completely fit in memory, the degree of parallelism has significant impact on system performances of HUS-Graph and GridGraph due to the efficient use of parallelism. Moreover, both the performances of PageRank and BFS improve as the number of threads increases. However, GraphChi shows poor scalability as we increase the number of threads. The main blame is GraphChi's deterministic parallelism that limits the utilization of multi-threads [5]. On the other hand, for the large graph UK2007, system performance is limited by disk I/O. Therefore, thread number has relatively less impact on the performances of the three systems.

Fig. 19 shows the performance improvement of WCC and SSSP on SK2005 when using different I/O devices. Compared with disk cases, GraphChi, GridGraph and HUS-Graph achieve an average speedup of 1.4x, 1.6x and 1.9x respectively when using SSD for WCC and SSSP. This indicates that HUS-Graph can benefit more from the utilization of SSD, since HUS-Graph enables selective (random) data access to load the active edges, which works well on SSD.

5 Related Work

Many scalable graph processing systems have recently been proposed. In this section, we introduce three categories of existing graph processing systems: distributed systems, single-machine shared-memory systems and single-machine disk-based (out-of-core) systems.

5.1 Distributed Systems

Distributed systems usually require to hold the whole input graph and all intermediate results and messages in memory. PEGASUS [23] and GBase [24] are based on MapReduce, which works relatively well for graph algorithms such as PageRank, but performs poorly for graph traversal algorithms. Pregel [1] supports vertex-centric computing model following Bulk-Synchronous Parallel (BSP) message passing model [25]. It abstracts away the complexity of programming in a distributed-memory environment and runs users' code in parallel on a cluster. However, this model usually suffers from expensive synchronization overheads. GraphLab [26] and PowerGraph [2] executes an asynchronous model and uses shared memory for communication among vertices instead of passing messages. Gemini [4] applies multiple optimizations targeting computation performance to build scalability on top of efficiency. Chaos [27], BlitzG [28] and DD-Graph [29] utilize secondary storage to scale distributed graph processing to out-of-core scenery. Gluon [30] introduces a new approach to build distributed memory graph analytics systems that exploit heterogeneity in processor types (CPU and GPU), partitioning policies and programming models.

5.2 Single-Machine Shared-Memory Systems

Single-machine shared-memory systems typically use a high-end server with hundreds or thousands gigabytes of DRAM to hold the whole graph. Ligra [11] is a lightweight shared-memory framework and provides a programming interface optimized for graph traversal algorithms. Ligra+ [31] integrates compression techniques such delta compression into Ligra. Unlike HUS-Graph that compresses the graphs to save the disk space and further reduce I/O traffic, it uses the compression techniques to enable faster in-memory parallel graph processing using less memory footprints. Polymer [32] is a NUMA-aware graph analytics system, which is motivated by a detailed study of NUMA characteristics. Medusa [33] and Garaph [34] fully exploit the power of modern hardware and efficiently support GPU-accelerated graph processing. GraphIt [35] is a novel DSL for graph processing that generates fast implementations for algorithms with different performance characteristics running on graphs with varying sizes and structures.

5.3 Single-Machine Disk-Based Systems

Out-of-core graph processing systems enable users to analyze, process and mine large graphs in a single PC by efficiently using disks. GraphChi [5] is a pioneering single-PC-based out-of-core graph processing system that supports vertex-centric computation and is able to express many graph algorithms. By using a novel parallel sliding windows method to reduce random I/O accesses, GraphChi is able to process large-scale graphs in reasonable time.

Following GraphChi, a number of out-of-core graph processing systems are proposed to improve the I/O performance. X-Stream [6] uses an edge-centric approach in order to minimize random disk accesses. In each iteration, it streams and processes the entire unordered list of edges during the scatter phase and applies updates to vertices in the gather phase. GridGraph [7] combines the scatter and gather phases into one streaming-apply phase and uses a 2-Level hierarchical partition to break graph into 1D-partitioned vertex chunks and 2D-partitioned edge blocks. It avoids writing updates to disk and enables selective scheduling to skip the inactive edge blocks. VENUS [14] explores a vertex-centric streamlined computing model that enables streams the graph data while performing computation. However, all these systems improve the I/O access locality at the expense of loading all graph data in all iterations, even though a large amount of data is not needed.

GraphQ [36] and Wonderland [37] use abstraction to accelerate graph processing. GraphQ improves the performance of graph queries when queries can be answered using only a few sub-graphs. Wonderland extracts graph abstractions to capture certain graph properties, and then performs abstraction-guided processing to infer better priority processing order and faster information propagation. While this abstraction-based method can accelerate convergence and reduce disk I/O, its scope of applications is limited to path-based monotonic graph algorithms.

Several systems use fast storage devices to accelerate out-of-core graph processing. FlashGraph [38] and Graphene [39] utilize SSD arrays and implement a semi-external memory graph engine to close the performance gap between in-memory and out-of-core graph processing. GraFBoost [40] adopts a sort-reduce accelerator to improve the I/O performance of flash storage. V-Part [41] extends GraFBoost by using a novel to vertex partition scheme to alleviate extra computation overheads. Compared with these systems, HUS-Graph has better adaptability and works well for both SSD and HDD.

[9] removes unnecessary I/O of out-of-core graph processing by employing dynamic partitions whose layouts are dynamically adjustable. These dynamic partitions compactly capture the set of active edges in current iteration. It achieves this by dropping inactive edges across iterations and delaying computations that cannot be performed due to missing edges. Although [9] can avoid the loading of useless data and eliminate random disk accesses, it has to write back the active edges. LUMOS [42] is a dependency-driven out-of-core graph processing system. It performs out-of-order execution to proactively propagate values across iterations while simultaneously providing synchronous processing guarantees.

6 Conclusion

In this paper, we present an I/O-efficient out-of-core graph processing system called HUS-Graph that maximizes the I/O performance by achieving a good balance between I/O traffic and I/O access locality. HUS-Graph adopts a hybrid update strategy including Row-oriented Push (ROP) and Column-oriented Pull (COP), to schedule disk I/O adaptively according to running features of graph algorithms. Furthermore, HUS-Graph adopts a locality-optimized dual-block graph representation to organize the graph data and an I/O-based performance prediction method that enables the system to dynamically select the optimal update model based on the I/O loads of current iteration. To save the disk space and further reduce I/O traffic, HUS-Graph implement a space-efficient storage format by combining several graph compression

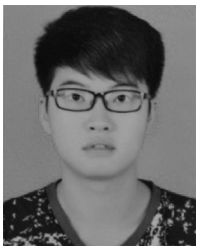
techniques. Our evaluation results show that HUS-Graph can be much faster than GraphChi and GridGraph, two state-of-the-art out-of-core systems.

Acknowledgments

This work was supported by National Defense Preliminary Research Project No. 31511010202, NSFC No. 61832020, 61772216, 61821003, and U1705261, Wuhan application basic research Project No. 2017010201010103, Hubei province technical innovation special Project No. 2017AAA129, and Fundamental Research Funds for the Central Universities. This work was also supported by the Open Project Program of Wuhan National Laboratory for Optoelectronics No. 2018WNLOKF006.

References

[1] G. Malewicz *et al.*, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.
[2] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 17–30.
[3] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 599–613.
[4] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 301–316.
[5] A. Kyrola, G. E. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a PC," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 31–46.
[6] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proc. ACM Symp. Operating Syst. Princ.*, 2013, pp. 472–488.
[7] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2015, pp. 375–386.
[8] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: Membership, growth, and evolution," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2006, pp. 44–54.
[9] K. Vora, G. Xu, and R. Gupta, "Load the edges you need: A generic I/O optimization for disk-based graph processing," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2016, pp. 507–522.
[10] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," *Sci. Program.*, vol. 21, no. 3–4, pp. 137–148, 2013.
[11] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," *ACM SIGPLAN Notices*, vol. 48, no. 8, pp. 135–146, 2013.
[12] X. Xu, F. Wang, H. Jiang, Y. Cheng, D. Feng, and Y. Zhang, "HUS-graph: I/O-efficient out-of-core graph processing with hybrid update strategy," in *Proc. 47th Int. Conf. Parallel Process.*, 2018, Art. no. 3.
[13] Z. Wang, Y. Gu, Y. Bao, G. Yu, and J. X. Yu, "Hybrid pulling/pushing for I/O-efficient distributed and iterative graph computing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2016, pp. 479–494.
[14] J. Cheng, Q. Liu, Z. Li, W. Fan, J. C. Lui, and C. He, "VENUS: Vertex-centric streamlined graph computation on a single PC," in *Proc. IEEE Int. Conf. Data Eng.*, 2015, pp. 1131–1142.
[15] W.-S. Han *et al.*, "TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2013, pp. 77–85.
[16] P. Boldi and S. Vigna, "The webgraph framework I: Compression techniques," in *Proc. Int. Conf. World Wide Web*, 2004, pp. 595–602.
[17] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu, "TripleBit: A fast and compact system for large scale RDF data," *Proc. VLDB Endowment*, vol. 6, pp. 517–528, 2013.
[18] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, and H. Yang, "NXgraph: An efficient graph processing system on a single machine," in *Proc. IEEE Int. Conf. Data Eng.*, 2016, pp. 409–420.
[19] Y. Cheng *et al.*, "A communication-reduced and computation-balanced framework for fast graph computation," *Front. Comput. Sci.*, vol. 12, no. 5, pp. 887–907, 2018.
[20] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *Proc. Int. Conf. World Wide Web*, 2010, pp. 591–600.
[21] P. Boldi, M. Santini, and S. Vigna, "A large time-aware web graph," *ACM SIGIR Forum*, vol. 42, no. 2, pp. 33–38, 2008.
[22] P. Sun, Y. Wen, D. Ta, and X. Xiao, "GraphMP: I/O-efficient big graph analytics on a single commodity machine," *IEEE Trans. Big Data*, to be published, doi: 10.1109/TBDATA.2019.2908384.
[23] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A peta-scale graph mining system implementation and observations," in *Proc. IEEE Int. Conf. Data Mining*, pp. 229–238.
[24] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos, "GBASE: A scalable and general graph management system," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2011, pp. 1091–1099.
[25] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.
[26] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endowment*, vol. 5, pp. 716–727, 2012.
[27] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: Scale-out graph processing from secondary storage," in *Proc. ACM Symp. Operating Syst. Princ.*, 2015, pp. 410–424.
[28] Y. Cheng *et al.*, "Using high-bandwidth networks efficiently for fast graph computation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 5, pp. 1170–1183, May 2019.
[29] Y. Cheng *et al.*, "A highly cost-effective task scheduling strategy for very large graph computation," *Future Gener. Comput. Syst.*, vol. 89, pp. 698–712, 2018.
[30] R. Dathathri *et al.*, "Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2018, pp. 752–768.
[31] J. Shun, L. Dhulipala, and G. E. Blelloch, "Smaller and faster: Parallel processing of compressed graphs with Ligra+," in *Proc. Data Compression Conf.*, 2015, pp. 403–412.
[32] K. Zhang, R. Chen, and H. Chen, "NUMA-aware graph-structured analytics," *ACM SIGPLAN Notices*, vol. 50, no. 8, pp. 183–193, 2015.
[33] J. Zhong and B. He, "Medusa: Simplified graph processing on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1543–1552, Jun. 2014.
[34] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai, "Garaph: Efficient GPU-accelerated graph processing on a single machine with balanced replication," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2015, pp. 195–207.
[35] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: A high-performance graph DSL," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, 2018, Art. no. 121.
[36] K. Wang, G. Xu, Z. Su, and Y. D. Liu, "GraphQ: Graph query processing with abstraction refinement—Scalable and programmable analytics over very large graphs on a single PC," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2015, pp. 387–401.
[37] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen, "Wonderland: A novel abstraction-based out-of-core graph processing system," in *Proc. 23rd Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2018, pp. 608–621.
[38] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "FlashGraph: Processing billion-node graphs on an array of commodity SSDs," in *Proc. USENIX Conf. File Storage Technol.*, 2015, pp. 45–58.
[39] H. Liu and H. H. Huang, "Graphene: Fine-grained IO management for graph computing," in *Proc. USENIX Conf. File Storage Technol.*, 2017, pp. 285–300.
[40] S.-W. Jun, A. Wright, S. Zhang, S. Xu, and Arvind, "GraFboost: Using accelerated flash storage for external graph analytics," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit.*, 2018, pp. 411–424.
[41] N. Elyasi, C. Choi, and A. Sivasubramaniam, "Large-scale graph processing on emerging storage devices," in *Proc. USENIX Conf. File Storage Technol.*, 2019, pp. 309–316.
[42] K. Vora, "LUMOS: Dependency-driven disk-based graph processing," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2019, pp. 429–442.



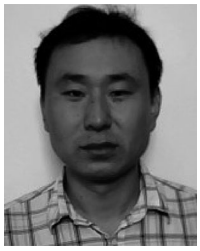
Xianghao Xu received the BE degree in computer science and technology from the LiaoNing University, ShenYang, China, in 2015. He is currently working toward the PhD degree majoring in computer architecture at the Huazhong University of Science and Technology, Wuhan, China. His current research interests include computer architecture and graph processing.

Fang Wang received the BE and master's degrees in computer science from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 1994 and 1997, respectively, and the PhD degree in computer architecture from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2001, where she is currently a professor of computer science and engineering. Her research interests include distribute file systems, parallel I/O storage systems, and graph processing systems. She has more than 50 publications in major journals and conferences, including the *Future Generation Computing Systems*, the *ACM Transactions on Architecture and Code Optimization*, HiPC, ICDCS, HPDC, and ICPP.

Hong Jiang (Fellow, IEEE) received the BE degree from the Huazhong University of Science and Technology, Wuhan, China, in 1982, the MASc degree from the University of Toronto, Toronto, Canada, in 1987, and the PhD degree from the Texas A&M University, College Station, Texas, in 1991. He is Wendell H. Nedderman endowed professor and chair of the Department of Computer Science and Engineering, University of Texas at Arlington. His research interests include computer architecture, computer storage systems, and parallel/ distributed computing. He serves as an associate editor of the *IEEE Transactions on Parallel and Distributed Systems*. 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*, *ACM Transactions on Storage*, *ACM Transactions on Architecture and Code Optimization*, *Journal of Parallel and Distributed Computing*, ISCA, MICRO, FAST, USENIX ATC, USENIX LISA, SIGMETRICS, MIDDLEWARE, ICDCS, IPDPS, OOPLAS, ECOOP, SC, ICS, HPDC, and ICPP.

Yongli Cheng received the BE degree from the Chang'an University, Xi'an, China, in 1998, the MS degree from the FuZhou University, FuZhou, China, in 2010, and the PhD degree from the Huazhong University of Science and Technology, Wuhan, China, 2017. He is currently a teacher of the College of Mathematics and Computer Science, FuZhou University currently. His current research interests include computer architecture and graph computing. He has several publications in major international conferences and journals, including HPDC, IWQoS, INFOCOM, ICPP, the *Future Generation Computing Systems*, *IEEE/ACM Transactions on Networking*, and *Frontiers of Computer Science*.

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), Wuhan, China, in 1991, 1994, and 1997, respectively. She is currently a professor and 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 international conferences, including the *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, *ACM Transactions on Storage*, *Journal of Computer Science and Technology*, FAST, USENIX ATC, ICDCS, HPDC, SC, ICS, IPDPS, and ICPP. She serves on the program committees of multiple international conferences, including SC 2011, 2013, and MSST 2012.



Yongxuan Zhang received the BE degree in computer science and technology from the Nanchang Hangkong University, Nanchang, China, in 2005. He is currently working toward the PhD degree majoring in computer science and technology at the Huazhong University of Science and Technology, Wuhan, China. His current research interests include graph processing and parallel/distributed processing.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Power Guarantee for Electric Systems Using Real-Time Scheduling

Eugene Kim, Youngmoon Lee, *Student Member, IEEE*, Liang He, *Senior Member, IEEE*, Kang G. Shin, *Life Fellow, IEEE*, and Jinkyu Lee, *Member, IEEE*

Abstract—Modern electric systems, such as electric vehicles, mobile robots, nano satellites, and drones, require to support various power-demand operations for user applications and system maintenance. This, in turn, calls for advanced power management that jointly considers power demand by the operations and power supply from various sources, such as batteries, solar panels, and supercapacitors. In this article, we develop a power scheduling framework for a reliable energy storage system with multiple power-supply sources and multiple power-demand operations. Specifically, we develop *offline power-supply guarantee analysis* and *online power management*. The former provides an offline power-supply guarantee such that every power-demand operation completes its execution in time while the sum of power required by individual operations does not exceed the total power supplied by the entire energy storage system at any time; to this end, we develop a plain power-supply analysis as well as its improved version using real-time scheduling techniques. On the other hand, the latter efficiently utilizes the surplus power available at runtime for improving system performance; we propose two approaches, depending on whether future scheduling information of power-demanding tasks is available or not. For evaluation, we perform simulations to evaluate both the plain and improved analyses for offline power guarantee under various synthetic power-demand operations. In addition, we have built a simulation model and demonstrated that the proposed framework with the offline analysis and online management not only guarantees the required power-supply, but also enhances system performance by up to 56.49 percent.

Index Terms—Offline power-supply guarantee, online power management, real-time scheduling, electric systems

1 Introduction

Recent advances on cyber-physical systems (CPS) require various mechanical/electrical operations for many physical platforms, such as electric vehicles, mobile robots, nano satellites, and drones. For example, a drone needs to (i) operate the motors to fly, (ii) power sensors, coolers, and communication modules, (iii) operate camera(s) and stepper motors to take pictures and deliver parcels, and (iv) perform the computation necessary to maintain its stability (see Fig. 13 in the supplement, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2020.2977041.). These *multiple power-demand operations* impose different power demands on the system and may be triggered at different times, requiring the system to effectively provide time-varying supply of power [1], [2], [3], [4], [5], [6]. Existing studies on the power scheduling problem focused on the reduction of peak power by scheduling multiple power-demand operations and analysis thereof [7], [8], [9], [10].

However, a complete solution to the power scheduling problem also needs to characterize energy storages/sources because it affects power capability for the power-demand operations. We address this need by targeting hybrid energy storage systems (HESSes) comprised of *multiple power-supply sources and storages*, such as batteries, supercapacitors, and renewable energy sources, whose concept and implementation have been proposed and explored in [4], [11], [12], [13], [14], [15], [16]. A combination of high-energy-density batteries and high-power-density supercapacitors enables the system to supply the required time-varying power for a longer time, improving system sustainability. Moreover, renewable power sources (e.g., solar, wind and geothermal energy) enhance the system's power capacity and reduce the load intensity on batteries and supercapacitors, thus prolonging their lifetime [17], [18], [19], [20], [21].

Researchers have studied the optimal design of a HESS to meet the power demands at the minimum cost [12], [22], [23], [24], [25]. They first explored possible storage configurations (i.e., size, type, connection) of multiple power-source supplies and storages, and then selected the best configuration by comparing the performance and the energy production cost. Search for an optimal design iterates to generate candidate configurations until one of the candidates satisfies the predefined performance and cost criteria [23]. However, these approaches do not guarantee power-sufficiency during operation, as the electric load and criteria are determined based on the load history or the designer's experience and intuition.

- *E. Kim and K. G. Shin are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109. E-mail: {kimsun, kgshin}@umich.edu.*
- *Y. Lee is with the Department of Robotics, Hanyang University, Ansan-si 15588, Korea. E-mail: youngmoonlee@hanyang.ac.kr.*
- *L. He is with the Department of Computer Science and Engineering, University of Colorado Denver, Denver, CO 80204. E-mail: liang.he@ucdenver.edu.*
- *J. Lee is with the Department of Computer Science and Engineering, Sungkyunkwan University (SKKU), Suwon-si 16419, Republic of Korea. E-mail: jinkyu.lee@skku.edu.*

Manuscript received 27 Mar. 2019; revised 17 Feb. 2020; accepted 23 Feb. 2020. Date of publication 28 Feb. 2020; date of current version 23 Mar. 2020. (Corresponding author: Jinkyu Lee.)
Recommended for acceptance by R. Ge.
Digital Object Identifier no. 10.1109/TPDS.2020.2977041

1045-9219 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

To ensure the power sufficiency, we develop a power scheduling and analysis framework for a reliable energy storage system with multiple power-supply sources and multiple power-demand operations, achieving the following goals.

G1. *Offline power-supply guarantee:* We provide an offline guarantee to complete every operation before its deadline (i.e., *operation-level power guarantee*) while keeping the amount of power supplied to the entire system no less than the sum of power required by individual operations at any time instant (i.e., *system-level power guarantee*).

G2. *Online power management:* We develop online power management that effectively utilizes the difference between the worst-case supplied/demanded power and the actual one for improving system performance.

This is a typical CPS problem in that we should address power scheduling and its analysis in the cyber space, based on comprehensive understanding of the physical characteristics of power supply and demand.

To achieve G1, we find similarities between the power-supply guarantee problem and a real-time scheduling problem that determines the execution order of real-time tasks; while the operation-level power guarantee corresponds to the task-level deadline satisfaction, the system-level power guarantee matches the computing platform's capacity constraint. Using the techniques of real-time scheduling, we solve the power-supply guarantee problem in two steps. First, we address the case of multiple power-demand operations with a single, uniform power-supply source (e.g., a battery pack), and develop a scheduling framework and offline power guarantee analysis. For the analysis, we develop a plain analysis that only considers each operation's executable interval (from its release to deadline). We then improve the plain analysis by iteratively reclaiming each operation's slack (the gap between the finishing time and the deadline). Second, based on the scheduling and analysis framework, we address the general problem—multiple power-demand operations with multiple power-supply sources. To this end, we develop two scheduling frameworks to utilize additional sporadic power-supply sources: one for sharing additional power-supply sources by all power-demand operations (called the uniform supply approach), and the other for assigning the sources to only some of power-demand operations (called the dedicated supply approach). Our solution for G1 not only demonstrates that the technique of real-time scheduling helps solve a CPS problem, but also addresses the design problem of a HESS by finding a combination of energy storages at minimum cost.

For the evaluation of G1, we synthesize various power-demand operations and compare the power-guarantee performance of the combination of (i) analysis version (i.e., the plain and improved analyses), and (ii) the existence of additional power-supply sources and the framework type for handling them (i.e., no additional supply, uniform and dedicated supply). We also consider two different sets of additional power-supply sources for the uniform and dedicated supply approaches. We have the following observations for the performance of power guarantee. First, the improved analysis dominates the plain one, meaning that the operation set power-guaranteed by the latter is always power-guaranteed by the former. Second, the performance difference between the former and the latter varies with (ii) and the sets of additional power-supply sources. Third, the dedicated supply outperforms the uniform supply for most cases, but not always.

In addition to making the offline power-supply guarantee (G1), we develop online power management (G2), which aims at increasing the utilization of the energy generated by renewable power-supply sources. That is, scheduling multiple power-supply sources and power-demand operations according to G1 necessarily yields surplus energy since G1 considers the worst case, i.e., the largest power demand and the smallest power supply. While we can utilize the surplus energy for improving various performance aspects of a HESS, we focus on reduction of the peak power supplied by the main energy storage (i.e., a battery pack) to improve its capacity and lifetime [15], [24], [26], [27], [28], [29]. We propose two online power management approaches: *static-demand-based* and *predicted-demand-based* approaches. The former operates effectively when the future scheduling information of power-demand operations is known, while the latter does for a more general case where only the past scheduling information is available.

We focus on a case study for a HESS-powered system to evaluate the proposed solutions for G1 and G2 together. The system for the case study consists of power-consuming components such as wheel motors, stepper motors, coolers, sensors and converters, whose operations are controlled by an application. The system is powered by a HESS consisting of Lithium-ion batteries, ultra-capacitors (UCs), and solar panels. Using the offline power-supply guarantee analysis, we can determine the minimum battery capacity without compromising power-supply guarantee. Our experimental results using a simulation model show that the system with our online power management schemes not only supplies a sufficient amount of power to the components during their operation, and but also reduces the battery's peak power by 56.49 percent.

In summary, this paper makes the following main contributions:

- Design of a power scheduling and analysis framework that consists of (i) offline power-supply guarantee, which is the first power guarantee analysis applicable to the design of a HESS, and (ii) online power management for a HESS to enhance the energy storage's performance,
- Demonstration of the effectiveness of the proposed framework via simulations with synthesized power-demand operation sets and a realistic simulation model for a case study, and
- Solution of an important CPS problem using real-time scheduling techniques.

The paper is organized as follows. Section 2 describes the characteristics of power-supply sources and power-demand operations, and then formally states our main problem. Sections 3 and 4 present the power scheduling and analysis framework for the offline power-supply guarantee, while Section 5 details online power management. Section 6 evaluates the offline power-supply guarantee using simulation. Section 7 focuses on a case study and evaluates the power

sufficiency and peak power reduction achieved by our solutions. Finally, the paper concludes with Section 8.

2 System Model and Problem Statement

2.1 System Model

As the first step for offline power-supply guarantee and online power management, we need to investigate the characteristics of power-demand operations and power-supply sources. Since our target power-demand operations and power-supply sources are the same as our preliminary conference version [30], we describe them in the supplement, available online, and briefly summarize here.

A system has a set of mechanical/electrical operations $\lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_{n^{\mathtt{d}}}\}$, where $n^{\mathtt{d}}$ is the number of operations.[1] The power usage of λ_i is modeled by the minimum inter-arrival time $T_i^{\mathtt{d}}$, its maximum power consumption $P_i^{\mathtt{d}}$, and the maximum length of execution $L_i^{\mathtt{d}}$.

We consider a battery-centric hybrid energy storage system (HESS) that is comprised of three parts: (i) an *energy-dense* lithium-ion battery pack $\Gamma_{\mathtt{bat}}$, (ii) a set of auxiliary renewable energy sources $\Gamma = \{\Gamma_1, \Gamma_2, \ldots \Gamma_{n^{\mathtt{s}}}\}$, and (iii) a *power-dense* energy buffer $\Gamma_{\mathtt{UC}}$. Each energy source Γ_i is modeled with the maximum inter-arrival time $T_i^{\mathtt{s}}$, the minimum supplied power $P_i^{\mathtt{s}}$, and the minimum length of supply duration $L_i^{\mathtt{s}}$.

Throughout the paper, $P(t)$ implies the actual power demand/supply at t, and P represents the maximum power demand or minimum power supply. Also, we use the term of an "operation" for an "instance" of the operation, when no ambiguity arises.

2.2 Problem Statement

Using the characteristics of power-demand operations and power-supply sources described in Section 2.1, we formally state the problem to be solved as:

Given a set of mechanical/electrical operations λ, the battery pack $\Gamma_{\mathtt{bat}}$, the energy buffer $\Gamma_{\mathtt{UC}}$, a set of auxiliary renewable energy sources Γ, and the operation interval $[0, t_{\mathtt{max}})$,

Determine $P^{\mathtt{s}}_{\mathtt{UC}}(t)$ and $\{P_i^{\mathtt{d}}(t)\}_{i=1}^{n^{\mathtt{d}}}$ for all $t \in [0, t_{\mathtt{max}})$ to achieve the following objective function.

$$\text{Minimize(S0)} \quad \int_0^{t_{\mathtt{max}}} P^{\mathtt{s}}_{\mathtt{bat}}(t) + P^{\mathtt{s}}_{\mathtt{bat-loss}}(t)\, dt,$$

$$\text{Subjectto(S1)} \quad \sum_{\lambda_i \in \lambda} P_i^{\mathtt{d}}(t) - P^{\mathtt{s}}_{\mathtt{UC}}(t) - \sum_{\Gamma_i \in \Gamma} P_i^{\mathtt{s}}(t) = P^{\mathtt{s}}_{\mathtt{bat}}(t) \le P^{\mathtt{s}}_{\mathtt{bat}},$$

$$\text{(S2)} \quad P^{\mathtt{s}}_{\mathtt{UC}}(t) \le P^{\mathtt{s}}_{\mathtt{UC}},$$

(S3) Every instance of each operation $\lambda_i \in \lambda$ finishes its execution within $T_i^{\mathtt{d}}$ time units after its release.

$\{P_i^{\mathtt{s}}(t)\}_{i=1}^{n^{\mathtt{s}}}$ are not control knobs in that their generated power is immediately stored in $\Gamma_{\mathtt{UC}}$ or served for power-demand operations. $P^{\mathtt{s}}_{\mathtt{bat}}(t)$ is also not, because it is determined once $P^{\mathtt{s}}_{\mathtt{UC}}(t)$ and $\{P_i^{\mathtt{d}}(t)\}_{i=1}^{n^{\mathtt{d}}}$ are determined as shown in S1. For each $P_i^{\mathtt{d}}(t)$, we determine the time instant at which each λ_i starts to execute. When it comes to $P^{\mathtt{s}}_{\mathtt{UC}}(t)$, we determine the amount of supplied power from $\Gamma_{\mathtt{UC}}$ to λ at each time instant t.

$P^{\mathtt{s}}_{\mathtt{bat-loss}}(t)$ in S0 denotes the amount of power dissipation at t caused by the power supply of the battery pack; in other words, we lose $P^{\mathtt{s}}_{\mathtt{bat-loss}}(t)$ of power due to supplying $P^{\mathtt{s}}_{\mathtt{bat}}(t)$ of power from the battery pack at t. Therefore, S0 implies the amount of energy used and dissipated by the battery pack; since the former (i.e., the amount of energy used) is not a control knob, we should reduce the latter (i.e., the amount of energy dissipated) using a simple circuit-based battery model [31]: $P^{\mathtt{s}}_{\mathtt{bat-loss}}(t) = I^2_{\mathtt{bat}}(t) \cdot R_{\mathtt{bat}}(t)$, where $R_{\mathtt{bat}}(t)$ is a battery's internal resistance, and $I_{\mathtt{bat}}(t)$ is the battery's discharge/charge current for supplying $P_{\mathtt{bat}}(t)$. Since power dissipation is quadratically proportional to discharge/charge battery current, the minimization of battery peak current may reduce energy dissipation of the battery pack, potentially extending the battery operation-time.

3 Scheduling of Multiple Power-Demand Operations With a Uniform Supply

In this section, we present how to schedule a set of power-demand operations (λ) when the battery pack ($\Gamma_{\mathtt{bat}}$) is the sole power-supply source. We first develop a scheduling framework that considers the characteristics of λ and $\Gamma_{\mathtt{bat}}$ described in Section 2.1. Under this scheduling framework, we then develop an offline power guarantee analysis that determines whether every operation in λ powered by the battery pack is performed before its deadline without suffering any power shortage. Finally, we derive an improved version of the offline power guarantee analysis, which accommodates significantly more (guaranteed) power-demand operations under the same scheduling framework.

3.1 Scheduling Framework

We would like to schedule a set of multiple operations (λ) so as to achieve a system-level and operation-level power guarantee under the uniform supply from the battery pack $\Gamma_{\mathtt{bat}}$. By "system-level power guarantee," we mean that the sum of power demand at any time instant should be no larger than maximum power capability of the battery pack, i.e., $\sum_{\lambda_i \in \lambda} P_i^{\mathtt{d}}(t) \le P^{\mathtt{s}}_{\mathtt{bat}}$, which is equivalent to S1.[2] To achieve the operation-level power guarantee, every operation should receive sufficient power within its period, which is S3.

Our scheduling framework employs the work-conserving policy based on the worst-case power demand and operation-level fixed-priority scheduling policy. The former implies that an operation λ_k can start to execute as long as its worst-case power demand $P_k^{\mathtt{d}}$ (as opposed to the actual power demand $P_k^{\mathtt{d}}(t)$) is no larger than the difference between the battery capability $P^{\mathtt{s}}_{\mathtt{bat}}$ and the sum of the worst-case power demand of currently-executing operations $\sum_{\lambda_i \in Q_{\mathtt{run}}} P_i^{\mathtt{d}}$. The latter implies that the scheduling framework prioritizes the operations that satisfy the above condition.

Since each operation exhibits the non-preemptive behavior as described in Section 2.1, an operation can start its execution only when it is released or the other operation is finished, and each operation, once started, continues its

1. Throughout the paper, we use the superscripts d and s for power demand and power supply, respectively.

2. Since this section considers the sole supply of $\Gamma_{\mathtt{bat}}$, we remove terms of the energy buffer and a set of renewable power-supply sources in S1.

execution until the completion. Therefore, the scheduling framework can be expressed (as in Algorithm 1) by describing actions for the situations. Lines 1–3 update the ready queue $Q_{\texttt{ready}}$ that contains operations ready to execute, and the running queue $Q_{\texttt{run}}$ for currently-executing operations. Lines 4–9 select operations to be started and move them from $Q_{\texttt{ready}}$ to $Q_{\texttt{run}}$. Although the scheduling framework prioritizes operations based on their priorities, a lower-priority operation λ_j in $Q_{\texttt{ready}}$ can start its execution earlier than a higher-priority operation λ_k in $Q_{\texttt{ready}}$, if the remaining power capability of the system (i.e., $P^{\texttt{s}}_{\texttt{bat}} - \sum_{\lambda_i \in Q_{\texttt{run}}} P^{\texttt{d}}_i$) is larger than $P^{\texttt{d}}_k$ but, no larger than $P^{\texttt{d}}_j$, as described in line 5.

Algorithm 1. Scheduling Framework

The following steps are performed whenever at least one mechanical/electrical operation is finished or released at t:
1: For each λ_k finished at t, $Q_{\texttt{run}} \leftarrow Q_{\texttt{run}} \setminus \{\lambda_k\}$
2: For each λ_k released at t, $Q_{\texttt{ready}} \leftarrow Q_{\texttt{ready}} \cup \{\lambda_k\}$
3: Sort $Q_{\texttt{ready}}$ by given operation-level fixed-priorities
4: **for** $\lambda_k \in Q_{\texttt{ready}}$ (a higher priority operation is chosen earlier) **do**
5: **if** $P^{\texttt{d}}_k \leq P^{\texttt{s}}_{\texttt{bat}} - \sum_{\lambda_i \in Q_{\texttt{run}}} P^{\texttt{d}}_i$ **then**
6: $Q_{\texttt{run}} \leftarrow Q_{\texttt{run}} \cup \{\lambda_k\}$
7: $Q_{\texttt{ready}} \leftarrow Q_{\texttt{ready}} \setminus \{\lambda_k\}$
8: **end if**
9: **end for**

3.2 Offline Power Guarantee Analysis

Since each operation is non-preemptive, we need to check if each operation λ_k can start its execution no later than $(T^{\texttt{d}}_k - L^{\texttt{d}}_k)$ time units after its release; once it starts to execute, it completes execution within its period without any preemption. Therefore, we focus on the interval of length $(T^{\texttt{d}}_k - L^{\texttt{d}}_k + \epsilon_t)$, which begins at the time of λ_k's release, and check whether the sum of energy consumed by other operations within the interval is strictly less than $(P^{\texttt{s}}_{\texttt{bat}} - P^{\texttt{d}}_k + \epsilon_p) \cdot (T^{\texttt{d}}_k - L^{\texttt{d}}_k + \epsilon_t)$, where ϵ_t and ϵ_p denote the time and power quantum (the smallest unit). Since the smallest power that prevents λ_k from execution at each time instant is $(P^{\texttt{s}}_{\texttt{bat}} - P^{\texttt{d}}_k + \epsilon_p)$, the above condition guarantees the start of λ_k's execution to be no later than $(T^{\texttt{d}}_k - L^{\texttt{d}}_k)$ time units after its release.

The remaining step is then to calculate $I_{k \leftarrow i}(\ell)$, the amount of energy demanded by instances of λ_i within an interval of length ℓ that contributes to prevention of λ_k from starting its execution. Note that for calculation of $I_{k \leftarrow i}(\ell)$, we limit the power demand at each time instant to $(P^{\texttt{s}}_{\texttt{bat}} - P^{\texttt{d}}_k + \epsilon_p)$, since we need to know whether the sum of all power demands at each time instant is no smaller than $(P^{\texttt{s}}_{\texttt{bat}} - P^{\texttt{d}}_k + \epsilon_p)$. Now, we will describe how to calculate an upper bound of $I_{k \leftarrow i}(\ell)$.

First, if λ_i has a higher priority than λ_k, then the upper-bound of $I_{k \leftarrow i}(\ell)$ is the maximum energy demanded by λ_i in an interval of length ℓ, which is calculated as

$$W_{k \leftarrow i}(\ell) = \min\big(P^{\texttt{d}}_i, P^{\texttt{s}}_{\texttt{bat}} - P^{\texttt{d}}_k + \epsilon_p\big) \times \min\Big(\ell, N_i(\ell) \cdot L^{\texttt{d}}_i + \min\big(L^{\texttt{d}}_i, \ell + T^{\texttt{d}}_i - L^{\texttt{d}}_i - N_i(\ell) \cdot T^{\texttt{d}}_i\big)\Big), \tag{1}$$

where $N_i(\ell) = \lfloor \frac{\ell + T^{\texttt{d}}_i - L^{\texttt{d}}_i}{T^{\texttt{d}}_i} \rfloor$. This calculation is similar to the workload calculation of real-time scheduling [32]. Briefly, $N_i(\ell)$ implies the number of instances of λ_i, and each of their periods is completely included within the interval of length ℓ (including the first instance of λ_i), which contributes $\min(P^{\texttt{d}}_i, P^{\texttt{s}}_{\texttt{bat}} - P^{\texttt{d}}_k + \epsilon_p) \cdot N_i(\ell) \cdot L^{\texttt{d}}_i$ of energy to $W_{k \leftarrow i}(\ell)$. The second part of $W_{k \leftarrow i}(\ell)$ represents the contribution of the operation whose period is partially included in the interval of length ℓ. Fig. 16 in the supplement, available online, shows an example of $W_{k \leftarrow i}(\ell)$ with $N_i(\ell) = 2$, which contributes $\min(P^{\texttt{d}}_i, P^{\texttt{s}}_{\texttt{bat}} - P^{\texttt{d}}_k + \epsilon_p) \cdot 2 \cdot L^{\texttt{d}}_i$ of energy, and the third instance of λ_i contributes $\min(P^{\texttt{d}}_i, P^{\texttt{s}}_{\texttt{bat}} - P^{\texttt{d}}_k + \epsilon_p) \cdot \min(L^{\texttt{d}}_i, \ell + T^{\texttt{d}}_i - L^{\texttt{d}}_i - N_i(\ell) \cdot T^{\texttt{d}}_i)$ of energy to $W_{k \leftarrow i}(\ell)$.

Second, if λ_i has a lower priority than λ_k, then we consider two sub-cases. Since each operation is non-preemptive, λ_i can execute before the start of λ_k's execution, if λ_i starts its execution before the release of λ_k. In this case, the energy demand is upper-bounded by $\min(P^{\texttt{d}}_i, P^{\texttt{s}}_{\texttt{bat}} - P^{\texttt{d}}_k + \epsilon_p) \cdot \min(L^{\texttt{d}}_i - \epsilon_t, \ell)$, which is an upper-bound of $I_{k \leftarrow i}(\ell)$ for the first sub-case of $P^{\texttt{d}}_i \geq P^{\texttt{d}}_k$. If $P^{\texttt{d}}_i < P^{\texttt{d}}_k$, then an upper-bound of $I_{k \leftarrow i}(\ell)$ can be larger than the first sub-case. That is, due to our worst-case-based work-conserving policy, it is possible for λ_i to start its execution before the start of λ_k's execution, whenever λ_k does not satisfy Line 5 of Algorithm 1 but λ_i does. In this case, we use the general upper-bound $W_{k \leftarrow i}(\ell)$ as an upper-bound of the second sub-case.

Combining all the results discussed so far, we develop a power guarantee analysis as follows.

Lemma 1. *Suppose every $\lambda_k \in \lambda$ satisfies Eq. (2). Then, every instance of every operation $\lambda_k \in \lambda$ finishes its execution within its period of length $T^{\texttt{d}}_k$, while guaranteeing the sum of power demands at any time instant is no larger than the battery power capability (i.e.,* S1 *and* S3 *hold).*

$$\sum_{\lambda_i \in \lambda \setminus \{\lambda_k\}} I_{k \leftarrow i}(T^{\texttt{d}}_k - L^{\texttt{d}}_k + \epsilon_t) < (P^{\texttt{s}}_{\texttt{bat}} - P^{\texttt{d}}_k + \epsilon_p) \cdot (T^{\texttt{d}}_k - L^{\texttt{d}}_k + \epsilon_t), \tag{2}$$

where $I_{k \leftarrow i}(\ell) = W_{k \leftarrow i}(\ell)$, if λ_k has a lower-priority than λ_i or $P^{\texttt{d}}_i < P^{\texttt{d}}_k$; $I_{k \leftarrow i}(\ell) = \min(P^{\texttt{d}}_i, P^{\texttt{s}}_{\texttt{bat}} - P^{\texttt{d}}_k + \epsilon_p) \cdot \min(L^{\texttt{d}}_i - \epsilon_t, \ell)$ otherwise.

Proof. As discussed so far, $I_{k \leftarrow i}(T^{\texttt{d}}_k - L^{\texttt{d}}_k + \epsilon_t)$ in Eq. (2) is an upper-bound of the amount of energy demanded by instances of λ_i in an interval of length $(T^{\texttt{d}}_k - L^{\texttt{d}}_k + \epsilon_t)$. Therefore, if Eq. (2) holds, then there exists an instant t_1 within the interval, such that the sum of power demands is less than or equal to $(P^{\texttt{s}}_{\texttt{bat}} - P^{\texttt{d}}_k)$. This means that λ_k can start its execution no later than $(T^{\texttt{d}}_k - L^{\texttt{d}}_k)$ time units after its release, implying that λ_k finishes its execution within its period. □

The lemma works not only for an offline power guarantee for the case of the sole supply, but also for a basis to develop an offline power guarantee for the general case to be discussed in Section 4. Also, the lemma can be used for addressing a design problem: calculation of the minimum capability of the battery cell that can supply given λ by finding the minimum $P^{\texttt{s}}_{\texttt{bat}}$ that satisfies the lemma for given λ.

3.3 Improved Power-Guarantee Analysis

We now develop a new power-guarantee analysis, which is an improved version of the one presented above. We first

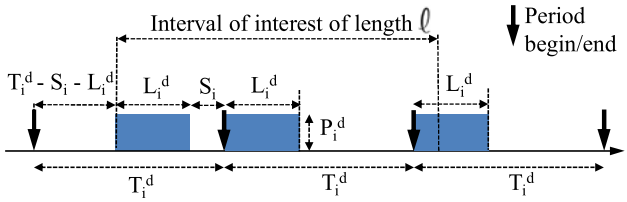


Fig. 1. An example of $W^*_{k\leftarrow i}(\ell, S_i)$ with $N^*_i(\ell, S_i) = 2$.

define a slack of $\lambda_i \in \lambda$ as the minimum time between the finishing time and the deadline of every instance of λ_i, and let S_i denote the slack of λ_i. In other words, if the absolute deadline of an instance of λ_i is t, its execution is completed no later than $(t - S_i)$. By definition, S_i is non-negative. If we use $S_i > 0$, the first instance in the interval of interest for $W_{k\leftarrow i}(\ell)$ cannot execute as late as possible (as shown in Fig. 16 in the supplement, available online); instead, the first instance should finish its execution S_i ahead of its deadline as shown in Fig. 1. We can reduce $W_{k\leftarrow i}(\ell)$ as $W^*_{k\leftarrow i}(\ell, S_i)$ as follows.

$$W^*_{k\leftarrow i}(\ell, S_i) = \min\left(P^{\mathrm{d}}_i, P^{\mathrm{s}}_{\mathrm{bat}} - P^{\mathrm{d}}_k + \epsilon_p\right) \times \min\Big(\ell, N^*_i(\ell, S_i) \cdot L^{\mathrm{d}}_i + \min\left(L^{\mathrm{d}}_i, \ell + T^{\mathrm{d}}_i - S_i - L^{\mathrm{d}}_i - N^*_i(\ell, S_i) \cdot T^{\mathrm{d}}_i\right)\Big), \tag{3}$$

where $N^*_i(\ell, S_i) = \lfloor \frac{\ell + T^{\mathrm{d}}_i - S_i - L^{\mathrm{d}}_i}{T^{\mathrm{d}}_i} \rfloor$. Similarly to $W_{k\leftarrow i}(\ell)$, the physical meaning of $W^*_{k\leftarrow i}(\ell, S_i)$ is maximum energy interference of λ_i on λ_k, which is upper-bounded by the maximum energy demanded by λ_i in an interval of length ℓ with given S_i. Except for S_i, the way to calculate $W^*_{k\leftarrow i}(\ell, S_i)$ is the same as $W_{k\leftarrow i}(\ell)$.

Replacing $W^*_i(\ell, S_i)$ into $W_i(\ell)$, we develop an improved version of the power guarantee analysis stated in the following lemma.

Lemma 2. *Suppose that every $\lambda_k \in \lambda$ satisfies Eq. (4), and S_i for every $\lambda_i \in \lambda$ is given. Then, every instance of every operation $\lambda_k \in \lambda$ finishes its execution within its period of length T^{d}_k, while guaranteeing the sum of power demands at any time instant is no larger than the battery power capability (i.e.,* S1 *and* S3 *hold).*

$$\sum_{\lambda_i \in \lambda \setminus \{\lambda_k\}} I^*_{k\leftarrow i}(T^{\mathrm{d}}_k - L^{\mathrm{d}}_k + \epsilon_t, S_i) < (P^{\mathrm{s}}_{\mathrm{bat}} - P^{\mathrm{d}}_k + \epsilon_p) \cdot (T^{\mathrm{d}}_k - L^{\mathrm{d}}_k + \epsilon_t), \tag{4}$$

where $I^*_{k\leftarrow i}(\ell, S_i) = W^*_{k\leftarrow i}(\ell, S_i)$, if λ_k has a lower-priority than λ_i or $P^{\mathrm{d}}_i < P^{\mathrm{d}}_k$; $I^*_{k\leftarrow i}(\ell) = \min(P^{\mathrm{d}}_i, P^{\mathrm{s}}_{\mathrm{bat}} - P^{\mathrm{d}}_k + \epsilon_p) \cdot \min(L^{\mathrm{d}}_i - \epsilon_t, \ell)$ otherwise.

Proof. By the definition of $W^*_{k\leftarrow i}(\ell, S_i)$, $I^*_{k\leftarrow i}(T^{\mathrm{d}}_k - L^{\mathrm{d}}_k + \epsilon_t, S_i)$ in Eq. (4) is an upper-bound of the amount of energy demanded by instances of λ_i in an interval of length $(T^{\mathrm{d}}_k - L^{\mathrm{d}}_k + \epsilon_t)$ when the slack of λ_i is S_i. The remainder of the proof is the same as Lemma 1. □

For power guarantee, Lemma 2 misses one important issue—how to calculate S_i. Using Eq. (4), we can calculate S_i as follows.

Lemma 3. *The slack of $\lambda_k \in \lambda$ (denoted by S_k) can be lower-bounded as:*

$$S_k \geq T^{\mathrm{d}}_k - L^{\mathrm{d}}_k - \frac{\sum_{\lambda_i \in \lambda \setminus \{\lambda_k\}} I^*_{k\leftarrow i}(T^{\mathrm{d}}_k - L^{\mathrm{d}}_k + \epsilon_t, S_i)}{P^{\mathrm{s}}_{\mathrm{bat}} - P^{\mathrm{d}}_k + \epsilon_p}. \tag{5}$$

Proof. The lemma holds by Eq. (4). That is, Eq. (4) can be rephrased as:

$$\frac{\sum_{\lambda_i \in \lambda \setminus \{\lambda_k\}} I^*_{k\leftarrow i}(T^{\mathrm{d}}_k - L^{\mathrm{d}}_k + \epsilon_t, S_i)}{P^{\mathrm{s}}_{\mathrm{bat}} - P^{\mathrm{d}}_k + \epsilon_p} < T^{\mathrm{d}}_k - L^{\mathrm{d}}_k + \epsilon_t. \tag{6}$$

Then, the physical meaning of the left-hand side of Eq. (6) is the maximum interval length that other tasks than λ_k prevent λ_k from execution. This means that the interval length between λ_k's release and finishing times is at most as much as the sum of the left-hand side of Eq. (6) (i.e., interference from other operations than λ_k) and L^{d}_k (i.e., the execution time of λ_k itself). Therefore, the slack value of λ_k is at least as much as the right-hand side of Eq. (5). □

The overall process on how to incorporate Lemma 3 into Lemma 2 is the same as the slack reclamation of real-time scheduling [33]. That is, in the first iteration, we apply Lemma 2 with $S_k = 0$ for every $\lambda_k \in \lambda$, and update $S_k > 0$ for every $\lambda_k \in \lambda$ using Lemma 3. In the second iteration, we repeat the same process with the new slacks. This process is repeated until every operation satisfies Eq. (4) (achieving the power guarantee) or there is no more slack update (failing the power guarantee).

In terms of power guarantee, it is trivial to show that the improved version (Lemma 2 with Lemma 3) is always better performance than Lemma 1. That is because Lemma 2 with $S_k = 0$ for every $\lambda_k \in \lambda$ is equivalent to Lemma 1. On the other hand, the improved version incurs high time-complexity due to some iteration. Due to the iteration, the time-complexity of the slack reclamation is known to be pseudo-polynomial [33]. Since we are interested in "offline" power guarantees, the pseudo-polynomial time-complexity is acceptable for most cases. In Section 6, we will compare the performance of the vanilla (simple) and improved versions of the proposed offline power guarantee.

4 Scheduling of Multiple Power-Demand Operations With Multiple Power-Supply Sources

This section addresses a more general situation than Section 3, in which additional power is sporadically generated from multiple power sources such as an RBS and a solar panel, and immediately stored in the energy buffer or used for power-demand operations. We will first address a scheduling challenge due to the existence of sporadic additional power supply. Second, we present two approaches, depending on how to distribute the additional power supply to power-demand operations. Finally, we describe how to incorporate the two approaches in the proposed offline power-guarantee analysis.

4.1 A Scheduling Challenge

Unlike the situation where a battery pack is the only power-supply source discussed in Section 3, a straightforward

approach cannot yield a system-level power guarantee, as shown in the following example.

Example 1. Suppose that additional power is supplied by Γ_1 in $[t_1, t_2)$, while the battery pack is the only supply in $[t_0, t_1)$ and $[t_2, t_3)$, where $t_0 < t_1 < t_2 < t_3$, as shown in Fig. 17a in the supplement, available online. Also, there are two operations λ_1 and λ_2 ready to execute at t_0, and $\lambda_1 \leq P^{\mathtt{s}}_{\mathtt{bat}}$ and $\lambda_1 + \lambda_2 > P^{\mathtt{s}}_{\mathtt{bat}}$, as shown in Fig. 17b in the supplement, available online. Suppose that we apply the worst-case-based work-conserving policy in Algorithm 1, implying we start execution of an operation in the ready queue as long as the system has enough remaining power supply to accommodate the worst-case power demand of the operation. Then, λ_2 can start its execution at t_1, but there is a problem at t_2, at which power supplied by Γ_1 ends. In $[t_2, t_3)$, the total amount of power demand is strictly larger than that of power supply, entailing either eviction of one of "non-preemptive" operations, or risking power shortage in executing operations, both of which are considered as a system failure.

Example 1 shows the need for a more fine-grained way to handle additional sporadic power-supply sources. To meet this need, we consider two policies depending on how to distribute the additional supply as follows.

- Calculate the additional "guaranteed" uniform supply $P^{\mathtt{s}}_{\mathtt{uni}}$, meaning that additional power-supply sources (and the energy buffer) can *always* provide power as much as $P^{\mathtt{s}}_{\mathtt{uni}}$ (as the battery pack provides up to $P^{\mathtt{s}}_{\mathtt{bat}}$). This entails the calculation of $P^{\mathtt{s}}_{\mathtt{uni}}$; once it is calculated, we can reuse the power scheduling and analysis framework presented in Section 3, by adding $P^{\mathtt{s}}_{\mathtt{uni}}$ to the existing uniform supply $P^{\mathtt{s}}_{\mathtt{bat}}$. In this case, all operations can share the power generated by additional power-supply sources.
- Assign additional power to a partial set of operations. Power generated by additional power-supply sources (and stored in the energy buffer) is used only when the operations in the partial set are executed. This entails the way to divide power generated by additional power-supply sources for individual power-demand operations.

In what follows, we will detail the above two approaches, including their scheduling frameworks.

4.2 Uniform Supply

In this approach, we calculate the additional guaranteed uniform supply $P^{\mathtt{s}}_{\mathtt{uni}}$ from additional power-supply sources. After calculating $P^{\mathtt{s}}_{\mathtt{uni}}$, we can reuse the scheduling framework in Algorithm 1. That is, we just change the $P^{\mathtt{s}}_{\mathtt{bat}}$ term in Line 5 to $P^{\mathtt{s}}_{\mathtt{bat}} + P^{\mathtt{s}}_{\mathtt{uni}}$. The main issue of this approach is to accurately calculate $P^{\mathtt{s}}_{\mathtt{uni}}$; the larger $P^{\mathtt{s}}_{\mathtt{uni}}$, the more operations to be accommodated.

The basic idea to obtain $P^{\mathtt{s}}_{\mathtt{uni}}$ is to calculate the amount of the minimum supplied energy in $[0, t)$ by considering the fact that the energy buffer has at least $P^{\mathtt{s}}_{\mathtt{UC}} \cdot L^{\mathtt{s}}_{\mathtt{UC}}$ of energy at 0 and each additional supply generates power at the end of its instances' periods. If we divide this amount by t and take the minimum, we guarantee to supply power as much as $P^{\mathtt{s}}_{\mathtt{uni}}$ in $[0, t)$, as stated in the following lemma.

Lemma 4. *We can calculate* $P^{\mathtt{s}}_{\mathtt{uni}}$ *using the following equation.*

$$P^{\mathtt{s}}_{\mathtt{uni}} = \min_{0 \leq t \leq \mathsf{LCM}} \frac{f(\Gamma_{\mathtt{UC}}, t) + \sum_{\Gamma_i \in \Gamma} f(\Gamma_i, t)}{t}, \tag{7}$$

where LCM *is the least common multiple of* $\{T^{\mathtt{s}}_i\}_{\Gamma_i \in \Gamma}$,

$$f(\Gamma_{\mathtt{UC}}, t) = P^{\mathtt{s}}_{\mathtt{UC}} \cdot \min(t, L^{\mathtt{s}}_{\mathtt{UC}}), \text{and} \tag{8}$$

$$f(\Gamma_i, t) = P^{\mathtt{s}}_i \cdot \left\lfloor \frac{t}{T^{\mathtt{s}}_i} \right\rfloor \cdot L^{\mathtt{s}}_i + P^{\mathtt{s}}_i \cdot \max\left(0, t - \left\lfloor \frac{t}{T^{\mathtt{s}}_i} \right\rfloor \cdot T^{\mathtt{s}}_i - (T^{\mathtt{s}}_i - L^{\mathtt{s}}_i)\right). \tag{9}$$

Proof. Since the amount of energy in the energy buffer at $t = 0$ is at least $P^{\mathtt{s}}_{\mathtt{UC}} \cdot L^{\mathtt{s}}_{\mathtt{UC}}$, the amount of the supplied energy from the energy buffer in $[0, t)$ is $P^{\mathtt{s}}_{\mathtt{UC}} \cdot t$ if $t \leq L^{\mathtt{s}}_{\mathtt{UC}}$, and at least $P^{\mathtt{s}}_{\mathtt{UC}} \cdot L^{\mathtt{s}}_{\mathtt{UC}}$ otherwise, which is recorded in $f(\Gamma_{\mathtt{UC}}, t)$ of Eq. (8). For given t, $\lfloor \frac{t}{T^{\mathtt{s}}_i} \rfloor$ means the number of instances of λ_i whose periods are completely included in $[0, t)$, and each instance generates energy no smaller than $P^{\mathtt{s}}_i \cdot L^{\mathtt{s}}_i$. The second term of Eq. (9) presents the minimum energy generated by the last instance whose period is partially included in $[0, t)$. Therefore, $\sum_{\Gamma_i \in \Gamma} f(\Gamma_i, t)$ represents the amount of generated energy by Γ in $[0, t)$. Since we assume that the capacity of the energy buffer is sufficiently large, we can always use power as much as the lower-bound of $\frac{f(\Gamma_{\mathtt{UC}}, t) + \sum_{\Gamma_i \in \Gamma} f(\Gamma_i, t)}{t}$ for $0 \leq t \leq \mathsf{LCM}$. □

Note that the additional power supply should be stored in the energy buffer $\Gamma_{\mathtt{UC}}$, and hence $P^{\mathtt{s}}_{\mathtt{uni}}$ cannot be larger than $P^{\mathtt{s}}_{\mathtt{UC}}$.

If LCM is very large or time-complexity is critically important, e.g., for online admission control for operations, we need a tractable way to calculate $P^{\mathtt{s}}_{\mathtt{uni}}$, which is covered in the following lemma.

Lemma 5. $P^{\mathtt{s}}_{\mathtt{uni}}$ *is at least as much as the minimum of* $P^{\mathtt{s}}_{\mathtt{UC}}$ *and* $\sum_{\Gamma_i \in \Gamma'} \frac{P^{\mathtt{s}}_i \cdot L^{\mathtt{s}}_i}{T^{\mathtt{s}}_i}$, *if* Γ' *satisfies Eq. (10).*

$$\sum_{\Gamma_i \in \Gamma'} (T^{\mathtt{s}}_i - L^{\mathtt{s}}_i) \cdot \frac{P^{\mathtt{s}}_i \cdot L^{\mathtt{s}}_i}{T^{\mathtt{s}}_i} \leq P^{\mathtt{s}}_{\mathtt{UC}} \cdot L^{\mathtt{s}}_{\mathtt{UC}}. \tag{10}$$

We will describe how to find Γ' *after the lemma.*

Proof. Since $\Gamma^{\mathtt{s}}_{\mathtt{UC}}$ can supply power at most $P^{\mathtt{s}}_{\mathtt{UC}}$, $P^{\mathtt{s}}_{\mathtt{uni}}$ should be no larger than $P^{\mathtt{s}}_{\mathtt{UC}}$. Otherwise, there should exist some Γ_i that always generates power, which is impossible because we cannot control if each Γ_i generates power.

While $\frac{P^{\mathtt{s}}_i \cdot L^{\mathtt{s}}_i}{T^{\mathtt{s}}_i}$ is the average rate of power generation by Γ_i, it cannot entirely contribute $P^{\mathtt{s}}_{\mathtt{uni}}$. This is, because there is no power generation in $[0, T^{\mathtt{s}}_i - L^{\mathtt{s}}_i)$. To supply $\frac{P^{\mathtt{s}}_i \cdot L^{\mathtt{s}}_i}{T^{\mathtt{s}}_i}$-rate power in $[0, T^{\mathtt{s}}_i - L^{\mathtt{s}}_i)$, we need to use $(T^{\mathtt{s}}_i - L^{\mathtt{s}}_i) \cdot \frac{P^{\mathtt{s}}_i \cdot L^{\mathtt{s}}_i}{T^{\mathtt{s}}_i}$ amount of power from $\Gamma_{\mathtt{UC}}$. Then, the generated energy from Γ_i in $[x \cdot T^{\mathtt{s}}_i - L^{\mathtt{s}}_i, x \cdot T^{\mathtt{s}}_i)$ (as much as $P^{\mathtt{s}}_i \cdot L^{\mathtt{s}}_i$) enables to supply $\frac{P^{\mathtt{s}}_i \cdot L^{\mathtt{s}}_i}{T^{\mathtt{s}}_i}$-rate power in $[x \cdot T^{\mathtt{s}}_i - L^{\mathtt{s}}_i, (x+1) \cdot T^{\mathtt{s}}_i - L^{\mathtt{s}}_i)$. Therefore, the LHS of Eq. (10) is an upper-bound of the amount of energy for every operation $\lambda_i \in \lambda'$ to contribute $\frac{P^{\mathtt{s}}_i \cdot L^{\mathtt{s}}_i}{T^{\mathtt{s}}_i}$ to $P^{\mathtt{s}}_{\mathtt{uni}}$.

Since the amount of power in $\Gamma_{\mathtt{UC}}$ at $t=0$ is $P^{\mathtt{s}}_{\mathtt{UC}} \cdot L^{\mathtt{s}}_{\mathtt{UC}}$, we enforce the constraint Eq. (10). □

To calculate $P^{\mathtt{s}}_{\mathtt{uni}}$ using Lemma 5, we introduce a heuristic to find Γ' that satisfies Eq. (10). Staring from $\Gamma' = \emptyset$, we repeat to add an operation in $\Gamma \setminus \Gamma'$ to Γ' until Eq. (10) is satisfied. Here, the order to be selected from $\Gamma \setminus \Gamma'$ is based on the following criteria (the larger value, the earlier selection):

$$\frac{\frac{P^{\mathtt{s}}_i \cdot L^{\mathtt{s}}_i}{T^{\mathtt{s}}_i}}{\frac{T^{\mathtt{s}}_i - L^{\mathtt{s}}_i}{T^{\mathtt{s}}_i} \cdot P^{\mathtt{s}}_i \cdot L^{\mathtt{s}}_i} = \frac{1}{T^{\mathtt{s}}_i - L^{\mathtt{s}}_i}. \tag{11}$$

We use the above criteria because a larger value of $\frac{1}{T^{\mathtt{s}}_i - L^{\mathtt{s}}_i}$ means more contribution of $P^{\mathtt{s}}_{\mathtt{uni}}$ (i.e., denominator) per usage of $P^{\mathtt{s}}_{\mathtt{UC}} \cdot L^{\mathtt{s}}_{\mathtt{UC}}$ (i.e., numerator).

We will describe how to use $P^{\mathtt{s}}_{\mathtt{uni}}$ in Section 4.4.

4.3 Dedicated Supply Approach

In this approach, we can determine $\lambda^{\mathtt{ded}}$, a set of operations completely powered by a set of additional power-supply sources Γ and the energy buffer $\Gamma_{\mathtt{UC}}$. Once we determine $\lambda^{\mathtt{ded}}$, operations in $\lambda \setminus \lambda^{\mathtt{ded}}$ can be executed according to Algorithm 1, and their power guarantee is judged by Lemma 1 with $\lambda \setminus \lambda^{\mathtt{ded}}$. On the other hand, each operation in $\lambda^{\mathtt{ded}}$ is fully supplied by Γ with $\Gamma_{\mathtt{UC}}$, and does not use power from $\Gamma_{\mathtt{bat}}$. Our policy is to execute each operation in $\lambda^{\mathtt{ded}}$ at the end of each period, which accommodates more operations in $\lambda^{\mathtt{ded}}$ (because this policy uses less initial energy from the energy buffer). Formally, $\lambda_k \in \lambda^{\mathtt{ded}}$ starts its execution at $r + T^{\mathtt{d}}_k - L^{\mathtt{d}}_k$, where r is the release time of an instance of λ_k, and $\Gamma_{\mathtt{UC}}$ supplies $P^{\mathtt{d}}_k(t)$ ($\leq P^{\mathtt{d}}_k$) amount of power to λ_k in $[r + T^{\mathtt{d}}_k - L^{\mathtt{d}}_k, r + T^{\mathtt{d}}_k)$.

Then, the remaining step is to determine $\lambda^{\mathtt{ded}}$. The basic idea is to calculate the maximum energy demanded by $\lambda^{\mathtt{ded}}$ in $[0,t)$ and the minimum energy supplied by Γ and $\Gamma_{\mathtt{UC}}$ in $[0,t)$. We check whether the former is not larger than the latter at all times, which is stated in the following lemma.

Lemma 6. *Every instance of every operation $\lambda_k \in \lambda_{\mathtt{ded}}$ finishes its execution at the end of each period (e.g., $[r + T^{\mathtt{d}}_k - L^{\mathtt{d}}_k, r + T^{\mathtt{d}}_k)$ where r is the release time of an instance of λ_k), only with Γ and $\Gamma_{\mathtt{UC}}$, if the following inequality holds for all $t \in [0, \mathsf{LCM})$.*

$$\sum_{\lambda_i \in \lambda^{\mathtt{ded}}} f(\lambda_i, t) \leq f(\Gamma_{\mathtt{UC}}, t) + \sum_{\Gamma_i \in \Gamma} f(\Gamma_i, t), \text{and} \tag{12}$$

$$f(\lambda_i, t) = P^{\mathtt{d}}_i \cdot \left\lfloor \frac{t}{T_i} \right\rfloor \cdot L^{\mathtt{d}}_i + P^{\mathtt{d}}_i \cdot \max\left(0, t - \left\lfloor \frac{t}{T^{\mathtt{d}}_i} \right\rfloor \cdot T^{\mathtt{d}}_i - (T^{\mathtt{d}}_i - L^{\mathtt{d}}_i)\right). \tag{13}$$

Proof. Since $f(\lambda_i, t)$ in Eq. (13) exhibits the same formula as Eq. (9), it calculates the maximum energy demanded by λ_i in $[0,t)$ when its instances are executed at the end of their periods. Therefore, the LHS (Left-Hand Side) of Eq. (12) is the maximum energy demanded by $\lambda^{\mathtt{ded}}$ in $[0,t)$. On the other hand, the RHS (Right-Hand Side) of the equation is the minimum energy supplied by Γ and $\Gamma_{\mathtt{UC}}$ in $[0,t)$ as explained in Lemma 4. Therefore, the lemma follows. □

If time-complexity is critical, we can use another necessary condition presented in the following lemma (likewise Lemma 5).

Lemma 7. *Every instance of every operation $\lambda_k \in \lambda_{\mathtt{ded}}$ finishes its power execution at the end of each period (i.e., $[r + T^{\mathtt{d}}_k - L^{\mathtt{d}}_k, r + T^{\mathtt{d}}_k)$ where r is the release time of an instance of λ_k), only with Γ and $\Gamma_{\mathtt{UC}}$ (without $\Gamma_{\mathtt{bat}}$), if the following inequality holds:*

$$\sum_{\lambda_i \in \lambda^{\mathtt{ded}}} \frac{P^{\mathtt{d}}_i \cdot L^{\mathtt{d}}_i}{T^{\mathtt{d}}_i} \leq P^{\mathtt{s}}_{\mathtt{uni}} \text{ in Lemma 4 (or Lemma 5).} \tag{14}$$

Proof. By the meaning of $P^{\mathtt{s}}_{\mathtt{uni}}$, Γ and $\Gamma_{\mathtt{UC}}$ can *always* supply $P^{\mathtt{s}}_{\mathtt{uni}}$-rate power. If we enforce each operation $\lambda_k \in \lambda_{\mathtt{ded}}$ executes its instances at the end of their periods, their cumulative demand is always not larger than $P^{\mathtt{s}}_{\mathtt{uni}}$-rate power. Therefore, as long as their total demanded power rate (i.e., the LHS of Eq. (14)) is no larger than $P^{\mathtt{s}}_{\mathtt{uni}}$, every instance of every operation $\lambda_k \in \lambda_{\mathtt{ded}}$ finishes its execution within its period of length $T^{\mathtt{d}}_k$, only with Γ and $\Gamma_{\mathtt{UC}}$. □

The remaining problem is then how to select $\lambda^{\mathtt{ded}}$ that satisfies Lemma 6 (or Lemma 7). Here we describe a simple, but effective heuristic. We sort $\lambda_i \in \lambda$, based on the ratio of the LHS to the RHS of Eq. (2). If the ratio of λ_i is larger than that of λ_k, we interpret that λ_i is more difficult to satisfy Eq. (2) than λ_j. Therefore, starting from $\lambda^{\mathtt{ded}} = \emptyset$, we repeat the following step until there is no operation to be moved: we select an operation λ_j with the largest ratio among operations in $\lambda \setminus \lambda^{\mathtt{ded}}$ such that $\lambda^{\mathtt{ded}} \cup \{\lambda_j\}$ satisfies Lemma 6 (or Lemma 7), and then add λ_j to $\lambda^{\mathtt{ded}}$.

Another simple strategy for selecting $\lambda^{\mathtt{ded}}$ is the lower-priority-first strategy. That is, as we use fixed-priority scheduling, the lower-priority task tends to violate Eq. (2) (as well as Eq. (4)). Starting from $\lambda^{\mathtt{ded}} = \emptyset$, we scan all $\lambda_j \in \lambda$ from the lowest-priority to highest-priority operations, and check Lemma 6 (or Lemma 7) is satisfied if λ_j is included in $\lambda^{\mathtt{ded}}$. If the lemma satisfied, we add λ_j to $\lambda^{\mathtt{ded}}$; otherwise, we do not add.

4.4 Incorporating Additional Supply to the Proposed Analysis

Section 4.2 calculates $P^{\mathtt{s}}_{\mathtt{uni}}$, the additional guaranteed uniform supply by additional power-supply sources, and Section 4.3 calculates $\lambda^{\mathtt{ded}}$, a set of operations completely powered by a set of additional power-supply sources. The uniform and dedicated approaches utilize $P^{\mathtt{s}}_{\mathtt{uni}}$ and $\lambda^{\mathtt{ded}}$, respectively, and improve the offline power guarantee as follows.

First, when it comes to the uniform supply, we can check a power guarantee of this approach by applying Lemma 1 (or Lemmas 2 and 3) for all $\lambda_k \in \lambda$ and replacing $P^{\mathtt{s}}_{\mathtt{bat}}$ with $P^{\mathtt{s}}_{\mathtt{bat}} + P^{\mathtt{s}}_{\mathtt{uni}}$. Second, as to the dedicated supply, we can check the power guarantee by checking Lemma 1 (or Lemma 2) only with $\lambda_i \in \lambda \setminus \lambda^{\mathtt{ded}}$. Note that for $\lambda_i \in \lambda^{\mathtt{ded}}$, we automatically guarantee their power sufficiency in that $\lambda^{\mathtt{ded}}$ is constructed so as to supply all power demands in $\lambda^{\mathtt{ded}}$ by Γ and $\Gamma_{\mathtt{UC}}$ without $\Gamma_{\mathtt{bat}}$.

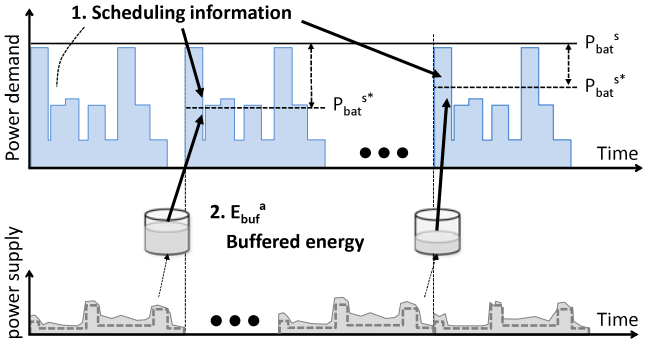


Fig. 2. Static-demand-based online power management: Additional buffered energy can be calculated based on buffer voltage. Future power demand scheduling information and the buffered energy amount are used to determine the buffer power.

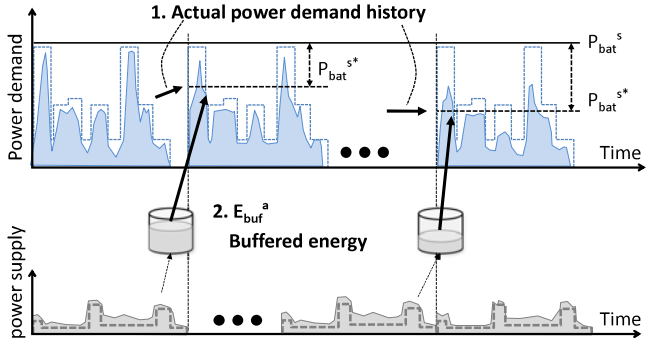


Fig. 3. Predicted-demand-based online power management: Power demand history and the buffered energy amount are used to determine the buffer power.

5 Online Power Management

Thus far, we have explored an offline analysis. The offline analysis provides the system-level power guarantee in our scheduling framework based on power-demand and supply task information. We have also described how the sporadic power supplies can improve the power-guarantee analysis. However, if one performs a set of power-demand operations with a set of power-supply sources that satisfy the offline power guarantee analysis in Section 4, there will be extra energy stored in the energy buffer as the analysis is based on the minimum (not actual) power supply. Thus, we propose an online power management framework, which adaptively controls the power of the energy buffer based on scheduling information. The goal of the online management is to reduce the peak power supplied by the battery while guaranteeing system-power capability that achieves the goal in Section 2.2; the online power management framework determines a desirable battery current bound to achieve the peak power reduction. Then, whenever battery current exceeds the current bound, the system controls the energy buffer to supply the stored energy to the system load so as to limit the battery current. The energy system must be able to supply the required power within its system configurations and energy buffer states.

In this section, we have explored two different approaches to determine the battery current bound. The first approach (called *static-demand-based online power management*) assumes that we can predict future scheduling information of power-demanding tasks and use the information to determine the limits as shown in Fig. 2. For the situation where future power-demand information is not available, the second approach (called *predicted-demand-based online power management*) uses past power requirement information only as seen in Fig. 3; it keeps calculating average and standard deviation of the past power requirement, and determines the battery current bound.

5.1 Static-Demand-Based Online Power Management

The static-demand-based online power management framework periodically controls power of the energy buffer, as shown in Fig. 2 and Algorithm 2. At t_0, the beginning of each period (for online power management) of length t_{p}, we calculate the amount of energy in $\Gamma_{\mathtt{UC}}$, which is necessary for an offline power guarantee for the current period $[t_0, t_0 + t_{\mathrm{p}})$ (denoted by $E^{\mathtt{m}}_{\mathtt{buf}}(t_0)$) as follows. For the uniform power-supply approach in Section 4.2, $E^{\mathtt{m}}_{\mathtt{buf}}(t_0)$ is simply calculated by $t_{\mathrm{p}} \cdot P^{\mathtt{s}}_{\mathtt{uni}}$ since the offline power guarantee exploits the property that $P^{\mathtt{s}}_{\mathtt{uni}}$ of power is *always* supplied by Γ. For the dedicated supply approach in Section 4.3, we calculate $E^{\mathtt{m}}_{\mathtt{buf}}(t_0)$ using the amount of energy consumed by $\lambda^{\mathtt{ded}}$ based on their maximum power demand parameters (i.e., P^{d}_i, not $P^{\mathrm{d}}_i(t)$) during the interval.

Once $E^{\mathtt{m}}_{\mathtt{buf}}(t_0)$ is calculated, we can utilize the energy from $\Gamma_{\mathtt{UC}}$ up to the difference between the amount of total energy stored in $\Gamma_{\mathtt{UC}}$ at t_0 (denoted by $E_{\mathtt{buf}}(t_0)$) and $E^{\mathtt{m}}_{\mathtt{buf}}(t_0)$. Note that we can measure the amount of energy in $\Gamma_{\mathtt{UC}}$ by monitoring the voltage level of $\Gamma_{\mathtt{UC}}$ at t_0 (denoted by $V_{\mathtt{buf}}(t_0)$), using $E_{\mathtt{buf}}(t_0) = \frac{1}{2} \cdot C_{\mathtt{buf}} \cdot V^2_{\mathtt{buf}}(t_0)$, where $C_{\mathtt{buf}}$ is a constant representing the capacitance of $\Gamma_{\mathtt{UC}}$ [15]. Lines 1 and 2 of Algorithm 2 represent the calculation of $E^{\mathtt{m}}_{\mathtt{buf}}(t_0)$ and $E_{\mathtt{buf}}(t_0)$.

Then, we utilize the extra energy up to as much as $E_{\mathtt{buf}}(t_0) - E^{\mathtt{m}}_{\mathtt{buf}}(t_0)$, for reducing the battery's peak power. We use energy from the energy buffer if the power usage of the battery pack is larger than a threshold $P^{\mathtt{s}*}_{\mathtt{bat}}(t_0)$. We determine $P^{\mathtt{s}*}_{\mathtt{bat}}(t_0)$ for the current period $[t_0, t_0 + t_{\mathrm{p}})$, using the scheduling results of the previous period $[t_0 - t_{\mathrm{p}}, t_0)$, which is the worst-case energy consumption by λ during that period. During each scheduling period, we can figure out how much can be stored in a buffer by $P^{\mathtt{s}*}_{\mathtt{bat}}(t_0)$ as follows.

$$f^{t_i}_{\mathtt{P2E}}(P_{\mathrm{s}}) = \begin{cases} -t_i \cdot P_{\mathrm{s}} + t_i \cdot P_i, & \text{if } P^{\mathtt{s}*}_{\mathtt{bat}} < P_i, \\ 0, & \text{otherwise,} \end{cases}$$

where P_i is a power consumption during a sub-period $t_i \in [t_0 - t_{\mathrm{p}}, t_0)$ as seen in Fig. 4. Note that P_i and t_i are not typically the same as P and t of any task.

Then, sum of $f^{t_i}_{\mathtt{P2E}}(P_{\mathrm{s}})$ represents total stored energy in the buffer because of the power threshold ($P^{\mathtt{s}*}_{\mathtt{bat}}$) as shown in Fig. 4. Next, $f_{\mathtt{E2P}}(E)$, the function which finds $P^{\mathtt{s}*}_{\mathtt{bat}}$ to store the amount of buffer energy (E), can be derived by applying the inverse function of $\Sigma_{i=1,\ldots,N} f^{t_i}_{\mathtt{P2E}}(P_{\mathrm{s}})$.

Finally, for given $E^{\mathtt{m}}_{\mathtt{buf}}(t_0)$ and $P^{\mathtt{s}*}_{\mathtt{bat}}(t_0)$, the online power management framework controls the energy stored in the energy buffer within a period $[t_0, t_0 + t_{\mathrm{p}})$ as follows. Suppose that the battery pack should supply X amount of power if there is no supply from $\Gamma_{\mathtt{UC}}$ at t for the purpose of the peak power reduction.[3] Then, the energy buffer supplies $X - P^{\mathtt{s}*}_{\mathtt{bat}}(t_0)$ of power, only when $X \geq P^{\mathtt{s}*}_{\mathtt{bat}}(t_0)$ and $E_{\mathtt{buf}}(t) \geq E^{\mathtt{m}}_{\mathtt{buf}}(t_0)$ hold.

3. The amount of energy $\Gamma_{\mathtt{UC}}$ (the energy buffer) should supply at t for an offline power guarantee already figured in X.

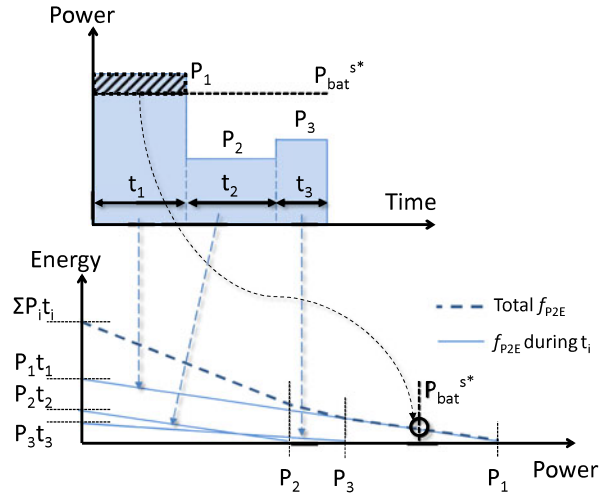


Fig. 4. Calculation of P^{s*}_{bat} using $f_{E2P}(E)$: This relation can be extracted based on power demand scheduling information.

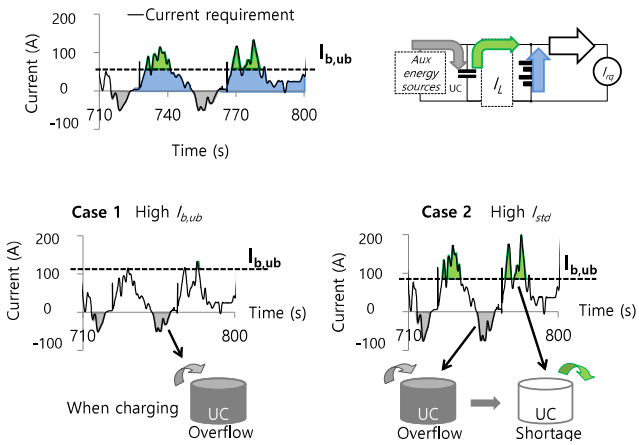


Fig. 5. Example of power buffering for reducing the peak power. Case 1 shows the UC state during charging and discharging if $I_{b,ub}$ and $I_{b,lb}$ are high. Large current bounds limit the discharging current from UC and increase the charging current to UC. Case 2 shows the UC state if actual peak currents are higher (and so I_{std} is higher). Higher (dis)charge currents cause larger (dis)charge current from UC, leading to more frequent UC overflow/shortages.

Algorithm 2. Static-Demand-Based Online Power Management

The following steps are performed at t_0, the beginning of each period of length t_p,

1: Calculate $E^{m}_{buf}(t_0)$ depending on the uniform/dedicated supply approach.
2: Calculate $E_{buf}(t_0)$ by $\frac{1}{2} \cdot C_{buf} \cdot V^2_{buf}(t_0)$.
3: $P^{s*}_{bat}(t_0) \leftarrow f_{E2P}(E_{buf}(t_0) - E^{m}_{buf}(t_0))$

5.2 Predicted-Demand-Based Online Power Management

The goal of online power management is to minimize the battery (dis)charge stress by determining effective (dis)charge current limits in real time. The predicted-demand-based approach to be described in this subsection determines the limits based on the past power requirements and the online buffer energy state. The normal distribution of power requirements has been used to identify the past power requirements patterns. We adapt the formulations and ideas of the authors' recent conference paper [34] for this approach. We first formally state the problem as follows.

Given the power requirement ($I_{rq}(t)$),

Determine the (dis)charge currents from the energy buffer $I_L(t)$, such that

$$\begin{aligned} &\text{Minimize(S4)} \quad \text{Discharge/charge stress} = \frac{1}{T_{op}} \int a \cdot e^{b \cdot |I_b(t)|}\, dt, \\ &\text{Subjectto(S5)} \quad I_{rq}(t) = I_L(t) + I_b(t), \end{aligned} \tag{15}$$

where I_L is a buffer current, a and b are the coefficients in the battery stress model. The energy storage system must supply the required power with current (I_{rq}) to the electric load. As energy conserves, the sum of current supplies from energy storage components (I_L, I_b) must be the power requirement for all $k \in [0, k_e)$ (S5). The objective function in (S4) captures the (dis)charge stress during an operation period of T_{op}.

In this paper, the battery stress is an external condition and electrochemical reaction that affects battery degradation and lifetime [35], [36]. This work utilizes the current degradation model to evaluate the impact of discharge/charge current management on degradation and lifetime, because power buffer can control discharge/charge current and degradation rate. We assume the (dis)charge stress to be exponential with the (dis)charge current, because large (dis)charge current exponentially accelerates the battery degradation by exciting high over-potential between solid-phase and solution-phase voltages [36].

Algorithm 3. Target Current Decision (I_L)

1: **while** 1 **do**
2: **if** $I_{b,ub} < I_{rq,n}$ **then** /* Peak power reduction */
3: $I_L \leftarrow I_{rq} - I_{b,ub}$
4: **else**
5: $I_L \leftarrow 0$
6: **end if**
7: **end while**

Algorithm 3 and Figs. 5 and 6 show the overall algorithm. Lines 2–6 of Algorithm 3 determine the discharge current from the power-dense battery (I_L) to meet the high (dis)charge current requirement (I_{rq}). When I_{rq} is greater than its upper bound ($I_{b,ub}$), the power-dense battery provides energy to reduce the discharge current of the energy-dense battery (Lines 2–3). For the effective power buffering, current upper bounds are critical because they dictate the amount of discharge current of the storage.

Now we describe how to determine the current bounds ($I_{b,ub}$). Algorithm 4 explains the main current bound searching algorithm and its preparation steps. The first preparation step is to calculate the available energy in the buffer (E_c). If we use an ultra-capacitor (UC) as the power-dense energy buffer, we can use the standard equation related to the capacitor energy ($E_c = \frac{1}{2} C_c V^2_{c,rated}$), where $V_{c,rated}$ denotes the UC-rated voltage and C_c denotes the UC capacitance.

Algorithm 4. Determination of the Current Bounds for Peak Current Reduction

$k \leftarrow 1$
while 1 **do**
  $E_c \leftarrow \frac{1}{2} C_c V_{c,rated}^2$
  $T_c \leftarrow \text{DisChargeCycleTimeUpdate}()$
  $I_{avg} \leftarrow \frac{k-1}{k} I_{avg} + \frac{1}{k} I_{rq}$
  $I_{std} \leftarrow \{\frac{k-1}{k} {I_{std}}^2 + \frac{1}{k}(I_{rq} - I_{avg})^2\}^{\frac{1}{2}}$
  $I_{ub} \leftarrow$ Find current bound$(E_c, I_{avg}, I_{std}, T_c)$
  $k \leftarrow \min(k+1, kValueMax)$
  Sleep()
end while

Next, it statistically figures out discharge/charge pattern including average charging/recharging cycle period (T_c), average current (I_{avg}) and its standard deviation (I_{std}). Then, the current bound is determined under the assumption that the required current would follow the normal distribution with average (I_{avg}) and standard deviation (I_{std}), over the next control period. When the average required current (I_{avg}) is high, the current bound I_{ub} should be increased to effectively buffer the recharging energy while reducing the charge stress. This buffered energy will be supplied to the load to mitigate the energy-dense battery discharge current as shown in Fig. 5. The standard deviation of the required current (I_{std}) also affects the determination of I_{ub}. A large standard deviation means the transfer of a large amount of energy with a high peak current. In such a case, I_{ub} should be increased to buffer energy effectively without incurring shortage of energy in the power-dense energy buffer as shown in Fig. 5.

Algorithm 5 and Fig. 6 describe how to determine I_{ub} based on the energy capacity of the buffer and the history of required power. To maximize the utilization of power-dense energy buffer, it should spend most energy in the buffer during the discharging period while minimizing peak discharge current (I_{ub}). This means I_{ub} should not only be set to as a small value as possible, but also make the supplied energy from the buffer (E_{bf}) equal to the stored energy in the buffer (E_c) under the load requirement pattern. In the algorithm, it searches proper I_{ub} meeting these conditions. For each case of I_{ub}, it calculates the supplied energy by integrating buffered power ($P_{bf} = V_b I_{bf} = V_b(i - I_{ub})$) during one charging/recharging cycle (T_c) as:

$$
\begin{aligned}
E_{bf} &= \int_0^{T_c} P_{bf}(t)dt = T_c \int_{I_{ub}}^{\infty} P_{bf}(i)\text{PDF}(i)di \\
&= T_c \int_{I_{ub}}^{\infty} V_b I_{bf}(i)\text{PDF}(i)di \\
&= T_c \int_{I_{ub}}^{\infty} V_b(i - I_{ub})\text{PDF}(i)di \\
&= T_c V_b \left[\int_{I_{ub}}^{\infty} i\text{PDF}(i)di - I_{ub} \int_{I_{ub}}^{\infty} \text{PDF}(i)di \right] \\
&= T_c V_b \left[\int_{I_{ub}}^{\infty} i\text{PDF}(i)di - I_{ub}(1 - \text{CDF}(I_{ub})) \right] \\
&= T_c V_b \Big[I_{std}^2 \text{PDF}(I_{ub}) + (I_{avg} - I_{ub})(1 - \text{CDF}(I_{ub})) \Big], \qquad (16)
\end{aligned}
$$

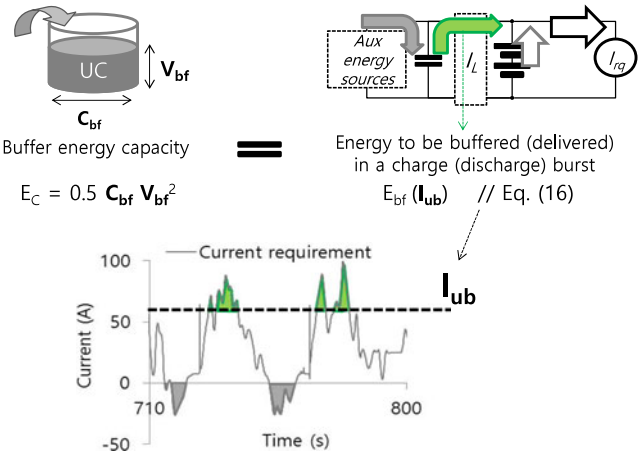


Fig. 6. Determination of current bounds for peak power reduction.

Algorithm 5. Find Current Bound$(E_c, I_{avg}, I_{std}, T_c)$

for $I_{ub} = I_{avg}; I_{ub} < I_{avg} + 3I_{std}; I_{ub} = I_{ub} + \frac{I_{std}}{20}$ **do**
  if $E_{bf}(I_{ub}, I_{avg}, T_c) \geq E_c$ **then**
    Break()
  end if
end for
return I_{ub}

where $\text{PDF}(i)$ (*likewise* $\text{CDF}(i)$) is a probability (*likewise* cumulative) density function of a current i, V_b is the battery output voltage, and P_{bf} (*likewise* I_{bf}) is the buffered power (*likewise* current). In the first line, we transform the buffered power over time, $P_{bf}(t)$, to the buffered power over current, $P_{bf}(i)$.[4] Then, we can statistically calculate the expected buffered energy based on a probability density function of a current, $\text{PDF}(i)$, and buffered current (I_{bf}) as shown in the second line. We assumed battery voltage and system output voltage is the same and constant, so the buffered power can be calculated by multiplying the voltage (V_b) and current flowing through the energy buffer ($I_{bf} = i - I_{ub}$) as described in the third line. After rearranging the terms in the equation, we can derive the equation calculating the amount of the buffered energy which is function of I_{ub}. Note that for a normal distribution, we can calculate $\int_X^Y i\text{PDF}(i)di$ in Eq. (16); see Eq. (17) in the supplement, available online.

6 Evaluation of Offline Power-Supply Guarantee Analysis

In this section, we demonstrate the performance of the offline power-supply guarantee analysis presented in Sections 3 and 4. First, we evaluate the analysis only with a uniform supply in Section 3, and then that with multiple power-supply sources in Section 4.

4. According to the mean ergodic theorem, $\lim_{T\to\infty} \frac{1}{T} \int_0^T X(t)dt \rightarrow \int_{-\infty}^{\infty} X(u)PDF(u)du$; i.e., time average of random process can be approximated by its average over the probability space.

TABLE 1
Parameters of the Power Supply: Renewable Energy Sources Γ_1–Γ_3, the Energy Buffer $\Gamma_{\mathtt{UC}}$, and the Battery Pack $\Gamma_{\mathtt{bat}}$

Tasks	$T^{\mathtt{s}}(s)$	$L^{\mathtt{s}}(s)$	$P^{\mathtt{s}}(W)$
Γ_1	6.0	2.0	3.0
Γ_2	6.0	1.5	3.0
Γ_3	1.5	0.2	0.2
$\Gamma_{\mathtt{UC}}$		3.0	3.0
$\Gamma_{\mathtt{bat}}$			40.0

TABLE 2
Parameters of the Power-Demand Operations: λ_1–λ_5

Tasks	$T^{\mathtt{d}}(s)$	$L^{\mathtt{d}}(s)$	$P^{\mathtt{d}}(W)$
λ_1	6.0	2.0	12.0
λ_2	6.0	2.0	9.6
λ_3	4.0	2.0	6.0
λ_4	4.0	1.0	6.0
λ_5	6.0	4.0	7.2

6.1 Results Without Additional Power-Supply Sources

To evaluate the offline power-supply guarantee analysis with a uniform supply, we need to determine parameters of the battery pack and a set of power-demand operations. For the power capability of the battery pack, we follow the parameters of the case study to be described in Section 7, setting $P^{\mathtt{s}}_{\mathtt{bat}}$ to 40W (see Table 1). For each power-demand operation λ_k, we determine its parameters $T^{\mathtt{d}}_k$, $L^{\mathtt{d}}_k$ and $P^{\mathtt{d}}_k$ as follows. $T^{\mathtt{d}}_k$ is uniformly chosen in $[1.0, 10.0]s$; $L^{\mathtt{d}}_k$ is uniformly distributed in $[0.1, 0.5 \cdot T^{\mathtt{d}}_k]$s; and $P^{\mathtt{d}}_k$ is uniformly chosen in $[5.0, 15.0]$W. We also set ϵ_t and ϵ_p to 0.1s and 0.1W. Note that the scales of the above parameters are determined by considering the parameters of power-demand operations in the case study to be described in Section 7 (see Table 2).

We consider 10 choices of n (1, 2, 3, ..., 10) for the number of operations in each set of power-demand operations. We randomly generate 10,000 operation sets for each n (a total of 100,000 operation sets), and check the ratio of operation sets power-guaranteed by:

- Plain−None: the plain analysis (i.e., Lemma 1), and
- Improved−None: the improved analysis (i.e., Lemmas 2 with Lemma 3).

We would like to mention that the guarantee ratio shown in the Y-axis in Fig. 7 means the ratio of operation sets, which are power-guaranteed by the corresponding analysis and framework. For example, in Fig. 7a, the dotted and solid lines for the X-axis of 4 (i.e., the number of operations) correspond to 78.30 and 88.16 percent, respectively. This means that out of 10,000 operation sets, 7,830 and 8,816 operation sets are

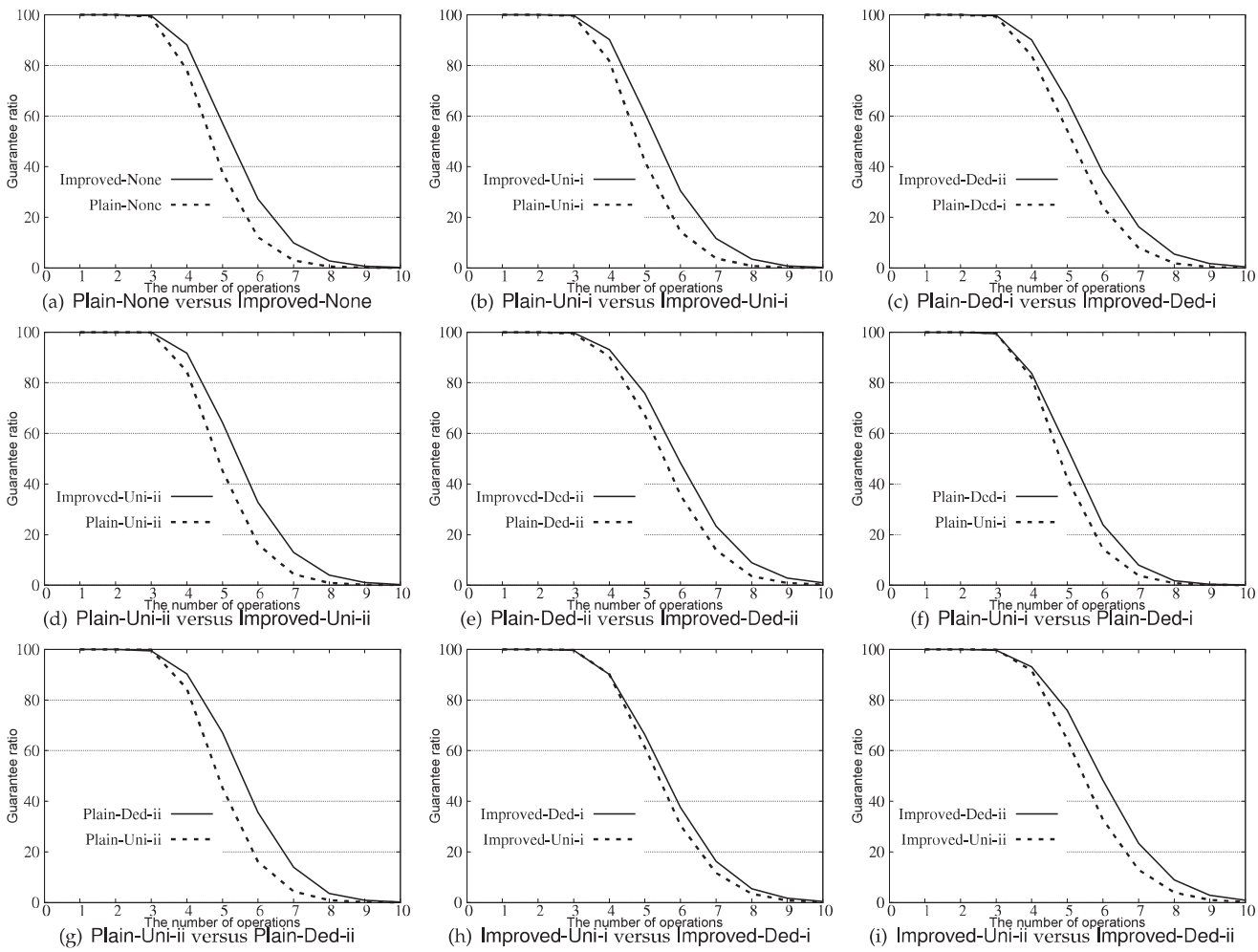


Fig. 7. The ratio of operation sets power-guaranteed under various situations.

power-guaranteed by Plain−None and Improved−None, respectively, when the X-axis is 4.

We have two observations on the results shown in Fig. 7a. First, both Plain−None and Improved−None guarantee the power supply of all operation sets when $n = 1$ or $n = 2$; the ratio of operation sets guaranteed decreases as n increases, and finally, the ratio becomes almost zero when $n = 10$. Second, Improved−None always results in no smaller ratio than Plain−None. The difference between the two ratios is maximized when $n = 5$ (57.22 percent versus 37.73 percent), which demonstrates the effectiveness of the improved analysis compared to the plain analysis.

6.2 Results With Multiple Power-Supply Sources

We now evaluate the offline power-supply guarantee analysis with multiple power-supply sources. Similar to the previous subsection, we need to determine parameters of not only power-supply sources, but also power-demand operations. For power-supply sources, we also follow the parameters of the case study to be described in Section 7. That is, we set $P^{\mathrm{s}}_{\mathtt{bat}}$, $P^{\mathrm{s}}_{\mathtt{UC}}$ and $L^{\mathrm{s}}_{\mathtt{UC}}$ to 40W, 3W and 3s, respectively, and λ_1, λ_2 and λ_3 have the parameters (6,2,3), (6,1.5,3) and (1.5,0.2,0.2), respectively, where (A, B, C) implies $T^{\mathrm{s}} = A$(s), $L^{\mathrm{s}} = B$(s) and $P^{\mathrm{s}} = C$(W), as shown in Table 1 for the case study to be described. We consider two situations for power-supply equipment in addition to $\Gamma_{\mathtt{bat}}$: (i) the system equipped with Γ_1, and (ii) the system equipped with Γ_1, Γ_2 and Γ_3. We will use the same operation sets as the previous section (i.e., 10,000 sets for each n), and check the ratio of operation sets guaranteed by the followings:[5]

- Plain−Uni−i: the uniform supply (Lemma 4) with the plain analysis (Lemma 1) when the additional power-supply is Γ_1,
- Improved−Uni−i: the uniform supply with the improved analysis (i.e., Lemmas 2 with Lemma 3) when the additional power-supply is Γ_1,
- Plain−Ded−i: the dedicated supply (Lemma 6) with the plain analysis when the additional power-supply is Γ_1,
- Improved−Ded−i: the dedicated supply with the improved analysis when the additional power-supply is Γ_1,
- Plain−Uni−ii: the uniform supply with the plain analysis when the additional power-supplies are Γ_1, Γ_2 and Γ_3,
- Improved−Uni−ii: the uniform supply with the improved analysis when the additional power-supplies are Γ_1, Γ_2 and Γ_3,
- Plain−Ded−ii: the dedicated supply with the plain analysis when the additional power-supplies are Γ_1, Γ_2 and Γ_3, and
- Improved−Ded−ii: the dedicated supply with the improved analysis when the additional power-supplies are Γ_1, Γ_2 and Γ_3.

We can summarize the results shown in Figs. 7b, 7c, 7d, 7e, 7f, 7g, 7h, and 7i as follows.

R1. The ratio by both Improved−X−Y and Plain−X−Y decreases as n increases, from 100 percent with $n = 1$ to near-zero with $n = 10$ as shown in all the figures.

R2. Improved−X−Y always results in no smaller ratio than Plain−X−Y, but the amount of improvement of Improved−X−Y over Plain−X−Y varies with the approach (uniform versus dedicated supply) and additional power-supply sources (Γ_1 versus Γ_1, Γ_2 and Γ_3); see Figs. 7b, 7c, 7d, and 7e.

R3. The dedicated supply (i.e., X−Ded−Y) exhibits better performance than the uniform supply (i.e., X−Uni−Y) as shown in Figs. 7f, 7g, 7h, and 7i.

R4. The gap between ratios by X−Ded−Y and X−Uni−Y when X=Improved is smaller than the gap when X=Plain; see Figs. 7f, 7g, 7h, and 7i.

R1 and R2 are straightforward. For R2, the maximum differences between the ratio by Improved−Y−Z and that by Plain−Y−Z for each pair of Y and Z are 19.21 percent ($= 61.3\% - 42.09\%$) with $n = 5$ when Y=Uni and Z=i, 18.92 percent ($= 64.23\% - 45.31\%$) with $n = 5$ when Y=Ded and Z=i, 13.73 percent ($= 37.64\% - 23.91\%$) with $n = 6$ when Y=Uni and Z=ii, and 12.74 percent ($= 48.33\% - 35.59\%$) with $n = 6$ when Y=Ded and Z=ii.

On the other hand, R3 and R4 are non-trivial. Although the ratio by X−Ded−Z seems higher than the ratio by X−Uni−Z in every situation, X−Ded−Z cannot dominate X−Uni−Z,[6] which is different from the dominant relation of Improved−Y−Z over Plain−Y−Z. That is, there exist many operation sets which are power-guaranteed by X−Uni−Y but not by X−Ded−Z; for example, there exist 1.12, 1.56, 0.53 and 0.79 percent such operation sets with $n = 5$ when the pair of (X and Z) is (Plain, i), (Improved, i), (Plain, ii) and (Improved, ii), respectively. Therefore, X−Uni−Z complements X−Ded−Z in terms of offline power guarantee. Also, we can observe R4. The gap between ratio by X−Ded−i and X−Uni−i with $n = 5$ is 12.15 percent ($= 54.24\% - 42.09\%$) when X=Plain, but the gap is reduced to 4.99 percent ($= 66.29\% - 61.3\%$) when X=Improved. Likewise, the same holds between the ratio by X−Ded−ii and by X−Uni−ii: 21.8 percent when X=Plain and 11.66 percent when X=Improved. This is because the improved analysis does not have much room for improvement since it already improved the power guarantee over the plain analysis.

7 Case Study for the Proposed Framework

We now conduct a case study and demonstrate how much real-world systems can benefit from the proposed framework. It consists of the offline power-supply guarantee analysis (in Sections 3 and 4) and online power management (in Section 5). We focus on whether or not they meet the goals stated in Section 2.2. To this end, we first build a simulation model on *Matlab Simulink* and run simulations with the proposed algorithms in the model, in order to demonstrate our framework guarantees power-supply requirement and reduces the energy dissipation. Hardware and parameters in the case study are derived from the prototype consisting of HESS developed in our earlier conference version [30]. We then present experimental results for the simulation

5. For the dedicated supply, we apply the lower-priority-first strategy for selecting $\lambda^{\mathtt{ded}}$.

6. A is said to dominate B when all operation sets power-guaranteed by B are also power-guaranteed by A.

model, demonstrating the HESS's power-supply guarantee and reduction of energy dissipation.

7.1 Case Study With the Simulation Model

We target an actual system as a case study, which is equipped with wheels, wheel motors, stepper motors, coolers and a HESS including a pack of lithium-ion batteries, a pack of UCs, an RBS and solar panels; the actual system comes from the prototype developed in our preliminary conference version [30]. We can determine the parameters of power-demand operations and power-supply sources from their power demand/supply profiles and specifications. Specifications of energy sources/storage are presented in Table 1, and those of the power-demand operations are listed in Table 2. Based on the parameters, we determine the minimum battery capacity achieving a power-supply guarantee for the system via the power guarantee analysis in Sections 3 and 4. We have then executed various sequences of operations while recording battery states to evaluate the proposed system. The applications and tasks considered in the case study are detailed as follows.

Our driving application is programmed to operate wheel motors to achieve the driving profile. Our motor control (λ_1) depends on the PID controller to achieve the required speed, its control interval ($T_1^{\mathtt{d}}$), maximum acceleration time (time to achieve the target speed, $L_1^{\mathtt{d}}$), and the maximum power ($P_1^{\mathtt{s}}$) are 6s, 2s, and 12W, respectively. To reflect various user applications, we also ran applications that sporadically actuate motors (λ_4 and λ_5). Some applications actuate stepper motors to control position (λ_2), while others (λ_3) use thermal fins to regulate temperature for system thermal stability. The power-demand operations are shown in Table 2.

We deployed two types of solar panels. One solar panel can generate 10W, and the other 25W at sunny day. Its performance dropped largely at cloudy. So, we assumed the minimum power rate would be 3W during 1.5 seconds for the small panel and 3W during 2 seconds for the large one every 6 seconds at our application environment (Γ_1 and Γ_2). We also assumed 0.2W of power is regenerated for 0.2 seconds every 1.5 seconds during driving application (Γ_3). We used an ultra capacitor pack for the energy buffer which has 400F capacitance and 2.7V of rated-voltage. We designed a switched mode DC/DC converter to move energy between the UCs and battery cells. For this experiments, we optimized the converter configuration to maximize the converter efficiency at 3W power transfer for 3 seconds ($\Gamma_{\mathtt{UC}}$). We targeted lithium-ion batteries each of which has 1400mAh of energy capacity and 4.2W of the minimum power capability, under the assumption that each cell has 3.0V minimum voltage and 1C-rate of power rate. We have to determine the number of battery cells (which is equivalent to the battery capacity $\Gamma_{\mathtt{bat}}$) to guarantee power capability for the power-demand operations.

When it comes to the simulation model for the case study, we utilize *Matlab Simulink* standard with the *Simscape* model library to simulate the architecture as shown in Fig. 18 in the supplement, available online. The boost DC-DC converter model is used to connect UC and battery. We extracted two RC battery parameters from battery cells which is used in the case study. We used RC parameters of the UC that is described in its data sheet [37].

7.2 Evaluation for the Case Study

In this subsection, we evaluate our offline power-supply guarantee analysis and online power management for the case study.

7.2.1 Offline Power-Supply Guarantee Analysis

As mentioned in the description of the applications and tasks in the case study, we schedule the five operations in Table 2 supplied by the power sources in Table 1. We compare the three frameworks associated with the plain offline power guarantee analysis in Section 3.2 (i.e., Lemma 1, denoted by Plain) and its improved version in Section 3.3 (i.e., Lemma 2 with Lemma 3, denoted by Improved) as follows: (a) the framework without additional supply sources (using $\Gamma_{\mathtt{bat}}$ only), (b) the framework with additional supply sources (using all the supply sources in Table 1), which operate as a uniform supply, and (c) the framework with additional supply sources (using all the supply sources in Table 1), which operate as a dedicated supply. We use the same notations explained in Section 6 as follows:

- Plain−None: (a) associated with the plain analysis,
- Improved−None: (a) associated with the improved analysis,
- Plain−Uni−ii: (b) associated with the plain analysis,
- Improved−Uni−ii: (b) associated with the improved analysis,
- Plain−Ded−ii: (c) associated with the plain analysis, and
- Improved−Ded−ii: (c) associated with the improved analysis.

Note that we use the same notations as Section 6.

If we apply Plain−None, Lemma 1 cannot guarantee the timely execution of λ_5 (as well as λ_4) with power requirement. That is, during the interval of length $(T_5^{\mathtt{d}} - L_5^{\mathtt{d}} + \epsilon_t) = 6.0 - 4.0 + 0.1 = 2.1$, $\sum_{\lambda_i \in \lambda \setminus \{\lambda_5\}} I_{5 \leftarrow i}(T_5^{\mathtt{d}} - L_5^{\mathtt{d}} + \epsilon_t)$ is the sum of $2.1 \cdot 12.0 + 2.1 \cdot 9.6 + 2.1 \cdot 6.0 + 2.1 \cdot 6.0 = 2.1 \cdot 33.6 = 70.56$, which is no less than $(P_{\mathtt{bat}}^{\mathtt{s}} - P_5^{\mathtt{d}} + \epsilon_p) \cdot (T_5^{\mathtt{d}} - L_5^{\mathtt{d}} + \epsilon_t) = (40.0 - 7.2 + 0.1) * (3.0 - 1.0 + 0.1) = 32.9 \cdot 2.1 = 69.09$. However, if we apply Improved − None, Lemma 2 with Lemma 3 guarantees that every power demand operation in Table 2 can finish its execution within its deadline, without experiencing power deficiency. That is, after three iterations for finding slack values in Lemma 3, the slack values of λ_1, λ_2, λ_3, λ_4 and λ_5 are 1.6, 1.6, 0.1, 0.7, 0.0, respectively; then, during the interval of length $(T_5^{\mathtt{d}} - L_5^{\mathtt{d}} + \epsilon_t) = 6.0 - 4.0 + 0.1 = 2.1$, $\sum_{\lambda_i \in \lambda \setminus \{\lambda_5\}} I_{5 \leftarrow i}(2.1)$ is the sum of $2.0 \cdot 12.0 + 2.0 \cdot 9.6 + 2.0 \cdot 6.0 + 1.4 \cdot 6.0 = 63.6$, which is less than 69.09.

We now explain how Improved−None calculates proper power requirement, compared to MAX and AVG, which denote the sum of the maximum power demand (i.e., $\sum_{\lambda_i \in \lambda} P_i^{\mathtt{d}} = 40.8W$), and the average of the maximum power demand (i.e., $\sum_{\lambda_i \in \lambda} \frac{P_i^{\mathtt{d}} \cdot L_i^{\mathtt{d}}}{T_i^{\mathtt{d}}} = 16.5W$). First, we can find the lowest battery capacity $\Gamma_{\mathtt{bat}}$ that makes Improved−None give power supply guarantee, which is $\Gamma_{\mathtt{bat}} = 38.0W$.[7] Then, Improved−None (38.0W) reduces the required battery capability. Also, it is straightforward that, if the battery

7. We can find such battery capacity by applying binary search.

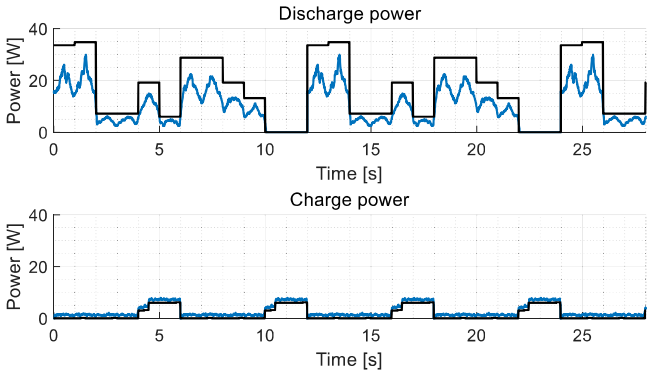


Fig. 8. Power demand and supply simulation with noise (*blue line*) and without noise (*black line*).

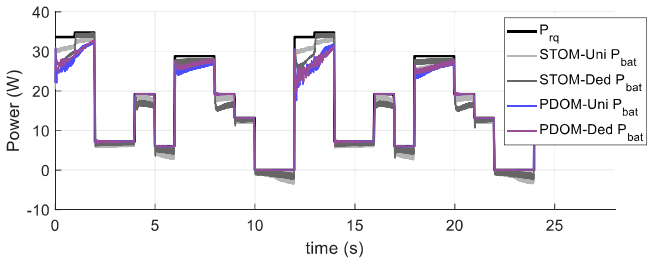


Fig. 9. Power scheduling simulation results with worst-case power demand and supply.

capacity is as much as AVG, we cannot supply the sufficient power in the worst case.

When it comes to Improved−Uni−ii and Improved−Ded − ii, they further reduce the required battery capability by consuming 36.3W and 34.8W, respectively.[8] Between the two, Improved−Ded−ii is the most effective framework for a power guarantee, in that it can reduce the required battery capability by 14.7 percent, compared to MAX, a naive approach without utilizing renewable energy sources.

7.2.2 Online Power Management

We target the case study explained in Section 7.1, and apply the simulation model to power demand and supply scheduling with our online management schemes. The worst-case power-demand operations are scheduled according to Table 2. The generated power is supplied into the energy buffer according to power-supply sources in Table 1. Fig. 8 shows the square-shaped worst-case power demand and supply results. We also generated more realistic total power demand and supply based on actual power supply/demand observations; we smoothed them and added uniform random noises. We then evaluated the schemes with ideal power supply and demand profiles and their smoothed profiles. That is, we consider two situations: (Sit1) the power demand and supply always exhibit their worst-case; and (Sit2) the power demand and supply include noises, which is more realistic. Under these situations, we have evaluated the following four online power management schemes:

8. Note that Plain−Ded−ii and Improved−Ded−ii choose λ_4 to which the additional power-supply sources are dedicated.

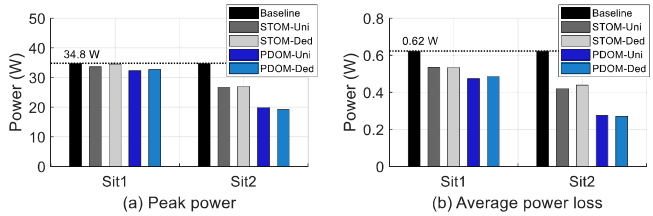


Fig. 10. Peak power (a) and average power loss (b) under Sit1 and Sit2.

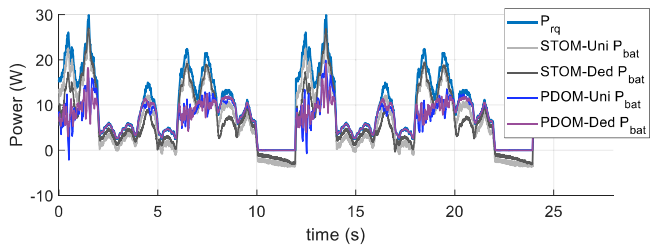


Fig. 11. Power scheduling simulation results with power supply and demand including noises.

- STOM−Uni: static-demand-based online management with the uniform supply,
- STOM−Ded: static-demand-based online management with the dedicated supply,
- PDOM−Uni: predicted-demand-based online management with the uniform supply, and
- PDOM−Ded: predicted-demand-based online management with dedicated supply.

We first evaluate the situation Sit1. Fig. 9 shows the result battery current under worst-case power supply and demand with the proposed online management methods, where P_{rq} and P_{bat} denote the total power demand and the amount of power supplied by the battery pack, respectively. STOM−Uni reduces battery power requirement evenly, while STOM−Ded decreases battery power requirement only when the dedicated tasks runs. PDOM−Uni and PDOM−Ded keep updating the current bound and limit battery current based on the current bound. STOM−Uni, STOM−Ded, PDOM−Uni, and PDOM−Ded reduce peak current to 3.44, 0.72, 7.30, and 6.01 percent, respectively, compared to P_{rq}. Detailed values under Sit1 can be seen in Fig. 10a.

We also note the average power dissipation is calculated by $P_{\mathrm{loss}} = \frac{1}{t_{\max}} \int_0^{t_{\max}} P^{\mathrm{s}}_{\mathrm{bat-loss}}(t)\, dt = \frac{1}{t_{\max}} \int_0^{t_{\max}} \Sigma I^2_{\mathrm{bat,i}}(t) R_{\mathrm{bat,i}}(t)\, dt$. STOM−Uni, STOM−Ded, PDOM−Uni, and PDOM−Ded decrease the average power dissipation by 14.07, 14.23, 23.88 and 22.01 percent, respectively, compared to P_{rq}. Detailed values under Sit1 can be seen in Fig. 10b.

Next, we evaluate the situation Sit2. Fig. 11 represents battery power of each approach with the more realistic power requirements. Fig. 12 shows the detailed peak power reductions. STOM−Uni, STOM−Ded, PDOM−Uni, and PDOM−Ded could decrease peak current respectively to 23.36, 22.44, 42.98 and 44.93 percent during the given operations, compared to P_{rq}. The average power dissipation of STOM−Uni, STOM−Ded, PDOM−Uni, and PDOM−Ded are reduced to 32.60, 29.48, 55.79, and 56.49 percent, respectively, compared to P_{rq}. Fig. 10a and (b) under Sit2 show the performance of the approaches under Sit2.

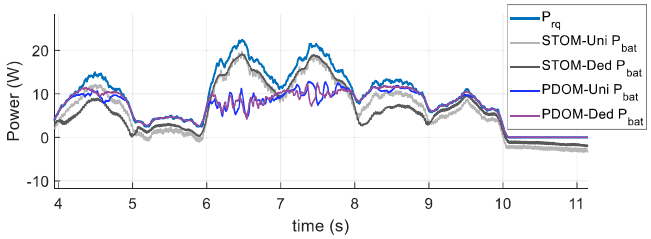


Fig. 12. Power scheduling simulation results with power supply and demand including noises (4s - 10s).

We can interpret the evaluation results as follows. First, if we compare the reduction of the peak current by the four online power management schemes in Sit1, with that in Sit2, that in Sit2 is more significant than that in Sit1. This is because Sit2 exhibits less power consumption of power-demand operations than their worst-case and more power generation by power-supply sources than their minimum; this yields more room to reduce the peak current, and our proposed online management schemes effectively exploit the room. Second, if we compare STOM$-^*$ and PDOM$-^*$, the latter yields better performance, since PDOM$-^*$ decides the current upper bound in real-time based on the average and standard deviation of recent current requirement instead of using worst-case task power requirement. This enables PDOM$-^*$ to reduce the peak current more aggressively while providing the buffer energy to the load without the buffer energy shortage. This improved peak power reduction leads to the lower average energy dissipation compared to STOM$-^*$. Third, while $^*-$Uni reduces the current uniformly, so peak current is always reduced by a certain amount, the peak current reduction of $^*-$Ded depends on which task is dedicated. Since the dedicated task (λ_4 in the case study) did not contribute the peak current in Sit1 and Sit2, $^*-$Uni would be more beneficial for peak current reduction. However, if dedicated task contributes the system peak current, $^*-$Ded could reduce peak power largely. Results in Figs. 9 and 11 show this trend clearly.

8 Conclusion

In this paper, we have proposed a power scheduling framework—as a guideline for the design of a HESS—that ensures power sufficiency for the worst-case power demand and supply using real-time scheduling schemes. To improve the runtime performance of the HESS further, we have also designed an online power management framework that utilizes the surplus energy from a real-time power supply, reducing the battery's peak power and hence extending its lifetime. We have evaluated the effectiveness of the offline power-supply guarantee using simulations, and validated the design with a simulation model based on the HESS-powered actual system running realistic applications, demonstrating power sufficiency with a lower-cost HESS and higher energy-efficiency.

In future, we would like to develop a design framework for general energy storage systems based on a power guarantee analysis. It will search for the optimal configurations of energy storage systems at their design or replacement time while considering power demand history and power supply's state-of-health. We also plan to build a power/energy management system that not only schedules power demand and supply, but also monitors and pro/diagnoses energy storages/sources.

Acknowledgments

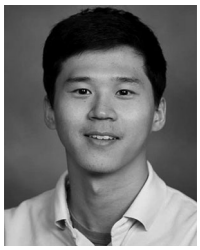
An earlier (shorter) version of this article was presented at the IEEE RTSS 2016 [30]. The work reported in this article was supported in part by the NSF under Grants CNS-1446117 and CNS-1739577, in part by the ONR under Grant No. N00014-18-1-2141, and by LG Chem Ltd. This research was also supported in part by the National Research Foundation of Korea (NRF) funded in part by the Ministry of Science and ICT under Grant 2019R1A2B5B02001794 and 2017H1D8A2031628. This research was also supported by the Ministry of Science and ICT, Korea, under the Grand Information Technology Research Center support program (IITP-2020-2015-0-00742) supervised by the IITP.

References

[1] P. D. Blair, "Modeling energy and power requirements of electric vehicles," *Energy Convers.*, vol. 18, no. 3, pp. 127–134, 1978.
[2] G. Cai, L. Feng, B. M. Chen, and T. H. Lee, "Systematic design methodology and construction of UAV helicopters," *Mechatronics*, vol. 18, no. 10, pp. 545–558, 2008.
[3] E. Pastor, J. Lopez, and P. Royo, "UAV payload and mission control hardware/software architecture," *IEEE Aerosp. Electron. Syst. Mag.*, vol. 22, no. 6, pp. 3–8, Jun. 2007.
[4] C. Chan and K. Chau, "An overview of power electronics in electric vehicles," *IEEE Trans. Ind. Electron.*, vol. 44, no. 1, pp. 3–13, Feb. 1997.
[5] T.-K. Lee, B. Adornato, and Z. Filipi, "Synthesis of real-world driving cycles and their use for estimating PHEV energy consumption and charging opportunities: Case study for midwest/U.S." *IEEE Trans. Veh. Technol.*, vol. 60, no. 9, pp. 4153–4163, Nov. 2011.
[6] E. Kim, J. Lee, and K. G. Shin, "Real-time prediction of battery power requirements for electric vehicles," in *Proc. ACM/IEEE 4th Int. Conf. Cyber-Physical Syst.*, Apr. 2013, pp. 11–20.
[7] Z. Li *et al.*, "On the feasibility of linear discrete-time systems of the green scheduling problem," in *Proc. IEEE 32nd Real-Time Syst. Symp.*, Nov. 2011, pp. 295–304.
[8] "Peak shaving through real-time scheduling of household appliances," *Energy Buildings*, vol. 75, pp. 133–148, 2014.
[9] T. Facchinetti and M. L. D. Vedova, "Real-time modeling for direct load control in cyber-physical power systems," *IEEE Trans. Ind. Informat.*, vol. 7, no. 4, pp. 689–698, Nov. 2011.
[10] G. Benetti, M. Delfanti, T. Facchinetti, D. Falabretti, and M. Merlo, "Real-time modeling and control of electric vehicles charging processes," *IEEE Trans. Smart Grid*, vol. 6, no. 3, pp. 1375–1385, May 2015.
[11] S. Pay and Y. Baghzouz, "Effectiveness of battery-supercapacitor combination in electric vehicles," in *Proc. IEEE Bologna Power Tech Conf.*, 2003, Art. no. 6.
[12] R. Kaiser, "Optimized battery-management system to improve storage lifetime in renewable energy systems," *J. Power Sources*, vol. 168, no. 1, pp. 58–65, 2007.
[13] S. Park, Y. Kim, and N. Chang, "Hybrid energy storage systems and battery management for electric vehicles," in *Proc. 50th ACM / EDAC / IEEE Des. Automat. Conf.*, 2013, pp. 1–6.
[14] J. Cao and A. Emadi, "A new battery/ultracapacitor hybrid energy storage system for electric, hybrid, and plug-in hybrid electric vehicles," *IEEE Trans. Power Electron.*, vol. 27, no. 1, pp. 122–132, Jan. 2012.
[15] E. Kim, K. Shin, and J. Lee, "Real-time discharge/charge rate management for hybrid energy storage in electric vehicles," in *Proc. IEEE Real-Time Syst. Symp.*, 2014, pp. 228–237.
[16] M. Ortuzar, J. Moreno, and J. Dixon, "Ultracapacitor-based auxiliary energy system for an electric vehicle: Implementation and evaluation," *IEEE Trans. Ind. Electron.*, vol. 54, no. 4, pp. 2147–2156, Aug. 2007.
[17] J. K. Shiau, D. M. Ma, P. Y. Yang, G. F. Wang, and J. H. Gong, "Design of a solar power management system for an experimental uav," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 45, no. 4, pp. 1350–1360, Oct. 2009.

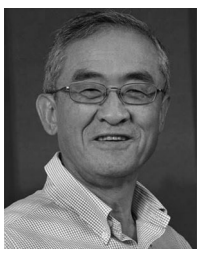
[18] N. Baldock and M. Mokhtarzadeh-Dehghan, "A study of solar-powered, high-altitude unmanned aerial vehicles," *Airc. Eng. Aerosp. Technol.*, vol. 78, no. 3, pp. 187–193, 2006.
[19] M. Z. Jacobson and M. A. Delucchi, "Providing all global energy with wind, water, and solar power, part I: Technologies, energy resources, quantities and areas of infrastructure, and materials," *Energy Policy*, vol. 39, no. 3, pp. 1154–1169, 2011.
[20] J. Paska, P. Biczel, and M. Kłos, "Hybrid power systems – an effective way of utilising primary energy sources," *Renewable Energy*, vol. 34, no. 11, pp. 2414–2421, 2009.
[21] S. Letendre, R. Perez, and C. Herig, "Battery-powered, electric-drive vehicles providing buffer storage for PV capacity value," in *Proc, Solar Conf.*, 2002, pp. 105–110.
[22] R. Baños, F. Manzano-Agugliaro, F. Montoya, C. Gil, A. Alcayde, and J. Gómez, "Optimization methods applied to renewable and sustainable energy: A review," *Renewable Sustain. Energy Rev.*, vol. 15, no. 4, pp. 1753–1766, 2011.
[23] O. Erdinc and M. Uzunoglu, "Optimum design of hybrid renewable energy systems: Overview of different approaches," *Renewable Sustain. Energy Rev.*, vol. 16, no. 3, pp. 1412–1425, 2012.
[24] E. Kim, J. Lee, and K. G. Shin, "Modeling and real-time scheduling of large-scale batteries for maximizing performance," in *Proc. IEEE Real-Time Syst. Symp.*, 2015, pp. 33–42.
[25] W. Zhou, C. Lou, Z. Li, L. Lu, and H. Yang, "Current status of research on optimum sizing of stand-alone hybrid solar–wind power generation systems," *Appl. Energy*, vol. 87, no. 2, pp. 380–389, 2010.
[26] M. Majima, S. Ujiie, E. Yagasaki, K. Koyama, and S. Inazawa, "Development of long life lithium ion battery for power storage," *J. Power Sources*, vol. 101, no. 1, pp. 53–59, 2001.
[27] S. S. Choi and H. S. Lim, "Factors that affect cycle-life and possible degradation mechanisms of a li-ion cell based on licoo2," *J. Power Sources*, vol. 111, no. 1, pp. 130–136, 2002.
[28] H. Kim and K. G. Shin, "Scheduling of battery charge, discharge, and rest," in *Proc. 30th IEEE Real-Time Syst. Symp.*, 2009, pp. 13–22.
[29] L. Benini, A. Macii, E. Macii, M. Poncino, and R. Scarsi, "Scheduling battery usage in mobile systems," *IEEE Trans. VLSI Syst.*, vol. 11, no. 6, pp. 1136–1143, Dec. 2003.
[30] E. Kim, Y. Lee, L. He, K. G. Shin, and J. Lee, "Offline guarantee and online management of power demand and supply in cyber-physical systems," in *Proc. IEEE Real-Time Syst. Symp.*, 2016, pp. 89–98.
[31] J. Zhang, S. Ci, H. Sharif, and M. Alahmad, "Modeling discharge behavior of multicell battery," *IEEE Trans. Energy Convers.*, vol. 25, no. 4, pp. 1133–1141, Dec. 2010.
[32] M. Bertogna and M. Cirinei, "Response-time analysis for globally scheduled symmetric multiprocessor platforms," in *Proc. IEEE Real-Time Syst. Symp.*, 2007, pp. 149–160.
[33] M. Bertogna, M. Cirinei, and G. Lipari, "Schedulability analysis of global scheduling algorithms on multiprocessor platforms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 4, pp. 553–566, Apr. 2009.
[34] E. Kim and K. G. Shin, "Optimal design and management of a hybrid energy storage system," in *Proc. IEEE Conf. Decis. Control*, 2019, pp. 208–215.
[35] K. Uddin, S. Perera, W. D. Widanage, L. Somerville, and J. Marco, "Characterising lithium-ion battery degradation through the identification and tracking of electrochemical battery model parameters," *Batteries*, vol. 2, no. 2, 2016. [Online]. Available: http://www.mdpi.com/2313-0105/2/2/13
[36] X. Lin, J. Park, L. Liu, Y. Lee, A. M. Sastry, and W. Lu, "A comprehensive capacity fade model and analysis for li-ion batteries," *J. Electrochemical Society*, vol. 160, no. 10, pp. A1701–A1710, 2013.
[37] EATON, Snap-in cylindrical supercapacitors, Accessed: Feb. 17, 2020. [Online]. Available: https://www.eaton.com/us/en-us/catalog/electronic-components/xv-supercapacitor.html

Eugene Kim received the BS degree in electrical engineering from Seoul National University, Republic of Korea, in 2011, and the MS and PhD degrees in computer science and engineering from the University of Michigan, Ann Arbor, in 2017. He is currently a battery algorithm engineer at Faraday Future. His research interests include cyber-physical systems and battery management system for electric vehicles, spacecraft, and mobile systems.



Youngmoon Lee (Student Member, IEEE) received the BS degre in electrical and computer engineering from Seoul National University, Korea, in 2014, the MS degree in computer science and engineering from the University of Michigan, Ann Arbor, in 2016, and the PhD degree in computer science and engineering from the University of Michigan, Ann Arbor, in 2019. He is currently an assistant professor with the Department of Robotics, Hanyang University, Korea. His research interests include cyber-physical systems, embedded systems, and mobile computing.

Liang He (Senior Member, IEEE) is currently an assistant professor with the University of Colorado Denver. He worked as a research fellow with the University of Michigan, Ann Arbor, during 2015–2017, as a research scientist with the Singapore University of Technology and Design during 2012–2014, and as a research assistant with the University of Victoria during 2009–2011. His research interests include CPSes, cognitive battery management, and networking. He has been a recipient of the best paper/poster awards of MobiSys17, QShine14, WCSP11, and GLOBECOM11, and a Best Paper Candidate of GLOBECOM14.



Kang G. Shin (Life Fellow, IEEE) is the Kevin & Nancy O'Connor professor of Computer Science with the Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor. His current research focuses on QoS-sensitive computing and networking as well as on embedded real-time and cyber-physical systems. He has supervised the completion of 85 PhDs, and authored/coauthored about 1,000 technical articles, a textbook and more than 40 patents or invention disclosures, and received numerous best paper awards, including 2019 Caspar Bowden Award for Outstanding Research in Privacy Enhancing Technologies, the Best Paper Awards from the 2011 ACM International Conference on Mobile Computing and Networking (MobiCom'11), the 2011 IEEE International Conference on Autonomic Computing, the 2010 and 2000 USENIX Annual Technical Conferences, as well as the 2003 IEEE Communications Society William R. Bennett Prize Paper Award and the 1987 Outstanding IEEE Transactions of Automatic Control Paper Award. He has also received several institutional awards, including the Research Excellence Award in 1989, Outstanding Achievement Award in 1999, Distinguished Faculty Achievement Award in 2001, and Stephen Attwood Award in 2004 from The University of Michigan; a Distinguished Alumni Award of the College of Engineering, Seoul National University in 2002; 2003 IEEE RTC Technical Achievement Award; and 2006 Ho-Am Prize in Engineering (the highest honor bestowed to Korean-origin engineers). He was a co-founder of a couple of startups and also licensed some of his technologies to industry.

Jinkyu Lee (Member, IEEE) received the BS, MS, and PhD degrees in computer science from the Korea Advanced Institute of Science and Technology (KAIST), Republic of Korea, in 2004, 2006, and 2011, respectively. He is an assistant professor with the Department of Computer Science and Engineering, Sungkyunkwan University (SKKU), Republic of Korea, where he joined in 2014. He has been a visiting scholar/research fellow with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI. in 2011–2014. His research interests include system design and analysis with timing guarantees, QoS support, and resource management in real-time embedded systems, mobile systems, and cyber-physical systems. He won the Best Student Paper Award from the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) in 2011, and the Best Paper Award from the 33rd IEEE Real-Time Systems Symposium (RTSS) in 2012.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Local-Density Subspace Distributed Clustering for High-Dimensional Data

Yangli-ao Geng, Qingyong Li, *Member, IEEE*, Mingfei Liang, Chong-Yung Chi, *Fellow, IEEE*, Juan Tan, and Heng Huang

Abstract—Distributed clustering is emerging along with the advent of the era of big data. However, most existing established distributed clustering methods focus on problems caused by a large amount of data rather than caused by the large dimension of data. Consequently, they suffer the "curse" of dimensionality (e.g., poor performance and heavy network overhead) when high-dimensional (HD) data are clustered. In this article, we propose a distributed algorithm, referred to as Local Density Subspace Distributed Clustering (LDSDC) algorithm, to cluster large-scale HD data, motivated by the idea that a local dense region of a HD dataset is usually distributed in a low-dimensional (LD) subspace. LDSDC follows a local-global-local processing structure, including grouping of local dense regions (atom clusters) followed by subspace Gaussian model (SGM) fitting (flexible and scalable to data dimension) at each sub-site, merging of atom clusters at every sub-site according to the merging result broadcast from the global site. Moreover, we propose a fast method to estimate the parameters of SGM for HD data, together with its convergence proof. We evaluate LDSDC on both synthetic and real datasets and compare it with four state-of-the-art methods. The experimental results demonstrate that the proposed LDSDC yields best overall performance.

Index Terms—High-dimensional clustering, distributed clustering, density-base clustering, subspace Gaussian model

1 Introduction

Clustering, as a branch of unsupervised learning, is a process of discovery and exploration for investigating inherent and hidden structures within a large dataset [1]. To be specific, clustering is to partition a set of data samples (or observations) into subsets. Each subset is a cluster, such that samples in a cluster are similar to one another, yet dissimilar to samples in other clusters. Clustering has been extensively applied to a variety of data-analysis tasks [2], [3]. However, two main challenges are faced by centralized clustering algorithms when handling big data. First, they require that all data are collected in a central machine, which may be prohibited in the practical scenario where a large amount of heterogeneous, complex data reside on different, distributed working computers which are connected to each other via local or wide area networks [4]. Second, large size and high dimension of big data may prohibit any clustering algorithm from operating in a single machine due to efficiency and cost considerations.

To meet these challenges, many clustering methods have been proposed to process big data in a distributed system. Existing distributed clustering algorithms can be basically divided into four groups: partition-based methods [5], grid-based methods [6], density-based methods [7] and model-based methods [7]. Partition-based methods employ an iterative strategy to divide data into a certain of clusters according to some predefined quantitative optimization criteria. Grid-based methods quantize the original data space into a finite number of grids, and then group the grids according to statistical characteristics of samples in each grid. Density-based methods first estimate the distribution density of objects in a feature space, and then search for high density regions (i.e., clusters) separated by regions of lower density. Model-based methods first optimize the fitting of local data for some (predefined) parametric statistical models, and then the estimated parameters are collected and used in a central machine to form clusters. There is no doubt that distributed clustering has become a fundamental tool for high-level big data analysis.

Recently, clustering of high-dimensional (HD) data in a distributed environment is emerging due to the rapid growth of data volume and the flourishing of distributed computing technologies, with possible applications including text grouping [8], image retrieval [9] and hyperspectral image analysis [10]. However, HD data involve several issues to distributed clustering methods. The first one is that the shapes of clusters become more complex as the dimension increases. Various shapes may bring about an intractable challenge to partition-based methods, which perform well mainly for convex clusters

- *Y.-A. Geng and Q. Li are with the Beijing Key Lab of Transportation Data Analysis and Mining, Beijing Jiaotong University, Beijing 100044, China. E-mail: {gengyla, liqy}@bjtu.edu.cn.*
- *M. Liang is with WeiXin Group, Tencent Company Limited, Beijing 100044, China. E-mail: mingfeiliang@bjtu.edu.cn.*
- *C.-Y. Chi is with Institute of Communications Engineering, National Tsing Hua University, Hsinchu 30013, Taiwan. E-mail: cychi@ee.nthu.edu.tw.*
- *J. Tan is with the Department of Business Administration, Beijing Technology and Business University, Beijing 100048, China. E-mail: tanjuan@btbu.edu.cn.*
- *H. Huang is with the Department of Electrical and Computer Engineering, University of Pittsburgh, Pittsburgh, PA 15260, and is also with JD Finance America Corporation. E-mail: heng.huang@pitt.edu.*

Manuscript received 10 Sept. 2019; revised 9 Jan. 2020; accepted 9 Feb. 2020. Date of publication 24 Feb. 2020; date of current version 23 Mar. 2020. (Corresponding authors: Juan Tan and Qingyong Li.) Recommended for acceptance by Q. Zheng. Digital Object Identifier no. 10.1109/TPDS.2020.2975550

1045-9219 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

[11]. The second one is that the number of possible grids grows exponentially with the number of dimensions. This exponential increase may impose heavy overheads on both computation and transmission of clustering information for grid-based methods [12]. The third one is that data samples become increasingly sparse as the dimension increases, thereby inflating the variance of estimated density and thus yielding a performance degradation to density-based algorithms [11]. The last one is that the parameter estimation for HD models is computation consuming. As a result, model-based methods usually need to compromise between performance accuracy and computation cost [13]. Overall, clustering HD distributed data effectively and efficiently is still an open problem.

Motivated by an empirical observation that a local dense region of a HD dataset is usually embedded in a low-dimensional (LD) subspace [14], [15], this paper proposes a novel approach for distributed clustering of HD datasets, referred to as Local-Density Subspace Distributed Clustering (LDSDC) algorithm, that operates in a local-global-local fashion. First, in each sub-site, LDSDC exploits a density-based method to divide its dataset into local regions. Second, the data in each local region is fitted by a subspace Gaussian model (SGM), which is flexible and scalable to the data dimension. Third, in the global site, LDSDC collects all the local SGMs from sub-sites, and based on them generates a global density distribution. Furthermore, local SGMs are merged according to "connection values" (to be defined in (6) below) under the global density distribution, and then the resulting merging information are broadcasted to all the sub-sites. Lastly, in each sub-site, local regions are merged into final clusters according to the received merging information. In brief, this work makes the following contributions.

1) A distributed clustering algorithm LDSDC based on SGM is proposed to handle large-scale and HD data.
2) An illustrative analysis of SGM is presented to demonstrate its superiority in high-dimensional data modeling.
3) A fast method for the estimation of SGM parameters is proposed, including an algorithm and its convergence proof.

A preliminary version of this work was published in ICPADS 2017 [16], in which a distributed algorithm named REMOLD is proposed. In this paper, it is significantly extended from both theoretical and practical aspects. Specifically, beyond the general Gaussian model utilized by REMOLD, the SGM (a compact statistical model) is considered for local cluster modeling, which proves efficient and effective in HD data clustering. Then we present a theoretical analysis to demonstrate its superiority over general Gaussian model in HD data modeling. Furthermore, a fast parameter-estimation method for the SGM is proposed. Then extensive experiments are presented to show the effectiveness of the proposed LDSDC by clustering HD data (instead of two-dimensional datasets adopted in [16]), including various synthetic datasets for different scales, dimensions and cluster shapes. Finally, we present some experimental results on real HD datasets, which further show the efficacy and practicability of the proposed LDSDC.

The rest of the paper is organized as follows. Section 2 introduces the related works and notations. Section 3 presents the proposed clustering method LDSDC. Section 4 details subspace Gaussian model. Section 5 provides some simulation results and experimental results to demonstrate the efficacy of the proposed LDSDC. Finally, we conclude the paper in Section 6.

2 Related Works and Notations

2.1 Distributed Clustering

Distributed clustering aims to handle data inherently residing on distributed sites, which cannot be collected in a central site due to practical issues such as privacy concerns and limited transmission bandwidth [17]. Unlike centralized clustering, distributed clustering relies heavily on the network structure because different networks have their own limitations and task objectives [18]. In general, distributed clustering methods are designed for two network topologies: global-sub-site networks and peer-to-peer (P2P) networks [19]. The global-sub-site paradigm demands a reliable global site to collect all necessary information from distributed sub-sites. By contrast, the P2P paradigm does not have a central server and all sites with limited view coverage of the entire network only exchange necessary information to perform their own local clustering tasks [18], [19]. In this paper, we focus on distributed clustering methods for the global-sub-site networks.

The well-known method, k-means [20] is the basis of many distributed methods. To avoid the exposure of private data in the transmission, Vaidya and Clifton [21] proposed a distributed k-means with the privacy-protection taken into consideration. It can achieve good performance when different sites contain different private attributes for a common set of entities. Distributed k-means algorithm is basically an ensemble-learning-based distributed clustering algorithm introduced in [22]. This method first does clustering in sub-site using k-means, then send all centroid values to the central site, and finally global centroid values of the underlying global clustering are obtained by using k-means again [23]. Jeong *et al.* [24] proposed a modified distributed k-means algorithm for clustering huge amount of distributed biological data. Balcan *et al.* [25] provided a distributed method for constructing a global core-set of data samples, based on which k-means clustering can handle distributed data efficiently. Overall, these methods are efficient enough to deal with big data, but their performance is on par with that of the conventional k-means method.

In [26], Xu *et al.* presented a parallel clustering method, based on the conventional centralized clustering algorithm DBSCAN [27]. First of all, it divides the dataset into several partitions at the global site, and then send each partition to a sub-site, where the partition is locally clustered by using DBSCAN. Finally, the clustering results obtained from the sub-sites are merged at the global site. In [4], a DBSCAN-based distributed clustering algorithm named DBDC was proposed. It first clusters data at each sub-site, and some specific core samples and the associated neighborhood parameters are extracted at the same time, which are then collected at a global site. Based on these core samples and parameters, DBSCAN is used to generate the final clusters at the global site. Though DBDC can efficiently find clusters with arbitrary shapes, the excessive input-parameters and high performance sensitivity to the input-parameters weaken its practical

applicability. Accordingly, approaches such as SDBDC [28] and LDBDC [29] have been proposed to improve on DBDC. On the whole, these methods can detect clusters with complex shapes in a distributed system, but they are confronted with a variety of densities of HD clusters.

Density-Peak (DP) clustering [30] is a centralized clustering algorithm that has distinctive advantages over many other algorithms. Based on DP clustering, Gong *et al.* [31] proposed an efficient distributed density peak clustering (EDDPC) algorithm. It employs Voronoi diagram and data replication/filtering that significantly reduce the amount of both useless distance measuring and data shuffle in data processing. Zhang *et al.* [32] proposed a distributed DP clustering algorithm, named LSHDDP, based on a locality sensitive hashing (LSH) function. The LSH function has the property: samples that are closer to each other have a higher collision probability. LSHDDP exploits LSH for partitioning data, performs local computation, and aggregates local results to obtain the final results. It has been shown that LSHDDP can achieve a factor of 2x speed-up over the EDDPC approach, while returning comparable clustering results. Overall, these methods can reproduce the performance of the centralized DP clustering, but they usually require transmission of a considerable amount of data samples, thus adverse to network-transmission and privacy-protection.

Kriegel *et al.* [33] proposed a distributed model-based clustering algorithm. Their method uses the expectation maximum (EM) method [34] for detecting local models in terms of mixtures of Gaussian distributions. Then, these local Gaussian distributions are merged in the global site to generate a more meaningful global model. In [35], Lin *et al.* presented a clustering technique with distributed EM mixture modeling. This method controls data sharing, and prevents disclosure of individual data attributes or any results that can be traced back to an individual site. In summary, these methods are effective and efficient for LD data, but usually they have to sacrifice accuracy to compromise the computation cost for HD data.

2.2 High-Dimensional Clustering

With regard to centralized clustering HD data, many methods utilize global dimension-reduction techniques followed by a standard clustering algorithm. The most popular dimension-reduction technique is Principal Component Analysis (PCA) [36]. PCA aims to find a linear mapping from the original space to a LD space such that the mapped data can preserve the variance of the original data. However, the distribution manifold of samples in the original feature space is sometimes non-linear. Applying PCA in this case will lose the non-linear structure information. To overcome this challenge, kernel PCA [37] utilizes a user-specified kernel (similarity matrix) to implicitly construct a non-linear mapping from the original feature space to a new space, and performs PCA in the new space via the kernel trick [38]. The effectiveness of the kernel PCA highly relies on the choice of the kernel, but lacking in feasible methods for the kernel choice. In [39], Tipping and Bishop proposed a probabilistic PCA, which gives PCA a statistical interpretation, provided that an observation sample $\mathbf{x} \in \mathbb{R}^M$ is generated by the sum of three latent terms, i.e., $\mathbf{x} = \mathbf{W}\mathbf{z} + \boldsymbol{\mu} + \epsilon\mathbf{n}$, where $\mathbf{z} \sim G(\mathbf{0}, \mathbf{I}_D)$ and $\mathbf{n} \sim G(\mathbf{0}, \mathbf{I}_M)$ are two latent random variables that follow the standard Gaussian distribution. $\mathbf{W} \in \mathbb{R}^{M\times D}$, $\boldsymbol{\mu} \in \mathbb{R}^M$ and $\epsilon \in \mathbb{R}$ are unknown parameters to be estimated by the probabilistic PCA. Generalized PCA [40] is an algebraic method to handle the case where data distribute in multiple LD subspaces. It has a solid theory foundation, but extra high complexity restricts its application. Robust PCA [41] estimates the principal subspace of data via convex programming and it has shown better performance than PCA while processing data in the presence of outliers.

On the other hand, there are some clustering methods designed to process HD data directly. CLIQUE [42] is a pioneering approach to cluster HD data. It discretizes the feature space into regular intervals in every dimension and thus generates a grid structure. Then dense cells that contain more samples exceeding a threshold are combined with other dense cells in a bottom-up approach [43]. Adjacent dense cells are merged to establish the actual clusters. CLIQUE scales linearly with input size, so it has good scalability as the data dimension in the data is increased. However, obtaining a meaningful clustering is dependent on proper tuning of the grid size and the threshold. This may be difficult in practice because the grid size and threshold are used across all dimension combinations in the dataset [1]. Many extended methods [44], [45], [46] have been proposed to overcome the drawbacks of CLIQUE. By integrating density-based clustering and subspace clustering, Kailing *et al.* [47] proposed a density connected subspace clustering method which can detect arbitrarily shaped clusters in subspaces of the original HD space. Charles *et al.* [48] introduced subspace Gaussian distribution for the research area of clustering and then proposed a HD data clustering method with it. This method follows a mixture-model framework, in which the models are limited to subspace Gaussian distribution. Then an EM-based algorithm is developed to estimate the parameters of the mixture-model. Finally, each sample is assigned to a cluster following the maximum posterior probability. Their experiments show that this method outperforms other HD clustering methods.

In summary, though distributed clustering and HD clustering have been extensively studied separately, limited works focus on clustering HD data with low overhead under distributed environments, which is also a common problem in recent applications. The LDSDC proposed in this work provides a feasible solution to this problem.

For ease of the ensuing presentation, some notations are defined hereafter. Italic symbols (e.g., a and A) represent scalars. Calligraphic symbols denote sets (e.g., $\mathcal{A}$) or subspaces (e.g., $\mathcal{S}$). Boldface capital (e.g., $\mathbf{A}$) and boldface lowercase (e.g., $\mathbf{a}$) symbols denote matrices and vectors, respectively. $\mathbf{I}_M$ and 1_M represent the identity matrix in $\mathbb{R}^{M\times M}$ and the all-one vector in $\mathbb{R}^M$, respectively. $\lambda_i(\mathbf{A})$ denotes the ith largest eigenvalue of a symmetric matrix $\mathbf{A}$. $\oplus_{i=1}^{I} \diamond_i \triangleq \mathrm{diag}(\diamond_1, \ldots, \diamond_I)$ is a block diagonal matrix, where the ith diagonal entry $\diamond_i$ can be a matrix or a scalar. $\|\mathbf{a}\|$ represents the ℓ_2-norm of $\mathbf{a}$. We use $\mathrm{Tr}(\mathbf{A})$, $\mathcal{R}(\mathbf{A})$ and $\|\mathbf{A}\|_F$ to represent respectively the trace, the range space and the Frobenius norm of $\mathbf{A}$. $|\cdot|$ denotes the cardinality of a set or the determinant of a square matrix.

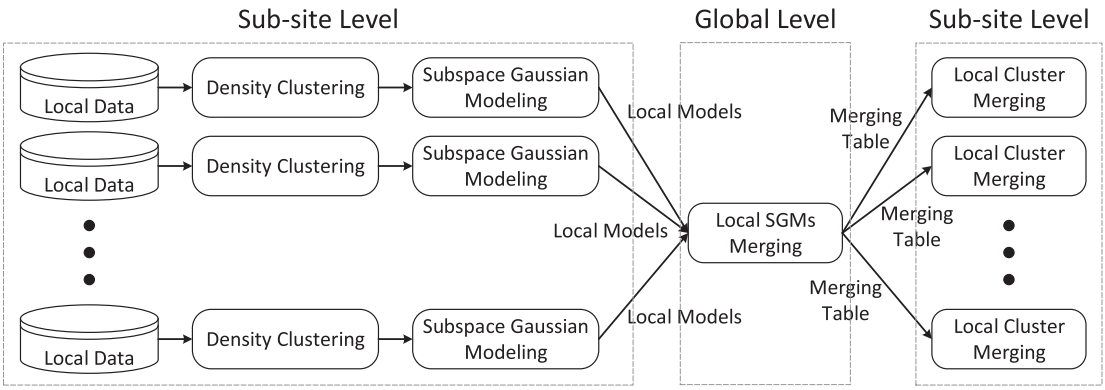


Fig. 1. The structure of the proposed LDSDC algorithm for distributed clustering.

3 Local-Density Subspace Distributed Clustering

In this section, we present the LDSDC algorithm. Consider a scenario that consists of a global site and W sub-sites. The total sample set $\mathcal{X} = \bigcup_{w=1}^{W} \mathcal{X}_w$ is distributed and stored in W sub-sites with $\mathcal{X}_w = \{\mathbf{x}_n^w \in \mathbb{R}^M\}_{n=1}^{N_w}$ stored in the wth sub-site. Our goal is to cluster $\mathcal{X}$ with low overhead in this distributed system. As shown in Fig. 1, our algorithm comprises four stages, as follows:

1) In each sub-site, LDSDC divides samples into local clusters using a density-based method, where each local cluster is termed an atom cluster.
2) In each sub-site, LDSDC models each atom cluster using a SGM. Then, estimated SGM parameters of all models are sent to the global site.
3) In the global site, a global density distribution is generated by integrating all local parameters. Based on the global density distribution, LDSDC calculates the connection values (defined in (6) below) for all the pairs of local models, and then merge similar models according to the connection values. Then the obtained merging result is sent back to each sub-site.
4) In each sub-site, similar atom clusters are merged according to the received merging result.

However, when the given dataset is ill-distributed in different sub-sites (e.g., samples stored in sub-sites are unbalanced, and/or many dense clusters are partitioned in different sub-sites). In this case, prior to applying the 4-stage LDSDC, data reshuffling according to Local Sensitive Hashing values [49] is needed, which proves effective in making data loading balanced and close samples assigned to the same sub-site [16].

Next, the above-mentioned four stages of the proposed LDSDC are presented in the following subsections, respectively.

3.1 Local Clustering (Stage 1)

In general, the overall distribution of a dataset is too complex to model. LDSDC partitions a local dataset into several local clusters such that each local cluster has a relative simple distribution for easing the subsequent modeling procedure. Specifically, LDSDC divides $\mathcal{X}_w$ into local clusters by exploiting a density-based method [50], which has shown superior performance among established single-machine clustering methods. The procedure is detailed next.

First, a K-Nearest-Neighbors (KNNs) kernel is used to estimate the density of each sample. Given a sample $\mathbf{x}$, its KNNs kernel density, denoted by $\rho(\mathbf{x})$, is defined as

$$\rho(\mathbf{x}) = \beta \sum_{\mathbf{y} \in \mathcal{N}_K(\mathbf{x})} \exp\left\{-\frac{\|\mathbf{x}-\mathbf{y}\|}{\sigma}\right\}, \tag{1}$$

where $\mathcal{N}_K(\mathbf{x})$ denotes the KNNs set of $\mathbf{x}$; $\sigma = \frac{1}{|\mathcal{X}_w|}\sum_{\mathbf{x}\in\mathcal{X}_w} \|\mathbf{x} - \mathbf{n}_K(\mathbf{x})\|$ is the mean of the distance between $\mathbf{x}$ and its Kth nearest neighbor $\mathbf{n}_K(\mathbf{x})$; β is a normalization factor. Intuitively, $\rho(\mathbf{x})$ measures how dense the sample distribution is around $\mathbf{x}$.

Second, LDSDC finds the samples with the locally maximal ρ value and marks them as core samples. Formally, the core sample set is defined as

$$\mathcal{O}_w = \left\{\mathbf{x} | \mathbf{x} \in \mathcal{X}_w, \rho(\mathbf{x}) \geq \max_{\mathbf{y}\in\mathcal{N}_K(\mathbf{x})} \rho(\mathbf{y})\right\}. \tag{2}$$

On the other hand, for a non-core sample $\mathbf{x}$, its higher-density nearest neighbor (HDNN) $\boldsymbol{\pi}(\mathbf{x})$ is defined as

$$\boldsymbol{\pi}(\mathbf{x}) = \mathop{\arg\min}_{\mathbf{y}\in\mathcal{N}_K(\mathbf{x}), \rho(\mathbf{y})>\rho(\mathbf{x})} \|\mathbf{x}-\mathbf{y}\|. \tag{3}$$

HDNN allows us to construct a directed graph $\mathcal{G}_w = (\mathcal{X}_w, \mathcal{E}_w)$, where each vertex is a sample and a direct edge exists from a non-core sample to its HDNN, namely, $\mathcal{E}_w = \{(\mathbf{x}, \boldsymbol{\pi}(\mathbf{x})) | \mathbf{x} \in \mathcal{X}_w \backslash \mathcal{O}_w\}$. Fig. 2 shows an illustrative example for $\mathcal{G}_w$, in which

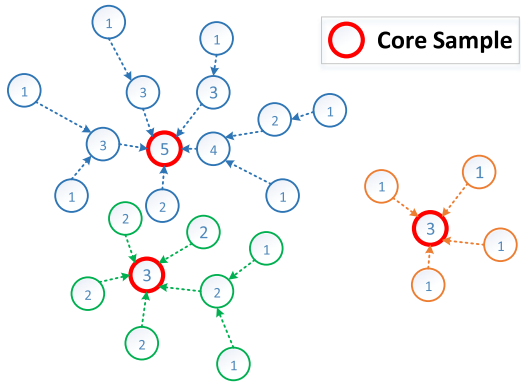


Fig. 2. Illustration for core samples and atom clusters. The atom clusters are represented by different colors and the core samples are marked using red bold circles. Trees rooted at the core samples form an acyclic graph. Each tree corresponds to an atom cluster.

each circle represents a sample with an integer indicating the magnitude of its kernel density for simplicity; a dotted arrow denotes a direct edge $(\mathbf{x}, \boldsymbol{\pi}(\mathbf{x}))$.

As we can see, in $\mathcal{G}_w$, starting from any non-core sample and following the directed edges, it will eventually reach a core sample. In other words, $\mathcal{G}_w$ can be partitioned into trees with disjoint vertices, where each tree is rooted at a core sample. A core sample and its descendants thus form a local cluster, called an *atom cluster*, which is denoted as $\mathcal{C}_s$.

3.2 Modelling Atom Clusters (Stage 2)

Atom clusters form the basis of final clusters, while a true cluster may consist of many atom clusters distributed in different sub-sites. Thus, a globally collecting step is needed to decide which atom clusters should be merged.

An atom cluster can be viewed as a local dense region of the given HD dataset, which is usually embedded in a LD subspace in the real world [14], [15]. In other words, the local samples though may be sparse in the whole HD space, usually are densely distributed in a LD subspace, in which the sample density is relatively stable. On the other hand, since the size of each atom cluster is usually small, to avoid overfitting [51], we do not pursue to exactly characterize the distribution of each atom cluster, but to give a "sketch" to it in an economical and efficient way. In this case, SGM proves competent in modeling HD samples embedded in LD subspace [48]. So we choose it as the sample model. Specifically, an atom cluster, denoted as $\mathcal{C}_s$, is modeled by a SGM characterized by a parameter set Φ_s. Then all the estimated parameter sets will be sent to the global site to further attain a global distribution density. How to efficiently estimate Φ_s for each $\mathcal{C}_s$ will be presented in Section 4.

3.3 Merging Models (Stage 3)

With the collected parameter sets of Φ_s, LDSDC defines a global density distribution by summing all local distributions. Specifically, for a sample $\mathbf{x}$ in the feature space, its global density $\rho_g(\mathbf{x})$ is given by

$$\rho_g(\mathbf{x}) = \sum_{s=1}^{S} \frac{|\mathcal{C}_s|}{|\mathcal{X}|} f(\mathbf{x}|\Phi_s), \tag{4}$$

where S is the total number of local models (atom clusters) and $f(\cdot|\Phi_s)$ denotes the probability density function of the SGM with parameter Φ_s. In practice, calculating $\rho_g(\mathbf{x})$ directly is time-consuming, so we use its log-lower-bound for the subsequent computations instead. Specifically, according to Jensen's inequality [52], the log-lower-bound $\underline{\rho_g}$ is given by

$$\log \rho_g(\mathbf{x}) \geq \sum_{s=1}^{S} \frac{|\mathcal{C}_s|}{|\mathcal{X}|} \log f(\mathbf{x}|\Phi_s) \triangleq \underline{\rho_g}(\mathbf{x}). \tag{5}$$

We empirically found that computing this bound can yield a factor of 1.5-3x speed-up with little accuracy loss.

Based on the log-lower-bound, the *connection value* between the ith and jth model $\alpha_{i,j}$ is defined as

$$\alpha_{i,j} = \min_{\theta \in [0,1]} \left\{ \underline{\rho_g}\big(\theta \boldsymbol{\mu}_i + (1-\theta)\boldsymbol{\mu}_j\big) \right\}, \tag{6}$$

where $\boldsymbol{\mu}_i \in \Phi_i$ and $\boldsymbol{\mu}_j \in \Phi_j$ denote the mean vector of the ith model and the jth model, respectively. Due to the non-convexity of (6), we approximate its solution via a simple but effective method as follows. First, divide the interval [0,1] into L consecutive intervals $[(l-1)/L, l/L], l = 1, \ldots, L$. Second, find a local minimal value for $\hat{\alpha}_{i,j}^l$ for $l = 1, \ldots, L$ by ternary search [53]. Finally, approximate the optimal solution to (6) by $\hat{\alpha}_{i,j} = \min_l \hat{\alpha}_{i,j}^l$. It can be shown that $\hat{\alpha}_{i,j} \to \alpha_{i,j}$ as $L \to \infty$.

Collecting connection values between different models can form a symmetric matrix $\mathbf{A} = \left[\hat{\alpha}_{i,j}\right]$ with size of S^2. The matrix $\mathbf{A}$ is normalized (with all its elements between 0 and 1) through the min-max normalization. Then, the models i and j are merged if $\hat{\alpha}_{i,j} \geq \delta$, where $\delta \in [0,1]$ is a threshold, thus yielding a merging table. Though δ can be set to any value in the range of $[0,1]$, there is only a small collection of δ values (its size is upper bounded by S) that affects the final merging. Furthermore, this collection together with δ can be found efficiently via solving a maximum spanning tree in the graph defined by the matrix $\mathbf{A}$ (by treating $\mathbf{A}$ as an adjacency matrix; see the FJDD algorithm in [50] for more details). In Section 5.5, we will discuss more about the choice of this parameter. Finally, LDSDC broadcasts this table to all the sub-sites.

3.4 Merging Local Clusters (Stage 4)

In this stage, each sub-site independently merges atom clusters according to the received merging table. Then the final clusters at each sub-site are generated accordingly. Let us conclude this section by a computational analysis to the proposed LDSDC algorithm.

Let N_w be the number of samples in the wth sub-site. The computational complexity of *Stage 1* is $O(N_w^2)$ since KNNs need to be computed for each sample. In *Stage 2*, the parameters of SGM can be estimated using Algorithm 1, whose complexity will be discussed in Section 4.2. In *Stage 3*, the cost of estimating (6) for a pair of models is $O(M^2C)$, where M is the data dimension and C is a small number (around 30 by our experience). We found that the connection value between two far apart models is usually negligibly small. Thus, we only consider $\sqrt{S}$ neighbors of each model in computing the connection value, where S is the number of models (atom clusters). As a result, the computational complexity of *Stage 3* is $O(S\sqrt{S}M^2C)$. Finally, *Stage 4* has a computational complexity of $O(N_w)$. The analysis of network transmission overhead is left in Section 4.1.

4 Subspace Gaussian Model

In this section, we present the fundamental SGM used for modeling each atom cluster (cf. Section 3.2). The formulation of SGM has been briefly presented by Bouveyron *et al.* [48] without in-depth discussion about the intuition behind it. In Section 4.1, we establish SGM in an illustrative manner to bridge the gap between the intuition and the formulation. Furthermore, we propose a novel fast parameter-estimation method for SGM, and then present a theoretical analysis of its correctness and convergency in Section 4.2.

4.1 Establishment of Subspace Gaussian Model

Although Gaussian model is a popular model in data representation, real HD samples usually distribute in a LD

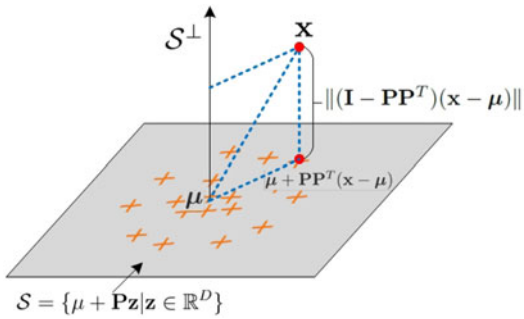


Fig. 3. A graphical interpretation of SGM.

subspace, and so the corresponding sample covariance matrix is nearly singular. This yields serious numerical problems in calculating the probability density of the standard Gaussian model. Furthermore, when the subspace dimension is far smaller than the original space dimension, the amount of parameters of the covariance matrix in the LD space may be significantly smaller than that in the HD space, implying higher overhead in computation and transmission cost for the latter. Some special Gaussian models have been proposed to reduce the number of parameters [33], but their assumption that sample components in different dimensions are statistically independent may not be very realistic.

Given an M-dimensional data space, SGM assumes that a singular Gaussian distribution is embedded in a D-dimensional affine subspace[1] $\mathcal{S}=\{\boldsymbol{\mu}+\mathbf{P}\mathbf{z}|\mathbf{z}\in\mathbb{R}^D\}$, where $\boldsymbol{\mu}\in\mathbb{R}^M$ and $\mathbf{P}\in\mathbb{R}^{M\times D}$ are respectively a shift vector and an orthonormal basis matrix of the D-dimensional subspace. Thus, the singular Gaussian distribution in this subspace is proportional to

$$\exp\Big\{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\mathbf{P}\mathbf{S}^{-1}\mathbf{P}^T(\mathbf{x}-\boldsymbol{\mu})\Big\}, \tag{7}$$

where $\mathbf{S}=\oplus_{d=1}^{D}\sigma_d$ is a diagonal matrix, i.e., the covariance matrix is diagonal under the basis matrix $\mathbf{P}$. For any sample $\mathbf{x}\in\mathbb{R}^M$, its density under SGM is related to two terms. As shown in Fig. 3, one is the density of $\mathrm{Prj}_{\mathcal{S}}\mathbf{x}$, where $\mathrm{Prj}_{\mathcal{S}}\mathbf{x}=\boldsymbol{\mu}+\mathbf{P}\mathbf{P}^T(\mathbf{x}-\boldsymbol{\mu})$ is the projection of $\mathbf{x}$ onto $\mathcal{S}$. The other is the distance from $\mathbf{x}$ to the affine subspace $\mathcal{S}$, which is $\|(\mathbf{I}-\mathbf{P}\mathbf{P}^T)(\mathbf{x}-\boldsymbol{\mu})\|$.

By incorporating the above two terms, the probability density function of SGM can be expressed as

$$\begin{aligned} f(\mathbf{x}|\boldsymbol{\mu},\mathbf{P},\mathbf{S},\sigma_0) =& \frac{1}{(2\pi)^{\frac{M}{2}}\sigma_0^{\frac{M-D}{2}}\sqrt{|\mathbf{S}|}} \\ &\times\exp\Big\{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\mathbf{P}\mathbf{S}^{-1}\mathbf{P}^T(\mathbf{x}-\boldsymbol{\mu})\Big\} \\ &\times\exp\Big\{-\frac{1}{2}\frac{\|(\mathbf{I}-\mathbf{P}\mathbf{P}^T)(\mathbf{x}-\boldsymbol{\mu})\|^2}{\sigma_0}\Big\}. \end{aligned} \tag{8}$$

On the right side of (8), the first term is the normalization factor; the second term models the singular Gaussian distribution in an affine subspace; the third term can be viewed as a decay factor which is exponentially related to the negative distance square from $\mathbf{x}$ to $\mathcal{S}$. From (8), one can see the total amount of the parameters ($\boldsymbol{\mu},\mathbf{P},\mathbf{S}$ and σ_0) is $M+M\times D+D+1\approx O(MD)$.

Compared with the standard Gaussian, SGM allows the covariance to be completely defined in an affine subspace. Furthermore, the total number of parameters of SGM is very flexible, which can easily adapt to the data dimension by choosing a suitable value of D. Note that when $D\ll M$, SGM has much smaller parameter size $O(MD)$ than the standard Gaussian with parameter size $O(M^2)$. Recalling that S is the total number of atom clusters and W is the number of subsites, from Fig. 1, it is evident that the network transmission overhead of our method is $S\times O(MD)+S\times W\approx O(S(MD+W))$. Furthermore, if the reshuffling pre-process is performed, the network transmission overhead will increase by $O(MN)$, where N is the sample set size.

4.2 Fast Parameter-Estimation for SGM

In this subsection, we derive a fast parameter-estimation method for SGM. Assume that $\sigma_1\geq\sigma_2\geq\cdots\geq\sigma_D$ without loss of generality, and that $\sigma_D\geq\sigma_0$ so that the SGM can capture the principal structure of the distribution of data. Thus, given a sample set $\mathcal{X}=\{\mathbf{x}_n\in\mathbb{R}^M\}_{n=1}^N$, the maximum likelihood (ML) estimation of SGM can be formulated as the following optimization problem:

$$\begin{aligned} \max_{\boldsymbol{\mu},\mathbf{P},\mathbf{S},\sigma_0}\quad & \sum_{n=1}^{N}\ln f(\mathbf{x}_n|\boldsymbol{\mu},\mathbf{P},\mathbf{S},\sigma_0) \\ \text{s.t.}\quad & \mathbf{P}^T\mathbf{P}=\mathbf{I}_D, \\ & \mathbf{S}=\bigoplus_{d=1}^{D}\sigma_d, \\ & \sigma_1\geq\sigma_2\geq\cdots\geq\sigma_D\geq\sigma_0>0. \end{aligned} \tag{9}$$

Solving problem (9) directly is difficult. Fortunately, by some transformations, (9) can be reformulated into a tractable form. To begin with, we introduce an auxiliary matrix $\mathbf{Q}\in\mathbb{R}^{M\times(M-D)}$ such that

$$[\mathbf{P},\mathbf{Q}]^T[\mathbf{P},\mathbf{Q}]=\mathbf{I}_M, \tag{10}$$

i.e., the columns of $\mathbf{Q}$ are a set of orthonormal basis vectors of the orthogonal complement of the range space $\mathcal{R}(\mathbf{P})$. Then, by letting

$$\boldsymbol{\Sigma}=[\mathbf{P},\mathbf{Q}](\mathbf{S}\oplus\sigma_0\mathbf{I}_{M-D})[\mathbf{P},\mathbf{Q}]^T, \tag{11}$$

(8) can be reformulated as

$$G(\mathbf{x}|\boldsymbol{\mu},\boldsymbol{\Sigma})=\frac{1}{(2\pi)^{\frac{M}{2}}\sqrt{|\boldsymbol{\Sigma}|}}\exp\Big\{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\Big\}, \tag{12}$$

which is nothing but the density function of a general Gaussian distribution in HD space [51]. Based on (12), the log-likelihood in (9) can be rewritten as

1. In this paper, we indiscriminately use the terms "affine subspace" and "subspace" for simplicity.

$$\begin{aligned}&\sum_{n=1}^{N}\ln G(\mathbf{x}_n|\boldsymbol{\mu},\boldsymbol{\Sigma})\\&=-\frac{1}{2}\sum_{n=1}^{N}\Big(M\ln 2\pi+\ln|\boldsymbol{\Sigma}|+(\mathbf{x}_n-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}_n-\boldsymbol{\mu})\Big)\\&=-\frac{N}{2}\Big(M\ln 2\pi+\ln|\boldsymbol{\Sigma}|+\mathrm{Tr}(\mathbf{C}\boldsymbol{\Sigma}^{-1})\Big),\end{aligned}\tag{13}$$

where

$$\mathbf{C}=\frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}_n-\boldsymbol{\mu})(\mathbf{x}_n-\boldsymbol{\mu})^T.\tag{14}$$

Notice that maximizing (13) is equivalent to minimizing $\ln|\boldsymbol{\Sigma}|+\mathrm{Tr}(\mathbf{C}\boldsymbol{\Sigma}^{-1})$, and thus the ML estimation of SGM is equivalently to solve

$$\begin{aligned}\min_{\boldsymbol{\mu},\mathbf{P},\mathbf{S},\sigma_0}\quad&\ln|\boldsymbol{\Sigma}|+\mathrm{Tr}(\mathbf{C}\boldsymbol{\Sigma}^{-1})\\\text{s.t.}\quad&\boldsymbol{\Sigma}=[\mathbf{P},\mathbf{Q}](\mathbf{S}\oplus\sigma_0\mathbf{I}_{M-D})[\mathbf{P},\mathbf{Q}]^T,\\&[\mathbf{P},\mathbf{Q}]^T[\mathbf{P},\mathbf{Q}]=\mathbf{I}_M,\\&\mathbf{S}=\bigoplus_{d=1}^{D}\sigma_d,\\&\sigma_1\geq\sigma_2\geq\cdots\geq\sigma_D\geq\sigma_0>0.\end{aligned}\tag{15}$$

By setting the derivative of the objective function in (15) with respect to (w.r.t.) $\boldsymbol{\mu}$ to zero, the optimal solution for $\boldsymbol{\mu}$ is given by

$$\boldsymbol{\mu}^{\star}=\frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n.\tag{16}$$

Substituting this into (14), one can see that $\mathbf{C}$ is nothing but the sample covariance matrix. Then, to find optimal $\mathbf{P}$ and $\mathbf{Q}$, we need the following proposition.

Proposition 1. *Suppose that $\mathbf{C}=\mathbf{F}\boldsymbol{\Lambda}\mathbf{F}^T$ is the eigenvalue decomposition of $\mathbf{C}$ defined by (14). If the problem (15) is solvable, then there exists an optimal solution such that $[\mathbf{P}^{\star},\mathbf{Q}^{\star}]=\mathbf{F}$.*

Proof. Because $\mathbf{C}$ is symmetric and $\boldsymbol{\Sigma}$ is positive definite, by using the fact

$$\mathrm{Tr}\Big(\mathbf{C}\boldsymbol{\Sigma}^{-1}\Big)\geq\sum_{m=1}^{M}\lambda_m(\mathbf{C})\cdot\lambda_{M-m+1}\Big(\boldsymbol{\Sigma}^{-1}\Big)=\sum_{m=1}^{M}\frac{\lambda_m(\mathbf{C})}{\lambda_m(\boldsymbol{\Sigma})}.$$

(Theorem 4.3.53 in [54]), we have

$$\begin{aligned}\ln|\boldsymbol{\Sigma}|+\mathrm{Tr}\Big(\mathbf{C}\boldsymbol{\Sigma}^{-1}\Big)&=\sum_{m=1}^{M}\ln\lambda_m(\boldsymbol{\Sigma})+\mathrm{Tr}\Big(\mathbf{C}\boldsymbol{\Sigma}^{-1}\Big)\\&\geq\sum_{m=1}^{M}\ln\lambda_m(\boldsymbol{\Sigma})+\sum_{m=1}^{m}\frac{\lambda_m(\mathbf{C})}{\lambda_m(\boldsymbol{\Sigma})}.\end{aligned}\tag{17}$$

Suppose that $\boldsymbol{\Sigma}^{\star}$ is an optimal solution to problem (9). By using the result (17) and the eigenvalue decomposition of $\mathbf{C}=\mathbf{F}\boldsymbol{\Lambda}\mathbf{F}^T$, we have

$$\begin{aligned}\ln|\boldsymbol{\Sigma}^{\star}|+\mathrm{Tr}\Big(\mathbf{C}(\boldsymbol{\Sigma}^{\star})^{-1}\Big)&\geq\sum_{m=1}^{M}\ln\lambda_m(\boldsymbol{\Sigma}^{\star})+\sum_{m=1}^{M}\frac{\lambda_m(\mathbf{C})}{\lambda_m(\boldsymbol{\Sigma}^{\star})}\\&=\ln\Big|\mathbf{F}\Big(\bigoplus_{m=1}^{M}\lambda_m(\boldsymbol{\Sigma}^{\star})\Big)\mathbf{F}^T\Big|\\&\quad+\mathrm{Tr}\Big(\mathbf{C}\Big[\mathbf{F}\Big(\bigoplus_{m=1}^{M}\lambda_m(\boldsymbol{\Sigma}^{\star})\Big)\mathbf{F}^T\Big]^{-1}\Big).\end{aligned}\tag{18}$$

This implies that $\mathbf{F}\big(\oplus_{m=1}^{M}\lambda_m(\boldsymbol{\Sigma}^{\star})\big)\mathbf{F}^T$ is an optimal solution such that $[\mathbf{P}^{\star},\mathbf{Q}^{\star}]=\mathbf{F}$. □

It can be verified that problem (15) is solvable if $D<\mathrm{rank}(\mathbf{C})$. This condition can be easily satisfied as N is much larger than D in practice. Thanks to Proposition 1, let $[\mathbf{P},\mathbf{Q}]=\mathbf{F}$ and then solving (15) reduces to solving

$$\begin{aligned}\min_{\{\sigma_d\}_{d=0}^{D}}\quad&\sum_{d=1}^{D}\Big(\ln\sigma_d+\frac{\lambda_d(\mathbf{C})}{\sigma_d}\Big)+\sum_{m=D+1}^{M}\Big(\ln\sigma_0+\frac{\lambda_m(\mathbf{C})}{\sigma_0}\Big)\\\text{s.t.}\quad&\sigma_1\geq\sigma_2\geq\cdots\geq\sigma_D\geq\sigma_0>0,\end{aligned}\tag{19}$$

and its optimal solution is given in the following proposition.

Proposition 2. *The optimal solution to (19) is given by*

$$\sigma_d^{\star}=\lambda_d(\mathbf{C})\quad(d=1,2,\ldots,D),\tag{20}$$

and

$$\sigma_0^{\star}=\frac{1}{M-D}\sum_{m=D+1}^{M}\lambda_m(\mathbf{C}).\tag{21}$$

So far we have derived the ML estimators of $\boldsymbol{\mu}$, $\mathbf{P}$, $\mathbf{S}$ and σ_0 (cf. (16), Propositions 1 and 2). However, calculating $\mathbf{P}^{\star}$ and $\mathbf{S}^{\star}$ needs the eigenvalue decomposition of the matrix $\mathbf{C}$, which leads to a computation complexity of $O(M^3)$. In addition, storing $\mathbf{C}$ requires a memory space of $O(M^2)$. Both the computation and the storage overheads are almost intolerable when M is large (e.g., $M>10^5$). To deal with this challenge, we propose a fast parameter-estimation method for the case $M^2\gg D\times N$.

First of all, due to

$$\begin{aligned}\sigma_0^{\star}&=\frac{1}{M-D}\sum_{m=D+1}^{M}\lambda_m(\mathbf{C})\\&=\frac{1}{M-D}\Big(\mathrm{Tr}(\mathbf{C})-\sum_{d=1}^{D}\lambda_d(\mathbf{C})\Big),\end{aligned}\tag{22}$$

only the D largest eigenvalues and the associated eigenvectors of $\mathbf{C}$ need to be computed. On the other hand, $\mathbf{C}$ can be expressed as $\mathbf{C}=\frac{1}{N}\bar{\mathbf{X}}\bar{\mathbf{X}}^T$, where $\bar{\mathbf{X}}=\mathbf{X}-\boldsymbol{\mu}\mathbf{1}_N^T\in\mathbb{R}^{M\times N}$ is the centralized sample matrix. Thus, the partial eigenvalue decomposition of $\mathbf{C}$ is equivalent to solving

$$\begin{aligned}\max_{\mathbf{P}}\quad&\frac{1}{2}\mathrm{Tr}(\mathbf{P}^T\bar{\mathbf{X}}\bar{\mathbf{X}}^T\mathbf{P})\\\text{s.t.}\quad&\mathbf{P}^T\mathbf{P}=\mathbf{I}_D.\end{aligned}\tag{23}$$

Inspired by the re-weighted optimization framework [55], we propose the following iterative two-step method to

solve problem (23)

$$1) \quad \text{Update } \mathbf{G} = \bar{\mathbf{X}}(\bar{\mathbf{X}}^T\hat{\mathbf{P}}); \tag{24a}$$

$$2) \quad \text{Solve } \hat{\mathbf{P}} = \underset{\mathbf{P}^T\mathbf{P}=\mathbf{I}_D}{\arg\max} \operatorname{Tr}(\mathbf{P}^T\mathbf{G}). \tag{24b}$$

Proposition 3. *Let $h(\mathbf{P})$ be the objective function of problem (23) and $\mathbf{P}_t$ denote the solution yielded by the two-step method (24a) and (24b) at the iteration t. Then $h(\mathbf{P}_t)$ increases with t monotonically and the sequence $\{\mathbf{P}_t\}$ converges to a KKT solution [52] to (23).*

Proof. Notice that $h(\mathbf{P})$ is strictly convex when $\mathbf{P} \in \mathcal{R}(\bar{\mathbf{X}})$ and $\nabla_{\mathbf{P}} h(\mathbf{P}) = \bar{\mathbf{X}}\bar{\mathbf{X}}^T\mathbf{P}$. If $\mathbf{P}_{t+1} \neq \mathbf{P}_t$, we have

$$h(\mathbf{P}_{t+1}) > h(\mathbf{P}_t) + \operatorname{Tr}\Big((\mathbf{P}_{t+1} - \mathbf{P}_t)^T \nabla_{\mathbf{P}} h(\mathbf{P}_t)\Big) \geq h(\mathbf{P}_t),$$

where the second inequality holds because

$$\mathbf{P}_{t+1} = \underset{\mathbf{P}^T\mathbf{P}=\mathbf{I}_D}{\arg\max} \operatorname{Tr}\big(\mathbf{P}^T \nabla_{\mathbf{P}} h(\mathbf{P}_t)\big). \tag{25}$$

Moreover, $h(\mathbf{P})$ is bounded above by $\frac{1}{2}\|\bar{\mathbf{X}}\|_F^2$. Therefore, $h(\mathbf{P}_t)$ increases with t monotonically and the sequence $\{\mathbf{P}_t\}$ must converge.

On the other hand, the Lagrangian of the optimization problem in (25) can be written as

$$\mathcal{L}(\mathbf{P}, \mathbf{A}) = \operatorname{Tr}\big(\mathbf{P}^T \nabla_{\mathbf{P}} h(\mathbf{P}_t)\big) + \operatorname{Tr}\big(\mathbf{A}^T\big(\mathbf{P}^T\mathbf{P} - \mathbf{I}_D\big)\big),$$

where $\mathbf{A} \in \mathbb{R}^{D\times D}$ is the Lagrange dual variable. Since $\mathbf{P}_{t+1}$ is the optimal solution to (25), according to the KKT condition, there exists an $\tilde{\mathbf{A}}$ such that

$$\nabla_{\mathbf{P}} \mathcal{L}(\mathbf{P}_{t+1}, \tilde{\mathbf{A}}) = \nabla_{\mathbf{P}} h(\mathbf{P}_t) + \mathbf{P}_{t+1}\big(\tilde{\mathbf{A}} + \tilde{\mathbf{A}}^T\big) = \mathbf{0}. \tag{26}$$

When $\mathbf{P}_{t+1} = \mathbf{P}_t$, (26) becomes

$$\nabla_{\mathbf{P}} h(\mathbf{P}_{t+1}) + \mathbf{P}_{t+1}\big(\tilde{\mathbf{A}} + \tilde{\mathbf{A}}^T\big) = \mathbf{0},$$

which implies $\mathbf{P}_{t+1}$ is also a KKT solution to (23). Thus we have completed the proof. □

The updating $\mathbf{G}$ in (24a) needs a computational complexity of $O(DMN)$. In (24b) for solving $\hat{\mathbf{P}}$, the optimal solution is known as $\hat{\mathbf{P}} = \mathbf{U}\mathbf{V}^T$ [56], where $\mathbf{U}$ and $\mathbf{V}$ are left and right singular vector matrices of $\mathbf{G} = \mathbf{U}\mathbf{D}\mathbf{V}^T$ (singular value decomposition), respectively. Thus, (24b) requires a computational complexity of $O(D^2M)$. Finally, we come up with the fast ML estimation method in Algorithm 1, with the computational complexity of $O(DMN + D^2M)$. Furthermore, it only needs a memory space of $O(M(D+N))$.

Algorithm 1. Fast ML Estimation for SGM

Input: Sample matrix $\mathbf{X} \in \mathbb{R}^{M\times N}$, Subspace dimension $D \in \mathbb{R}$
Output: $\hat{\boldsymbol{\mu}} \in \mathbb{R}^M, \hat{\mathbf{P}} \in \mathbb{R}^{M\times D}, \hat{\mathbf{S}} \in \mathbb{R}^{D\times D}, \hat{\sigma}_0$

$\hat{\boldsymbol{\mu}} = \frac{1}{N}\mathbf{X}\mathbf{1}_N$;
$\bar{\mathbf{X}} = \mathbf{X} - \hat{\boldsymbol{\mu}}\mathbf{1}_N^T$;
Initialize $\hat{\mathbf{P}}$ as $[\mathbf{I}_D, \mathbf{0}]^T \in \mathbb{R}^{M\times D}$;
repeat
  Update $\mathbf{G} = \bar{\mathbf{X}}(\bar{\mathbf{X}}^T\hat{\mathbf{P}})$;
  Decompose $\mathbf{G}$ into $\mathbf{U}\mathbf{D}\mathbf{V}^T$;
  Update $\hat{\mathbf{P}} = \mathbf{U}\mathbf{V}^T$;
until *Convergence*;
$\hat{\mathbf{S}} = \frac{1}{N}\mathbf{D}$;
$\hat{\sigma}_0 = \frac{1}{M-D}\Big(\frac{1}{N}\sum_{m=1}^{M} \bar{\mathbf{x}}_m^T\bar{\mathbf{x}}_m - \operatorname{Tr}(\hat{\mathbf{S}})\Big)$, where $\bar{\mathbf{x}}_m^T$ is the mth row of $\bar{\mathbf{X}}$;

5 Experiments

We evaluate the performance of the proposed LDSDC algorithm by experiment. All experiments are implemented on a server cluster comprising four machines. Each machine is equipped with an Intel E7-4809 v2 1.90G 48-core CPU, 60 GB memory, running Spark 2.1.0.

5.1 Baseline Methods and Parameter Setting

We select four state-of-the-art distributed clustering algorithms as baseline methods for performance comparison, including k-means‖ [57], DBDC [4], LSHDDP [32], REMOLD [16]. Note that REMOLD is a preliminary version of our work. Among the baseline methods, DBDC and LSHDDP share a common density meansure, which is described as follows.

- ϵ cut-off density. It is defined as the number of samples falling inside a ball of radius ϵ. The parameter ϵ is estimated as the 2 percent position of the ascending ordered distances among all sample pairs according to the recommendation in [32].

Table 1 lists detailed parameter settings for all the methods under test. The best performance is obtained for each method by searching throughout the parameter space stated in this table. To make the results stable and repeatable, the reshuffling pre-process is performed for all methods.

TABLE 1
Simulation Settings of All the Methods Under Test

Method	Setting
k-means‖	The parameter k is set to the real class number. We run it five times for each dataset with different random seeds.
DBDC	Use ϵ cut-off density. The parameter $MinPts$ is determined by $\lambda \cdot \bar{\rho}$, where λ is selected from $\{0.01, 0.05, 0.1, 0.5, 1\}$ and $\bar{\rho}$ is the mean density. Each outlier is assigned to its nearest cluster.
LSHDDP	Use ϵ cut-off density. The samples with top k gamma values (refer to [30]) are chosen as cluster centers, where k is the real class number. We set parameters $M = 10$ and $\pi = 3$ according to the recommendation in [32].
REMOLD	The parameter K is set to $\sqrt{N/W}$, where N and W are the numbers of samples and sub-sites, respectively. The parameter δ is enumerated in a candidate collection Δ, which is extracted by an auxiliary algorithm [16].
LDSDC (Proposed)	The parameter D is set to the smallest integer such that the retained eigenvalue ratio (i.e., the ratio of the sum of retained eigenvalues to the sum of total eigenvalues) of the sample covariance matrix (cf. (14)) is greater than 70%. All the other parameters are determined by the same way as REMOLD.

TABLE 2
Ten Synthetic Gaussian Datasets

Dataset	#Samples	#Dimensions	#Clusters	Number of Clusters with Embedded Dimension										#Partitions (sub-sites)
				2-D	4-D	8-D	16-D	32-D	64-D	128-D	256-D	512-D	1024-D	
G-1	50,000	2	20	20										4
G-2	100,000	4	40	20	20									4
G-3	150,000	8	60	20	20	20								4
G-4	200,000	16	80	20	20	20	20							4
G-5	250,000	32	100	20	20	20	20	20						8
G-6	300,000	64	120	20	20	20	20	20	20					8
G-7	350,000	128	140	20	20	20	20	20	20	20				16
G-8	400,000	256	160	20	20	20	20	20	20	20	20			16
G-9	450,000	512	180	20	20	20	20	20	20	20	20	20		32
G-10	500,000	1,024	200	20	20	20	20	20	20	20	20	20	20	32

The first column is dataset code, where G-l implies its dimension is 2^l. Columns from two to four are the number of samples, dimensions and clusters in each dataset, respectively. Columns from five to fourteen present the distribution of clusters with different embedding dimensions, where D-D is the abbreviation of D-dimension. The last column gives the number of partitions (i.e., sub-sites) for each dataset.

5.2 Datasets

We evaluate our method on three series of datasets, i.e., synthetic Gaussian datasets, synthetic norm-ball surface datasets and real-world datasets, which are presented in the following subsections, respectively.

5.2.1 Synthetic Gaussian Datasets

First, we test all the five algorithms on the most common Gaussian datasets. Each dataset consists of Gaussian clusters with various embedding dimensions. Specifically, an M-dimensional cluster with D embedding dimension is generated through the following procedure.

1) Generate a Gaussian random (element-wise) matrix $\mathbf{T} \in \mathbb{R}^{M\times D}$ and normalize it by $\frac{\mathbf{T}}{\|\mathbf{T}\|_F}$.
2) Generate a shift vector $\boldsymbol{\mu} \in \mathbb{R}^M$ whose elements are independently sampled from a uniform distribution over the interval $[-\frac{20}{\log M}, \frac{20}{\log M}]$.
3) A sample is generated as

$$\mathbf{x} = \mathbf{T}\mathbf{z} + \boldsymbol{\mu} + \boldsymbol{\epsilon}, \tag{27}$$

where $\mathbf{z} \sim G(\mathbf{0}, \mathbf{I}_D)$ and $\boldsymbol{\epsilon} \sim G(\mathbf{0}, \frac{1}{10M}\mathbf{I}_M)$.
4) Repeat 3) 2,500 times, and these samples constitute a cluster (i.e., each cluster consists of 2,500 samples).

TABLE 3
Ten Synthetic Norm-Ball Surface Datasets

Dataset	#Samples	#Dimensions	#Clusters	Surface Set	#Partitions
S-1	1500	2	3	$\mathbb{S}(2)$	2
S-2	3,000	4	3	$\mathbb{S}(3)$	2
S-3	6,000	8	3	$\mathbb{S}(4)$	2
S-4	12,000	16	3	$\mathbb{S}(5)$	4
S-5	24,000	32	3	$\mathbb{S}(6)$	4
S-6	48,000	64	3	$\mathbb{S}(7)$	4
S-7	96,000	128	3	$\mathbb{S}(8)$	8
S-8	182,000	256	3	$\mathbb{S}(9)$	8
S-9	364,000	512	3	$\mathbb{S}(10)$	16
S-10	728,000	1,024	3	$\mathbb{S}(11)$	32

The first column is dataset code, where S-l implies its dimension is 2^l. Columns from two to four are the numbers of samples, dimensions and clusters in each dataset, respectively. The fifth column shows the corresponding norm-ball surface set $\mathbb{S}(D) = \{0.5\mathcal{E}_{0.5}^{D,2^{D-1}}, \mathcal{E}_1^{D,2^{D-1}}, 1.5\mathcal{E}_2^{D,2^{D-1}}\}$. The last column gives the number of partitions (i.e., sub-sites) for each dataset.

Ten datasets with dimension from 2 to 1,024 are generated and detailed information about these datasets are listed in Table 2.

5.2.2 Synthetic Norm-Ball Surface Datasets

Unlike Gaussian datasets, in which clusters are shaped like ellipsoids and not very difficult in clustering, synthetic norm-ball surface datasets include clusters with non-convex shapes, which is more challenging to most methods. Specifically, an ℓ_p-norm ball surface in D-dimensional space refers to the set $\mathcal{E}_p^D = \{\mathbf{z}|\mathbf{z} \in \mathbb{R}^D, \|\mathbf{z}\|_p = 1\}$. Furthermore, by left multiplying a semi-orthonormal matrix $\mathbf{P} \in \mathbb{R}^{M\times D}(D < M)$, $\mathcal{E}_p^D$ can be embedded in an M-dimensional space. We denote the embedded norm-ball surface as $\mathcal{E}_p^{D,M} = \{\mathbf{P}\mathbf{z}|\mathbf{z} \in \mathbb{R}^D, \|\mathbf{z}\|_p = 1\}$. Thus norm-ball surface datasets are generated by sampling from three different norm-ball surfaces $0.5\mathcal{E}_{0.5}^{D,M}$, $\mathcal{E}_1^{D,M}$ and $1.5\mathcal{E}_2^{D,M}$, where D and M are chosen to produce data with different dimensions. We generate ten datasets whose detailed information is given in Table 3, and Fig. 4 gives two illustrative examples.

5.2.3 Real-World Datasets

We select five HD real-world datasets for our experiments. Among them, Salinas, PaviaC and PaviaU are three hyperspectral images (HSIs) [58]. Salinas was collected by the AVIRIS sensor over agricultural crops in Salinas Valley, California,

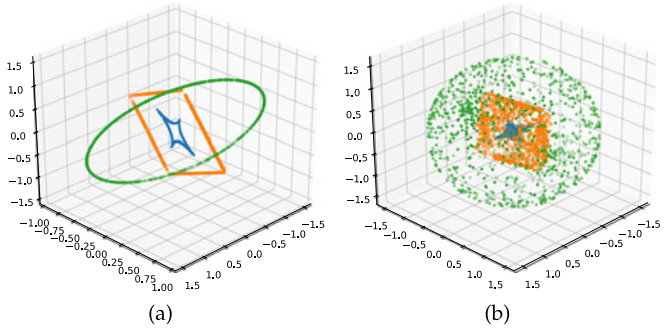


Fig. 4. Two illustrative examples for three norm-ball surface datasets, including (a) an example sampled from $0.5\mathcal{E}_{0.5}^{2,3}$ (blue), $\mathcal{E}_1^{2,3}$ (orange) and $1.5\mathcal{E}_2^{2,3}$ (green), and (b) an example sampled from $0.5\mathcal{E}_{0.5}^{3,3}$ (blue), $\mathcal{E}_1^{3,3}$ (orange) and $1.5\mathcal{E}_2^{3,3}$ (green).

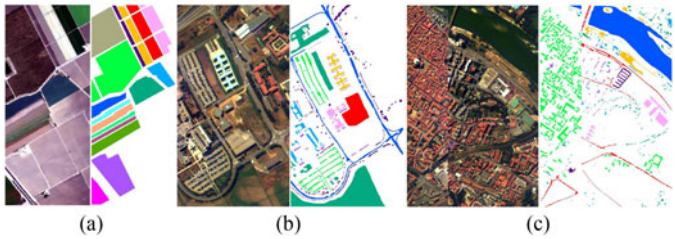


Fig. 5. Hyperspectral datasets of color composite images together with the ground truth maps for (a) Salinas, (b) PaviaC, and (c) PaviaU, respectively.

and is characterized by high spatial resolution (3.7-meter pixels). Its size is 512 lines by 217 samples, in addition to 204 spectral bands used. It has 16 classes including vegetables, bare soils, and vineyard fields. PaviaU and PaviaC are two scenes acquired by the ROSIS sensor during a flight over Pavia, nothern Italy. The spatial size is 610×340 for PaviaU and 1096×715 for PaviaC. PaviaU and PaviaC were acquired using 103 spectral bands and 102 spectral bands, respectively. Both of them contain 9 classes. Fig. 5 shows the color composite images and the ground truth maps for the three datasets. We also consider another real-world dataset MNIST [59], containing 60,000 examples of handwritten digits from 0 to 9, and each example has a dimension of 784 (28×28). Fig. 6 shows some samples of MNIST. MNIST1M [60] is an extended version of MNIST with 1,000,000 samples. Table 4 lists the detailed information about these datasets.

5.3 Performance Metric

We evaluate the clustering quality of all the algorithms under test by two measures: normalized mutual information (NMI) and purity. The larger the two measures, the better the clustering quality.

NMI [61] is a clustering metric from the information-theory, and it quantifies the "amount of information" as follows. Given a dataset V of size N, suppose there are I clusters and J true classes. Let N_i, $N^{(j)}$ and $N_i^{(j)}$ denote the number of samples in cluster i, true class j, and both cluster i and true class j, respectively. Then, NMI is computed by

$$\mathrm{NMI} = \frac{\sum_{i=1}^{I}\sum_{j=1}^{J}\frac{N_i^{(j)}}{N}\log\frac{NN_i^{(j)}}{N_i N^{(j)}}}{\sum_{i=1}^{I}\frac{N_i}{N}\log\frac{N_i}{N}\sum_{j=1}^{J}\frac{N^{(j)}}{N}\log\frac{N^{(j)}}{N}}.$$

Purity [62] counts the number of correctly assigned samples divided by the size of the dataset. Given a dataset with size N, suppose that $\{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_I\}$ and $\{\mathcal{W}_1, \mathcal{W}_2, \ldots, \mathcal{W}_J\}$ are the set of clusters and true classes, respectively. The purity is defined by

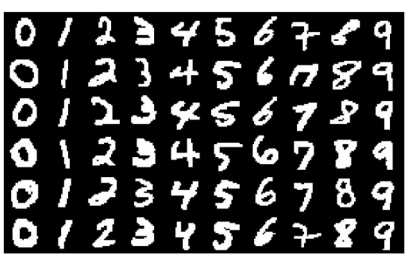


Fig. 6. Some typical samples of the MNIST dataset.

TABLE 4
Detailed Information for Five Real-World Datasets

Dataset	#Samples	#Dimensions	#Clusters	#Partitions
Salinas	111,104	206	16	8
PaviaU	207,400	105	9	8
PaviaC	783,640	104	9	16
MNIST	60,000	784	10	4
MNIST1M	1,000,000	784	10	32

Columns from two to four are the number of samples, dimensions and clusters in each dataset, respectively. The last column gives the number of partitions (i.e., sub-sites) for each dataset.

$$\mathrm{purity} = \frac{1}{N}\sum_{i=1}^{I}\max_{1\le j\le J}|\mathcal{C}_i \cap \mathcal{W}_j|.$$

Notice that *purity* cannot be used for trade-off between the quality of the clustering versus the number of clusters (e.g., dividing a dataset into singletons can get the perfect score), so we report the purity only for the clustering result with the highest value of NMI.

In addition, we also report the amount of transmission cost and the running time (machine time spent on the whole clustering process) for each method in the conducted experiments.

5.4 Experimental Results

5.4.1 Results on Synthetic Gaussian Datasets

The clustering results for the Gaussian datasets are shown in Fig. 7.

From Fig. 7a, one can observe that, in terms of NMI measure, LSHDDP (denoted as "+") performs best; both LDSDC (denoted as "△") and REMOLD (denoted as "✱") also perform well with comparable performance; k-means||

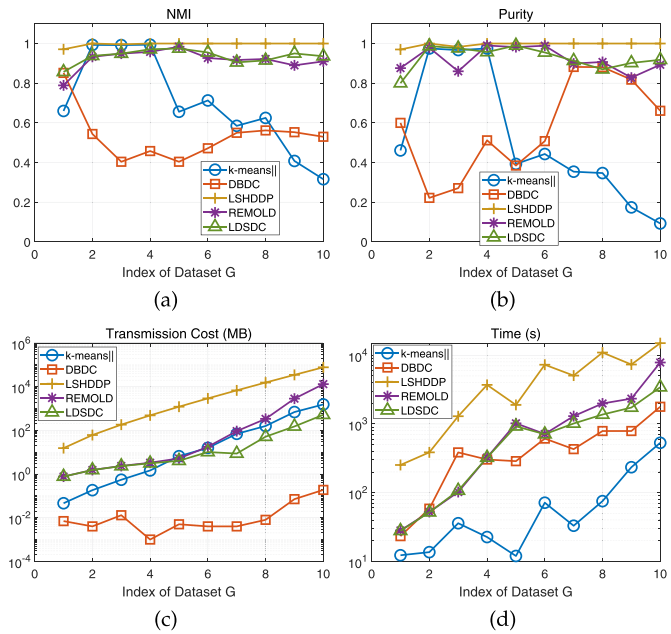


Fig. 7. Performance comparison of the five methods under test on the Gaussian datasets. (a) NMI, (b) purity, (c) transmission cost (mega bytes in semi-log scale) (d) running time (seconds in semi-log scale), w.r.t. the index of the datasets, respectively.

(denoted as "○") achieves almost perfect performance for the LD datasets G-2, G-3 and G-4, but its performance seriously degrades as data dimension increases possibly due to many local minima in HD datasets; and DBDC (denoted as "□") performs worst except for the LD dataset G-1 because the model used is too simplified to accurately model the complex distribution in HD datasets. In terms of the purity measure, the corresponding performances of all the algorithms are shown in Fig. 7b. The performance behaviors of k-means||, LSHDDP, REMOLD and LDSDC observed from Fig. 7a also apply to Fig. 7b. DBDC shows different low-quality performance behaviors in Figs. 7a and 7b, although its performance is comparable to LDSDC and REMOLD for the datasets of G-7, G-8 and G-9 in Fig. 7b, indicating that it may over-divide the datasets into finer clusters.

The corresponding transmission cost (mega bytes) is presented in Fig. 7c. A common performance behavior for all the algorithms except DBDC can be seen from this figure, showing that the transmission cost linearly increases with the data dimension in the semi-log scale. Furthermore, LSHDDP has the heaviest transmission cost due to reshuffling the dataset several times; k-means|| has lower transmission costs (proportional to the product of the size and the dimension of the dataset) for LD datasets G-1 through G-4 than LDSDC and REMOLD, while LDSDC has lower transmission costs for HD datasets G-6 through G-10 than k-means|| and REMOLD. It is noticeable that their transmission costs are almost the same for the dataset G-5 (median dimension), and specifically, REMOLD needs to transmit 13 gigabytes but the cost of LDSDC is just 0.5 gigabytes for the HD dataset G-10. DBDC has the lowest transmission cost since the amount of the parameters for transmission in its own operation rule is not sensitive to the volume of datasets.

The running time (seconds) comparison for all the methods under test is shown in Fig. 7d. Again, a common performance behavior for all the algorithms can be seen from this figure, showing that the trend of running time linearly increases with the data dimension in semi-log scale. The trends associated with LSHDDP, LDSDC and REMOLD have similar growth rate, but LSHDDP (with highest NMI score) has much higher running time than LDSDC and REMOLD due to extraordinary repeated distance calculations of numerous sample pairs. Instead, LDSDC (with smaller growth rate than REMOLD for G-7 through G-10) achieves NMI and purity performances comparable to LSHDDP at the expense of more economical computation cost. For instance, the former can handle the dataset G-10 within an hour, but the latter needs over four hours. The running time for DBDC is basically smaller than that for both LDSDC and REMOLD together with smaller growth rate. Finally, the running time for k-means|| is the lowest, showing its high efficiency in handling large but LD datasets.

5.4.2 Results on Synthetic Norm-Ball Surface Datasets

The clustering results for the norm-ball surface datasets are shown in Fig. 8.

Fig. 8a exhibits the NMI scores for all the methods under test. Some observations from this figure are as follows. The proposed LDSDC (△) significantly outperforms all the other

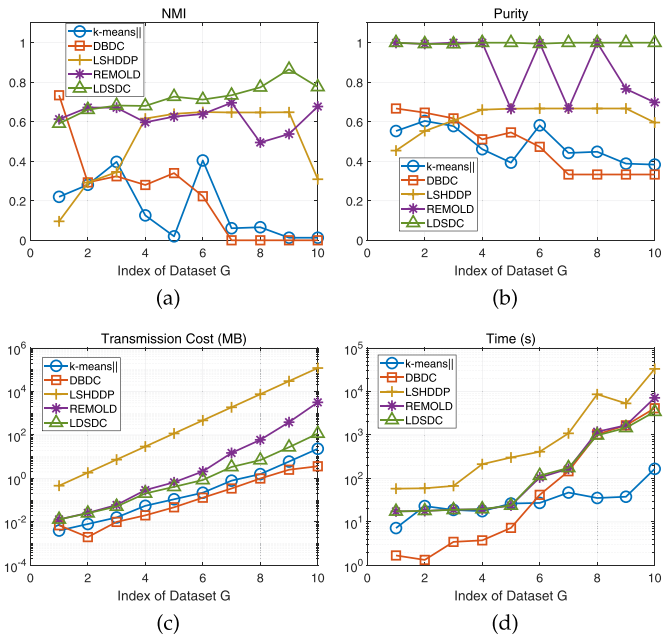


Fig. 8. Performance comparison of the five methods under test on the norm-ball surface datasets. (a) NMI, (b) purity, (c) transmission cost (mega bytes in semi-log scale), (d) running time (seconds in semi-log scale), w.r.t. the index of the datasets, respectively.

methods with NMI scores between 0.6 and 0.85, and its performance is insensitive to both of the data dimension and cluster structure. Overall, REMOLD (✱) performs second and LSHDDP (+) performs third. However, k-means|| (○, designed for classifying convex clusters) and DBDC (□, with insufficient core samples in modeling the HD distribution) basically fail for all the datasets despite the highest NMI score of the latter for the LD dataset S-1. In terms of purity, as shown in Fig. 8b, the performance ranking and behaviors of all the algorithms are basically consistent with those associated with Fig. 8a, namely, the proposed LDSDC performs best with almost perfect purity scores (near 1), REMOLD performs second with purity scores between 0.65 and 1, and LSHDDP performs third with purity scores below 0.65, and k-means|| and DBDC share the bottom performance. These experimental results demonstrate that the proposed LDSDC algorithm is much robust to complex non-convex clusters and data dimension than all the other algorithms.

The corresponding transmission cost (mega bytes in semi-log scale) is presented in Fig. 8c. The linear-increase behavior of transmission cost in semi-log scale observed from Fig. 7c can also be observed from Fig. 8c for all the algorithms under test. Though their growth rates are also similar, their transmission cost ranks are LSHDDP (rank 1, highest), REMOLD (rank 2), LDSDC (rank 3), k-means|| (rank 4) and DBDC (rank 5, lowest). It is noticeable that REMOLD and LDSDC seem to share similar network costs for the first four datasets, but the former expends the cost for the HD dataset S-10 about 30 times higher than the latter due to larger growth rate in semi-log scale. As for the running time comparison (seconds in semi-log scale) shown in Fig. 8d, basically, the linear-increase trends (in semi-log scale) versus the data dimension associated with Fig. 7d also apply to Fig. 8d for all the algorithms under test, showing that their running time ranks are

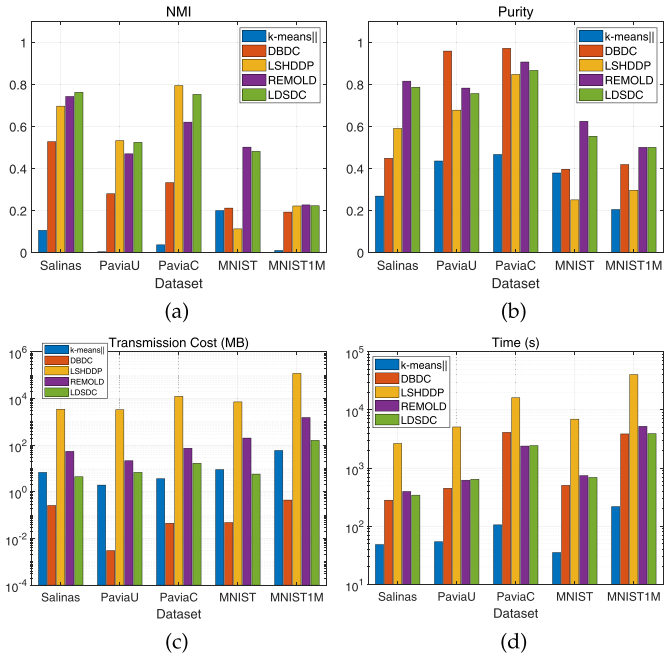


Fig. 9. Performance comparison of the five methods under test on the five real-world datasets by bar graph, including (a) NMI, (b) purity, (c) transmission cost (mega bytes in semi-log scale) and (d) running time (seconds in semi-log scale).

LSHDDP (rank 1, highest), REMOLD (rank 2), LDSDC (rank 3), k-means|| (rank 5 over HD datasets) and DBDC (rank 5 over LD datasets). Finally, let us mention that LDSDC and REMOLD expend almost the same running time over all the datasets except for the HD dataset S10, and that k-means|| has a stable low running-time cost for all the datasets (owing to its simple computation at each iteration), while DBDC has an amazingly low running-time cost for the first four datasets but its running-time cost rapidly increases for the other HD datasets. Overall, the above experimental results on various synthetic datasets have demonstrated that the proposed LDSDC outperforms the other four sate-of-the-art algorithms.

5.4.3 Results on Real-World Datasets

Fig. 9 shows the performance comparison of the five algorithms using five real-world datasets. One can see from Fig. 9a that in terms of NMI, LDSDC (green bar) gets the hightest score only for Sainas, and the second highest score for the other four datasets. However, its NMI performance is much worse for the two datasets MNIST and MNST1M possibly due to the binary nature of the data (either around 0 for black pixels or around 255 for white pixels), thereby causing lower modeling accuracy since the SGM itself is a continuous distribution model. However, we would like to emphasize that the performance of LDSDC is either the best, or the second best but very comparable to the best for each dataset, thereby exhibiting its performance consistence (and robustness) over the five datasets. Among the five methods, only REMOLD (denoted by purple bar) achieves competitive performance compared to LDSDC, while the other algorithms cannot maintain consistent good performance over the five datasets.

Fig. 9b shows the corresponding performance comparison results in terms of the purity score. From this figure, one can observe that the performance of REMOLD is either the best, or the second best but comparable to the best for each dataset, thereby exhibiting its performance consistence (and robustness) over the five datasets. Nevertheless, the performance of the proposed LDSDC is very comparable to that of REMOLD over all the datasets. Hence, these results also support LDSDC's good performance quality and consistence over the five datasets.

The corresponding transmission cost (mega bytes) and running time (seconds) are shown in Figs. 9c and 9d (semi-log plots), respectively. As mentioned above, the performance qualities (NMI and purity) of k-means|| (denoted by blue bar), DBDC (denoted by red bar) and LSHDDP (denoted by dark yellow bar) are neither satisfactory nor consistent over the five datasets. Instead, let us only focus on transmission cost and running time associated with REMOLD and LDSDC. One can observe, from Figs. 9c and 9d, that their running times are comparable to each other, while the transmission cost associated with latter is much lower than the former (in semi-log scale). Therefore, these experimental results on the five real-world datasets well support the overall superior efficacy (performance quality, computation cost and network overhead) of the proposed LDSDC algorithm over the other four state-of-the-art algorithms.

5.5 Parameter Sensitivity Analysis

The clustering performance of the proposed LDSDC depends on the number of sub-sites (NoS) and three user-specified parameters, i.e., K (number of K NNs), D (dimension of subspace in SGM) and δ (threshold for model merging). We investigate the performance sensitivity of these parameters with seven representative datasets (G-6, G-7, S-7, S-8, Salinas, PaviaU and MNIST) in the following subsections, respectively.

5.5.1 Sensitivity to Number of Sub-Sites

In this experiment, LDSDC is tested under the default parameter settings given in Table 1, and the experimental results are obtained for NoS between 2 and 18 such that the running time is within a reasonable range for each dataset for ease of illustration. From Fig. 10, one can observe that the NMI scores decrease slowly with NoS in trend, but not very sensitive to NoS in spite of some fluctuations for S-7 and S-8 datasets. As for the metric of purity, similar performance behavior can be observed in Fig. 10b. On the other hand, as shown in Fig. 10c, the transmission costs increase linearly in trend, though the linear trends for G-7 and MNIST are much higher than for the other datasets due to their diverse cluster distributions and high-dimensionality. Finally, as shown in Fig. 10d, the running times decrease with NoS exponentially instead.

In conclusion, the increase of the NoS will lead to slight degradation of the clustering performance and moderate or mild growth in the transmission cost on one hand, exponentially higher efficiency on the other hand.

5.5.2 Sensitivity to K and D

Let $\mathrm{RER} \triangleq \frac{\sum_{i=1}^{D} \lambda_i(\mathbf{C})}{\mathrm{Tr}(\mathbf{C})}$ be the retained eigenvalue ratio of the data covariance matrix $\mathbf{C}$ (cf. (14)) in the D-dimensional

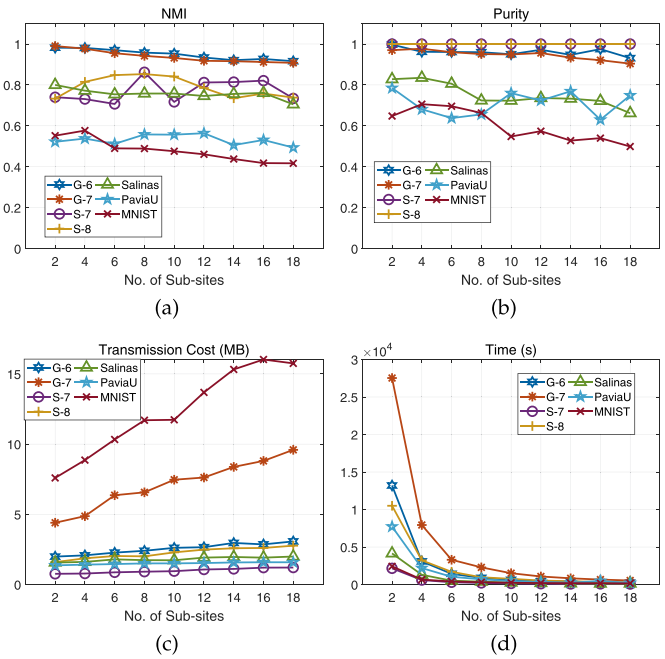


Fig. 10. The clustering performance of LDSDC as a function of the number of sub-sites, including (a) NMI, (b) purity, (c) transmission cost (mega bytes) and (d) running time (seconds).

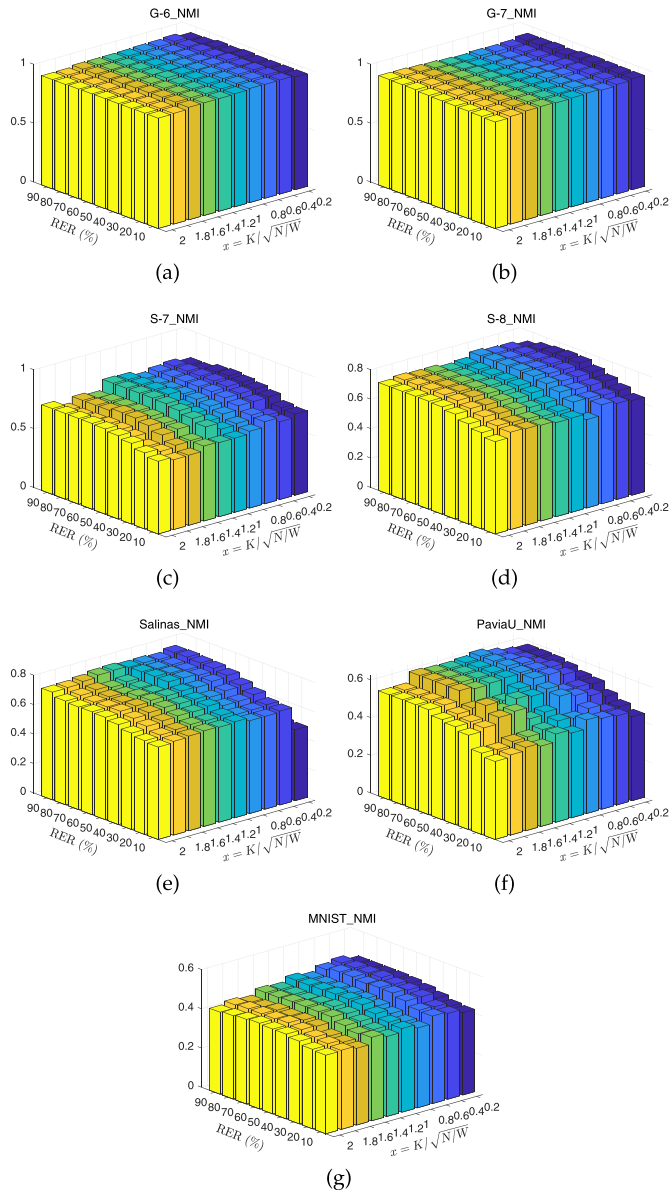


Fig. 11. The clustering performance (measured by NMI) of LDSDC as a function of $x = K/\sqrt{N/W}$ and RER % (which is larger for larger D) for datasets (a) G-6, (b) G-7, (c) S-7, (d) S-8, (e) PaviaU, (f) Salinas and (g) MNIST, respectively.

subspace of SGM. Obviously, the larger the RER, the larger the parameter D. The experimental results for NMI versus $x = K/\sqrt{N/W}$ and RER (instead of D) are displayed in Fig. 11 where $0.2 \leq x \leq 2$ and $10\% \leq \text{RER} \leq 90\%$ for each dataset. Overall, the variation of NMI is mild, indicating LDSDC's low performance sensitivity to (K, RER); the smaller the value of K or the larger the value of RER, the better the NMI performance. Moreover, as shown in Fig. 11a through Fig. 11d, the NMI performance is better for G-6 and G-7 (synthetic Gaussian datasets) than for S-7 and S-8 (synthetic norm-ball surface datasets) besides larger fluctuations for the latter; as shown in Fig. 11e through Fig. 11g, it is better for Salinas and PaviaU datasets than for MNIST dataset in spite of slightly larger fluctuations for PaviaU.

In summary, the performance of LDSDC is not very sensitive to K and RER, for which a good choice is $K \approx \sqrt{N/W}$ and $\text{RER} \geq 70\%$.

5.5.3 *Sensitivity to* δ

In this subsection, for the parameter $\delta \in [0, 1]$, we present the qualitative analysis, followed by the quantitative analysis. First of all, the larger δ, the larger the number of clusters, implying that a positive correlation between them. As mentioned in Section 3.3, only a small collection of finite δ values, denoted as the set Δ here, can affect the final clustering. Moreover, the mapping from $\delta \in \Delta$ to the number of clusters is bijective (one-to-one and onto). Hence, LDSDC can uniquely yield the clusters with the desired clustering number. However, if the prior information about the clustering number is not available, LDSDC can produce a series of hierarchical results by enumerating each $\delta \in \Delta$, thereby providing a comprehensive perspective to the underlying structure in the dataset.

To further analyze how δ affects the clustering performance, the NMI performances of the LDSDC are displayed in Fig. 12 for all the datasets in our experiments as δ varies. One can see, from the left plot (Gaussian datasets) in Fig. 12, that the NMI starting from $\delta = 0$ increases as δ increases either mildly or rapidly like mountain climbing, and finally reaches the top of the mountain (a plateau or roughly a plateau) for $\delta \geq 0.8$. From the mid plot (norm-ball surface datasets) in Fig. 12, we can see that each NMI increases with δ in a very nonlinear manner due to complex distributions of the datasets, and the maximum NMI appears for $\delta \geq 0.8$ except the dataset S-9 for which NMI has been on a plateau for $\delta \in [0.6, 0.77]$. As for the right plot (real-world datasets) in Fig. 12, each NMI increases with δ in a less tractable (more mild) manner than that associated with Gaussian datasets (norm-ball surface datasets), and it reaches either a plateau or the maximum over [0.8,1].

In conclusion, in practical applications, we suggest that users can choose a suitable δ from the intersection

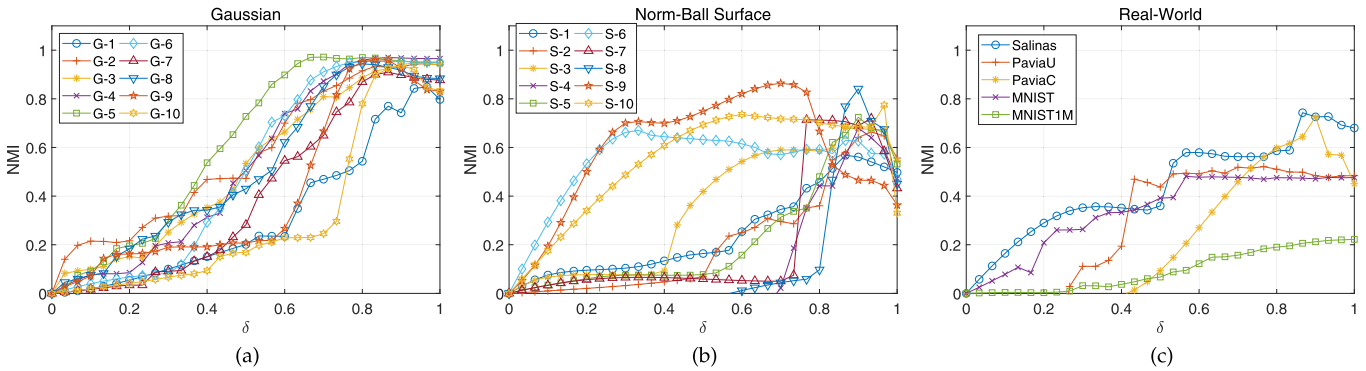


Fig. 12. The clustering performance (measured by NMI) of LDSDC as a function of δ for (a) Gaussian datasets, (b) norm-ball surface datasets, and (c) real-world datasets.

of [0.8,1] and Δ through the elbow method based on intrinsic metrics [1].

6 Conclusion

Based on the fact that local samples of HD data usually reside in LD data spaces, we have presented a distributed clustering algorithm (LDSDC algorithm) as shown in Fig. 1, especially for HD datasets with complex cluster structures. The proposed LDSDC algorithm uses the SGM for modeling every local cluster (atom cluster) in the LD space (conducted in every sub-site), and then get upgraded SGMs for all the collected local atom clusters in the global site together with a cluster merging table (determined by the threshold parameter δ for model merging) for each sub-site to yield the final clustering result. Some experimental results using synthetic datasets and real-world datasets were presented to demonstrate the effectiveness of the proposed LDSDC algorithm and its overall superior performance (clustering quality, network overhead, and computation cost) over four benchmark algorithms (k-means||, DBDC, LSHDDP and REMOLD), in spite of some performance degradation when the given datasets are basically constituted by finite discrete feature values (e.g., datasets MNIST and MNIST1M). Finally an analysis for the proposed LDSDC about its performance sensitivity to the system parameters was presented.

Some further studies are left in the future, including extension of the proposed LDSDC from facilitator sub-site networks to P2P networks, design of an automatic strategy for the parameter δ, and development of a privacy-preserving LDSDC due to increasing concerns of personal privacy.

Acknowledgments

This work was supported by the National Key Research and Development Program of China (No. 2017YFC1501503), Beijing Natural Science Foundation (No. L191016), National Social Science Foundation of China (No. 18CSH019), Beijing Municipal Education Commission Research Program (No. SM20191001107, PXM2019_014213_000007), China Scholarship Council Program (No. 201907090062), and Ministry of Science and Technology, ROC (No. 109-2221-E-007-024).

References

[1] J. Han *et al.*, *Data Mining: Concepts and Techniques*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann., 2011.

[2] M. E. Newman, "Modularity and community structure in networks," *Proc. Nat. Acad. Sci. United States Amer.*, vol. 103, no. 23, pp. 8577–8582, 2006.

[3] C. D. Wang, J.-H. Lai, and J.-Y. Zhu, "Graph-based multiprototype competitive learning and its applications," *IEEE Trans. Syst., Man Cybern. C*, vol. 42, no. 6, pp. 934–946, Nov. 2012.

[4] E. Januzaj *et al.*, "DBDC: Density based distributed clustering," in *Proc. Int. Conf. Extending Database Technol.*, 2004, pp. 88–105.

[5] W. Gan *et al.*, "Data mining in distributed environment: A survey," *Wiley Interdisciplinary Rev.: Data Mining Knowl. Discov.*, vol. 7, no. 6, 2017, Art. no. e1216.

[6] P. Berkhin, "A survey of clustering data mining techniques," in *Grouping Multidimensional Data*. Berlin, Germany: Springer, 2006, pp. 25–71.

[7] A. Rosato *et al.*, "Recent advances on distributed unsupervised learning," in *Proc. Int. Workshop Neural Netw.*, 2015, pp. 77–86.

[8] N. Kushwaha and M. Pant, "Link based BPSO for feature selection in big data text clustering," *Future Gener. Comput. Syst.*, vol. 82, pp. 190–199, 2018.

[9] Y. Gu *et al.*, "Clustering-driven unsupervised deep hashing for image retrieval," *Neurocomputing*, vol. 368, pp. 114–123, 2019.

[10] J. M. Murphy and M. Maggioni, "Unsupervised clustering and active learning of hyperspectral images with nonlinear diffusion," *IEEE Trans. Geosci. Remote Sens.*, vol. 57, no. 3, pp. 1829–1845, Mar. 2019.

[11] A. Fahad *et al.*, "A survey of clustering algorithms for big data: Taxonomy and empirical analysis," *IEEE Trans. Emerg. Topics Comput.*, vol. 2, no. 3, pp. 267–279, Sep. 2014.

[12] R. Xu and D. Wunsch, "Survey of clustering algorithms," *IEEE Trans. Neural Netw.*, vol. 16, no. 3, pp. 645–678, May 2005.

[13] C. Bouveyron and C. Brunet-Saumard, "Model-based clustering of high-dimensional data: A review," *Comput. Statist. Data Anal.*, vol. 71, pp. 52–78, 2014.

[14] S. T. Roweis and L. K. Saul, "Nonlinear dimensionality reduction by locally linear embedding," *Science*, vol. 290, no. 5500, pp. 2323–2326, 2000.

[15] K. Chakrabarti and S. Mehrotra, "Local dimensionality reduction: A new approach to indexing high dimensional spaces," in *Proc. Int. Conf. Very Large Data Bases*, 2000, pp. 89–100.

[16] M. Liang, Q. Li, Y.-A. Geng, J. Wang, and Z. Wei, "REMOLD: An efficient model-based clustering algorithm for large datasets with spark," in *Proc. IEEE 23rd Int. Conf. Parallel Distrib. Syst.*, 2017, pp. 376–383.

[17] Q. Tong *et al.*, "Efficient distributed clustering using boundary information," *Neurocomputing*, vol. 275, pp. 2355–2366, 2018.

[18] L. Zeng *et al.*, "Distributed data mining: A survey," *Inf. Technol. Manage.*, vol. 13, no. 4, pp. 403–409, 2012.

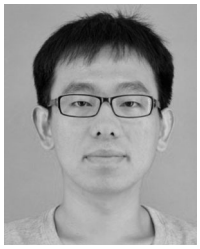
[19] K. M. Hammouda and M. S. Kamel, "Models of distributed data clustering in peer-to-peer environments," *Knowl. Inf. Syst.*, vol. 38, no. 2, pp. 303–329, 2014.

[20] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A K-means clustering algorithm," *J. Roy. Statist. Soc.. Series C (Appl. Statist.)*, vol. 28, no. 1, pp. 100–108, 1979.

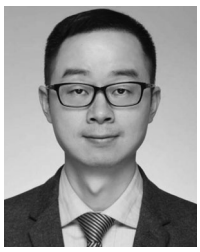
[21] J. Vaidya and C. Clifton, "Privacy-preserving K-means clustering over vertically partitioned data," in *Proc. 11th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2003, pp. 206–215.

[22] G. Ji and X. Ling, "Ensemble learning based distributed clustering," in *Proc. Pacific-Asia Conf. Knowl. Discov. Data Mining*, 2007, pp. 312–321.

[23] N. K. Visalakshi and K. Thangavel, "Distributed data clustering: A comparative analysis," in *Foundations of Computational, Intelligence Volume 6. Studies in Computational Intelligence*, vol. 206. Berlin, Germany: Springer, 2009, pp. 371–397.
[24] J. Jeong *et al.*, "Integration of distributed biological data using modified k-means algorithm," in *Proc. Int. Conf. Emerg. Technol. Knowl. Discov. Data Mining*, 2007, pp. 469–475.
[25] M.-F. F. Balcan *et al.*, "Distributed k-means and k-median clustering on general topologies," in *Proc. 26th Int. Conf. Neural Inf. Process. Syst.*, 2013, pp. 1995–2003.
[26] X. Xu *et al.*, "A fast parallel clustering algorithm for large spatial databases," *Data Mining Knowl. Discov.*, vol. 3, no. 3, pp. 263–290, 1999.
[27] M. Ester *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proc. 2nd Int. Conf. Knowl. Discov. Data Mining*, 1996, vol. 96, pp. 226–231.
[28] E. Januzaj *et al.*, "Scalable density-based distributed clustering," in *Proc. Eur. Conf. Princ. Data Mining Knowl. Discov.*, 2004, pp. 231–244.
[29] N. I. Wei-Wei *et al.*, "Local density based distributed clustering algorithm," *J. Softw.*, vol. 19, no. 9, pp. 2339–2348, 2008.
[30] A. Rodriguez and A. Laio, "Clustering by fast search and find of density peaks," *Science*, vol. 344, no. 6191, pp. 1492–1496, 2014.
[31] S. Gong and Y. Zhang, "EDDPC: An efficient distributed density peaks clustering algorithm," *J. Comput. Res. Develop.*, vol. 53, pp. 1400–1409, 2016.
[32] Y. Zhang, S. Chen, and G. Yu, "Efficient distributed density peaks for clustering large data sets in MapReduce," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 12, pp. 3218–3230, Dec. 2016.
[33] H.-P. Kriegel, P. Kroger, A. Pryakhin, and M. Schubert, "Effective and efficient distributed model-based clustering," in *Proc. 5th IEEE Int. Conf. Data Mining*, 2005, pp. 258–265.
[34] A. P. Dempster *et al.*, "Maximum likelihood from incomplete data via the EM algorithm," *J. Roy. Statist. Soc. Series B (Methodol.)*, vol. 39, pp. 1–38, 1977.
[35] X. Lin *et al.*, "Privacy-preserving clustering with distributed EM mixture modeling," *Knowl. Inf. Syst.*, vol. 8, no. 1, pp. 68–81, 2005.
[36] I. T. Jolliffe, *Principal Component Analysis and Factor Analysis*. New York, NY, USA: Springer, 2002, pp. 150–166.
[37] S. Mika *et al.*, "Kernel PCA and de-noising in feature spaces," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 1999, pp. 536–542.
[38] T. Hofmann *et al.*, "Kernel methods in machine learning," *The Ann. Statist.*, vol. 36, pp. 1171–1220, 2008.
[39] M. E. Tipping and C. M. Bishop, "Probabilistic principal component analysis," *J. Roy. Statist. Soc. Series B (Methodol.)*, vol. 61, no. 3, pp. 611–622, 1999.
[40] R. Vidal, Y. Ma, and S. Sastry, "Generalized principal component analysis (GPCA)," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27, no. 12, pp. 1945–1959, Dec. 2005.
[41] E. J. Candès *et al.*, "Robust principal component analysis?" *J. ACM*, vol. 58, no. 3, pp. 11:1–11:37, 2011.
[42] R. Agrawal *et al.*, "Automatic subspace clustering of high dimensional data," *Data Mining Knowl. Discov.*, vol. 11, no. 1, pp. 5–33, 2005.
[43] I. Assent, "Clustering high dimensional data," *Wiley Interdisciplinary Rev., Data Mining Knowl. Discov.*, vol. 2, no. 4, pp. 340–350, 2012.
[44] C.-H. Cheng *et al.*, "Entropy-based subspace clustering for mining numerical data," in *Proc. 5th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 1999, pp. 84–93.
[45] H. Nagesh *et al.*, "Adaptive grids for clustering massive data sets," in *Proc. SIAM Int. Conf. Data Mining*, 2001, pp. 1–17.
[46] C. M. Procopiuc *et al.*, "A Monte Carlo algorithm for fast projective clustering," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2002, pp. 418–427.
[47] K. Kailing *et al.*, "Density-connected subspace clustering for high-dimensional data," in *Proc. SIAM Int. Conf. Data Mining*, 2004, pp. 246–256.
[48] C. Bouveyron *et al.*, "High-dimensional data clustering," *Comput. Statist. Data Anal.*, vol. 52, no. 1, pp. 502–519, 2007.
[49] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proc. Annu. Symp. Comput. Geometry*, 2004, pp. 253–262.
[50] Y.-A. Geng *et al.*, "RECOME: A new density-based clustering algorithm using relative KNN kernel density," *Inf. Sci.*, vol. 436-437, pp. 13–30, 2018.
[51] C. M. Bishop, *Pattern Recognition and Machine Learning*. Berlin, Germany: Springer, 2006.
[52] C.-Y. Chi *et al.*, *Convex Optimization for Signal Processing and Communications: From Fundamentals to Applications*. Boca Raton, FL, USA: CRC Press, 2017.
[53] M. S. Bajwa, A. P. Agarwal, and S. Manchanda, "Ternary search algorithm: Improvement of binary search," in *Proc. IEEE Int. Conf. Comput. Sustainable Global Develop.*, 2015, pp. 1723–1725.
[54] R. A. Horn and C. R. Johnson, *Matrix Analysis*, 2nd ed. New York, NY, USA: Cambridge Univ. Press, 2012.
[55] F. Nie *et al.*, "Optimal mean robust principal component analysis," in *Proc. 31st Int. Conf. Mach. Learn.*, 2014, vol. 32, pp. 1062–1070.
[56] J. Huang *et al.*, "Spectral rotation versus k-means in spectral clustering," in *Proc. 27th AAAI Conf. Artif. Intell.*, 2013, pp. 431–437.
[57] B. Bahmani *et al.*, "Scalable k-means++," *Proc. VLDB Endowment*, vol. 5, no. 7, pp. 622–633, 2012.
[58] Hyperspectral remote sensing scenes, Accessed: Feb. 2020. [Online]. Available: http://www.ehu.eus/ccwintco/index.php/Hyperspectral_Remote_Sensing_Scenes
[59] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[60] G. Loosli *et al.*, "Training invariant support vector machines using selective sampling," in *Large Scale Kernel Machines*. Cambridge, MA, USA: MIT Press, 2007, pp. 301–320.
[61] A. Strehl and J. Ghosh, "Cluster ensembles—A knowledge reuse framework for combining multiple partitions," *J. Mach. Learn. Res.*, vol. 3, no. 3, pp. 583–617, 2002.
[62] R. R. Larson, "Introduction to information retrieval," *J. Amer. Soc. Inf. Sci. Technol.*, vol. 61, no. 4, pp. 852–853, 2010.



Yangli-ao Geng received the BS degree in information management and information system from Zhengzhou University, Zhengzhou, China. He is currently working toward the PhD degree in the School of Computer Science and Technology, Beijing Jiaotong University, Beijing, China. His research interests include data mining and machine learning.



Qingyong Li (Member, IEEE) received the BS degree in computer science and technology from Wuhan University, Wuhan, China, in 2001, and the PhD degree in computer science and technology from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2006. He is currently a professor with Beijing Jiaotong University, Beijing. His research interests include computer vision and artificial intelligence.

Mingfei Liang received the master's degree in computer science from Beijing Jiaotong University, Beijing, China, in 2018. He is currently an algorithmic engineer with Tencent, Beijing, China. His research interests include data mining and big data.

Chong-Yung Chi (Fellow, IEEE) received the BS degree from the Tatung Institute of Technology, Taipei, Taiwan, in 1975, the master's degree from National Taiwan University, Taipei, Taiwan, in 1977, and the PhD degree from the University of Southern California, Los Angeles, California, in 1983 all in electrical engineering. Currently, he is a professor of National Tsing Hua University, Hsinchu, Taiwan. He has published more than 240 technical papers, including more than 85 journal papers (mostly in the *IEEE Trans. Signal Processing*), more than 140 peer-reviewed conference papers, three book chapters, and two books, including a recent textbook, Convex Optimization for Signal Processing and Communications from Fundamentals to Applications, CRC Press, 2017 (which has been popularly used in a series of invited intensive short courses at the top-ranking universities in Mainland China since 2010 before its publication). Recently, he received 2018 IEEE Signal Processing Society Best Paper Award. His current research interests include signal processing for wireless communications, convex analysis and optimization for blind source separation, biomedical and hyperspectral image analysis, and graph signal processing. He has been a Technical Program Committee member for many IEEE sponsored and co-sponsored workshops, symposiums and conferences on signal processing and wireless communications, including co-organizer and general co-chairman of 2001 IEEE Workshop on Signal Processing Advances in Wireless Communications (SPAWC). He was an associate editor (AE) for four IEEE Journals, including the *IEEE Trans. Signal Processing* for nine years (5/2001–4/2006, 1/2012–12/2015), and he was a member of Signal Processing Theory and Methods Technical Committee (SPTM-TC) (2005–2010), a member of Signal Processing for Communications and Networking Technical Committee (SPCOM-TC) (2011–2016), and a member of Sensor Array and Multichannel Technical Committee (SAM-TC) (2013–2018), IEEE Signal Processing Society.

Juan Tan received the PhD degree in management from Hunan University, Changsha, China. She is currently an associate professor with the Beijing Technology and Business University, Beijing, China. Her research interests include data mining and innovation management.

Heng Huang received the BS and MS degrees from Shanghai Jiao Tong University, Shanghai, China, in 1997 and 2001, respectively, and the PhD degree in computer science from Dartmouth College, Hanover, New Hampshire, in 2006. He is a John A. Jurenko Endowed professor in computer engineering with the Electrical and Computer Engineering Department, University of Pittsburgh, Pittsburgh, Pennsylvania. He is also a consulting researcher with JD Finance American Corporation. His research interests include machine learning, big data mining, bioinformatics, and neuroinformatics.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

FULL-KV: Flexible and Ultra-Low-Latency In-Memory Key-Value Store System Design on CPU-FPGA

Yunhui Qiu, Jinyu Xie, Hankun Lv, Wenbo Yin, Wai-Shing Luk, Lingli Wang, *Member, IEEE*, Bowei Yu, Hua Chen, Xianjun Ge, Zhijian Liao, and Xiaozhong Shi

Abstract—In-memory key-value store (IMKVS) has gained great popularity in data centers. However, big data brings great challenges in performance and power consumption because of the general-purpose Von Neumann computer architecture. Remote direct memory access (RDMA) technology supporting zero-copy networking could partly alleviate the problem but is still not efficient for KVS. To overcome this problem, we present a flexible and ultra-low-latency IMKVS system named FULL-KV, based on a CPU-FPGA heterogeneous architecture. The FPGA serves as a KVS accelerator that can bypass the CPU and implement both the network stacks and the KVS processing with a highly parallel hardware architecture. The system latency of FULL-KV can achieve as low as 1.5μs/2.2μs for the PUT/GET operation, which is 3.0x/1.5x faster than current state-of-the-art hardware-based KVS systems. Besides, FULL-KV can support 4x larger values (up to 4M bytes). Given a total Ethernet bandwidth of 20Gbps, the peak throughput of the single-node FULL-KV can reach 26.0 million key-value operations per second (Mops). In the two-node test system with a commercial Ethernet switch, the peak throughput can reach 52Mops, manifesting the system scalability and practicability.

Index Terms—CPU-FPGA heterogeneous architecture, Hardware accelerator, In-memory key-value store, ultra-low-latency performance

1 Introduction

Data centers have been expanding rapidly in scale with the explosive growth of big data. Due to the heavy workload, they are facing great challenges in both system performance and power consumption. High-speed memory is thus in huge demand to pursuit higher performance. Thanks to the advancement of semiconductor technology, memory cost has decreased rapidly, and hence RAM usage has increased greatly in data centers. According to the usage of RAM, there are two main kinds of data storage systems, namely memory-cached systems and in-memory systems. As shown in Fig. 1a, the former ones store the data in non-volatile memory such as the hard disk drive (HDD) or Flash memory (Solid State Disk, SSD) whereas DRAM is acted as a cache for performance improvement. The HDD-based memory-cached storage systems have been widely used in data centers, such as Dynamo [1], Memcached [2]. Several flash-based systems have also been proposed in recent years [3], [4], [5]. The in-memory storage systems keep data entirely and permanently in DRAM, using non-volatile memory for data backups, such as Redis [6] and RAMCloud [7].

Compared with memory-cached storage systems, in-memory storage systems have several obvious advantages, including low latency, high bandwidth, and high I/O throughput, at the expense of high cost and power consumption [8]. For example, RAMCloud can provide 100-1000x higher throughput and 100-1000x lower access latency than disk-based systems [9]. Moreover, although the energy consumption per bit of DRAM is worse than HDD and Flash memory, in-memory storage can provide higher energy efficiency for applications with high throughput requirements, measured in terms of energy per operation [9].

To promote the system performance and reduce the power consumption, a variety of innovative technologies have been applied to data storage systems, from kernel-space network stack to user-space stack, then to hardware-based implementations, such as remote direct memory access (RDMA), further offloading both the network stack and partial application functions to FPGA for acceleration, as shown in Fig. 1b, where RNIC indicates the RDMA-based network interface card (NIC).

Network communication, as an indispensable infrastructure of the distributed storage system, is a key factor that determines the performance and power consumption. To decrease the overhead of the generality of the traditional kernel-space network stacks, user-space stacks were proposed [10], [11]. Some user-space network I/Os, such as DPDK [12], can provide line-rate performance for high-speed network links (up to hundreds of Gbps) [13]. Furthermore, RDMA that features zero-copy technology makes a huge step in reducing the network communication cost.

- *Y. Qiu, J. Xie, H. Lv, W. Yin, W.-S. Luk, and L. Wang are with the State Key Laboratory of ASIC and System, Fudan University, Shanghai 201203, China. E-mail: {yhqiu18, jyxie18, hklv18, wbyin, luk, llwang}@fudan.edu.cn.*
- *B. Yu, H. Chen, X. Ge, Z. Liao, and X. Shi are with the Huawei Chengdu Research Institute, Chengdu 611731, China. E-mail: {yubowei, watson.chen, hexianjun, liaozhijian, shixiaozhong}@huawei.com.*

Manuscript received 18 June 2019; revised 18 Jan. 2020; accepted 10 Feb. 2020. Date of publication 14 Feb. 2020; date of current version 26 Mar. 2020. (Corresponding author: Yunhui Qiu.)
Recommended for acceptance by M. Kandemir.
Digital Object Identifier no. 10.1109/TPDS.2020.2973965

1045-9219 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

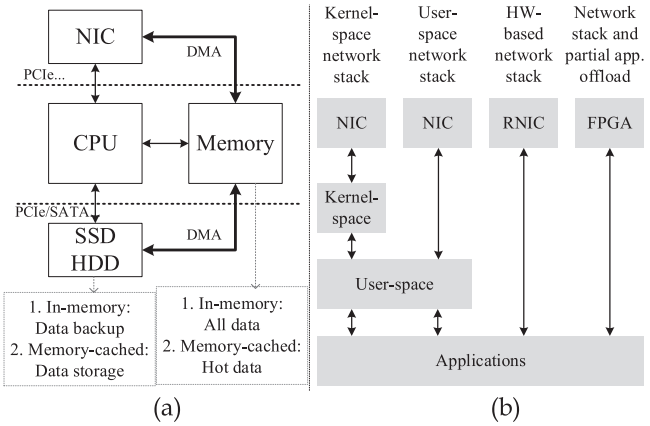


Fig. 1. Data storage system development. (a) The generic storage server architecture. In the in-memory storage systems, all the data are stored in the host memory while SSD or HDD is used for backup. The memory-cached systems store all the data in SSD or HDD and cache the hot data in the host memory. (b) The storage system development trend. To reduce the network communication overhead, the network stack first shifts from the kernel space to user space, and then hardware-based implementations, such as RDMA, are proposed. Furthermore, both the network stack and partial application functions are offloaded to FPGA for acceleration.

Two types of technologies have been evolved, namely one-side RDMA [14], [15], [16], [17] and two-side RDMA [18], [19]. The former shifts the storage workload to clients. It may suffer from the high overhead of the communication and synchronization between the clients and servers. The latter performs the storage in the servers where the performance is bounded by CPU [20].

In the data-intensive applications, the Von Neumann computer architecture has been a bottleneck that limits the system performance and causes high power consumption. It is mainly because the separation of compute and storage leads to huge data copy and communication costs. Datacenter, as one of the most data-intensive applications, suffers heavily from such an architecture. RDMA technology partly alleviates the problems by reducing the CPU involvement in the network stacks. But better solutions are emerging based on the FPGA deployments in data centers. A new research trend is to offload part of the storage processing together with the network stacks from CPU to FPGA to achieve higher performance and lower power consumption.

Currently, most of the studies [20], [21], [22] are based on the key-value store (KVS) systems, where the basic storage units are associative arrays, called key-values. The KVS system is one type of NoSQL database. It has been widely used in contemporary data centers, such as Dynamo [1], Memcached [2], Redis [6], and RAMCloud [7]. Under different application scenarios, the key-value length can be fixed or variable. For example, in Facebook's Memcached deployment, the USR pool to store user-account status information has two key sizes and one value size, whereas many more pools, such as APP, ETC, VAR, and SYS, have variable-size key-values [23]. In many KVS systems, small-size key-values dominate the workload, but there are also a few large-size ones. For instance, the ETC pool in Facebook's Memcached has a few very large values about 1M bytes. The length of the string-type value can reach up to 512M bytes in Redis [6]. These requests with large values consume a significant share of the available resources due to their much longer service times [24]. However, the supported value size was less than 4K bytes in most current hardware-based KVS systems [20], [21], [22], [25], [26], [27]. Only a few supported large-size values [5], [28], but no more than 1M bytes. There is still room to increase the value size in FPGA-based KVS systems.

In those hardware-based systems, the storage schemes vary with the storage medium, including SRAM, Content Addressable Memory (CAM), DRAM, and Flash memory. Examples are:

1. The hash table is stored in SRAM and values in DRAM [21].
2. Key-values are stored together in CAM [28].
3. Both the hash table and values are stored in onboard DRAM [22], [25], [26], [27].
4. The hash table is stored in DRAM and values in Flash memory [5].
5. The hash table is stored in onboard DRAM and values in the host memory [20].

Since the storage capacities of both SRAM and CAM are very small, the first two storage schemes are rather limited in data-center applications. The third and fourth schemes are suitable to establish a KVS system independent of CPU, thus making it hard to interact with the CPU for dynamic memory management. Considering the popularity of in-memory KVS (IMKVS) systems, it is more promising to adopt the fifth scheme. However, the supported value length is not long enough in current hardware-based KVS systems employing this scheme [20].

In summary, although considerable studies have been carried out in hardware-based KVS systems, few can apply to the IMKVS system and provide hardware-level support for large values, such as up to 4M bytes. In this paper, we present FULL-KV, a flexible and ultra-low-latency IMKVS system based on the CPU-FPGA heterogeneous architecture. FPGA serves as a KVS accelerator that can bypass CPU and implement both the network stacks and the KVS processing. Several novel technologies are exploited to achieve ultra-low latency in response time, high scalability in storage capacity, great flexibility in the value size, high efficiency in the KVS operation processing, and full Ethernet line-rate performance in system throughput. First, we design a highly parallel accelerator architecture to achieve ultra-low-latency performance. Second, we adopt a decoupling design scheme that stores the hash table in onboard DRAM memory and values in the host memory. The workload of the system can scale up with the storage capacity. Third, we design a novel fragmentation strategy that can provide support for super large values. Fourth, we propose a new memory pre-allocator, which can hide the latency of the interrupt processing and the memory-mapped I/O (MMIO) of the PCIe interface through the host memory pre-allocation. It can not only dramatically reduce the latency but also improve the processing efficiency for the PUT operation. Based on the above improvements, FULL-KV can achieve high efficiency: at most one PCIe-DMA for each GET or PUT operation. Finally, with the design schemes of pipeline, cache, and parallelism, FULL-KV can attain full Ethernet line-rate performance, including the verified bandwidths of 10Gbps and 20Gbps. Besides, FULL-KV has a

scalable architecture, which can achieve higher performance by increasing the functional modules.

FULL-KV is based on our previous work [29], and we make the following extensions:

1. Design a storage request collision (e.g., read-after-write on the same KV) detector that can significantly reduce the pipeline stalling rate. The processing of the KVS operations can run at nearly fully-pipelined performance unless the collision happens, which is at ultra-low possibility. The peak throughput for the PUT operation is promoted from 4.3 million operations per second (Mops) to 13.0Mops on average for each 10Gbps Ethernet (GbE) interface.
2. Enable the host CPU to build or update the hash table in the onboard DRAM memory after power-off or value address changes due to the host memory management.
3. Broaden the Ethernet bandwidth by combining two 10GbE interfaces, which can double the system throughput and keep the system latency almost the same.
4. Scale the number of the storage nodes to two and introduce a commercial Ethernet switch between the client and storage nodes. The total system throughput is doubled, achieving 51.9Mops at most, which verifies the scalability and practicability of FULL-KV.

Compared with current hardware-based KVS systems [5], [28], FULL-KV has higher flexibility, where the supported value range is 4x wider (from 1 byte to 4M bytes). The system latencies for the PUT and GET operations are as low as 1.5μs and 2.2μs (no Ethernet switch), respectively, which are 3.0x and 1.5x faster than current state-of-the-art KVS systems [16], [20], [25]. With two 10GbEs, the peak throughput of the single-node FULL-KV can reach 26.0Mops, achieving nearly full utilization of the Ethernet bandwidth. When the number of the storage nodes scaled to two, the system throughput can be doubled, to 52Mops, whether there is the Ethernet switch or not.

2 Background and Related Work

In KVS systems, the basic storage unit is a key-value pair (KVP), where the value is directly or indirectly associated with the key. The key-value length can be fixed or variable under different application scenarios. In current hardware-based KVS systems, few can support large-size values. The maximum value size was 1M bytes in [5], [28]. Nevertheless, in [28], the value was divided into multiple parts by software APIs and stored in multiple locations if its length was larger than 352 bits. This would not only tremendously reduce storage efficiency but also increase the processing latency. In [5], Flash memory was used to store values whereas DRAM as a data cache. Although the supported value size is rather large, the gap still exists compared with the software-based KVS systems.

According to the storage mode of KVPs, there are decoupling and non-decoupling design schemes. The latter stores a KVP as a whole, whereas the former associates the value with the key through a hash table. The hash table contains keys and the storage addresses of values. It can be stored separately from values. When the value size is large, the former scheme is more economical and efficient on the utilization of storage space [21], [22]. Since our KVS system provides support for large-size values, we adopt the decoupling design scheme. It is also widely used in current FPGA-based KVS systems, but the storage scheme for the hash table and values varies with the storage medium.

The selection of the storage medium is determined by two factors: storage capacity and data access latency. Nowadays, there are three common choices: SRAM, DRAM, and Flash memory. Their commercial storage capacity increases sequentially, but the data access speed decreases in turn. In general, the storage space needed for the hash table is much smaller than that for values. Therefore, SRAM or DRAM is usually utilized to store the hash table, whereas DRAM or Flash memory for values. In [21], the hash table was stored in SRAM. Fully-pipelined performance could be achieved because of the low latency of SRAM. But the small storage capacity of SRAM would limit its applications in data centers. Therefore, DRAM has been applied to store the hash table in many KVS systems [5], [20], [22]. For the storage of values, onboard DDR3 memory was used in [22], [25], [26], [27], whereas Flash memory was used in [5]. Both schemes can build a KVS system independent of CPU to achieve high performance and low power consumption, especially the latter scheme. But it is hard for them to manage memory utilization dynamically and provide software application programming interfaces (APIs) for the administers. A more promising storage scheme is to store the hash table in DRAM and values in the host memory. In this way, the hardware-based accelerator can apply to the IMKVS systems to further reduce the access latency and decrease the power consumption significantly. This scheme was employed in [20], which leveraged a programmable network interface card (NIC) to extend RDMA primitives. High performance could be achieved but the supported value size is rather limited.

In KVS systems, different KVs may be mapped to the same storage address, which will lead to the hash collision problem. A variety of hashing algorithms have been proposed to resolve the problem, including cuckoo hashing [30], hopscotching hashing [31], and chained hashing [22]. Chained hashing is based on the chained list. The latency of operations increases as the workload of the hash table rises. Hopscotching hashing consumes constant time on average for a lookup, but it may need to search several buckets for each operation, which is more complex than cuckoo hashing. Cuckoo hashing can guarantee a constant time for a lookup and achieve high storage utilization [32]. Therefore, we employ the cuckoo hashing to resolve hash collisions in FULL-KV.

There is another problem in KVS systems, namely the data consistency problem, such as read-after-write on the same KV. It is because the hash table is a shared data memory and there is a processing latency between the searching and updating procedures. In [21], the previous requests during the latency are cached in registers. The problem can be eliminated by searching both the hash table and the registers. Since the hash table is stored in BRAM, whose access latency is only one or two clock cycles, the required registers are very few and fully-pipelined performance can be implemented easily. But if the hash table is stored in DRAM, the latency increases heavily. Although more registers can be used to cache the previous requests, it is very hard to achieve fully-pipelined performance when the

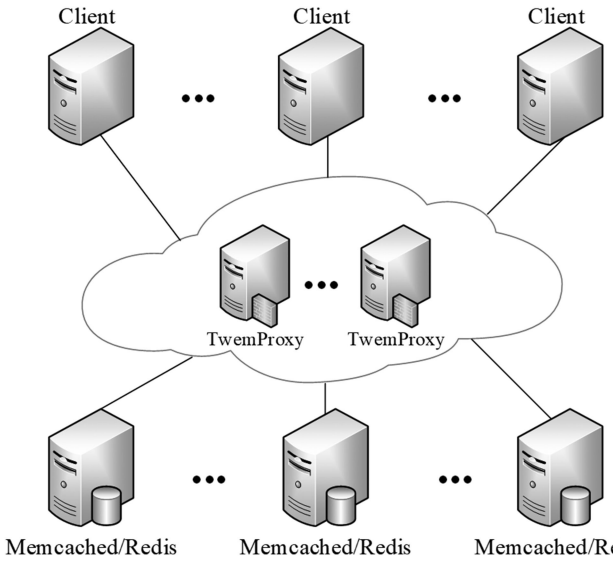


Fig. 2. Deployment architecture of the distributed KVS system with TwemProxy [37].

problem happens. In [20], the author proposed another solution: using hash functions together with BRAM memory to detect the data conflict. If different keys are mapped to the same BRAM address, it is treated as data conflict. Otherwise, the pipeline can run smoothly. This method is easy to implement and can guarantee data conflict is detected. But it is over-detected. The data conflict rate can become rather high due to the small BRAM capacity and hash collision. We propose a novel scheme to detect the data conflict based on the solution in [21]. A certain number of previous requests are cached in registers and compared with current request to check data conflict. If it does not happen, the system can run in a fully-pipelined performance. Otherwise, the pipeline will be stalled until the previous request is completed. Thanks to the wide range of the keys, the pipeline stalling rate is ultra-low.

With the exponentially growing of workload in the data center, a distributed KVS system has scaled up to tens of thousands of servers. For example, the IMKVS system, RAMCloud, is designed to support at least 10,000 servers [9]. In such a large-scale distributed system, it is common that failures and repairs occur, with the rate growing linearly with the system scale [33]. Consistent hashing [34] is a terrific solution that can recover the data stored in the failed servers with very small overhead. Only the KVs originally mapped to the failure node need to be remapped to nearby nodes, without remapping all the KVs. Due to its high scalability and easy manageability, consistent hashing has been used in Memcached [2] and Redis [6]. Furthermore, consistent hashing based distributed KVS systems are emerging in data centers [35], such as Sheepdog [36]. Based on consistent hashing, Twitter proposes TwemProxy, a fast and lightweight proxy for Memcached and Redis protocol, to improve the scalability of the distributed system [37]. The system architecture is shown in Fig. 2. FULL-KV serves as a server as in such systems. To explore the scalability of FULL-KV as a storage server, we scale FULL-KV to multiple nodes and use a custom client and Ethernet switch to test the system performance.

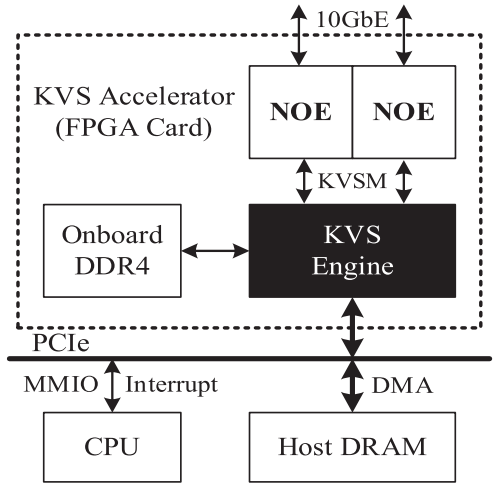


Fig. 3. FULL-KV architecture based on CPU-FPGA, with two network offload engines (NOEs).

3 FULL-KV System Architecture

3.1 Single-Node Architecture

FULL-KV is based on a CPU-FPGA heterogeneous architecture, where the FPGA card is connected with the host computer through the PCIe interface, as shown in Fig. 3. The FPGA card serves as a KVS accelerator, which can bypass the host CPU and directly process all standard KVS operations and network communication. There are two main modules in the KVS accelerator: the KVS engine (KVSE) and the network offload engine (NOE). KVSE is designed for the processing of KVS operations with the following functions:

1. Process KVS requests from the two NOEs and then return responses;
2. Build and manage the hash table stored in the onboard DDR4 memory;
3. Retrieve or store values from or to the host DRAM memory through the PCIe DMA;
4. Communicate with the host CPU through the interrupt and the MMIO of the PCIe interface.

NOE implements the processing of the user datagram protocol (UDP) with FPGA. It is responsible for the transmission of KVS request and response messages over the 10GbE. Two NOEs are equipped to increase the total bandwidth of the Ethernet. The IP/MAC addresses of the two NOEs can be both configured by the host CPU through the PCIe MMIO.

According to the characteristic of the KVS messages, we propose a novel application layer protocol, named key-value store message (KVSM) protocol, to format the KVS request and response messages. As shown in Fig. 4, a KVSM can be

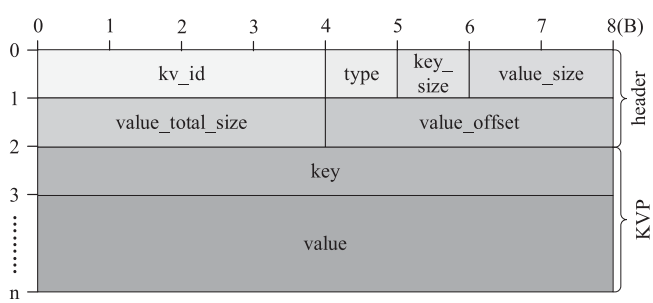


Fig. 4. KVSM packet format, including two parts: header and KVP.

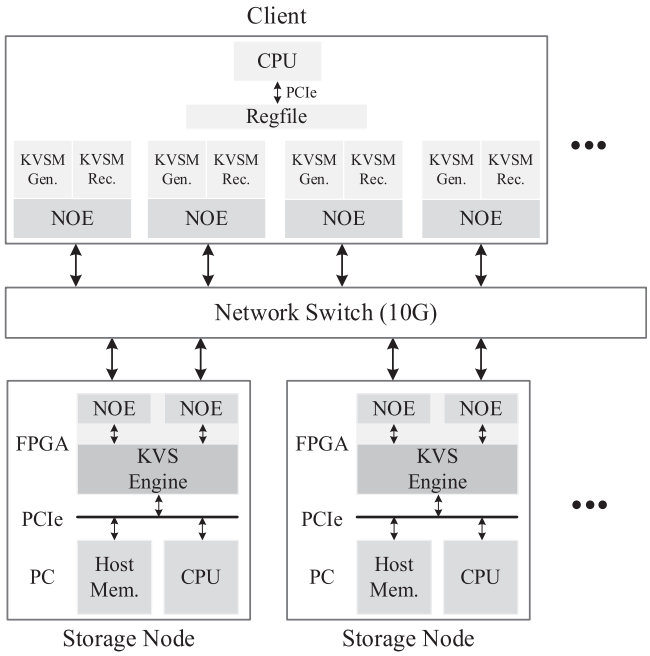


Fig. 5. The multi-node system architecture of FULL-KV. The clients communicate with the storage nodes through a 10G Ethernet switch, routed by IP/MAC address.

divided into two parts: a header and a KVP. The *kv_id*, *type*, and *key_size* indicate the KVSM identifier, the KVS operation type, and the key length, respectively. The *value_size*, *value_total_size*, and *value_offset* indicate the value length in the current KVSM, the total value length, and the offset of the fragmented value, respectively. They are used for the fragmentation processing of large-size values. The length of each item is shown in Fig. 4. The key size is eight bytes, whereas the value length is no more than 1K bytes in a KVSM. The KVSM protocol is used for the communication between KVSE and the NOEs.

3.2 Multi-Node Architecture

Due to the high scalability and manageability of the consistent hashing based distributed KVS system, we design FULL-KV as the server in such a system. To explore the scalability of FULL-KV, we deploy a multi-node system architecture for test and verification, which can be equipped with multiple storage nodes and clients, as shown in Fig. 5. We use a commercial Ethernet switch as a router based on the IP/MAC addresses. Since the IP/MAC addresses of each storage node and the client can be configured by the host CPUs, any two NOE interfaces can communicate with each other. This significantly improves the flexibility of the system deployment.

According to the strategy of consistent hashing, the KVs transferred via different NOE interfaces are partitioned to different key regions. Therefore, the peak system throughput on the server-side can increase linearly with the number of storage nodes. The hash partitions are handled by the proxies shown as Fig. 2, whose capabilities may limit the system performance. To test the scalability of FULL-KV, we utilize our custom client rather than the proxy plus client to simulate the multi-client behaviors. The other reason for the utilization of the custom client is that it can make full use of the bandwidth of the 10GbE.

The custom client also consists of an FPGA card and a computer. The FPGA card contains four NOEs, four *KVSM Generators*, and four *KVSM Receivers*, as shown in Fig. 5. *KVSM generator* can generate request KVSMs of different modes, such as PUT/GET/DELETE/Random operations for fixed/variable-length values. *KVSM receiver* mainly receives the response KVSMs. The host computer can configure the number and mode of the request KVSMs, the IP/MAC addresses of the NOEs, and attain the statistics about the system performance through the PCIe-DMA.

4 Implementation

The architecture of KVSE is illustrated in Fig. 6, which consists of the following modules:

1. *KVSM Parser* parses the request KVSMs from NOE and then transmits the request headers to *Hash Engine* and values to *C2H Controller*.
2. *KVSM packer* formats the processed results of the KVS operations into response KVSMs and then transmits them to NOE.
3. *Hash Engine* looks up the hash table and executes corresponding processing based on the operation type after receiving a request header. In case of PUT miss, where the key is not found, *Hash Engine* will submit a request to *Segment Management* for contiguous memory space to store the whole value. A DMA descriptor in the direction of card-to-host (C2H) will be then transmitted to *C2H Controller*. In case of GET hit, *Hash Engine* will submit a host-to-card (H2C) DMA descriptor to *H2C Controller*. The descriptor contains the address and length of the value stored in the host DRAM memory.
4. *DDR4 Controller* provides easy-to-use user interfaces for *Hash Engine* to access the onboard DDR4 memory. Since the clocks used for this module and *Hash Engine* are from different sources, cross-clock-domain asynchronous FIFOs are utilized inside for data synchronization.
5. *Segment Management* caches a certain amount of preallocated host memory segments so that it can provide host memory space to *Hash Engine* instantly when needed.
6. *C2H Controller* stores values to the host memory through PCIe-DMA, whereas *H2C Controller* fetches values from the host memory through PCIe-DMA. Both modules will execute fragmentation processing for large-size values. In other words, transfer the values in slices.
7. *Synchronizer* enables the host CPU to build or update the hash table after power-off or value address changes due to host memory management.
8. *Event Controller* collects event messages and then generates interrupt requests (IRQs) to inform the host CPU for corresponding processing.
9. *DMA Controller* controls the transmission of values from or to the host memory, the interrupt processing, and the PCIe MMIO. It can provide four PCIe-DMA channels for data transmission.
10. *Regfile* manages a set of configuration registers used for the communication between the KVS accelerator and the host CPU.

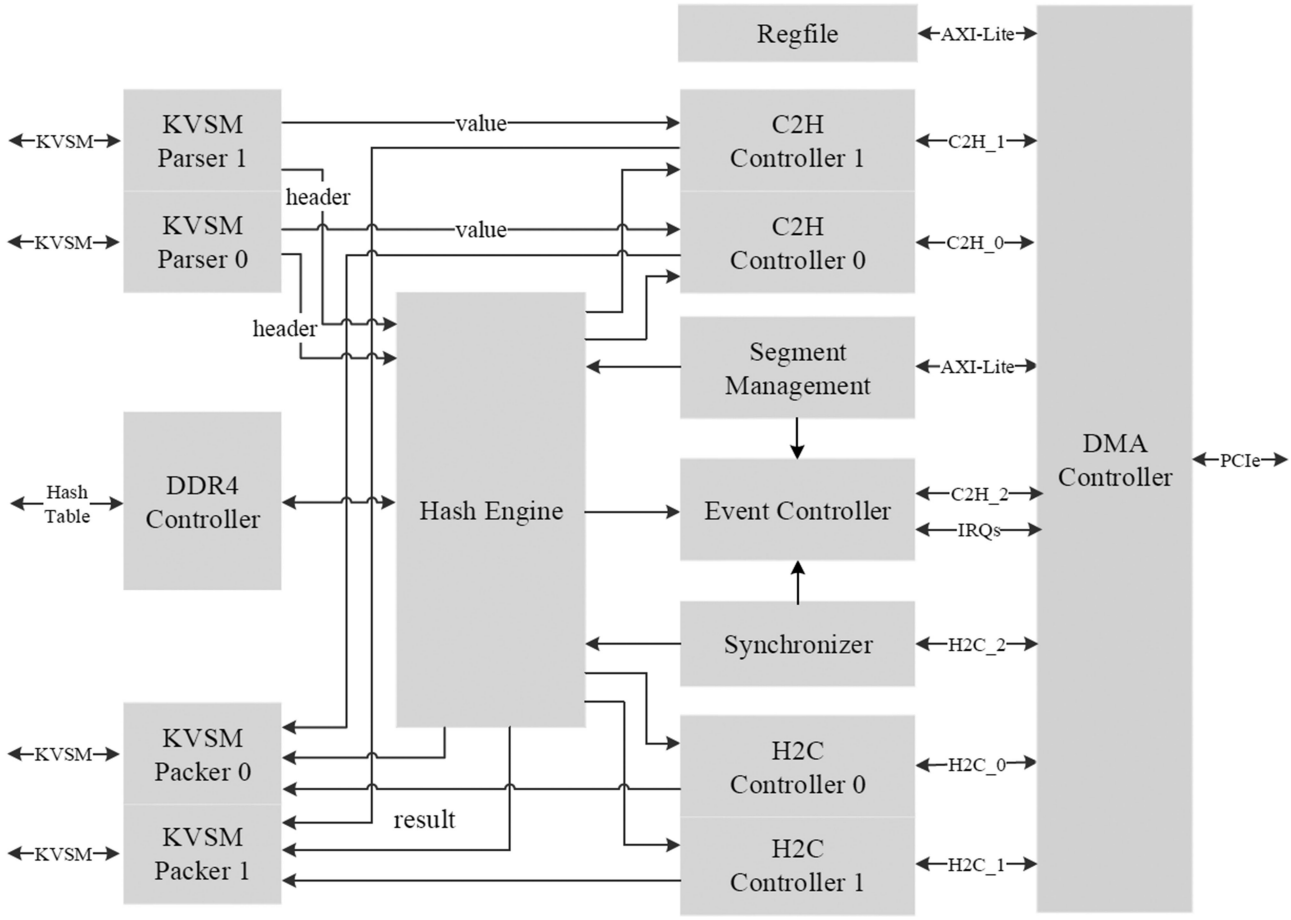


Fig. 6. Parallel architecture of KVSE with one Hash Engine, one DDR4 Controller, one PCIe-DMA Controller, and two sets of KVSM Parser/Packer and C2H/H2C Controller.

KVSE is connected with two NOEs, one of which links with *KVSM Parser/Packer 0* and the other with *KVSM Parser/Packer 1*. As described in Section 3.2, the KVs from the two NOEs are partitioned to different key regions, so there is no temporal dependency between the requests from the two NOEs. They can be treated as fully separate channels.

The architecture of KVSE is highly scalable. There are three kinds of peripheral devices/interfaces, namely Ethernet, DDR4, and PCIe. One *Hash Engine* and one *DDR4 Controller* are corresponding to one DDR4 physical interface. One *KVSM Parser/Packer* and one *C2H/H2C Controller* are corresponding to one NOE and one PCIe-DMA channel. We can adjust the number of these modules to attain better performance according to the bandwidths of the Ethernet, DDR4, and PCIe.

Take the PUT operation for an example to illustrate the run flow of the system. During system initialization, the host CPU pre-allocates a set of host memory segments and write relevant information to *Segment Management* through PCIe MMIO. After receiving a request from NOE, *KVSM parser* parses the request KVSM and submits a request header to *Hash Engine* and the value to *C2H Controller* for the PUT operation. *Hash Engine* then looks up the hash table to detect if there is the same key. If yes, it transfers the result to *KVSM Packer*. Otherwise, it first submits a request to *Segment Management* for contiguous memory space to store the value. After receiving the memory address, it writes a new entry to the hash table and generates a whole DMA descriptor to *C2H Controller*. Besides, a log message is sent to *Event Controller*. According to the value size, *C2H Controller* may slice the whole descriptor to several fragmented descriptors, each corresponding to one DMA transmission of the value. When the cached value data reaching one frame (1K bytes by default), a descriptor is submitted to *DMA Controller* to launch the value transmission. After completing all the transmissions, it sends the result to *KVSM Packer*, which will format the result into a response KVSM.

4.1 KVSM Parser and Packer

KVSM Parser is designed to parse the request KVSMs from NOE. It extracts *kv_id*, *type*, *key_size*, *value_total_size*, and *key* from the KVSM and generates a request header submitting to *Hash Engine*. Although a large KVP is distributed in a series of KVSMs, only one header is generated according to the first KVSM so that only one hash table entry is created for the KVP. The parsed values, if there are in the KVSMs, are transmitted to *C2H Controller*. Therefore, the value transmission and the hash table management can run in parallel.

KVSM Packer formats the processed results of all the KVS requests into response KVSMs, which are then transferred to NOE. There are six kinds of results from three sources: *Hash Engine*, *C2H Controller*, and *H2C Controller*. *Hash Engine* may generate four kinds of results: PUT hit, GET miss, and DELETE hit/miss. *C2H Controller* and *H2C Controller* generate the other two results, PUT miss and GET hit, respectively.

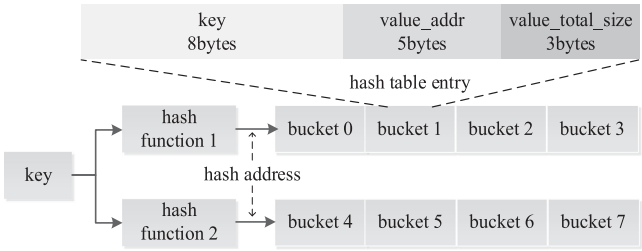


Fig. 7. Hash mapping structure with two hash functions and eight buckets. Each bucket stores a hash table entry, containing an 8-byte *key*, 5-byte *value_addr*, and 3-byte *value_total_size*.

4.2 Hash Engine

In FULL-KV, we employ a decoupling design scheme: storing the hash table in onboard DDR4 memory and values in the host DRAM memory. *Hash Engine* is designed to build and manage the hash table, which consists of a great number of fixed-size hash table entries. Each one is defined as 16 bytes, as shown in Fig. 7, where *value_total_size* and *value_addr* indicate the total size and the storage address of the value, respectively.

In terms of the design of *Hash Engine*, we improve the Cuckoo hashing algorithm [30] to make full use of the bandwidth (512 bits per clock cycle) of DDR4. Two hash functions are used, each mapping a key to four candidate buckets, as shown in Fig. 7. Each bucket can store one hash table entry. Jenkins's hash function [38] is selected in the design due to its excellent performance and high popularity in the open-source community.

The eight hash table entries are read out in two DDR4 clock cycles. The request key is then compared with those contained in the eight entries simultaneously, leveraging the parallelism available in FPGA. When a new hash table entry needs inserting into the hash table, a polling mechanism is used to select the inserting position from the empty buckets, which is in ascending order of the numbers shown in Fig. 8. When the hash collision happens, which means the optional eight buckets are all occupied in our design, one old hash table entry has to be kicked out in order to insert the new one. The kicked-out one will be then reinserted into the hash table. Similarly, the kicking-out position is also selected by polling as shown in Fig. 8.

The processing for a KVS operation can be divided into three steps, as shown in Fig. 9. First, calculate hash addresses by the two hash functions. Second, search the hash table according to hash addresses and then compare the request key with those in the eight buckets. Finally, execute corresponding processing based on the comparison result and the

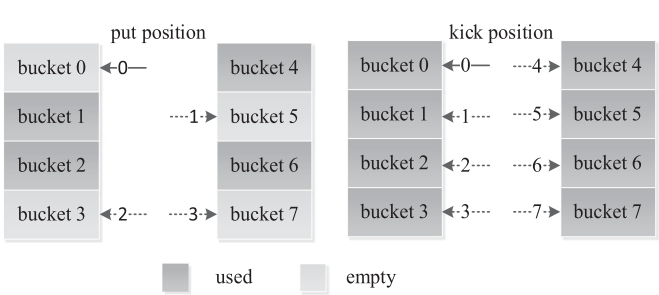


Fig. 8. Inserting and kicking-out strategy that selects the available position by polling.

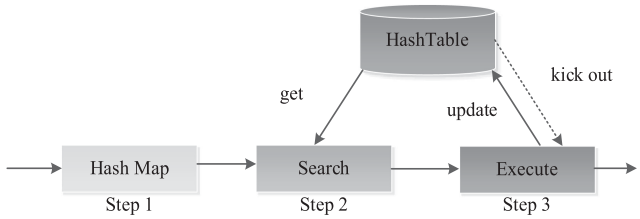


Fig. 9. Hash operation data-flow with three steps: Hash Map, Search, and Execute.

operation type. In case of PUT miss or DELETE hit, the hash table needs updating: inserting a new hash table entry or deleting one. In the former case, a hash table entry may be kicked out and then reinserted due to the hash collision. All these cases may lead to data consistency problems, such as the read-after-write problem. More precisely, if either of the two hash addresses of the current request is the same with those of the previous hash-table-updating (e.g., PUT/DEL) requests during the last two steps, the data consistency problem happens. Fortunately, the probability is actually rather low since the range of the key is very large.

To resolve the problem, we propose a novel design scheme: compare the two hash addresses of the current request with those of a certain number of previous hash-table-updating requests before the second step. If none of them are the same, execute step 2 and 3 at once. Otherwise, stall the pipeline until the previous request with the same hash address completes step 3. Besides, the number of the compared previous requests guarantees all the requests not updating the hash table yet are included.

The hardware architecture of *Hash Engine* is shown in Fig. 10. The *hash_fun* and *flag* modules are designed to implement step 1, where the latter module marks the hash-table-updating requests. The *ht_addr_cmp* and *FIFO* modules are used to resolve the data consistency problem. When it happens, hash addresses cannot be read out from *FIFO* until the previous request causing the problem to update the hash table. The *P2S*, *DDR4 controller*, *S2P*, *key_cmp*, and *kick_key_cmp* modules implement step 2. The *P2S* and *S2P* modules are designed for serial-parallel conversion. The *key_cmp* and *kick_key_cmp* modules compare the current request key with the ones stored in the eight buckets and the kicked-out hash table entries, respectively. Similarly, a certain number of kicked-out hash table entries are cached in the *kick_key_cmp* modules. The *key buffer* and *value_size & op_type buffer* are used for data synchronization in the pipeline. The *Hash Execute* module is corresponding to step 3. According to the operation type and the comparison result, it may generate a C2H/H2C DMA descriptor, hash table entry, reinserting request, log and event messages.

4.3 C2H & H2C Controller

C2H Controller retrieves values from the host memory through PCIe-DMA. On the contrary, *H2C Controller* stores values into the host memory. Since the value size may be very large, we design a novel fragmentation processing mechanism to improve storage efficiency.

For a single KVP, *KVSM Parser* submits only one header, and *Hash Engine* generates only one hash table entry. In case of PUT miss and GET hit, *Hash Engine* submits a whole

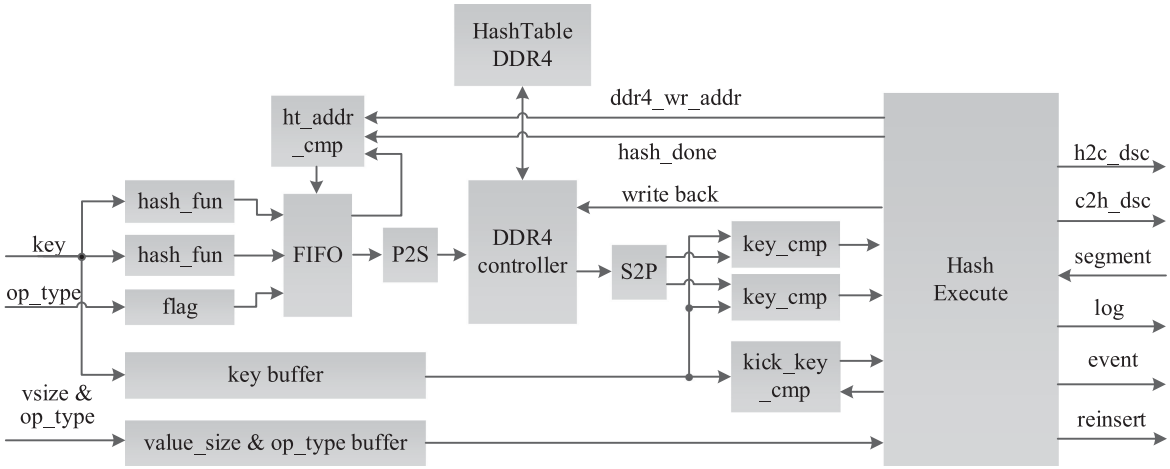


Fig. 10. *Hash Engine* architecture, almost fully-pipelined except that the output of the FIFO will be stalled when the data consistency problem happens.

DMA descriptor to *C2H Controller* and *H2C Controller*, respectively. Both modules will slice the descriptor into several fragmented descriptors, each corresponding to a frame of value data, 1KB by default, as shown in Fig. 11. In *C2H Controller*, a fragmented descriptor is submitted to *DMA Controller* to launch DMA transmission after the amount of the value data reaches one frame. In *H2C Controller*, multiple fragmented descriptors are first submitted to *DMA Controller*. After the amount reaches a frame, the received value data are directly packed into a response KVSM and transmitted to *KVSM Packer*.

In both modules, the value buffer is designed as a ring buffer implemented with dual-port BRAM memory, as shown in Fig. 11. Thanks to the hardware parallelism, reading and writing can be carried out simultaneously. As long as writing is ahead of reading at first, there are always valid data in the buffer to feed the pipeline. Therefore, the value transmission between the host memory and the FPGA card can achieve high throughput.

4.4 Segment Management

Segment Management can provide host memory space for storing values to *Hash Engine* instantly when needed by means of memory pre-allocation. Traditionally, in order to store values into the host memory, interrupts are used to inform the host CPU to allocate memory space. The memory addresses are then written back to the requester. This long interrupt processing time will dramatically increase the latency of the PUT operation. We design a novel memory allocation scheme to hide the latency.

During system initialization, the host CPU pre-allocates a set of host memory segments (256, by default) and write the relevant information to a segment buffer in *Segment Management* through the PCIe MMIO. After receiving a request from *Hash Engine*, it can allocate proper memory space instantly. When the utilization of the segments reaches a certain threshold (32, by default), an event message will be submitted to *Event Controller*, which will then inform the host CPU to update the used segments. This guarantees that there are always available segments in *Segment Management* unless the host CPU cannot allocate any new memory segment. Therefore, the latency of the PUT operation can be significantly reduced.

The segment buffer is implemented by a dual-port BRAM memory, as shown in Fig. 12. The default segment size is 4MB, which is the maximum size of contiguous memory space the host CPU can allocate. Each segment is managed mainly by its 64-bit address. When a value is inserted,

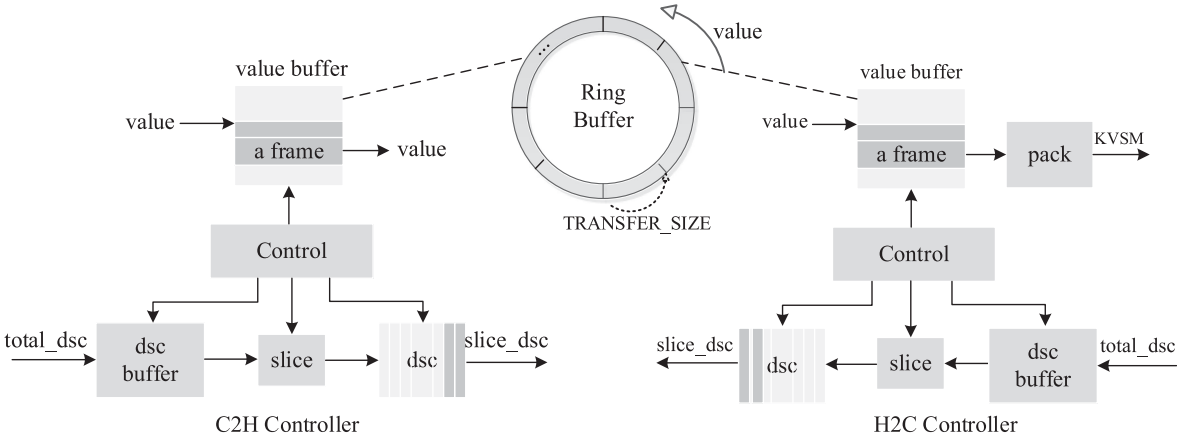


Fig. 11. The hardware architectures of *C2H Controller* (left) and *H2C Controller* (right), where the value buffer is designed as a ring buffer based on dual-port BRAM.

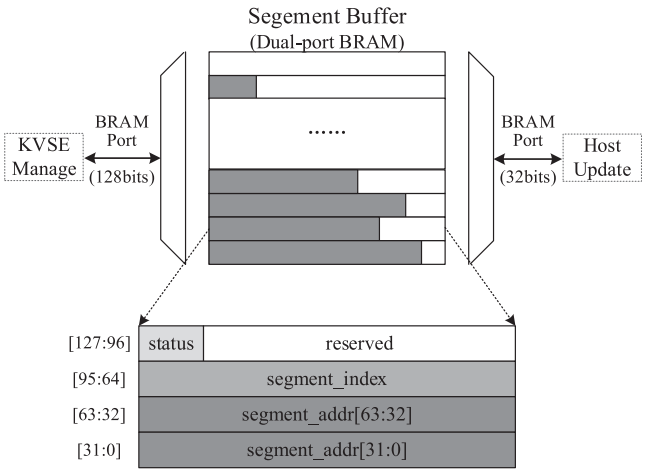


Fig. 12. Dual-port segment buffer structure and the cached segment entry format.

the remaining memory space is marked by updating the offset address which refers to the least significant 22 bits. In the segment entry, the *status* is used to indicate the busy and empty/full status of the segment. The *segment_index* is used by the host CPU. Of the two BRAM ports, one is used to provide memory space for values, and the other to initialize and update segments by the host CPU.

4.5 Synchronizer

When FULL-KV restarts or the value addresses change due to the host memory management, the hash table stored in the onboard DDR4 memory needs recovering or updating. *Synchronizer* provides the capacity for the host to build and update the hash table.

The hardware architecture of *Synchronizer* is shown in Fig. 13. The host CPU caches the data to be synchronized in the *sync_segment* and informs *Synchronizer* the storage addresses by *syn_regs*, whose values are updated by PCIe MMIO. According to the addresses, *Synchronizer* generates H2C DMA descriptors to fetch the data. The *sync_data* from the host is then transmitted to *Hash Engine*. After completing the transmission, *Synchronizer* will generate event messages to inform the host that the data in *sync_buffer* has been fetched.

4.6 Event Controller

Event Controller collects event messages from *Hash Engine*, *Segment Management*, and *Synchronizer*. It then transfers these messages to the host memory by PCIe-DMA and generates IRQs to inform the host CPU for corresponding processing. Its architecture is shown in Fig. 14. The messages from each

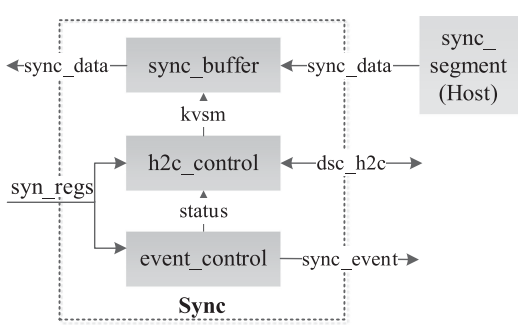


Fig. 13. The hardware architecture of *Synchronizer*.

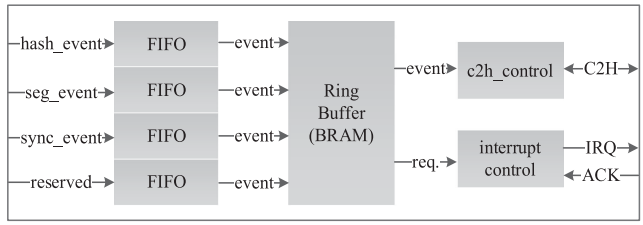


Fig. 14. The architecture of *Event Controller* that collects event messages and informs the host CPU by interrupts.

source are first cached in the FIFOs and then merged in a ring buffer. Every once in a while, DMA descriptors will be generated if there is any message collected. The messages are then transmitted to the *sync_segment*.

Due to the long latency of interrupt processing, we design a batch scheme to improve efficiency. Two registers are used to record the start and end addresses of the cumulative messages in the *sync_segment* during one interrupt period. If the start and end addresses are different, an IRQ will be generated. After the host CPU processes the messages, the registers will be updated and compared to decide the generating of a new IRQ.

5 Theoretical Performance

FULL-KV is based on a CPU-FPGA heterogeneous architecture, as shown in Fig. 3, which can be divided into two main modules: NOE and KVSE. In such a system, there are three data-paths that may determine the theoretical performance of the system, namely network communication, onboard DDR4 access, and PCIe transmission. Since the size of the hash table entry is much smaller than that of the KVs, the requirement for the DDR4 bandwidth is much lower than those for the network and PCIe bandwidth. In general, the PCIe bandwidth is higher than the network bandwidth in such systems. Therefore, we examine the theoretical performance based on network bandwidth.

Suppose the network bandwidth can be fully utilized, we can calculate the theoretical maximum throughputs and minimum latencies of NOE, KVSE, and FULL-KV based on the following factors.

1. Network transmission bandwidth B, which consists of the running frequency f and the data width w.

$$B = fw. \tag{1}$$

2. KVSM length L, where the value length l is variable and the sum length of the header and the key l_0 is fixed. If a KVSM contains the value data, then

$$L = l_0 + l. \tag{2}$$

Otherwise,

$$L = l_0. \tag{3}$$

3. The success rate of KVS operations α, which only affects the response KVSM length for the GET operation. If it is a success, the length is $l_0 + l$, otherwise l_0. Therefore, the average length is

$$\alpha(l_0 + l) + (1 - \alpha) = l_0 + \alpha l. \tag{4}$$

5.1 Theoretical Maximum Throughput

The theoretical throughput V can be calculated according to the basic function (5).

$$V = \frac{B}{L} \ . \tag{5}$$

For NOE, the theoretical maximum throughput can be calculated as follows. Given the running frequency f_n, the data width w_n, the throughputs V_n for the PUT and GET operations are shown as (6) and (7), respectively.

$$V_n = \frac{f_n w_n}{l_0 + l} \tag{6}$$

$$V_n = \frac{f_n w_n}{l_0 + \alpha l} \ . \tag{7}$$

Similarly, the theoretical maximum throughput of KVSE can be calculated as follows. Given the running frequency f_k, the data width w_k, the throughput V_k for the PUT operation is

$$V_k = \frac{f_k w_k}{l_0 + l} \ , \tag{8}$$

whereas the throughput for the GET operation is

$$V_k = \frac{f_k w_k}{l_0 + \alpha l} \ . \tag{9}$$

The theoretical maximum throughput of FULL-KV is the minimum throughput of NOE and KVSE. For the PUT operation, the system throughput is

$$V_s = \frac{min\{f_k w_k, f_n w_n\}}{l_0 + l} \ . \tag{10}$$

For the GET operation, the system throughput is

$$V_s = \frac{min\{f_k w_k, f_n w_n\}}{l_0 + \alpha l} \ . \tag{11}$$

5.2 Theoretical Minimum Latency

The theoretical latency can be divided into two parts: the constant processing time and the data transmission time in the network. Since the transmission of the failed KVS operation costs less time, we only calculate the latencies of the successful KVS operations.

For NOE, the theoretical minimum latency can be calculated as follows. Given the running frequency f_n, the data width w_n, and the constant processing time T_{np0} for the PUT operation and T_{ng0} for the GET operation, the latencies T_n for each operation are shown as (12) and (13), respectively.

$$T_n = T_{np0} \ + \frac{l_0 + l}{f_n w_n} \tag{12}$$

$$T_n = T_{ng0} \ + \frac{l_0 + l}{f_n w_n}. \tag{13}$$

In KVSE, the processing of the KVS operations and the value transmission are in two different data paths. For the PUT operation, the request KVSM determines the latency; whereas for the GET operation, the response KVSM determines the latency. Therefore, the KVS processing and the value transmission can be carried out in parallel for the PUT operation; whereas for the GET operation, the value transmission happens after the KVS processing.

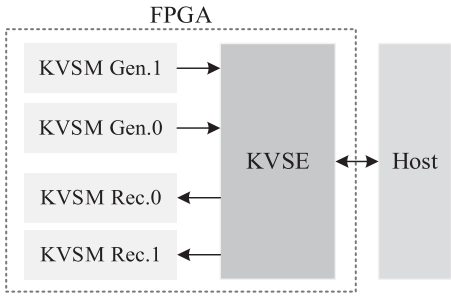


Fig. 15. Test system architecture for KVSE, with two KVSM generators and receivers.

The theoretical minimum latency can be calculated as follows. Given the running frequency f_k, the data width w_k, the holding time due to the KVS processing T_{kh0}, the constant processing time T_{kp0} for the PUT operation and T_{kg0} for the GET operation, the latencies T_k for each operation are shown as (14) and (15), respectively.

$$T_k = max\left\{T_{kh0}, T_{kp0} + \frac{l_0 + l}{f_k w_k}\right\}, \tag{14}$$

$$T_k \ = \ T_{kg0} \ + \frac{l_0 + l}{f_k w_k}. \tag{15}$$

The theoretical minimum latency of the system is related to the maximum latencies of NOE and KVSE. The constant processing time of the system is longer than those of KVSE and NOE. Given the holding time T_{sh0}, the constant processing time T_{sp0} for the PUT operation and T_{sg0} for the GET operation, the system latencies T_s for each operation are shown as (16) and (17), respectively.

$$T_s = \ max\left\{T_{sh0}, T_{sp0} + \frac{l_0 + l}{min\{f_k w_k, f_n w_n\}}\right\}, \tag{16}$$

$$T_s = T_{sg0} \ + \frac{l_0 + l}{min\{f_k w_k, f_n w_n\}}. \tag{17}$$

6 Evaluation of KVSE

6.1 Experimental Setup

FULL-KV is based on a CPU-FPGA heterogeneous architecture, where KVSE and NOE are implemented on FPGA. The FPGA platform is a Xilinx Kintex UltraScale FPGA KCU105 board, which is equipped with an onboard 2GiB DDR4 memory, two 10G SFP+ interfaces, and a PCIe gen3 x8 interface. Through PCIe, the FPGA board is connected to a host computer, Dell OptiPlex 7050, which has an Intel i7-7700 eight-core CPU, a set of DDR4 memory with a total capacity of 23.3GiB, and a 256GB SSD. The operating system in the host is CentOS 6.10.

In FULL-KV, the onboard DDR4 memory is used to store the hash table, running at 300MHz. Values are stored in the host DDR4 memory and transmitted from or to KVSE via PCIe-DMA. UDP protocol is adopted in NOE based on the

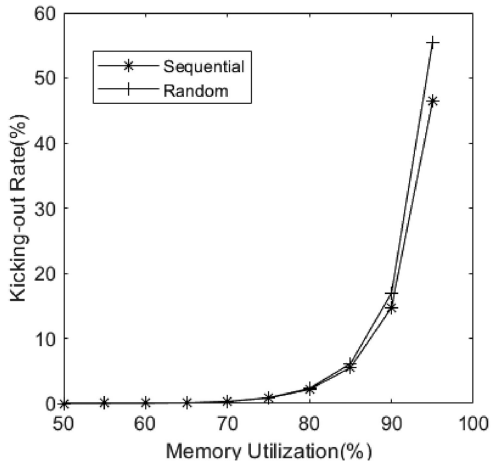


Fig. 16. The relations between kicking-out rate and the memory utilization in the hash table, including two cases: sequential and random kicking out.

10GbE. Except for the hash table, KVSE runs at 250MHz, whereas NOE at 156.25MHz.

To evaluate the performance of KVSE, we remove the NOEs in FULL-KV and use two KVSM generators/receivers connecting directly with KVSE, as shown in Fig. 15. So the test system is just implemented in an FPGA card plus a computer.

6.2 Hash Table Evaluation

As described in Section 4.2, an old hash table entry may be kicked out and need to be reinserted into the hash table later when inserting a new entry. The kicking-out rate is mainly related to memory utilization. Therefore, we use software building the same model as the hardware implementation to simulate the relationship between the above two factors.

The simulated memory size is 2GiB, as large as the onboard DDR4 memory in the FPGA platform. Two same hash functions are used to generate two hash addresses, each corresponding to four buckets. Since the bucket size is 128 bits in hardware, the total bucket number is 2^27. We use arrays in C language as buckets, so the hash addresses are mapped to array addresses. The 64-bit keys are generated by the random function in the C standard library. When a key is kicked out from the buckets, it will be reinserted later. After the memory utilization reaches a certain threshold, the kicking-out rate will be calculated according to the total kicking-out number and the total inserting number.

We test two kicking-out schemes: sequentially as illustrated in Fig. 8 and randomly. The results are shown in Fig. 16. When the memory utilization is less than 65 percent, the kicking-out rates approximately equal 0. With memory utilization rises, the rates grow faster and faster, and the latter one exceeds the first one. This indicates that the sequentially kicking-out scheme is a better solution. At the points of 90 and 95 percent about the memory utilization, the smaller rates are 14.7 and 46.4 percent, respectively.

6.3 Performance of KVSE

Based on the system setup, we test the throughputs and latencies of KVSE for the PUT and GET operations under different value sizes, from 1 byte to 4M bytes, as shown in Figs. 17a and 17b. We also plot the performance of KVSE in our previous conference-version work [29] for comparison. To show the performance improvement more visually, we calculate the ratios of throughputs and latencies between this paper and our previous work, and plot them in Fig. 17c.

In this paper, the peak throughput of KVSE can reach 31.6Mops/34.4Mops for the PUT/GET operations, whereas the peak throughput is 4.4Mops/27.6Mops in our previous work. The peak throughput is promoted by 7.2x/1.2x for the PUT/GET operations. There are two main reasons for the high promotion of the PUT throughput. First, also most importantly, we design a KVS request collision detector in *Hash Engine*. The pipeline stalling rate decreases from always to rarely, so much higher throughput can be yielded. Second, the KVSM receiving and transmitting ports are both increased to two. The request/response KVSMs in the two channels can be parsed/packed simultaneously. But the requests can only be submitted to *Hash Engine* alternatively. Therefore, its contribution to peak throughput improvement is less than 2x. This is also the reason for the peak throughput increase for the GET operations. It is also the only reason because the GET request does not stall the pipeline of *Hash Engine*. With the increase of the value size, the transmission for the KVSMs becomes the dominant factor in place of the

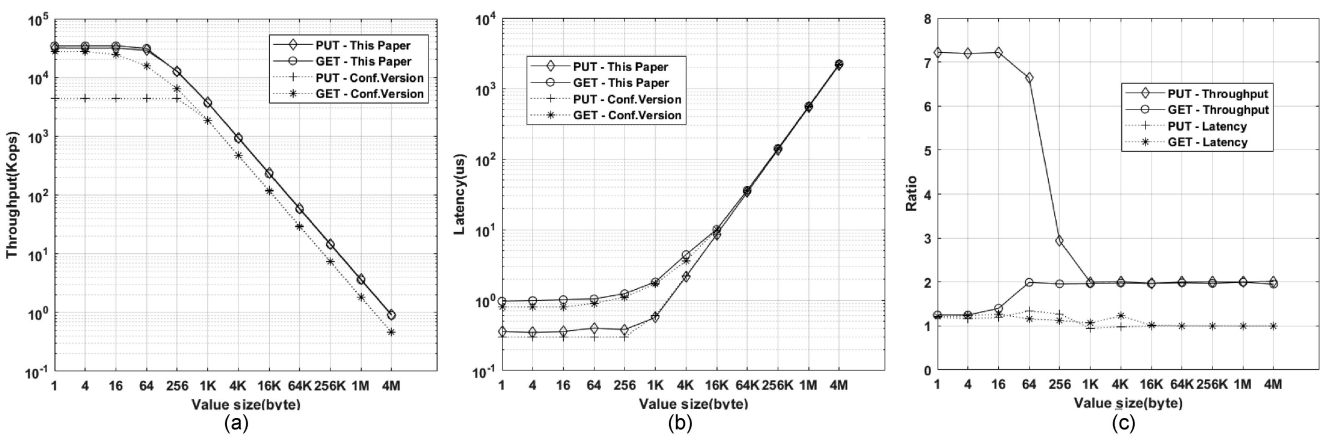


Fig. 17. Throughput and latency of KVSE under different value size in this paper and our previous conference-version paper [29]. (a) KVSE throughput for the PUT and GET operations in both works. (b) KVSE latency for the PUT and GET operations in both works. (c) The ratios of throughputs and latencies between this paper and our previous work for the PUT and GET operations.

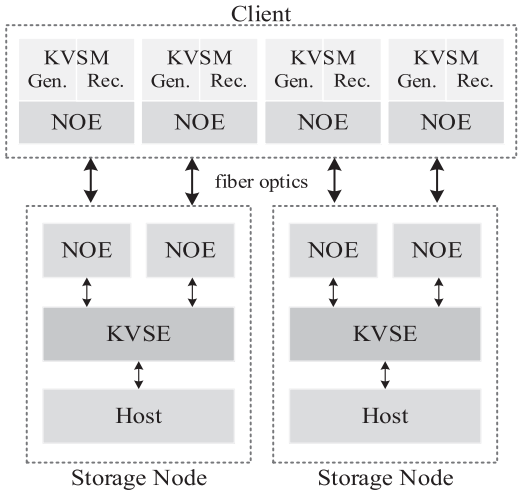


Fig. 18. Test system architecture for single-node FULL-KV.

processing of *Hash Engine*. Therefore, the throughputs for both PUT and GET operations become directly proportional to the value size, and the ratios in Fig. 17c approach 2.

The lowest latency of KVSE is $0.4\mu s/1.0\mu s$ for the PUT/GET operation in this paper, slightly larger than that in our previous work, which is $0.3\mu s/0.8\mu s$. There are two main reasons. First, the implementations of the collision detector and some other logic consume more processing time. Second, the average latency of the two KVSM channels is larger than one because *Hash Engine* has only one request submitting port. When the value size increases, *Hash Engine* becomes the less dominant factor, the ratios in Fig. 17c thus approach 1.

7 Evaluation of Single-Node FULL-KV

7.1 Experimental Setup

To evaluate the single-node performance of FULL-KV, we deploy a test system with one client and two storage nodes, as shown in Fig. 18. Each storage node contains two NOEs with a total Ethernet bandwidth of 20Gbps. The client contains four NOEs, four KVSM generators, and four KVSM receivers, the same as the client in Fig. 5. The four NOEs in the two storage nodes are directly connected with the four NOEs in the client via a fiber optic cable. Since the four sets of NOE, KVSM generator and KVSM receiver are totally independent, we calculate the average performance of the two storage nodes as the single-node performance of FULL-KV.

We have found the system bottleneck was the 10GbE in our previous work [29]. After making several extensions, especially broadening the Ethernet bandwidth in this paper, we need to identify if the Ethernet bandwidth is still the system bottleneck. By comparing the performances of the single-node FULL-KV and KVSE, we can get the answer.

Due to the functional simplicity of the client, we select a different set of hardware configurations for it. The FPGA platform is Xilinx Virtex-7 FPGA VC709 board, which is equipped with four 10G SFP+ interfaces and a PCIe gen3 x8 interface. The host computer is Dell OptiPlex990, containing an Intel i5-2400 quad-core CPU, a set of 15.5 GiB DDR3 memory, and a 256GB HDD. The operation system is Centos 6.8.

7.2 Throughput

We test the throughputs of the single-node FULL-KV for the PUT and GET operations under different value sizes, from one byte to 4M bytes. For comparison, we plot them together with the throughputs of KVSE and our previous work [29] in Fig. 19.

The experimental peak throughput of the single-node FULL-KV for the PUT/GET operations can reach 26.0Mops/25.9Mops, 6.0x/1.9x larger than that of our previous work, which is 4.3Mops/13.6Mops. The reasons for the improvement are similar to that for KVSE, as described in Section 6.3. Since the bandwidth of two NOEs is smaller than that of KVSE, the impact of *Hash Engine* on throughputs is reduced. Therefore, the improvement decreases for the PUT operations and increases for the GET operations. With the value size rises, the improvement approaches 2x. Under all the value sizes, both the PUT and GET throughputs of KVSE exceeds those of the single-node FULL-KV. This indicates that the Ethernet bandwidth still limits the data transmission and becomes the system bottleneck although the bandwidth is doubled.

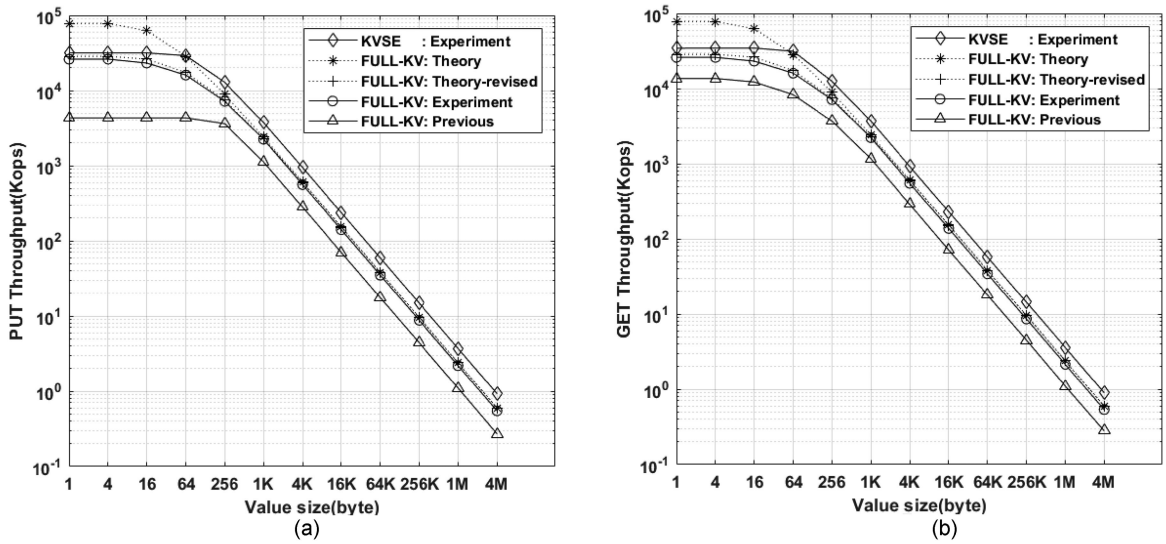


Fig. 19. Theoretical and experimental throughputs for the PUT (a) and GET (b) operations under different value sizes, including experimental throughputs of KVSE, single-node FULL-KV, and our previous work [29], theoretical and revised theoretical throughputs of single-node FULL-KV.

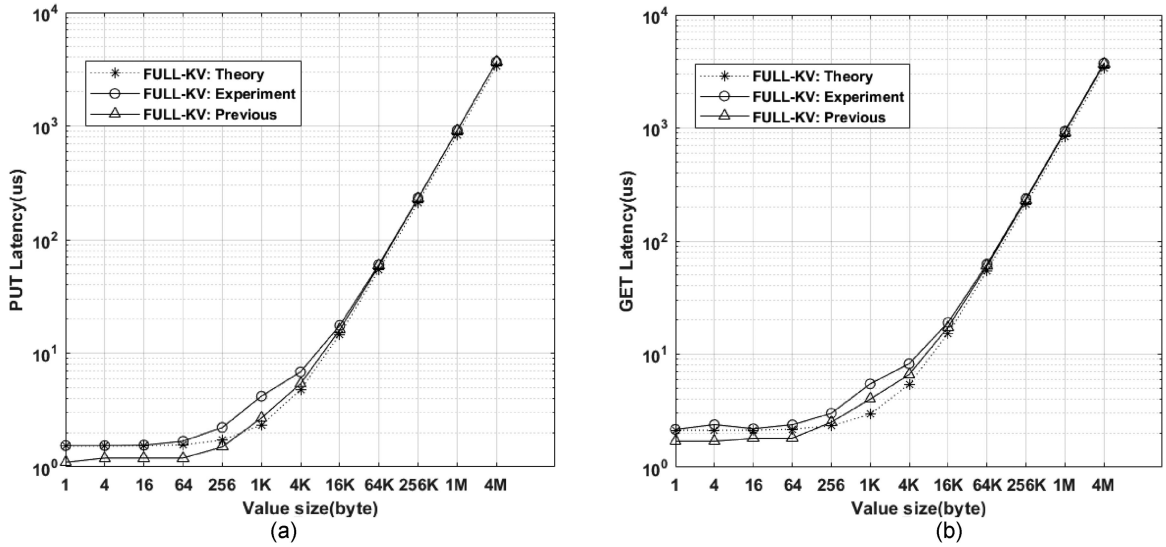


Fig. 20. Theoretical and experimental latencies for the PUT (a) and GET (b) operations under different value sizes, including theoretical and experimental latencies of the single-node FULL-KV, and experimental latencies of our previous work [29].

We also calculate the theoretical maximum throughputs of the single-node FULL-KV using equations (8), (9), (10), and (11), where the parameter values are as follows.

$$
\begin{aligned}
&w_n = w_k = 2{*}64\text{bits}\\
&f_n = 156.25\text{MHz}\\
&f_k = 250\text{MHz}\\
&l_0 = 192\text{bits}\\
&\alpha = 1.
\end{aligned}
$$

Considering the practical packet transmission of Ethernet, IP/MAC headers will be added to the packets and there is a time interval between packets. So we revise the theoretical parameter value of l_0, from 192 bits to 640 bits, and get another group of theoretical throughputs.

The theoretical and revised theoretical throughputs for the PUT and GET operations are shown in Figs. 19a and 19b. We can see that the experimental and revised theoretical throughputs are almost the same for both operations under all the value sizes. This manifests that FULL-KV can achieve fully-pipelined performance and make full use of the bandwidth of two 10GbEs.

Compared with DRAM-only BlueCache [5], which also provides support for large-size values, the peak throughput of the single-node FULL-KV is 6.3x/6.5x higher for the PUT/GET operations. Moreover, the supported value size of FULL-KV is 4x larger, up to 4M bytes.

7.3 Latency

The experimental latencies of the single-node FULL-KV and our previous work under different value sizes are shown in Figs. 20a and 20b. The lowest latency of FULL-KV is 1.5μs/2.2μs for the PUT/GET operation, slightly larger than that of our previous work. This results from two main reasons: the increase of the logic complexity and the serial input port for requests in *Hash Engine*, as introduced in Section 6.3. With the value size increases, the transmission of KVSMs becomes the dominant factor, so the latencies in this paper and our previous work are almost the same.

We also calculate the theoretical minimum latencies of FULL-KV using equations (14), (15), (16), and (17), where the parameter values are:

$$
\begin{aligned}
&w_n = w_k = 64\text{bits}\\
&f_n = 156.25\text{MHz}\\
&f_k = 250\text{MHz}\\
&l_0 = 192\text{bits}\\
&T_{sh0} = 1.5\mu\text{s}\\
&T_{sp0} = 1.5\mu s\\
&T_{sg0} = 2.1\mu s.
\end{aligned}
$$

The theoretical latencies for the PUT and GET operations are also plotted in Figs. 20a and 20b, which show that the experimental and the theoretical minimum latencies are almost the same for both the PUT and GET operations. This manifests again that the Ethernet bandwidth can be fully utilized and the fragmentation processing of large-size values has high efficiency.

Compared with the state-of-the-art KVS systems, as shown in Table 1, FULL-KV is 3.0x/1.5x faster for the PUT/GET operation.

TABLE 1
Comparison of FULL-KV With Other KVS Systems Under the Workload of 12-Byte KVPs

KVS	Comment	Latency(μs)	
		GET	PUT
Pilaf [15]	One-sided RDMA Reads	12	15
FaRM [14]	One-sided RDMA for GET	4.5	~10
DrTM-KV [16]	One-sided RDMA and HTM	**3.4**	6.3
HERD'16 [18]	Two-sided RDMA, 12cores	5	5
Mega-KV [39]	GPU	280	280
KV-Direct [20]	Programmable NIC	4.3	5.4
Caribou [27]	DRAM-only	~30	~26
BlueCache [5]	DRAM-only or Flash	20	20
Xilinx'13 [25]	FPGA	3.5	**4.5**
FULL-KV	CPU-FPGA	**2.2**	**1.5**
Speedup	–	**1.5x**	**3.0x**

TABLE 2
FPGA Resource Utilization of Single FULL-KV Storage Node

Module name	LUT	LUTRAM	FF	BRAM
KVSE	18755	1216	32139	166
DDR4 Controller	16852	2075	21597	26.5
DMA Controller	34457	3827	32351	63
NOE	13802	292	17540	25
System total	100619 (42%)	12660 (11%)	124170 (26%)	368.5 (61%)

7.4 FPGA Resource Utilization

The FPGA Resource utilization of a single FULL-KV storage node on a KCU105 board is shown in TABLE 2. Except that the BRAM utilization reaches 61 percent, LUT, LUTRAM, and Flip-Flop utilizations are all less than one half, where LUTRAM no more than one eighth. Since both 10GbEs onboard are used and their total bandwidth is still the performance bottleneck, there is no need to increase resource utilization to improve the performance. To get a balance between the system performance and resource utilization, it is better to customize an FPGA board that contains matched Ethernet bandwidth, DDR4 bandwidth, and PCIe bandwidth. Considering the storage scale, the capacities of the onboard DDR4 memory and the host memory need to scale with the workloads.

7.5 CPU Utilization

Fig. 21 shows the host CPU and memory utilization history when putting about 2048 KVs where every value size is 4MiB. During the system initialization, the host CPU needs to pre-allocate 256 segments of 4MiB, so there is a step in Fig. 21. The memory utilization rises from 2.0 to 6.4 percent of the 23.3GiB. FULL-KV then begins to receive a large number of KVS requests. About 4096 KVs of 4MiB size are inserted into the host memory. The memory utilization increases from about 6.4 to 74.4 percent.

Most of the time, the workload of the eight-core CPU keeps less than 5 percent. Only two small peaks occur, but less than 15 percent. This manifests that the host CPU can get a great release from the heavy workload, and more hardware accelerators can be added in a host server to achieve higher performance.

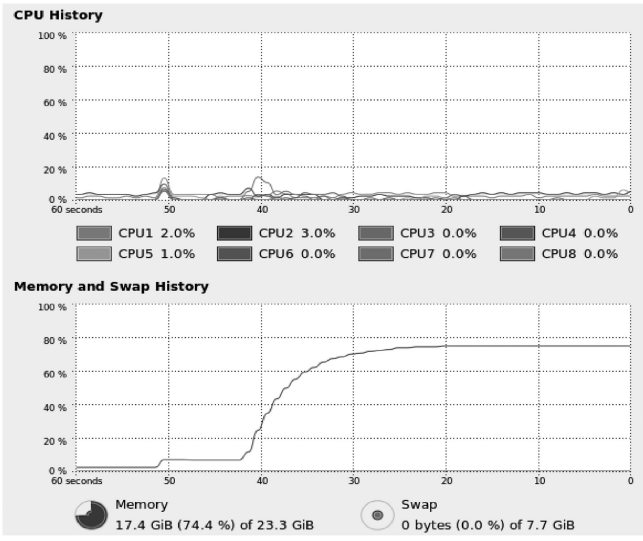


Fig. 21. Host CPU and memory utilization history when putting about 4096 KVs of 4MiB size.

TABLE 3
Comparison of FULL-KV With Other KVS Systems in Power Efficiency

KVS	Power Efficiency(Kops/W)	
	GET	PUT
Memcached [2]	~5	~5
RAMCloud [7]	~14	~14
FaRM [14]	~30	~15
HERD'16 [18]	~490	~300
Xilinx'13 [25]	106	106
KV-Direct [20]	1487	942
FULL-KV	**648**	**650**

7.6 Power Efficiency

When a FULL-KV storage node works at peak throughput, the system power consumption is 40W. The host computer consumes 16W when the FPGA board is turned off. Therefore, the power consumption of the hardware accelerator is 24W. In terms of KVS operations per watt, we can calculate the power efficiency is 648 Kops/W for the GET operations and 650 Kops/W for the PUT operations.

Compared with the state-of-the-art KVS systems in Table 3, FULL-KV is at least 20x more power-efficient than the software-based KVS systems. The power efficiency of the RDMA-based and FPGA-based KVS systems varies greatly because it depends on the system configurations such as CPU core number, NIC number, and NIC bandwidth. Compared with Xilinx'13 [25], which was based on 10GbE, FULL-KV is 6x more power-efficient. Since KV-Direct [20] used 40GbE and aimed to improve system throughput for small-size KVs, its throughput was much higher and thus power efficiency was 2.3x/1.4x higher than ours. FULL-KV has some performance loss for small-size KVs in order to provide support for super-wide ranges of value size.

8 Evaluation of Multiple-Node FULL-KV

8.1 Experimental Setup

The architecture of the test system for multi-node FULL-KV is shown as Fig. 5. We use one client, one commercial Ethernet switch, and two storage nodes for the test. The configurations for the client and the storage nodes are the same as introduced in Section 7.1. The Ethernet switch is H3C LS-6520-24S-SI, equipped with 24 SFP+ interfaces of 10Gbps.

8.2 System Performance

Fig. 22a shows the throughputs of the two-node FULL-KV with an Ethernet switch for the GET and PUT operations. The peak throughput can reach 51.9Mops/51.6Mops for the PUT/GET operations, 12.1x/3.8x larger than our previous work [29]. When evaluating the single-node performance of FULL-KV, we test two storage nodes and calculate the average throughput to get more accurate results. The total performance is used here to compare with the test system with

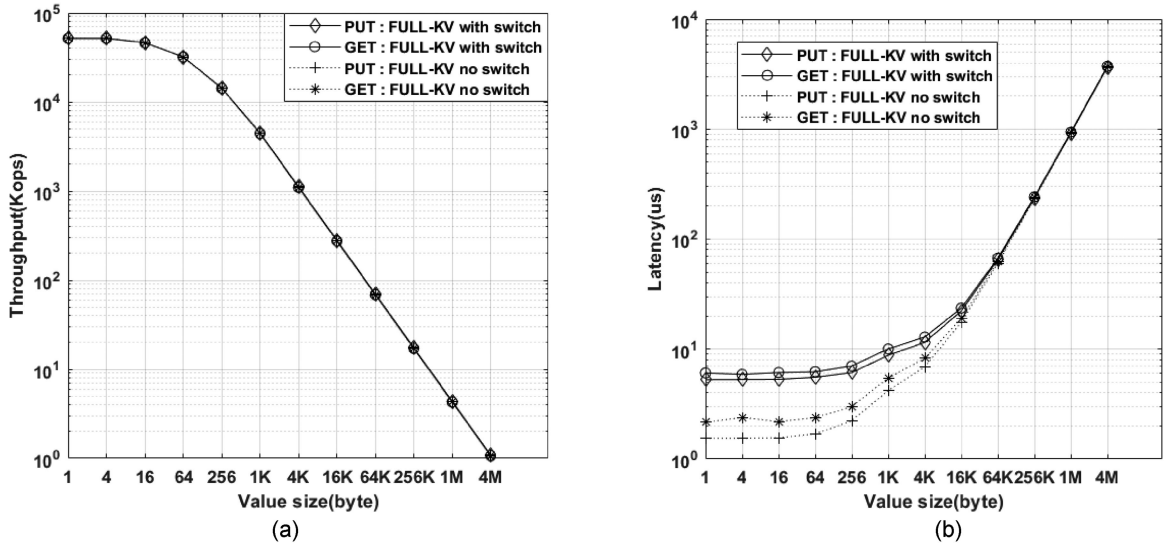


Fig. 22. The system throughputs (a) and latencies (b) of two-node FULL-KV with and without Ethernet switch for the PUT/GET operations.

a switch. We can see that the throughputs of both systems for the PUT/GET operations are almost the same. This manifests that FULL-KV is totally compatible with current industrial networking communication and has linear scalability with the storage nodes increase.

The system latencies of both systems for the GET and PUT operations are shown in Fig. 22b. Adding the Ethernet switch, the lowest system latency increases from $1.5\mu s/2.2\mu s$ to $5.3\mu s/6.0\mu s$ for the PUT/GET operation. The switch brings a latency of $3.8\mu s$. With the value size rises, the network transmission of the KVSMs becomes the dominated factor, the impact of the switch is greatly reduced.

9 Conclusion

In this paper, FULL-KV, a flexible and ultra-low-latency IMKVS system is presented based on a CPU-FPGA heterogeneous architecture. First, due to the design of a highly parallel accelerator architecture, the system can achieve ultra-low-latency performance. Second, we adopt a decoupling design scheme that stores the hash table in the onboard DDR4 memory and values in the host memory, making the storage capacity of the KVS system highly scalable. Third, through the design and implementation of a novel fragmentation mechanism on hardware, FULL-KV can support very large values, and thus the flexibility is substantially broadened. Fourth, for a single KVS operation, such as PUT, at most one DMA data transmission is needed, which can significantly improve the processing efficiency. Fifth, by increasing the number of Ethernet interfaces and extending the system architecture, FULL-KV attains a doubled throughput while keeping the latency almost the same. Finally, we introduce the commercial Ethernet switch and build a multi-node test system, which verifies the system practicability and scalability. Since we aim to improve the performance and power consumption of the IMKVS system, the data backup and failure recovery have not been well considered in this paper. In the future, more effort can be paid to improve system reliability.

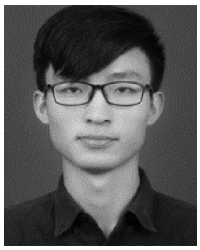
Acknowledgments

This work was supported in part by the National Science Foundation of China under Grant No. 61971143.

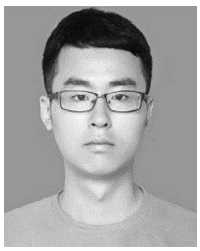
References

[1] G. DeCandia *et al.*, "Dynamo: amazon's highly available key-value store," in *Proc. ACM SIGOPS Operating Syst. Rev*, 2007, pp. 205–220.
[2] Memcached. 2019. [Online]. Available: https://www.memcached.org
[3] B. Debnath, S. Sengupta, and J. Li. "FlashStore: high throughput persistent key-value store," *Proc. VLDB Endowment*, vol. 3, no. 1-2, pp. 1414–1425, 2010.
[4] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "SILT: A memory-efficient, high-performance key-value store," in *Proc. 23rd ACM Symp. Operating Syst. Princ.*, 2011, pp. 1–13.
[5] S. Xu, S. Lee, S. Jun, M. Liu, J. Hicks, and Arvind, "BlueCache: A scalable distributed flash-based key-value store," *Proc. VLDB Endowment*, vol. 10, no. 4, pp. 301–312, 2016.
[6] Redis. 2019. [Online]. Available: https://redis.io/topics/data-types-intro
[7] J. Ousterhout *et al.*, "The RAMCloud storage system," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, 2015, Art. no. 7.
[8] Y. Zhang, C. Guo, D. Li, R. Chu, H. Wu, and Y. Xiong, "Cubicring: Enabling one-hop failure detection and recovery for distributed in-memory storage systems," in *Proc. 12th USENIX Symp. Networked Syst. Des. Implementation*, 2015, pp. 529–542.
[9] J. Ousterhout *et al.*, "The case for RAMClouds: Scalable high-performance storage entirely in DRAM," *ACM SIGOPS Operating Syst. Rev.*, vol. 43, no. 4, pp. 92–105, 2010.
[10] I. Marinos, R. N. Watson, and M. Handley, "Network stack specialization for performance." *Proc. ACM SIGCOMM Comput. Commun. Rev.*, 2014, pp. 175–186.
[11] E. Jeong *et al.*, "mTCP: A highly scalable user-level {TCP} stack for multicore systems," *Proc. 11th USENIX Symp. Netw. Syst. Des. Implementation*, 2014, pp. 489–502.
[12] DPDK. 2019. [Online]. Available: https://www.dpdk.org
[13] S. Li *et al.*, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform," *Proc. ACM SIGARCH Comput. Architecture News*, 2015, pp. 476–488.
[14] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," *Proc. 11th USENIX Symp. Netw. Syst. Des. Implementation*, 2014, pp. 401–414.
[15] C. Mitchell, Y. Geng, and J. Li, "Using one-sided RDMA reads to build a fast, CPU-efficient key-value store," in *Proc. Presented Part USENIX Annu. Techn. Conf.*, 2013, pp. 103–114.
[16] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using RDMA and HTM," *Proc. 25th Symp. Operating Syst. Princ.*, 2015, pp. 87–104.

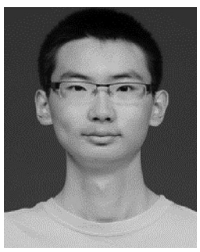
[17] B. Cassell, T. Szepesi, B. Wong, T. Brecht, J. Ma, and X. Liu, "Nessie: A decoupled, client-driven key-value store using RDMA." *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 12, pp. 3537–3552, Dec. 2017.
[18] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 295–306, 2015.
[19] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance RDMA systems," in *Proc. USENIX Annu. Tech. Conf.*, 2016, pp. 437–450.
[20] B. Li *et al.*, "Kv-direct: High-performance in-memory key-value store with programmable nic," in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 137–152.
[21] W. Liang, W. Yin, P. Kang, and L. Wang, "Memory efficient and high performance key-value store on FPGA using Cuckoo hashing," in *Proc. 26th Int. Conf. Field Programmable Logic Appl.*, 2016, pp. 1–4.
[22] Z. István, G. Alonso, M. Blott, and K. Vissers, "A hash table for line-rate data processing," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 2, 2015, Art. no. 13.
[23] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," *Proc. ACM SIGMETRICS Perform. Eval. Rev.*, 2012, pp. 53–64.
[24] D. Didona and W. Zwaenepoel, "Size-aware sharding for improving tail latencies in in-memory key-value stores," in *Proc. Netw. Syst. Des. Implementation*, 2019, pp. 79–94.
[25] M. Blott, K. Karras, L. Liu, K. Vissers, J. Bär, and Z. István, "Achieving 10Gbps line-rate key-value stores with FPGAs," in *Proc. 5th USENIX Workshop Hot Topics Cloud Comput.*, 2013.
[26] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala, "An FPGA memcached appliance," *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, 2013, pp. 245–254.
[27] Z. István, D. Sidler, and G. Alonso, "Caribou: Intelligent distributed storage," *Proc. VLDB Endowment*, vol. 10, no. 11, 2017, pp. 1202–1213.
[28] J. W. Lockwood and M. Monga, "Implementing ultra low latency data center services with programmable logic," in *Proc. IEEE 23rd Annu. Symp. High-Perform. Interconnects*, 2015, pp. 68–77.
[29] Y. Qiu, H. Lv, J. Xie, W. Yin, and L. Wang, "Ultra-low-latency and flexible in-memory key-value store system design on CPU-FPGA," *Proc. Int. Conf. Field-Programmable Technol.*, 2018, pp. 145–152.
[30] R. Pagh and F. F. Rodler, "Cuckoo hashing", *J. Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.
[31] M. Herlihy, N. Shavit, and M. Tzafrir, "Hopscotch hashing," in *Proc. Int. Symp. Distrib. Comput.*, 2008, pp. 350–364.
[32] M. Dvorak and J. Korenek, "Low latency book handling in FPGA for high frequency trading," in *Proc. 17th Int. Symp. Des. Diagnostics Electron. Circuits Syst.*, 2014, pp. 175–178.
[33] G. Swart, "Spreading the load using consistent hashing: a preliminary report," in *Proc. Int. Symp. Parallel Distrib. Comput.*, 2004, pp. 169–176.
[34] D. R. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. S. Levine, and D. M. Lewin., "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web," *Proc. Symp. Theory Comput.*, 1997, pp. 654–663.
[35] W. Xie and Y. Chen, "Elastic consistent hashing for distributed storage systems," *Proc. Int. Parallel Distrib. Process. Symp.*, 2017, pp. 876–885.
[36] Sheepdog. 2019. [Online]. Available: https://github.com/sheepdog/sheepdog
[37] Twitter's TwemProxy. 2019. [Online]. Available: "https://github.com/twitter/twemproxy"
[38] B. Jenkins, "hashword2 functions for Hash table lookup," 2006. [Online]. Available: http://burtleburtle.net/bob/c/lookup3.c
[39] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang, "Mega-KV: A case for GPUs to maximize the throughput of in-memory key-value stores," *Proc. VLDB Endowment*, vol. 8, no. 11, pp. 1226—1237, 2015.



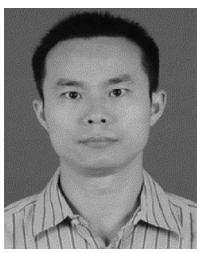
Yunhui Qiu received the BE degree in microelectronic science and engineering from Fudan University, in 2018, where he is working toward the MS degree. His current research interests include the acceleration system design of the CPU-FPGA heterogeneous architecture, distributed storage systems, and nonvolatile memory controller.



Jinyu Xie received the BE degree in microelectronic science and engineering from Fudan University, in 2018, where he is currently working toward the MS degree. His current research interests include network acceleration, high-performance computing, and distributed systems.

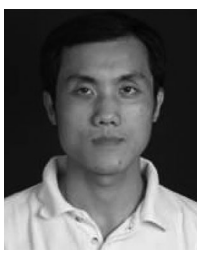


Hankun Lv received the BE degree in microelectronic science and engineering from Fudan University, China, in 2018, He is currently working toward the master's degree in the State Key Laboratory of ASIC and System, Fudan University. His research interests include digital circuit design, CPU-FPGA heterogeneous architecture design, and key-value storage.



Wenbo Yin is working as an engineer with the School of Microelectronics, Fudan University. His current research interests include key-value store, embedded system design, and industrial automation.

Wai-Shing Luk received the BS degree in electronics from the Chinese University of Hong Kong, in 1988, the MPhil and PhD degrees in computer science and engineering from the same university, in 1993 and 1996, respectively. In 1997, he was awarded a postdoctoral fellowship at Katholieke Universiteit Leuven in Belgium. He worked as a senior R&D engineer at Synopsys in the USA from 1999. He joined Fudan University as an associate professor in 2004. His current research interest includes clock skew scheduling, statistical optimization, and algorithms in VLSI physical design.



Lingli Wang (Member, IEEE) received the MS degree in electrical engineering from Zhejiang University, Hangzhou, China, in 1997, and the PhD degree in electrical engineering from Edinburgh Napier University, Edinburgh, UK, in 2001. He was with the Altera European Technology Center, High Wycombe, Buckinghamshire, UK, for four years. In 2005, he joined Fudan University, Shanghai, China, where he is currently a full professor with the State Key Laboratory of ASIC and System, School of Microelectronics. His current research interests include logic synthesis, reconfigurable computing, and quantum computing.

Bowei Yu received the bachelor's degree from Chongqing University, in July 2002. He joined Huawei Technologies Company Ltd, in 2005. Currently, he is engaged in the storage network. His research interests are future networking, especially memory fabric, and networking acceleration.

Hua Chen received the bachelor's degree in computer science and technology from Wuhan Engineering University, in 2004. He joined Huawei Technologies Company Ltd, in 2006. He has an in-depth study of storage hardware systems. Currently, he is responsible for the research and design of storage systems, and is an important participant in the Huawei-developed storage systems. His research interest includes co-design of software and hardware.

Xianjun Ge received the bachelor's degree in the Department of Computing from Hehai University, Nanjing, China, in July 2005. He joined Huawei Technologies Company Ltd, in 2008. He is engaged in TCP/IP and RDMA networking technology and responsible for the design of the storage network infrastructure architecture. He is interested in areas including network function virtualization (NFV), network congestion control, and quality of experience (QoE) measurement. He has secured six patents as the first author.

Zhijian Liao received the master's degree in computer engineering from the School of Information Science and Technology, Southwest Jiaotong University, Chengdu, China, in 2008. He is currently a senior software engineer at Huawei Technologies Co. Ltd. His research interest include networking communication, especially data-center networking.

Xiaozhong Shi received the bachelor's degree in electronic information engineering from the University of Electronic Science and Technology. He is currently the chief engineer of Huawei Technologies Company Ltd. He has more than 16 years of product development and technical research experience in ICT fields such as the MSTP, OTN, and DWDM areas. Now he is responsible for technical planning in the field of IT storage. During his technical research and development period, he has published 11 conference papers and secured nine invention patents.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Analyzing the Performance Trade-Off in Implementing User-Level Threads

Shintaro Iwasaki, *Student Member, IEEE*, Abdelhalim Amer, *Member, IEEE*, Kenjiro Taura, *Member, IEEE*, and Pavan Balaji, *Member, IEEE*

Abstract—User-level threads have been widely adopted as a means of achieving lightweight concurrent execution without the costs of OS-level threads. Nevertheless, the costs of managing user-level threads represent a performance barrier that dictates how fine grained the concurrency exposed by an application can be without incurring significant overheads; this in turn may translate into insufficient parallelism to exploit highly parallel systems. This article is a deep dive into the fundamental costs in implementing user-level threads. We first identify that one of the highest sources of fork-join overheads stems from *deviations*, events that incur context switching during the execution of a thread and disrupt a run-to-completion execution. We then conduct an in-depth investigation of a wide spectrum of methods with respect to how they handle deviations while covering both parent- and child-first scheduling policies. Our methodology involves a comprehensive instruction- and cache-level analysis of all methods on several modern CPU architectures. The primary finding of our evaluation is that dynamic promotion methods that assume the absence of deviation and dynamically provide context-switching support offer the best trade-off between performance and capability when the likelihood of deviation is low.

Index Terms—Multithreading, multitasking, scheduling, user-level threads, context switch, task parallelism

1 Introduction

Multithreading is the predominant form of parallelism to exploit modern highly parallel multicore and many-core processors. On self-bootable systems, such as traditional servers and the second generation Intel MIC accelerators, the majority of programming systems map their threading abstractions to OS-level threads (e.g., most systems target the POSIX Threads (Pthreads) specification, which itself maps Pthreads to OS-level threads). This approach is known to be too heavyweight to exploit dynamic, irregular, and massive parallelism because of its expensive thread management involving OS kernel operations. Hence, numerous studies have proposed lightweight implementations of threads that bypass the OS layer and rely mostly on user-space operations. In this paper, we call these implementations *user-level threads* (ULTs).[1] Numerous production and research threading libraries including major production OpenMP runtimes (as tasks) [1], [2], [3], Qthreads [4], Nanos++ [5] (used in OmpSs [6]), Converse [7] (used in Charm++ [8]), Filaments [9], MassiveThreads [10], and Argobots [11] have adopted ULTs as an implementation of abstract parallel units.

Despite ULTs being more lightweight than OS-level threads, managing ULTs remains a runtime overhead that should be minimized since it dictates how fine-grained work executed by threads can be. For instance, if we assume that an application can tolerate at most 5 percent threading overheads and that thread management costs average $1\ \mu s$, then thread granularity (in terms of time to execute a unit of work) must be at least $20\ \mu s$. For the application to decompose work into more fine-grained portions (e.g., to express more parallelism), threading overheads must be reduced.[2] There is a lower bound on these overheads that is related to supporting the most basic form of concurrency, which requires managing thread descriptors and allowing basic scheduling primitives, such as fork and join operations. Beyond these, we identified that supporting context switching is the next most significant source of overheads.

A context switch involves a ULT (*current*) jumping to another ULT (*target*) by restoring an execution context of the target often after saving the context of the current ULT. Context switching is necessary in order to support complex control flows such as yielding threads of execution (e.g., `pthread_yield()` provided by Pthreads), intermediate termination (e.g., `pthread_exit()`), efficient synchronization (e.g., `pthread_cond_wait()`), and child-first scheduling [12].

1. Literature on a *parallel programming model* tends to refer to such a lightweight parallel unit as a "task", while their *underlying implementations* are often discussed as "threads." This paper focuses on implementations and thus uses "threads" and "ULTs".

2. Example: reducing granularity to $10\ \mu s$ with the same overhead tolerance of 5 percent requires lowering threading overheads to $0.5\ \mu s$

- *S. Iwasaki is with the Department of Information and Communication Engineering, University of Tokyo, Tokyo 113-8654, Japan. E-mail: iwasaki@eidos.ic.i.u-tokyo.ac.jp.*
- *A. Amer is with Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL 60439. E-mail: aamer@anl.gov.*
- *K. Taura is with the Department of Information and Communication Engineering, University of Tokyo, Tokyo 113-8654, Japan. E-mail: tau@eidos.ic.i.u-tokyo.ac.jp.*
- *P. Balaji is with Mathematical and Computer Science, Argonne National Laboratory, Lemont, IL 60439-4844. E-mail: balaji@anl.gov.*

Manuscript received 22 Aug. 2019; revised 14 Jan. 2020; accepted 12 Feb. 2020. Date of publication 28 Feb. 2020; date of current version 31 Mar. 2020. (Corresponding author: Shintaro Iwasaki.) Recommended for acceptance by D. S. Nikolopoulos. Digital Object Identifier no. 10.1109/TPDS.2020.2976057

1045-9219 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

```
thd_desc_t *create_thd(void (*f)(void *), void *arg);
void join_thd(thd_desc_t *thd);
void yield_thd(void);
```

(a) Threading operations that appear in this paper. **RtC** does not support a yield operation (`yield_thd()`) since it requires a context-switching capability (Section 2.1). Other operations such as intermediate termination and synchronization primitives (e.g., a barrier and a mutex) are omitted since they do not appear in our example codes.

```
1  void comp(void *arg) { [...]; }
2  // A parallel version of the following loop:
3  // for (int i = 0; i < n; i++) comp(args[i]);
4  void parallel_loop(void **args, int n) {
5    thd_desc_t *thds[n];
6    for (int i = 0; i < n; i++) // fork ULTs.
7      thds[i] = create_thd(comp, args[i]);
8    for (int i = 0; i < n; i++) // join ULTs.
9      join_thd(thds[i]);
10 }
```

(b) Example code using `create_thd()` and `join_thd()`.

Fig. 1. Basic threading API of a user-level threading library we use in this paper and example code with this API.

Given the practical importance of context switching, several threading runtimes made it compulsory and thus pay the associated performance overheads regardless of the application control flow [5], [7], [10]. Others are more mindful of the context-switching overheads and thus focus on minimizing them by giving up the context-switching capability altogether [1], [2], [3], [9]. A few runtime systems expose both models as distinct language or runtime abstractions so that users can choose either of them based on application requirements [4], [11].

Our first take at this problem showed that it is possible to dynamically provide the context-switching capability later when needed [13]. We referred to this type of method as a *dynamic promotion* technique. Our prior study was based on a direct correlation between yielding and context switching; that is, a context switch triggered by a yield operation requires thread promotion. This observation was biased by the parent-first scheduling assumption, however, which does not apply to child-first scheduling [12]. To capture causes of thread promotion regardless of the scheduling policy, this paper borrows the notion of *deviation* from Spoonhower et al. [14]. This concept lets us shed light on the fundamental causes of thread promotion in various patterns of execution as well as helps us design thread management methods that reduce the corresponding overheads. We then evaluate the performance characteristics of a large spectrum of user-level threading methods with representative real-world codes and link their execution patterns to probabilities of deviation. Our primary finding is that dynamic promotion techniques exhibit the best trade-off between performance and capabilities when the chances of deviation are low.

Specifically, the contributions of our paper are as follows:

- Identifying deviation as the fundamental cause that incurs context-switching and thus imposes the associated fork-join overheads when implementing ULTs;
- In-depth characterization of the performance versus capability trade-off with respect to the probability of deviation while covering all feasible methods for building a generic threading library, including a few methods missing from prior literature;
- Identifying stack management as an orthogonal dimension to the problem and demonstrating its applicability to several threading techniques and its tremendous effect on performance and memory usage;
- Highly optimized implementations of all methods within the same threading library (Argobots [11]) for a fair comparison of all the threading techniques;
- Coverage of major hardware architectures in the high-performance computing community—Intel Skylake, Intel Knights Landing (KNL), ARM 64, and IBM POWER8 processors—to highlight the importance of lightweight user-level threading techniques for architectures that employ less powerful cores and have larger thread contexts;
- Evaluating all the threading methods with N-body, machine learning, and graph analytics codes, in which deviations happen during execution. The results indicate that dynamic promotion techniques that defer context management until a deviation happens show the best performance versus capability trade-off when deviations are unlikely.

This manuscript is an extension of the conference paper previously published by the authors [13]. This paper has more comprehensive coverage by including new threading methods (with respect to stack allocation timing and scheduling policies) and a wider range of modern hardware architectures. In addition, the notion of deviation captures causes of context switching outside a yield operation. Our experiments with the new POWER8 implementation highlight the efficacy of dynamic promotion techniques on different hardware with the corresponding calling conventions. Furthermore, our deeper evaluation with a microbenchmark reveals that the number of created ULTs also affects the performance trade-off. As a result, this paper depicts a comprehensive picture of the performance versus capability trade-off between user-level threading techniques regarding stack allocation strategies, scheduling policies, CPU architectures, and thread counts, which helps programmers select the best threading methods that fit their hardware architectures and application workloads.

1.1 Scope of the Paper

We explore neither a granularity control technique that serializes threads [15], [16] nor a scheduling technique that improves memory locality and alleviates scheduling overheads (e.g., thread pool contention) [17], [18]; our approach tackles the granularity issue from a different aspect by minimizing threading overheads, which can coexist with granularity control strategies and scheduling methods proposed in the previous studies. We do not discuss other parallel programming paradigms (e.g., an event-driven programming model [19]); the focus of our work is a common multithreaded programming model. The target of this work is user-level threading techniques to build a generic threading library without source-to-source translations, compiler modifications (e.g., Cilk [20]), or kernel modifications (e.g., Cilk-M [21]).

2 Basics of a User-Level Threading Library

Our explanation in this work is based on a threading library with a simplified API sketched in Fig. 1, which can be found in most threading libraries. Among functions listed in

```
1 void scheduler() {
2   while (true)
3     if (thd_desc_t *thd = pop_pool())
4       thd->f(thd->arg) // schedule thd.
5 }
```

Fig. 2. Pseudocode of *RtC*.

Fig. 1a, fork and join are the most basic operations; a fork function (`create_thd()`) creates a ULT, and a join function (`join_thd()`) waits for the completion of a given ULT and frees its resource.[3] A *thread pool* is a data structure to keep ready ULTs. A ready ULT is popped from a thread pool and executed by a *scheduler* that runs with its own stack on the corresponding OS-level thread (*worker*). Our following explanation assumes a work-stealing model [23] for load balancing; each worker has its own thread pool and attempts to steal a ready ULT from another worker's pool if needed (e.g., when its local pool is empty).[4] This fork-join mechanism is powerful enough to parallelize several parallel patterns including a parallel loop presented in Fig. 1b.

Consider the simplest thread implementation that supports only fork and join. The essence of fork and join operations is a schedulable function that can be detached from the current execution context and later invoked. Compared with a function call, minimal additional operations to implement such ULTs are twofold: (1) a thread descriptor that stores completion status, a function pointer, and its argument and (2) a scheduling mechanism that keeps thread descriptors and runs a ready ULT, both of which are fundamental for detaching and deferring the execution of the function. A threading method that satisfies only these requirements is the simplest and most lightweight. This technique, however, abandons all threading features that require a context switch; that is, once scheduled, such a ULT does not stop until completion. Hence we call it a *run-to-completion* thread (*RtC*). We first explain the implementation of *RtC* and show why *RtC* can only run to completion. We then describe how to overcome the limitation of *RtC*.

2.1 Run-to-Completion Thread (*RtC*)

We present the pseudocode of *RtC* in Fig. 2 and its execution flow in Fig. 3. *RtC* requires only a thread descriptor and a scheduler; on thread creation, *RtC* allocates a thread descriptor that holds a function pointer and its argument and pushes it to a thread pool. A thread in a pool is pulled by a scheduler running on a worker and simply *called* on top of the scheduler. Compared with an immediate function call natively supported by programming languages, *RtC* incurs overheads of thread descriptor management and scheduling including thread pool operations, both of which are indispensable costs to detach the execution.

Although *RtC* has the smallest fork-join overheads, it lacks threading capabilities that require a context switch because a simple function call welds together a scheduler and an invoked thread; a scheduler that spawns an *RtC* thread cannot be resumed while the invoked thread is running. Consider a

3. We do not impose fully strict computation [22] and allow arbitrary synchronization operations (including a barrier and a mutex) between threads in order to maintain flexibility and generality.

4. This paper does not assume a specific implementation of thread pools and work-stealing algorithms.

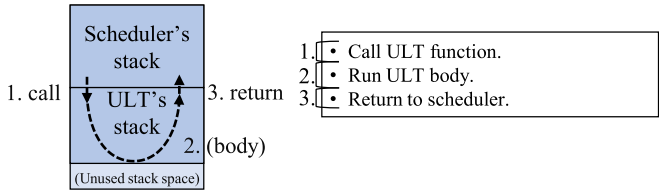


Fig. 3. Flow of fork-join (*RtC*).

yield operation (`yield_thd()` in Fig. 1a) that returns the control from a ULT to a scheduler. In order to restore the context of the scheduler, values of hardware registers (including a stack pointer and an instruction address) must be reinstated. Nevertheless, *RtC* saves none of them explicitly on invocation; thus, although these values are possibly stored somewhere in the call stack of *RtC* as instructed by a compiler, a threading library cannot retrieve these values. Even if registers could be restored, because the invoked *RtC* thread and the scheduler share the same stack region, any stack growth caused by a function call or an invocation of another *RtC* thread would overwrite the call stack of the previous *RtC* thread. This scheduler-thread welding deprives *RtC* of threading features that require an independent invoker's context; unsupported features are not only yielding but also intermediate termination, efficient synchronization, and child-first scheduling. This limitation critically lowers the practicality of *RtC*.

2.2 Thread With Full Threading Capabilities

RtC lacks a context-switching capability because it bonds contexts of a scheduler and a thread together. If their contexts are maintained independently, however, a ULT can return to a scheduler at any point. A *fully fledged* threading technique creates and maintains a thread context in order to support full threading capabilities. Such a thread allows efficient scheduling, but it suffers from context management overheads. To understand the difference in performance and capabilities between these two opposite threading techniques, we first explain user-level context switch, an essential operation to implement fully fledged threads. Our implementation of user-level context switch follows that of Boost C++ Libraries [24]; similar codes are found in major threading packages, for example, in Qthreads [4], Nanos++ [5], Converse [7], and MassiveThreads [10] as well as Argobots [11]. We note that most ULT implementations do not maintain signal masks and compiler-level thread-local storage for every ULT, so they are shared among ULTs running on the same worker. Fig. 4 presents the pseudocode of user-level context switch. This implementation represents a context as a single pointer to the call stack (`ctx_t*` in the figure) since all the other data are saved at the top of the stack. Since a caller of `switch_ctx()` is responsible for saving and restoring *caller-saved* registers before and after calling

```
1 void switch_ctx(ctx_t **self_ctx, ctx_t *target_ctx) {
2   Push callee-saved registers // save the current context.
3   Push the parent instruction address
4   *self_ctx = stack_pointer
5   stack_pointer = target_ctx // restore the target context.
6   Pop the target instruction address to regA // regA is caller-saved.
7   Pop callee-saved registers
8   Jump to *regA
9 }
```

Fig. 4. Pseudo assembly code of user-level context switch.

```
1  void start_ctx(ctx_t **self_ctx, void *stack, void (*f)(void *),
2                 void *arg) {
3    Push callee-saved registers // save the current context.
4    Push the parent instruction address
5    *self_ctx = stack_pointer
6    stack_pointer = stack // start f on top of stack.
7    f(arg)
8  }
9  void end_ctx(ctx_t *target_ctx) {
10   stack_pointer = target_ctx // restore the target context.
11   Pop the target instruction address to regA // regA is caller-saved.
12   Pop callee-saved registers
13   Jump to *regA
14 }
```

Fig. 5. Pseudo assembly code to start and finish thread contexts.

```
1  thread_local ctx_t *g_sched_ctx // worker-local variable.
2  void scheduler() {
3    while (true)
4      if (thd_desc_t *thd = pop_pool()) {
5        if (!thd->is_started) {
6          thd->is_started = true
7          start_ctx(&g_sched_ctx, thd->stack, thd_wrapper, thd)
8        } else
9          switch_ctx(&g_sched_ctx, thd->ctx)
10       if (!thd->is_finished)
11         enqueue_pool(thd) // return thd to pool.
12     }
13 }
14 void thd_wrapper(thd_desc_t *thd) {
15   thd->f(thd->arg) // thd->f and thd->arg are given by users.
16   thd->is_finished = true
17   end_ctx(g_sched_ctx)
18 }
```

Fig. 6. Pseudocode of *Full*.

`switch_ctx()`, `switch_ctx()` itself needs to manage only *callee-saved* registers (lines 2 and 7).[5] This routine first saves all the callee-saved registers including an instruction address on top of the stack (lines 2 and 3) and stores the current stack pointer in `self_ctx` (line 4). Then, `switch_ctx()` updates the stack pointer to the stack address pointed to by `target_ctx` (line 5) and restores the instruction address and the callee-saved register values from the stack of the target in reverse order (lines 6 and 7). The target, which is suspended in `switch_ctx()`, is resumed by jumping to the target instruction address (line 8). We note that all of these operations are executed in the user space.

If a scheduler's context has been saved properly, `switch_ctx()` enables a ULT to save its context and resume a scheduler whenever it needs to return to a scheduler. This method, however, is inappropriate for initiating a ULT because `switch_ctx()` takes `target_ctx` that must have been already initialized. This routine always saves the context of the current ULT, but this action is unnecessary when a ULT finishes because that ULT will never be resumed again. To efficiently handle these cases, we split the functionality of `switch_ctx()` and create two methods, `start_ctx()` and `end_ctx()`, to start and finish contexts, respectively. Fig. 5 shows the pseudocodes of these functions. Their implementations come from the first and the latter parts of `switch_ctx()`. `start_ctx()` saves the context of the current thread (lines 3–5) but freshly executes a function `f()` on top of `stack` (lines 6 and 7), while `end_ctx()` restores and resumes the target context (lines 10–12) without saving the current context.

Fully fledged threads that support the full threading capabilities are implemented with the three context-switching functions described above. In reality, we can find two implementations of fully fledged threads that have been developed to support different scheduling policies; one is for *parent-first* scheduling, and the other is for *child-first* scheduling.[6] We first explain parent-first fully fledged threads and then child-first threads.

2.2.1 *Parent-First Fully Fledged Thread (*Full*)*

A parent-first scheduling policy is the same as the scheduling order of *RtC*; on `create_thd()`, a parent (i.e., a caller) pushes a child thread to a thread pool and resumes the execution of the parent itself, and later a scheduler executes the child stored in the thread pool. For example, in Fig. 1b, a parent thread that runs `parallel_loop()` first creates all child threads and pushes them to a thread pool in the loop (lines 6–7). On `join_thd()` (line 9), the parent thread checks the completion of each child thread. If the child thread is not completed (e.g., by this worker or other workers), the parent cannot make progress and thus context-switches to a scheduler and runs a ready ULT.

In this paper, we refer to the implementation of a fully fledged threading technique with parent-first scheduling as *Full*. Fig. 6 shows the pseudocode of *Full*. The scheduler first pops a ULT (`thd`) from a pool (line 4) and starts it by `start_ctx()` (line 7) if `thd` has not been executed previously; otherwise it resumes `thd` by `switch_ctx()` (line 9) since its context has already been initialized. A user-given thread function is called in a wrapper function `thd_wrapper()` (line 15) so that `end_ctx()` is executed on completion (line 17) because a ULT invoked by `start_ctx()` cannot return to the parent scheduler just by a standard `return` procedure.

Since both `start_ctx()` and `switch_ctx()` save the scheduler's context in `g_sched_ctx`, the scheduler can be resumed by `switch_ctx()` or `end_ctx()` at any time, thus allowing yielding, intermediate termination, and efficient synchronization. Child-first scheduling also requires user-level context switch, as we describe in the next section.

2.2.2 *Child-First Fully Fledged Thread (*C-Full*)*

Child-first scheduling [12] is a different scheduling policy from that of *RtC* and *Full*; under the child-first scheduling policy, on thread creation, a parent thread yields to a child thread, and the child pushes the parent into a thread pool so that another scheduler can steal the continuation of the parent ULT. After the child completes, it preferably jumps back to the parent thread if the parent is still in the thread pool. For instance, in Fig. 1b, a caller of `parallel_loop()` (i.e., a parent) pushes its continuation to a thread pool and executes a child thread first on `create_thd()` (line 7). Parallelization is achieved by exposing a continuation of a parent thread to other workers. If no work stealing happens, the child returns to the parent context on completion and the parent creates a next child thread in the loop (lines 6–7). Since this child-first scheduling naturally executes threads in sequential order (or depth-first order) if no work stealing

5. Threading libraries must save and restore all callee-saved registers specified by application binary interfaces (ABIs) because, without a special compiler help, libraries are unable to obtain information about which callee-saved registers are read after calling `switch_ctx()`.

6. Parent-first scheduling is sometimes called *help-first* scheduling while child-first is called *work-first*.

```
1  thread_local thd_desc_t *g_current_thread // worker-local variable.
2  thd_desc_t *create_thd(...) {
3    thd_desc_t *thd = allocate_thd_desc_t()
4    init_thd_desc(thd, ...)
5    thd->parent = g_current_thread
6    start_ctx(&g_current_thread, thd->stack, thd_wrapper, thd)
7    return thd
8  }
9  void thd_wrapper(thd_desc_t *thd) {
10   push_local_pool(thd->parent)
11   thd->f(thd->arg) // thd->f and thd->arg are given by users.
12   thd->is_finished = true
13   ctx_t *next_ctx = pop_local_pool_or_get_sched_ctx()
14   end_ctx(next_ctx) // child-first scheduling expects next == parent.
15 }
```

Fig. 7. Pseudocode of *C-Full*.

happens, it is often adopted to parallelize divide-and-conquer recursive algorithms for better locality [20], [22]. Such a child-first fully fledged thread, which we call *C-Full* in this paper, can also be implemented with the context-switching functions shown in Figs. 4 and 5.

The pseudocode of *C-Full* is presented in Fig. 7. *C-Full* performs a context switch in a thread creation function (`create_thd()`); after allocating and initializing a thread descriptor, a parent thread saves its context and jumps to a child thread by `start_ctx()` (line 6). The child pushes the parent to a local thread pool (line 10) before running a user-given thread function (line 11) to expose concurrency. On completion, the child thread checks the next thread in the pool, which is ideally the parent thread so that execution order is depth-first. However, the child does not always succeed in taking the parent because it might have been either stolen by another scheduler or resumed by threading operations (e.g., `yield_thd()`). If this is the case, the child thread jumps to another thread if it exists; otherwise, the child thread returns to the scheduler (`scheduler()` in Fig. 6). We note that *C-Full* has also the full threading capabilities since all parent and child threads and schedulers maintain their contexts independently.

2.3 Performance Comparison

In both the parent- and child-first cases, the management of call stacks and callee-saved registers plays a key role in supporting full threading capabilities and child-first scheduling. In real applications, however, the ULT is often executed as if it were just *called* by following a normal function-call procedure; *Full* threads can finish without any context switch during execution, and *C-Full* threads can just run to completion and return to the parent thread.

To analyze the performance difference, we use a notion of *deviation* appearing in [14].[7] A deviation in the work by Spoonhower *et al*. [14] is defined as an event that prevents a ULT from sequential execution (i.e., execution order where all thread creations are inlined). Since sequential execution is often most efficient in terms of memory locality [14], [25], the number of deviations has been used as a metric that represents how far the resulting parallel execution differs from sequential execution. This idea works well for child-first scheduling; the number of deviations can be zero if neither work stealing nor yielding happens. If we follow the original definition, however, all fork and join operations of parent-first threads incur deviations because, unlike child-first

7. We note that this is called differently in other literature; for example, such an event is called "drifted" in [25].

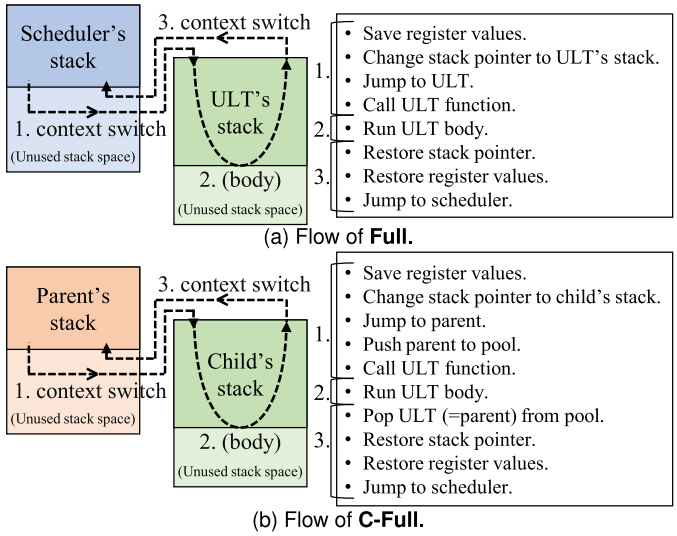


Fig. 8. Flow of fork-join when no deviation happens.

order, parent-first order is different from sequential execution order even when a parent-first ULT is executed in a run-to-completion manner. This paper, therefore, generalizes the notion of deviation by defining it as an event causing an execution that is different from sequential execution *except* such an event on forking and joining parent-first threads. With this definition, the number of deviations under parent-first scheduling can be zero in a case where no context switch happens during the execution of a thread. Deviation includes any threading operations requiring context switch during execution (e.g., a yield operation, intermediate termination, and synchronization) and, in the child-first case, an event where a parent thread is stolen by another scheduler. We note that no deviation is allowed with *RtC*.

Fig. 8 illustrates the execution paths of *Full* and *C-Full* without deviation. Comparison of Fig. 8 with Fig. 3 shows that both *Full* and *C-Full* incur the following additional overheads compared with *RtC*, lowering the performance of *Full* and *C-Full* even when no deviation happens.

1. Save callee-saved registers on ULT invocation (`start_ctx()`).
2. Restore callee-saved registers on ULT completion (`end_ctx()`).
3. Manage call stacks for `thd->stack`.

To quantify the performance difference, we created a microbenchmark that controls the chances of deviation by adding a yield operation. Specifically, we ran a microbenchmark that creates and joins N empty ULTs as shown in Fig. 9. In this

```
1  void kernel(void *yield_flag) {
2    if (yield_flag != NULL)
3      yield();
4  }
5  void microbenchmark(int n) {
6    void *yield_flags[N];
7    thd_desc_t *thds[N];
8    // n yield_flags are set to non-NULL, while 0 <= n <= N.
9    set_yield_flags(yield_flags, n);
10   for (int i = 0; i < N; i++) // fork ULTs.
11     thds[i] = create_thd(kernel, yield_flags[i]);
12   for (int i = 0; i < N; i++) // join and free ULTs.
13     join_thd(thds[i]);
14 }
```

Fig. 9. Microbenchmark that forks and joins N ULTs while random n out of N ULTs encounter deviations invoked by `yield_thd()`.

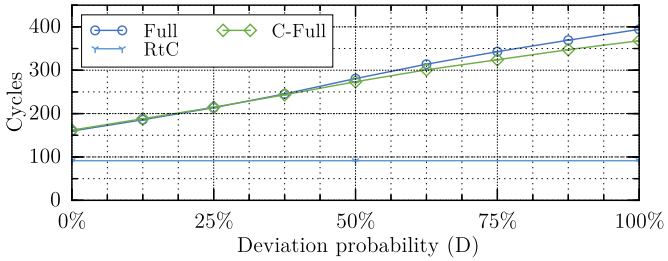


Fig. 10. Fork-join overheads on an Intel Skylake machine using a microbenchmark presented in Fig. 9 ($N = 4,096$). *RtC* shows the performance at $D = 0\%$ because *RtC* does not allow any deviation.

benchmark, n randomly chosen ULTs yield once, so $n/N\%$ of ULTs encounter deviations. We define a deviation possibility D as n/N and changed D by controlling n while fixing N to 4,096. We ran this microbenchmark on a single core of an Intel Skylake processor (see Section 4 for details).

Fig. 10 shows the fork-join overheads regarding the deviation probability (D). Our result is the arithmetic mean of all iterations. Because *RtC* does not allow deviation, we draw a horizontal line which has the value at $D = 0\%$ (i.e., no deviation). The result shows that even when no deviation takes place, the overhead of *Full* is 1.7x higher than that of *RtC* because of the context management. We note that although their scheduling policies are different, *Full* and *C-Full* perform similarly because the expensive operations including register and stack management are common.

Ideally, *Full* would perform as well as *RtC* when no deviation occurs, and so would *C-Full*. However, it has been an open question whether this performance gap is inevitable when employing full threading capabilities or whether other threading techniques offer different performance and capability trade-offs. In the next section we analyze the performance discrepancy and investigate threading techniques that exist between these two opposite directions, that are more efficient than *Full* and *C-Full* when no deviation happens, and that keep full threading capabilities.

3 Lightweight User-Level Threading Techniques With Dynamic Promotion

In this section we analyze the performance gap between fully fledged techniques (*Full* and *C-Full*) and *RtC* and explore intermediate threading techniques. The analysis uses the same microbenchmark presented in the preceding section (Fig. 9) and progressively cuts down the overheads of *Full* and *C-Full* at $D = 0\%$. Each step of the analysis finds a lightweight threading technique that has a different trade-off between performance with and without deviation and programming constraints.

```
1  void start_ctx_RoC(ctx_t **self_ctx, void *stack, void (*f)(void *),
2                     void *arg) {
3    Push callee-saved registers
4    Push an instruction address
5    *self_ctx = stack_pointer
6    stack_pointer = stack
7    f(arg) // a user function is directly called.
8    return
9  }
10 void end_ctx_to_sched(void) {
11   end_ctx(g_sched_ctx)
12 }
```

Fig. 11. Pseudo assembly code of context switch in *RoC*.

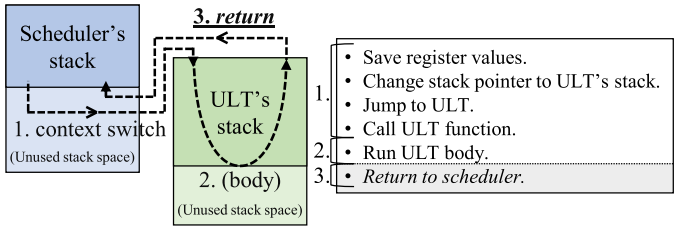


Fig. 12. Flow of *RoC* when no deviation happens. The difference from *Full* (Fig. 8a) is written in italic.

3.1 Parent-First Scheduling

We first look at parent-first threading techniques. Our analysis reduces the overhead of *Full* toward that of *RtC* while keeping the capabilities of *Full*. Instruction breakdowns and performance data of all the parent-first methods are summarized in Figs. 16 and 17, respectively.

3.1.1 *Removing Context Switch on Completion (*RoC*)*

Our instruction analysis shows a large difference in instruction counts between *Full* and *RtC* at $D = 0\%$; even if no deviation occurs, *Full* performs context switch twice, imposing as many as 50 instructions. The first context switch from a scheduler to a ULT is necessary in order to make a scheduler resumable at any point. If no deviation occurs, however, the second manipulation of callee-saved registers is unnecessary since the register values of the scheduler are restored by a user-given thread function (`thd->f()`). A *return-on-completion* technique (*RoC*) exploits the fact that the first context switch is inevitable but the last one can be omitted if no deviation takes place during execution; *RoC* replaces the second context switch by a standard return procedure to reduce the context-switching cost on completion.

Fig. 11 presents the pseudocode of a function that invokes *RoC*, and Fig. 12 illustrates its execution flow without deviation. `start_ctx_RoC()` first saves callee-saved registers (lines 3 and 4), changes a stack (lines 5 and 6) as `start_ctx()` does (Fig. 5), and directly calls a thread function `f()` (line 7). If a created thread has not encountered a deviation, the parent scheduler has never been resumed, so the *RoC* thread can simply return to a scheduler without restoring callee-saved registers saved at lines 3 and 4 because they were restored by `f()`. Thus, `start_ctx_RoC()` can return to a scheduler by a `return` instruction.[8] However, the scheduler cannot simply be resumed by a return procedure if deviations have happened because a deviation staled the callee-saved registers saved in `f()`. In order to address this issue without extra overheads, the return address stored in the call stack of `start_ctx_RoC()` is updated to `end_ctx_to_sched()` when a first deviation happens; if the *RoC* thread has confronted a deviation,

8. We assume a return mechanism similar to that of the x86/64 ABI [26]; a `return` instruction pops an instruction address from the call stack and jumps to that address. Unlike the x86/64 instruction set [27], however, several architectures including ARM [28] and POWER [29] do not have such a multifunctional return instruction. Nevertheless, their ABIs [30], [31] adopt similar calling conventions, which save an instruction address at a predefined location at a stack frame boundary. Thus we can implement the same algorithm on these architectures by combining multiple instructions as most compiler-generated codes do in a function epilogue.

```
1  void start_ctx_SS(ctx_t **self_ctx, void *stack, void (*f)(void *),
2                    void *arg) {
3    *self_ctx = stack_pointer
4    stack_pointer = stack
5    f(arg) // a user function is directly called.
6    return
7  }
8  void end_ctx_invoke_sched(void) {
9    stack_pointer = scheduler's stack_top
10   scheduler()
11 }
```

Fig. 13. Pseudo assembly code of context switch in *SS*.

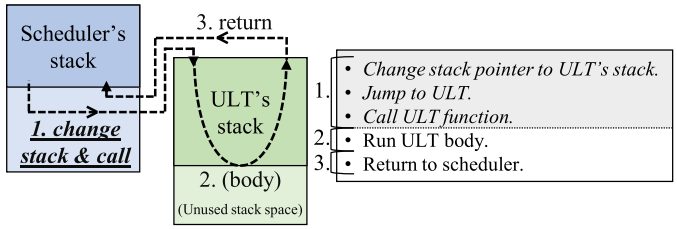


Fig. 14. Flow of *SS* when no deviation happens. The difference from *RoC* (Fig. 12) is written in italic.

`start_ctx_RoC()` does not directly return to the scheduler but jumps to `end_ctx_to_sched()` by `return` so that the context of the scheduler can be properly restored by `end_ctx()` (line 11). We note that only the first deviation needs to modify the return address, so succeeding deviation events do not incur any overhead.

When D is 0 percent, *RoC* omits one context switch per fork-join and successfully saves 24 instructions compared with *Full*, achieving 16 percent less overheads than does *Full*. However, *RoC* degrades performance when D is large (4 percent worse at $D = 100\%$) because when a deviation happens, *RoC* performs the same number of context switches but complicates the control flow.

3.1.2 *Removing Context Switch on Invocation (*SS*)*

Although *RoC* successfully skips register manipulations on completion at $D = 0\%$, saving a context on invocation makes *RoC* slower than *RtC*. We save the scheduler's context in order to resume it later, but if the scheduler does not need to preserve its state including local variables and its progress, we can freshly start a new one. We call this property of a scheduler *statelessness*. We propose a new threading technique *stack separation* (*SS*) that separates stacks but does not save a context of the scheduler on invocation, while this technique requires a stateless scheduler.

Fig. 13 shows the pseudocode of *SS*. After changing a stack pointer (lines 3 and 4), *SS* directly calls a thread function (line 5). As illustrated in Fig. 14, if no deviation happens, it returns to the scheduler with a standard `return` (line 6) as *RoC* does. If a deviation occurs, the return address in the call stack of the *SS* thread is updated so that *SS* jumps to a function without restoring the outdated scheduler's context. However, *SS* cannot resume the scheduler by `end_ctx_to_sched()` in Fig. 11 because *SS* does not save the scheduler's callee-saved registers. Instead, *SS* calls `scheduler()` on the stack of the original scheduler, which flushes all the local variables in the call stack and the progress stored in an instruction address.

SS further reduces 14 instructions compared with *RoC* when D is 0 percent, achieving 14 percent higher performance than *RoC* does. However, *SS* lowers performance if a deviation happens (7 percent slower than *Full* at $D = 100\%$) because *SS* essentially needs to rerun a scheduler from the beginning of the function, which is unnecessary if the scheduler context is properly saved.

Although *SS* performs better than *Full* and *RoC* at $D = 0\%$, *SS* imposes a programming constraint that requires a stateless scheduler, narrowing the applicability of *SS*. A random work-stealing scheduler [23] can be implemented as stateless, but we note that not all schedulers are trivially stateless; for example, a scheduler is not stateless if it saves counters in local variables to select a victim of work stealing or if it sleeps when work stealing fails continuously.

3.1.3 *Lazy Stack Allocation (*Full-L, RoC-L, *and* SS-L*)*

SS remains slower than *RtC*. We observe that *RtC* incurs fewer L1 and L2 cache misses than do the other techniques at $D = 0\%$ because *RtC* accesses only the scheduler's stack while each invocation of *Full*, *RoC*, and *SS* touches an independent call stack that is preallocated on creation. Such an eager stack allocation strategy is common in practice to facilitate management of a thread descriptor and a stack; it allows a runtime to reduce memory management operations by allocating together thread descriptors and their corresponding stacks (i.e., use part of a stack region as a descriptor). Nevertheless, this practice increases the memory accesses since each ULT invocation accesses a different stack area that is unlikely in caches. As a result, *Full*, *RoC*, and *SS* increase L1 and L2 cache misses at $D = 0\%$.

However, not all the ready ULTs need to have independent stacks; only simultaneously active ULTs require independent stacks. To reduce the memory footprint, we introduce a *lazy stack allocation* method (LSA) that decouples the management of thread descriptors and stacks and assigns a stack at invocation time. Since most ULTs are forked and joined sequentially when D is small, a call stack can be reused across thread invocation. *Full*, *RoC*, and *SS* can adopt LSA without changing their context-switching algorithms. We refer to these techniques by adding a suffix *-L* to their names.

We observe that *Full-L*, *RoC-L*, and *SS-L* achieve slightly higher performance than do the original techniques by successfully reducing L1 and L2 cache misses at $D = 0\%$; their numbers of L1 and L2 cache misses are almost the same as those of *RtC*. However, LSA adds 11 instructions to manage a stack and a thread descriptor independently, so LSA possibly degrades performance on different machines that have different instruction, memory, and cache costs. The results also show that the advantage of LSA becomes negligible as D gets higher since more ULTs need independent stacks; as a result, the additional allocation operations incurred by LSA lower the performance. We note that LSA promotes stack reuse and thus can reduce the memory footprint when D is small, which is evaluated in Section 4.1.4.

3.1.4 *Removing Stack Change (*SC*)*

Threading overheads still exist in the stack management, which fundamentally makes *SS-L* slower than *RtC*. If a scheduler is stateless, however, a scheduler can be spawned

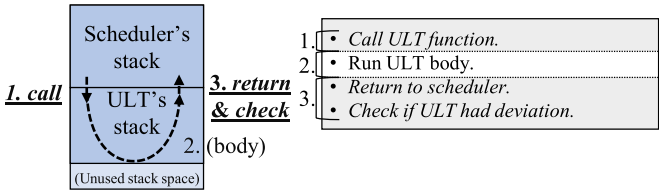


Fig. 15. Flow of *SC*. The difference from *SS* (Fig. 14) is written in italic.

on top of the newly allocated stack, which eliminates management of both callee-saved registers and stacks. The technique that newly creates a scheduler has been adopted by some runtimes [32], [33], [34]. We refer to this technique as *scheduler creation* (*SC*). When an *SC* thread encounters a deviation for the first time, it spawns a ULT with a new stack and starts a scheduler on top of it. At the same time, the original scheduler currently running the *SC* thread is invalidated by updating a flag in order to keep the number of active schedulers. On completion, a scheduler checks the invalidation flag; if invalidated, it jumps to an active scheduler using `g_sched_ctx`.

In addition to the requirement of a stateless scheduler as *SS* and *SS-L* have, *SC* imposes a new constraint on the stack size; because stacks are shared with *SC* threads and schedulers, the stack size of all *SC* threads must be the same as that of the scheduler, forcing users to adopt the largest stack size that fits all threads in a program. This constraint is significant when one application contains multiple types of threads each of which requires a different stack size.

Fig. 16 summarizes the instruction breakdowns with and without deviation and Fig. 17 shows the performance and cache misses of eight parent-first threading techniques. Fig. 16a shows that at $D = 0\%$*SC* adds only three instructions to check the invalidation flag. As a result, the overhead of *SC* is as small as that of *RtC* at $D = 0\%$ while *SC* supports all the threading capabilities that may cause deviations. However, restarting a scheduler on a new stack is expensive in terms of the number of instructions and memory accesses; hence, *SC* shows the worst performance among the seven methods at $D = 100\%$.

3.2 Child-First Scheduling

In this section, we apply the same analysis methodology to child-first techniques. We use the same microbenchmark to evaluate their overheads. Their instruction breakdowns and performance are summarized in Figs. 20 and 21.

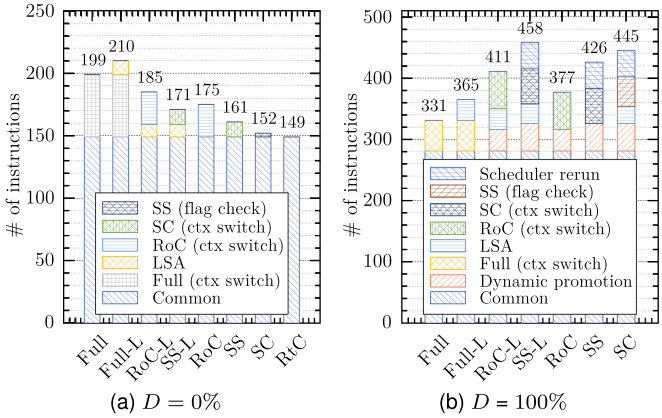


Fig. 16. Instruction breakdown of fork and join operations on Skylake (parent-first methods).

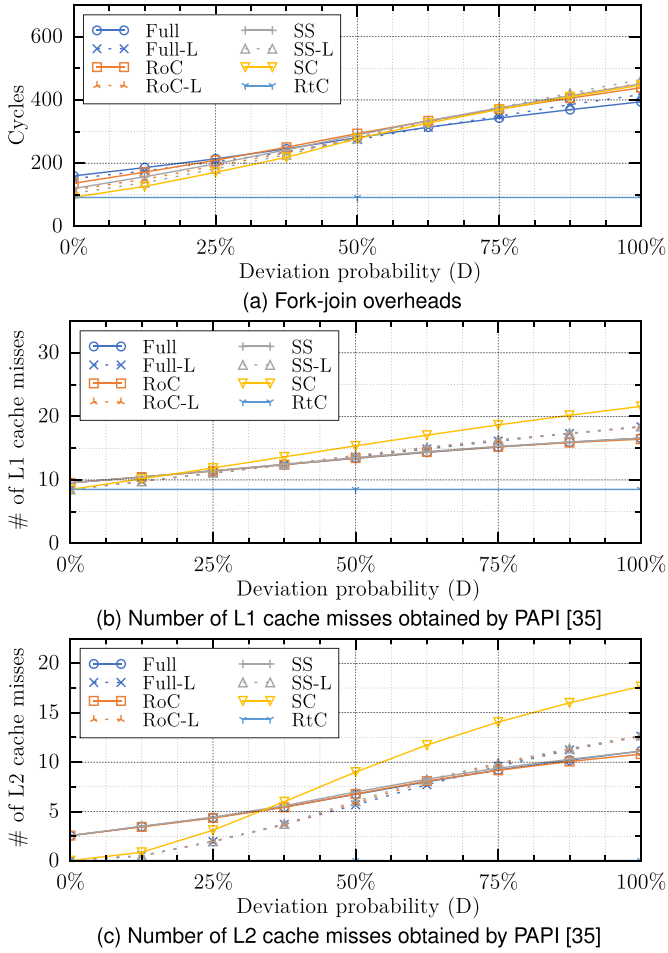


Fig. 17. Performance of the parent-first threading methods on Skylake. Almost no L3 cache miss happens in this experiment because each ULT accesses a small portion of a call stack; we therefore omit the data.

3.2.1 *Eager Stack Release* (C-Full-E)

C-Full suffers from large L1 and L2 cache misses because each ULT has an independent stack. LSA, which allocates stack on invocation, seems promising to reduce cache misses incurred by stack accesses. However, LSA itself is not applicable to child-first techniques because creation and invocation are done in the same function (i.e., `create_thd()`). To promote stack reuse, we devise an *eager stack release* method (ESR) that frees stacks not when threads are joined (i.e., `join_thread()`) but on completion of ULTs. ESR allows consecutively spawned ULTs to reuse the same stack region if no deviation happens; however, ESR needs to decouple the management of thread descriptors and stacks, adding extra overheads to handle them separately. We call this technique *C-Full-E*.

Although ESR imposes 11 instructions for independent management of thread descriptors and stacks, *C-Full-E* successfully eliminates L2 cache misses and reduces L1 cache misses, achieving an overall performance improvement of 13 percent when no deviation takes place. In addition, as will be evaluated in Section 4.1.4, this stack reuse can dramatically reduce the memory footprint when the deviation probability

```
1  thd_desc_t *create_thd(...) {
2    thd_desc_t *thd = allocate_thd_desc_t()
3    init_thd_desc(thd, ...)
4    thd->parent = g_current_thread
5    start_ctx_RoC(&g_current_thread, thd->stack, thd_wrapper, thd)
6    if (stolen_by_another_worker)
7      *(thd->stack + RETURN_ADDRESS_OFFSET) = end_ctx_to_sched
8    return thd
9  }
10 void thd_wrapper(thd_desc_t *thd) {
11   push_local_pool(thd->parent)
12   thd->f(thd->arg) // thd->f and thd->arg are given by users.
13   thd->is_finished = true
14   ctx_t *next_ctx = pop_local_pool_or_get_sched_ctx()
15   if (next_ctx == thd->parent->ctx)
16     return
17   else
18     end_ctx(next_ctx)
19 }
```

Fig. 18. Pseudo assembly code of context switch in *C-RoC*.

is low. However, ESR fails to effectively reuse stacks and degrades performance by additional stack and thread descriptor management as the deviation probability increases.

3.2.2 *Removing Context Switch on Completion (*C-RoC *and* C-RoC-E*)*

As *Full-L* does, *C-Full-E* manipulates callee-saved registers on both invocation and completion. Unlike parent-first scheduling, child-first scheduling must preserve the context of the parent ULT since it is controlled by the users. Hence, child-first scheduling needs to maintain an independent stack and manage callee-saved registers on invocation. If no deviation happens, however, the parent thread can be resumed by a return function using a *return-on-completion* technique. This technique is applicable to both *C-Full* and *C-Full-E*; we call them *C-RoC* and *C-RoC-E*, respectively. Child-first return-on-completion techniques, however, need to deal with a deviation caused by work stealing to the parent thread.[9] Therefore, *C-RoC* and *C-RoC-E* need an algorithm that allows a thief to update the return address in the call stack of the child ULT.

We first look at the algorithm of *C-RoC*. The pseudocode of *C-RoC* is shown in Fig. 18. The first context switch uses `start_ctx_RoC()` presented in Fig. 11; if no deviation happens, the child ULT can return to the parent (line 16). As *RoC* does, the first occurrence of any threading operation that causes a deviation updates the return address stored in the stack. In the case of *C-RoC*, however, deviations can be caused by work stealing; even if no context-switching operation is performed by the child ULT, the child may not simply return to the parent ULT if the parent has been stolen. To handle this case, the thief updates the return address of the child (lines 6 and 7), which does not exist in *RoC*. We note that there is no data race between an update by a thief (line 7) and reading a return address by a child (line 16) because the child ULT performs `return` only after taking the parent ULT (line 14). As illustrated by Fig. 19, *C-RoC* successfully removes callee-saved register management on completion when no deviation takes place.

In addition to changing the stack management, *C-RoC-E* requires a small modification to *C-RoC* because at line 7 in

9. It does not happen with parent-first scheduling since a scheduler, which corresponding to a parent in child-first scheduling, is never stolen by another worker.

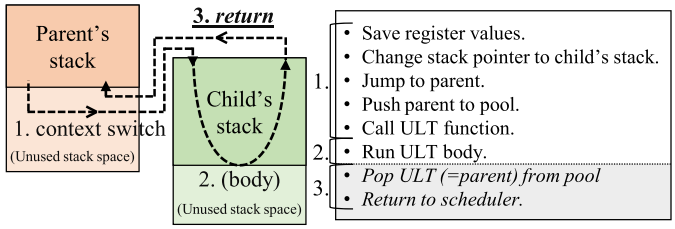


Fig. 19. Flow of *C-RoC* when no deviation occurs. We emphasize the difference from *C-Full* (Fig. 8b) by italicizing it.

Fig. 18a parent may update a child stack that has been already freed under the ESR policy. A thread descriptor of a child is, however, always available in `create_thd()` since the descriptor has not yet been returned to the caller of `create_thd()`. *C-RoC-E*, therefore, does not access a return address in the call stack but reads a member variable in a thread descriptor. This change makes an update by a thief safe but imposes additional overheads for reference in comparison with directly manipulating values in a call stack at $D = 0\%$.

The instruction breakdowns and performance of the four child-first threading methods are summarized in Figs. 20 and 21. Fig. 20 shows that at $D = 0\%$ *C-RoC* and *C-RoC-E* in total reduce 5 and 4 instructions compared with *C-Full* and *C-Full-E*, respectively. However, *C-RoC* increases the memory footprint because the complicated operations in `thd_wrapper()` require larger stack space, which increases L1 and L2 cache misses. Overall, *C-RoC* is 10 percent slower than *C-Full* even if no deviation happens. *C-RoC-E* overcomes this issue of memory footprint by reusing stacks, which successfully trims down the overhead by 29 percent compared with *C-Full* at $D = 0\%$ while worsening performance by 7 percent at $D = 100\%$.

3.3 Summary

Table 1 shows the trade-off regarding performance and programmability. *Full*, *Full-L*, *RoC*, *RoC-L*, and all the child-first threading techniques have no programming constraints because they save a parent context, while *SS*, *SS-L*, and *SC* require stateless schedulers. *SC* has an additional constraint on stack size, which further narrows its applicability. *RtC* supports no threading capability that requires a context switch such as yielding, intermediate termination, and efficient synchronization. However, highly constrained threading techniques perform better if no deviation happens; in the case of parent-first scheduling, *RtC* and *SC* perform

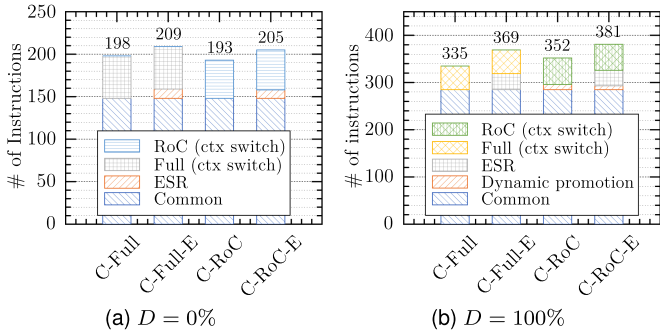


Fig. 20. Instruction breakdown of fork and join operations on Skylake (child-first methods).

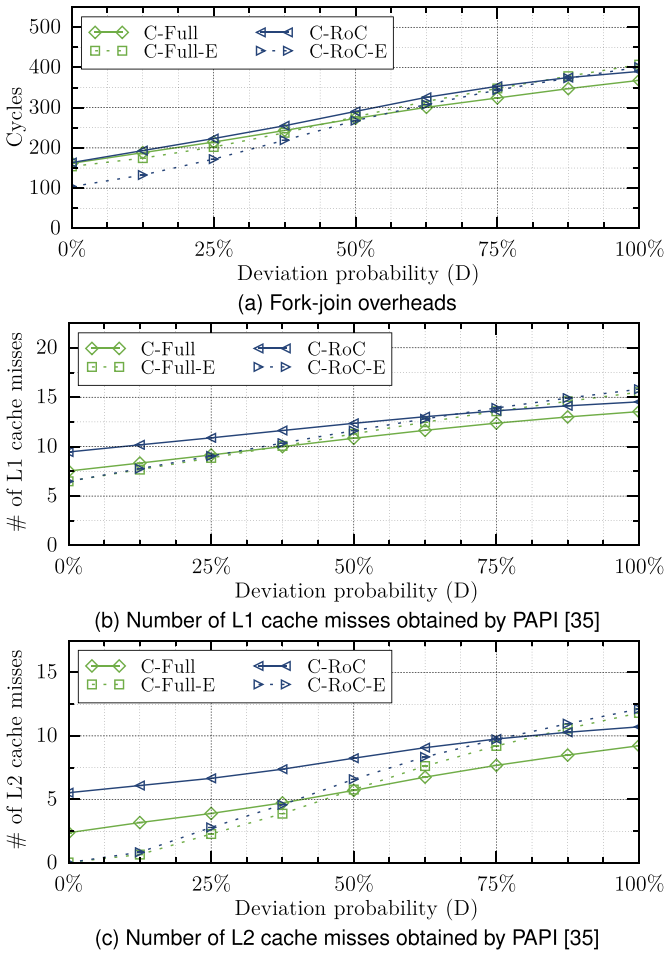


Fig. 21. Performance of the child-first methods on Skylake. Higher levels of caches do not suffer from cache misses in this experiment because each ULT accesses a small portion of a call stack; we therefore omit the data.

better than the others. We also note that in both parent- and child-first cases, threading techniques that show better performance at low D tend to perform worse at large D because if deviation happens, dynamic promotion methods that lazily manage the stack and callee-saved registers incur extra overheads than do eager methods. We note that these techniques can co-exist in a single threading library without impacting other techniques. We will discuss how to choose the best technique in Section 4.5.

3.4 Coverage of Our Techniques

Table 2 summarizes the coverage of our analysis. An area labeled with (*) in the table denotes an absence of practical techniques. In the following, we explain reasons why these techniques are infeasible for general threading libraries.

*Saving Registers (*1)*. Our analysis does not include threading techniques that do not change a stack but explicitly maintain callee-saved registers on invocation. Intuitively, if a stack is shared between a parent and a scheduler, resuming a scheduler is prohibitive since a scheduler can potentially overwrite the invoked stack. On the other hand, if we totally rerun a new scheduler as *SC* does, storing callee-saved registers is pointless. In the past, however, such techniques have been proposed for child-first scheduling [36], [37], [38], [39]. We note that these techniques are not suitable for building threading libraries because compiler modifications are required. We discuss their techniques in Section 5.

*Restarting Parent ULTs (*2)*. With parent-first scheduling, *SS*, *SS-L*, and *SC* restart a stateless scheduler on deviation. This technique is not applicable to child-first threads, however, since parent ULTs are in most cases not stateless; rerunning a parent ULT loses not only the result computed by the parent ULT but also a child thread descriptor if the parent is in the midst of the thread creation function. This is an impractical restriction as a thread, so we do not show child-first techniques that require stateless parent ULTs.

*Eager Stack Management for SC (*3)*. From the viewpoint of stack management, *SC* follows the LSA policy; *SC* allocates a stack for a scheduler not on creation but on deviation. One might suggest allocating a stack and a thread descriptor together on creation for *SC*, but such an eager stack allocation strategy does not work for *SC*. *Full*, *RoC*, and *SS* always keep the same pair of a stack and a thread descriptor, while *SC* needs to decouple the management of the stack and thread descriptor since a stack required on deviation is assigned to a new scheduler, not a thread associated with a thread descriptor.

4 Evaluation

In this section, we evaluate the performance of all the threading techniques presented in the preceding sections with a microbenchmark and three fine-grained parallel applications. All the parent- and child-first threading techniques were implemented in Argobots [11], a highly optimized

TABLE 1
Summary of the Twelve Threading Techniques

			$D = 0\%$			$D = 100\%$	Constraints
		LSA/ESR	Change stack?	# of Register Managements	Overheads	Overheads	
Parent-First	Full (*Fully Fledged Thread*)	No	Yes	2	Highest	Lowest	No
	Full-L (*Fully Fledged Thread (LSA)*)	Yes	Yes	2			No
	RoC (*Return on Completion*)	No	Yes	1			No
	RoC-L (*Return on Completion (LSA)*)	Yes	Yes	1			No
	SS (*Stack Separation*)	No	Yes	0			Scheduler must be stateless.
	SS-L (*Stack Separation (LSA)*)	Yes	Yes	0			Scheduler must be stateless.
	SC (*Scheduler Creation*)	Yes	No	0		Highest	Scheduler must be stateless. Stack size is shared.
	RtC (*Run to Completion*)	-	No	0	Lowest	-	No deviation is allowed.
Child-First	C-Full (*Fully Fledged Thread*)	No	Yes	2	Highest	Lowest	No
	C-Full-E (*Fully Fledged Thread (ESR)*)	Yes	Yes	2			No
	C-RoC (*Return on Completion*)	No	Yes	1			No
	C-RoC-E (*Return on Completion (ESR)*)	Yes	Yes	1	Lowest	Highest	No

TABLE 2
Coverage of Our Analysis

Change Stack?	# of Register Managements	LSA/ESR	Parent-First	Child-First
Yes	2	No	Full	C-Full
		Yes	Full-L	C-Full-E
	1	No	RoC	C-RoC
		Yes	RoC-L	C-RoC-E
	0	No	SS	(*2)
		Yes	SS-L	
No	2	No	(*1)	
		Yes		
	1	No		
		Yes		
	0	No	(*3)	
		Yes	SC/RtC	

user-level threading library. Our experimental environments are described in Table 3. All the programs were compiled with `-O3`. We set the same stack size (64 KB for ExaFMM and 16 KB for the others) to both ULTs and schedulers to the advantage of *SC*. All results reported in this paper are the arithmetic mean. The error bars in the charts indicate the 95 percent confidence intervals.

4.1 Fork-Join Microbenchmark

We first evaluate the threading overheads with the fork-join microbenchmark used in the preceding sections; the code is presented in Fig. 9. This microbenchmark repeats creating and joining N ULTs on a single worker. Deviations are artificially introduced by `yield_thd();` the deviation probability D is calculated by n/N, where n ULTs uniformly selected out of N ULTs yield once. We used a lightweight private pool [11] to minimize overheads of pool operations. In the microbenchmark, the set of N fork-join operations was repeated $2^{19}/N$ times and obtained the average of the execution time. The result of *RtC* is at $n = 0$ (i.e., $D = 0\%$) because *RtC* cannot yield. We ran this microbenchmark 50 times on Skylake, KNL, POWER8, and ARM64.

4.1.1 Performance With Different Deviation Probabilities

Fig. 22 shows the results with various D values where N is fixed to $4{,}096$ in order to see how the deviation probability affects the threading overheads. For better visibility, we separate results by scheduling type; Fig. 22a shows the parent-first techniques while Fig. 22b plots only the child-first ones. First, all the results indicate the same performance trend: *SC*, *SS*, and *RoC* outperformed *Full* at $D = 0\%$ because these dynamic promotion techniques alleviate the context management overheads when no deviation takes place. In the case of child-first scheduling, at $D = 0\%$ *C-RoC* outperformed *C-Full* on KNL, POWER8, and ARM64 by reducing context-switching overheads, while it degraded performance on Skylake because of complex control as we discussed in Section 3.2.2. LSA and ESR (*-L* and *-E*) mitigated cache misses at the cost of additional stack management overheads, with elevated performance overall. The performance of *SS-L* and *SC* was close to that of *RtC*, but these threading techniques have programming constraints as discussed in Section 3.

On the other hand, at a larger D, *SC*, *SS*, *RoC*, and *C-RoC* were slower than the traditional fully fledged techniques (*Full* and *C-Full*) because the dynamic promotion techniques lose their advantages and become extra overheads. LSA and ESR (*-L* and *-E*) further degraded the performance since they no longer promote stack reuse and result in additional overheads to manage thread descriptors and stacks independently.

Although there is a fundamental scheduling difference between parent- and child-first techniques, the results of corresponding techniques (e.g., *RoC* and *C-RoC*) are similar because the expensive context-switching operations are common; in terms of context-switching overheads, there is no significant performance difference between parent- and child-first scheduling, while the applicability of dynamic promotion techniques is limited in the child-first cases. We note that, as pointed out by vast literature (e.g., [40], [41], and [42]), the scheduling policies have been known to affect the application performance. KMeans and ExaFMM in our evaluation showcase the difference in application-level performance, while in both cases the dynamic promotion methods enhance performance by reducing threading overheads.

4.1.2 Performance With Different Numbers of ULTs

With the same microbenchmark, we examined the effect of the dynamic promotion techniques by varying ULT counts (N) while fixing D to 0 and 100 percent. Fig. 23 shows fork-join overheads with different N. Overall the performance trend is the same; at $D = 0\%$ the dynamic promotion techniques (*SC*, *SS*, *RoC*, and *C-RoC*) are faster than fully fledged techniques (*Full* and *C-Full*). With smaller N, however, LSA and ESR (*-L* and *-E*) are insignificant because stack accesses hit caches without LSA and ESR; on the contrary, decoupling the management of thread descriptors and stacks negatively affects the performance even at $D = 0\%$. The results indicate that if only few ULTs are used in the runtime, LSA and ESR do not contribute to performance improvement and possibly just impose additional overheads.

TABLE 3
Experimental Environments Used in the Paper

Name	Skylake	KNL	POWER8	ARM64
Processor	Intel Xeon Platinum 8180M	Intel Xeon Phi 7210	IBM S822LC (10 cores)	AMD Opteron A1120
Architecture	Skylake	Knights Landing	POWER8 LE	ARMv8-A
Frequency	2.5 GHz	1.3 GHz	2.9 GHz	1.7 GHz
# of sockets	2	1	2	1
# of cores	56	64	20	4
# of HWTs	112	256	160	4
Memory	396 GB	198 GB	130 GB	8 GB
OS	Red Hat 7.5	Red Hat 7.5	Red Hat 7.6	openSUSE 42.2
Compilers	Intel Compiler 17.2.174	Intel Compiler 17.2.174	IBM XL Compilers 16.1.1	GNU Compilers 4.8.5

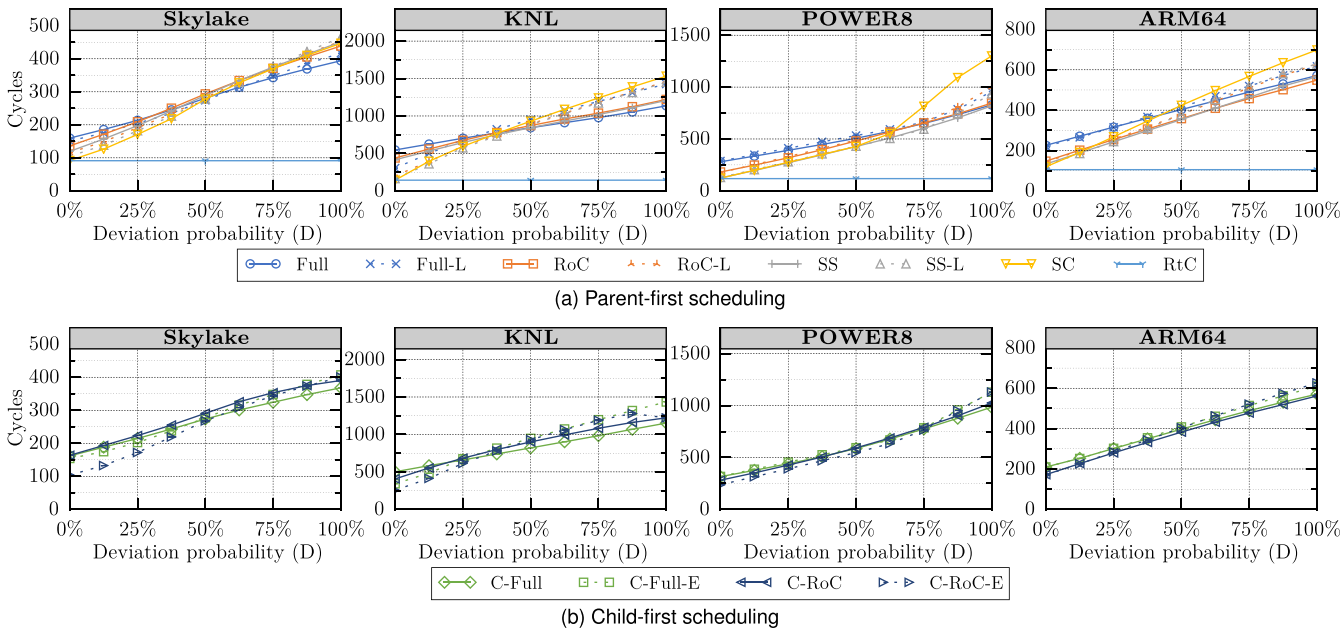


Fig. 22. Cycles per fork-join with various D values ($N = 4{,}096$).

4.1.3 Performance on Different Architectures

The effectiveness of the dynamic promotion techniques varies on different architectures. Figs. 22a and 22b indicate that KNL shows the largest performance difference at $D = 0\%$; the gaps between *Full* and *RtC* are 1.7x, 3.8x, 2.4x, and 2.2x while speedups of *C-RoC-E* over *C-Full* are 1.4x, 2.0x, 1.3x, and 1.3x on Skylake, KNL, POWER8, and ARM64, respectively. This difference comes from the design of KNL. In comparison with Skylake, which is a general-purpose Intel CPU, in spite of the identical calling convention, KNL showed a larger gap because of its throughput-oriented architecture; KNL poorly performs pointer-based operations with many branches and noncontiguous memory accesses, both of which highly impact the context-switching performance. The context-switching overhead on POWER8 is also high. For example, at $D = 0\%$ and $N = 128$, the performance gap between *Full* and *RtC* is 2.7x while the gaps of Skylake, KNL, and ARM64 are between 1.9x and 2.2x. Context switching on POWER8 is costly because more instructions are required to save its larger context; the context size of POWER8 is as large as 528 bytes because its ABI marks more registers as callee-saved [31]. In contrast, the size of x86/64 and ARMv8-A is only 64 bytes and 176 bytes, respectively [26], [30]. We observe that the dynamic promotion techniques are more effective on architectures that are throughput oriented or have a large context.

4.1.4 Memory Usage With Different Deviation Probabilities

We measured the memory usage of each threading technique to evaluate the impact of stack reuse. We ran the same benchmark with various D values where N is fixed to 65,536. Fig. 24 shows the maximum memory consumption on Skylake obtained with `ru_maxrss` of `getrusage()`. Results of the other machines were almost identical and hence are omitted. Fig. 24 shows that at smaller D threading techniques with LSA and ESR (*Full-L*, *RoC-L*, *SS-L*, *C-Full-E*, and *C-RoC-E*) significantly reduced the memory footprint by reusing a stack, as do *RtC* and *SC*. On the other hand, methods without stack reuse always consumed about 1 GB (equal to the stack size (16 KB) multiplied by $N = 65{,}536$). Although the modern hardware has abundant memory resources, this difference can be more significant when the stack size is set to a large value or more ULTs are created. In such a case, dynamic promotion techniques with LSA and ESR would be beneficial.

We evaluated the overheads of the threading techniques with a microbenchmark. The following evaluates the benefits of the dynamic promotion techniques with three fine-grained parallel applications: KMeans, ExaFMM, and Graph500. These applications utilize a context switch to efficiently schedule other ready work when currently running ULTs need to wait for locks, completions of other ULTs, or communications.

4.2 OpenMP-Parallelized KMeans

OpenMP is one of the most widely used parallel programming systems for multithreading. OpenMP offers threads and tasks as yieldable parallel units (i.e., `barrier` and `taskyield`)[10], so they were created as fully fledged threads in ULT-based OpenMP systems [6], [46], [47]. However, not all OpenMP threads and tasks encounter deviations in real programs (e.g., no task scheduling during execution). The dynamic promotion techniques are expected to improve performance when deviations rarely happen. We used OpenMP-parallelized KMeans for evaluation.

KMeans is a machine learning algorithm that partitions N data points into K clusters. Our benchmark is based on the KMeans implementation found in NU-MineBench [48]. In the KMeans algorithm, a point is considered belonging to a cluster with the nearest center. The algorithm first

10. The specification allows no operation for `taskyield` [43], while several studies pointed out the usefulness of yieldable tasks [44], [45]. In this benchmark, we assume tasks yield at `taskyield`.

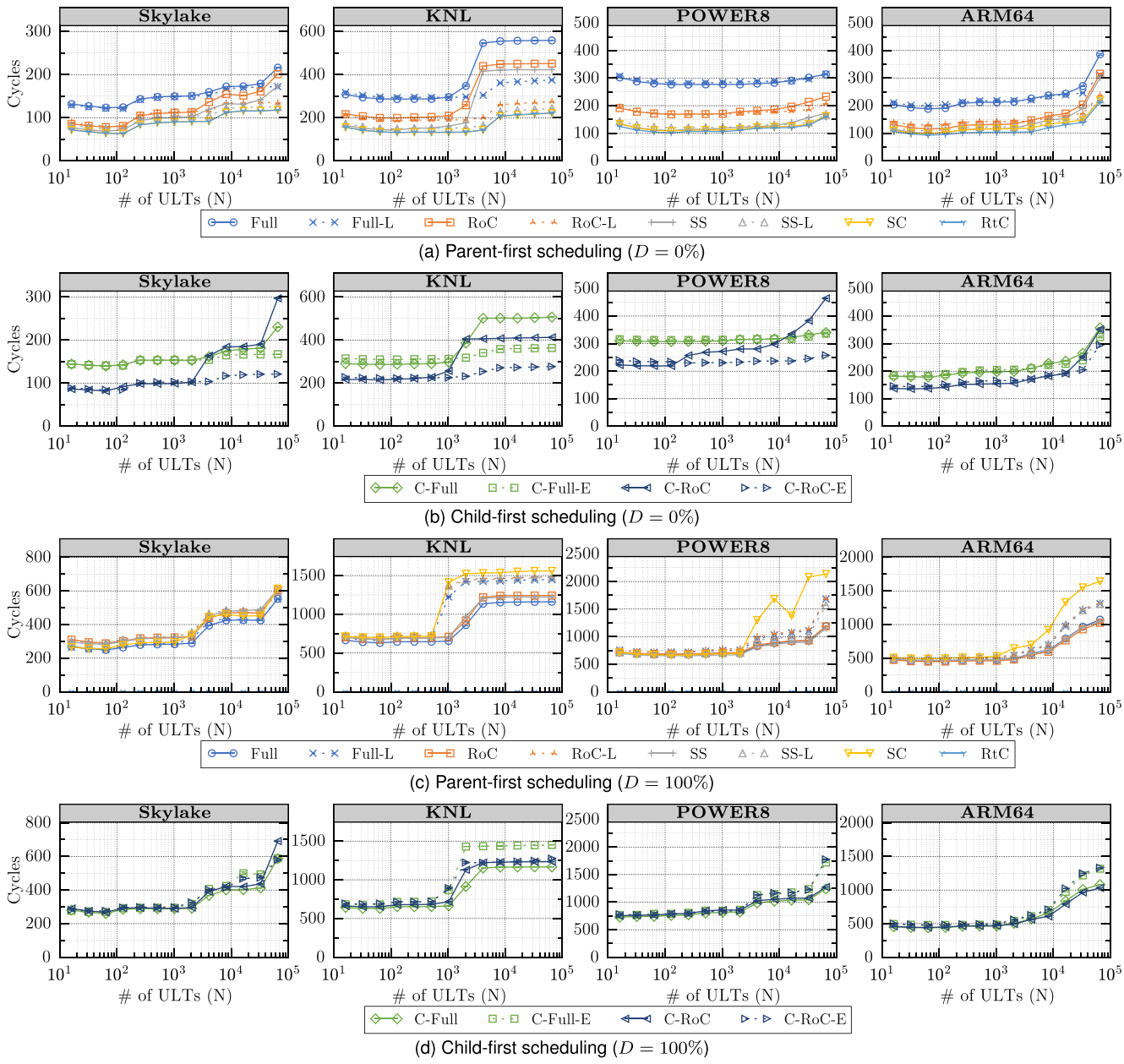


Fig. 23. Cycles per fork-join with various numbers of ULTs (D is fixed to either 0 percent or 100 percent).

randomly distributes each center of K clusters and repeats updating the cluster centers to the centroids of their points until the positions of the centers get stable enough. The computation of the new centroids is parallelized by a simple method adopted by Chabbi *et al.* [49]; in our benchmark, each of N ULTs is associated with a data point and updates the partial sum of the centroid of the nearest cluster. At the end of an iteration a master ULT sums up the partial results. The partial sums are shared among workers, so the updates are protected by locks to avoid data race.

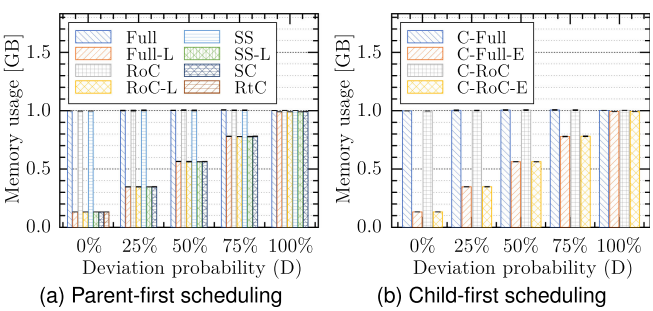


Fig. 24. Total memory consumption on Skylake where $N = 65,536$.

To control the lock granularity, we artificially change the number of replications per cluster, which we denote by r. When $r = 1$, each cluster has one partial sum protected by a corresponding lock, so any attempt to update the partial sum of the same cluster incurs lock contention. Creating multiple partial sums increases the reduction cost at the end of iterations but alleviates contention. When $r > 1$, every cluster has r partial sums each of which is accessed by only r/W workers, where W is the number of workers. Accordingly, no contention occurs if $r = W$.

The kernel of KMeans was parallelized with OpenMP by a nested parallel loop; the outer loop creates W OpenMP

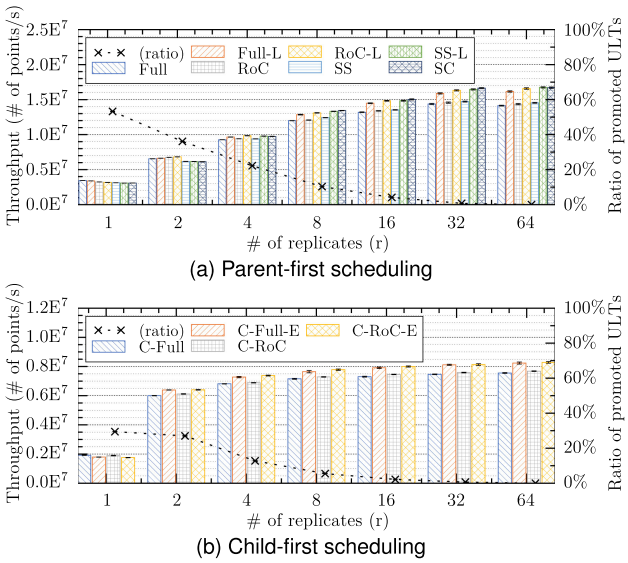


Fig. 25. Throughput of KMeans using 64 cores. The ratio of promoted ULTs is calculated by dividing the number of promoted ULTs by the number of created ULTs during execution. We obtain these results with *SC* for parent-first scheduling and *C-RoC-E* for child-first scheduling. Other dynamic promotion techniques show similar results.

threads each of which spawns N/W tasks in the inner loop. An Argobots-based OpenMP runtime library called BOLT [50] maps OpenMP threads and tasks to ULTs. We used KNL for the evaluation, so W was set to 64 in this benchmark. We built the program with Intel compilers,[11] while we needed to apply manual vectorization to the compute kernel to exploit SIMD units in KNL. We used the first 10 percent data of KDD Cup 1999 [51]; our experiment classifies $N = 5.0 \times 10^5$ points, each of which has 41 floating-point features,[12] into $K = 24$ clusters as instructed by the original problem statement. We changed r from 1 to 64 and measured the performance with different ULT types.

To exploit better locality, we set the OpenMP's `close` affinity for the parent-first threading techniques. The affinity of ULTs can be implemented by limiting the access of a specific pool; BOLT implements the OpenMP's affinity by limiting ULTs associated with OpenMP threads to be scheduled by a specific worker associated with a specific core [50]. Although this strategy works well in a parent-first case, such an affinity setting inhibits dynamic load balancing in a child-first case. Consider a case where r is 64 and no deviation happens in innermost ULTs (an inner OpenMP tasks). Under the child-first scheduling policy, not a child ULT but a parent ULT (i.e., an outer OpenMP thread) is made stealable. Because of the affinity setting, however, a parent ULT cannot be scheduled other than by a specific worker, disabling dynamic load balancing across workers. Thus, we disabled the affinity setting for the child-first threading techniques.

Fig. 25 shows the average throughputs of 64 executions each of which repeats the KMeans algorithm five times after a warm-up (one execution). An increase in replicates alleviates lock contention and reduces the deviation probability, elevating overall performance. At a larger r, LSA and ESR

11. BOLT is a runtime library compatible with LLVM and Intel OpenMP compilers, so the compiler modification is unnecessary.

12. We arbitrarily map string-typed values to floating-point values.

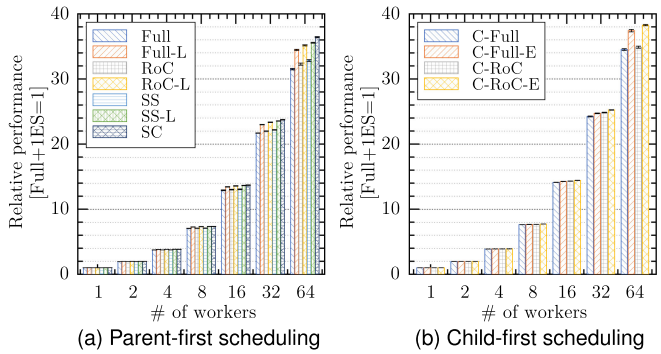


Fig. 26. Relative performance of ExaFMM on KNL. The baseline is the performance of *Full* with a single worker.

(*-L* and *-E*) enhances performance. Reducing context-switching overheads (*SC*, *SS-L*, *RoC-L*, and *C-RoC-E*) further improves throughputs. With fewer replicates, fully fledged techniques perform better, but the absolute performance is worse because of significant lock contention. The results show that the dynamic promotion techniques speed up programs if deviations happen infrequently, whereas the threading overheads often get negligible when deviations are frequent because the causes of deviation become the performance bottleneck. We also note that although the dynamic promotion methods improve performance with both parent- and child-first scheduling policies, the KMeans algorithm prefers parent-first scheduling because it suits the OpenMP's affinity setting.

4.3 ExaFMM

ExaFMM [52] is a highly optimized $O(N)$ N-Body solver using a fast multiple method. In the kernel, a tree is traversed in a divide-and-conquer manner, and leaf nodes calculate the actual forces. Recursive divide-and-conquer task parallelism has been known to efficiently parallelize the ExaFMM kernel [53]. Deviations can happen while waiting for completion of child ULTs, but they never occur in leaf nodes because they just perform computation without synchronization. The most efficient solution seems mapping leaf nodes to *RtC* and internal nodes to non-*RtC* ULTs. However, this optimization requires identifying leaf ULTs on creation, which is not only cumbersome but also expensive if the leaf condition is complicated. The dynamic promotion techniques alleviate the programmers' burden without hurting performance.

We ran ExaFMM ten times on KNL with `-ncrit 16 -t 0.15 -P 4 -dual -n 524288` as arguments. As the number of workers changed, we adjusted `-nspawn` to keep the number of created ULTs per worker constant (within 5 percent of error) while `-nspawn 256` was set with 64 workers. We measured the performance of the tree traversal where the program spends more than 90 percent of the total execution time. We manually vectorized the compute kernels to efficiently utilize SIMD units in KNL. To reduce internal nodes, we changed the way of work decomposition and collapsed internal nodes in the traverse tree while keeping the computation of leaf nodes.

Fig. 26 presents the performance of ExaFMM with different numbers of workers. Since the deviation probability is

TABLE 4
Experimental Environment of Graph500

Processor	Intel Xeon Phi 7230	Architecture	Knights Landing
Frequency	1.3 GHz	# of sockets	1
# of cores	64	# of HWTs	128
Memory	99 GB	OS	CentOS 7.6
Compilers	Intel Compiler 17.0.4	Interconnect	Intel Omni-Path

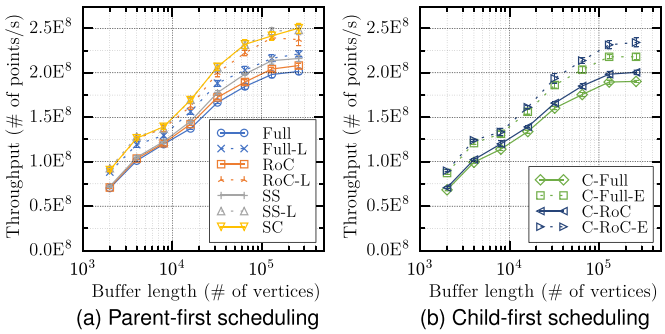
low (regardless of the number of workers, approximately 1.5 percent of ULTs are dynamically promoted), the dynamic promotion techniques achieved better performance. LSA and ESR (*-L* and *-E*) improved performance with 64 workers. Reduction of context-switching overheads contributes to a 9 percent speedup for parent-first (*SC* over *Full-L*) and 2 percent for child-first scheduling (*C-RoC-E* over *C-Full-E*). This ExaFMM showcases the merit of child-first scheduling; the child-first methods overall perform better than the parent-first ones because child-first scheduling can efficiently exploit locality when parallelism is deep and narrow [12].

We note that the dynamic promotion techniques are suitable for divide-and-conquer recursive parallelism; in a k-ary tree approximately $1 - \frac{1}{k}\%$ of ULTs are leaves;[13] and therefore if no deviation happens in leaf ULTs (e.g., no yielding), D is approximately $\frac{1}{k}\%$. The results in Fig. 22 show that the dynamic promotion techniques perform better than the fully fledged threads when D is less than 30 to 50 percent. Because D is at most 50 percent ($k = 2$), the dynamic promotion techniques are beneficial in most cases.

4.4 Distributed Graph500

Graph500 [54] is a well-known benchmark that traverses a distributed graph in a breadth-first manner. Our Graph500 is based on the reference implementation of MPI+Thread found in [55]. Since each process has only a part of the whole graph, interprocess communication is necessary in order to visit vertices in remote subgraphs. Specifically, in each iteration, every process repeats visiting adjacent vertices. A process can update a vertex if locally owned, while it needs to send a message to another process if the vertex exists in a remote node. In order to avoid the finest communication, message aggregation is commonly adopted. Each process has buffers associated with all the target ranks to store visit messages; messages are sent only when a buffer gets full and thus needs to be flushed. Because the bottleneck of Graph500 is communication, hybrid parallelism is often used to reduce the intranode communication overheads. MPI+Thread, where ULT is used as Thread, has been studied to exploit fine-grained communication on a distributed system [56], [57], because a ULT can efficiently switch to another ULT when an MPI function blocks. We note that the current state-of-the-art MPI+Thread implementation [58] sometimes acquires a lock even in nonblocking MPI calls (e.g., `MPI_Isend()` and `MPI_Test()`) since MPI functions need to periodically handle global progress for active messages and nonblocking collectives, which is typically protected by a global lock. The dynamic promotion techniques are expected to reduce overheads in cases where ULTs do not call an MPI function or happen to avoid lock contentions in the MPI runtime.

13. Denote the number of internal nodes in a task tree N and the number of leaf nodes n. When $N = 1$, (N, n) is $(1, k)$. Because 1 leaf can be replaced with 1 internal node and k leaves, (N, n) becomes $(N, N(k-1)+1)$. Hence, the ratio of leaf ULTs is calculated as $\frac{n}{N+n} \approx 1 - \frac{1}{k}$ with a larger N.



(a) Parent-first scheduling (b) Child-first scheduling

Fig. 27. Traversed edges per second of Graph500 using 1,024 cores.

We parallelized the Graph500 implementation in [55] with Argobots and evaluated the performance over Argobots-aware MPICH [58]. We associated one visit with a ULT to focus on threading overheads. A buffer length B is a parameter to control the communication granularity; with a larger B, more memory is consumed, but more messages are aggregated. We set a scale factor to 26, so the whole graph over nodes consists of 2^{26} vertices. We executed this benchmark on 16 KNLs described in Table 4. We spawned a single MPI process per node, each of which ran 64 workers, so 1,024 workers were used in total. Fig. 27 shows the averages of ten times execution. In this setting, the ratio of promoted ULTs is less than 1.0 percent, highlighting the efficacy of the dynamic promotion techniques. The fully fledged (*Full* and *C-Full*) techniques should perform better with extremely small B and higher deviation probability, while the communication overheads would mask their benefit.

4.5 How to Choose the Best Threading Technique

This paper investigated several user-level threading techniques that have different performance characteristics and programming constraints. As all the threading techniques can coexist in a single library, users and developers can choose the suitable techniques from Table 1 for their assuming workloads. Practically, the first decision should be a choice of either parent- or child-first scheduling. As demonstrated in our evaluation, parent-first scheduling tends to perform better for shallow parallelism (e.g., loop parallelism in KMeans) while a child-first scheduling is preferred when the parallelism is deep and nested (e.g., divide-and-conquer parallelism in ExaFMM). More sophisticated strategy would be a mixed scheduling policy [59], which is out of scope of this work. The optimal technique under a specific scheduling policy should be chosen based on scenarios regarding the number of created ULTs, typical deviation probability, and required threading capabilities, all of which depend on their algorithms, machines, and inputs.

If the user has no idea about the application behavior, we recommend *RoC-L* for parent-first scheduling or *C-RoC-E* for child-first scheduling; they perform well at low deviation probability with minimum memory footprint while retaining all the threading capabilities. Nevertheless, among fully capable threads, these two do not always perform the best. When

deviation probability is high, they are slower than *Full* and *C-Full*. As we have seen in the evaluation, however, when deviations are frequent, threading overheads often become negligible because events that cause deviations (e.g., lock contention and blocking communication) hide the benefit of lightweight threads. *RoC* and *C-RoC* also outperform *RoC-L* and *C-RoC-E* when deviations rarely happen, and fewer ULTs are used because LSA and ESR are not effective when stack accesses hit caches. Although our microbenchmark uses an empty function for a thread function, real thread functions are likely to require larger function stacks for computation, rendering LSA and ESR more beneficial. We recommend *RoC-L* or *C-RoC-E* in general, but the most promising approach is the automatic selection of the best threading techniques, which is one direction of our future work.

5 Related Work

Although numerous parallel systems have adopted ULTs as an implementation of lightweight parallel units, the focus of the past papers on ULT-based systems is not a threading technique but other components such as programmability, usability, portability, abstraction, and other performance optimizations such as scheduling and thread pool implementations. The performance comparison between these parallel systems measured only the overall performance [60], [61] and fundamentally lacked a detailed performance analysis of threading techniques. This section describes notable work out of countless studies on ULTs from the aspect of user-level threading techniques.

Fully Fledged Threads. Fully fledged threads are widely used to implement ULTs with full threading capabilities. For example, Qthreads [4], Nanos++ [5], Converse [7], MassiveThreads [10], and Argobots [11] are well-known threading libraries that use fully fledged threads. Their stack management policies are different, however. For example, Converse 6.9.0, MassiveThreads 1.00, and Argobots 1.0rc2 employ a parent-first fully fledged ULT without LSA while Qthreads 1.15 and Nanos++ 0.15 implement it with LSA. MassiveThreads 1.00 also supports a child-first thread, which is implemented with ESR. The trade-off disclosed by this paper would be helpful for these runtimes to choose the best thread implementation based on their assuming workloads. Evaluating the performance impact of choosing the optimal technique in these libraries is lifted as our future work.

Saving Registers. We did not evaluate techniques that do not change stacks but only save registers, but some previous studies including LazyThreads [36], StackThreads/MP [37], [38], and Fibril [39] proposed such techniques for child-first scheduling. Cilk 1.0-3 over Tapir/LLVM [62] is an actively developed multitasking framework that adopts this idea. Their approaches assume the following premises:

1. All local variables in the stack are addressed by a frame pointer instead of a stack pointer.
2. The call stack is not dynamically grown after a function prologue.
3. All threads are joined in a function that creates them (i.e., fully strict computation [22]).

Under these premises, a parent can call a child function on top of the parent stack after saving (or clobbering) callee-saved registers. The algorithm works as follows. If no work stealing happens, the child just returns to the parent. When another worker steals the parent ULT, the thief worker restores the original registers while setting a newly allocated stack to a stack pointer and resumes the parent on top of the new stack. Premise 1 guarantees that spaces for all local variables have already been allocated or reserved before the child invocation and these locations are referenced by a frame pointer. Premise 2 assures no stack growth, so a parent thread will not erode the stack used by the child thread. We note that premise 2 allows function calls because the stack address of a new function is based on a stack pointer. Premise 3, which narrows the expressiveness of parallelization, is required in order to prevent the caller of the parent ULT from overwriting the stack of the child thread prior to the completion of a child.

However, premises 1 and 2 require compiler modifications, and therefore StackThreads/MP [38], LazyThreads [36], and Cilk [20] modified a compiler. Yang and Mellor-Crummey [39] tried to avoid compiler modifications by adding a GCC compiler flag, `-fno-omit-frame-pointer`, but it does not guarantee premise 1. Unfortunately, the current popular C compilers do not provide a flag that guarantees premises 1 and 2. These techniques are not evaluated in this paper because our work targets threading techniques without compiler modifications.

Stack Separation. Some studies have proposed methods that omit register manipulations but change only the stacks. Their approaches are different from ours in that a thread invocation function adopts a special calling convention that only marks registers for a stack pointer and an instruction address as callee-saved (e.g., Intel CilkPlus [63]). This approach can be seen as a technique that utilizes a calling convention to save all the necessary registers (including registers marked as callee-saved in widely adopted ABIs). This approach requires patching a compiler to recognize a custom calling convention, whereas neither *SS* nor *SS-L* requires compiler modification.

Scheduler Creation. A few parallel systems have adopted the scheduler creation technique. Chores [32] and Wool [33] are parent-first threading libraries that utilized this method to reduce threading overheads. Concurrent Cilk [34] is a child-first threading library that adopted this technique to implement a yield feature in Intel CilkPlus [64]. The past work, however, solely implemented the scheduler creation technique and thus lacked performance comparison and analysis of programming constraints. We also note that their approaches specially handle ULTs that encountered deviations, so promoted ULTs are differently scheduled from unpromoted ULTs. Our techniques uniformly schedule all ULTs including promoted *SC* threads.

Run-to-Completion Threads. Numerous runtime systems including Filaments [9], Qthreads [4],[14] and Argobots [11] support a run-to-completion thread in order to eliminate all the cost associated with user-level context switch. OpenMP *task* implementations found in the popular OpenMP runtimes [1], [2], [3] and `task` in Intel TBB [65] are essentially classified as *RtC* threads but not "run to completion" in a narrow sense because they can wait for the completion of children.

14. Qthreads executes a thread on top of the scheduler's stack when `QTHREAD_SPAWN_SIMPLE` is specified. No blocking operation (i.e., `qthread_yield()`) inside threads is allowed.

In general, *RtC* is lightweight and easy to implement, but its constraint significantly limits the applicability because it cannot perform a context switch at an arbitrary point. Several papers have argued yieldable threads in non-yieldable threading packages from performance and programmability perspectives. For example, Zakian et al. [34] showed that a yield operation in Cilk [20] enables efficient blocking communication and synchronization while several papers on OpenMP [44], [45] reported the same benefits of yieldable OpenMP tasks. Graph500 in our evaluation is a good example that *RtC* cannot execute; removing a yield operation from a polling loop in the MPI runtime might cause a deadlock.

Other Threading Techniques. Several threading techniques cannot be classified into the categories above. Sivaramakrishnan *et al*. [66] proposed MultiMLton, which allows relocation of function stacks. This technique might be applicable to functional languages, but it can hardly support C/C++ programs. Cilk-M [21] enables stack copying by modifying OS to expose the same address space so that a pointer reference to a call stack is valid after copying a stack. This technique requires OS modification. Tascell [67] is a compiler-based technique adopting a lazy task creation policy. This technique invokes threads in a sequential manner and lazily creates logical threads if necessary by backtracking call stacks. Acar *et al*. [18] propose a threading technique that lazily creates parallel threads at a *heartbeat*. This method requires the cactus stack implementation [36], which breaks the interoperability with precompiled libraries and thus is not suitable for a generic threading library.

6 Concluding Remarks

This work extensively explores user-level threading techniques that are suitable for threading libraries from the viewpoint of threading overheads. Our in-depth instruction- and cache-level analysis of twelve methods revealed their performance characteristics and programming constraints. We found that deviation inhibits the run-to-completion execution of thread and highly impacts fork-join overheads. We implemented all the techniques in the same runtime system and evaluated fork-join overheads on Skylake, KNL, POWER8, and ARM64 architectures. Our evaluation with a microbenchmark and three fine-grained applications indicates that the dynamic promotion techniques that defer the context management show the best trade-off between fork-join overheads and programming constraints when the chances of deviation are low.

Our quest is a comprehensive understanding of the design and implementation for lightweight threading libraries. This work solely investigates fork-join performance. Arguably, other factors including schedulers and thread pools are known to highly affect the overall performance. Investigating their design and performance is our future work.

Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, in particular, its subproject on Scaling OpenMP with LLVm for Exascale performance and portability (SOLLVE). The authors gratefully acknowledge the computing resources provided and operated by LCRC and JLSE at Argonne National Laboratory. This article is based upon work supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357.

References

[1] D. Novillo, "OpenMP and automatic parallelization in GCC," in *Proc. GCC Developers' Summit*, 2006, pp. 23–24.
[2] The LLVM Project. "LLVM OpenMP Runtime Library," 2015. [Online]. Available: https://openmp.llvm.org/Reference.pdf
[3] Intel® OpenMP* Runtime Library. 2016. [Online]. Available: https://www.openmprtl.org/
[4] K. B. Wheeler, R. C. Murphyand, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," in *Proc. 22nd IEEE Int. Parallel Distrib. Process. Symp.*, 2008, pp. 1–8.
[5] X. Martorell, J. Labarta, N. Navarro, and E. Ayguadé, "A library implementation of the nano-threads programming model," in *Proc. 2nd Eur. Conf. Parallel Process.*, 1996, pp. 644–649.
[6] A. Duran *et al*., "OmpSs: A proposal for programming heterogeneous multi-core architectures," *Parallel Process. Lett.*, vol. 21, no. 02, pp. 173–193, Mar. 2011.
[7] L. V. Kalé, J. Yelon, and T. Knauff, "Threads for interoperable parallel programming," in *Proc. 9th Int. Workshop Languages Compilers Parallel Comput.*, 1996, pp. 534–552.
[8] L. V. Kale and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," in *Proc. 8th Annu. Conf. Object-Oriented Program. Syst. Languages Appl.*, 1993, pp. 91–108.
[9] D. K. Lowenthal, V. W. Freeh, and G. R. Andrews, "Efficient support for fine-grain parallelism on shared-memory machines," *Concurrency: Practice Experience*, vol. 10, no. 3, pp. 157–173, Mar. 1998.
[10] J. Nakashima and K. Taura, "MassiveThreads: A thread library for high productivity languages," *Concurrent Objects Beyond*, vol. 8665, pp. 222–238, Jan. 2014.
[11] S. Seo *et al*., "Argobots: A lightweight low-level threading and tasking framework," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 3, pp. 512–526, Mar. 2018.
[12] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr., "Lazy task creation: A technique for increasing the granularity of parallel programs," in *Proc. ACM Conf. LISP Functional Program.*, 1990, pp. 185–197.
[13] S. Iwasaki, A. Amer, K. Taura, and P. Balaji, "Lessons learned from analyzing dynamic promotion for user-level threading," in *Proc. IEEE/ACM Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2018, pp. 23:1–23:12.
[14] D. Spoonhower, G. E. Blelloch, P. B. Gibbons, and R. Harper, "Beyond nested parallelism: Tight bounds on work-stealing overheads for parallel futures," in *Proc. 21st Annu. Symp. Parallelism Algorithms Archit.*, 2009, pp. 91–100.
[15] A. Duran, J. Corbalán, and E. Ayguadé, "An adaptive cut-off for task parallelism," in *Proc. IEEE/ACM Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2008, pp. 36:1–36:11.
[16] U. A. Acar, V. Aksenov, A. Charguéraud, and M. Rainey, "Provably and practically efficient granularity control," in *Proc. 24th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2019, pp. 214–228.
[17] A. Tzannes, G. C. Caragea, U. Vishkin, and R. Barua, "Lazy scheduling: A runtime adaptive scheduler for declarative parallelism," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 3, pp. 10:1–10:51, Sep. 2014.
[18] U. A. Acar, A. Charguéraud, A. Guatto, M. Rainey, and F. Sieczkowski, "Heartbeat scheduling: Provable efficiency for nested parallelism," in *Proc. 39th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2018, pp. 769–782.
[19] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, "Cooperative task management without manual stack management," in *Proc. General Track Annu. Conf. USENIX Annu. Tech. Conf.*, 2002, pp. 289–302.
[20] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 1998, pp. 212–223.
[21] I.-T. A. Lee, S. Boyd-Wickizer, Z. Huang, and C. E. Leiserson, "Using memory mapping to support cactus stacks in work-stealing runtime systems," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Techn.*, 2010, pp. 411–420.
[22] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proc. 5th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 1995, pp. 207–216.

[23] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999.

[24] Boost C++ Libraries. 2018. [Online]. Available: https://www.boost.org/

[25] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in *Proc. 12th Annu. ACM Symp. Parallel Algorithms Archit.*, 2000, pp. 1–12.

[26] J. Hubička, A. Jaeger, M. Matz, and M. Mitchell, "System V application binary interface AMD64 architecture processor supplement," Oct. 2013. [Online]. Available: https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

[27] Intel Corporation. 2016. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.html

[28] ARM Limited. 2015. [Online]. Available: https://developer.arm.com/docs/den0024/latest

[29] IBM Corporation. 2018. [Online]. Available: https://openpowerfoundation.org/?resource_lib=ibm-power-isa-version-2-07-b

[30] "Procedure call standard for the ARM 64-bit architecture," May 2013. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf

[31] "64-Bit ELF V2 ABI specification power architecture workgroup specification revision 1.4," May 2017. [Online]. Available: http://openpowerfoundation.org/wp-content/uploads/resources/leabi/leabi-20170510.pdf

[32] D. L. Eager and J. Jahorjan, "Chores: Enhanced run-time support for shared-memory parallel computing," *ACM Trans. Comput. Syst.*, vol. 11, no. 1, pp. 1–32, Feb. 1993.

[33] K.-F. Faxén, "Wool – A work stealing library," *ACM SIGARCH Comput. Archit. News*, vol. 36, no. 5, pp. 93–100, Jun. 2009.

[34] C. S. Zakian, T. A. Zakian, A. Kulkarni, B. Chamith, and R. R. Newton, "Concurrent cilk: Lazy promotion from tasks to threads in C/C++," in *Proc. Revised Selected Papers 28th Int. Workshop Languages Compilers Parallel Comput.*, 2016, pp. 73–90.

[35] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 3, pp. 189–204, Aug. 2000.

[36] S. C. Goldstein, K. E. Schauser, and D. E. Culler, "Lazy threads: Implementing a fast parallel call," *J. Parallel Distrib. Comput.*, vol. 37, no. 1, pp. 5–20, Aug. 1996.

[37] K. Taura and A. Yonezawa, "Fine-grain multithreading with minimal compiler support - a cost effective approach to implementing efficient multithreading languages," in *Proc. ACM SIGPLAN 1997 Conf. Program. Lang. Des. Implementation*, 1997, pp. 320–333.

[38] K. Taura, K. Tabata, and A. Yonezawa, "StackThreads/MP: Integrating futures into calling standards," in *Proc. 7th ACM SIGPLAN Symp. Princ. Practice Parallel Program.*, 1999, pp. 60–71.

[39] C. Yang and J. Mellor-Crummey, "A practical solution to the cactus stack problem," in *Proc. 28th ACM Symp. Parallelism Algorithms Archit.*, 2016, pp. 61–70.

[40] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2009, pp. 1–12.

[41] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins, "OpenMP task scheduling strategies for multicore NUMA systems," *Int. J. High Perform. Comput. Appl.*, vol. 26, no. 2, pp. 110–124, May 2012.

[42] A. Huynh, D. Thain, M. Pericàs, and K. Taura, "DAGViz: A DAG visualization tool for analyzing task-parallel program traces," in *Proc. 2nd Workshop Vis. Perform. Anal.*, 2015, pp. 1–8, Art no. 3.

[43] OpenMP Architecture Review Board, "OpenMP application program interface version 5.0," 2018. [Online]. Available: Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[44] E. Ayguadé *et al.*, "The design of OpenMP tasks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 3, pp. 404–418, Mar. 2009.

[45] J. Schuchart, K. Tsugane, J. Gracia, and M. Sato, "The impact of taskyield on the design of tasks communicating through MPI," in *Proc. 13th Int. Workshop OpenMP*, 2018, pp. 3–17.

[46] P. E. Hadjidoukas and V. V. Dimakopoulos, "Support and efficiency of nested parallelism in OpenMP implementations," in *Concurrent and Parallel Computing: Theory, Implementation and Applications*, 2008, pp. 185–204. [Online]. Available: http://www.novapublishers.org/catalog/product_info.php?products_id=6803

[47] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and R. Namyst, "ForestGOMP: An efficient OpenMP environment for NUMA architectures," *Int. J. Parallel Program.*, vol. 38, no. 5, pp. 418–439, Oct. 2010.

[48] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, "MineBench: A benchmark suite for data mining workloads," in *Proc. IEEE Int. Symp. Workload Characterization*, 2006, pp. 182–188.

[49] M. Chabbi, M. Fagan, and J. Mellor-Crummey, "High performance locks for multi-level NUMA systems," in *Proc. 20th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2015, pp. 215–226.

[50] S. Iwasaki, A. Amer, K. Taura, S. Seo, and P. Balaji, "BOLT: Optimizing OpenMP parallel regions with user-level threads," in *Proc. 28th Int. Conf. Parallel Archit. Compilation Techn.*, 2019, pp. 29–42.

[51] KDD Cup 1999 Data. 1999. [Online]. Available: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

[52] R. Yokota and L. A. Barba, "A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems," *Int. J. High Perform. Comput. Appl.*, vol. 26, no. 4, pp. 337–346, Nov. 2012.

[53] K. Taura, J. Nakashima, R. Yokota, and N. Maruyama, "A task parallelism meets fast multipole methods," in *Proc. 3rd Workshop Latest Advances Scalable Algorithms Large-Scale Syst.*, 2012, pp. 617–625.

[54] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, Cray User Group, 2010, pp. 1–5, Art no. 14C.

[55] A. Amer, H. Lu, P. Balaji, and S. Matsuoka, "Characterizing MPI and hybrid MPI+Threads applications at scale: Case study with BFS," in *Proc. 15th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2015, pp. 1075–1083.

[56] H. Lu, S. Seo, and P. Balaji, "MPI+ULT: Overlapping communication and computation with user-level threads," in *Proc. 17th Int. Conf. High Perform. Comput. Commun.*, 2015, pp. 444–454.

[57] T. Fukuoka, W. Endo, and K. Taura, "An efficient inter-node communication system with lightweight-thread scheduling," in *Proc. 21st Int. Conf. High Perform. Comput. Commun.*, 2019, pp. 687–696.

[58] MPICH | High-Performance Portable MPI. 2019. [Online]. Available: https://www.mpich.org/

[59] Q. Chen, M. Guo, and Z. Huang, "CATS: Cache aware task-stealing based on online profiling in multi-socket multi-core architectures," in *Proc. 26th ACM Int. Conf. Supercomputing*, 2012, pp. 163–172.

[60] A. Podobas, M. Brorsson, and K.-F. Faxén, "A comparative performance study of common and popular task-centric programming frameworks," *Concurrency Comput.: Pract. Experience*, vol. 27, no. 1, pp. 1–28, Jan. 2015.

[61] G. W. Price and D. K. Lowenthal, "A comparative analysis of fine-grain threads packages," *J. Parallel Distrib. Comput.*, vol. 63, no. 11, pp. 1050–1063, Nov. 2003.

[62] T. B. Schardl, W. S. Moses, and C. E. Leiserson, "Tapir: Embedding fork-join parallelism into LLVM's intermediate representation," in *Proc. 22nd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2017, pp. 249–265.

[63] "Intel® Cilk™ plus application binary interface specification," Dec. 2011. [Online]. Available: https://www.cilkplus.org/sites/default/files/open_specifications/CilkPlusABI_1.1.pdf

[64] C. E. Leiserson, "The Cilk++ concurrency platform," in *Proc. 46th Annu. Des. Autom. Conf.*, 2009, pp. 522–527.

[65] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. Newton, MA, USA: O'Reilly Media, Jul. 2007.

[66] K. Sivaramakrishnan, L. Ziarek, R. Prasad, and S. Jagannathan, "Lightweight asynchrony using parasitic threads," in *Proc. 5th ACM SIGPLAN Workshop Declarative Aspects Multicore Program.*, 2010, pp. 63–72.

[67] T. Hiraishi, M. Yasugi, S. Umatani, and T. Yuasa, "Backtracking-based load balancing," in *Proc. 14th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2009, pp. 55–64.

Shintaro Iwasaki (Student Member, IEEE) received the BS and MS degrees from the University of Tokyo, in 2015 and 2017, respectively. He is working toward the PhD degree at the University of Tokyo. He is also a predoctoral appointee at Argonne National Laboratory. His research interests include parallel programming languages, compiler optimizations, parallel runtime systems, and scheduling techniques.

Abdelhalim Amer (Member, IEEE) is currently an assistant computer scientist with the Mathematics and Computer Science Division at Argonne National Laboratory. His research falls generally under the parallel and distributed computing landscape. More specifically, his focus is on communication runtimes and their interaction with threading models on massively parallel systems.

Kenjiro Taura (Member, IEEE) received the BS, MS, and DSc degrees from the University of Tokyo, in 1992, 1994, and 1997, respectively. He is currently a professor with the Department of Information and Communication Engineering, University of Tokyo. His major research interests include centered on parallel/distributed computing and programming languages. His expertise includes efficient dynamic load balancing, parallel and distributed garbage collection, and parallel/distributed workflow systems.

Pavan Balaji (Member, IEEE) holds an appointments as a computer scientist and group lead at Argonne National Laboratory, where he leads the Programming Models and Runtime Systems group. His research interests include parallel programming models and runtime systems for communication and I/O on extreme-scale supercomputing systems, modern system architecture, cloud computing systems, data-intensive computing, and big-data sciences.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

T-BASIR: Finding Shutdown Bugs for Cloud-Based Applications in Cloud Spot Markets

Abdullah Alourani, *Student Member, IEEE*,
Ajay D. Kshemkalyani, *Senior Member, IEEE*, and Mark Grechanik, *Senior Member, IEEE*

Abstract—One of the major advantages of cloud spot instances in cloud computing is to allow stakeholders to economically deploy their applications at much lower costs than that of other types of cloud instances. In exchange, spot instances are often exposed to revocations (i.e., terminations) by cloud providers. With spot instances becoming pervasive, terminations have become a part of the normal behavior of cloud-based applications; thus, these applications may be left in an incorrect state leading to certain bugs. Unfortunately, these applications are not designed or tested to deal with this behavior in the cloud environment, and as a result, the advantages of cloud spot instances could be significantly minimized or even entirely negated. We propose a novel solution to automatically find these bugs and locate their causes in the source code. We evaluate our solution using 10 popular open-source applications. The results show that our solution not only finds more instances and different types of these bugs compared to the random approach, but it also locates the causes of these bugs to help developers improve the design of the shutdown process and is more efficient in finding instances of these bugs since it interposes at the system call layer.

Index Terms—Cloud computing, cloud spot markets, shutdown bugs of cloud-based applications, kernel modules, irregular terminations of cloud-based applications, spot instance revocations

1 Introduction

Cloud computing enables cloud customers to rent resources (e.g., virtual machines (VMs)) on as-needed basis to run their applications. That is, cloud customers do not have to buy and host expensive hardware to run their applications, and instead they rent resources for their applications from cloud computing facilities. This is an essential difference between cloud computing systems and distributed systems, which require application owners to buy and host expensive hardware to run their applications. As the deployment cost is an integral part of applications deployed on the cloud, the cost-efficiency of provisioning resource to these applications becomes a priority, and it is of growing significance, since the total spending that will be affected by cloud computing is over $1 trillion by 2020 [1].

Many cloud providers such as Amazon Web Services offer four types of instances (i.e., VMs) [2]: on-demand, reserved, dedicated, and spot (also known as preemptible). Cloud customers can pay for renting on-demand instances per hour without long-term commitments, and they cost the most. Also, they can rent reserved instances for a long term by making an upfront payment to cloud providers and thus pay a much lower rate than on-demand instances. A variation of reserved instances is a dedicated host, which is a physical server that is assigned only to a specific customer, and nobody besides this customer can use the resources of this host. Unlike the fixed-cost paying schemes mentioned above, a variable-cost paying scheme allows cloud customers to specify the price they are willing to pay for renting a spot instance to run their applications [2], and, depending on the varying demand from cloud customers, the price of this spot instance can go up if the demand increases and the number of available instances that can be supported by a finite number of physical resources in a data center of cloud providers decreases [3]. Conversely, the price of this spot instance can go down if the demand decreases and the number of available instances increases. If the customer's price is greater than the cloud provider's price that depends on the demand, a spot instance will be provisioned to customers' applications at the customer's price. However, when spot instances are already provisioned to customer applications and the cloud provider's price goes above the customer's price, the cloud providers will revoke those spot instances within two minutes by sending termination signals, thus resulting in *revocations of those spot instances* [3], whose occurrences are very difficult to predict [4]. As a result, even though cloud customers sometimes rent spot instances at 90 percent lower costs compared to on-demand [2], their applications that run in spot instances can be terminated based on price fluctuations that happen frequently, thus these applications may switch to an incorrect state leading to certain bugs [5], [6].

In general, terminations could be seen as *regular* when an application receives a termination signal in the context of predefined protocols, or *irregular* when an application receives a termination signal without using any context of predefined

- *A. Alourani is with the Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607, and also with Majmaah University, Al Majma'ah 15341, Saudi Arabia. E-mail: aalour2@uic.edu.*
- *A. D. Kshemkalyani and M. Grechanik are with the Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607. E-mail: {ajay, drmark}@uic.edu.*

Manuscript received 29 Oct. 2019; revised 19 Feb. 2020; accepted 7 Mar. 2020. Date of publication 13 Mar. 2020; date of current version 7 May 2020. (Corresponding author: Abdullah Alourani.)
Recommended for acceptance by Q. Zheng.
Digital Object Identifier no. 10.1109/TPDS.2020.2980265

1045-9219 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

protocols. Hence, the revocations of spot instances often lead to irregular terminations of cloud-based applications. Note that an application can be irregularly terminated in two modes. We assume that the reason for executing an application is to run an algorithm that implements the requirements of this application to provide the required results. First, an application could be irregularly terminated during the execution of the application's algorithm. Second, an application could be irregularly terminated during the execution of the shutdown sequence of the application when the execution of the application's algorithm is completed. Moreover, irregular terminations do not affect stateless applications but often affect stateful applications relying on the results of ongoing calculation by applications under irregular terminations. These stateful applications might change to incorrect states when they are terminated before their shutdown sequences are entirely executed. In general, resources utilized by an application under irregular termination can be called ***R**esources **A**ffected by **T**ermination (RAT)*. When an application (A) encounters irregular terminations while interacting with another application (B), B is considered RAT because it might be left in an incorrect state until it identifies that A is already terminated.

EC2 spot markets contain approximately 7,600 independent spot prices for different types of instances among 44 availability zones (i.e., data centers) in 16 regions [7]. With spot instances becoming pervasive, irregular terminations have become a part of the normal behavior of cloud-based applications. ***B**ugs of cloud-based **A**pplications resulting from **S**pot **I**nstance **R**evocations (BASIR)* result from errors in the implementation of the shutdown instructions of these applications that occur only during spot instance revocations. When these applications are being irregularly terminated, they might lose their states that lead to BASIR, such as data loss, inconsistent states, performance bottlenecks, hangs, crashes, deadlocks, locked resources, or these applications that cannot restart/terminate. On top of poor user experience from seeing these bugs, other bugs result in situations where cloud-based applications could not be restarted without manual interventions. In finer detail, when an application invokes synchronization system calls to lock a file and perform an update on the file inode's field that specifies if the file shall be persisted and this application is being irregularly terminated before the update is completed, system calls (i.e., fsync) of this application that are responsible for synchronizing the data of an open file to the storage device will become a "no-op", causing data loss of this file [8].

In general, heartbeat or timeout mechanisms might reduce the number of BASIR that require interaction between external processes (or threads) that run in different instances and an application's processes (or threads) run in a spot instance under irregular terminations, i.e., deadlocks, hangs, and performance bottlenecks. However, these mechanisms may not be useful for other types of BASIR that solely depend on ongoing calculations by applications deployed on a spot instance under irregular terminations, i.e., data corruption, data loss, crashes, and inconsistent states of shared data objects. Cloud-based applications that run in spot instances are not designed or tested to deal with this behavior in the cloud environment. The shutdown sequence of a cloud-based application is often left untested because developers often assume that a cloud-based application is properly terminated as long as its processes are terminated. It is very difficult to find BASIR because a termination signal can be initiated at every execution state of a cloud-based application, leading to a significantly larger search space of application states. Unfortunately, the absence of testing the effect of spot instance revocations on cloud-based applications will likely lead to a large number of BASIR. As a result, the advantages of cloud spot instances could be significantly minimized or even entirely negated.

In this paper, we propose a novel solution to automatically find BASIR and locate their causes in the source code of cloud-based applications. We develop our solution for ***T**esting for BASIR (`T-BASIR`)* that uses kernel modules (KMs) [9] to find these bugs and generate traces of their causes in the source code. `T-BASIR` is comprised of two major components. (1) Automating BASIR detection using KMs that contain the following main phases: (i) sending termination signals to certain system calls of a cloud-based application, and (ii) measuring the impacts on the state of RAT when the cloud-based application is irregularly terminated during the execution of these system calls. (2) Identifying the causes of BASIR using Tracer KM, which modifies the flow of executions through intercepting a termination signal to collect execution traces from the stack of a cloud-based application before the application receives the termination signal. BASIR and the traces of BASIR can be analyzed during application testing by developers, who look for fixes for BASIR to reduce or even eliminate the number of these bugs when cloud-based applications encounter irregular terminations. The motivation behind this work is to design a technique enabling developers to test the effect of spot instance revocations on cloud-based applications. The main contributions of this research work are:

- We address a new and challenging problem for cloud-based applications that results from irregular terminations due to spot instance revocations.
- To the best of our knowledge, `T-BASIR` is the first automated solution to find bugs of applications resulting from cloud spot instance revocations.
- We evaluate `T-BASIR` using 10 popular open-source applications. Our results show that `T-BASIR` not only finds more instances and different types of BASIR (e.g., performance bottlenecks, data loss, locked resources, and applications that cannot restart) compared to the random approach, but it also locates the causes of BASIR to help developers improve the design of the shutdown process for cloud-based applications during their testing.
- `T-BASIR`'s code and our experimental results are publicly available [10], [11].

A preliminary version of these results appears in [12].

2 Related Work

In this section, we discuss the related work concerning spot instance revocations and application bugs.

2.1 Spot Instance Revocations

To the best of our knowledge, `T-BASIR` is the first automated solution for testing the effect of spot instance revocations on

TABLE 1
Comparison of T-BASIR With the Related Work Concerning Spot Instance Revocations and Application Bugs

(a) Spot Instance Revocations

Modeling Spot Markets		Employing Fault-tolerance Mechanisms			Testing Impact on Applications
Bidding Strategies	Prediction Schemes	Replication	Checkpointing	Migration	
PADB [13]	DrAFTS [14]	Multifaceted Policy [15]	Spoton [16]	Smart Spot Instances [17]	
DBA [18]	Calibration [19]	Proteus [20]	Checkpointing Schemes [21]	Hotspot [22]	T-BASIR
AMAZING [23]	Quantitative Models [24]	Spotcheck [3]	ExoSphere [25]		

(b) Application Bugs

Buggy Templates	Rules and Specifications	Historical Bugs	RAT
Metal Checkers [26]	Alattin [27]	HCM [28]	
PMD [29]	Pr-miner [30]	FixCache [31]	T-BASIR
FindBugs [32]	AFG [33]	Bug Prediction [34]	

The top part of Table (a) indicates existing works that aim to mitigate spot instance revocations. The following row designates the methodology of the proposed solution, followed by a row that designates specific methods. The bottom part of Table (b) indicates existing works that aim to find application bugs. The next row designates the methodology of the proposed solution and the cells contain the name of the proposed solutions.

cloud-based applications. Most of the prior works focused on reducing the effect of spot instance revocations using fault-tolerance methods, such as replication [3], [15], [20], checkpointing [16], [21], [25], and VM migration [17], [22]. Voorsluys *et al.* [15] proposed a fault-aware resource allocation approach that applies the price of spot instances, runtime estimation of applications, and task duplication mechanisms to economically run batch jobs in spot instances. Yi *et al.* [21] proposed checkpointing schemes to reduce the computation price of spot instances and the completion time of tasks. Shastri *et al.* [22] proposed a resource container that enables applications to self-migrate to new spot VMs in a way that optimizes cost-efficiency as the spot prices change.

In addition, other researchers worked on modeling spot markets to reduce the spot instance cost and the performance penalty that results from a high number of revocations, by designing optimal bidding strategies [13], [18], [23] and developing prediction schemes [14], [19], [24]. Song *et al.* [13] proposed an adaptive bidding approach that leverages the spot price history information to choose the bid strategy that increases the profit for brokers of the cloud service. Javadi *et al.* [19] proposed a statistical approach to analyze changes in spot price variations and the time between price variations to explore characterization of spot instances that are required to design fault-tolerant algorithms for applications deployed on cloud spot instances.

2.2 Application Bugs

T-BASIR is the first automated solution to identify instances of BASIR. T-BASIR measures the impact on the state of RAT when the application is irregularly terminated to identify BASIR, as discussed in Section 4.3. Existing bug finding tools are not applicable to BASIR because they rely on searching through the application's execution paths for certain inputs to check if the state value of an application varies from the expected value that represents the input value of the next instruction in this execution path. However, a termination signal can be initiated at every execution state of applications, leading to a significantly larger search space of these states. Prior works required users to provide the buggy templates in order to find application bugs [26], [29], [32], whereas other works automatically inferred rules and specifications by mining existing applications in order to find application bugs [27], [30], [33]. Kermenek *et al.* [33] proposed a probabilistic approach that automatically infers specifications from a source code of an application and uses them to detect incorrect and missing properties in specifications. Other researchers focused on predicting application bugs using historical data of reported bugs [28], [31], [34]. Giger *et al.* [34] proposed a bug prediction approach that learns from source code and change metrics to predict application bugs.

2.2.1 Critical Analysis

In summary, Table 1 briefly gives a comparison of T-BASIR from different existing works that aim to mitigate spot instance revocations and find application bugs. While many of the prior works focused on reducing the effect of spot instance revocations by modeling spot markets and using fault-tolerance methods, these works are subject to altering pricing algorithms and are exposed to incurring overhead related to application completion time and deployment cost, respectively. In contrast, T-BASIR focuses on testing the effect of spot instance revocations on cloud-based applications. Also, although the other prior works focused on finding application bugs using buggy templates, rules and specifications, and historical bugs, these works are subject to limited inputs. However, T-BASIR measures the impact on the state of RAT when the application is irregularly terminated to identify BASIR, as discussed in Section 4.3. As a result, T-BASIR is the first tool that sheds light on the effect of spot instance revocations on cloud-based applications.

3 Problem Statement

In this section, we provide a background on shutdown processes and revocation notifications, discuss sources of BASIR, illustrate the BASIR problem, and formulate the problem statement.

3.1 Shutdown Processes and Revocation Notifications

The shutdown process of an application is often initiated during the execution of application instructions in response to

termination signals. This allows the application to switch its execution control to execute predefined shutdown instructions that save the state of the application and the state of its artifacts within a certain timeout before the operating system removes the application process from the memory. It is very difficult to specify in which sequence instructions should be executed during the shutdown of an application. Doing so requires the knowledge of the execution state of an application at any point when this application receives a termination signal. Also, specifications describing the shutdown process of an application and which states are incorrect are rarely documented. The shutdown process of an application is often left untested because developers often assume that an application is properly terminated as long as its processes are terminated. As a result, the shutdown process of applications may fail to be completed within a certain timeout, leading to an incorrect state that affects the execution of subsequent instances of this application.

In general, cloud providers revoke (i.e., terminate) spot instances after a brief two-minute notification. The revocation notifications are often sent to spot instances when the demand from cloud customers increases and the number of available spot instances that can be supported by a finite number of physical resources in a data center of cloud providers decreases. If the customer's price is greater than the cloud provider's price that depends on the demand, a spot instance will be provisioned to customers' applications at the customer's price. However, when spot instances are already provisioned to customer applications and the cloud provider's price goes above the customer's price, the cloud providers will revoke those spot instances within two minutes by sending termination signals [3]. The cloud providers give spot instances two-minute revocation notifications to enable applications that run in spot instances to be gracefully shut down within the two-minute revocation notice time. However, the brief two-minute revocation notice is not often enough to complete the shutdown process of applications, especially when the applications' memory footprints are greater than 4 GB [16]. As a result, when these applications are being terminated during the execution of the shutdown process of these applications, they might lose their states that lead to BASIR.

3.2 Sources of BASIR

There are two primary sources of BASIR. The first one is spot instance revocations. The other one is shutdown bugs of cloud-based applications.

3.2.1 Spot Instance Revocations

The revocations of spot instances are based on price fluctuations that happen based on demand of spot instances from many cloud customers. The cloud providers often revoke spot instances when the demand increases and the number of available spot instances that can be supported by a finite number of physical resources in a data center of cloud providers decreases. It is very difficult to determine in advance spot instance revocations that depend on the varying demands of cloud customers [4]. Doing so requires cloud customers (i.e., application's owners) to understand how the demands of the spot instances change, how the costs of the allocated spot instances change, and how to make trade-offs between the demands and these costs [1]. As a result, price fluctuations that depend on the demand have a high influence on the number of spot instance revocations.

In addition, it is very difficult for cloud customers to decide a price they are willing to pay for renting a spot instance to run their applications in such a way that reduces the deployment cost and the number of spot instance revocations [35]. When spot instances are already provisioned to cloud customer applications and the customer's price is close to zero, there is a high probability that those spot instances will be revoked by cloud providers. Also, when a cloud customer requests spot instances and the customer's price is close to zero, there is a very low probability that those spot instances will be provisioned to cloud customer applications. Conversely, if cloud customers set their prices close to on-demand instances' prices, cloud customers may reduce the number of revocations of spot instances that are provisioned to their applications, but cloud customers may not benefit from a significant discount of spot instances that is up to 90 percent compared to on-demand instances [2]. As a result, without knowing a demand from different cloud customers in advance, the challenge for cloud customers is to choose a price of spot instances that is both significantly lower than the price of on-demand instances and greater than the cloud provider's price to minimize the cost of the deployment and the number of spot instance revocations.

3.2.2 Shutdown Bugs of Cloud-Based Applications

The shutdown bugs of applications often result from errors in the implementation of a cleanup process of these applications that occurs only during their shutdowns. The shutdown sequence of an application is often left untested because developers often assume that an application is properly terminated as long as its processes are terminated. Developers often depend on the assumption that the operating system cleans the process space to a certain extent in any case. Also, specifications describing the shutdown process of an application and which states are incorrect are rarely documented. Unfortunately, existing bug finding tools (e.g., PMD [29] and FindBugs [32]) are not applicable to BASIR because they rely on searching through the application's execution paths for certain inputs to check if the state value of an application varies from the expected value that represents the input value of the next instruction in this execution path. However, a termination signal can be initiated at every execution state of applications, leading to a significantly larger search space of these states. On top of that, the shutdown sequence of an application varies based on the type of termination signals [9].

In addition, it is very difficult to analyze irregular terminations, even for a single execution path of an application for certain inputs since termination signals can be initiated at every point during the execution of the path resulting in deviations from the execution path [36]. For example, termination signals that are initiated during the execution of the third-party's instructions could change the application state, resulting in BASIR. Also, it is very difficult to specify in which sequence instructions should be executed during the shutdown of an application. Doing so requires the knowledge of the execution

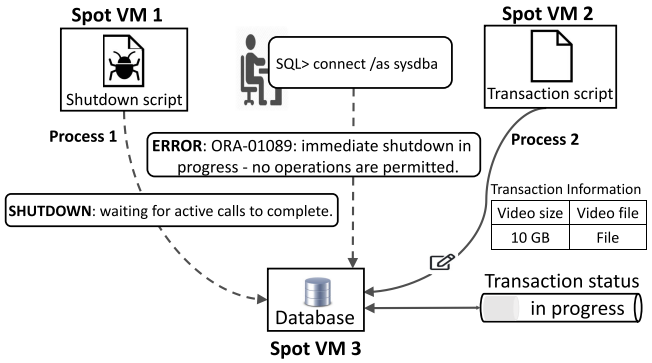


Fig. 1. An illustrative example of BASIR.

state of an application at any point when this application receives a termination signal. Furthermore, multiple termination signals can be initiated during the execution of the shutdown instructions of an application, leading to a significantly larger search space.

3.3 Illustrative Example

The BASIR problem with a cloud-based application is illustrated in Fig. 1. As discussed in Section 3.2, BASIR results from two primary sources: shutdown bugs of applications and spot instance revocations. We show an instance of BASIR that arises from the interactions between a shutdown bug of an application, which comes from a real shutdown bug [5], and the revocation of a spot VM that represents the normal behavior of spot VMs. Our illustrative example shows a typical cloud-based application where a cloud-based application and its artifacts are often replicated across multiple VMs to improve its fault tolerance and reduce its network latency. The cloud-based application and its artifacts are deployed on three spot VMs, where spot VM 1 contains an Oracle shutdown script that reflects a routine script for databases in production, spot VM 2 contains a transaction script that uploads a video file with a large size (e.g., 10 GB), and spot VM 3 contains an Oracle database.

Suppose that the Oracle shutdown script in spot VM 1 that runs on a particular process (Process 1) is executed to terminate the Oracle database that runs in spot VM 3 at the same time another process (Process 2) in spot VM 2 is holding the lock on this Oracle database to perform the transaction. Hence, Process 1 will be waiting until Process 2 releases the lock from the Oracle database. However, consider what happens when spot VM 2 is revoked as a part of the normal behavior of spot VMs while the transaction that is executed by Process 2 is still ongoing. Since Process 2 does not release the lock before the revocation of spot VM 2, the Oracle database will hang and consume needlessly resources until Process 1 determines that Process 2 is gone. The Oracle database prevents users from performing other operations (see the error message in the middle of Fig. 1), since the database is waiting for active calls to be finished (see the log on the left side of Fig. 1). Furthermore, if the spot VM 3 that contains the database is also revoked, this revocation (i.e., an irregular termination of the database) may not only produce an inconsistent state of various data or an incorrect state of artifacts in the database but also may affect the execution of subsequent instances of the database.

Additionally, we point out to multiple real-world bugs resulting from irregular terminations to shed light on the effect of spot instance revocations on applications. Irregular revocations could cause severe bugs, such as EX file system corruption [37], data loss on Atom editor [38], data loss on XFS file system [39], data corruption on Docker container [40], SQLite file corruption [41], database corruption on Docker [42], and Leveldb database corruption [43]. Also, the Linux documentations describe that although Linux can often repair file system corruption due to a power failure, some situations may require manual interventions to repair non-recoverable file system issues [44].

3.4 The Problem Statement

With spot instances becoming pervasive, bugs of cloud-based applications resulting from spot instance revocations have become a very important concern for cloud customers (i.e., application's owners). In this paper, we address a new and challenging problem of testing the effect of spot instance revocations on cloud-based applications – *how to find bugs of cloud-based applications that result from spot instance revocations*. Also, Eqs. (1) and (2) describe how to search through RAT for certain execution points (i.e., system calls) to check if the value t'_{ij} of RAT j during the execution of an execution point i when a cloud-based application is irregularly terminated varies from the expected value t_{ij} that represents the value of RAT j during the execution of an execution point i when a cloud-based application is regularly terminated. Once a difference b_{ij} is found, this difference is added to the matrix B of potential BASIR

$$B := T - T' \tag{1}$$

$$\begin{gathered} b_{ij} = \begin{cases} 0 & t_{ij} = t'_{ij} \\ (t_{ij} - t'_{ij}) & t_{ij} \neq t'_{ij} \end{cases} \\ \forall\, i \in \{1, \ldots, n\}, \forall\, j \in \{1, \ldots, m\}, \\ t_{ij} \in T, \quad t'_{ij} \in T', \quad b_{ij} \in B. \end{gathered} \tag{2}$$

Here, T is a matrix of size $n \times m$, n and m designate the total number of execution points (i.e., system calls) and RAT, respectively, for regular terminations of a cloud-based application, t_{ij} is the value of RAT j during the execution of an execution point (i.e., a system call) i when a cloud-based application is regularly terminated. T' is another matrix of size $n \times m$ for irregular terminations of a cloud-based application, t'_{ij} is the value of RAT j during the execution of an execution point i when a cloud-based application is irregularly terminated. Also, B is another matrix of size $n \times m$ for potential BASIR, b_{ij} is the difference between t_{ij} and t'_{ij}.

The root of this major problem is that cloud-based applications that are exposed to irregular terminations are not designed or tested to deal with this behavior in the cloud environment. Thus, when cloud-based applications are being irregularly terminated, their current state might be lost, which leads to certain bugs, such as data loss, inconsistent states, performance bottlenecks, hangs, crashes, deadlocks, or locked resources. On top of poor user experience from seeing these bugs, other bugs result in situations where cloud-based applications could not be restarted without manual interventions. As a result, the advantages of

cloud spot instances could be significantly minimized or even entirely negated. To the best of our knowledge, T-BASIR is the first automated solution to identify instances of BASIR. BASIR results from two primary sources: shutdown bugs of applications and spot instance revocations. However, since spot revocations are unpredictable and cloud-based applications are not designed or tested to deal with cloud spot revocations, BASIR is a critical problem for cloud customers, and T-BASIR is an essential tool to shed light on the effect of spot instance revocations on cloud-based applications. Thus, when the number of spot instance revocations or the number of shutdown bugs of applications increase, the number of BASIR will likely increase, and vice versa.

Specifically, we use kernel modules to find these bugs and generate traces of their causes in the source code. With our solution, developers can analyze the found bugs and their traces to improve the design of the shutdown process for cloud-based applications during the testing of these applications. Automatically finding these bugs is extremely difficult, in general, especially since a termination signal can be initiated at every execution state of applications, leading to a significantly larger search space.

4 Solution

In this section, we introduce KMs, explain why we use KMs, describe how we utilize KMs in T-BASIR and outline the architecture and workflow of T-BASIR.

4.1 Why we use Kernel Modules in T-BASIR

A KM is a mechanism for (un)loading some codes into an operating system at runtime without rebooting the operating system to extend its functionalities [9]. KMs facilitate modifying the flow of executions, handling the interruption of termination signals, and accessing the information of kernel space functions. There are three main reasons for using KMs rather than modules in the user space. First, using modules in the user space, it is very difficult to synchronize between a process of a cloud-based application that performs a specific operation (e.g., write) on certain resources and a process that sends a termination signal to this application. Second, it is very difficult to time the execution of a particular instruction of a cloud-based application in the user space because an operating system that runs in the kernel space determines the schedule of executing this instruction. Third, some termination signals (e.g., SIGKILL) often invoke the signal handlers in the kernel space instead of the signal handler in the user space (i.e., a signal handler that is defined in the source code of a cloud-based application) [9]. In contrast, KMs have complete control over the execution in the kernel space at runtime. As a result, T-BASIR uses KMs to ensure termination signals are sent to certain points in the execution of a cloud-based application and to measure the impact on the state of RAT at these points of the execution in order to find BASIR.

4.2 Why we use Synchronization System Calls in T-BASIR

In general, the synchronization system calls are responsible for managing the access of shared data objects among multiple processes (or threads). T-BASIR focuses on the synchronization system calls since the irregular terminations of synchronization system calls may negatively affect not only the state of shared data objects causing bugs (e.g., data loss, data corruption) but also the state of external processes (or threads) that run on different instances and interact with the process of terminated system calls, causing bugs (e.g., deadlocks and performance bottlenecks). However, although a write system call is another important type of system call that is responsible for modifying the value of data objects, the irregular termination of write system calls may negatively affect only the modified data objects, causing bugs (e.g., data loss, data corruption). Thus, the irregular termination of synchronization system calls may cause more bugs that are related to data objects and processes (or threads) within the critical section of synchronization system calls, compared to the irregular termination of the write system calls that may cause bugs related to only the modified data objects.

4.3 Automating BASIR Detection Using KMs

In T-BASIR, our terminator KM specifies when we send a termination signal during the execution of cloud-based applications that mimics the irregular terminations, as discussed in Section 1. An essential goal is to identify which instructions of applications are more likely to lead to BASIR in order to send termination signals during the executions of these instructions. Given that BASIRs are more likely to be exposed when instructions use resources to perform certain operations (e.g., write) that are often accessed when specific system calls [9] (e.g., acquire-lock) are invoked, we favor instructions whose executions access these resources. Our terminator KM sends a termination signal during the execution of these system calls, which correspond to specific instructions in the source code. Our terminator KM uses the number of a system call with KProbe and JProbe interfaces [9] to intercept the execution of these system calls and, hence, ensures that a termination signal is sent to certain points of the execution. In summary, our terminator KM sends termination signals only during the execution of these instructions to increase the degree of precision for finding BASIR. In the RANDOM approach, a termination signal is sent to any point in the execution of a cloud-based application. Our hypothesis is that our terminator KM is more effective than randomly sending termination signals to any instructions because determining to which instruction a termination signal should be sent is highly correlated to the probability of finding BASIR. We verify our hypothesis with the experimental data in Section 6

$$B(T,T') = \sum_{i=1}^{n}\sum_{j=1}^{m} D(t_{ij},t'_{ij}) \text{ where } t \in T,\ \ t' \in T' \tag{3}$$

$$D\big(t_{ij},t'_{ij}\big) = \begin{cases} 0 & t_{ij} = t'_{ij} \\ 1 & t_{ij} \neq t'_{ij} \end{cases}. \tag{4}$$

Here, T is a matrix of size $n \times m$, n and m designate the total number of system calls and RAT, respectively, for regular terminations of a cloud-based application, t_{ij} is the value of RAT j during the execution of a system call i when a cloud-based application is regularly terminated. T' is another matrix of size $n \times m$ for irregular terminations of a cloud-based application, t'_{ij} is the value of RAT j during the execution of a system call i when a cloud-based application

is irregularly terminated. D is the delta function that evaluates the presence of BASIR by comparing the difference between the value of RAT when a cloud-based application is regularly terminated and the value of the same RAT when this application is irregularly terminated during the execution of the same system call. B is the summation function that computes the total number of BASIR by analyzing executions between irregular and regular terminations of a cloud-based application for m RAT and n system calls.

In T-BASIR, our detector KM determines when irregular terminations lead to BASIR. We use the values of RAT (e.g., variables and artifacts) for cloud-based applications to identify the presence of BASIR. Initially, we randomly select a set of system calls of a cloud-based application. Then, we use our identifier KM to record the values of RAT that are used by these system calls when a cloud-based application is regularly terminated. For each system call, we run this application to collect the values of the RAT when this application is irregularly terminated. Our detector KM uses Eq. (4) to measure the difference between the value of RAT when the cloud-based application is regularly terminated and the value of the same RAT when the cloud-based application is irregularly terminated during the execution of the same system call. We use the difference operation to evaluate the presence of BASIR by analyzing executions between irregular and regular terminations, since we assume that running a single execution path of a cloud-based application for certain inputs multiple times leads to the same values of the RAT in different runs. When the value of the RAT after irregular terminations varies from the expected value of the RAT at the same point in the execution after regular terminations, it indicates a potential instance of BASIR. Hence, once a difference is found, the detector KM uses Eq. (3) to add this difference to the total number of potential BASIR and collects the traces of this BASIR, as discussed in Section 4.4. As a result, with T-BASIR, developers can analyze the found instances of BASIR and their traces to improve the design of the shutdown process for cloud-based applications during the testing of these applications.

Algorithm 1. T-BASIR's Algorithm for Finding BASIR and Locating Their Causes

1: **Inputs:** KM Configuration Ω, Application $\mathcal{A}$
2: **LoadIdentifierKMs** (Ω)
3: **while** $\mathcal{A} \neg$ **Terminate do**
4: $\mathcal{T} \leftarrow$ **IdentifySyscallRAT**($\mathcal{A}$, Ω)
5: **end while**
6: **UnloadIdentifierKMs** (Ω)
7: **LoadTerminatorDetectorKMs** (Ω)
8: **for** each system call i **in** $\mathcal{T}$ **do**
9: **for** each RAT j **in** $\mathcal{T}$ **do**
10: $t'_{ij} \leftarrow$ **MeasureSyscallRAT**($\mathcal{A}$, Ω)
11: **if** $t_{ij} \neq t'_{ij}$ **then**
12: $\mathcal{B} \leftarrow \mathcal{B} + 1$
13: $\mathcal{C} \leftarrow$ **CollectTraces**(t'_{ij})
14: **end if**
15: **RestoreAppInitialState**($\mathcal{A}$)
16: **end for**
17: **end for**
18: **UnloadTerminatorDetectorKMs** (Ω)
19: **return** $\mathcal{B}$, $\mathcal{C}$

T-BASIR is illustrated in Algorithm 1 that contains the following main phases: (i) send termination signals to certain system calls of a cloud-based application, and (ii) measure the impacts on the state of RAT when the cloud-based application is irregularly terminated during the execution of these system calls. The algorithm for T-BASIR takes in the entire set of inputs for the cloud-based application, its snapshot, and the KM configurations Ω, containing the identifier, terminator and detector KMs. Starting from Step 2, the algorithm loads the identifier KM into an operating system. In T-BASIR, we use lock system calls, where a thread locks certain resources to perform read or write operations. In Steps 3-5, the identifier KM randomly selects a set of system calls and records the values of RAT that are used by these system calls when the cloud-based application is regularly terminated. In Step 6, the identifier KM is unloaded from the operating system. In Step 7, the terminator and the detector KMs are loaded into the operating system. In Steps 8-17, for each system call and RAT, the algorithm repeatedly runs the snapshot of the cloud-based application, and then the terminator KM sends a termination signal to the cloud-based application during the execution of this system call. For each run, the detector KM uses Eq. (4) to measure the difference between the value of RAT when the cloud-based application is regularly terminated and the value of the same RAT when this application is irregularly terminated during the execution of the same system call. Once a difference is found, the detector KM uses Eq. (3) to add this difference to the total number of potential BASIR and collects its traces, as discussed in Section 4.4. The cycle of Steps 8-17 repeats until the set of system calls is completed. Finally, in Step 18, the terminator and the detector KMs are unloaded from the operating system. The found instances of BASIR and their traces are returned in Step 19 as the algorithm ends.

4.4 Identifying the Causes of BASIR

Tracer KM is at the core of the T-BASIR tracer to identify the causes of BASIR. We provide an overview and describe the implementation design of the T-BASIR tracer.

4.4.1 Overview of T-BASIR Tracer

Our goal is to automatically determine specific instructions in the source code of cloud-based applications that lead to BASIR when these applications encounter irregular terminations. In order to contrast instructions that lead to BASIR, we rely on the stack trace approach [45] that can be used to collect execution traces from the stack in the memory when a cloud-based application is irregularly terminated. The stack traces contain a sequence of method calls with corresponding instructions, which often represents the current point in the execution path. These traces are often difficult to capture because termination signals can be initiated at every point in the execution of a cloud-based application, leading to a significantly larger search space. Hence, existing tracing tools [45] are not applicable to BASIR because the stack traces of applications are gone as soon as these applications are terminated. However, our tracer KM in T-BASIR can intercept a termination signal before this signal is delivered to a cloud-based application, as discussed

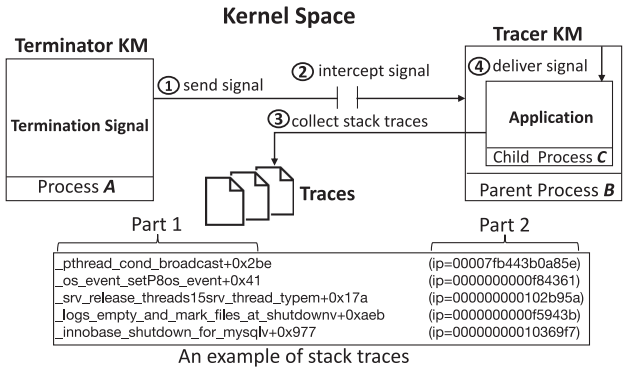


Fig. 2. The implementation design of T-BASIR tracer.

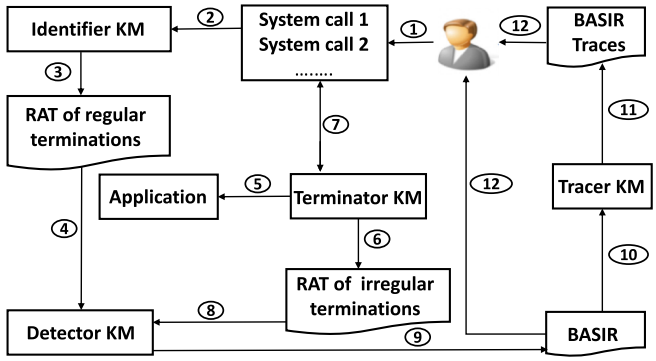


Fig. 3. The architecture and workflow of T-BASIR.

in Section 4.4.2. As a result, during application testing, developers can use these traces to identify corresponding instructions in the source code that lead to instances of BASIR.

4.4.2 Implementation Design of the T-BASIR Tracer

The implementation design of the T-BASIR tracer in the kernel space is illustrated in Fig. 2. The rectangles denote Terminator KM, Tracer KM, and a cloud-based application with their processes. The arrows indicate the actions between these KMs and the cloud-based application, and the numbers in the circles show the sequence of operations in the T-BASIR tracer.

Our tracer KM can intercept a termination signal before this signal is delivered to the cloud-based application because this application runs inside our tracer KM, as illustrated on the right side of Fig. 2 (i.e., this application runs on a child process of the tracer/parent process). In particular, when a termination signal is sent (1) to the cloud-based application, our tracer KM first intercepts (2) this signal to collect and store (3) the execution traces of the cloud-based application in files (e.g., text files) and then delivers (4) this signal to terminate this application.

In general, it is very difficult to time the execution of a particular instruction of a cloud-based application in the user space because an operating system that runs in the kernel space determines the schedule of executing this instruction. Conversely, KMs have complete control over the execution in the kernel space at runtime. Hence, our tracer KM modifies the flow of executions through intercepting a termination signal to collect execution traces from the stack of a cloud-based application before the application receives the termination signal. In particular, our tracer KM uses Libunwind interfaces [46] to generate execution traces from the stack memory of the cloud-based application. An example of stack traces for a cloud-based application (e.g., MySQL) is illustrated in the bottom of Fig. 2. The first part of these traces refers to the sequence of method calls with corresponding instructions that represents the current point in the execution path when the cloud-based application is being irregularly terminated, and the second part of these traces refers to the methods' instruction pointers. As a result, the traces of BASIR can be reviewed by developers during application testing to identify which instructions in the execution path may lead to instances of BASIR.

4.5 T-BASIR's Architecture and Workflow

The architecture of T-BASIR is illustrated in Fig. 3. The rectangles indicate components of T-BASIR, the arrows denote the data flow between components, and the numbers in the circles show the sequence of processes in the workflow.

The input to T-BASIR is the entire set of inputs for a cloud-based application that performs specific operations (e.g., write) on certain resources (i.e., RAT), which often invoke particular system calls (e.g., acquire-lock) to use these resources. Initially, a set of system calls of the cloud-based application is chosen at random (1). For each system call, RATs are identified (2), and *Identifier KM* records (3) the values of RAT that are used by this system call when the cloud-based application is regularly terminated. These values of RAT that represent the expected values of the RAT, as discussed in Section 4.3, are passed (4) to *Detector KM*. *Terminator KM* sends (5) a termination signal to the cloud-based application during the execution of each system call. The values of RAT that are used by these system calls when the cloud-based application is irregularly terminated are collected (6). The evaluation is evolved using *Terminator KM* until the set of system calls is completed (7).

When the values of RAT for all system calls are collected, these values of RAT are passed (8) to *Detector KM*. *Detector KM* uses Eq. (4) to measure the difference between the value of RAT when the cloud-based application is irregularly terminated and the expected value of the same RAT when the cloud-based application is regularly terminated during the execution of the same system call. When the value of the RAT after irregular terminations varies from the expected value of the RAT at the same point in the execution after regular terminations, it indicates a potential instance of BASIR. Then, when a difference is found, *Detector KM* uses Eq. (3) to add this difference to the list of potential BASIR (9). Once the list of potential BASIR is obtained (10), *Tracer KM* collects (11) the traces of BASIR that contain a sequence of method calls with corresponding instructions, as discussed in Section 4.4. The found instances of BASIR and their traces are given to the developers for further evaluation (12).

5 Empirical Evaluation

In this evaluation section, we state our *Research Questions (RQs)*, illustrate subject applications, describe our methodology to evaluate T-BASIR, and outline threats to its validity.

RQ_1: How effective is T-BASIR compared to the random approach in finding more instances of BASIR?

TABLE 2
Overview of the Applications: Their Names Followed by the Versions of the Applications, and the Total Number of Accessed Futexes and Their System Calls When These Applications Restart After Regular Terminations

Application	Version	Futexes	Syscalls
MySQL	v5.7.25	58	132
Cassandra	v3.0.17	35	138
PostgreSQL	v10.6	3	5
CouchDB	v2.3.0	25	11920
MongoDB	v3.0.6	61	1201
Hbase	v2.1.2	53	808
Docker	v18.09.0	45	1583
Hadoop	v3.0.3	34	1716
ZooKeeper	v3.4.12	35	910
Hive	v2.1.1	32	874

TABLE 3
The Comparison of the Results of BASIR for T-BASIR and RANDOM

Application	T-BASIR	RANDOM
MySQL	✓	×
Cassandra	×	×
PostgreSQL	×	×
CouchDB	✓	✓
MongoDB	✓	×
Hbase	✓	×
Docker	×	×
Hadoop	✓	×
ZooKeeper	✓	×
Hive	×	×

RQ_2: How effective is T-BASIR in finding different types of BASIR?

RQ_3: Is T-BASIR more effective than the random approach in causing more impacts on the application behaviors?

5.1 Subject Applications

We evaluated T-BASIR on 10 open-source subject applications. An overview of the subject applications is shown in Table 2. These applications are multithreaded, have high popularity indexes, come from different domains, and are written by different programmers. The synchronization mechanism of these applications relies on a futex system call [47], which is a fast user-space synchronization method that puts specific threads to sleep/wait or wakes waiting threads when specific conditions become true. Each critical section in these applications often uses certain futex variables that are stored in particular memory addresses and are used by multiple threads to access this critical section through futex system calls [47].

5.2 Methodology

For each application, we first use the Strace tool [9] to ensure that its synchronization mechanism relies on futex system calls. As discussed in Section 4.3, T-BASIR analyzes the values of the RAT between regular and irregular terminations at the same point in the execution to identify BASIR. RATs are the logs of the subject applications, the logs of the Linux kernel, the number of accessed futexes, and the number of futex system calls. An application is irregularly terminated using the RANDOM approach, where a termination signal is sent to any point in the execution of this application, and in T-BASIR, where a termination signal is sent to specific points in the execution of this application (i.e., during the executions of futex system calls). T-BASIR uses the logs to identify different types of BASIRs that lead to different effects on the behaviors of applications to answer RQ_1 and RQ_2. T-BASIR also identifies other cases of BASIR when the logs do not contain error messages. For example, T-BASIR identifies when applications cannot restart without manual interventions using the process status tool [9]. Also, we measure the impacts on the behaviors of the subject applications to answer RQ_3. When an application restarts after irregular terminations, we check if values for the total number of accessed futexes and their system calls vary from the expected values when this application restarts after regular terminations for 20 seconds, which is set experimentally. Once a significant change is identified, as discussed in Section 4.3, T-BASIR adds this change to the total number of potential BASIR and collects its traces. T-BASIR is implemented using KMs, KProbe, and JProbe interfaces [9]. The experiments for the subject applications were carried out using 10 virtual machines. Each subject application was deployed on Ubuntu 18.04 LTS VM with 4 GB of memory and 4 GHz CPU. For each application, we created a snapshot to ensure a similar state of the test environment after irregular terminations.

5.3 Threats to Validity

Our implementation of T-BASIR deals with only futex system calls, whereas other applications may use different synchronization mechanisms (e.g., semaphore system calls [9]). While this is a potential threat, it is unlikely a major threat, since T-BASIR can be adjusted to support other types of synchronization mechanisms. In order to use T-BASIR with other applications, the developer can change only the system call type in the KMs so that T-BASIR identifies other types of system calls.

We experimented with only synchronization system calls, whereas other types of system calls (e.g., information flow, creation, preparatory, and termination) could also result in different effects on the behaviors of applications when these applications are terminated during the execution of other types of system calls. In contrast, understanding the effect of different types of system calls on the behavior of the applications is beyond the scope of this empirical study and shall be considered in future studies.

6 Empirical Results

In this section, we discuss the experimental results to answer the RQs listed in Section 5.

6.1 Finding More Instances of BASIR

The experimental results to answer RQ_1 are shown in Table 3 and summarize the found instances of BASIR when the subject applications encounter irregular terminations using T-BASIR and RANDOM approaches. We focus on determining whether these applications restart without manual interventions after they are irregularly terminated using T-BASIR

and RANDOM. The experimental results show that T-BASIR causes MySQL, CouchDB, MongoDB, HBase, Hadoop, and ZooKeeper not to restart without manual interventions, whereas the RANDOM approach causes only CouchDB to not restart without manual interventions. Our explanation is that the RANDOM approach was able to cause CouchDB not to restart without manual interventions, since CouchDB uses an extremely high number of futex system calls, as shown in Table 2. Hence, the RANDOM approach may accidentally hit these futex system calls, resulting in an instance of BASIR.

On the other hand, T-BASIR was not able to cause PostgreSQL, Cassandra, Docker, and Hive not to restart without manual interventions. Our explanation is that PostgreSQL uses an extremely low number of futex system calls as shown in Table 2. This situation puts T-BASIR at a disadvantage to find BASIRs since causing BASIR often requires more interactions among threads that often occur when a large number of futex system calls are executed. Cassandra runs on Java processes using a Java Virtual Machine (JVM), and T-BASIR uses Java processes instead of the application name processes (i.e., Cassandra) to specify the desired process of an application for receiving termination signals. Subsequently, JVM may play some roles in reducing the effect on Cassandra since Cassandra receives termination signals through the JVM. Docker uses the resource isolation features for the kernel. T-BASIR uses KMs to send termination signals to the process of the subject applications. Hence, these features may play some roles in reducing the effect on Docker when Docker receives termination signals. Even though the Hive server restarts after irregular terminations using T-BASIR, its HCatalog component fails to restart. This observation allows us to conclude that even though irregular terminations may not show an impact on the restart state of an application, it does not mean that the other components of this application have no impacts too. In summary, our results show that T-BASIR causes six subject applications not to restart without manual intervention, whereas the RANDOM approach causes only one subject application not to restart without manual intervention, thus *positively addressing* $\boldsymbol{RQ}_1$.

6.2 Finding Different Types of BASIR

When we investigate $\boldsymbol{RQ}_2$, we observe that unlike the RANDOM approach, T-BASIR leads to other types of BASIR. Since we are more familiar with the MySQL components, we further analyze and discuss the effects of other types of BASIR for MySQL. We observe that the logs of MySQL report the following message. `[Note] InnoDB: page_cleaner: 1000 ms intended loop took 848417ms` [11]. The message shows that the `page_cleaner` method that is responsible for writing data from memory into the disk takes a very long time from 1 second, which is expected, to 848 seconds (~14 minutes). This result demonstrates a major problem, since it results in not only performance bottlenecks but also data loss. We analyze the effect of data loss by creating a virtual machine with 1 GB of memory, and we use `MySQLlap client` to perform large write operations (e.g., inserting hundreds of records) using multiple threads. We then load T-BASIR into the operating system to send the termination signals during the execution of these system calls. Interestingly, we observed that once MySQL restarts, the recently written data is lost. This bug is also reported on the following web page [6]. Also, we observed the following error message: `[ERROR] InnoDB: Unable to lock ./ibdata1 error: 11` [11]. The error message shows that T-BASIR prevents MySQL from performing a clean shutdown and hence results in locked ibdata1, which is a file that includes the shared tablespace containing the internal data of InnoDB. Unlike the RANDOM approach, T-BASIR also leads to other types of BASIR, such as performance bottlenecks, data loss, and locked resources. This result confirms that T-BASIR also results in different types of BASIR, compared to the RANDOM approach, thus *positively addressing* $\boldsymbol{RQ}_2$. As a result, when irregular terminations result in BASIR, T-BASIR collects the traces that contain a sequence of method calls with corresponding instructions, as discussed in Section 4.4. Hence, developers can use these traces to improve the design of the shutdown process for the subject applications during the testing of these applications.

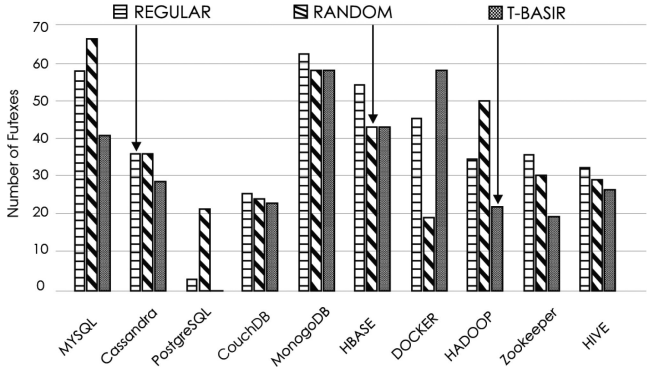


Fig. 4. The comparison of the total number of accessed futexes for regular and irregular terminations.

6.3 Impact of T-BASIR on the Behaviors of Applications

The results of the experiments are presented in the histogram plot in Fig. 4 that summarizes the number of accessed futexes for the subject applications when these applications restart after regular and irregular terminations using T-BASIR and RANDOM approaches. These futexes often control the access of shared resources in critical sections across various threads/processes of an application. Different futexes often correspond to different execution paths since these futexes control the access of critical sections in different methods of an application. We observe that the number of accessed futexes varies between regular and irregular terminations using T-BASIR and RANDOM approaches. This observation suggests that the execution paths between regular and irregular terminations of an application change where newly accessed futexes (i.e., extra futexes) may have been accessed in the recovery execution paths, or other futexes that are often used during the execution of the application startup may not have been accessed (i.e., missing futexes) [36]. We observe that, except for Docker, most numbers of accessed futexes when applications are irregularly terminated using T-BASIR are lower than the number of accessed futexes when applications are regularly terminated or irregularly terminated using the RANDOM approach. A higher change in the number of accessed futexes often indicates a higher change in the execution paths when an application restarts

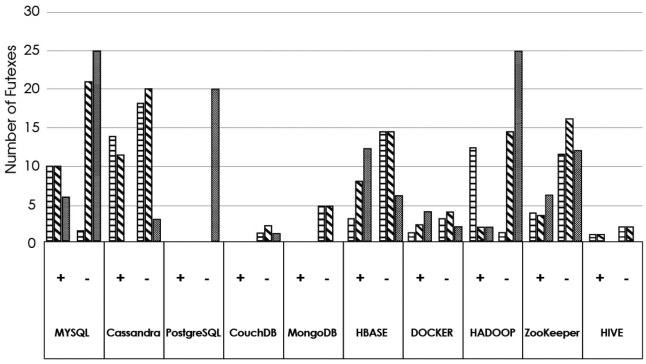


Fig. 5. The horizontal stripes, diagonal stripes, and dotted bars represent the change of accessed futexes between RANDOM and REGULAR, T-BASIR and REGULAR, and T-BASIR and RANDOM approaches, respectively. The plus and minus symbols specify extra and missing futexes, respectively.

after regular and irregular terminations. Further details about the results for all applications are shown in Fig. 5, where the number of extra and missing futexes are provided. Interestingly, we observe that there is a change in the number of accessed futexes between T-BASIR and RANDOM approaches, which suggests when an application encounters irregular terminations using different approaches, it often leads to different execution paths for the application. Hence, this observation confirms that the change in the execution paths not only indicates the recovery execution paths but also indicates other execution paths that may result in instances of BASIR [36]. As a result, these experimental results demonstrate that when applications encounter irregular terminations using different approaches, it often leads to different execution paths, which result in different impacts on the behaviors of these applications.

To investigate $\boldsymbol{RQ}_3$ further, we present the change in the number of futex system calls for CouchDB in Table 4 when this application restarts after regular and irregular terminations using T-BASIR and RANDOM. The experimental results for other applications can be found in appendix (see the supplemental file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2020.2980265). We assume that running recovery execution paths of an application multiple times leads to the same values of the futex system calls for certain futexes in different runs. Hence, when the number of these futex system calls of an application after irregular termination using T-BASIR varies from the number of these futex system calls of this application after irregular termination using the RANDOM approach, it suggests the former recovery execution paths deviate from the latter recovery execution paths, which often indicates different impacts on the behaviors of this application. In particular, we observe that the number of futex system calls when CouchDB is irregularly terminated using T-BASIR, except for a few futexes, is often greater than the number of futex system calls when CouchDB is regularly terminated or irregularly terminated using the RANDOM approach. This result suggests that irregular terminations that are initiated by T-BASIR often lead to more impacts on the behaviors of applications compared to the RANDOM approach, since the higher number of futex system calls indicates not only more thread contentions but also a higher chance of locked resources. Interestingly, we observe that a futex with the last four digits of the memory address $0x0610$ for CouchDB has a significant decrease in the number of its futex system calls between regular and irregular terminations, which suggests some threads that use this futex may be prevented (i.e., locked) from reaching this point in the execution.

TABLE 4
The Comparison of the Total Number of Futex System Calls for CouchDB After Regular and Irregular Terminations

Address	REGULAR	RANDOM	T-BASIR
0x12c8	3	0	0
0x0190	400	512	522
0x01d0	417	516	528
0x0210	396	518	526
0x0250	409	506	518
0x0290	414	522	530
0x02d0	412	512	528
0x0310	397	526	534
0x0350	402	518	528
0x0390	449	578	584
0x03d0	403	520	532
0x0410	405	522	538
0x0450	392	523	530
0x0490	563	705	686
0x04d0	396	520	528
0x0510	391	506	507
0x0550	382	514	522
0x0590	6	8	10
0x05d0	11	15	13
0x0610	5245	3402	3315
0xdf78	3	3	0
0xf7f8	3	6	3
0x95c8	19	5	9
0x95cc	1	1	1
0x9660	1	1	1

Finally, we also observe that a futex with the last four digits of the memory address of $0x0020$ appears in extra futexes across different applications, such as Hadoop, HBase, and Hive, when they are restarted after irregular terminations. This observation suggests that this futex is invoked by recovery instructions of JVM, which is also reported on the collected traces of these applications [11]. Hence, fixing these recovery instructions of JVM will reduce or even eliminate the number of BASIR for all applications that rely on JVM. In summary, these experimental results demonstrate that T-BASIR not only results in different impacts on the behaviors of these applications but also leads to more impacts on the behaviors of these applications compared to the RANDOM approach, thus *positively addressing* $\boldsymbol{RQ}_3$. As a result, when certain futexes result in significant changes in the behavior of applications, the traces of these futexes can be reviewed by developers to analyze how the changes of these futexes and their traces may lead to instances of BASIR.

7 Conclusion and Future Work

We addressed a new and challenging problem for cloud-based applications that results from spot instance revocations. We proposed a novel solution to automatically find Bugs of cloud-based Applications that result from Spot

instance Revocations (BASIR) and to locate their causes in the source code. We developed our solution for Testing the BASIR (T-BASIR), and we evaluated it using 10 popular open-source applications. The results show that T-BASIR finds more instances of BASIR and different types of BASIR, such as performance bottlenecks, data loss and locked resources, and applications that cannot restart, compared to the Random approach. With T-BASIR, developers can analyze the traces of BASIR to improve the design of the shutdown process for cloud-based applications during their testing and, hence, to gain the advantage of cloud spot instances in the cloud. This enables stakeholders to economically deploy their applications on the cloud spot instances. To the best of our knowledge, T-BASIR is the first automated solution to find bugs of cloud-based applications resulting from spot instance revocations.

In the future, we plan to investigate the following research directions to extend our work.

- *Exploring the Impact of other Types of System Calls.* We plan to study the effect of I/O system calls that are responsible for reading/writing data from/to storage on applications when these applications are irregularly terminated during the execution of I/O system calls. For example, we will test the irregular termination of sync system calls that are responsible for synchronizing cached writes from volatile buffers to non-volatile buffers (i.e., persistent storage) to ensure that changes on volatile buffers are successfully flushed and committed to persistent storages on an irregular revocation. When sync system calls (i.e., fsync) are interrupted, due to irregular termination, we expect that cached writes in volatile buffers will likely not be committed to non-volatile storage, causing data loss. Another example of I/O system calls is write system calls. Let us suppose that concurrent write system calls are executed by separate processes/threads writing into a single buffer. However, consider what happens when one of these write system calls is interrupted by irregular termination. These processes/threads may not put all the data in a row, which causes data corruption.
- *Building Reliable Applications against Revocations.* We plan to build revocation-robust applications in cloud spot markets to reduce the number of BASIR when these applications encounter irregular terminations. In particular, our goal is to optimize the design of the shutdown sequence for these applications using certain specifications that describe the shutdown sequence. These specifications can be defined based on the developers' recommendations of the found instances of BASIR or common design flaws in the applications' shutdown process (i.e., bug reports in code repositories, as discussed in Section 3.3). For example, applications should make their buffered writes short (i.e., reducing the dirty data buffer time), and applications should first flush the primary data of applications in volatile buffers and then flush the secondary data of these applications (e.g., log files) in volatile buffers. Also, partitions used in applications should be mounted as read-only and temporary remounted as read and write during the write operations. However, if an irregular termination occurs during the write operations, partitions should be remounted as read-only, which will likely force flushing volatile-buffers faster, and additional writes to volatile buffers should be blocked. Hence, closing files that are opened for writing may reduce the negative effect on these files due to irregular terminations, whereas files that are open for reading will likely not be affected by irregular terminations. In general, applications should operate based on the magnitude of the termination interval to determine whether buffered writes can be stored in permanent stores, and they should also indicate whether the shutdown process is completed successfully. Finally, although such specifications cannot guarantee BASIR-free applications, they will likely reduce the number of BASIR when these applications encounter irregular terminations.
- *Comparing the Execution Time for Different Approaches.* We plan to present the comparison results of the execution time for T-BASIR and RANDOM approaches in future works. The comparison results of the execution time require us to set up 10 virtual machines for 10 subject applications and rerun all experiments for both approaches.

REFERENCES

[1] A. Alourani, M. A. N. Bikas, and M. Grechanik, "Search-based stress testing the elastic resource provisioning for cloud-based applications," in *Proc. 10th Int. Symp. Search-Based Softw. Eng.*, 2018, pp. 149–165. [Online]. Available: https://doi.org/10.1007/978-3-319-99241-9_7

[2] AWS, "Amazon EC2 spot instances," 2020. [Online]. Available: https://aws.amazon.com/ec2/spot/

[3] P. Sharma, S. Lee, T. Guo, D. E. Irwin, and P. J. Shenoy, "SpotCheck: Designing a derivative IaaS cloud on the spot market," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, pp. 16:1–16:15. [Online]. Available: https://doi.org/10.1145/2741948.2741953

[4] B. Sharma, R. K. Thulasiram, P. Thulasiraman, and R. Buyya, "Clabacus: A risk-adjusted cloud resources pricing model using financial option theory," *IEEE Trans. Cloud Comput.*, vol. 3, no. 3, pp. 332–344, Third Quarter 2015.

[5] D. Burleson, "Fix hanging shutdown: Waiting for active calls to complete," 2020. [Online]. Available: http://www.dba-oracle.com/t_hanging_shutdown_waiting_for_active_tasks_to_complete.htm

[6] M. Mäkelä, "Move the InnoDB doublewrite buffer to flat files," 2020. [Online]. Available: https://jira.mariadb.org/browse/MDEV-11659

[7] S. Shastri and D. E. Irwin, "Cloud index tracking: Enabling predictable costs in cloud spot markets," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 451–463. [Online]. Available: https://doi.org/10.1145/3267809.3267821

[8] J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram, "Finding crash-consistency bugs with bounded black-box crash testing," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 33–50. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/mohan

[9] J. Keniston, "The Linux kernel documentation," 2020. [Online]. Available: https://www.kernel.org/

[10] T-basir code on github, 2020. [Online]. Available: https://github.com/Abdullah-687/T-BASIR

[11] Our source code and experimental data, 2020. [Online]. Available: https://www.dropbox.com/s/0z4qndkv6dwkzhu/T-BASIR.zip?dl=0

[12] A. Alourani, A. D. Kshemkalyani, and M. Grechanik, "Testing for bugs of cloud-based applications resulting from spot instance revocations," in *Proc. 12th IEEE Int. Conf. Cloud Comput.*, 2019, pp. 243–250. [Online]. Available: https://doi.org/10.1109/CLOUD.2019.00050

[13] Y. Song, M. Zafer, and K.-W. Lee, "Optimal bidding in spot instance market," in *Proc. IEEE INFOCOM*, 2012, pp. 190–198.

[14] R. Wolski, J. Brevik, R. Chard, and K. Chard, "Probabilistic guarantees of execution duration for amazon spot instances," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2017, Art. no. 18.
[15] W. Voorsluys and R. Buyya, "Reliable provisioning of spot instances for compute-intensive applications," in *Proc. IEEE 26th Int. Conf. Adv. Inf. Netw. Appl.*, 2012, pp. 542–549.
[16] S. Subramanya, T. Guo, P. Sharma, D. E. Irwin, and P. J. Shenoy, "Spoton: A batch computing service for the spot market," in *Proc. 6th ACM Symp. Cloud Comput.*, 2015, pp. 329–341. [Online]. Available: https://doi.org/10.1145/2806777.2806851
[17] Q. Jia, Z. Shen, W. Song, R. van Renesse, and H. Weatherspoon, "Smart spot instances for the supercloud," in *Proc. 3rd Workshop CrossCloud Infrastructures Platforms*, 2016, Art. no. 5.
[18] M. Zafer, Y. Song, and K.-W. Lee, "Optimal bids for spot VMs in a cloud for deadline constrained jobs," in *Proc. IEEE 5th Int. Conf. Cloud Comput.*, 2012, pp. 75–82.
[19] B. Javadi, R. K. Thulasiramy, and R. Buyya, "Statistical modeling of spot instance prices in public cloud environments," in *Proc. 4th IEEE Int. Conf. Utility Cloud Comput.*, 2011, pp. 219–228.
[20] A. Harlap, A. Tumanov, A. Chung, G. R. Ganger, and P. B. Gibbons, "Proteus: Agile ML elasticity through tiered reliability in dynamic resource markets," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 589–604.
[21] S. Yi, D. Kondo, and A. Andrzejak, "Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud," in *Proc. IEEE 3rd Int. Conf. Cloud Comput.*, 2010, pp. 236–243.
[22] S. Shastri and D. Irwin, "HotSpot: Automated server hopping in cloud spot markets," in *Proc. Symp. Cloud Comput.*, 2017, pp. 493–505.
[23] S. Tang, J. Yuan, and X.-Y. Li, "Towards optimal bidding strategy for amazon EC2 cloud spot instance," in *Proc. IEEE 5th Int. Conf. Cloud Comput.*, 2012, pp. 91–98.
[24] C. Wang, Q. Liang, and B. Urgaonkar, "An empirical analysis of amazon EC2 spot instance features affecting cost-effective resource procurement," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 3, no. 2, 2018, Art. no. 6.
[25] P. Sharma, D. E. Irwin, and P. J. Shenoy, "Portfolio-driven resource management for transient cloud servers," in *Proc. ACM SIGMETRICS / Int. Conf. Meas. Model. Comput. Syst.*, 2017, Art. no. 59. [Online]. Available: https://doi.org/10.1145/3078505.3078511
[26] B. Chelf, D. R. Engler, and S. Hallem, "How to write system-specific, static checkers in metal," in *Proc. ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng.*, 2002, pp. 51–60.
[27] S. Thummalapenta and T. Xie, "Alattin: Mining alternative patterns for detecting neglected conditions," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2009, pp. 283–294.
[28] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 78–88.
[29] PMD, "An extensible cross-language static code analyzer," 2020. [Online]. Available: http://pmd.sourceforge.net
[30] Z. Li and Y. Zhou, "PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 306–315, 2005.
[31] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 489–498.
[32] FindBugs, "Findbugs documents and publications," 2020. [Online]. Available: http://findbugs.sourceforge.net/publications.html
[33] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler, "From uncertainty to belief: Inferring the specification within," in *Proc. 7th Symp. Operating Syst. Des. Implementation*, 2006, pp. 161–176.
[34] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level bug prediction," in *Proc. ACM-IEEE Int. Symp. Empir. Softw. Eng. Meas.*, 2012, pp. 171–180.
[35] Q. Liang, C. Wang, and B. Urgaonkar, "Spot characterization: What are the right features to model," in *Proc. Int. Workshop Syst. Analytics Characterization*, 2016. [Online]. Available: https://sites.google.com/site/sacconference2016/submission
[36] D. N. Armstrong, H. Kim, O. Mutlu, and Y. N. Patt, "Wrong path events: Exploiting unusual and illegal program behavior for early misprediction detection and recovery," in *Proc. 37th Annu. Int. Symp. Microarchit.*, 2004, pp. 119–128.
[37] Juniper, "Ex file system corruption," 2020. [Online]. Available: https://kb.juniper.net/InfoCenter/index?page=content&id=KB20570
[38] P. Musubi, "Data loss on atom editor," 2020. [Online]. Available: https://github.com/atom/atom/issues/11406
[39] D. Lebedev, "Data loss on XFS file system," 2020. [Online]. Available: https://superuser.com/questions/84257/xfs-and-loss-of-data-when-power-goes-down
[40] C. Patry, "Data corruption on docker container," 2020. [Online]. Available: https://github.com/scality/cloudserver/issues/662
[41] B. Duddridge, "SQLite file corruption," 2020. [Online]. Available: https://github.com/couchbase/couchbase-lite-ios/issues/1482
[42] R. Fay, "Database corruption on docker," 2020. [Online]. Available: https://github.com/drud/ddev/issues/748
[43] Benny, "LevelDB database corruption," 2020. [Online]. Available: https://github.com/google/leveldb/issues/733.
[44] S. Frampton, "File system corruption after power outage or system crash," 2020. [Online]. Available: https://www.tldp.org/LDP/lame/LAME/linux-admin-made-easy/crash-repair.html
[45] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen, "On the use of stack traces to improve text retrieval-based bug localization," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 151–160.
[46] D. Watson, "Libunwind documentation," 2020. [Online]. Available: https://www.nongnu.org/libunwind/docs.html
[47] H. Franke, R. Russell, and M. Kirkwood, "Fuss, futexes and furwocks: Fast userlevel locking in Linux," in *Proc. AUUG Conf.*, 2002, pp. 479–495.

Abdullah Alourani (Student Member, IEEE) received the bachelor's degree in computer science from Qassim University, Buraydah, Saudi Arabia, and the master's degree in computer science from DePaul University, Chicago, Illinois. He is currently working toward the PhD degree in the Department of Computer Science, University of Illinois at Chicago, Chicago, Illinois. His current research interests include the areas of cloud computing, distributed systems, and software engineering. He is a student member of the ACM.

Ajay D. Kshemkalyani (Senior Member, IEEE) received the BTech degree in computer science from the Indian Institute of Technology, Bombay, Mumbai, Maharashtra, India, in 1987, and the PhD degree in computer science from the Ohio State University, Columbus, Ohio, in 1991, respectively. He is currently a professor with the Department of Computer Science, University of Illinois at Chicago. His research interests include distributed computing, computer networks, and concurrent systems. In 1999, he received the US National Science Foundation Career Award. He has served on the editorial board of the *Elsevier Journal Computer Networks*, and the *IEEE Transactions on Parallel and Distributed Systems*. He has coauthored a book entitled *Distributed Computing: Principles, Algorithms, and Systems* (Cambridge University Press, 2011). He is a distinguished scientist of the ACM.

Mark Grechanik (Senior Member, IEEE) received the PhD degree in computer sciences from the University of Texas at Austin, Austin, Texas. He is currently an associate professor with the Department of Computer Science, University of Illinois at Chicago. He has served as a member of ACM SigSoft Executive Committee since 2009. His research area is software engineering in general, with particular interests in software testing, evolution, and reuse. He is also interested in problems that lie at the intersection of software engineering and data privacy. He has a unique blend of strong academic background and long-term industry experience. He is a senior member of the ACM.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Probabilistic Consistency Guarantee in Partial Quorum-Based Data Store

Xin Yao, *Student Member, IEEE* and Cho-Li Wang, *Senior Member, IEEE*

Abstract—Many NoSQL databases support quorum-based protocols, which require a subset of replicas (called a quorum) to respond to each write/read operation. These systems configure the quorum size to tune the operation latency and adopt multiple consistency levels. Some recent works illustrate that using probability models to quantify the chance of reading the last update is important because it could avoid returning stale values under eventual consistency. There are two challenging issues: (1) from inconsistent replicas, how to determine the minimum quorum size (i.e., the lowest access latency) to read the newest data at a specified probability; (2) node failure frequently happens in large-scale systems, how to guarantee the probability-based consistent reads. This article presents Probabilistic Consistency Guarantee (PCG), which is the first dynamic quorum decision and failure-aware quantification model. PCG model respectively quantifies the server-side consistency after the latest write, which reflects the object's time-varying update progress, and the possibility of reading this update when responding to the end-users. Our theoretical analysis derives several formulas to determine the quorum size of a read quorum and the consensus result selected from this quorum is the data updated by the last write at the user-specified probability. When some replicas are unavailable, our model knows how to rescale the quorum and read values from surviving replicas could reduce the stale reads caused by node failures. The experimental results in Cassandra demonstrate that the PCG model can achieve up to 77.7 percent more accurate predictions and reduce up to 48.9 percent read latency than those of the previous model.

Index Terms—Quorum-replicated database, eventual consistency, probabilistic consistency model

1 Introduction

Modern distributed database systems could support many popular applications, such as cloud database services [1], [2], [3], geographically distributed data store for social networks [4], and parameter storage system for distributed deep learning [5]. For large-scale deployment, these systems usually use replicas to balance increasing workloads and provide better reliable service, e.g., Microsoft Azure reduces query latency by using up to 12 replicas to handle over millions of requests [6]. When considering consistency, availability, and partition tolerance, the replica-based system can only achieve two of them simultaneously [7], [8]. Brewer *et al.* [7] summarized this trade-off as the CAP conjecture, which is proved by Gilbert and Lynch [8]. Abadi *et al.* [9] improved the theory by proposing a trade-off between consistency and latency when failures are absent. Quantifying data consistency described in these theories is critical [10], [11], [12] because it can evaluate the staleness of data returned from the eventually consistent systems in many scenarios while offering significant latency improvements.

A large-scale quorum-based datastore, which keeps N replicas of each data item, has been widely deployed in cloud storage systems [1], [2], [13], grid [14], federated environment [15] and so on. When a write updates a data item, the coordinator sends this write to all replicas and waits for the responses from W replicas (write quorum). Similarly, to read a data item, the coordinator returns the most updated value observed from a read quorum of randomly chosen R replicas [16]. The strict quorum systems [17], [18] ($W + R > N$) require that write quorum and read quorum are overlapped to guarantee strong consistency. Some previous studies [10], [16], [19], [20], [21] introduce the partial quorum system ($W + R \leq N$), which allows the coordinator to contact fewer replicas. It offers eventually consistency [22] and reduces the latency, but sometimes reads stale value. As a remedy, existing quorum systems (e.g., Dynamo [1] and Cassandra [2]) provide multiple consistency levels (ONE, QUORUM, ALL, etc.) by configuring different quorum sizes. Adjusting quorum properties (e.g., the quorum size) could adapt to different user workloads while still providing correctness is a challenge for faster data access.

We analyze data staleness in partial quorum systems under a global timestamp assumption, which can be achieved by using NTP to synchronize each server [11]. This assumption is widely used in research papers and real systems [10], [11], [20]. Previous models have serious concerns because they simplify the quantification by ignoring the intermediate states of updating N replicas. Probabilistically Bounded Staleness (PBS) [10], [20] described a formula[1] to calculate the probability of reading a k-staleness value *t seconds* after the last version was written. Such quantification successfully explained why the users could often read the most updated data even from the inconsistent

• *The authors are with the Department of Computer Science, University of Hong Kong, Hong Kong. E-mail: {xyao, clwang}@cs.hku.hk.*

Manuscript received 21 Aug. 2019; revised 20 Dec. 2019; accepted 1 Feb. 2020. Date of publication 13 Feb. 2020; date of current version 26 Mar. 2020. (Corresponding author: Xin Yao.)
Recommended for acceptance by W. Yu.
Digital Object Identifier no. 10.1109/TPDS.2020.2973619

1. The basic formula in PBS is $1 - \binom{N-W}{R} / \binom{N}{R}$. In a configuration example (100, 30, 30), the probability of two quorums do not overlap is less than $1.88 * 10^{-6}$ [23].

1045-9219 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

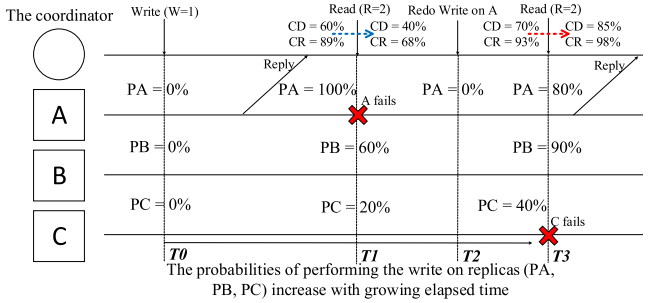


Fig. 1. Consistency quantification: calculating CD and CR during the inconsistency window [26] by using PCG model.

systems. However, when determining the read quorum size, PBS's formula derives an equation that is hard to solve and it [24] uses trial and error [25] to find a sub-optimal answer. Probabilistic CAP (PCAP [11]) introduced a control loop to search the adaptive control knob, which prefers delaying the read query (i.e., causing a longer write latency) to meet the consistency requirements to read a *t-freshness* data. Harmony [12] provided a formula to compute the minimal read quorum size by assuming all operations arrivals follow the Poisson distribution. Different from the assumption, practical workloads usually have various patterns (e.g., write/read intensive workloads) during both peak and off-peak periods. Furthermore, once node failure happens, these models cannot reflect how unavailable replicas affect the consistency analyses. To fill these gaps, we introduce two metrics to separately quantify the server-side consistent update (we call it Consistency Degree) and the client-side consistent read (we call it Consensus Ratio) (Section 3.1).

Consistency Degree ($CD[(K, V_x, t_x)]$). It describes the probability of complete updates on available replicas of a data item (K, V), t_x after the write (Write(K, V_x)). Since the same key could be updated by different writes, these replicas may keep multiple versions of data. As the time intervals (i.e., inconsistency window) to finish updating those replicas are different, only some replicas have the newest data and the read comes during the inconsistency window may get stale data. To calculate the percentage of updated replicas in the inconsistent system, we quantify the possibility of each replica has finished the last write at an instance of time t_x as its time-based update progress.

Consensus Ratio ($CR[(K, V_x, t_x)]$). Based on the time-based update progress (determined by the time intervals t_x) of each replica and the read quorum size R, it can calculate the possibility of reading the value updated by this write. For instance, in Fig. 1, a read reaches the system at T_1 after a write, three replicas respectively have 100, 60 and 20 percent chance of being updated to V_x. Given the read quorum size, we derive the consensus ratio (e.g., 89 percent if R equals 2).

Contributions. Different from previous studies, PCG model solves the following concerns in traditional models based on *CD* and *CR*: (1) given a bounded time t_x after the last write, our model can dynamically calculate the minimum quorum size (i.e., the shortest latency) to read the updated data at a probability (CR), e.g., R is 2 to guarantee 80 percent chance of reading this last write (see T_1 in Fig. 1 before A fails), (2) we consider node failure and the system-wide update progress (CD) may change (CD reduces from 60 to 40 percent because A fails at T_1, while it increases from 70 to 85 percent because C fails at T_2); quorum size is also tuned to keep required consistent reads; we also quantify the trade-off between resume time and staleness of recovered data.

- We define two metrics, namely CD and CR, to analyze eventually consistency. CD shows time-based update progress of a given write on all replicas, while CR is the probability of reading this write. Compared with consistency quantification in previous models, our formulas are succinct but more accurate.
- We derive a formula and some variants to reveal the relationship between CD and CR. One of these variants ($R \geq \log_{(1-CD)}(1 - CR)$) can directly determine the adaptive quorum size for each consistent read at any given probability and reduce the read latency.
- We extend our analysis in the case of reading k-staleness data. When unavailable replicas appear, our model also reveals the change of system-wide update progress and rescales up/down the quorum size to meet the required probability of each consistent read. We demonstrate the whole model in Cassandra [2].

2 Motivation and Example

2.1 Motivation

Some replicas might have been updated before the coordinator receives their responses of the successful write (i.e., the replica which is not a write quorum or late-responded). Previously quantitative models [10], [19], [25] (e.g., PBS, HARP) ignore these possibilities and this may lead to errors in consistency quantification. The asymmetry network, which means different latency distributions exist among the coordinators and replicas (see Fig. 4 in Section 4.1.1), complicates the analyses on the consistency degree. Our model evaluates latency distributions by collecting the recent latencies across machines. In Fig. 1, we can calculate PA (100 percent), PB (60 percent), and PC (20 percent) based on the historical latency data, thus derive the consistency degree (60 percent, the mean value of PA, PB, and PC). According to PBS, the probability of reading the write is $1 - \binom{3-1}{2}/\binom{3}{2} = 67\%$ at T_1, because of ignoring PB and PC. The practical probability is larger than 67 percent since the read still has opportunities to read the updated value from replica B and C even they do not reply to the coordinator.

Previous prediction models are also hard to be easily applied in tuning the operation latency according to the consistency requirement (e.g., given a probability of reading the last update). If the user wants to have at least 70 percent chance of reading the write at T_1, previous mechanisms try all possible R (e.g., increasing from 1 to N) to find an adaptive one that the returned value meets the user-specified consistency requirement. There are two reasons: (1) these formulas cannot calculate the exact read quorum size (R),[2] (2) the results strongly rely on the assumptions of user

2. In PBS, $1 - \binom{N-W}{R}/\binom{N}{R} = CR$ is difficult to calculate R with given N, W, and CR, especially when N and W are large. It can only use trial and error.

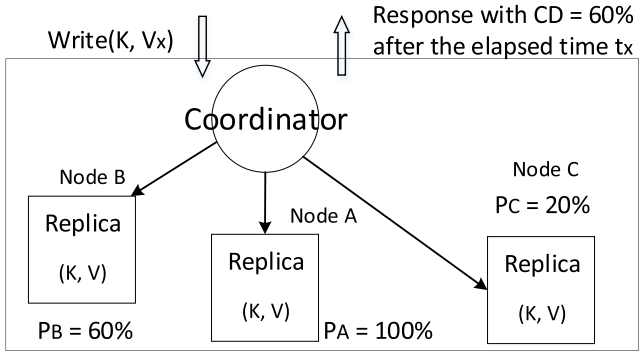


Fig. 2. Diagram of a write operation. P_A (100%) is the probability replica A has applied this operation t_x after this write, P_B (60%) and P_C (20%) are in the same case. The consistency degree is 60%.

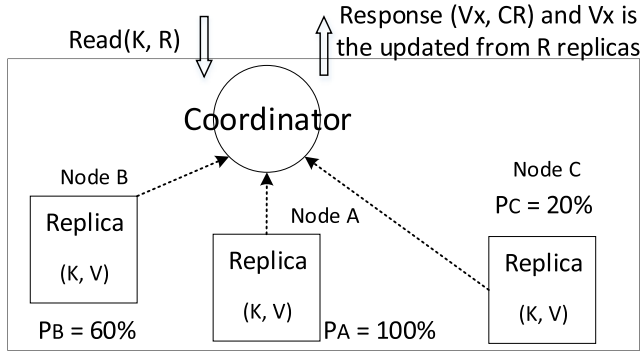


Fig. 3. Diagram of a read operation (t_x after the write). The system returns the most updated data from R (read quorum size) replicas. The consensus ratio of reading this write increases with R growing from 1 (60%) to 3 (100%).

behaviors (e.g., read and write operations arrived according to a Poisson process in Harmony [12]).

Furthermore, practical systems also encounter many failures, such as power fault and network partition. Most previous studies do not consider these failures and the system still keeps the read quorum size R and returns the updated value from surviving replicas. Given the same R, the system is difficult to approximate the desired probability of consistent read in normal cases because some replicas are unavailable and they could cause different impacts on the consistency. A work [24] based on PBS observes consistency to change when a node fails, while it cannot figure out how to adjust the read quorum size to guarantee the user-specified consistency level. In PCG model, we can calculate CD under node failures: (1) the lost replica is highly synchronized (the blue dotted arrow at T_1), the consistency degree reduces and the consensus ratio also decreases (i.e., a higher probability of stale reads); (2) the lost replica has a low probability of being updated (the red dotted arrow at T_3), the consistency degree increases and users are more likely to read the updated value (i.e., users do not need the same read quorum size as before). Based on the analyses, it furthers tune the read quorum size that guarantees the CR to avoid stale reads or longer read latencies. Compared with using PBS: (1) PCG explains why consistency is improved/reduced when a node fails; (2) PCG's formula determines the exact read quorum size; (3) PCG can also discuss the recovery process on newly allocated replicas and counter them when determining R.

2.2 An Example

To help understand these new concepts, we illustrate them in the distributed storage system with a replication factor 3. For a given key K, we assume there are replica A, B, and C.

After a write operation (see Fig. 2), these replicas have different probabilities of finishing this operation, caused by different network latencies. Replica A has completed this write (i.e., 100 percent), the probability of that replica B receives this write is 60 percent, and the probability of updating the value on replica C is only 20 percent due to a long network delay. Therefore, the consistency degree of applying this write to the data item in the 3-replicated storage system is $\frac{1}{3} * (100\% + 60\% + 20\%) = 60\%$. In Fig. 3, we can calculate the consensus ratio based on the read quorum size and the probability of finishing this operation on each replica.

We first assume R equals 1. There are $\binom{3}{1} = 3$ cases: (1) get the result from replica A, the probability of getting stale value is $(1 - 100\%) = 0$, (2) get the result from replica B, the probability of getting stale value is $(1 - 60\%) = 40\%$, (3) get the result from replica C, the probability of getting stale value is $(1 - 20\%) = 80\%$. Therefore, the probability of getting stale value is $\frac{1}{3} * (0 + 40\% + 80\%) = 40\%$, while the consensus ratio of the most updated value is $1 - 40\% = 60\%$.

If the read quorum size is 2, there are $\binom{3}{2} = 3$ cases: (1) get the result from replica A and replica B, the probability of getting stale value is $(1 - 100\%) * (1 - 60\%) = 0$, (2) get the result from replica B and replica C, the probability of getting stale value is $(1 - 60\%) * (1 - 20\%) = 32\%$, (3) get the result from replica C and replica A, the probability of getting stale value is $(1 - 20\%) * (1 - 100\%) = 0$. Therefore, the probability of getting stale value is $\frac{1}{3} * (0 + 32\% + 0) = 10.67\%$, while the consensus ratio becomes $1 - 10.67\% = 89.33\%$.

With the same consistency degree, the quorum that consists of two replicas can be more likely to read the up-to-date value than that only has one replica ($89.33\% > 60\%$). In the absence of failures, R could be 1 to guarantee a 60 percent chance of reading the updated value; while R should be 2 when the user specifies a stronger consistent read (e.g., 70 percent). When R equals 3, users must read the most recent value in the example because replica A is strongly synchronized.

To consider the node failure case, we first assume replica A is unavailable and the consistency degree ($\frac{1}{2} * (60\% + 20\%) = 40\%$) will decrease. In case of accessing one replica, the system can only get value from replica B or C, and the consensus ratio becomes 40 percent. It reduces almost 20 percent probability of reading the updated value and the system cannot serve the queries with specifying CR higher than 40 percent (e.g., 60 percent). To catch up on this gap, the system can increase the read quorum size. Collecting results from two replicas can improve the consensus ratio ($1 - (1 - 60\%) * (1 - 20\%) = 68\%$). On the other hand, if the lost replica is replica C, consistency degree rises to be $\frac{1}{2} * (100\% + 60\%) = 80\%$. Compared with $R = 2$ in the normal system to guarantee a 70 percent chance of reading the updated value, the read quorum size could be optimized to one (i.e., a shorter latency) after the replica C fails.

3 System Modeling

3.1 Consistency Degree Modeling

Each write operation applies a new version of a data item and breaks the consistent state among its all replicas. The coordinator broadcasts this update to N replicas and only waits for W responses. Before the coordinator receives the response from a replica, PBS cannot ensure whether this replica finishes the write or not. Different from PBS, we quantify the possibility of updating the value on each replica whether it responds to the coordinator or not, thus our prediction considers the start time of the write operation. Compared with using PCAP (O($nlog(n)$), n is the length of the log list that records the historical operations), our model's computational complexity (O(N), N is the number of replicas) is also reduced. In most cases, N is less than n.

We conduct probabilistic methods to quantify the possibility of a replica finishes a write, concerning two aspects: (1) network latency between the coordinator and the replica is expressed as a one-way delay and follows a probability distribution, (2) the elapsed time after the coordinator sends the write to the replica. Note that, we use $[(K, V_x)]$ to stand for $[(K, V_x, t_x)]$, which means a write (Write(K, V_x)) is sent to a data item and the elapsed time after this write is t_x.

Definition 1 $\mathbf{P_i}[(\mathbf{K}, \mathbf{V_x}, \mathbf{t_x})]$. *A write (Write(K, V_x)) comes to the system, $P_i[(K, V_x)]$ is the probability that the ith replica has been updated to V_x and the elapsed time after this write is t_x ($t \in [0, t_x]$)*

$$P_i[(K, V_x)] = P_i(t_x) = \int_0^{t_x} Distribution(t)dt$$

The replication factor of each data item is N. To calculate the percentage of complete updates on these replicas, we need to evenly consider $P[(K, V_x)]$ of all replicas:

Definition 2 Consistency Degree (CD$[(\mathbf{K}, \mathbf{V_x}, \mathbf{t_x})]$). *consistency degree of the data item (CD[(K, V_x)]) quantifies the possibility of all its replicas are updated by a write operation (Write(K, V_x)) after the elapsed time t_x.*

Naturally, we compute the mean value of all replicas' probabilities of being updated to V_x. It can be used to evaluate the consistency degree of the data object in the system

$$CD[(K, V_x)] = \frac{1}{N}\sum_{i=1}^{N} P_i[(K, V_x)] = \frac{1}{N}\sum_{i=1}^{N} P_i(t_x). \tag{1}$$

3.2 Consensus Ratio Modeling

Since all replicas of a data item are not consistent, the results returned to reading this data item are different. With knowing the probability of updating a write on each replica, we can quantify the possibility of reading this write.

Definition 3 Consensus Ratio (CR$[(\mathbf{K}, \mathbf{V_x}, \mathbf{t_x})]$). *A read comes to an inconsistent system t_x after a write, consensus ratio (CR[(K, V_x)]) is the probability of the responding result is V_x (i.e., the updated value) via some agreement protocols (e.g., the most recent value from R replicas involved in the read operation).*

According to the Definition 3, consensus ratio also involves two variables: (1) the time interval between the read and the write (Write(K, V_x)), (2) the read quorum size R. In a quorum-based system, the users may get different values from different replicas because they are not synchronized after a write. To respond to a read with R read quorums, the system can return the most updated value from them. There are total $\binom{N}{R}$ cases to select R members from N replications, while we assign all possible combinations as a set $S = \{s_j \mid s_j \subset \{1, 2, \ldots, N\} \text{ and } |s_j| = R\}$.

In Definition 1, $P_i[(K, V_x)]$ is the probability of the ith replica performs Write(K, V_x) when the read operation comes (t_x after this update).

Definition 4. *For a given element s_j from S, we define X_j as the probability of the user cannot read the most updated data from every replica in s_j*

$$X_j = \prod_{i \in s_j} (1 - P_i[(K, V_x)]).$$

Since the system evenly picks one element from S, the probability of reading results from R replicas in each s_j is $\frac{1}{\binom{N}{R}}$. We can get the expectation of CR

$$\begin{aligned} CR[(K, V_x)] &= 1 - \frac{1}{\binom{N}{R}} \sum_{s_j \in S} (X_j) \\ &= 1 - \frac{1}{\binom{N}{R}} \sum_{j=1}^{\binom{N}{R}} \left(\prod_{i \in s_j} (1 - P_i[(K, V_x)]) \right) \\ &= 1 - \frac{1}{\binom{N}{R}} \sum_{j=1}^{\binom{N}{R}} \left(\prod_{i \in s_j} (1 - P_i(t_x)) \right). \end{aligned} \tag{2}$$

If W replicas (the write quorums) have already replied successful write to the coordinator, their probabilities should be quantified as 1. In Definition 4, $X_j = 0$ when s_j contains at least one write quorum, which is sure to update the most recent write ($1 - P[(K, V_x)] = 0$). We further define a set S', which consists of s_j from S and each s_j does not contain any write quorum

$$\begin{aligned} CR[(K, V_x)] &= 1 - \frac{1}{\binom{N}{R}} \sum_{s_j \in S'} \Big(\prod_{i \in s_j} (1 - P_i(t_x)) \Big) \\ &= 1 - \frac{1}{\binom{N}{R}} \sum_{j=1}^{\binom{N-W}{R}} \Big(\prod_{i \in s_j} (1 - P_i(t_x)) \Big). \end{aligned} \tag{3}$$

In Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2020.2973619, we prove Bailis's formula [10], [20] is a particular case of our formula.

3.3 Lower Bound for Consensus Ratio

Equation (2) can accurately calculate the expectation of CR. However, when N and R become larger, the overhead of computing the precision result becomes significant (the computational complexity is around $O(R * \binom{N}{R})$). Similar to previous studies, it is impossible to directly apply

Equations (2) or (3) to calculate the adaptive read quorum size. Therefore, we provide the lower bound of the Equation (2).

Theorem 3.1. *A read comes to the system t_x after a write. It needs to get the values from R replicas and returns the most up-to-date value as the responding result with a probability. The lower bound of this probability is*

$$1 - (1 - CD[(K, V_x)])^R.$$

According to the Muirhead's Inequality [27], we can derive a formula in a much simpler form, whose value is smaller than the Equation (2) (for each $P_i(t_x) \in [0, 1]$)

$$\begin{aligned} CR[(K, V_x)] &= 1 - \frac{1}{\binom{N}{R}} \sum_{j=1}^{\binom{N}{R}} \left(\prod_{i \in s_j} (1 - P_i(t_x)) \right) \\ &\geq 1 - \frac{1}{\binom{N}{R}} \left(\binom{N}{R} \left(\frac{\sum_{i=1}^{N} (1 - P_i(t_x))}{N} \right)^R \right) \\ &= 1 - \left(\frac{N - \sum_{i=1}^{N} P_i(t_x))}{N} \right)^R. \end{aligned} \quad (4)$$

In Definition 2, $\frac{1}{N} \sum_{i=1}^{N} P_i(t_x) = CD[(K, V_x)]$. We replace the sum of $P_i[(K, V_x)]$ in Equation (4)

$$CR[(K, V_x)] \geq 1 - (1 - CD[(K, V_x)])^R. \quad (5)$$

In Equation (5), when the consistency degree of the data object is fixed, the lower bound of consensus ratio is increasing with accessing more number of replicas. Especially, the consensus ratio equals to 1 when all replicas in the system are fully synchronized

$$1 \geq \lim_{CD \to 1} CR \geq \lim_{CD \to 1} 1 - (1 - CD)^R = 1.$$

Theorem 3.1 presents the lower bound of the probability that the user can read the updated data item (K, V_x). With the consistency degree given by Equation (1) and the read quorum size of each read operation, we can quantify how staleness of the data the system returns to the read with at least what probability. Moreover, compared with Equation (2), the computational complexities of Equation (5) (O(N)) are greatly reduced. On the other hand, the variant of Equation (5) allows the user to set a probability as CR and then calculates the minimum read quorum size R to optimize the read latency.

Corollary 3.1.1. *A read comes to the system t_x after a write. CD [(K, V_x)] is the probability of completed update on the system. The user wants to read this write with the probability no less than CR[(K, V_x)], the read quorum size should be*

$$R \geq \log_{(1 - CD[(K, V_x)])} (1 - CR[(K, V_x)]). \quad (6)$$

Corollary 3.1.2. *(*with knowing the write quorum*) A read comes to the system t_x after a write. W replicas have replied successful write to the coordinator. The lower bound of Equation (3) is*

$$CR[(K, V_x)] \geq 1 - \frac{\binom{N-W}{R}}{\binom{N}{R}} \left(\frac{N(1 - CD[(K, V_x)])}{N - W} \right)^R. \quad (7)$$

Correspondingly, we can calculate R in this case to satisfy the user-specified CR[(K, V_x)]

$$R \geq \log_{\frac{N(1 - CD[(K, V_x)])}{N - W}} (1 - CR[(K, V_x)]). \quad (8)$$

In Appendix B, available in the online supplemental material, we present the proof of Corollary 3.1.2[3] and show why R (the read quorum size) calculated via Equation (6) can also cover the cases in this corollary.

Some reads will trigger the read-repair mechanism in Cassandra. The read coordinator could asynchronously update the out-of-date replicas of a read quorum with the most recent version. It could be regarded as replaying a write operation (i.e., write the most recent value again) on those out-of-date replicas and we further discuss it in Section 3.4.

3.4 Probability Guarantee in Nodes Failure

When failures happen and the ith replica is unavailable, the system cannot read the data item from this replica. To provide high availability for user queries with the surviving replicas, we reappraise the system metrics after removing all lost replicas from the quorum-based system. Different from previous works, our analyses find, under some failure cases (e.g., the replica with a low probability of applying the most recent update), the users could have a less chance of reading data from those stale replicas because they are inaccessible. In the other cases, there are two ways to avoid stale reads: (1) increasing the read quorum size but it is limited by the number of the surviving replicas, (2) waiting for a longer time, which can improve consistency degree.

First, we need to recalculate the consistency degree of the data item by counting only the surviving replications after node failures.

Corollary 3.1.3. *Node failure happens (r replications are unavailable) in the system t_x after a write. Only the surviving replicas can be accessed. According to Definition 2, we regard all surviving replicas as a new system (Q_s) with (N-r) replicas and the consistency degree of this new system is*

$$CD_s[(K, V_x)] = \frac{\sum_{i \in Q_s} P_i[(K, V_x)]}{N - r} = \frac{\sum_{i \in Q_s} P_i(t_x)}{N - r}. \quad (9)$$

A read comes to the system. Only W_s surviving replicas have responded successful write (Write(K, V_x)) to the coordinator. Similar to Equations (7) and (8), the lower bound of consensus ratio is

$$CR_s[(K, V_x)] \geq 1 - \frac{\binom{N-r-W_s}{R}}{\binom{N-r}{R}} \left(\frac{(N - r)(1 - CD_s[(K, V_x)])}{N - r - W_s} \right)^R,$$

3. When W equals N, $1 - CD[(K, V_x)] = 0$ and R = 1 because N replicas are consistent in this case. Similar case ($CD[(K, V_x)]$=1) in Equation (6). Equation (7) works when $W + R \leq N$ (partial quorum); $CR[(K, V_x)]$=1 when $W + R > N$.

and the rescaled read quorum size is

$$R_s \geq \log_{\frac{(N-r)(1-CD_s[(K,V_x)])}{N-r-W_s}}(1 - CR_s[(K, V_x)]).$$

Corollary 3.1.3 is based on Corollary 3.1.2. It shows three factors in node failures, which affect the probability of updated reads: (1) the number of unavailable replicas (r), (2) the number of surviving replicas that belong to the write quorum (W_s), (3) the new consistency degree ($CD_s[(K, V_x)]$). These factors are also related, for example, the consistency degree must reduce when all unreachable replicas belong to the write quorums. Especially, when W_s equals 0, we can simplify the formulas in Corollary 3.1.3. Therefore, we calculate the consensus ratio (using Equation (5)) or the read quorum size (using Equation (6)) according to the new consistency degree after failures.

The system has two ways to recover the lost replicas: (1) replaying the write operations on the newly allocated replicas, (2) copying the value from the surviving replicas. In the first way, we only need to recalculate the $P[(K, V_x, t_x)]$ for each new replica, t_x after replaying the write (Write(K, V_x)) on it. In the second case, we directly use Corollary 3.1.3 to calculate the probabilities of these new replicas that are updated. However, these probabilities will not grow after recovery. When the system randomly chooses R replicas to respond to a read operation, it creates dependency[4] between the new replicas and those replicas which are used for recovery. The Cassandra's read-repair mechanism [24] is to "replay" the last write operation on the outdated replicas of a read quorum, which is the same with replaying the last write on newly allocated replicas in node failure cases. It changes P_i of those replicas and CD in the whole system.

3.5 Quantifying Staleness

The practical system usually processes multiple writes on the same data item. Staleness exists because some replicas do not finish the most recent write and still keeps the value updated by the former write operations. A read operation might get a k-staleness value, whose version equals one of the versions written by the last k+1 preceding writes [20]. In general cases, these writes are independent and they might be out of orders when arriving at different replicas.

Definition 5. $\mathbf{P_i[(K, V_{[m-k,m]})]}$: *after performing m update operations (Write(K, V_1), Write(K, V_2), ..., Write(K, V_m)) on a data item with given latencies, $P_i[(K, V_{[m-k,m]})]$ is the probability that the ith replica has been updated to the value, which is one of $\{V_{m-k}, V_{m-k}, \ldots, V_m\}$. (We assume $m \geq k$)*

$$P_i[(K, V_{[m-k,m]})] = 1 - \prod_{x=m-k}^{m} (1 - P_i[(K, V_x)]).$$

Corollary 3.1.4 (extension of reading a k-staleness value). *A data item is updated by m writes. The consistency degree of the data item quantifies that the possibility of all replicas hold a k-staleness value (the value updated by the last k writes) is*

$$CD[(K, V_{[m-k,m]})] = \frac{1}{N}\sum_{i=1}^{N} P_i[(K, V_{[m-k,m]})] = 1 - \frac{1}{N}\sum_{i=1}^{N}\prod_{x=m-k}^{m}(1 - P_i[(K, V_x)]).$$

*A read comes to the system after m writes. W_k replicas have responded successful write (*any one of the last k writes*). Similar to Theorem 3.1 and Corollary 3.1.2, the lower bound of consensus ratio (CR$[(K, V_{[m-k,m]})]$) of reading a k-staleness value is:*

$$1 - \frac{\binom{N-W_k}{R}}{\binom{N}{R}}\left(\frac{N\prod_{x=m-k}^{m}(1 - CD[(K, V_x)])])}{N - W_k}\right)^R.$$

Compare with Equation (7), we can find: (1) $W_k \geq W$, thus $\binom{N-W_k}{R} * (\frac{N}{N-W_k})^R \leq \binom{N-W}{R} * (\frac{N}{N-W})^R$, (2) $\prod_{x=m-k}^{m}(1 - CD[(K, V_x)]) < 1 - CD[(K, V_m)]$. With the same read quorum size, CR$[(K, V_{[m-k,m]}]$ is higher than CR$[(K, V_m)]$. It mathematically explains why the users could have a better chance of reading a k-staleness value than reading the updated value from the inconsistent systems.

4 Implementation and Experiments

The testbed contains 16 nodes, and each node is equipped with 2 quad-core (Intel Xeon E5540 @ 2.53 GHz), 16 GB DDR3 memory, 250 GB RPM SATA hard disks and Broadcom 5709 dual-port Ethernet (1,000 Mb/s). The OS is Ubuntu 18.04. Our cluster is under a local area network (LAN) environment, where the latencies between nodes and coordinators are randomly short or long at different periods. All nodes are synchronized according to a time server using ntpdate [28]. We measure the latency distribution of this cluster and validate our formula by using the Monte Carlo method. We also implement PCG model in Cassandra [2], which is an in-memory NoSQL database.

4.1 Micro-Benchmark

4.1.1 Latency Distribution and Consistency Degree

We measure the latency distribution of the cluster, which is used to calculate the consistency degree. We choose one node as the sender and the other 15 nodes are receivers. The sender broadcasts 10,000 packages to the other 15 nodes via using the ping tool. In Fig. 4, we show the statistical results of the half round-trip time. The points in different colors show the frequency of the latency from the sender to different receivers. Similar to the observation from Ailidani *et al.* [29], the statistical results could be fit to the normal latency distribution (the dotted lines in black) in our cluster.

In Fig. 5, we use the cumulative distribution function (CDF) of the normal distribution to calculate the probability of the completed update on each replica. For each replica, we regard the ratio of the arrived packages to all 10,000 packages as the probability of finishing synchronization. These probabilities are used to evaluate the consistency degree. In Fig. 6, the red dots present the mean value of these probabilities of receiving packages from the sender, while the dotted lines in black are the result of Equation (1).

4. This kind of access problem is our future work, and we will simply introduce it in Appendix C, available in the online supplemental material.

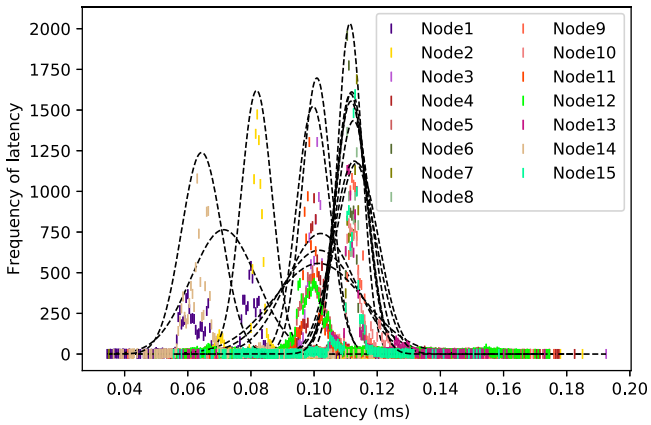


Fig. 4. Latency distribution of the cluster under LAN.

Algorithm 1. Calculating the Lower Bound of CR in Each Read Using Cassandra Nodetool

Input: $N, W, R, t_x, latencies[N]$
Output: $LBCR$: the lower bound of CR

```
1  if N < (W + R) then          // strict quorum
2    return 1;
3  P = new list[N];
4  for i from 1 to N do         // P_i of each replica
5    if i ∈ write quorums then
6      P[i] = 1;
7    else
8      nd = NormalDistribution(latencies[i].mean(), latencies
       [i].std());
9    P[i] = nd.cumulativeProbability(t_x);
   / * With P_i of each replica, we can calculate
   accurate CR via Equations (2) or (3). Here, we
   only calculate the lower bound of CR.  * /
10 CD = sum(P)/N;          // Calculate CD;
11 μ = N/(N-W);
12 α = C(N-W, R) / C(N, R);
13 CR =1-α×math.power(μ×(1-CD), R);      // Get LBCR;
14 return CR;
```

Simulation and experimental results represent a similar tendency and the error is acceptable. However, in more complex cases (e.g., geo-replication), latencies may not follow a normal distribution anymore. They can be described

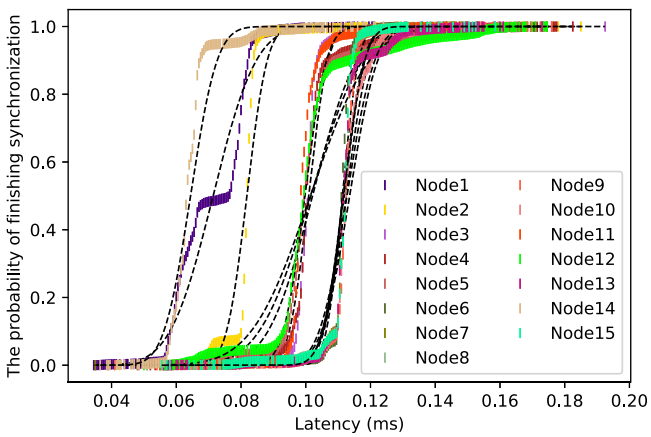


Fig. 5. P_i of each replica in the system with the elapsed time.

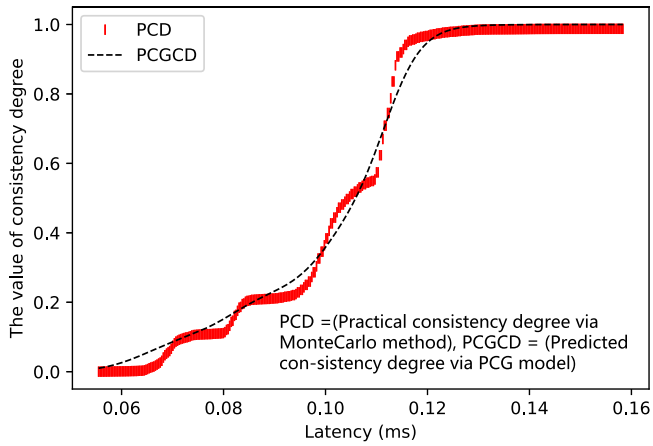


Fig. 6. Measuring consistency degree of the data object with the elapsed time.

in other ways, such as self-similar network traffic [30] and heavy tail distribution. However, we stress that PCG model does not concern itself with which network latency distributions are deployed. Under the same experimental environment, PCG model can perform better than PBS when we calculate the probability of reading the most recent write. We show the results and analyses the reasons in Section 4.2.

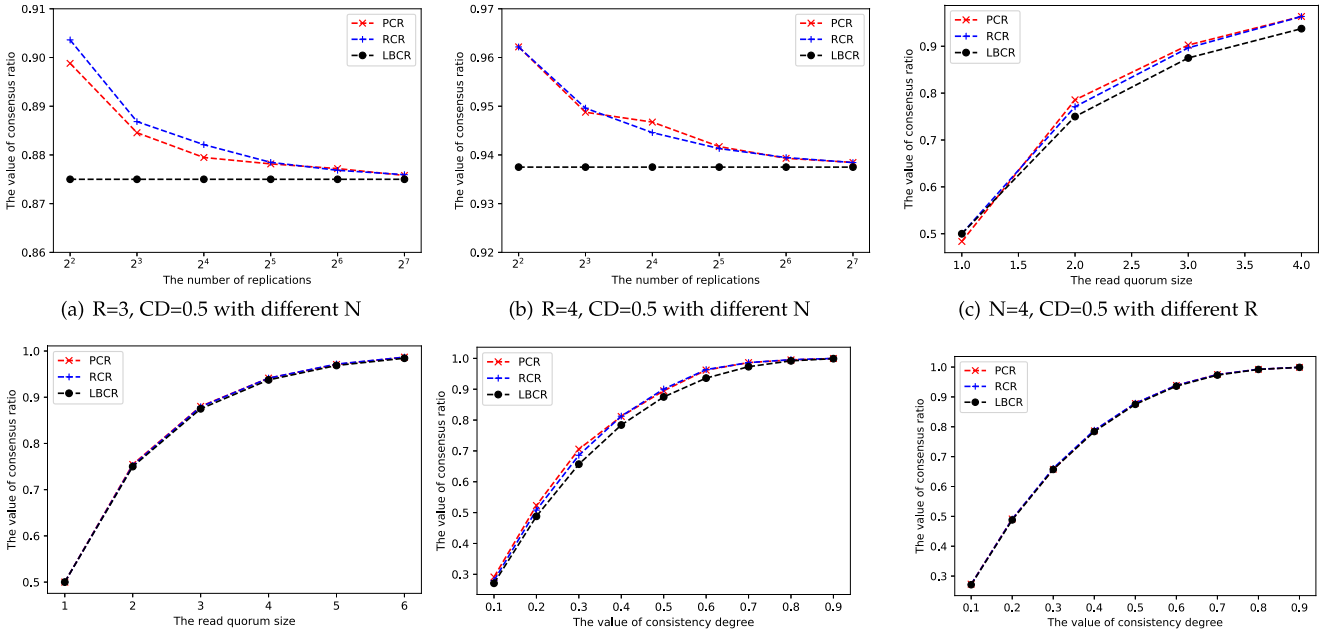
In Cassandra, similar to PBS, we create a new class, namely PCGLatency, and it collects the recent latencies across servers, e.g., the time interval between the coordinator that sends a write and the replica that produces a response. We perform large numbers of writes to acquire enough data about latency for approximating the latency distribution.

4.1.2 Consensus Ratio

We demonstrate Theorem 3.1 and its corollaries in this section. We create multiple processes in each node and each process holds one replica and participates in the synchronization after updates. We try 100,000 queries in Monte Carlo method to count the practical consensus ratio (PCR). The real consensus ratio (RCR) is the accurate prediction via Equation (2), while the lower bound of consensus ratio (LBCR) is calculated by Equation (5). In this simulation, we do not count the write quorums.

In Figs. 7a and 7b, we show how the number of replicas (N) can influence the accuracy of PCG model. With increasing N, both results of PCR and RCR becomes closer to the result of LBCR. It shows the errors between PCR and LBCR is about 2 percent when N is 4 and within 0.94 percent when N is no less than 8. This conclusion proves Equation (5) (i.e., LBCR) can almost stand for the consensus ratio in practical systems. In Fig. 7c, we can see the consensus ratio is higher with more read quorums. Compared with the case N equals 4, the results in Fig. 7d, where N is larger, present fewer gaps between PCR and LBCR. In Fig. 7e, the consensus ratio grows with the increasing consistency degree. We can extract a similar relationship between the accuracy of LBCR and N.

This experiment supports two points: (1) LBCR can be infinitely close to PCR with increasing N, but with less computational complexity (O(N) $< O(R * \binom{N}{R})$), (2) LBCR only relates to CD and the read quorum size R in each read



(a) R=3, CD=0.5 with different N (b) R=4, CD=0.5 with different N (c) N=4, CD=0.5 with different R

(d) N=32, CD=0.5 with different R (e) N=4, R=3 with different CD (f) N=32, R=3 with different CD

Fig. 7. Measuring consensus ratio in cluster, PCR = (Practical consensus ratio via Monte Carlo method), RCR = (Real consensus ratio with accurate calculation), LBCR = (Lower bound of consensus ratio).

operation; there is a positive correlation between LBCR and these two factors.

4.2 Practical Systems: Apache Cassandra

We implement PCG model in Apache Cassandra 3.11.0 [2]. It is deployed on the cluster with 16 nodes and we set the replication factor to 8. We run a mixed workload (100,000 write operations and 100,000 read operations) via the Cassandra stress tool, which aims at collecting the latency between the coordinator and those replicas. These latency records are the input of Algorithm 1.

In this section, we first present the result of quantifying k-staleness data. Then, we show how our mechanism can rescale the read quorum size under node failures via disabling access to the specified nodes. Finally, we port the code of Cassandra-PBS (for Cassandra 1.2) [31] to compare the probability of consistent read with elapsed time.

Different from simulation, in Cassandra, the coordinator can receive the replies from the write quorums. Therefore, we apply Corollary 3.1.2 (see in Algorithm 1) to calculate the consensus ratio (lines 11-13). The replica which finishes the update operation shows its P_i equals one (lines 5-6), while other replicas calculate their P_i according to the normal distribution (lines 8-9).

In Fig. 8, we separately calculate the consistency degree (the lines in blue) and consensus ratio (the lines in red) of different staleness values in the system. The result presents the latency-staleness trade-off in the system. With the same read quorum size (R = 3 in the experiment), the system is more likely to respond a k_1-staleness value than a k_2-staleness value ($k_1 > k_2$) after the same latency.

Compared with Algorithm 1, we sort these replicas by the value of P_i in ascending order and remove the first two replicas (low P_i) or the last two replicas (high P_i) between line 9 and line 10. These steps aim at removing replicas, which have the highest or lowest P_i, and they are not chosen when calculating CR. After blocking accesses to these specified replicas, N minus the number of unavailable nodes and W reduces if the removed replicas belong to write quorums ($P_i = 1$). From the result in Fig. 9, we can draw some conclusions: (1) the consistency degree grows when the unavailable nodes have $P_i < CD(normal)$ (the line in blue), (2) the consistency degree reduces when the unavailable nodes have $P_i > CD(normal)$ (the line in red). Therefore, the same read quorum size may not guarantee the same consensus ratio of reading the most updated value.

To calculate the read quorum size with the specified CR, we only need to replace R of the input with CR and line 13 with $R = math.log(1 - CR, \mu * (1 - CD))$ (see Corollary 3.1.3). In Figs. 10a, 10b and 10c, we can identify two viewpoints: (1) under the same consistency degree, the system involves larger read quorum sizes in queries to satisfy a higher user-specified

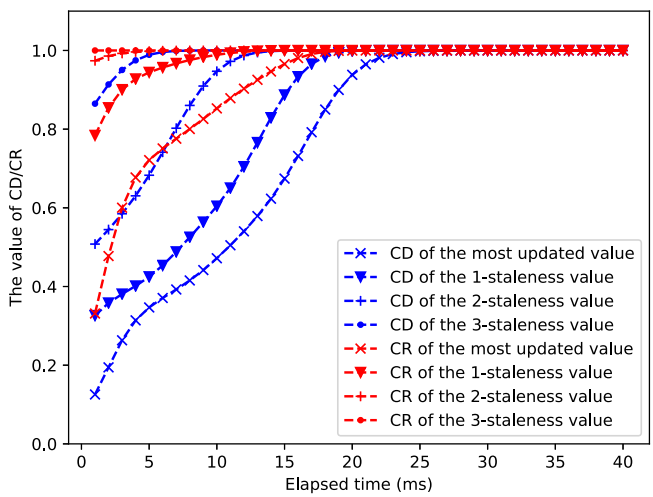


Fig. 8. The CD and CR (R=3 in queries to fetch staleness values) in Apache Cassandra (replication factor = 8).

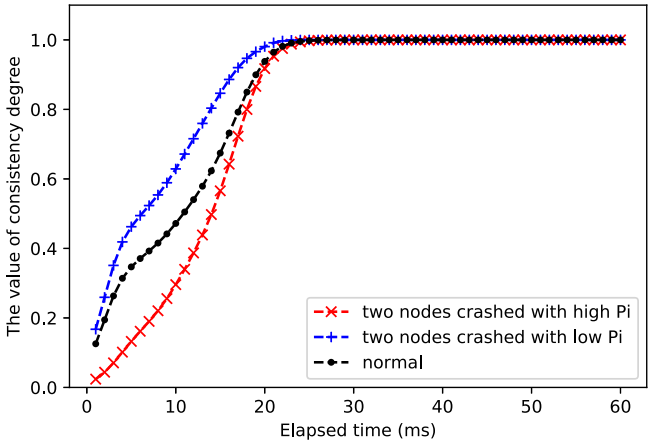


Fig. 9. The consistency degree changes in three cases of Apache Cassandra (replication factor = 8).

consensus ratio. (2) rescaling the read quorum size to maintain the consensus ratio is directly related to P_i of the unavailable replicas. For example, 10 milliseconds after a write: (1) the read quorum size equals one (the black line in Fig. 10a) to ensure $CR \geq 0.3$, (2) the read quorum size equals two (the black line in Fig. 10b) to ensure $CR \geq 0.5$, (3) the read quorum size equals four (the black line in Fig. 10c) to ensure $CR \geq 0.9$. Our mechanism can decrease the read quorum size when the consistency degree grows (the line in blue) and increase the read quorum size when the consistency degree reduces (the line in red). Especially, Fig. 10c shows it needs more than six replicas (i.e., all surviving replicas). As discussed in Section 3.4, the system can only return the result to users after accessing all replicas to achieve such a probability of updated read.

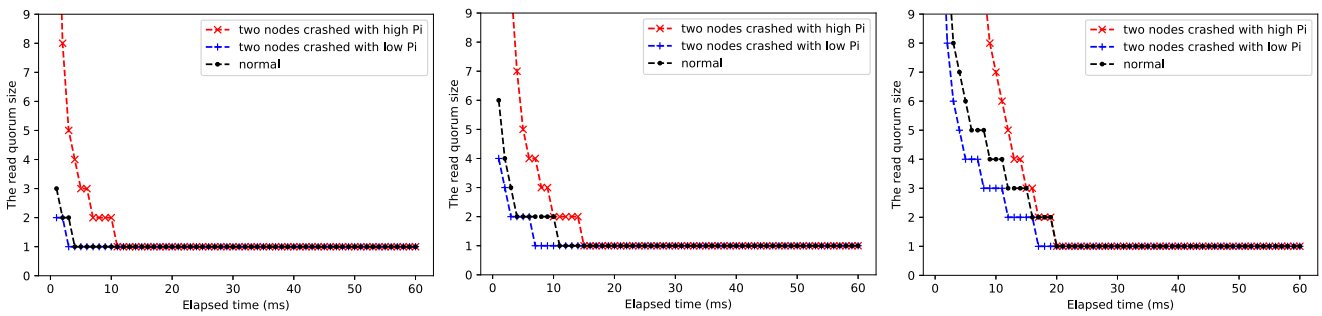
Accuracy. We compare the accuracy of predicting the percentage of reading the most recent write between our model and PBS. To be fair, we set the read quorum size to 3 in this experiment. As discussed in Section 1, PBS needs to decide W and R before measuring the WARS model. Figs. 11a and 11b shows PCGCR (the lines in red) are closer to PCR (the lines in black) than PBS, which means our Formula (7) outperforms PBS with more accuracy: (1) on the 16-nodes Casandra cluster, we evaluate PBS with $W = 5$ (PBS_35) and $W = 6$ (PBS_36). Compared with PBS_36 (the lines in blue), PCG is 65.6 percent closer to the PCR (at 3rd milliseconds after an update), compared with PBS_35 (the lines in green), PCG is 77.7 percent closer to the PCR (at 2nd milliseconds after an update). (2) on the 32-nodes Casandra cluster, we evaluate PBS with $W = 8$ (PBS_38) and $W = 13$ (PBS_313). Compared with PBS_313 (the lines in blue), PCG is 38.8 percent closer to the PCR (at 1st milliseconds after an update), compared with PBS_38 (the lines in green), PCG is 59.6 percent closer to the PCR (at 2nd milliseconds after an update). Each replica will receive the update operation from the coordinator in Cassandra. However, PBS only measures the probability of consistent reads according to the quorums involved in the write operation. Our PCG model considers probabilistic updates on all replicas and the prediction is more accurate.

Latency. To achieve the probability (0.5 in this experiment) of reading the most recent update in the 16-nodes Casandra cluster, the read latency changes based on different timestamps (see Fig. 13a): (1) In PBS, the read quorum size is limited by $W + R \leq N + 1$. PBS cannot determine R at the early stage (e.g., no value returned at 5 milliseconds after an update), while PCG model can dynamically decide the read quorum size at any timestamp; (2) Compared with PBS, PCG model can give the minimum read quorum size, thus respectively reduce the read latency up to 33.3% (PBS(W = 6)), 38.3% (PBS(W = 5)), and 48.9% (PBS (W = 1)). In Fig. 13b, we show the result of achieving at least 90 percent of reading the most recent update in the 32-nodes Casandra cluster and PCG model could reduce the read latency up to 34.3% (PBS(W = 13)), 60.1% (PBS (W = 8)), and 76.8% (PBS(W = 1)).

In Fig. 12, we evaluate the read latency by running a mixed workload in Yahoo Cloud Serving Benchmark (YCSB) [32]. YCSB allows users to assign four kinds of operation: Insert, Update, Read, and Scan. We use three kinds of workloads, while each has 200,000 operations: (1) the read proportion is 0.8 and update proportion is 0.2 under Zipfian distribution; (2) the read proportion is 0.5 and update proportion is 0.5 under Zipfian distribution; (3) the read proportion is 0.5 and update proportion is 0.5 under uniform distribution. Compared with PBS, PCG model respectively reduces the read latency up to 30.5% (PBS(W = 6)), 34.0% (PBS(W = 5)), and 47.1% (PBS(W = 1)).

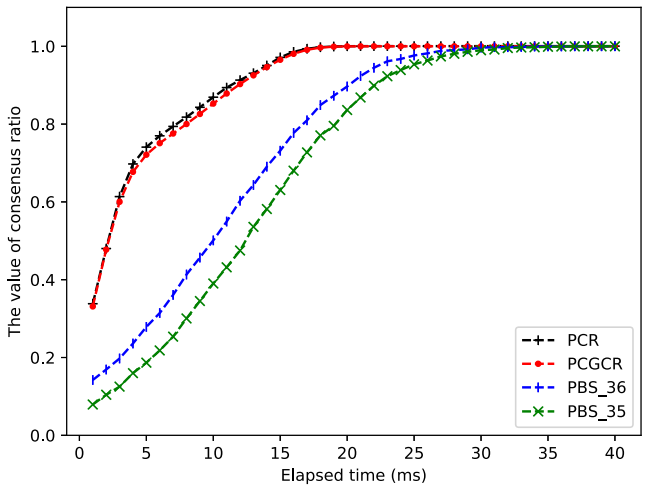
5 Related Work

PBS. The research most related to quantify eventual consistency is PBS model [10], [20]. PBS described the probability of staleness across versions (k-staleness) and time (t-visibility) in the partial quorum systems (N, W, R) [33]. It was based on WARS model [10], which measured the message

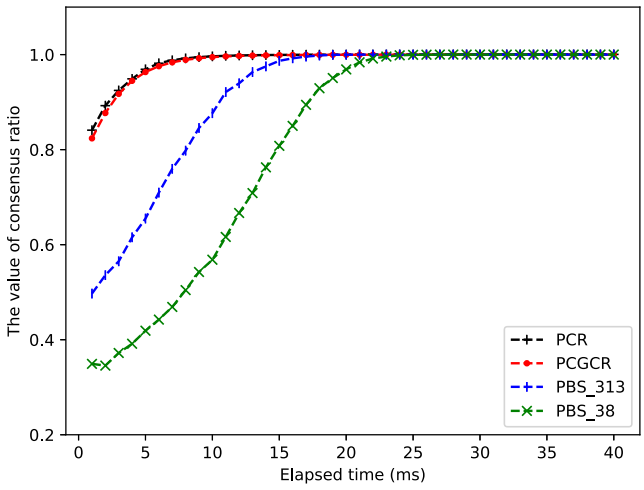


(a) The read quorum size guarantee CR $\geq$ 0.3 (b) The read quorum size guarantee CR $\geq$ 0.5 (c) The read quorum size guarantee CR $\geq$ 0.9

Fig. 10. Rescaling different read quorum sizes to guarantee the consensus ratio of reading the most updated value in three cases of Apache Cassandra (replication factor = 8).

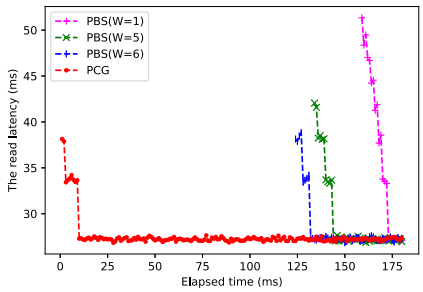


(a) A 16-nodes Apache Cassandra Cluster (N=8). PCR = (Practical consensus ratio via Monte Carlo method), PCGCR = (Lower bound of consensus ratio in PCG), PBS_36 = (PBS with WARS model(R=3, W=6)), PBS_35 = (PBS with WARS model(R=3, W=5)).

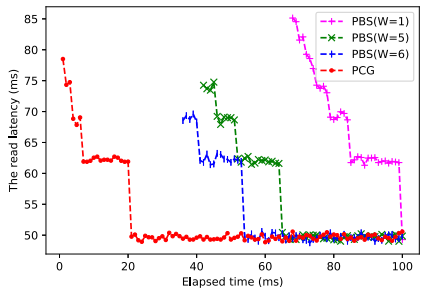


(b) A 32-nodes Apache Cassandra Cluster (N=16). PCR = (Practical consensus ratio via Monte Carlo method), PCGCR = (Lower bound of consensus ratio in PCG), PBS_38 = (PBS with WARS model(R=3, W=8)), PBS_313 = (PBS with WARS model(R=3, W=13)).

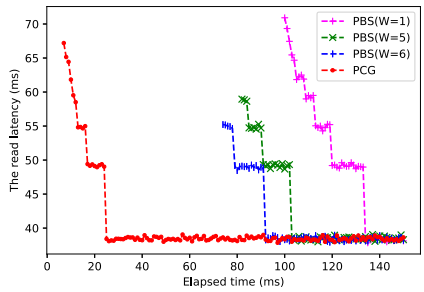
Fig. 11. Measuring CR in Apache Cassandra.



(a) The read proportion is 0.8 and update proportion is 0.2 under Zipf distribution.



(b) Both the read proportion and update proportion are 0.5 under Zipf distribution.



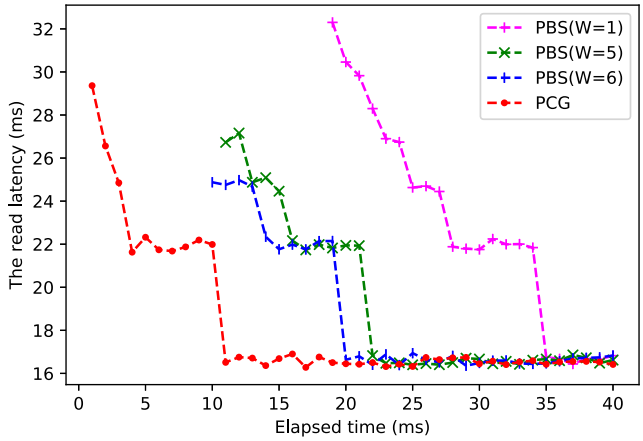
(c) Both the read proportion and update proportion are 0.5 under uniform distribution.

Fig. 12. The read latency of running a mixed workload (20 0000 operations) in 16-nodes Apache Cassandra (N=8) to achieve the probability (0.9) of reading the updated value by using YCSB benchmark. PBS(W=1) = (PBS with W=1), PBS(W=5) = (PBS with W=5), PBS(W=6) = (PBS with W=6), PCGCR = (PCG).

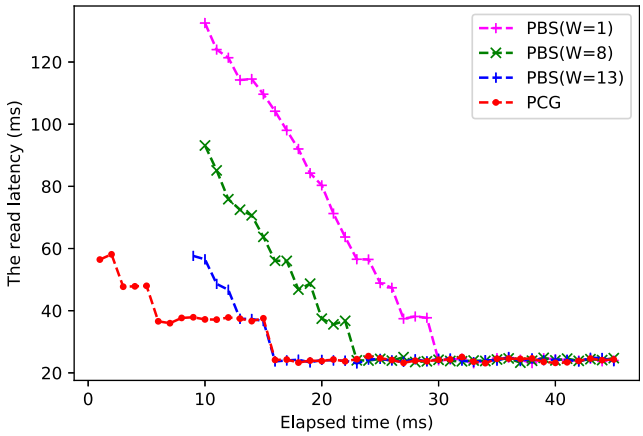
latency (e.g., round-trip time) between the coordinator and the replicas. In PBS, the coordinator needed to wait for W responses when executing write operations, while R responses when collecting results from R replicas. The probability of consistent reads could be decided by N, W and R: $(1 - \binom{N-W}{R}/\binom{N}{R})$ [10], [34]. This idea was wonderful and it examined a fundamental trade-off between operation latency and data consistency. However, when N, W, and R becomes larger, it incurs overhead to evaluate WARS model. Eventual consistency is widely used in the future scalable clusters. A large number of replicas are becoming the norm in all non-trivial use cases, such as large-scale parameter stores and blockchain. Compared with WARS, it is easier to measure the network latency distribution in PCG model. Based on consistency degree, we explore the relationship between the consensus ratio and the read quorum size to develop a tunable probability-based consistency with a tunable probability. It is more accurate and flexible for users to trade off performance against staleness. Our PCG model also considers the possible failures and automatically adjusts the read performance to guarantee the consistent reads. Similar to applying PBS in datacenter networks [34], PCG model is adaptive to different network typologies (it only decides how to calculate P_i) and scenarios.

PCAP. Probabilistic CAP [11] introduced PCAP control loop, which is time-inefficient because of using a multiplicative approach (similar to trial and error) to search an adaptive control knobs to meet the SLA that guarantees only a convergent probability of fresh read. It defined t-freshness, which asked a read operation to read the value written by any write operation. These writes should starts or after a bounded time, which is t seconds before this read. PCAP argued that the main difference between t-freshness and similar concepts in PBS is to consider the start time of each write operation instead of the end time (i.e., when all write quorums replied to the coordinator).[5] Both the calculations of CD and CR can also begin from the start time of write operations and acquire more accurate quantification results. Our model (O(N)) causes less computational complexity than PCAP ((O(n log(n))) and will be suitable for real-time evaluation.

5. However, we found the implementation of PBS in Cassandra [31] can start its prediction from the start time of write operations.



(a) A 16-nodes Apache Cassandra Cluster (N=8, CR=0.5), PBS(W=1) = (PBS with W=1), PBS(W=5) = (PBS with W=5), PBS(W=6) = (PBS with W=6), PCGCR = (PCG).



(b) A 32-nodes Apache Cassandra Cluster (N=16, CR=0.9), PBS(W=1) = (PBS with W=1), PBS(W=8) = (PBS with W=8), PBS(W=13) = (PBS with W=13), PCGCR = (PCG).

Fig. 13. The read latency of running a mixed workload (10 0000 write/read operations) in Apache Cassandra to achieve the probability of reading the updated value by using Cassandra stress tool.

Other Models. Shankha Chatterjee *et al.* [35] proposed a model to describe, for a given key K and value V, the probability that V participates in a consistency anomaly for K. In their assumption, the operation history was infinite and the read and write operations arrived according to a Poisson process (Similar in Harmony [12]). In practical systems, describing the distribution of users' behaviors is difficult (e.g., different workloads in YCSB [32]). On the other hand, when the system is arranged, the network delay between two nodes is easier to be predicted in most cases. Our model removes the assumption about the distribution of the arriving operations. It analyzes how each write operation makes influences on the consistency state among those replicas of the same data based on the given escaped time and the network latency distribution.

Tunable Consistency Frameworks. Some related studies introduced mechanisms to implement systems as fast as possible, consistent when necessary [36]. RedBlue consistency [36] divided the operation into two classes. The blue operations executed locally and were lazily synchronized to other replicas, while the red operations were serialized and required immediate cross-site coordination. Rachid Guerraoui *et al.* [37] proposed an incremental consistency design in replicated systems. This approach, namely, incremental consistency guarantees (ICG), combined multiple consistency models (in fact, ICG only involved two consistency views) in a single operation. These systems reduce the operation latency by following the eventual consistency model in most cases and ensure strong consistency only when necessary. Based on machine learning, OptCon [38] can predict whether the specified latency and staleness thresholds can achieve one of the predetermined consistency levels. However, these tuning consistency models are coarse-grained and the users can only choose several consistency options instead of controlling the probability of reading the most updated value. Therefore, they always try to achieve a fully consistent state once the user asks for the most updated data. Our mechanism flexibly sets the read quorum size, which guarantees it can achieve the user-specified probability of returning the most updated value without waiting for a fully consistent system.

On the other hand, some research papers presented the fine-grained tuning of the consistency-latency trade-off. Continuous partial quorums (CPQ) [39], [40] allows each operation to choose multiple consistency levels with a tunable probability. This technique cannot tell the probability of reading the value updated by the most recent write and it is hard for users to quantify the accuracy of the responded results. HARP [25] compared results of observed timestamps and detected possible stale reads by try and error, even when failures occur. It only resolved stale reads under failures by issuing a reread operation, instead of rescaling the read quorum size in our model. Sudha Krishnamurthy *et al.* [41] proposed a flexible QoS model and provided different consistency guarantees for applications with their timeliness and consistency requirements. Their state-based replica selection algorithm returned that it needs more replicas once the system cannot achieve the minimum probability of reading updated value. Furthermore, it sorted all replicas in descending order of the elapsed response times and it preferred the replicas with lower latency. Load imbalance problem is easier to appear and our PCG model can avoid it (i.e., write/read quorums are randomly chosen).

Strict Quorum Systems. In the strict quorum systems, reading the updated value should be 100 percent guaranteed. Some state-of-the-art cloud storage systems, e.g., UniDrive [18], used a quorum-based distributed lock system to manage the metadata file. The distributed lock system usually requires strong consistency provided by the strict quorum systems. The incremental consistency design [37] could integrate the partial quorum system with the strict quorum system to optimize the performance. In this case, the PCG model could determine the minimum R (the shortest latency) to guarantee the desired consistency level.

6 Conclusion

Since the time interval for all replicas become consistent is usually unbounded [42], the consistency-latency trade-off in the quorum-based database is a fundamental topic that has been widely discussed [20], [43]. In this paper, we proposed a novel approach to quantify this trade-off and it provides a fine-grained tuning framework in the context of eventually

consistent systems. We predict the system-wide consistency degree with a given escaped time (t_x) after a write and suggests the number of replicas involved in a query to read a k-staleness value with a specified probability. Unlike the traditional analysis in Shankha's work, our theoretical results do not depend upon assuming the operations arrived according to a Poisson process. We also designed a rescale method, which is based on the update progress of the unavailable nodes, to adapt the read quorum size and maintain the consensus ratio of returning a reliable result. Some key findings are listed below:

- Our proposed PCG model can examine whether the system-wide consistency degree has achieved the user-specified consistency level or not, once given the latency after an update.
- Our formula can accurately calculate the consensus ratio of reading bounded staleness data. To reduce the computation complexity, we derive a closed-form solution to calculate the lower bound of the consensus ratio. It achieves up to 77.7 percent percent more accuracy and reduces up to 48.9 percent read latency than Bailis's formula.
- Based on this succinct formula, the system can also determine the exact read quorum size to guarantee the user-specified consensus ratio. Even node failure happens, it can scale up/down the read quorum size to maintain the probability of reading k-staleness data. This method provides a fine-grained trade-off between performance and consistency.

Acknowledgments

The authors would like to thank the editors and reviewers for their suggestions, which improved the quality of this article. This research was supported by Hong Kong RGC Collaborative Research Fund E-RB29.

References

[1] 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. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294281

[2] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Operating Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[3] D. Gašpar and M. Mabić, "NoSQL databases as social networks storage systems," in *Proc. ENTRENOVA Conf. Proc.*, 2017. [Online]. Available: https://ssrn.com/abstract=3282559

[4] N. Bronson *et al.*, "TAO: Facebook's distributed data store for the social graph," in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 49–60. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson

[5] Bertini36, "Sample application combinating tensorflow, spark and cassandra," Oct. 2016. [Online]. Available: https://github.com/bertini36/tensorsparkandra

[6] M. Azure, "Scale partitions and replicas for query and indexing workloads in azure search," 2019. [Online]. Available: https://docs.microsoft.com/en-us/azure/search/search-capacity-planning

[7] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proc. 19th Annu. ACM Symp. Princ. Distrib. Comput.*, 2000. [Online]. Available: http://doi.acm.org/10.1145/343477.343502

[8] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002. [Online]. Available: http://doi.acm.org/10.1145/564585.564601

[9] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer*, vol. 45, no. 2, pp. 37–42, Feb. 2012.

[10] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica, "Probabilistically bounded staleness for practical partial quorums," *Proc. VLDB Endowment*, vol. 5, no. 8, pp. 776–787, Apr. 2012.

[11] M. R. Rahman, L. Tseng, S. Nguyen, I. Gupta, and N. Vaidya, "Characterizing and adapting the consistency-latency tradeoff in distributed key-value stores," *ACM Trans. Auton. Adaptive Syst.*, vol. 11, no. 4, pp. 20:1–20:36, Jan. 2017. [Online]. Available: http://doi.acm.org/10.1145/2997654

[12] H. E. Chihoub, S. Ibrahim, G. Antoniu, and M. S. Pérez, "Harmony: Towards automated self-adaptive consistency in cloud storage," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2012, pp. 293–301.

[13] T. E. Lea, L. Jehl, and H. Meling, "Towards new abstractions for implementing quorum-based systems," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 2380–2385.

[14] R. Latip, M. Othman, A. Abdullah, H. Ibrahim, and M. N. Sulaiman, "Quorum-based data replication in grid environment," *Int. J. Comput. Intell. Syst.*, vol. 2, no. 4, pp. 386–397, 2009. [Online]. Available: https://doi.org/10.1080/18756891.2009.9727670

[15] Á. García-Pérez and A. Gotsman, "Federated byzantine quorum systems (extended version)," 2018. [Online]. Available: https://arxiv.org/ftp/arxiv/papers/1811/1811.03642.pdf

[16] M. G. Merideth and M. K. Reiter, "Selected results from the latest decade of quorum systems research," in *Replication*, B. Charron-Bost, F. Pedone, and A. Schiper, Eds. Berlin, Germany: Springer-Verlag, 2010, pp. 185–206. [Online]. Available: http://dl.acm.org/citation.cfm?id=2172338.2172348

[17] R. Jiménez-Peris, M. Patiño Martínez, G. Alonso, and B. Kemme, "Are quorums an alternative for data replication?" *ACM Trans. Database Syst.*, vol. 28, no. 3, pp. 257–294, Sep. 2003. [Online]. Available: http://doi.acm.org/10.1145/937598.937601

[18] H. Tang, F. Liu, G. Shen, Y. Jin, and C. Guo, "Unidrive: Synergize multiple consumer cloud storage services," in *Proc. 16th Annu. Middleware Conf.*, 2015, pp. 137–148. [Online]. Available: http://doi.acm.org.eproxy.lib.hku.hk/10.1145/2814576.2814729

[19] D. Malkhi, M. K. Reiter, A. Wool, and R. N. Wright, "Probabilistic quorum systems," *Inf. Comput.*, vol. 170, no. 2, pp. 184–206, Nov. 2001.

[20] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica, "Quantifying eventual consistency with PBS," *Commun. ACM*, vol. 57, no. 8, pp. 93–102, Aug. 2014. [Online]. Available: http://doi.acm.org/10.1145/2632792

[21] I. Abraham and D. Malkhi, "Probabilistic quorums for dynamic systems," in *Proc. Int. Symp. Distrib. Comput.*, 2003, pp. 60–74.

[22] S. Burckhardt, *Principles of Eventual Consistency*, vol. 1. Boston, MA, USA: Now, Oct. 2014. [Online]. Available: https://www.microsoft.com/en-us/research/publication/principles-of-eventual-consistency/

[23] A. Aiyer, L. Alvisi, and R. A. Bazzi, "On the availability of non-strict quorum systems," in *Proc. 19th Int. Conf. Distrib. Comput.*, 2005, pp. 48–62. [Online]. Available: http://dx.doi.org/10.1007/11561927_6

[24] A. Davidson, A. Rubinstein, A. Todi, P. Bailis, and S. Venkataraman, "Adaptive hybrid quorums in practical settings," 2012. [Online]. Available: https://pdfs.semanticscholar.org/2662/fac49d3963e121c0290fa6f4bda78c0a74da.pdf

[25] Y. Tang, H. Sun, X. Wang, and X. Liu, "An efficient and highly available framework of data recency enhancement for eventually consistent data stores," *Front. Comput. Sci.*, vol. 11, no. 1, pp. 88–104, Feb. 2017. [Online]. Available: https://doi.org/10.1007/s11704-016-6041-1

[26] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009. [Online]. Available: http://doi.acm.org/10.1145/1435417.1435432

[27] R. F. Muirhead, "Some methods applicable to identities and inequalities of symmetric algebraic functions of *n* letters," *Proc. Edinburgh Math. Soc.*, vol. 21, pp. 144–162, 1902.

[28] N. developer team, "NTP: The network time protocol," Jan. 2014. [Online]. Available: http://www.ntp.org/

[29] A. Ailijiang, A. Charapko, and M. Demirbas, "Dissecting the performance of strongly-consistent replication protocols," in *Proc. Int. Conf. Manage. Data*, 2019, pp. 1696–1710. [Online]. Available: http://doi.acm.org/10.1145/3299869.3319893

[30] K. Park and W. Willinger, *Self-Similar Network Traffic and Performance Evaluation*, 1st ed. New York, NY, USA: Wiley, 2000.
[31] pbailis, "Apache cassandra with PBS," 2012. [Online]. Available: https://github.com/pbailis/cassandra-pbs
[32] brianfrankcooper, "Yahoo cloud serving benchmark," Oct. 2012. [Online]. Available: https://github.com/brianfrankcooper/YCSB
[33] D. K. Gifford, "Weighted voting for replicated data," in *Proc. 7th ACM Symp. Operating Syst. Princ.*, 1979, pp. 150–162. [Online]. Available: http://doi.acm.org/10.1145/800215.806583
[34] X. Wang, H. Sun, T. Deng, and J. Huai, "On the tradeoff of availability and consistency for quorum systems in data center networks," *Comput. Netw.*, vol. 76, no. C, pp. 191–206, Jan. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2014.11.006
[35] S. Chatterjee and W. Golab, "Brief announcement: A probabilistic performance model and tuning framework for eventually consistent distributed storage systems," in *Proc. ACM Symp. Princ. Distrib. Comput.*, 2017, pp. 259–261. [Online]. Available: http://doi.acm.org/10.1145/3087801.3087850
[36] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *Proc. 10th USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 265–278.
[37] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi, "Incremental consistency guarantees for replicated objects," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 169–184.
[38] S. Sidhanta, W. Golab, S. Mukhopadhyay, and S. Basu, "OptCon: An adaptable SLA-aware consistency tuning framework for quorum-based stores," in *Proc. 16th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2016, pp. 388–397.
[39] M. McKenzie, H. Fan, and W. Golab, "Continuous partial quorums for consistency-latency tuning in distributed NoSQL storage systems," 2015. [Online]. Available: https://arxiv.org/abs/1507.03162
[40] M. McKenzie, H. Fan, and W. Golab, "Fine-tuning the consistency-latency trade-off in quorum-replicated distributed storage systems," in *Proc. IEEE Int. Conf. Big Data*, 2015, pp. 1708–1717. [Online]. Available: https://doi.org/10.1109/BigData.2015.7363942
[41] S. Krishnamurthy, W. H. Sanders, and M. Cukier, "An adaptive framework for tunable consistency and timeliness using replication," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2002, pp. 17–26.
[42] A. Bouajjani, C. Enea, and J. Hamza, "Verifying eventual consistency of optimistic replication systems," *SIGPLAN Notices*, vol. 49, no. 1, pp. 285–296, Jan. 2014. [Online]. Available: http://doi.acm.org/10.1145/2578855.2535877
[43] T. Jungnickel, *On the Feasibility of Mult-Leader Replicaton in the Early Tiers*. Berlin, Germany: Technische Universitt Berlin, 2018. [Online]. Available: https://depositonce.tu-berlin.de/bitstream/11303/7947/3/jungnickel_tim.pdf

Xin Yao (Student Member, IEEE) received the BEng degree from the Department of Computer Science, Huazhong University of Science and Technology, Wuhan, China, in 2016. He is currently working toward the PhD degree in computer science at the University of Hong Kong, Hong Kong. His research interests include software systems for cloud computing, including large-scale deep learning systems, consistency/consensus models designing, and fault tolerance technologies.

Cho-Li Wang (Senior Member, IEEE) received the BS degree in computer science and information engineering from the National Taiwan University, Taipei, Taiwan, in 1985, and the PhD degree in computer engineering from the University of Southern California, Los Angeles, California, in 1995. He is currently a professor with the Department of Computer Science, University of Hong Kong. His research is broadly in the areas of parallel architecture, software systems for cluster computing, and virtualization techniques for cloud computing. His recent research projects involve the development of parallel software systems for multicore/GPU computing and multi-kernel operating systems for future manycore processor. He has published more than 150 papers in various peer reviewed journals and conference proceedings. He is/was on the editorial boards of several scholarly journals, including the *IEEE Transactions on Cloud Computing*, *IEEE Transactions on Computers*, and *Journal of Information Science and Engineering*. He also serves as a coordinator (China) of the IEEE Technical Committee on Parallel Processing.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Automatic Generation of High-Performance FFT Kernels on Arm and X86 CPUs

Zhihao Li, Haipeng Jia, Yunquan Zhang, *Member, IEEE*, Tun Chen, Liang Yuan, and Richard Vuduc, *Member, IEEE*

Abstract—This article presents AutoFFT, a template-based code generation framework that can automatically generate high-performance FFT kernels for all natural-number radices. AutoFFT is based on the Cooley-Tukey FFT algorithm, which exploits the symmetric and periodic properties of the DFT matrix, as the outer parallelization framework. Because butterflies are the core operations of the Cooley-Tukey algorithm, we explore additional symmetric and periodic properties of the DFT matrix and formulate multiple optimized calculation templates to further reduce the number of floating-point operations for butterflies of arbitrary natural numbers. To fully exploit hardware resources, we encapsulate a series of optimizations in an assembly template optimizer. Given any DFT problem, AutoFFT automatically generates C FFT kernels using these calculation templates and converts them into efficient assembly kernels using the template optimizer. Through a series of experiments on Arm, Intel, and AMD processors, we show that AutoFFT-generated kernels can outperform those in Fastest Fourier Transform in the West (FFTW), the Arm Performance Libraries (ARMPL), and the Intel Math Kernel Library (MKL).

Index Terms—AutoFFT, FFT, code generation, template, DFT

1 Introduction

The discrete Fourier transform (DFT) is a basic discrete transform used to perform Fourier analysis. The definition of the DFT is presented in Eq. (1):

$$Y_k = \sum_{i=0}^{n-1} x_i W_n^{ik} = \sum_{i=0}^{n-1} x_i \cdot e^{-\frac{2\pi j}{n}ik}, \tag{1}$$

where x is the input sequence of n complex numbers, Y is the corresponding output sequence, $k \in [0, n-1]$, $j = \sqrt{-1}$, and $W_n = e^{-\frac{2\pi j}{n}}$, which is called the twiddle factor (twiddle for short). The DFT matrix $(W_n^{ik})_{n\times n}$ is an expression of the DFT as a transformation matrix, as presented in Eq. (2). The DFT can be represented by applying the DFT matrix to the input vector through matrix multiplication. From this point of view, the DFT is essentially a matrix-vector operation, and the computational complexity of the naïve matrix-vector implementation is $O(n^2)$

$$(W_n^{ik})_{n\times n} = \begin{pmatrix} W_n^0 & W_n^0 & \dots & W_n^0 \\ W_n^0 & W_n^1 & \dots & W_n^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ W_n^0 & W_n^{n-1} & \dots & W_n^{(n-1)(n-1)} \end{pmatrix}. \tag{2}$$

Many previous studies [1], [2], [3], [4], [5], [6] reduce the computational complexity of the DFT from $O(n^2)$ to $O(n \log n)$ by exploiting the symmetric and periodic properties of the DFT matrix. The Cooley-Tukey algorithm [1] is the most widely used fast Fourier transform (FFT) algorithm in many practical applications [7], [8], [9], [10], [11], [12], [13]. It adopts a divide-and-conquer approach to recursively break down a large DFT into smaller DFTs, achieving an $O(n \log n)$ complexity. For $N = r^v$ with integers N, r, and v, r is called the radix. We present the derivation of the radix-2 FFT in Eq. (3)

$$\begin{aligned} Y_k &= \sum_{i=0}^{n-1} x_i W_n^{ik} \\ &= \sum_{i=0}^{n/2-1} x_{2i} W_n^{2ik} + \sum_{i=0}^{n/2-1} x_{2i+1} W_n^{(2i+1)k} \\ &= \sum_{i=0}^{n/2-1} f_i \cdot W_{n/2}^{ik} + W_n^k \sum_{i=0}^{n/2-1} g_i \cdot W_{n/2}^{ik}. \end{aligned} \tag{3}$$

The radix-2 FFT divides a large DFT of size n into two smaller DFTs of size $n/2$ in each recursive stage until it obtains indivisible DFTs of size 2 (hence the name "radix-2 FFT"). A DFT of size 2 is also called a radix-2 butterfly. Because the butterflies are critical to the FFT computation, in this paper, we define the butterfly kernel to calculate one

- *Z. Li is with the SKL of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100864, China, the University of Chinese Academy of Sciences, Beijing 100049, China, and also with the Georgia Institute of Technology, Atlanta, GA 30332. E-mail: lizhihao@ict.ac.cn.*
- *H. Jia, Y. Zhang, T. Chen, and L. Yuan are with the SKL of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100864, China. E-mail: {jiahaipeng, zyq, chentun, yuanliang}@ict.ac.cn.*
- *R. Vuduc is with the School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA 30332. E-mail: richie@cc.gatech.edu.*

Manuscript received 15 Oct. 2019; revised 9 Feb. 2020; accepted 26 Feb. 2020. Date of publication 13 Mar. 2020; date of current version 7 May 2020. (Corresponding author: Haipeng Jia.) Recommended for acceptance by Y. Yang. Digital Object Identifier no. 10.1109/TPDS.2020.2977629

1045-9219 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

butterfly, which is the core computing unit of the FFT computation, and define the FFT kernel to calculate multiple butterflies (described in Section 5.2).

The performance of the FFT implementation is restricted by its computational complexity and is dependent on the effectiveness of finer optimizations. Therefore, this paper proposes a template-based code generation framework, named AutoFFT, that autogenerates optimized FFT kernels in assembly by capitalizing on the symmetric and periodic properties of the DFT matrix.

First, we summarize and extract two novel and highly optimized patterns (the pair and quad patterns) with a reduced number of floating-point operations for butterflies in the following steps: 1) For the radix-r butterfly, we investigate the symmetric properties in the horizontal and vertical directions for radices of all natural numbers, which are divided into three cases ($r = 2m + 1$, $r = 4m + 2$, and $r = 4m + 4$). By observing the calculation of each output Y_k, Y_k's twiddles are symmetric in the horizontal direction, and we reduce the multiplications for non-power-of-two and power-of-two radices by factors of 2 and 4, respectively. In the vertical direction, some outputs share a similar symmetric feature, so we can further reduce the arithmetic complexity for non-power-of-two and power-of-two radices by factors of 2 and 4, respectively. 2) We study a periodic property inside each Y_k for these three cases. By exploiting this property, the arithmetic complexity can be further reduced by a factor of $gcd(k, r)$, where gcd is the greatest common divisor.

Second, AutoFFT generates high-performance FFT kernels based on the pair and quad patterns. In the beginning, the computational template designer takes these two highly optimized patterns as inputs and formalizes them into computational templates. Then these computational templates are used by a C FFT kernel generator to generate C FFT kernels according to the derived equations of these two patterns. Subsequently, an assembly template optimizer, which is composed of the optimization templates and the register allocation strategy, is introduced to transform C kernels into high-performance assembly FFT kernels for varying architectures. Considering AutoFFT is built on templates, which are the abstractions of typical calculation patterns in the FFT computation, we refer to it as the template-based code generation framework, which is the first framework that can directly generate high-performance assembly FFT kernels by taking advantage of the optimized pair and quad patterns.

At runtime, to obtain the best FFT plan of a given DFT problem, AutoFFT adopts a pruning-based dynamic programming approach to empirically search for the optimal plan. The optimal plan contains the necessary parameters that are used to construct the corresponding butterfly network of the best plan and invokes required FFT kernels to perform the FFT computation. The experiments show that AutoFFT outperforms the current state-of-the-art, such as the Fastest Fourier Transform in the West (FFTW) [14], [15], [16], ARM Performance Library (ARMPL) [17], and Intel Math Kernel Library (MKL) [18], [19].

The key contributions and innovations of this paper are summarized as follows:

- This paper is the first work that systematically summarizes and extracts the integral and general mathematical expressions for the symmetric and periodic properties of the DFT matrix. These mathematical expressions minimize the number of floating-point operations for all natural-number radices: the pair pattern for odd numbers and the quad pattern for even numbers. Other libraries can also accelerate their butterflies based on these two optimized patterns.
- We propose a template-based code generation framework, which is the first framework that is able to automatically generate high-performance assembly FFT kernels based on the highly optimized pair and quad patterns for varying CPU architectures.
- Based on our code generation framework, we realize a high-performance FFT library for ARMv8, Intel Haswell, and AMD Zen CPUs. This library is on average 2.14, 2.15, and 1.7 times faster than FFTW, ARMPL, and Intel MKL, respectively.

This paper extends a conference version [20]. In particular, it adds: 1) compared to the conference version, which can generate FFT kernels of prime-number and power-of-two radices, this paper extends the generality of the framework and is able to generate kernels for all natural numbers; 2) this paper ports AutoFFT to AMD Zen CPUs and supplements the experimental results; and 3) this paper illustrates the implementations and optimizations adopted in AutoFFT's butterfly network.

The remainder of this paper is organized as follows. Section 2 presents related studies, and Section 3 provides an overview of the AutoFFT framework. Section 4 deduces the pair and quad patterns, and Section 5 illustrates how AutoFFT autogenerates high-performance assembly FFT kernels based on these two patterns. Section 6 gives the implementations and optimizations of our butterfly network, and Section 7 presents the experimental results. Finally, Section 8 concludes the paper.

2 Related Work

Many FFT algorithms [1], [2], [3], [4], [5], [6], [21], [22] have been proposed to compute the DFT. As the most popular FFT algorithm, the Cooley-Tukey algorithm [1] is supported by most mainstream FFT libraries. These libraries are optimized by vendors or researchers to achieve high performance on specific hardware architectures.

These highly efficient vendor-supplied FFT libraries, which include ARMPL [17], Intel MKL [18], [19], AMD Optimizing CPU Libraries (AOCL) [23], IBM Engineering and Scientific Subroutine Library (ESSL) [24], and Apple vDSP [25], are appropriately optimized using single instruction multiple data (SIMD) techniques for specific architectures. These implementations have undergone extensive architecture-dependent tuning on specific microarchitecture features to pursue peak system performance.

Because these vendor-specific libraries cannot be ported to processors of other vendors, many excellent auto-tuning systems have been developed. FFTW [14], [15], [16] and UHFFT [26], [27] conduct performance tuning in two stages. In the install-time stage, the special-purpose compiler generates several highly optimized instruction set architecture (ISA)-specific SIMD codelets to boost performance. In the runtime stage, its adaptive framework creates and empirically

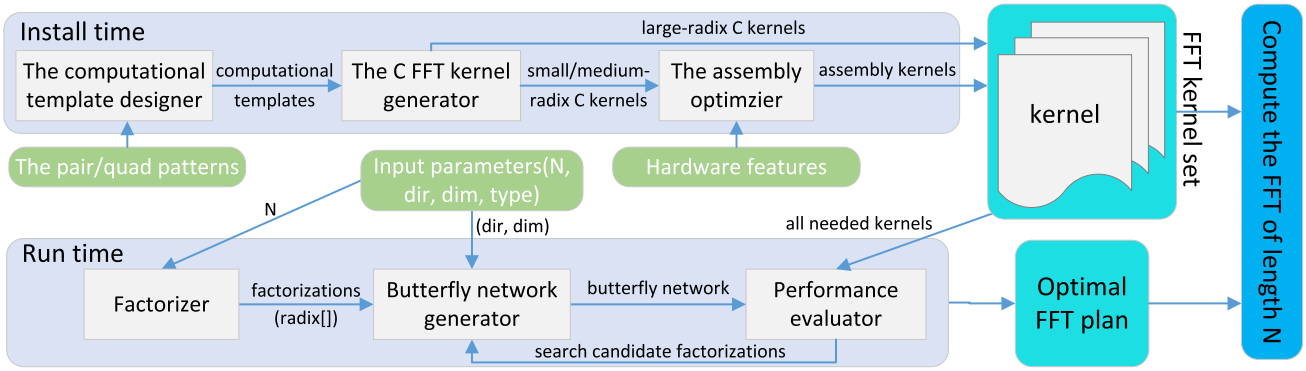


Fig. 1. Overview of AutoFFT.

chooses the optimal plan according to the input parameters and hardware features. The optimal plan organizes and assembles required pregenerated codelets to perform FFTs that involve large DFT problems. SPIRAL [28], [29] is a three-stage framework. In the first stage, SPIRAL expresses the mathematical formulas in the Signal Processing Language (SPL) [22] for a given transform. The second stage converts the SPL formulas into Σ-SPL [30], and then performs various computational optimizations such as code reordering in Σ-SPL. Finally, the optimized Σ-SPL is translated into high-level code (SIMD intrinsics) for the given architecture [29]. Other open-source FFT implementations include Ne10 [31], which contains FFT kernels that have been heavily optimized for ARM-based CPUs equipped with NEON SIMD capabilities, the Fastest Fourier Transform in the East (FFTE) [32], which is a Fortran FFT package that supports problem sizes of the form $2^a3^b5^c$ for CPUs and GPUs, and the Fastest Fourier Transform in the South (FFTS) [33], which is a hand-optimized library for Intel and ARM CPUs.

Significant progress has been made over the past decade in the fields of code generation and automatic performance tuning on GPUs. In addition to high-performance vendor-tuned FFT libraries, such as cuFFT for NVIDIA GPUs [34], clFFT for AMD GPUs [35], and genFFT for Intel processor graphics [36], researchers have developed several FFT implementations [37], [38], [39], [40], [41]. Although the factors that affect the FFT performance on GPUs can be quite different than those on CPUs, most GPU frameworks are similar to FFTW: 1) autogenerating or manually writing optimized FFT kernels that exploit the underlying hardware features, such as GPU memory hierarchy and warp/wavefront scheduling, and 2) searching for the optimal plan to exploit available computational resources for a given transform.

According to the definition of the twiddle factor, the symmetric and periodic properties of the DFT matrix are inherited from trigonometric functions, such as the sine and cosine functions. Many FFT libraries, such as FFTW, Intel MKL, ARMPL, SPIRAL, Ne10, and so on, utilize these properties more or less to optimize their FFT kernels, especially for power-of-two radices. However, according to the public materials of these libraries, none of them has provided integral and general mathematical expressions to combine the symmetry and periodicity for all radices of natural numbers. Besides, different from other FFT libraries, AutoFFT is built on the templates, which are the abstractions of typical calculation patterns in the FFT computation. These templates are defined based on the optimized pair and quad patterns and used to autogenerate low-level assembly FFT kernels. FFTW's codelets are some fundamental kernels used to calculate small DFTs. So FFTW's codelets are conceptually equivalent to AutoFFT's FFT kernels, which means that codelets and templates belong to different granularities (templates are in a finer granularity than codelets). Compared with the prior work, this paper contains the following three contributions. First, this paper is the first work that systematically summarizes and extracts the integral mathematical expressions of the symmetric and periodic properties (the pair and quad patterns) to minimize the arithmetic complexity for all radices. Second, we formalize these two patterns into templates so that these two patterns can be applied in the code generation. Third, we build a novel framework AutoFFT, which is based on the template methodology, to enable the assembly code generation. Benefiting from the template-based code generation mechanism, AutoFFT is able to directly generate efficient assembly FFT kernels for varying architectures without compiler intervention.

This paper compares AutoFFT with FFTW, ARMPL, and Intel MKL on ARM, Intel, and AMD CPU platforms. ARMPL and Intel MKL are the official math libraries for ARM and Intel processors, respectively. These two vendor-tuned libraries are well-optimized to achieve high performance. The performance of FFTW is typically superior to that of other publicly available FFT libraries and is even competitive with vendor-tuned libraries. In contrast to vendor-tuned libraries, FFTW's performance is portable, which means that FFTW performs well on most architectures. According to AMD official description in [23], its official math library AOCL regards FFTW as its FFT module.

3 The AutoFFT Framework

This section introduces the two stages of AutoFFT, as shown in Fig. 1. The install-time stage generates efficient FFT kernels, and the runtime stage empirically searches for the optimal plan to construct an efficient butterfly network for a given FFT size and the underlying hardware features.

3.1 The Install-Time Stage

The install-time stage is responsible for generating high-performance FFT kernels that are defined to calculate multiple butterflies. To reduce the number of floating-point operations of different butterflies as many as possible, the pair

and quad patterns are derived from the symmetric and periodic properties of the DFT matrix $(W_r^{ik})_{r\times r}$. These two patterns are the foundation of the code generation framework. The details of how to deduce these two patterns will be given in Section 4. The install-time stage adopts three components to capitalize on these two patterns.

First, we describe the computational template designer. To make full use of the pair and quad patterns, a computational template designer is introduced. This designer takes these two highly optimized patterns as inputs and then obtains high-level computational templates as outputs. The computational templates are composed of meta templates and hybrid templates. Meta templates are predefined C preprocessor macros that represent some basic complex number operations. Unlike meta templates, hybrid templates are strongly related to radix r. For different radix r, the number and type of meta templates used by hybrid templates are different. Hence, hybrid templates are defined and extracted as C functions according to the derived equations of the pair and quad patterns.

Second, we describe the C FFT kernel generator. This generator takes the computational templates as inputs and then enables the automatic generation of high-level FFT kernels with the lowest arithmetic complexity for all radices according to Algorithm 1, which is the concrete implementation of the pair and quad patterns.

Algorithm 1. butterfly_kernel(out, in, tw, r, i, isFirst)

Input: in[]: the input data of current stage; tw[]: twiddles; r: radix; i: the ith butterfly; isFirst: whether it is the first stage.
Output: out[]: output data.

```
1: Load inputs into tmpIn[] from in[] according to i
2: if isFirst=0 then
3:   Load twiddles into tmpTW[] from tw[]
4:   for i←1 to r do
5:     CPX_MUL(tmpIn[i], tmpIn[i], tmpTW[i-1])
6:   end for
7: end if
8: o←(r/2) mod 2
9: CALC_LIKE_TERMS(add, sub, tmpIn, r)
10: CALC_OUT_SPECIAL(tmpOut, add, sub, r)
11: if r is an odd number then
12:   for i←1 to r/2 do
13:     CALC_OUT_PAIR(tmpOut, add, sub, i, r)
14:   end for
15: else if r is an even number then
16:   for i←1 to r/4 do
17:     CALC_OUT_QUAD(tmpOut, tmpIn, i, r, o)
18:   end for
19: end if
20: Store outputs from tmpOut[] to out[]
```

Third, we describe the assembly optimizer. To further exploit hardware resources, the optimizer defines optimization templates that match the underlying hardware capabilities, such as SIMD vectorization, register resources, and the instruction set. In addition, it also categorizes small/medium/large radices (described in Section 5.3.3). After that, the optimizer takes the C FFT kernels and optimization templates as inputs and transforms the C FFT kernels into high-performance assembly FFT kernels according to the register allocation strategy.

Due to the limited register resources in modern CPUs, it is difficult to transform the C FFT kernels of large radices into efficient assembly code. Hence, AutoFFT can directly generate assembly FFT kernels for small and medium radices without compiler intervention and generate C FFT kernels for large radices. These kernels make up our FFT kernel set. As the computational core of the FFT computation, these kernels have a crucial impact on the overall performance and will be invoked as needed at runtime.

3.2 The Runtime Stage

According to the Cooley-Tukey algorithm [1], a given transform size is recursively factorized into smaller sizes until they can no longer be decomposed. When the given size is large, the size decomposition tree can be large too. Because each branch of this tree represents a factorization for the given FFT size, we call this decomposition tree as the factorization tree, and each branch of the factorization tree is also known as an FFT plan. The runtime stage is responsible for reducing the search space of many potential FFT plans and seeking the optimal plan for a given DFT problem. The runtime stage consists of three components.

First, we describe the number factorizer. The factorizer takes the FFT size as input, and then recursively factorizes the given FFT size and builds the factorization tree. As a result, it outputs many potential FFT plans. Each FFT plan represents one factorization that holds the radix for each stage of the butterfly network in $radix[]$.

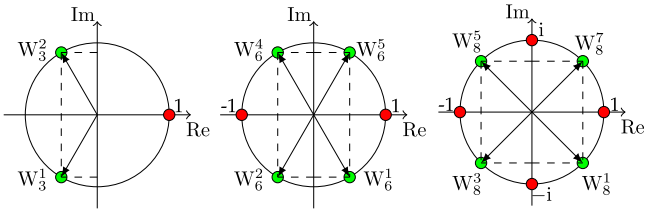
Second, we describe the butterfly network generator. AutoFFT adopts the Stockham auto-sort FFT [42], [43] network, which is SIMD-friendly and requires no explicit bit-reversal permutation. The generator takes the necessary input parameters ($radix[]$, the transform direction, and dimension) to construct the butterfly network.

Third, we describe the performance evaluator. Since the factorization tree can be large, the evaluator adopts a depth-first search to identify the shortest path of the tree and prune unneeded branches. Because the Cooley-Tukey algorithm is a memory-intensive algorithm, a branch with a shorter path indicates that its network contains fewer stages, which may lead to more efficient memory accesses. Subsequently, a bottom-up dynamic programming method is adopted on the pruned tree by making full use of the recursive structure of the FFT. We build a performance table to record the minimum execution time of the subsequences with various input and output strides, so we do not need to reevaluate the same subsequences with the same strides. Finally, the evaluator evaluates all candidate factorizations to determine the best factorization.

After obtaining the optimal plan, we construct the Stockham butterfly network according to the best factorization and call the needed FFT kernels to efficiently solve the given DFT problem. The auto-tuning framework of AutoFFT refers to the current state-of-the-art [16], [27], [39], and this paper does not discuss the runtime stage in detail but rather focuses on the FFT kernel generation.

4 Optimized Calculation Patterns

This section focuses on the algorithmic optimizations on butterflies. Performing algorithmic optimizations on butterflies



(a) Radix 3 (b) Radix 6 (c) Radix 8

Fig. 2. These three subfigures present the symmetries of twiddles of (a) radix 3, (b) radix 6, and (c) radix 8, respectively. The red points should be processed separately and the green points contain good symmetry.

and extracting general calculation patterns can bring two great benefits. First, because butterflies are the core operations of the Cooley-Tukey algorithm, reducing the floating-point operations of the butterfly kernels can significantly enhance the overall performance. Second, these calculation patterns regularize the calculation of the butterfly kernels, facilitating and enabling the code generation process. In this section, we systematically summarize the integral and general mathematical expressions for the symmetric and periodic properties and extract relatively optimized calculation patterns for radices of all natural numbers.

In our previous work [20], we empirically divide radix r into two cases from the perspective of engineering requirements: 1) r is a prime number and 2) r is a power of two. However, this division manner of r limits the generality of the code generation framework, which means the framework does not support generating butterfly kernels for radices beyond these two cases, such as radices 6, 9, 10, and so on. However, a native radix may be faster than mixed radices. For example, radix 6 can be faster than the combination of radix 2 and radix 3 when processing FFTs of power-of-six sizes.

In this paper, we improve and supplement the generality of AutoFFT based on the previous work [20] so that it can generate butterfly kernels for all natural numbers. Here, the natural numbers are divided into three cases: $r = 2m + 1$, $r = 4m + 2$, and $r = 4m + 4$. To simplify the representation of the formulas in this section, we introduce the following symbols:

$$c_i = x_i + x_{r-i}, \text{ and } d_i = x_i - x_{r-i}$$
$$u_i = x_{r/2-i} + x_{r/2+i}, \text{ and } v_i = x_{r/2-i} - x_{r/2+i}.$$

4.1 The Symmetry Property

4.1.1 The $r = 2m + 1$ Radices

Let $r = 2m + 1$. In this case, twiddles are symmetric about the x-axis. Fig. 2a presents twiddles' characteristics of radix 3, which complies with $r = 2m + 1$.

According to Eq. (1), when calculating Y_0 for the radix-r butterfly, because $k = 0$, the following equality is always true: $W_r^{ik} = W_r^0 = 1$. Thus, Y_0 is calculated separately using Eq. (4)

$$Y_0 = x_0 + \textstyle\sum_{i=1}^{m} c_i. \tag{4}$$

For the remaining outputs Y_k with $k \in [1, r-1]$, we minimize the number of floating-point operations by taking advantage of the horizontal and vertical symmetries of the DFT matrix.

Regarding the horizontal symmetry, W_r^{ik} and $W_r^{(r-i)k}$ are symmetric about the x-axis, and $W_r^{(r-i)k} = W_r^{-ik}$ which means that $Re(W_r^{ik}) = Re(W_r^{-ik})$ and $Im(W_r^{ik}) = -Im(W_r^{-ik})$. We refer to this as the horizontal symmetry since it exploits the reuse along each row in Eq. (2). For each Y_k, we define $W_r^{ik} = a_{ik} + b_{ik} \cdot j$ and $W_r^{(r-i)k} = W_r^{-ik} = a_{ik} - b_{ik} \cdot j$. Based on this symmetry, the calculation of Y_k can be reduced by uniting the like terms $x_i W_r^{ik}$ and $x_{r-i} W_r^{(r-i)k}$, as presented in Eq. (5)

$$\begin{aligned} Y_k &= \sum_{i=0}^{r-1} x_i W_r^{ik} = x_0 + \sum_{i=1}^{r-1} x_i W_r^{ik} \\ &= x_0 + \sum_{i=1}^{m} c_i \cdot a_{ik} + \sum_{i=1}^{m} d_i \cdot b_{ik} \cdot j. \end{aligned} \tag{5}$$

Regarding the vertical symmetry, by observing Eqs. (5) and (6), we find that Y_k and Y_{r-k} are like terms. We refer to this as the vertical symmetry since it exploits the reuse between rows in Eq. (2). Similar to Eq. (5), Y_{r-k} can be calculated using Eq. (6). Accordingly, we further reduce the number of floating-point operations by uniting the like terms and calculating Y_k and Y_{r-k} together

$$\begin{aligned} Y_{r-k} &= \sum_{i=0}^{r-1} x_i W_r^{i(r-k)} \\ &= x_0 + \sum_{i=1}^{r-1} x_i W_r^{ir-ik} \\ &= x_0 + \sum_{i=1}^{r-1} x_i W_r^{-ik} \\ &= x_0 + \sum_{i=1}^{m} c_i \cdot a_{ik} - \sum_{i=1}^{m} d_i \cdot b_{ik} \cdot j. \end{aligned} \tag{6}$$

4.1.2 The $r = 4m + 2$ Radices

Let $r = 4m + 2$. In this case, twiddles are symmetric about the x-axis and the y-axis. Fig. 2b presents the symmetric characteristics of radix-6 twiddles, which complies with $r = 4m + 2$.

For Y_0 and $Y_{r/2}$, all required twiddles can be simplified to $W_r^0 = 1$ and $W_r^{r/2} = -1$; therefore, Y_0 and $Y_{r/2}$ are calculated separately using Eq. (7)

$$\begin{aligned} Y_0 &= \sum_{i=0}^{r-1} x_i W_r^{i \cdot 0} = \sum_{i=0}^{r/2-1} (x_i + x_{r-i-1}) \\ Y_{r/2} &= \sum_{i=0}^{r-1} x_i W_r^{i \cdot r/2} = \sum_{i=0}^{r/2-1} (x_{2i} - x_{2i+1}). \end{aligned} \tag{7}$$

Compared with the $r = 2m + 1$ case, the $r = 4m + 2$ case has better symmetry in both horizontal and vertical directions.

Regarding the horizontal symmetry, when $r = 4m + 2$, W_r^{ik}, $W_r^{(r-i)k}$, $W_r^{(r/2-i)k}$, and $W_r^{(r/2+i)k}$ are symmetric to some extent. Because $W_r^{(r/2-i)k} = (-1)^k W_r^{-ik}$ and $W_r^{(r/2+i)k} = (-1)^k W_r^{ik}$, we have $W_r^{ik} = a_{ik} + b_{ik} \cdot j$, $W_r^{(r-i)k} = a_{ik} - b_{ik} \cdot j$, $W_r^{(r/2-i)k} = (-1)^k (a_{ik} - b_{ik} \cdot j)$ and $W_r^{(r/2+i)k} = (-1)^k (a_{ik} + b_{ik} \cdot j)$. Y_k's calculation can be reduced by uniting these like terms $x_i W_r^{ik}$, $x_{r/2-i} W_r^{(r/2-i)k}$, $x_{r/2+i} W_r^{(r/2+i)k}$ and $x_{r-i} W_r^{(r-i)k}$, as presented in Eq. (8) where $k \in [1, m]$

$$\begin{aligned} Y_k &= x_0 + (-1)^k x_{r/2} \\ &\quad + \sum_{i=1}^{m}(c_i + (-1)^k u_i)a_{ik} \\ &\quad + \sum_{i=1}^{m}(d_i - (-1)^k v_i)b_{ik} \cdot j. \end{aligned} \tag{8}$$

Regarding the vertical symmetry, Y_{r-k}, $Y_{r/2-k}$, and $Y_{r/2+k}$ can be calculated using Eq. (9). Because the four outputs Y_k, Y_{r-k}, $Y_{r/2-k}$, and $Y_{r/2+k}$ contain like terms, we optimize the calculation by uniting their like terms and calculating these four results together

$$\begin{aligned} Y_{r-k} &= x_0 + (-1)^k x_{r/2} \\ &\quad + \sum_{i=1}^{m}(c_i + (-1)^k u_i)a_{ik} - \sum_{i=1}^{m}(d_i - (-1)^k v_i)b_{ik} \cdot j \\ Y_{r/2-k} &= x_0 + (-1)^{k+1} x_{r/2} + \sum_{i=1}^{m}(-1)^i(c_i - (-1)^k u_i)a_{ik} \\ &\quad - \sum_{i=1}^{m}(-1)^i(d_i + (-1)^k v_i)b_{ik} \cdot j \\ Y_{r/2+k} &= x_0 + (-1)^{k+1} x_{r/2} + \sum_{i=1}^{m}(-1)^i(c_i - (-1)^k u_i)a_{ik} \\ &\quad + \sum_{i=1}^{m}(-1)^i(d_i + (-1)^k v_i)b_{ik} \cdot j. \end{aligned} \tag{9}$$

4.1.3 The $r = 4m + 4$ Radices

Let $r = 4m + 4$. The calculation pattern for the $r = 4m + 4$ radices is very similar to the $r = 4m + 2$ case, where the twiddles are symmetric about the x-axis and the y-axis. Their main difference is that the former contains extra two twiddles that are always symmetric about the x or y axis, such as $W_r^{r/4\cdot k}$ and $W_r^{3r/4\cdot k}$ in Y_k's calculation. Hence, Y_0, $Y_{r/2}$, $Y_{r/4}$, and $Y_{3r/4}$ can be calculated separately. The calculations of Y_0 and $Y_{r/2}$ remain the same as the $r = 4m + 2$ case using Eq. (7). Regarding $Y_{r/4}$, and $Y_{3r/4}$, they are calculated using Eq. (10). Fig. 2c presents twiddles' characteristics of radix 8, which complies with $r = 4m + 4$

$$\begin{aligned} Y_{r/4} &= \sum_{i=0}^{r-1} x_i W_r^{i\cdot r/4} \\ &= \sum_{i=0}^{r/4-1}((x_{4i} - x_{4i+2}) - (x_{4i+1} - x_{4i+3}) \cdot j) \\ Y_{3r/4} &= \sum_{i=0}^{r-1} x_i W_r^{i\cdot 3r/4} \\ &= \sum_{i=0}^{r/4-1}((x_{4i} - x_{4i+2}) + (x_{4i+1} - x_{4i+3}) \cdot j). \end{aligned} \tag{10}$$

The horizontal and vertical symmetries in the $r = 4m + 4$ radices are also similar to those in the $r = 4m + 2$ case. At the same time, it should be noted that there are subtle differences between these two cases within the equations when calculating $Y_{r/2-k}$ and $Y_{r/2+k}$.

Regarding the horizontal symmetry, because $W_r^{(r/2-i)k} = (-1)^k W_r^{-ik}$ and $W_r^{(r/2+i)k} = (-1)^k W_r^{ik}$, we can unite like terms $x_i W_r^{ik}$, $x_{r/2-i} W_r^{(r/2-i)k}$, $x_{r/2+i} W_r^{(r/2+i)k}$ and $x_{r-i} W_r^{(r-i)k}$ for Y_k, as presented in Eq. (11) where $k \in [1, m]$

$$\begin{aligned} Y_k &= x_0 + (-1)^k x_{r/2} + x_{r/4} W_r^{r/4\cdot k} + x_{3r/4} W_r^{3r/4\cdot k} \\ &\quad + \sum_{i=1}^{m}(c_i + (-1)^k u_i)a_{ik} + \sum_{i=1}^{m}(d_i - (-1)^k v_i)b_{ik} \cdot j. \end{aligned} \tag{11}$$

Regarding the vertical symmetry, we use Eq. (12) to calculate Y_{r-k}, $Y_{r/2-k}$, and $Y_{r/2+k}$. Because the four outputs Y_k, Y_{r-k}, $Y_{r/2-k}$, and $Y_{r/2+k}$ are like terms, we optimize the calculations by uniting the like terms and calculating these four results together

$$\begin{aligned} Y_{r-k} &= x_0 + (-1)^k x_{r/2} \\ &\quad + \sum_{i=1}^{m}(c_i + (-1)^k u_i)a_{ik} - \sum_{i=1}^{m}(d_i - (-1)^k v_i)b_{ik} \cdot j \\ &\quad + x_{r/4} W_r^{r/4\cdot(r-k)} + x_{3r/4} W_r^{3r/4\cdot(r-k)} \\ Y_{r/2-k} &= x_0 + (-1)^{k+1} x_{r/2} + \sum_{i=1}^{m}(-1)^i(c_i + (-1)^k u_i)a_{ik} \\ &\quad - \sum_{i=1}^{m}(-1)^i(d_i - (-1)^k v_i)b_{ik} \cdot j \\ &\quad + x_{r/4} W_r^{r/4\cdot(r/2-k)} + x_{3r/4} W_r^{3r/4\cdot(r/2-k)} \\ Y_{r/2+k} &= x_0 + (-1)^{k+1} x_{r/2} + \sum_{i=1}^{m}(-1)^i(c_i + (-1)^k u_i)a_{ik} \\ &\quad + \sum_{i=1}^{m}(-1)^i(d_i - (-1)^k v_i)b_{ik} \cdot j \\ &\quad + x_{r/4} W_r^{r/4\cdot(r/2+k)} + x_{3r/4} W_r^{3r/4\cdot(r/2+k)}. \end{aligned} \tag{12}$$

4.2 The Periodic Property

The number of floating-point operations can be further reduced by capitalizing on the twiddle periodicity. Let $p = gcd(r, k)$ and $q = r/p$. When $p = 1$, Y_k's twiddles contain only one period. When $p \neq 1$, Y_k's twiddles periodically repeat at intervals of length q. We refer to this as the periodic property. Consequently, inputs with a distance of q multiply the same twiddle; therefore, we can obtain Eq. (13)

$$\begin{aligned} Y_k &= \sum_{i=0}^{r-1} x_i W_r^{ik} \\ &= \sum_{i=0}^{r-1} x_i W_r^{(ik \bmod r)} \\ &= \sum_{s=0}^{p-1}\sum_{t=0}^{q-1} x_{t+s\cdot q} W_r^{tk} \\ &= \sum_{t=0}^{q-1}\left(\sum_{s=0}^{p-1} x_{t+s\cdot q}\right) \cdot W_r^{tk}. \end{aligned} \tag{13}$$

In this way, we divide Y_k's r twiddles into p groups, and each group contains the same q twiddles. Benefiting from this periodic property, like terms occur when calculating Y_k; thus, we can reduce the number of floating-point operations by uniting these like terms, as presented in Eq. (13).

We define a new complex sequence $\hat{x}$ to represent the inner summation of Eq. (13). Because each Y_k contains its own $\hat{x}$, when k is given, p and q are settled. Hence, each element of $\hat{x}$ is defined in Eq. (14) with $t \in [0, q-1]$; thus, the Y_k in Eq. (13) can be re-expressed as the Y_k in Eq. (14)

CPX_ADD(out,in1,in2): out.r=in1.r+in2.r out.i=in1.i+in2.i	**CPX_SUB**(out,in1,in2): out.r=in1.r-in2.r out.i=in1.i-in2.i
CPX_MLA(out,in1,in2,s): out.r=in1.r+s*in2.r out.i=in1.i+s*in2.i	**CPX_MUL_S**(out,in,s): out.r=s*in.r out.i=s*in.i
CPX_MUL(out,in1,in2): rr=in1.r*in2.r ii=in1.i*in2.i ri=in1.r*in2.i ir=in1.i*in2.r out.r=rr-ii out.i=ri+ir	**CPX_OUT**(head,tail,A,B): head.r=A.r-B.i head.i=A.i+B.r tail.r=A.r+B.i tail.i=A.i-B.r

Fig. 3. Meta templates supported in AutoFFT.

$$\begin{aligned}\hat{x}_{(t,k)} &= \sum_{s=0}^{p-1} x_{t+s\cdot q}\\ Y_k &= \sum_{t=0}^{q-1} \hat{x}_{(t,k)} \cdot W_r^{tk}.\end{aligned} \tag{14}$$

Next, we extract the periodic properties of these three cases ($r = 2m + 1$, $r = 4m + 2$, and $r = 4m + 4$) to further reduce the number of floating-point operations. At a higher level of abstraction, the periodic properties can be seen as the outer framework of the symmetric properties.

4.2.1 The $r = 2m + 1$ Radices

In our previous work [20], because we have not considered the periodicity of the $r = 2m + 1$ case, it only support the generation of prime radices, such as 3 and 5. Here, we further generalize the $r = 2m + 1$ radices by formulating and combining the horizontal and vertical symmetries with the periodicity, so that we can generate radices of all odd numbers, such as radix 9. We split and parameterize the calculation of Eqs. (5) and (6) into parts that are denoted as A, C, and D, as shown in Eq. (15). Based on these parameters, the Y_k in Eq. (5) and the Y_{r-k} in Eq. (6) are re-expressed in Eq. (16). Since this calculation pattern can calculate two results at a time, we call it *the pair pattern*

$$\begin{aligned}A &= \hat{x}_{(0,k)}\\ C &= (\hat{x}_{(i,k)} + \hat{x}_{(q-i,k)})a_{ik}\\ D &= (\hat{x}_{(i,k)} - \hat{x}_{(q-i,k)})b_{ik} \cdot j\end{aligned} \tag{15}$$

$$\begin{aligned}Y_k &= A + \sum_{i=1}^{m} C + \sum_{i=1}^{m} D\\ Y_{r-k} &= A + \sum_{i=1}^{m} C - \sum_{i=1}^{m} D.\end{aligned} \tag{16}$$

4.2.2 The $r = 4m + 2$ and $r = 4m + 4$ Radices

Considering that there are few differences in symmetry and periodicity between the $r = 4m + 2$ case and the $r = 4m + 4$ case, we use a unified way to represent them. We define ($o = r/2 \mod 2$) to distinguish these two cases. When $r = 4m + 2$, $o = 1$; otherwise, $o = 0$. Unlike our previous work [20], which only supports the $r = 4m + 4$ case for power-of-two radices, this paper can generate radices of all even number including power-of-two radices.

To combine the horizontal and vertical symmetries with the periodicity for the $r = 4m + 2$ and $r = 4m + 4$ radices, we parameterize their calculation patterns of Y_k, Y_{r-k}, $Y_{r/2-k}$, and $Y_{r/2+k}$ using A, B, C, and D, as shown in Eq. (17). Based on these parameters, these outputs can be re-expressed as Eq. (18). Since this calculation pattern can calculate four results at a time, we call it *the quad pattern*. Note that when $r = 4m + 4$, $Y_{r/4}$ and $Y_{3r/4}$ are calculated separately here based on A and B. When $k = r/4$, $o = 0$ and $p = q = 4$; therefore, we obtain $Y_{r/4} = A + B$, and $Y_{3r/4} = A - B$

$$\begin{aligned}A &= \hat{x}_{(0,k)} - \hat{x}_{(q/2,k)}\\ B &= (1-o)((\hat{x}_{(q/4,k)} - \hat{x}_{(3q/4,k)}) \cdot Im(W_{4p}^{k}) \cdot j)\\ C &= ((\hat{x}_{(i,k)} + \hat{x}_{(q-i,k)})\\ &\quad + (-1)^{k+o}(\hat{x}_{(q/2-i,k)} + \hat{x}_{(q/2+i,k)}))a_{ik}\\ D &= ((\hat{x}_{(i,k)} - \hat{x}_{(q-i,k)})\\ &\quad - (-1)^{k+o}(\hat{x}_{(q/2-i,k)} - \hat{x}_{(q/2+i,k)}))b_{ik} \cdot j\end{aligned} \tag{17}$$

$$\begin{aligned}Y_k &= A + B + \sum_{i=1}^{q/4-1} C + \sum_{i=1}^{q/4-1} D\\ Y_{r-k} &= A - B + \sum_{i=1}^{q/4-1} C - \sum_{i=1}^{q/4-1} D\\ Y_{r/2-k} &= A - B + \sum_{i=1}^{q/4-1} (-1)^i \cdot C - \sum_{i=1}^{q/4-1} (-1)^i \cdot D\\ Y_{r/2+k} &= A + B + \sum_{i=1}^{q/4-1} (-1)^i \cdot C + \sum_{i=1}^{q/4-1} (-1)^i \cdot D.\end{aligned} \tag{18}$$

5 FFT Kernel Generation

5.1 The Computational Template Designer

The template-based code generation system is built on the pair and quad patterns. To make full use of these patterns and enable the code generation, we introduce a computational template designer. The template designer takes these two patterns as inputs and then obtains high-level computational templates as outputs. The computational templates are the basis of the code generation system. They consist of meta templates and hybrid templates. Meta templates are predefined hardcoded C preprocessor macros representing some basic arithmetic operations on complex numbers. Hybrid templates are defined as high-level C functions extracted by the designer according to the pair and quad patterns (Eqs. (16) and (18)). This is because the value of radix r determines the number and type of meta templates used in hybrid templates. Note that the input sequence x is equal to the original inputs multiplied by twiddles, and Y is the transformed result.

5.1.1 Meta Templates

As presented in Section 4, butterfly calculations consist of some arithmetic operations on complex numbers; therefore, we define these operation units as meta templates in Fig. 3.

```
CALC_LIKE_TERMS(add[],sub[],in[],r):
 m=(r-1)/2, t=r-1;
 if(r%2==0)
  CPX_ADD(add[0],in[0],in[r/2]);
  CPX_SUB(sub[0],in[0],in[r/2]);
 else
  add[0]=in[0];sub[0]=-in[0];
 endif
 for(h=1; i<=m; h++,t--)
  CPX_ADD(add[h],in[h],in[t]);
  CPX_SUB(sub[h],in[h],in[t]);
 end for
```

```
CALC_OUT_SPECIAL(out[],add[],sub[],r):
 m=r/2,out[0]=out[m]=0
 for(i=0;i<=m;i++)
  CPX_ADD(out[0],out[0],add[i]);
 end for
 if(r%2==0)
  for(i=0;i<m;i+=2)
   CPX_ADD(out[m],out[m],add[i]);
   CPX_SUB(out[m],out[m],add[i+1]);
  end for
 end if
```

```
CALC_OUT_PAIR(out[],in[],k,r):
 A=C=D=0; p=gcd(r,k); q=r/p;
 _in = period_in(in, p, q);
 A=pair_A(_in,0,k);
 CPX_ADD(C,C,A);
 for(i=1; i<q/2; ++i)
  _C=pair_C(_in,i,q-i,k)
  _D=pair_D(_in,i,q-i,k)
  CPX_ADD(C,C,_C);
  CPX_ADD(D,D,_D);
 end for
 CPX_OUT(out[k],out[r-k],C,D);
```

```
CALC_OUT_QUAD(out[],in[],k,r,o):
 A = 0;
 B = 0;
 C1 = C2 = 0;
 D1 = D2 = 0;
 p=gcd(r,k); q=r/p;
 _in = period_in(in, p, q);

 for(i=1; i<q/4; ++i)
  C=quad_C(_in,i,q-i,q/2-i,q/2+i,k,o)
  D=quad_D(_in,i,q-i,q/2-i,q/2+i,k,o)
  CPX_ADD(C1,C1,C);
  CPX_ADD(D1,D1,D);
  if(i%2==1)
   CPX_SUB(C2,C2,C);
   CPX_SUB(D2,D2,D);
  else
   CPX_ADD(C2,C2,C);
   CPX_ADD(D2,D2,D);
  end if
 end for

 A=quad_A(_in,0,q/2,k);
 B=quad_B(_in,q/4,3q/4,k,p,o);

 if((k==1) && (o==0))
  out[r/4]=A+B;
  out[3*r/4]=A-B;
 end if

 out[k]=quad_Yk(A,B,C1,D1);
 out[r-k]=quad_Yrsk(A,B,C1,D1);
 out[r/2-k]=quad_Yr2sk(A,B,C2,D2);
 out[r/2+k]=quad_Yr2pk(A,B,C2,D2);
```

Fig. 4. Hybrid templates supported in AutoFFT.

- ***CPX_ADD***(), ***CPX_SUB***(), *and* ***CPX_MUL***() represent complex addition, subtraction, and multiplication, respectively.
- ***CPX_MLA***$(out, in1, in2, s)$ represents the fused multiply-add (FMA) operation on complex numbers. It is used for the accumulation operations in Eqs. (16) and (18). Likewise, ***CPX_MUL_S***(out, in, s) is used for these two equations to complete the multiplication between a complex number in and a real number s.
- ***CPX_OUT***$(head, tail, A, B)$ is used to calculate a pair of results (such as Y_k and Y_{r-k}, and $Y_{r/4-k}$ and $Y_{3r/4-k}$), as presented in Eqs. (16) and (18).

5.1.2 Hybrid Templates

Hybrid templates are extracted and formalized by the computational template designer to implement the pair pattern (Eq. (16)) for $r = 2m + 1$ radices, and the quad pattern (Eq. (18)) for $r = 4m + 2$ and $r = 4m + 4$ radices, as shown in Fig. 4.

- ***CALC_LIKE_TERMS***(add, sub, in, r) unites like terms. When $r = 2m + 1$, according to Eq. (15), add stores $(x_0), (x_1 + x_{r-1}), \ldots, (x_m + x_{r-m})$, and sub stores $(-x_0)$, $(x_1 - x_{r-1}), \ldots, (x_m - x_{r-m})$. When r is an even number, according to Eq. (17), add stores $(x_0 + x_m)$, $(x_1 + x_{r-1}), \ldots, (x_m + x_{r-m})$, and sub stores $(x_0 - x_m)$, $(x_1 - x_{r-1}), \ldots, (x_m - x_{r-m})$; in contains $x_0 + \cdots + x_{r-1}$.
- ***CALC_OUT_SPECIAL***(out, add, sub, r) adopts Eq. (4) to separately calculate Y_0 when r is an odd number; when r is an even number, it separately calculates Y_0 and $Y_{r/2}$ using Eq. (7).
- ***CALC_OUT_PAIR***(out, add, sub, k, r) implements the pair pattern (Y_k and Y_{r-k}) according to Eq. (16). Because this process only involves basic complex number operations, we simplify the expressions for calculating Eq. (15)'s $\hat{x}/A/C/D$ for readability.
- ***CALC_OUT_QUAD***(out, add, sub, k, r) implements the quad pattern (Y_k, $Y_{r/2-k}$, $Y_{r/2+k}$, and Y_{r-k}) according to Eq. (18). In addition, $Y_{r/4}$ and $Y_{3r/4}$ are calculated separately. We simplify the expressions for calculating Eq. (17)'s $\hat{x}/A/B/C/D$ for readability.

Algorithm 2. FFT_kernel(out, in, tw, r, butterfly_num, isFirst)

Input: in[]: inputs; tw[]: twiddles; r: radix; butterfly_num: the number of butterflies; isFirst: whether it is the first stage.
Output: out[]: outputs.
1: **for** i←0 **to** butterfly_num **do**
2: butterfly_kernel(out, in, tw, r, i, isFirst)
3: **end for**

5.2 The C FFT Kernel Generator

After we obtain the computational templates, a C FFT kernel generator is designed to generate integral FFT kernels that can independently calculate DFTs. According to Algorithm 1, this generator takes a given radix r (an arbitrary natural number) as input and calls the needed computational templates, then an efficient C FFT kernel for the given radix r can be automatically generated.

The butterfly kernel is the core computing module for calculating one butterfly, so its implementation and optimization are critical to the overall performance. In Algorithm 1, lines 1∼7 load and multiply the inputs and twiddles and then store the results in the temporary complex array $tmpIn$. In the first stage, all twiddles equal 1; thus, lines 3∼6 are skipped. Line 9 calls *CALC_LIKE_TERMS*() to calculate the like terms that can be reused by other hybrid templates. When r is an odd number, line 10 calls *CALC_OUT_SPECIAL*() to calculate Y_0, and lines 12∼14 then perform the pair pattern. When r is an even number, line 10 calculates Y_0 and $Y_{r/2}$, and lines 16∼18 perform the quad pattern. Finally, line 20 stores the results in out.

The FFT kernel is adopted to process multiple butterflies in a for-loop, as shown in Algorithm 2. According to the Stockham FFT network, adjacent butterflies can be calculated together, so we can SIMDize the for-loop to improve the performance of the FFT kernel.

5.3 The Assembly Template Optimizer

The C FFT kernel generator is mostly concerned with reducing floating-point operations. However, the underlying hardware designs for factors such as registers and pipeline structure also have a high impact on the performance. To further exploit the hardware resources, we need to accelerate and transform the C FFT kernels into assembly kernels. To automate the transformation process for varying architectures, the assembly template optimizer is introduced. The optimizer takes optimization templates as inputs, and then automatically transforms the C FFT kernels into assembly according to the register allocation strategy, which is defined based on the underlying architectures. The optimization templates are hardcoded architecture-specific assembly templates that are transformed from the high-level meta templates.

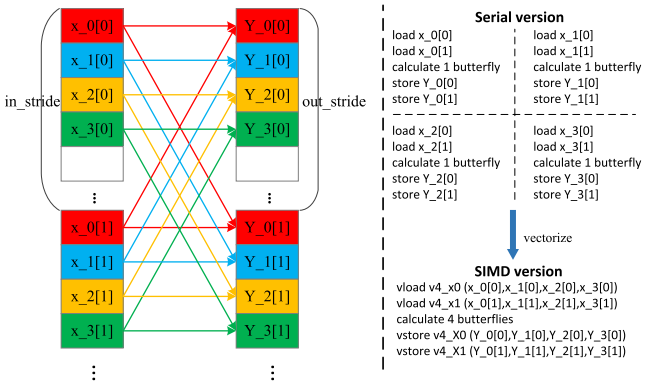


Fig. 5. SIMDize four adjacent radix-2 butterflies. Note that in the ith stage, $in_stride = N/2$ and $out_stride = 2^i$.

5.3.1 Butterfly Vectorization

Most modern CPUs provide SIMD techniques to boost performance; therefore, AutoFFT adopts the SIMD-friendly Stockham FFT network. In this network, the inputs and outputs of adjacent butterflies are contiguous in memory; therefore, Algorithm 2 can be SIMDized to calculate multiple butterflies simultaneously. Fig. 5 presents how to SIMDize multiple radix-2 butterflies for a sequence of length N. Four adjacent radix-2 butterflies are represented by four colors. One radix-2 butterfly requires two inputs (such as x_0[0] and x_0[1]) and yields two outputs (Y_0[0] and Y_0[1]).

5.3.2 Instruction Mapping

We define instruction mapping rules to translate the high-level computational templates into architecture-specific hardcoded optimization templates by selecting and scheduling efficient assembly instructions. AutoFFT currently focuses on the ARMv8 ISA and the x86-64 ISA. Their instruction mapping rules are listed in Fig. 6. When SIMDizing butterflies on the ARMv8 architecture, we use two 128-bit registers to separately hold four complex numbers' real and imaginary parts. On the Intel Haswell and AMD Zen architectures, we use one 256-bit register to hold four complex numbers by interleaving their real and imaginary parts. We adopt this approach for the following reasons: 1) AVX2 does not support efficient load/store instructions such as ld2/st2 instructions in ARM NEON; and 2) AVX2 provides the *vaddsubps* instruction to efficiently perform multiplication on complex numbers for the interleaved pattern.

To keep every execution unit of the processor busy with instructions, AutoFFT reorders the instruction streams using the following two methods. 1) Detach instructions with dependencies, especially for memory instructions and corresponding arithmetic instructions. Because the latencies of memory instructions are high, we insert independent instructions between the memory instructions and the corresponding dependent arithmetic instructions. 2) Rearrange independent instructions based on the functionalities of the issue ports. For example, the issue ports 0/1/5 of the Haswell architecture perform arithmetic operations, and independent instructions such as floating-point multiplication, FMA, and shuffle can be dispatched to these three ports in parallel [44].

Benefiting from the optimization templates, when new architectures emerge, we only need to implement the corresponding optimization templates. As shown in Fig. 6, the optimization templates use register aliases instead of physical vector registers. In the following, we define the register allocation strategy for different radices to transform the C FFT kernels into corresponding assembly FFT kernels.

Meta Templates	ARMv8 Optimization Templates	x86-64 Optimization Templates
CPX_ADD(out,in1,in2): out.r=in1.r+in2.r out.i=in1.i+in2.i	**CPX_ADD**(out_r,out_i,in1_r,in1_i,in2_r,in2_i): fadd out_r.4s, in1_r.4s, in2_r.4s fadd out_i.4s, in1_i.4s, in2_i.4s	**CPX_ADD**(out,in1,in2): vaddps %in1, %in2, %out
CPX_SUB(out,in1,in2): out.r=in1.r-in2.r out.i=in1.i-in2.i	CPX_SUB(out_r,out_i,in1_r,in1_i,in2_r,in2_i): fsub out_r.4s, in1_r.4s, in2_r.4s fsub out_i.4s, in1_i.4s, in2_i.4s	**CPX_SUB**(out,in1,in2): vsubps %in1, %in2, %out
CPX_MLA(out,in1,in2,s): out.r=in1.r+s*in2.r out.i=in1.i+s*in2.i	**CPX_MLA**(out_r,out_i,in1_r,in1_i,in2_r,in2_i,s): fmla out_r.4s, in2_r.4s, s.s[0] fmla out_i.4s, in2_i.4s, s.s[0]	**CPX_MLA**(out,in1,in2,s): vfmadd231ps %in2, %s, %out
CPX_MUL_S(out,in,s): out.r=s*in.r out.i=s*in.i	**CPX_MUL_S**(out_r,out_i,in_r,in_i,s): fmul out_r.4s, in_r.4s, s.s[0] fmul out_i.4s, in_i.4s, s.s[0]	**CPX_MUL_S**(out,in,s): vmulps %in, %s, %out
CPX_MUL(out,in1,in2): rr=in1.r*in2.r ii=in1.i*in2.i ri=in1.r*in2.i ir=in1.i*in2.r out.r=rr-ii out.i=ri+ir	**CPX_MUL**(out_r,out_i,in1_r,in1_i,in2_r,in2_i): fmul rr.4s, in1_r.4s, in2_r.4s fmul ii.4s, in1_i.4s, in2_i.4s fmul ri.4s, in1_r.4s, in2_i.4s fmul ir.4s, in1_i.4s, in2_r.4s fsub out_r.4s, rr.4s, ii.4s fadd out_i.4s, ri.4s, ir.4s	**CPX_MUL**(out,in1,in2): vmovsldup %in2, %tmp0 vmovshdup %in2, %tmp1 vmulps %in1, %tmp0, %tmp0 vpshufd $ 0xb1, %tmp1, %tmp1 vfmaddsub213ps %tmp1, %tmp0, %out
CPX_OUT(head,tail,A,B): head.r=A.r-B.i head.i=A.i+B.r tail.r=A.r+B.i tail.i=A.i-B.r	**CPX_OUT**(h_r,h_i,t_r,t_i,A_r,A_i,B_r,B_i): fsub h_r.4s, A_r.4s, B_i.4s fadd h_i.4s, A_i.4s, B_r.4s fadd t_r.4s, A_r.4s, B_i.4s fsub t_i.4s, A_i.4s, B_r.4s	**CPX_OUT**(head,tail,A,B): vshufps 0xb1, %B, %B, %tail vaddsubps %tail, %A, %head vfmsubadd231ps ONE(%rip), %A, %tail

Fig. 6. Instruction mapping rules between meta templates and optimization templates.

5.3.3 Register Allocation Strategy

Vector registers are scarce resources in modern CPUs. We design a strict vector register allocation strategy to make full use of register resources and enable the automatic generation of assembly FFT kernels. The register allocation strategy contains two steps. The first step is the the register alias usage step, which is used to reduce the difficulty of the generation of assembly FFT kernels. The second step is the vector register mapping step, which is used to complete the mappings between register aliases and vector registers.

Register Alias Usage. The number of vector registers in modern CPUs is limited. ARMv8 processors contain 32 128-bit vector registers, while Intel Haswell and AMD Zen processors contain 16 256-bit vector registers. Because AMD Zen and Intel Haswell have the same amount of registers, and their register alias usages are the same, we omit the repeated description of AMD Zen's register alias usage for simplicity. During the process of assembly FFT kernel generation, vector registers become increasingly scarce as the radix increases, so we introduce register aliases to reduce the difficulty of assembly code generation. By using register aliases, we do not need to worry about whether the number of physical vector registers meets the needs of assembly code generation. In this step, only register aliases are used in the generated assembly FFT kernels.

The main idea of register alias usage is to group register aliases according to their functionalities in Algorithm 1 and to strictly define the usage rules and the required amount of register aliases of each group. According to Algorithm 1, AutoFFT divides register aliases into four groups: the input group, the twiddle group, the temporary group, and the

output group. We now analyze the required register aliases of the four groups for the radix-r kernel.

On the ARMv8 architecture, the input group requires $2r$ registers for $tmpIn$, the twiddle group requires $2r-2$ registers for $tmpIW$, and the temporary group requires $2r$ registers for add/sub. The other required registers in the temporary group and the output group are as follows: 1) When r is an odd number, the temporary variables (rr/ii/ri/ir used in $CPX_MUL()$ and C/D used in $CALC_OUT_PAIR()$) occupy 8 registers. In addition, 4 registers are needed to hold a pair of outputs. Thus, the sum of the required registers is $6r+10$. 2) When r is an even number, $CALC_OUT_QUAD()$'s $A/B/C1/C2/D1/D2$ occupy 12 temporary registers, and 8 registers for four outputs are required. Thus, the sum of the required registers is $6r+18$. On the Haswell architecture, because we use one 256-bit register to process 4 complex numbers, when r is an odd number, the sum of the required registers is $3r+5$; when r is an even number, the sum of the required registers is $3r+9$.

Vector Register Mapping. So far, the assembly FFT kernels generated from the register alias usage step only use register aliases, however, to complete the code generation and obtain runnable assembly FFT kernels, we still need to replace register aliases with physical vector registers. Hence, the main idea of the vector register mapping step is to construct the mappings between register aliases and physical vector registers. Because AMD Zen and Intel Haswell have the same register resources, we omit the repeated description of AMD Zen's register mapping rule for simplicity. The register mapping rule varies for different radices and can be divided into the following three cases based on Algorithm 1:

(1) Small Radices. According to the analysis in the register alias usage step, 1) when r is an odd number, on the ARMv8 architecture, where the required vector registers are $6r+10 \leq 32$, we have $r \leq 11/3$. On the Haswell architecture, where $3r+5 \leq 16$, we have $r \leq 11/3$; therefore, radix 3 is a small radix. 2) When r is an even number, on the ARMv8 architecture, where $6r+18 \leq 32$, we have $r \leq 7/3$; on the Haswell architecture, where $3r+9 \leq 16$, we have $r \leq 7/3$. Therefore, radix 2 is a small radix. Hence, there are sufficient registers for each group to independently perform their tasks for radices 2 and 3.

(2) Medium Radices. As the radix grows, the four groups require more vector registers, and the register resources become insufficient to independently perform the tasks for the four groups. Thus, we reuse registers according to the following four rules: 1) Reuse $tmpIW$. the operations between twiddles and inputs are independent; therefore, it is unnecessary to load all twiddles at once, and we suggest loading 4 twiddles each time. After the complex number multiplications are completed, the registers can be reused for the next 4 twiddles. In addition, after completing lines 1∼7 of Algorithm 1, the registers used for $tmpIW$ can be freed. 2) Reuse $tmpIn$. For $CALC_LIKE_TERMS()$ in line 9, we introduce one temporary complex number to stagger the registers used for add/sub and $tmpIn$; then, add/sub can reuse $tmpIN$'s registers. 3) Reuse the temporary registers. Temporary registers can also be used for the rr/ii/ri/ir of $CPX_MUL()$. 4) Reuse $tmpOut$. In lines 13 and 17, after obtaining an output, we immediately store it in memory, which means the output group only needs to maintain one complex number.

As analyzed above, on the ARMv8 architecture, lines 1∼7 in Algorithm 1 require $2r+8$ registers for $tmpIn$ and $tmpIW$. Subsequently, we have the following: 1) When r is an odd number, lines 9∼20 require only $2r+6$ registers: the temporary group requires $2r$ registers for add/sub, 4 registers are used for the temporary variables, and $tmpOut$ requires 2 registers. By adopting the four reuse rules, $2r+8$ registers are sufficient because they can be reused in lines 9∼20; if $2r+8 \leq 32$, we have $r \leq 12$. Similarly, the Haswell architecture requires $r+4$ registers; if $r+4 \leq 16$, we have $r \leq 12$; therefore, 5, 7, 9, and 11 are medium radices. 2) When r is an even number, on the ARMv8 architecture, lines 9∼20 require $2r+12$ registers, which is larger than $2r+8$: add/sub require $2r$ registers, the temporary variables require 12 registers, and $tmpOut$ requires 2 registers. When $2r+14 \leq 32$, we have $r \leq 9$. Similarly, the Haswell architecture requires $r+7$ registers. When $r+7 \leq 16$, we have $r \leq 9$; therefore, 4, 6, and 8 are medium radices.

(3) Large Radices. When $r > 11$, vector register resources are insufficient: we have to use stack or memory instructions to temporarily hold relevant data, which degrades the performance. Large radices require more vector registers, which are limited in modern CPUs. AutoFFT provides C FFT kernels to perform the FFT computation for large radices. Specialized FFT algorithms, such as Rader's algorithm [2], can achieve better performance than the Cooley-Tukey algorithm for large radices.

To replace register aliases with specific physical vector registers for assembly kernels of small and medium radices, the template optimizer maintains a register resource pool and a lookup register usage table based on the register mapping rule to guarantee the consistency of vector register usage across the instruction streams. As analyzed above, because there are sufficient vector registers for small radices, the optimizer can directly take out the required registers in the resource pool and map specific vector registers to corresponding register aliases in the register usage table. For medium radices, because we have strictly defined the register mapping rule for them above, the optimizer determines the exact reuse time of different registers based on the four register reuse rules and fills out the register usage table. When a vector register is ready for reuse, it will be labeled the "usable" tag and returned to the register resource pool.

The register allocation strategy essentially is determined by the number and the width of SIMD registers, so it can be easily extended to the future ARM SVE and AVX-512 by replacing the number and the width of corresponding SIMD registers. Because the FFT kernels of radices 2/3/4/5/6/7/8/9/11 (small and medium radices) are well-optimized assembly code, AutoFFT is best at computing FFTs of sizes of the form $2^a3^b5^c7^d11^e$ (the exponents are arbitrary). FFTs of other sizes are slower because they need to call the C FFT kernels of large radices. Regardless, sizes of the form $2^a3^b5^c7^d11^e$ meet the needs of most applications.

6 Butterfly Network Optimizations

The butterfly network determines data access patterns and the order in which the needed FFT kernels are executed for

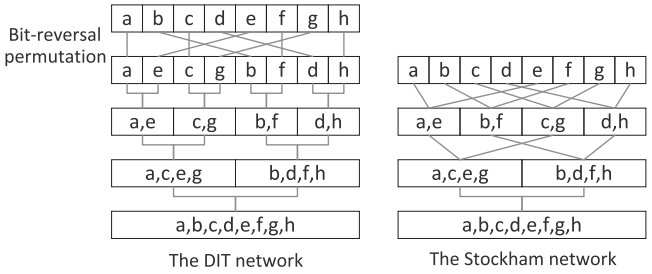


Fig. 7. Dataflows of the DIT network and the Stockham network when processing the FFT of size eight.

a given DFT problem. Even for the same butterfly network, different implementations and optimizations can lead to different performances. Fig. 7 presents the differences between the traditional decimation-in-time (DIT) [45] network and the Stockham network when processing the FFT of size eight. This section describes the specific implementations and optimizations adopted in AutoFFT's network.

6.1 Specific Implementations of the Stockham Network

The Stockham network is organized as a three-level network in AutoFFT. These three levels are embodied as three for-loops in the concrete implementation, as shown in Algorithms 2 and 3. Here, we define the outermost for-loop (Algorithm 3's line 3) as the stage loop, the middle for-loop (Algorithm 3's line 7) as the section loop, and the innermost loop as the butterfly loop (Algorithm 2's line 1). In this stage-section-butterfly network, the number of butterflies in each stage is always equal to the given FFT size N over the adopted radix r (N/r). In addition, one stage contains at least one section, which contains at least one butterfly. Fig. 8 shows the stage-section-butterfly network of an eight-point FFT based on radix 2. In this example, the first stage contains four sections, and each section contains one radix-2 butterfly; the second stage contains two sections, and each section contains two radix-2 butterflies; and the third stage contains one section, which contains four radix-2 butterflies.

Algorithm 3 presents the pseudocode of the three-level network. Compared with traditional Cooley-Tukey networks, such as the DIT network and decimation-in-frequency (DIF) [45] network, the three-level network contains three advantages: 1) No explicit bit-reversal permutation. In traditional butterfly networks, such as the DIT network shown in Fig. 7, the explicit bit-reversal permutation is required to reorder data from the natural order to the bit-reversed order. The bit-reversal permutation introduces extra and incoherent memory accesses, which can be expensive. Moreover, incoherent memory accesses can cause difficulties in unifying memory access patterns of inputs and outputs. Because both inputs and outputs of the three-level network are in the natural order, no explicit bit-reversal permutation is required. 2) SIMD-friendly. Considering that memory layouts of inputs and outputs of adjacent butterflies in each section are contiguous, we can easily SIMDize multiple adjacent butterflies in each section. 3) Mixed radices support. Since inputs and outputs are in the natural order, our network supports the lengths of arbitrary combinations of varying radices.

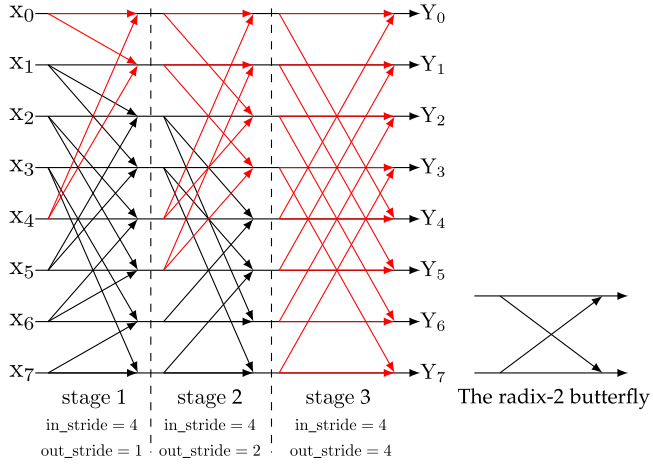


Fig. 8. The stage-section-butterfly network diagram for the FFT of size eight. Red arrows represent one section in each stage.

Algorithm 3. The Overview of the Butterfly Network

Input: in[]: inputs; radix[]: required radices in each stage; tw[]: twiddles; stage_num: stages in the network; section_num: sections in each stage.
Output: out[]: outputs.

```
1: r←radix[0], butterfly_num=1
2: FFT_kernel(out, in, NULL, r, section_num, 1)
3: for i←1 to stage_num do
4:   butterfly_num=butterfly_num*r
5:   r←radix[i] // radix in stage i
6:   section_num=section_num/r
7:   for j←0 to section_num do
8:     FFT_kernel(out, in, tw, r, butterfly_num, 0)
9:   end for
10: end for
```

6.2 Optimizations in the First Stage

As shown in Fig. 8 and Algorithm 3, the first stage of the network is processed and optimized independently to further exploit the parallelism: 1) Unlike the other stages, twiddles in the first stage are always equal to the constant 1 according to the definition of twiddles, so memory read operations of twiddles and complex number multiplications between inputs and twiddles can be completely eliminated. 2) Since each section of the first stage only contains one butterfly, instead of SIMDizing butterflies within one section in the section level similar to other stages, we need to directly SIMDize butterflies across multiple sections in the first stage. 3) To avoid explicit bit-reversal permutation and unify memory access patterns for other stages, we rearrange the memory layout of computational results of the first stage, so the output stride of memory store operations in the first stage is different from those in other stages. The output stride can be calculated using Eq. (19), where $radix[]$ holds the radix of each stage. Regarding the input stride in_stride, it is equal to $N/radix$ for all stages

$$out_stride = \begin{cases} 1, & \text{stage } 1 (i = 0) \\ \prod_{j=0}^{i-1} radix[j], & \text{stage } i+1 \ (i > 0) \end{cases}. \tag{19}$$

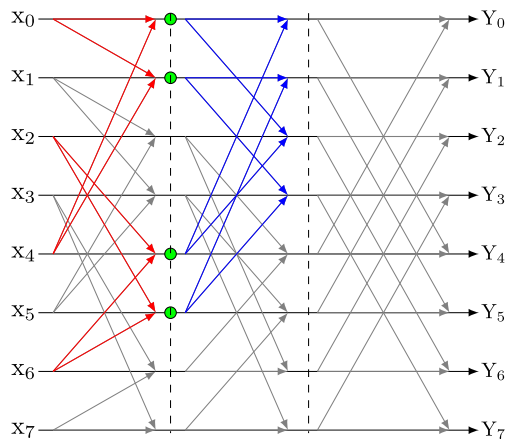


Fig. 9. The depth-first butterfly execution order in the first two stages. The order in which the butterflies of the first stage (red butterflies) is calculated depends on what inputs are required for the butterflies of the second stage (blue butterflies). For simplicity, we assume the SIMD width is 2 (SIMDize two blue butterflies) in this example.

6.3 Cache-Friendly Butterfly Execution Order

The CPU cache system has an important impact on the performance of the FFT. Usually, the execution order of butterflies is performed stage-by-stage (breadth-first order) in accordance with the butterfly network. For example, butterflies in the second stage need to wait for all butterflies in the first stage to complete before they can be executed. In this way, when the FFT size is large, it is very likely to cause L1 cache misses: the preceding butterfly calculation results in the current stage will be replaced out of L1 cache by the results of subsequent butterflies, resulting in cache misses when butterflies in the next stage read their inputs.

To make better use of the cache system and reduce L1 cache misses, we adopt a cache-friendly manner to determine the execution order of butterflies in the butterfly network: 1) Butterflies of the first two stages are executed in the depth-first order; and 2) butterflies of the remaining stages are executed in the breadth-first order. Fig. 9 presents the depth-first execution order of the first two stages. In this example, we assume the SIMD width is 2 for simplicity. To ensure that the inputs (green points) for the blue butterflies in the second stage are not replaced out of L1 cache when reading them, we will calculate the blue butterflies as soon as the calculations of the red butterflies are complete.

7 Performance Evaluation

This section evaluates the performance of AutoFFT on server-grade ARMv8 and x86-64 (Intel Haswell and AMD Zen) CPUs. AutoFFT supports complex/real and out-of-place/in-place FFT computations. Because FFTW, ARMPL, and Intel MKL are the most widely used and mature FFT libraries, we compare the performance of AutoFFT with them. For a one-dimensional (1D) FFT of length N with an execution time of t seconds, we report its performance in GFlops according to Eq. (20) [46], which is adopted in the well-known benchmark benchFFT [47]. Note that the x-axis of figures in this section represents the transform size N

$$GFlops = \frac{5N \cdot \log_2 N \cdot 10^{-9}}{t}. \tag{20}$$

TABLE 1
Experimental Environment

CPU	FT-2000+	Xeon E7-4850 v3	Hygon C86 7185
Arch.	AArch64	Haswell	Zen
Frequency	2.2 GHz	2.2 GHz	2.0 GHz
SIMD	128	256	256
L1 cache	32 KB	32 KB	32 KB
GCC	4.9.3	5.5.0	4.9.4
FFTW	3.3.8	3.3.8	3.3.8
ARMPL	19.2.0	-	-
Intel MKL	-	2019 Update 4	-

The experimental conditions are listed in Table 1. Our experiments take the C version of AutoFFT as the baseline to determine the performance boost achieved by AutoFFT's assembly kernels. In our experiments, we use the FFTW_MEASURE flag for all FFT plans. Considering Intel MKL's FFTW interfaces and its API share the same source code [48], we use the FFTW interfaces for Intel MKL in our experiments. In addition, because AMD takes FFTW as their official FFT library of AOCL [23], our experiments use the pre-built FFTW-3.3.8 library, which is compiled and provided by AOCL. For simplicity, we define a new naming scheme (a two-part string) that reflects the name of the FFT library (AutoFFT/FFTW/ARMPL/MKL) and out-of-place/in-place (out/in) FFT computations for the experimental figures. For example, AutoFFT-out denotes an out-of-place transform of AutoFFT. Because the 1D complex-to-complex (C2C) FFT is the core operation of other transforms, we conduct an in-depth analysis of the performance of the 1D C2C FFT.

Fig. 10 shows the performances of the 1D C2C FFTs of AutoFFT, FFTW, ARMPL, and the baseline on the ARMv8 architecture. AutoFFT is faster than FFTW and ARMPL for both single-precision (SP) and double-precision (DP) sequences. From the C2C FFTs' performance curves on the

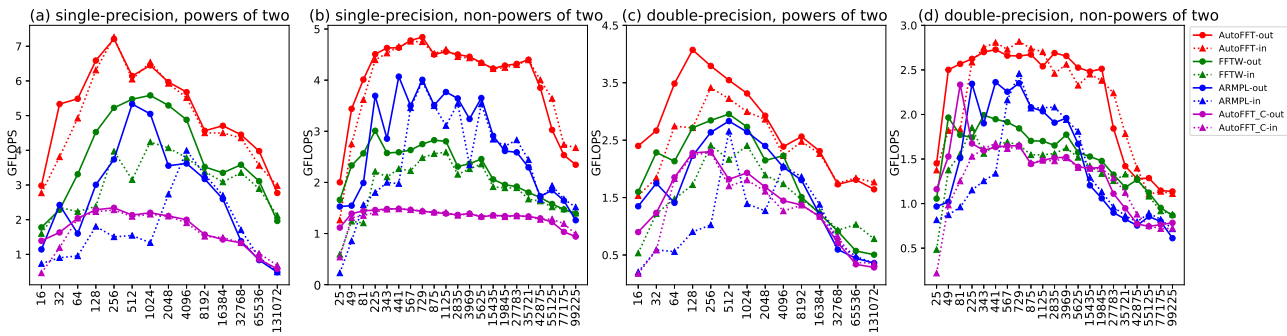


Fig. 10. The 1D C2C FFT performances on ARMv8 CPUs.

TABLE 2
The C2C FFT Average Speedups Obtained by AutoFFT on ARMv8 CPUs

Data Size	Single Precision			Double Precision		
	FFTW	ARMPL	Baseline	FFTW	ARMPL	Baseline
2^n	1.53	3.01	3.24	1.72	2.56	2.63
Non-2^n	1.98	1.77	2.99	1.49	1.63	1.76

TABLE 3
The C2C FFT Average Speedups Obtained by AutoFFT on Intel Haswell CPUs

Data Size	Single Precision			Double Precision		
	FFTW	MKL	Baseline	FFTW	MKL	Baseline
2^n	2.82	1.46	10.1	1.93	1.59	4.7
Non-2^n	2.84	1.52	8.32	1.97	1.72	4.44

ARMv8 architecture in Fig. 10, we can conclude the following. 1) These three libraries achieve similar performance trends. When the FFT size is small and the transformed data can reside in the cache system, the performance increases as the FFT size increases; when the FFT size is large and the cache system cannot hold all needed data, the cache miss rate is high; so the performance decreases as the FFT size increases. Hence, the performances of these libraries first increase and then decrease as the FFT size increases. 2) Compared with other libraries, the performance of AutoFFT stands out when processing FFTs of non-power-of-two sizes. In addition, the performance gaps between AutoFFT and the other libraries on single-precision sequences are larger than those on double-precision sequences. 3) Compared with AutoFFT and FFTW, ARMPL's performance gaps between the out-of-place transform and the corresponding in-place transform are larger. We believe that the performance of ARMPL's in-place transforms can be further improved. Table 2 lists the C2C FFT average speedups of AutoFFT compared with FFTW, ARMPL, and the baseline on ARMv8 CPUs.

Fig. 11 shows the performances of the 1D C2C FFTs for AutoFFT, FFTW, Intel MKL, and the baseline on the Intel Haswell architecture. Based on the performance curves of C2C FFTs on the Intel Haswell architecture in Fig. 11, we can conclude the following. 1) The performance trends among AutoFFT, FFTW, and Intel MKL are similar, but FFTW is generally slower than AutoFFT and Intel MKL. 2) Compared with AutoFFT's C kernels, its assembly kernels achieve higher speedup on Intel Haswell than on ARMv8. 3) Intel MKL performs very well when the FFT size is a power of two. For FFT sizes below 2,048, AutoFFT is faster than Intel MKL. However, when the FFT size exceeds 2,048, Intel MKL is close to that of AutoFFT, especially for DP floating-point data. There are two possible reasons for this difference in performance between AutoFFT and Intel MKL. First, in addition to the Cooley-Tukey algorithm, Intel MKL may adopt other FFT algorithms, such as the four-step FFT algorithm (or six-step, depending on the number of transpositions) [49], the split-radix [5], and the Rader-Brenner [6] algorithms, to obtain higher performance at large scales. Second, because our FFT kernels are autogenerated, they can be further improved by adopting instruction reordering and data prefetching. Table 3 lists the C2C FFT average speedups of AutoFFT compared with FFTW, ARMPL, and the baseline on Intel Haswell CPUs.

Fig. 12 shows the performances of the 1D C2C FFTs for AutoFFT, FFTW, and the baseline on the AMD Zen architecture. As shown in Fig. 12, the performance of AutoFFT is relatively stable at different sizes, and even when the data size is large, it still achieves good performance. Besides, the

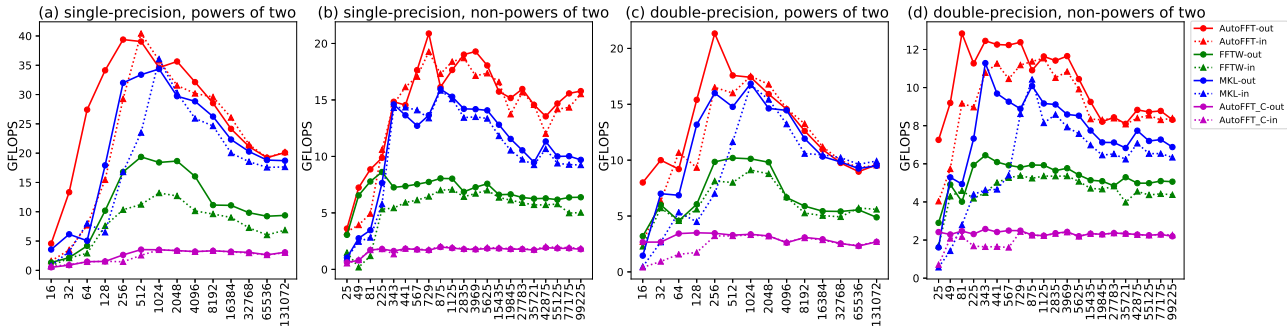


Fig. 11. The 1D C2C FFT performances on Intel Haswell CPUs.

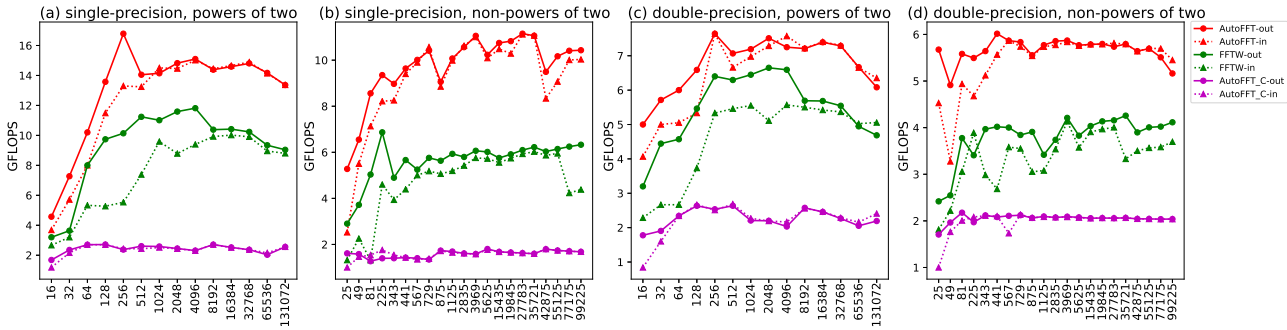


Fig. 12. The 1D C2C FFT performances on AMD Zen CPUs.

TABLE 4
The C2C FFT Average Speedups Obtained by AutoFFT on AMD Zen CPUs

Data Size	Single Precision		Double Precision	
	FFTW	Baseline	FFTW	Baseline
2^n	1.59	5.33	1.35	3
Non-2^n	1.92	5.96	1.61	2.77

TABLE 5
The Average Speedups of the Symmetric (the Horizontal and Vertical Symmetries) and Periodic Properties Over the DFT Implementation on ARMv8 CPUs

Data Size	Single Precision			Double Precision		
	Opt	*Sym*	*Hor*	*Opt*	*Sym*	*Hor*
6^n	2.9	2.51	1.73	2.74	2.23	1.64
9^n	2.76	2.3	1.51	2.59	2.14	1.6

performances of AutoFFT's in-place and out-of-place transforms are close and stable, however, most of FFTW's in-place transforms are slower than its corresponding out-of-place transforms. Table 4 lists the C2C FFT average speedups of AutoFFT compared with FFTW, ARMPL, and the baseline on AMD Zen CPUs. Compared with FFTW, AutoFFT has significant performance advantages on the AMD Zen architecture.

As illustrated in Section 4, the symmetric (the horizontal and vertical symmetries, described in Section 4.1) and periodic (described in Section 4.2) properties of the DFT matrix are adopted to reduce the number of floating-point operations of the butterflies. We call these three algorithmic optimizations as the horizontal optimization, the vertical optimization, and the periodic optimization, respectively. In this part, we evaluate the performance effect of these three optimizations by comparing them with the DFT implementation (the baseline for these three algorithmic optimizations). For simplicity, we define *Opt* to represent the fully optimized kernels (optimized by the horizontal, vertical, and periodic optimizations), *Sym* to represent the kernels optimized by the horizontal and vertical optimizations, and *Hor* to represent the kernels only optimized by the horizontal optimization. Fig. 13 presents the performance speedups of *Opt*, *Sym*, and *Hor* by comparing them with the baseline. From this figure, we can see the performance improvements brought by these three optimizations step by step. Theoretically, for non-power-of-two radices, the horizontal and vertical symmetries reduce arithmetic operations by a factor of 4 (Both the horizontal and vertical optimizations can halve the amount of the arithmetic operations by combining like terms). However, *Sym* and *Hor* are only up to 2.79 times and 1.87 times faster than the baseline, respectively. The main reason why *Sym* cannot achieve the theoretical 4x performance boost (and why *Hor* cannot achieve the theoretical 2x performance boost) is that although we reduced the arithmetic operations of the Cooley-Tukey FFT kernels by a factor of 4, we did not reduce any memory operations. As we all know, memory

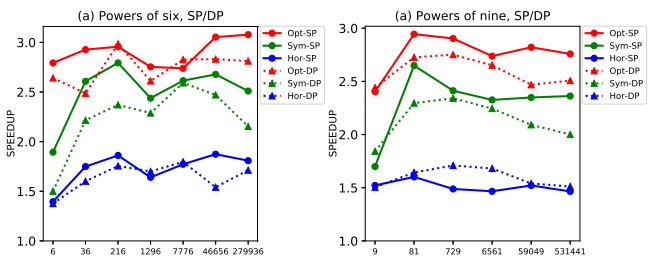


Fig. 13. These two subfigures present the performance speedups of *Opt*, *Sym*, and *Hor* over the DFT implementation of power-of-six and power-of-nine sizes respecively on the ARMv8 architecture. From these two subfigures, we can clearly see the empirical performance effect of the horizontal, vertical, and periodic optimizations.

operations are much more costly than arithmetic operations in modern CPUs. Because the periodic property can further reduce the arithmetic floating-point operations by a factor of $gcd(k, r)$ for each output Y_k of radix r, *Opt* achieves better performance than *Sym*. The performance curves in Fig. 13 confirm this point: the red curves of these two subfigures are above the corresponding green curves. The average speedups of *Opt*, *Sym*, and *Hor* over the baseline are listed in Table 5.

In many applications, the inputs or outputs are real numbers. AutoFFT currently supports the following transforms [50]: real-to-complex (R2C); complex-to-real (C2R); three types of real-to-real (R2R) transforms (real-to-"half-complex"/"half-complex"-to-real (R2HC/HC2R); the discrete Hartley transform (DHT); four kinds of discrete cosine transforms (DCT I∼IV); and four kinds of discrete sine transforms (DST I∼IV)). To reduce unnecessary computations and memory access, we adopt two reduction methods to summarize and extract unified optimization patterns of real FFTs. The first one is the complex reduction. For FFTs with N real number inputs, the complex reduction is defined to take N real numbers as $N/2$ complex numbers and perform the $N/2$-point complex FFT, after that, split operations are applied on the transformed results. Each real FFT conducts a split operation according to its definitions. The second one is the real reduction. Because DCT/DST contains special symmetric properties in its inputs, the real reduction is defined to reduce a N-point real FFT to a $N/2$-point real FFT. Due to the space limitations of this paper, we present the performances of the R2C/C2R and R2R REDFT01/REDFT10 real FFTs below.

In Table 6, we provide the real FFT average speedups on ARMv8, Intel Haswell, and AMD Zen architectures. 1) On the ARMv8 architecture, because ARMPL does not support real FFTs other than R2C/C2R FFTs, we provide the performances of only ARMPL's R2C/C2R FFTs here. Fig. 14 shows that the performances of AutoFFT's real FFTs outperform those of FFTW, ARMPL, and the baseline, especially when the FFT size is large. In general, most of the characteristics of the real FFTs' performance curves are similar to the C2C FFTs' performance curves on the ARMv8 architecture. However, compared with other libraries, ARMPL's performance curves of R2C/C2R FFTs decrease faster as the FFT size increases. Its performance curves are very close to or even lower than those of the baseline at large scales, especially for the in-place transforms. 2) Fig. 15 shows the performances of real FFTs of AutoFFT, FFTW, Intel MKL, and the baseline on the Intel Haswell architecture. For R2C/C2R FFTs, FFTW is generally slower than AutoFFT and Intel MKL, and AutoFFT is faster than MKL FFT in most cases, as shown in Figs. 15a and 15b. For R2R REDFT01/REDFT10,

TABLE 6
The Real FFT Average Speedups Obtained by AutoFFT on ARMv8, Intel Haswell, and AMD Zen Architectures

Architecture	FFT Type	Single Precision				Double Precision			
		FFTW	ARMPL	MKL	Baseline	FFTW	ARMPL	MKL	Baseline
ARMv8	R2C/C2R	1.71	2.01	-	2.82	1.61	1.91	-	1.77
	R2R	2.04	-	-	2.66	1.51	-	-	1.58
Intel Haswell	R2C/C2R	2.54	-	1.64	7.98	1.9	-	1.47	4.14
	R2R	4.42	-	2.1	5.99	3.57	-	2.06	3.07
AMD Zen	R2C/C2R	1.76	-	-	5.09	1.56	-	-	2.48
	R2R	3.54	-	-	3.94	2.47	-	-	2

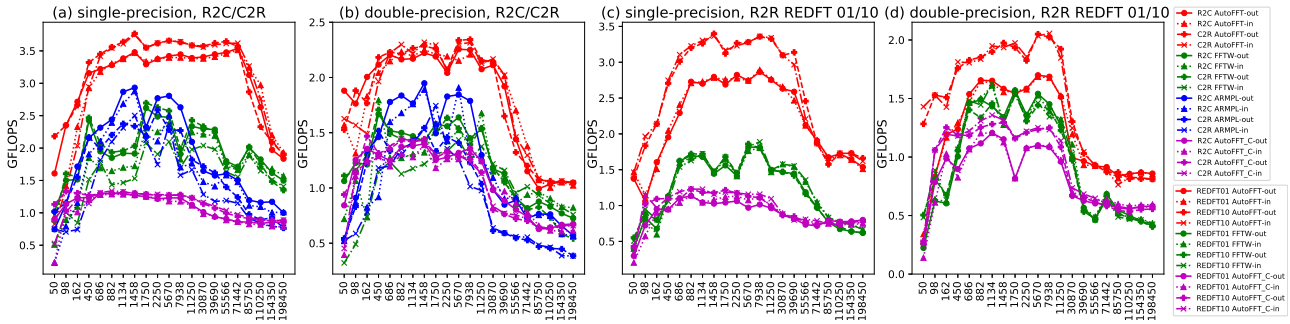


Fig. 14. The 1D real FFT performances on ARMv8 CPUs.

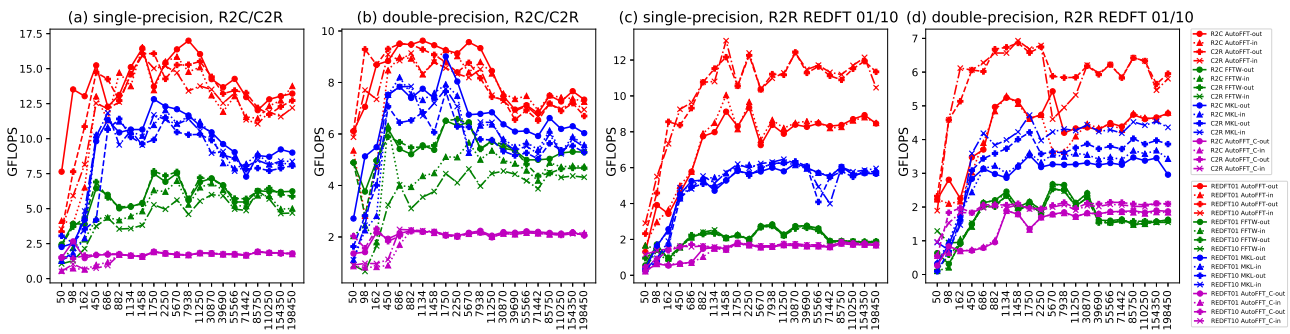


Fig. 15. The 1D real FFT performances on Intel Haswell CPUs.

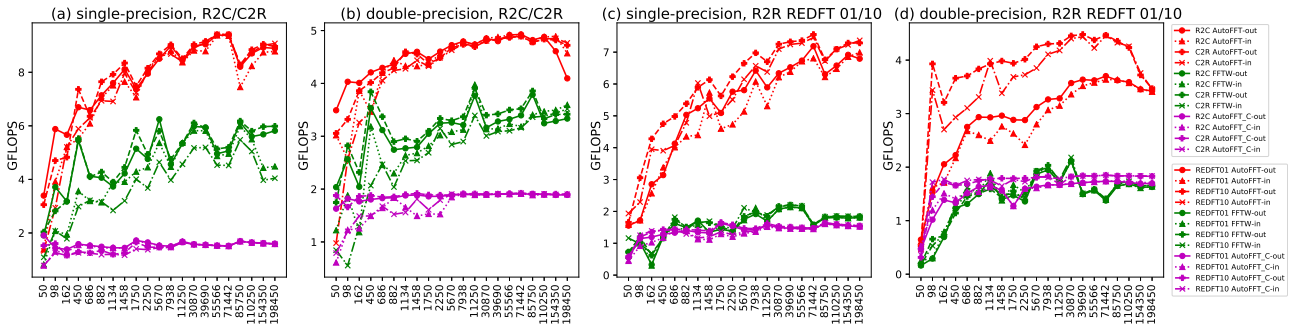


Fig. 16. The 1D real FFT performances on AMD Zen CPUs.

AutoFFT outperforms the other two libraries, and FFTW is much slower than AutoFFT and Intel MKL, and the R2R FFT performances of FFTW and the baseline are very close, as shown in Figs. 15c and 15d. 3) Fig. 16 presents the performances of real FFTs of AutoFFT, FFTW and the baseline on the AMD Zen architecture. The performances in Fig. 16 show that AutoFFT outperforms FFTW, and we believe that AutoFFT is the best choice for the AMD Zen CPUs. Different from R2C/C2R FFTs in Figs. 16a and 16b, we can see that the R2R FFT performances of FFTW and the baseline are very close, as shown in Figs. 16c and 16d. This phenomenon also exists in Intel Haswell architecture, and we believe that FFTW can further optimize real FFTs and achieve better performances on these two architectures.

TABLE 7
The Average and Maximum Speedups of AutoFFT

Architecture	Software	Average	Max
ARMv8	FFTW	1.7	2.04
	ARMPL	2.15	3.01
	Baseline	2.43	3.24
Intel Haswell	FFTW	2.75	4.42
	MKL	1.7	2.1
	Baseline	6.09	10.1
AMD Zen	FFTW	1.98	3.54
	Baseline	3.82	5.96

At the end of this section, we summarize the average and maximum performance improvements of AutoFFT by comparing it with FFTW, ARMPL, Intel MKL, and the baseline on ARMv8, Intel Haswell, and AMD Zen architectures in Table 7.

8 Conclusion

This paper proposes a template-based framework named AutoFFT that makes full use of the experience of the domain and optimization experts to automatically generate extremely high-performance FFT code of radices of all natural numbers for ARM, Intel, and AMD platforms. AutoFFT thus substantially reduces the laborious work of developing assembly kernels manually. The experiments show that AutoFFT performs generally better than FFTW, ARMPL, and Intel MKL. Our future work will concentrate on extending the template-based methodology to other numerical algorithms.

Acknowledgments

This work was supported by the National Key Research and Development Program of China under Grant Nos. 2107YFB0202105, 2016YFB0200803, and 2017YFB0202302; the National Natural Science Foundation of China under Grant Nos. 61602443, 61432018, 61521092, and 61502450.

References

[1] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Math. Comput.*, vol. 19, no. 90, pp. 297–301, 1965.
[2] C. M. Rader, "Discrete fourier transforms when the number of data samples is prime," *Proc. IEEE*, vol. 56, no. 6, pp. 1107–1108, Jun. 1968.
[3] L. Bluestein, "A linear filtering approach to the computation of discrete fourier transform," *IEEE Trans. Audio Electroacoustics*, vol. 18, no. 4, pp. 451–455, Dec. 1970.
[4] D. Kolba and T. Parks, "A prime factor FFT algorithm using high-speed convolution," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. 25, no. 4, pp. 281–294, Aug. 1977.
[5] P. Duhamel and H. Hollmann, "Split radix'FFT algorithm," *Electron. Lett.*, vol. 20, no. 1, pp. 14–16, 1984.
[6] C. Rader and N. Brenner, "A new principle for fast fourier transformation," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. 24, no. 3, pp. 264–266, Jun. 1976.
[7] J. W. Cooley, P. A. Lewis, and P. D. Welch, "The fast fourier transform and its applications," *IEEE Trans. Educ.*, vol. 12, no. 1, pp. 27–34, Mar. 1969.
[8] K. Asanovic *et al.*, "The landscape of parallel computing research: A view from berkeley," EECS Dept., Univ. California at Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2006–183, 2006.
[9] P. Costa, "A FFT-based finite-difference solver for massively-parallel direct numerical simulations of turbulent flows," *Comput. Math. Appl.*, vol. 76, no. 8, pp. 1853–1862, 2018.
[10] Z. Li *et al.*, "Efficient parallel optimizations of a high-performance SIFT on GPUs," *J. Parallel Distrib. Comput.*, vol. 124, pp. 78–91, 2019.
[11] D. Zhang, Z. Chen, C. Xiao, M. Qin, and H. Wu, "Accurate simulation of turbulent phase screen using optimization method," *Optik*, vol. 178, pp. 1023–1028, 2019.
[12] C. Gong, W. Bao, and G. Tang, "A parallel algorithm for the Riesz fractional reaction-diffusion equation with explicit finite difference method," *Fractional Calculus Appl. Anal.*, vol. 16, no. 3, pp. 654–669, 2013.
[13] C. Gong, W. Bao, G. Tang, B. Yang, and J. Liu, "An efficient parallel solution for Caputo fractional reaction–diffusion equation," *J. Supercomput.*, vol. 68, no. 3, pp. 1521–1537, 2014.
[14] M. Frigo and S. G. Johnson, "The fastest Fourier transform in the west," Massachusetts Inst. Technol., Cambridge, MA, USA, Tech. Rep. MIT-LCS-TR-728, Sep. 1997.
[15] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 1998, pp. 1381–1384.
[16] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proc. IEEE*, vol. 93, no. 2, pp. 216–231, Feb. 2005.
[17] ARM, "Arm performance libraries (ArmPL) 19.2.0," 2019. [Online]. Available: https://static.docs.arm.com/101004/1920/arm_performance_libraries_reference_101004_1920_00_en.pdf
[18] E. Wang *et al.*, "Intel math kernel library," in *High-Performance Computing on the Intel® Xeon Phi™*. Berlin, Germany: Springer, 2014, pp. 167–188.
[19] Intel, "Intel math kernel library (Intel MKL) 2019 update 4," 2019. [Online]. Available: https://software.intel.com/en-us/mkl
[20] Z. Li *et al.*, "AutoFFT: A template-based FFT codes auto-generation framework for arm and x86 CPUs," in *Proc. Int. Conf. High Perform. Comput., Netw. Storage Anal.*, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356138
[21] G. Bruun, "z-transform DFT filters and FFT's," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. 26, no. 1, pp. 56–63, Feb. 1978.
[22] J. Xiong, J. Johnson, R. Johnson, and D. Padua, "SPL: A language and compiler for DSP algorithms," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2001, pp. 298–308. [Online]. Available: http://doi.acm.org/10.1145/378795.378860
[23] AMD, "AOCL: AMD optimizing CPU libraries," 2020. [Online]. Available: https://developer.amd.com/amd-aocl/
[24] IBM, "ESSL: IBM engineering and scientific subroutine library," 2020. [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSFHY8_6.1/navigation/welcome.html
[25] Apple, "The apple accelerate libraries - vDSP," 2020. [Online]. Available: https://developer.apple.com/documentation/accelerate/vdsp/fast_fourier_transforms
[26] D. Mirković, R. Mahasoom, and L. Johnsson, "An adaptive software library for fast fourier transforms," in *Proc. 14th Int. Conf. Supercomput.*, 2000, pp. 215–224. [Online]. Available: http://doi.acm.org/10.1145/335231.335252
[27] S. Lennart Johnsson, "Automatic performance tuning in the UHFFT library," in *Proc. Int. Conf. Comput. Sci. Lecture Notes Comput. Sci.*, 2001, pp. 71–80.
[28] M. Puschel *et al.*,"SPIRAL: Code generation for DSP transforms," *Proc. IEEE*, vol. 93, no. 2, pp. 232–275, Feb. 2005.
[29] F. Franchetti *et al.*, "SPIRAL: Extreme performance portability," *Proc. IEEE*, vol. 106, no. 11, pp. 1935–1968, Nov. 2018.
[30] F. Franchetti, Y. Voronenko, and M. Püschel, "Formal loop merging for signal transforms," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 315–326, 2005.
[31] ARM, "Arm Ne10 project," 2020. [Online]. Available: https://github.com/projectNe10/Ne10
[32] D. Takahashi, "FFTE: A fast fourier transform package," 2014. [Online]. Available: http://www.ffte.jp/
[33] A. Blake and M. Hunter, "Dynamically generating FFT code," *J. Signal Process. Syst.*, vol. 76, no. 3, pp. 275–281, 2014.
[34] Nvidia, "CuFFT library," 2020. [Online]. Available: https://docs.nvidia.com/pdf/CUFFT_Library.pdf
[35] AMD, "A software library containing FFT functions written in OpenCL," 2020. [Online]. Available: https://github.com/clMathLibraries/clFFT

[36] D. Petre, A. T. Lake, and A. Hux, "OpenCL™ FFT optimizations for intel® processor graphics," in *Proc. 4th Int. Workshop OpenCL*, 2016, pp. 12:1–12:4. [Online]. Available: http://doi.acm.org/10.1145/2909437.2909451
[37] A. Nukada, Y. Maruyama, and S. Matsuoka, "High performance 3-D FFT using multiple CUDA GPUs," in *Proc. 5th Annu. Workshop Gen. Purpose Process. Graph. Process. Units*, 2012, pp. 57–63. [Online]. Available: http://doi.acm.org/10.1145/2159430.2159437
[38] Y. Dotsenko, S. S. Baghsorkhi, B. Lloyd, and N. K. Govindaraju, "Auto-tuning of fast fourier transform on graphics processors," *SIGPLAN Notices*, vol. 46, no. 8, pp. 257–266, Feb. 2011. [Online]. Available: http://doi.acm.org/10.1145/2038037.1941589
[39] Y. Li, Y.-Q. Zhang, Y.-Q. Liu, G.-P. Long, and H.-P. Jia, "MPFFT: An autotuning FFT library for OpenCL GPUs," *J. Comput. Sci. Technol.*, vol. 28, no. 1, pp. 90–105, 2013.
[40] A. Gholami, J. Hill, D. Malhotra, and G. Biros, "AccFFT: A library for distributed-memory FFT on CPU and GPU architectures," *CoRR*, vol. abs/1506.07933, 2015. [Online]. Available: http://arxiv.org/abs/1506.07933
[41] C. Cecka, "Low communication FMM-accelerated FFT on GPUs," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2017, pp. 54:1–54:11. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126919
[42] T. G. Stockham, Jr., "High-speed convolution and correlation," in *Proc. Spring Joint Comput. Conf.*, 1966, pp. 229–233. [Online]. Available: http://doi.acm.org/10.1145/1464182.1464209
[43] P. N. Swarztrauber, "Vectorizing the FFTs," in *Parallel Computations*, G. Rodrigue, Ed. New York, NY, USA: Academic, 1982, pp. 51–83. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780125921015500075
[44] Intel, "Intel 64 and IA-32 architectures optimization reference manual (chapter 2.1)," 2016. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf
[45] A. V. Oppenheim, *Discrete-Time Signal Processing*. New Delhi, India: Pearson Education India, 1999.
[46] M. Frigo and S. G. Johnson, "The benchmarking methodology of benchFFT," 2020. [Online]. Available: http://www.fftw.org/speed/
[47] M. Frigo and S. Johnson, "benchFFT," 2020. [Online]. Available: http://www.fftw.org/benchfft
[48] Intel, "Intel math kernel library developer reference's Appendix C: FFTW interface to Intel math kernel library," 2020. [Online]. Available: https://software.intel.com/sites/default/files/mkl-2019-developer-reference-c_2.pdf
[49] D. H. Bailey, "FFTs in external or hierarchical memory," *J. Supercomput.*, vol. 4, no. 1, pp. 23–35, 1990.
[50] X. Wang, H. Jia, Z. Li, and Y. Zhang, "Implementation and optimization of multi-dimensional real FFT on ARMv8 platform," in *Proc. Int. Conf. Algorithms Archit. Parallel Process.*, 2018, pp. 338–353.

Zhihao Li is currently working toward the PhD degree with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. In addition, he is a joint PhD student with the School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, Georgia. His research interests include high performance computing, optimized FFT library, and many-core/large-scale parallel programming method.

Haipeng Jia received the PhD degree from the Ocean University of China, Qingdao, China, in 2012. He was a visiting PhD student with the Institute of Software, Chinese Academy of Sciences from 2010 to 2012. He is currently an assistant professor with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences. His research interests include heterogeneous computing and many-core parallel programming method.

Yunquan Zhang (Member, IEEE) received the PhD degree with computer software and theory from the Chinese Academy of Sciences, Beijing, China, in 2000. He is currently a full professor of computer science with the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include the areas of high performance computing, with particular emphasis on large scale parallel computation and programming models, and high-performance parallel numerical algorithms.

Tun Chen received the master degree with computer science from Hunan Normal University, Changsha, China, in 2018. He is currently working toward the PhD degree with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His research interests include high performance computing, heterogeneous computing, in-core parallelism, and optimized FFT library.

Liang Yuan received the PhD degree from the Institute of Software, Chinese Academy of Sciences, Beijing, China, in 2013. He was a visiting PhD student with the University of Rochester, in 2011. He is currently an assistant professor with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences. His research interests include large-scale parallel computing and heterogeneous computing.

Richard Vuduc (Member, IEEE) received the BS degree with computer science from Cornell University, Ithaca, New York, and the PhD degree in computer science from the University of California, Berkeley, California. He is currently a professor with the School of Computational Science and Engineering, Georgia Institute of Technology. His research lab, the HPC Garage, is interested in high-performance computing, with an emphasis on algorithms, performance analysis, and performance engineering.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Fireplug: Efficient and Robust Geo-Replication of Graph Databases

Ray Neiheiser, Luciana Rech, Manuel Bravo, Luís Rodrigues, *Senior Member, IEEE*, and Miguel Correia, *Senior Member, IEEE*

Abstract—Although graph-databases have been assuming an increasing relevance in applications that exhibit strong dependability requirements, including tolerance to malicious faults, few works have addressed Byzantine fault tolerance in this particular context, and previous attempts suffer from lack of flexibility and poor performance. This article describes and evaluates Fireplug, a flexible architecture to build robust geo-replicated graph databases. Fireplug can be configured to tolerate from crash to Byzantine faults, both within and across different datacenters. Furthermore, Fireplug is robust to bugs in existing graph database implementations, as it allows to combine multiple graph database instances in a cohesive manner. Thus, Fireplug can support many different deployments, according to the performance/robustness trade-offs imposed by the target application. Our evaluation shows that Fireplug is able implement Byzantine fault tolerance without penalty when compared to the built-in replication mechanism of Neo4j, which only supports crash faults. Additionally, performance optimizations introduced by Fireplug improve the overall performance by up to 900 percent in geo-replicated scenarios.

Index Terms—Graph databases, geo-replication, N-version programming, byzantine faults

1 Introduction

Graphs offer an elegant data representation for problems that would not be easily expressed otherwise. In fact, many areas benefit from the use of graphs, including social and natural sciences, engineering, and economics [1]. Unsurprisingly, the increasing number and relevance of applications using graphs as a data model spurred the development of several graph databases which are optimized to support graph storage and query. Examples include Neo4j [2], OrientDB [3], and TAO [4]. These specialized databases can outperform classical relational databases to support graph processing: for instance, Neo4j was shown to be 3x faster than MySQL on several tasks, such as graph traversal [5].

Interestingly, some of the applications that leverage graph databases often require simultaneously strong consistency, security, and scalability [6]. For instance, Neo4j and OrientDB customers include security firms, investigation units, media companies (*Sky*, *Comcast*, *Warner*), and trade companies (*Ebay* or *Global 500 Logistics*), which use graph databases to offer real time product routing and delivery to their clients [6], [7]. Therefore, deriving solutions that can replicate graph databases in an efficient and robust manner, is a challenge not only of technical interest but also of practical relevance.

Despite the facts above, and somehow surprisingly, very few works have addressed Byzantine fault tolerance in this particular context. Furthermore, as we will discuss in the related work section, the few works that have attempted to do so either offer inflexible, non-customizable, solutions, present poor performance, and are prone to attacks that exploit vulnerabilities on the underlying graph-database software (see Section 2). In this paper, we describe the design and implementation of Fireplug, an efficient architecture to build robust geo-replicated transactional graph databases, that aims at filling this gap.

Fireplug can be configured to tolerate faults on individual machines and/or faults that compromise (or disconnect) an entire datacenter. Further, Fireplug can be configured to tolerate different types of faults, from crash faults to arbitrary faults (also known as Byzantine faults). Tolerating Byzantine faults is key to build a dependable system as the causes for both natural and human-driven Byzantine faults are becoming more prevalent, including for instance, the increasing gate density in silicon and the threat of cyber-criminality. Furthermore, Byzantine faults can cause serious revenue loss, e.g., the Stuxnet worm [8], identified in 2010, caused substantial delays in nuclear research at Iran. More recently, the attack on the Linux Mint distribution in 2016 [9], in which a corrupted version was uploaded to their site, compromised users that installed it. Similar attacks could be made on graph databases, for instance, to disable fraud detection systems.

Since a vulnerability in the codebase of a given graph database can expose all running instances to correlated faults, Fireplug supports software diversity and implements N-version programming, letting different replicas deploy

- *R. Neiheiser is with the Departamento de Engenharia de Automação e Sistemas, Universidade Federal de Santa Catarina, Florianópolis 88040-900, Brazil. E-mail: ray.neiheiser@googlemail.com.*
- *L. Rech is with the Departamento de Informática e Estatística, Universidade Federal de Santa Catarina, Florianópolis 88040-900, Brazil. E-mail: luciana.rech@ufsc.br.*
- *M. Bravo is with the IMDEA Software Institute, Madrid, Spain. E-mail: manuel.bravo@imdea.org.*
- *L. Rodrigues and M. Correia are with the INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Lisbon 1649-004, Portugal. E-mail: {ler, miguel.p.correia}@tecnico.ulisboa.pt.*

Manuscript received 15 June 2018; revised 14 Feb. 2020; accepted 12 Mar. 2020. Date of publication 18 Mar. 2020; date of current version 9 Apr. 2020. (Corresponding author: Ray Neiheiser.) Recommended for acceptance by C. A. Morin. Digital Object Identifier no. 10.1109/TPDS.2020.2981019

1045-9219 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

different graph databases [10]. This shields Fireplug from bugs and attacks on vulnerabilities of specific graph databases' implementations: code made by independent teams, even using different programming languages, that goes through different release procedures, is less likely to suffer from the same bugs. Graph databases are large open source projects, where it is hard to eliminate all vulnerabilities that can be exploited by attackers (via buffer overflows, query language attacks, etc.). In 2016 only, a long list of open-source software with critical security flaws has been identified [11]. Examples include vulnerabilities in the glibc Linux library, in MySQL, and in OpenJDK. Fireplug currently supports 4 graph databases: Neo4j, OrientDB, Titan, and Sparksee. This makes attacks such as Stuxnet much harder against Fireplug that attackers would need to compromise different systems, using different vulnerabilities, to be successful. Furthermore, they would need to activate the attack simultaneously on different locations, to avoid the fault to be detected by the voting mechanisms that Fireplug uses.

From the algorithmic and system's implementation perspective, Fireplug makes the following contributions:

- Fireplug offers a common interface for multiple graph databases; this interface, that we have named GRADAM, augments the underlying implementation with multi-versioned data. Multi-versioning shields read-only transactions from aborting due to concurrent update-transactions. It is key to ensure efficient reads even when stronger guarantees are provided.
- Fireplug supports both intra- and inter-datacenter replication. Thus, it tolerates both errors that are common when using commodity hardware and natural or human-caused disasters. Replication also has the potential to reduce latency by letting clients interact with the closest datacenter(s), and to enhance scalability under read-dominated workloads.
- Fireplug implements novel variations of Byzantine fault-tolerant algorithms to implement both read-only and update transactions. These variations, that we named *Graph-DUR*, are based on the deferred update approach [12], a fault tolerance technique that has proved to be effective for supporting database replication. The implementations of these variants coexist in the system, and can be selected depending on the characteristics of the underlying infrastructure and of the system workloads.
- Fireplug's architecture is flexible. The goal is to allow a variety of configurations of the system such that application fault tolerance and performance requirements are met. For instance, N-version programming can be used to make each datacenter Byzantine fault-tolerant and then just assume crash faults at the level of an entire datacenter, making inter-datacenter replication more efficient. However, if one is concerned with attacks that can compromise an entire datacenter, one can also make the inter-datacenter operation Byzantine fault-tolerant. Fireplug can also be configured to run in a single datacenter and to tolerate only crash faults; this is interesting because Fireplug has no overhead compared to the native replication scheme protocols of the graph databases even though it provides stronger consistency and tolerates Byzantine failures, as these rely on single master algorithms and do not leverage the availability of multiple replicas.
- An open-source implementation of the resulting prototype is available on github.[1]

The system has been extensively evaluated. We have observed that Fireplug shows no overhead compared to the native replication mechanism of Neo4j when tolerating not only crash faults (as Neo4j) but also Byzantine faults. Additionally, our architecture shows significant performance gains when compared to more traditional architectures.

2 Related Work

Fault Tolerance. Fault tolerance is a fundamental property of dependable systems [13]. In this paper we are mainly concerned with tolerating two types of faults, namely *crash* faults [14] and *Byzantine* faults [15], [16]. We say that a node fails by crashing, if it operates correctly up to a point where it stops functioning. On the opposite, a node subject to a Byzantine fault can exhibit an arbitrary behaviour. Byzantine faults may have natural causes but may also be the result of a deliberate attack by a malicious intruder.

Database Replication. Many replication techniques leverage the existence of an atomic broadcast primitive like Paxos [17]—and its variants—which is one of the most widely used tools for tolerating not only crash faults but also Byzantine faults. Not surprisingly, a number of previous works have also considered the deployment of atomic broadcast protocols across multiple datacenters [18], [19], [20]. These protocols can be used to replicate databases in multiple datacenters, at different locations [21], [22].

Among these techniques, *Deferred Update Replication* (DUR) has been shown to be particularly effective [12]. In DUR, a transaction is executed only against one local replica; such that writes are locally cached, and reads are served locally. At commit time, the transaction's write and read set are sent to other replicas for validation, by means of an atomic broadcast primitive. If no conflicts are detected, writes are atomically executed, otherwise writes are discarded, and the transaction aborted. Of particular interest to us is the work of [23], that introduces an implementation of DUR able to tolerate Byzantine faults. Fireplug uses a variant of this last protocol that, unlike previous solutions, is specially designed for graph databases. The differences between our specialized DUR variant and [23] are twofold: our implementation considers graph semantics; and we avoid digital signatures for validating reads. Avoiding signatures in this context is relevant, not only because they are computationally expensive but, above all, because they consume additional storage. Since most objects in graph databases are very small, the overhead of maintaining signatures can increase the size of the database by an order of magnitude.

Some works have shown that it is possible to offer semantics stronger than serializability by using specialized, highly precise, clock synchronization services, namely, Spanner offers linearizability [22] and FaRM [24] offers opacity. None of these systems has been designed for graph databases or to

1. https://github.com/Raycoms/fireplug

tolerate Byzantine faults. The architecture proposed here is more general and more robust, as it avoids the use of clock synchronization to ensure safety properties, which could be a liability in the Byzantine setting.

N-version Programming. Research on N-version programming, or software diversity, started in the 1970s and raised considerable interest [25]. As discussed in [25], faults are often caused by design flaws. N-version programming aims at tolerating these design faults, using a range of independently-designed software components, which has also been shown to decrease the likelihood of malicious intrusions [26]. Unfortunately, due to the large deployment effort required to combine enough distinct implementations, N-version programming is—with a few exceptions [27]—seldom used practice.

MITRA [28] is an example of the use of software diversity to shield the system from Byzantine faults. Although it is designed for relational databases, a similar approach may be an asset when replicating graph databases.

Graph Databases. Graph Databases [29] are database management systems that have been optimized to store, query and update graph structures. In graph databases relationships are first-class citizens on the graph data model. This is not true in other database management systems, where relations between entities have to be inferred using other abstractions such as foreign keys, making the task of querying the graph an inefficient join-intensive procedure. To avoid these limitations, graph databases store pointers in the corresponding vertices and edges. Fireplug gathers a set of features that make it unique when compared to other graph databases. First, it efficiently tolerates not only crash faults but also Byzantine faults. Many graph databases implement a variant of semi-active replication for fault tolerance, but they tolerates only crash faults; they do not consider Byzantine faults. Finally, above all, it shields the system from software vulnerabilities by relying on software diversity.

Graph Processing. There is a recent surge of interest in large scale graph processing, starting with Google' s Pregel [30]. The objective is to process large graphs in parallel, similarly to what MapReduce does for unstructured data. Apache Giraph is an open source graph processing system that tolerates node crashes like Pregel [31]. There are several others that also tolerate crashes, such as PowerGraph [32] and Trinity [33]. Greft, instead, tolerates Byzantine faults [34]. Nevertheless, none of these systems uses replication for fault tolerance. They use checkpointing mechanisms that periodically save the state of processes in persistent storage, allowing future recovery in case of failure.

Wide Area BFT. Our algorithms are based on atomic broadcast protocols for the Byzantine fault-tolerant model, also known as BFT protocols [16]. A number of algorithms have been proposed to implement BFT in the wide-area and our work is inspired by these previous results. Most notably, Steward [20] proposes an hierarchical BFT protocol, where a node from each local group is elected to represent the group in a global group. As it will be explained later, we use the same general approach but a different implementation: while Steward requires members from the local group to sign all actions of the primary, to achieve better performance in the common case, we let the primary proceed alone, and use our knowledge about the application semantics to validate its actions before applying any updates. Recent work in the blockchain area has also started to address the wide-area performance of consensus. However, most of these works, such as [35], use dissemination strategies based on gossip, that fail to guarantee deterministic termination.

Summary of Related Work. The importance of graph databases has been growing and they are now used in applications that have strong reliability requirements. A framework that can provide Byzantine fault tolerance in a scalable, flexible, and efficient manner is needed. Many of the ingredients required to build such frameworks have been previous explored in the literature. However, they need to be adapted to operate efficiently with graph databases, and need to be integrated with each other to build a coherent whole.

3 Fireplug Overview

Fireplug is a transactional graph database management system. It is designed to run in one or more datacenters. Each datacenter runs one or several nodes (or servers) and each node runs a (potentially different) instance of the graph database. We target a full replication scenario, where all nodes maintain a full copy of the graph. Typically, the latency among nodes residing within the same datacenter is significantly smaller than the latency among nodes in different datacenters. This may have an impact on the performance of Fireplug but not on its correctness, as we assume an asynchronous system model. Naturally, the cost of coordinating replicas across multiple datacenters (in our case, using atomic broadcast) may not be negligible (we discuss these costs in the evaluation section). Still, experience with large-scale geo-replicated systems, such as Google's Megastore [21] and Spanner [22] has shown that several application are willing to pay these costs to ensure liveness in face of an outage that affects an entire datacenter.

Fireplug offers flexibility at multiple levels. (i) Each node can be configured to run a different graph database. Currently, Fireplug supports Neo4j, OrientDB, Titan, and Sparksee. These graph databases may be oblivious to the Fireplug replication mechanisms but, still, the resulting assembly behaves as a single, serializable, database. (ii) It permits multiple fault models to coexist in a single deployment. Note that in a system where all nodes must be configured to tolerate the same type of faults, one may be forced to choose the most restrictive model, unnecessarily degrading the overall system performance.

Note that although Fireplug has been designed to leverage software diversity at the level of the graph database, in the current prototype, the Fireplug middleware itself runs a single implementation. In theory, one could also run multiple implementations of our libraries in different nodes, but the development effort to derive such versions is out of the scope of our research project.

3.1 System Components

The main software components of Fireplug, depicted in Fig. 1, are the following:

- The application front-end machines (which we designate by clients). We assume that clients receive end-user requests and run the application.

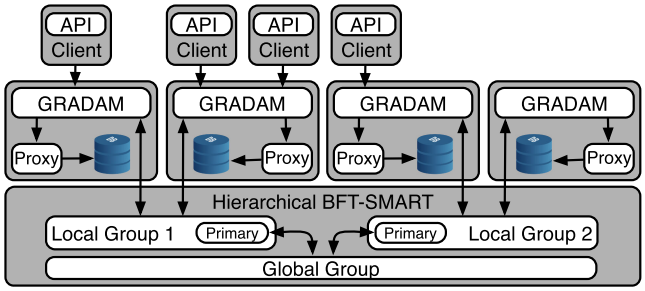


Fig. 1. Fireplug architecture.

- A middleware integration layer called the *common GRAph DAtabase replication Middleware*(GRADAM). Its goal is to support the inter-operation of multiple graph databases in a cohesive environment by unifying their interface, as each database is likely to offer a slightly different interface.
- A proxy that provides a uniform interface to the graph database so that GRADAM can abstract away these details. The proxy augments the underlying database to support multi-versioned data.
- Several variants of a new replication protocol, *Graph-DUR*, that implements deferred update replication tolerant to Byzantine faults and specifically designed for graph databases.
- As part of its replication protocol, Fireplug implements *Hierarchical BFT-SMaRt*, an atomic broadcast abstraction implemented as a hierarchical composition of multiple instances of the BFT-SMaRt service [16]. An instance of BFT-SMaRt that has members exclusively from the same datacenter is denoted a local group. An instance of BFT-SMaRt that has members from different datacenters is denoted a global group (our architecture uses a single global group).

3.2 Proxy

Due to the software diversity support by Fireplug, we defined a common interface (access class) to be used by the Fireplug clients. The interface defines the methods that each individual access class has to implement in order to ensure interoperability. The *start* and *terminate* methods are required to start a new instance of the database and terminate it. The *beginTransaction* and *endTransaction* methods delimit a transaction. We support the most fundamental operations, namely *create*, *read*, *update*, and *delete*, to ensure that the interface can be implemented by a wide range of graph databases. Each of these operations has its corresponding method in the common interface. These methods accept, as a parameter, either a vertice or an edge object. Fireplug requires an instance of GRADAM to be implemented for each different graph database. We do not see this as an impairment for adoption, because GRADAM is relatively easy to implement. In our prototype, the implementation of GRADAM for the different databases required the following number of lines of code: 509 for Titan, 673 for Sparksee, 441 for OrientDB, and 738 for Neo4j.

3.3 System Operation

Clients connect to an instance of GRADAM, typically at the closest server, and coordinate the execution of transactions that are composed of several read and write operations. Transactions are marked by the *startTransaction* and *endTransaction* delimiters. The responsibility of a GRADAM instance is threefold: (i) it bridges local clients with the local graph database replica, translating between the common interface (the one exposed to clients) and each particular graph database interface; (ii) it augments the underlying databases with multi-versioning data; (iii) it interacts with the remaining GRADAM instances, running on remote replicas, to ensure that all transactions that commit are serializable.

The communication among multiple GRADAM instances running at different nodes is coordinated by our own DUR protocol. For some operations, namely to implement update-transactions, the explicit exchange of messages among instances is done using an atomic broadcast abstraction. This can be configured to tolerate both crash and Byzantine faults, and to offer different qualities of service. As already mentioned, the atomic broadcast service leverages BFT-SMaRt [16], an open source library that implements Byzantine-tolerant state machine replication. Our implementation uses a hierarchical combination of multiple BFT-SMaRt groups and makes use of two broadcast services offered by BFT-SMaRt, namely a (non-ordered) reliable broadcast and a (totally ordered) atomic broadcast (later detailed in Section 4.2).

4 Graph-DUR: Replication in Fireplug

Replication in Fireplug is managed using several variants of the Byzantine-tolerant DUR proposed in [23]. We first discuss the major differences between our variants (that we have named Graph-DUR) and [23]. Then, we describe how update and read-only transactions are processed. Finally, we detail the implementation of the atomic broadcast primitive integrated into Fireplug, a fundamental abstraction for Graph-DUR.

4.1 Adaptation of DUR for Graph Databases

Semantic-Awareness. Graph-DUR considers the semantics of the graph structure in an effort to reduce conflicts, a technique commonly used in both transactional memory [36], [37] and geo-replicated distributed systems [38], [39].

First, clients can potentially merge updates before pushing them to Fireplug to reduce the number of operations that need to be managed by the underlying databases. For instance, updating a vertex can be merged with its preceding vertex creation request, and deleting a vertex invalidates preceding updates or creations on the same vertex.

Second, the conflict detection algorithm has been adapted to also take the semantics of graph operations into account, in particular, we avoid flagging conflicts when the operations are known to be commutative. For instance, imagine two concurrent transactions attempting to remove the same vertex from the graph. In a transactional NoSQL database, the conflict detection algorithm would abort one of the transactions, given that both transactions "update" the same key. Our conflict detection algorithm takes into account that both operations commute, allowing both to commit. In order to implement such conflict detection algorithm, Graph-DUR requires clients to explicitly track not only reads and writes, but also create and delete operations. Thus, commutative operations can be efficiently spotted at the certification phase and unnecessary aborts are precluded.

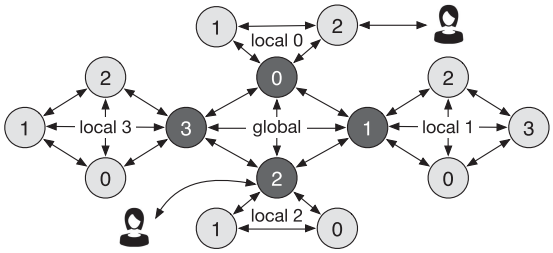


Fig. 2. Hierarchical architecture.

Signature-Free Read Validation. In a Byzantine-tolerant setting a single instance of the database cannot be trusted. If the instance is faulty, it may return bogus or stale data. In [23] this is addressed by having the servers sign every data item. In the case of relational databases this is valid because it is possible to sign tables or sections. Nevertheless, when using graph databases, this is not a viable option. Different graph databases store their data in fairly independent nodes and relationships. Therefore, implementing this mechanism would imply signing every node and relationship in the graph, adding a severe overhead. In fact, depending on the size of the objects, adding signatures can increase the size of the database and the correlated bandwidth cost by an order of magnitude. The tradeoff is that we need to validate read-only transactions by comparing the contents of different instances. In order to decrease the complexity and delay caused by global validation, we propose several optimizations in Section 4.4.

4.2 Hierarchical Atomic Broadcast

Some variants of Graph-DUR require the execution of an atomic (totally) ordered broadcast primitive across multiple datacenters. We have implemented this primitive as a hierarchical composition of multiple instances of BFT-SMaRt.

Specification. We assume a set of n processes, $\Pi = \{p_1, \ldots, p_n\}$ which may fail by crashing, or by behaving maliciously. If a process does not fail, we say it is a *correct* process, and otherwise it is a *faulty* process.

Like other atomic broadcast primitives, our hierarchical atomic broadcast is defined by the HAB-broadcast(m) operation, where m is the message to be broadcast. The HAB-broadcast(m) operation may trigger the HAB-deliver(m) event at a process. The hierarchical atomic broadcast primitive satisfies the following properties:

Validity. If a process p_i delivers a message m, some process p_j has broadcast m.

Integrity. Every process delivers a message at most once.

Agreement. If a correct process p_i delivers a message m, then all correct processes eventually deliver m.

Total order. If two correct processes p_i and p_j deliver two messages m and m', then p_j delivers m before m' iff p_i delivers m before m'.

Instantiation. The general architecture of the implementation is depicted in Fig. 2. In each datacenter, we setup an atomic broadcast group that coordinates all replicas that reside in that datacenter. Then, one replica from each datacenter is elected to participate in an inter-datacenter atomic broadcast group—we refer to each of these nodes as *primary*. Thus, instead of coordinating all replicas in a single large atomic broadcast group—an approach that we refer as *flat*, coordination is achieved by executing a sequence of actions on the smaller intra-datacenter and inter-datacenter atomic broadcast groups. We denote the inter-datacenter group simply as the *global* group and the internal group in datacenter i as $local_i$. For liveness, we assume that the system is augmented with an unreliable failure detector that can trigger the change of a faulty or stalled primary [40].

The global group can be configured to tolerate f crash or f Byzantine faults, requiring $2f + 1$ or $3f + 1$ participants respectively. Note that if the global group is configured to tolerate Byzantine faults and less than $3f + 1$ datacenters are available, then datacenters may have to participate in the global group with more than one node. Local groups can also be either crash- or Byzantine-tolerant, offering extra flexibility when configuring the system to improve performance. If the global group is configured to only tolerate crashes, local groups will not be able to tolerate Byzantine faults. The right configuration is scenario-specific, and depends both on the user requirements and on the estimated power of the adversary.

Operation. Hierarchical Atomic Broadcast is implemented as follows. Clients initiate a HAB-broadcast by sending messages directly to the global group. These messages are then totally ordered by BFT-SMaRt (where consensus is used to assign a sequence number to each message) and delivered to all members of that group. Differently from the standard BFT-SMaRt implementation, in the HAB-broadcast variant, the message is delivered together with the signed votes from a quorum of members of the global group. This quorum of signatures certifies the sequence number that has been assigned to the message (by the global group). Then, each member of the global group broadcasts the message (along with the corresponding set of signatures) to all members of the local group. For this purpose, it uses the more efficient unordered Byzantine broadcast service of BFT-SMaRt. This ensures that if a correct member of the local group receives the message, all members do receive the message. Messages are delivered to the remaining members of each local group in the order defined by the global group. Later in the evaluation section we show that this strategy is much more scalable than trying to perform atomic broadcast on a single group that contains all replicas.

4.3 Update Transactions

Fireplug implements two different variations of the Graph-DUR protocol. Both variants execute transactions against a single replica and certify concurrent transactions in total order. However, each variant uses a different communication pattern and a different set of replicas involved in the certification. We have called these variants Hierarchical-Broadcast Global-Certification (HB) and Hierarchical-Certification Global-Apply (HC) respectively. Each of these variants represents a different trade-off between CPU and Network resource consumption. We explain these variants in the following subsections.

4.3.1 Client Algorithm

The client algorithm is the same for both variants of Graph-DUR and is presented in Algorithm 1. The clients maintains a read-set ($t.RS$), a write-set ($t.WS$), a create-set ($t.CS$), and

a delete-set ($t.DS$), where it stores all the objects that have been, respectively, read, updated, created, and deleted in the course of the transaction. Updates, deletions, and creations, which are cached at the client, will only reach the servers on commit. To make sure clients read the changes made during the transaction, on each read the result will be matched with the write-set and then updated accordingly. The client starts by selecting a node against which it will execute the update transaction optimistically.

Algorithm 1. Client Code

```
1: function BEGINTRANSACTION(t, transactionClass)        ▷ Start Transaction
2:     t.RS ← ∅                                          ▷ Initialize read-set
3:     t.WS ← ∅                                          ▷ Initialize write-set
4:     t.CS ← ∅                                          ▷ Initialize create-set
5:     t.DS ← ∅                                          ▷ Initialize delete-set
6:     t.server ← best_closest_server(transactionClass)  ▷ Pick a server
7:     t.ts ← ⊥                                          ▷ Transaction timestamp
8: end Function
9: function READ(t,oid)                                  ▷ Read operation
10:    if (oid, v, ts) ∉ (t.RS ∪ t.WS ∪ t.CS ∪ t.DS) then
11:        (oid,v,ts) ← t.server.SERVER_READ(oid)
12:        t.Rs ← (oid, v, ts)
13:        if t.ts = ⊥ then
14:            t.ts ← ts
15:        else if t.ts < ts then
16:            abort
17:        end if
18:    end if
19:    return v
20: end function
21: function CREATE(t,oid)
22:    t.CS ← (oid)                                      ▷ Save in create-set
23: end function
24: function DELETE(t,oid)
25:    t.DS ← (oid)                                      ▷ Save in delete-set
26: end function
27: function WRITE(t,oid, v)
28:    t.WS ← (oidi, v)                                  ▷ Save in write-set
29: end function
30: function ENDTRANSACTION(t)
31:    if HB-mode on then
32:        HAB-BROADCAST(hb-validation, t, ts, t.RS, t.WS, t.DS, t.CS) to all servers.
33:    else                                              ▷ HC mode on
34:        ABCAST(hc-validation, t, ts, t.RS, t.WS, t.DS, t.CS) to servers in global group.
35:    end if
36:    wait for identical (t, outcome) from f + 1 servers
37:    return outcome
38: end function
```

The client also keeps a variable $t.ts$, that stores the snapshot used for performing reads; all reads are consistent with this snapshot. This variable is initialized with the timestamp of the first object read by the transaction (line 14). Additionally, on each read, the timestamp of the object in the database has to be compared with the transactions *snapshotId* to guarantee the isolation property of the transaction. If the timestamp corresponds to a transaction which has been committed after the start of the current transaction, the transaction will be aborted (line 16). Since GRADAM supports multi-versioning, the database will retrieve a version matching the transactions *snapshotId* to avoid the abort.

From the client's perspective, the only difference between running transaction on HB or HC mode is how the validation of the transaction is requested when it attempts to commit (line 31). In HB mode, the validation request is sent to all replicas using the hierarchical atomic broadcast primitive described before. In HC mode the validation request is sent just to the members of the global group, using the default atomic broadcast primitive of BFT-SMaRt.

Algorithm 2. Certification Procedure

```
1: global_ts = 0                                         ▷ Initiate global snapshot Id
2: function GET_SETVER_TS
3:     return global_ts
4: end function
5: function SERVER_READ(oid, snapshot)
6:     (oid, v, ts) ← RETRIEVE(oid, snapshot)
7: end function
8: function VALIDATE_READSET(read_set)
9:     for ∀ (oid, v, ts) ∈ read_set do
10:        (oid, v', ts') ← RETRIEVE(oid)
11:        if v ≠ v' or ts ≠ t' then
12:            return abort
13:        end if
14:    end for
15: end function
16: function VALIDATE_UPDATESET(ts, w_set, d_set, c_set)
17:    for ∀ (oid, v) ∈ c_set ∪ d_set ∪ w_set do
18:        (oid, v', ts') ← RETRIEVE(oid)
19:        if ts < t' then
20:            return abort
21:        end if
22:    end for
23: end function
24: function VALIDATE_TRANSACTION(ts, t.RS, t.WS, t.DS, t.CS) ▷ called in total order
25:    result ← VALIDATE_READSET(t.RS)
26:    if result = abort then
27:        return abort
28:    end if
29:    result ← VALIDATE_UPDATESET(ts,t.WS, t.DS, t.CS)
30:    return (result, global_ts)
31: end function
```

4.3.2 Certification Algorithm

The certification algorithm is presented in Algorithm 2. This algorithm can be executed by any server, as long as certification requests are processed in total order. Each node and relationship contain, additionally to the timestamp, the hash of the node itself. This will be used to check whether the server tries to send outdated, wrong or corrupted data to the client. The certification check, for a transaction ts, may be broken down into two fundamental steps:

- The validation procedures check if the transaction does not violate serializability. For that, we check that no transaction ts' with an update set that overlaps

with ts update set or with ts read set has committed after ts started (lines 8–15 and 16–23);
- The validation procedures checks that the values that have been returned to the client by the delegate replica match the values stored locally. For this purpose, the procedures check if all hashes $h' \in v' \in t.rs$ are equal to their corresponding values and hashes $h \in v$ in the database.

If the transaction passes both checks it can be committed. Otherwise it needs to be aborted.

4.3.3 Hierarchical-Broadcast Global-Certification

The Hierarchical-Broadcast Global-Certification variant of Graph-DUR is the one closer to the classical DUR. In this variant, update transactions are executed following the DUR algorithm: optimistically against a single replica (typically the closest one) and validated in parallel by all replicas in total order, to ensure serializability. As a result of this algorithm, all correct replicas certify the same sequence of transactions, in the same order. Thus, all correct replicas will reach a consistent decision regarding the commit or abort of the transaction.

The pseudo-code is given in Algorithm 3. When the validation request is received in total order from the hierarchical broadcast protocol, all nodes apply the validation procedure described before and inform the client of the outcome. Additionally, if the transaction commits, the update set is applied to the local database. HB requires a single execution of the totally ordered broadcast algorithm at commit time. This is used to ensure that all replicas can certify the transaction in exactly the same order. This ensures that the certification, that is deterministic, yields exactly the same results at all correct replicas. The drawback of this approach is that it suffers from redundant CPU consumption, as all replicas perform essentially the same work every time an update transaction attempts to commit.

Algorithm 3. Hierarchical-Broadcast Global-Certification

```
1: function HAB-DELIVER(hc-validation, ts, t.RS, t.WS, t.DS,
   t.CS) from client c        ▷ called in total order by all replicas
2: (result, global_ts) ← VALIDATE_TRANSACTION(ts, t.RS, t.WS,
   t.DS, t.CS)
3:    send result to client c
4:    if result = commit then
5:       APPLY_UPDATES(global_ts, t.WS, t.DS, t.CS)
6:    end if
7: end function
```

4.3.4 Hierarchical-Certification Global-Apply

The Hierarchical-Certification Global-Apply variant of Graph-DUR aims at trading processing time for communication. In this variant, a transaction is just certified by the nodes in the global group. These nodes produce a signed proof of the transaction outcome. Then, the updates associated with a transaction are broadcast together with the proof that the associated transaction was serializable and can be committed, and applied to the local datastore by all nodes without the need of going through local certification.

The pseudo code is given in Algorithm 4. In detail, when a client finishes a transaction containing update requests, it broadcasts the transaction to the global group only. The transaction is delivered in total order to the members of the global group that perform the certification (line 2), sign the result, and broadcast the result to the global group, using the unordered Byzantine tolerant primitive (line 3). Additionally, the servers send the result to the client (line 4). If the certification fails the transaction is aborted and no further steps are required. If the transaction commits, each member of the global group collects $f + 1$ identical commit messages (line 6). If no quorum can be achieved, the transaction is aborted. If a quorum is achieved, the update set is applied locally and sent to all members of the local group (lines 8 and 9). Finally, when the values are received by the members of the local group, each member checks if the $f + 1$ commit-votes are valid and then applies the update set in the order defined by *global_ts* (line 15).

Algorithm 4. Hierarchical-Certification Global-Apply

```
1: function ABCAST-DELIVER(hb-validation, ts, t.RS, t.WS, t.DS,
   t.CS) from client c   ▷ called in total order by replicas in the
   global group
2:    (result, global_ts)← VALIDATE_TRANSACTION(ts, t.RS, t.WS,
      t.DS, t.CS)
3:    RBCAST(hc-result, ts, result) to servers in global group.
4:    send result to client c
5:    if result = commit then
6:       result-set ← collect f + 1 RBCAST-DELIVER(hc-result, ts,
         commit) from other servers in the global group
7:       if |result-set| ≥ f + 1 then
8:          APPLY_UPDATES(global_ts, t.WS, t.DS, t.CS)
9:          RBCAST(global-apply, ts, result-set, global_ts, t.WS,
            t.DS, t.CS) to all server of my local group
10:      end if
11:   end if
12: end function
13: function ABCAST-DELIVER(global-apply, ts, result-set,
   global_ts, t.WS, t.DS, t.CS)
14:   if result-set is valid then
15:      APPLY_UPDATES(global_ts, t.WS, t.DS, t.CS)
16:   end if
17: end function
```

Note that, given that $f + 1$ votes to commit the transaction are enough to certify that the transaction validation succeeded, it is not strictly required that all members of the global group certify the transaction. Therefore, in the most common case (i.e, when no Byzantine faults occur) it is possible to load balance the work of certifying the transactions among the members of the global group. It is clear that HC, even with this optimization, requires significantly more communication than HB, given that $f + 1$ votes need to be collected before the update set can be applied. However, instead of requiring all nodes to validate the transaction, with HC only a quorum of nodes need to perform this task. In scenarios where capacity of nodes is very heterogeneous, HC can bring advantages since it is possible to offload the validation work to the more powerful machines.

4.4 Read-Only Transactions

To improve performance, read-only transactions are executed differently from classical DUR. Fireplug supports

read-only transactions with several different resilience levels. One dimension of resilience is concerned with the geographical scope of the fault: *locally-safe* reads, that cross check the results using two nodes of the same datacenter and; *globally-safe* reads, that cross check the results using nodes of different datacenters (globally-safe reads should be used if there is the threat of an entire datacenter becoming compromised). The other dimension of resilience is concerned with the database instance: *version-unaware* reads do not care about the database instance when validating reads and *version-aware* reads always validate read transactions using different graph databases (in order to resist to vulnerabilities that compromise all instances of the same graph database).

Furthermore, Fireplug considers 3 different implementations of each resilience level, namely: *pessimistic* read-only transactions (PR), *optimistic latency-driven* read-only transactions (OR-L), and *optimistic throughput-driven* read-only transactions (OR-T). We describe the three different implementations below:

- Pessimistic read-only transactions are executed as these were update transactions by broadcasting them, in total order, to all replicas in the global cluster. Thus, pessimistic read-only transactions are as expensive as update transactions. Nevertheless, this implementation has some advantages when compared to the other two implementations described below. First, it guarantees that all replicas have a state consistent with the local replica when validating the read transaction. This ensures that, if the local replica is correct, the validation of the read transaction always succeeds. Second, even if the local replica is faulty, the fault is always detected.
- Optimistic latency-driven also broadcasts the read-only transaction to all replicas, but use the unordered broadcast primitive instead. Therefore, when remote nodes attempt to validate the transaction, they may not have an up-to-date state. Nodes that are up-to-date (i.e., that have already installed the snapshot that was used for the client when executing the read transaction), can validate the transaction and send the result back to the client. The client can continue as soon as it received f additional validation confirmations from the desired set of replicas (the exact set depends on the desired resilience). This ensures a quicker validation, although it still requires global communication.
- Finally, optimistic throughput-driven reads is the least expensive implementation. The read transaction is broadcast to f other nodes with the desired resilience properties using point-to-point communication. This avoids global communication. Unfortunately, if the selected targets are not up-to-date, the client may be forced to select additional targets, which may increase the latency of reads.

In both optimistic executions if the read transaction returns an abort, read transactions are re-executed in pessimistic mode since the client cannot be sure if he read wrong or just outdated data. Although this may cause additional overhead in the worst case, given that workloads in graph databases are typically read-dominated [4], in most of the cases, the validation will succeed without requiring the execution of a totally ordered broadcast. This brings significant performance gains in the most frequent case.

4.5 Configuring Graph-DUR

As described before, Graph-DUR must be configured differently depending on the type of failures one wants to tolerate. In this section, we discuss how Graph-DUR should be configured to tolerate three types of failures: crash faults, Byzantine nodes, and Byzantine datacenters. We also briefly discuss how Graph-DUR can detect these failures and how to recover from them.

4.5.1 Crash Faults

When a node crashes or is suspected, if that node is not a primary, no special action is performed since it is transparently handled by the atomic broadcast protocol. But, if the node is a primary (member of the global group), corrective measures need to be performed. First a new primary is elected from the local group of the faulty node. If the old primary is still active (and was just slower), it will remove itself from the global group. If the old primary crashed, the new primary of the local group pro-actively joins the global group, obtains the delivery log of the global group, and reliably broadcasts to the local group all messages that have not been propagated to the local group by the faulty primary.

4.5.2 Tolerating Byzantine Nodes

A Byzantine node can be a non-primary or a primary. If it is a non-primary, any misbehaviour will be detected by a client, that will not be able to validate the result of the transaction executed against the faulty node. The client simply selects another server (the client may also report the fault, such that the server is eventually replaced). If the faulty node is a primary we need to consider two scenarios, namely, the faulty node can misbehave in the global group or in the local group (or in both). We discuss these two scenarios below.

If a faulty primary misbehaves in the global group in a manner that prevents the global group to make progress (for instance, it it is the global group's primary), it will be suspected by the correct nodes in the global group, forcing a view change. The correct members of the global group will then report this fact to the affected local group, such that a new primary is elected. A faulty primary may also behave correctly in the global group but fail to forward transactions to its local group. To detect this scenario, clients expect a confirmation that a write transaction has been applied from $f + 1$ members of each local group. Failure to collect such quorum from a given local group will also lead to the corresponding primary being suspected and to the election of a new primary.

4.5.3 Tolerating Byzantine Datacenters

Fireplug can also be configured to tolerate faults at the level of an entire datacenter, as long as the deployment includes at least $3f + 1$ datacenters or the datacenter offering multiple replicas to the global cluster is not faulty. In fact, the protocol described in the previous section tolerates the failure of f entire datacenters or of f machines in one of the datacenters.

5 Evaluation

In this section, we present the results of an experimental study of Fireplug. We answer the following questions: What are the differences in performance among the two update protocols (HB and HC) and how these compared with classical DUR based on a flat group? What is the impact of multi-versioning? How do the different read-only protocols perform? How well does Fireplug perform in terms of throughput when compared with the native replication schemes of the underlying graph databases?

The evauation shows not only the absolute performance of Fireplug but also its overhead with regard to different baselines: Section 5.1 compares the performance against non-hierarchical protocol, Section 5.5 compares with the native replication scheme of Neo4j, and Section 5.3 shows the overhead of the multi-version features.

In the experiments we use five different workloads: Read-only (100 percent Reads), Read-heavy (0.2 percent Writes), Conflict-prone (5-20 percent Writes), Balanced (25-75 percent Writes), and Write-only (100 percent Writes).

All workloads are based on real usage of graph databases (such as Facebook's TAO [4]) just varying the different write levels. The dataset has been synthetically generated using gMark, a well-founded approach for schema-driven generation of graphs and query workloads [41]. It represents a social-network and has 100,000 vertices and approximately 230,000 edges. Each experiment was executed three times in order to avoid statistical errors, and the results presented are the means of the throughput (operations per second) observed on all servers during a 6 minutes execution interval. We discarded the first and the last minute of each experiment, to avoid effects of the warm-up and cool-down periods.

The experiments were carried out in the Grid'5000 testbed [42]. Each database replica runs on a separate dedicated machine with 64 GB RAM and eight 2.0 GHz cores. Client instances were started on the same physical machines as the server they connect to. Some scenarios use multiple data centers: scenarios with 8 servers use two data centers, each with 4 servers each (2 representatives in the global group) and scenarios with 12 servers use 4 data centers each with 3 servers (one representative in the global group). Because the latency among Grid'5000 clusters is very small, we emulated larger geo-replicated scenarios by using `netem` [43] to add a 20 ms delay in the links that connect machines from different datacenters.

5.1 Update Transactions: HB Flat Versus HB Hieraquical

To compare the performance of the different protocols used to process update transaction we use a write-only workload and assess what is the maximum throughput achievable with each protocol in different scenarios. We set the percentage of writes to 100 percent, to assess the load writes impose on the system in both configurations. We gradually increased the number of clients until the throughput reached a plateau, to capture the maximum capacity of that configuration: for the flat configuration the max-throughput was achieved by using 5 to 60 clients depending on the number of servers; for the hierarchical configuration the number of clients ranged from 30 to 60. The results are depicted in Fig. 3, showing the difference in percent of both approaches for different numbers of nodes. As it can be observed, the advantage of Fireplug increases significantly with an increasing number of servers up to 900 percent with 12 servers. This is due to the fact that the message complexity of the algorithms grows quadratically with the number of members in each group. We chose to display the difference in performance in percent to highlight the increasing difference between both approaches with an increasing number of servers.

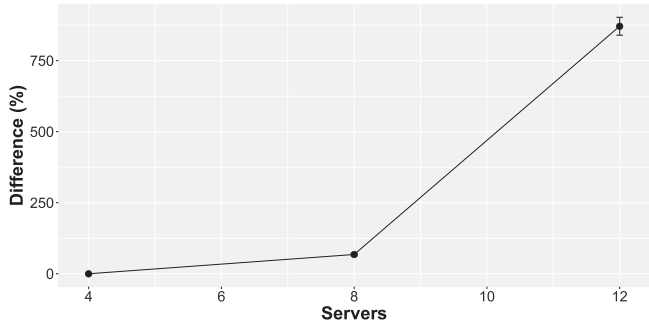


Fig. 3. Comparison between flat and Fireplug's throughput.

5.2 Update Transactions: HB Versus HC

We now compare HB with Hierarchical-Certification Global-Apply. As discussed in Section 4.3, HB is expected to have an advantage over HC in most execution cases since the primaries do not have to calculate extra signatures and also do not have to exchange them before sending them to the local replicas. Nevertheless, HB puts more load on the local (non-primary) replicas than HC, given that with HB local replicas also have to execute and verify the transactions. Following our analysis, the time to execute and verify a transaction takes, on average, more than twice the time it takes to do the verification only. This means that with HB local replicas are subject to approximately twice the load than local replicas running HC (for each write transaction).

From the discussion above, one can reason as follows. If the system is not overloaded, HB is expected to perform better, because it requires a smaller number of communication and processing steps. If the system is overloaded and machines are homogeneous, the performance of both HC and HB will fall. However, it is possible to sustain more load by upgrading only machines in the global cluster and switching from HB to HC, given that HC alleviates the burden on the local replicas. Our experimental results, confirm this analysis.

Fig. 4 shows the performance of HB versus HC in two scenarios: a scenario where all machines have high capacity (homogeneous) and a scenario where only the machines in the global cluster have high capacity, and the local replicas have lower capacity (half the capacity of primaries). With 25 percent writes, the local replicas are not overloaded in any of the scenarios, thus HB performs much better than HC (almost 40 percent better). However, with 50 percent writes, in the scenario where local replicas are less powerful, they can no longer keep-up with the load, thus HC outperforms HB (by almost 20 percent). Finally, with 75 percent writes, even when all machines have high capacity, HC performs better. In this case, the amount of writes saturates even the more powerful machines but, since local replicas have less

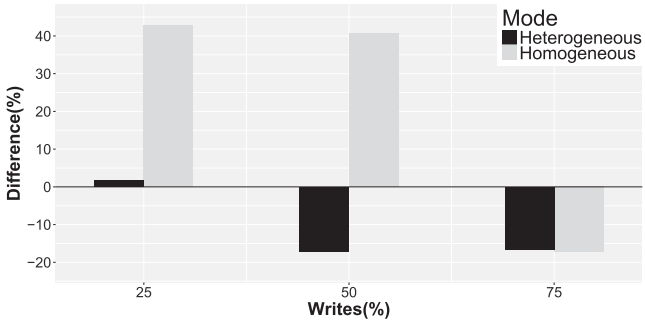


Fig. 4. Comparison between HC and HB's throughput.

work in HC, they have some free cycles to serve read transactions improving the overall throughput. In summary, HB should be used in steady state (we assume the system is provisioned to avoid overload in normal operation) and HC can be used to mitigate the effects of workload peaks. A fully discussion of self-adaptation mechanisms for Fireplug is out of the scope of this paper. We just point out that we also implemented sensors on each replica that monitor the size of the request queues and that allow to detect when nodes are overloaded.

5.3 Impact of Multi-Versioning Features

The graph databases we use do not natively support multi-versioning. However, our middleware enhances these graph databases with the multi-versioning features. We now study the impact of these extensions. On one hand, these features introduce additional complexity in the GRADAM layer, thus they have the potential to slowdown the performance of Fireplug in contention-free scenarios. On the other hand, multi-versioning shields read-only transaction from aborts, so it may offer improvements in scenarios where contention exists. In particular, read transactions can read from a consistent snapshot, defined at the time the transaction starts, even if other transactions update the graph concurrently. This allows read transactions to commit in scenarios where a single-version database would cause them to abort.

We assess this by running our hierarchical framework with a total of 8 replicas with increasing write-load (1-20 percent) in two different settings: With multi-versioning enabled and without multi-versioning. Additionally, we query only 10 percent of the dataset to increase contention.

The results are presented in Fig. 5. It can be seen that, with very little writes multi-versioning adds an overhead of over 20 percent to the system. However, with an increasing

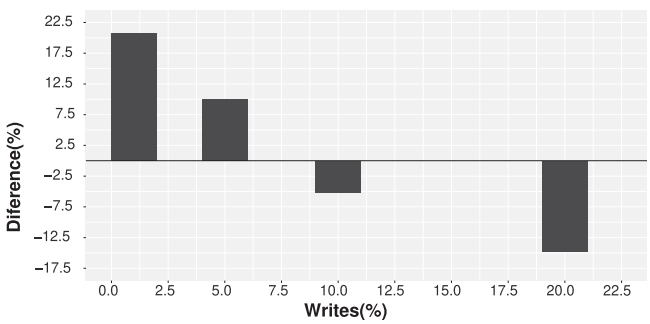


Fig. 5. Single- and multi-versioned database throughput.

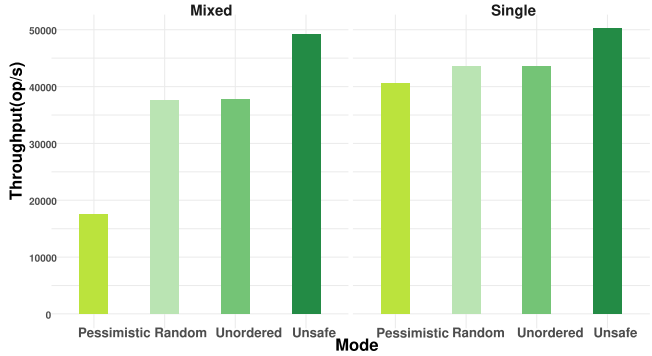


Fig. 6. Throughput for read-modes with 4 servers.

percentage of writes the advantages of multi-versioning start to compensate this overhead. In the case of 10 percent of writes multi-versioning already increases the performance of the system by 5 percent. Increasing the writes 20 percent further increases the advantage of multi-versioning to 15 percent, as expected.

5.4 Read-Only Transactions

In order to show the differences between the proposed read-modes, we performed experiments using both a single group and an hierarchical setup with 4 datacenters, each one with 4 servers.

For the single group scenario, we experimented two different deployment scenarios: a deployment that uses multiple implementations of the database (namely Neo4j and OrientDB) where the average latency among servers was 20 ms (dubbed "mixed/wide-area" setup) and a deployment with a single database (Neo4j) and an average latency among servers of 1ms (dubbed "mixed/local-area" setup). In both cases we executed 100 percent reads. The results of these experiments are depicted in Fig. 6. As expected, the unsafe execution shows the highest throughput of all four alternatives and shows no significant difference in the single or mixed database setup. Also, as expected the pessimistic (ordered) verification shows the worst performance and greatly varies between the single database and mixed database setup. This happens since in the single database setup we only use the fastest database implementations and, therefore, the performance bottleneck is still in the network, while in the mixed setup the performance bottleneck shifts to the slowest database as it has to verify all messages as well. For the same reason the unordered and random access executions show a slightly improved performance in the single database setup. Interestingly, both show almost identical performance, as the unordered execution has the advantage of having only to wait for $f + 1$ equal responses, while the random verification eventually hits a slower database and has to wait for its response.

For the hierarchical scenario with 12 servers, depicted in Fig. 7, we set the latency among members of the local groups to 1 ms and the latency among members of the global group to 20 ms. As before, we used scenarios with a single database implementation (Neo4j) and with multiple implementations (Neo4j and OrientDB). Finally, we subjected both scenarios to a read-only workload and to a read-heavy workload. The results follow the trend observed in the single group case. As expected the unsafe execution

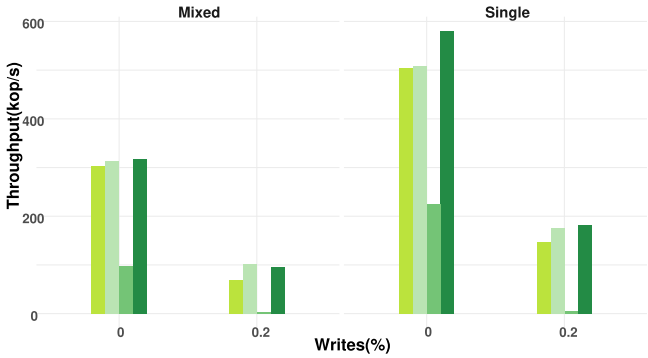


Fig. 7. Throughput for read-modes with 12 servers.

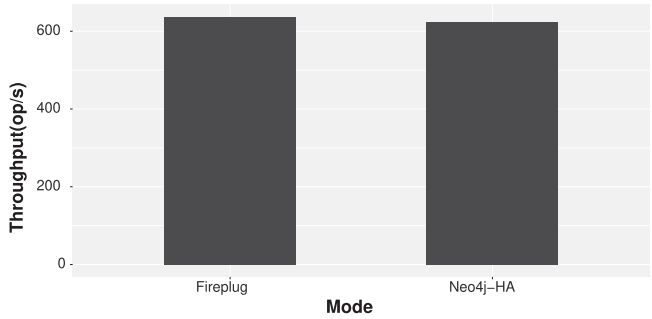


Fig. 8. Neo4j-HA versus Fireplug.

shows a better performance in almost all cases. Only in the mixed setup with 0.2 percent the pessimistic execution shows a small disadvantage since OrientDB is unable to handle as many write requests early in the execution and ends up falling behind significantly. Additionally, as depicted, the overall performance depends heavily on the database setup (single versus mixed) and on the write levels (0 percent versus 0.2 percent). We considered very low write levels since the focus of this experiment was to measure the the performance of the system in a read-dominated setting similar to values observed by Facebook in [44].

The pessimistic (ordered) execution performance drops heavily when introducing writes into the system as expected, since reads are now queued behind the writes which leads to a strong performance toll. Additionally, in this setup the difference between the local and global verification is stronger than in Fig. 6 since in the previous experiment the global cluster received only requests from the one local cluster and not from multiple local clusters, but in the current execution, the global cluster has to handle requests from multiple datacenters which results in a bottleneck.

5.5 Fireplug Against Native Replication

Finally, we compare the performance of Fireplug against the best performing native replication system of all databases we used. In particular we compared a deployed of a single group running 4 replicas of Neo4j (Neo4j-HA) and a similar cluster running an homogeneous deployment of Fireplug. We compared the Byzantine-tolerant version of Fireplug. Since the treatment of write-transactions is the main difference between Neo4j-HA and Fireplug we executed the experiments with 100 percent writes.

Results are depicted in Fig. 8. As shown in the bar graph, Fireplug shows a 2 percent performance advantage over Neo4j considering 100 percent writes even though Fireplug has been configured to tolerate Byzantine failures and Neo4j only tolerates crash faults. Additionally, due to the total order and deferred update replication, Fireplug provides serializability while Neo4j provides only snapshot isolation. This happens since Neo4j relies on a single master replication scheme and has a complicated locking mechanism which results in a lot of transactions running into deadlocks which then have to be aborted. Fireplug avoids this issue by executing the transactions in total order, aborting only when conflicts arise, which has a lower impact on the system performance.

6 Conclusion

We presented Fireplug, a flexible architecture to build robust geo-replicated transactional graph databases. Fireplug combines in a novel way ideas from N-version programming, a hierarchical Byzantine-tolerant state-machine replication, and a deferred update transactional protocol specialized for graph databases to build a cohesive, flexible replicated graph database that can be configured to tolerate different threats. Our hierarchical architecture shows better performance and scalability when compared to more traditional architectures such as flat and single-master. Our evaluation also shows that the optimistic read modes bring significant performance gains by slightly weakening consistency.

Future work includes the integration of self-adjusting mechanisms such as a load-balancer to dynamically adjust the load depending on the instantiated databases. We also plan on studying whether our techniques could be applied under partial replication, a more scalable setting.

Acknowledgments

This article is dedicated to the memory of our dear colleague Prof. Lau Cheuk Lung who motivated us to pursue this work. This work was supported by the FCT via projects PTDC/ EEI-SCR/ 1741/ 2014 (Abyss) and UIDB/ 50021/ 2020; and by the CNPq/Brasil via project 401364/2014-3.

References

[1] M. Newman, "The structure and function of complex networks," *SIAM Rev.*, vol. 45, no. 2, pp. 167–256, 2003.
[2] Neo4j, "The graph foundation for the enterprise," 2016. [Online]. Available: https://neo4j.com/
[3] OrientDB, "Main page," 2016. [Online]. Available: http://orientdb.com/
[4] N. Bronson *et al.*, "TAO: Facebook's distributed data store for the social graph," in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 49–60.
[5] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A comparison of a graph database and a relational database: A data provenance perspective," in *Proc. 48th Annu. Southeast Regional Conf.*, 2010, pp. 42:1–42:6.
[6] Neo4j, "Our customers," 2016. [Online]. Available: https://neo4j.com/customers/
[7] OrientDB, "Customers," 2016. [Online]. Available: http://orientdb.com/customers/
[8] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *IEEE Security Privacy*, vol. 9, no. 3, pp. 49–51, May 2011.
[9] L. Mint, "Beware of hacked ISOs," 2017. [Online]. Available: http://blog.linuxmint.com/?p=2994
[10] S. Brilliant, J. Knight, and N. Leveson, "The consistent comparison problem in N-version software," *IEEE Trans. Softw. Eng.*, vol. 15, no. 11, pp. 1481–1485, Nov. 1989.

[11] White source, "Top open source security vulnerabilities," 2017. [Online]. Available: https://www.whitesourcesoftware.com/whitesource-blog/open-source-security-vulnerability/
[12] D. Sciascia, F. Pedone, and F. Junqueira, "Scalable deferred update replication," in *Proc. 42nd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2012, pp. 1–12.
[13] J. Laprie, "Dependability: A unifying concept for reliable computing and fault-tolerance," in *Proc. Dependability Resilient Comput.*, 1989.
[14] L. Lamport and M. Massa, "Cheap paxos," in *Proc. 34th IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2004, pp. 307–314.
[15] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, Nov. 2002.
[16] A. Bessani, J. Sousa, and E. Alchieri, "State machine replication for the masses with BFT-SMART," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2014, pp. 355–362.
[17] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25, 2001.
[18] P. Marandi, M. Primi, and F. Pedone, "Multi-ring paxos," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2012, pp. 1–12.
[19] Y. Amir *et al.*, "Scaling Byzantine fault-tolerant replication to wide area networks," in *Proc. 36th IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2006, pp. 105–114.
[20] Y. Amir *et al.*,"Steward: Scaling Byzantine fault-tolerant replication to wide area networks," *IEEE Trans. Dependable Secure Comput.*, vol. 7, no. 1, pp. 80–93, Jan. 2010.
[21] J. Baker *et al.*, "Megastore: Providing scalable, highly available storage for interactive services," in *Proc. 5th Biennial Conf. Innovative Data Syst. Res.*, 2011, pp. 223–234.
[22] J. Corbett *et al.*, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, 2013, Art. no. 8.
[23] F. Pedone and N. Schiper, "Byzantine fault-tolerant deferred update replication," *J. Brazilian Comput. Soc.*, vol. 18, no. 1, pp. 3–18, 2012.
[24] A. Shamis *et al.*, "Fast general distributed transactions with opacity," in *Proc. Int. Conf. Manage. Data*, 2019, pp. 433–448.
[25] A. Avizienis, "The N-version approach to fault-tolerant software," *IEEE Trans. Softw. Eng.*, vol. 11, no. 12, pp. 1491–1501, Dec. 1985.
[26] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro, "OS diversity for intrusion tolerance: Myth or reality?" in *Proc. 41st IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2011, pp. 383–394.
[27] I. Gashi, P. Popov, and L. Strigini, "Fault tolerance via diversity for off-the-shelf products: A study with SQL database servers," *IEEE Trans. Dependable Secure Comput.*, vol. 4, no. 4, pp. 280–294, Oct. 2007.
[28] A. Luiz, L. Lung, and M. Correia, "MITRA: Byzantine fault-tolerant middleware for transaction processing on replicated databases," *SIGMOD Rec.*, vol. 43, no. 1, pp. 32–38, May 2014.
[29] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2013.
[30] G. Malewicz *et al.*, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.
[31] Apache, "Apache giraph," 2017. [Online]. Available: http://giraph.apache.org/
[32] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th Symp. Operating Syst. Des. Implementation*, 2012, pp. 17–30.
[33] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proc. Int. Conf. Manage. Data*, 2013, pp. 505–516.
[34] D. Presser, L. C. Lung, and M. Correia, "Greft: Arbitrary fault-tolerant distributed graph processing," in *Proc. 4th IEEE Int. Congress Big Data*, 2015, pp. 452–459.
[35] G. Danezis and D. Hrycyszyn, "Blockmania: From block DAGs to consensus," University College London, London, U.K.:, 2018.
[36] M. Herlihy and E. Koskinen, "Transactional boosting: A methodology for highly-concurrent transactional objects," in *Proc. 13th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2008, pp. 207–216.
[37] N. Herman *et al.*, "Type-aware transactions for faster concurrent code," in *Proc. 11th Eur. Conf. Comput. Syst.*, 2016, Art. no. 31.
[38] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Proc. 13th Int. Conf. Stabilization Saf. Secur. Distrib. Syst.*, 2011, pp. 386–400.
[39] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Proc. 23rd ACM Symp. Operating Syst. Princ.*, 2011, pp. 385–400.
[40] T. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.
[41] G. Bagan, A. Bonifati, R. Ciucanu, G. H. Fletcher, A. Lemay, and N. Advokaat, "gMark: Schema-driven generation of graphs and queries," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 4, pp. 856–869, Apr. 2017.
[42] Grid'5000, "Grid'5000, a scientific instrument designed to support experiment-driven research," 2017. [Online]. Available: https://www.grid5000.fr/
[43] Netem, "Network emulator," 2019. [Online]. Available: http://www.linuxfoundation.org/collaborate/workgroups/networking/netem
[44] Facebook, "TAO: The power of the graph," 2016. [Online]. Available: https://www.facebook.com/notes/facebook-engineering/tao-the-power-of-the-graph/10151525983993920/

Ray Neiheiser received the master's degree in computer science from the Federal University of Santa Catarina (UFSC), Brazil, and he is currently working toward the PhD degree at the Federal University of Santa Catarina (UFSC), Brazil. His area of interest is the construction of reliable distributed systems. In the context of this, he specifically focuses on Byzantine fault-tolerant consensus and distributed ledger technology.

Luciana Rech received the graduate degree from the University of Cruz Alta, Brazil in computer science, the master's degree in computer science from the Federal University of Santa Catarina, Brazil, and the PhD degree in electrical engineering from the Federal University of Santa Catarina, Brazil. She is an associate professor with the Informatics and Statistics Department (INE) of the Federal University of Santa Catarina (UFSC), Brazil, and a member of the Distributed Systems Research Laboratory (LAPESD).

Manuel Bravo received the double PhD degree in computer science from the Universidade de Lisboa, Portugal, and the Université Catholique de Louvain, Belgium. He is a postdoctoral researcher with the IMDEA Software Institute in Madrid, Spain. His interest is in the design and implementation of distributed systems. Specifically, he is interested in understanding replication and consistency in such systems.

Luís Rodrigues (Senior Member, IEEE) is a professor at Instituto Superior Técnico, Universidade de Lisboa and a researcher at INESC-ID. His research interests lie in the area of reliable distributed systems. He is co-author of more than 150 papers and three textbooks on these topics. He is a senior member of the ACM.

Miguel Correia (Senior Member, IEEE) is is associate professor with habilitation at Instituto Superior Técnico, Universidade de Lisboa, and senior researcher at INESC-ID. He has more than 150 publications. His research focuses on cybersecurity and dependability (aka fault tolerance), typically in distributed systems, in the context of different applications (blockchain, cloud, and mobile).

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Evaluation of Stream Processing Frameworks

Giselle van Dongen, *Member, IEEE* and Dirk Van den Poel, *Senior Member, IEEE*

Abstract—The increasing need for real-time insights in data sparked the development of multiple stream processing frameworks. Several benchmarking studies were conducted in an effort to form guidelines for identifying the most appropriate framework for a use case. In this article, we extend this research and present the results gathered. In addition to Spark Streaming and Flink, we also include the emerging frameworks Structured Streaming and Kafka Streams. We define four workloads with custom parameter tuning. Each of these is optimized for a certain metric or for measuring performance under specific scenarios such as bursty workloads. We analyze the relationship between latency, throughput, and resource consumption and we measure the performance impact of adding different common operations to the pipeline. To ensure correct latency measurements, we use a single Kafka broker. Our results show that the latency disadvantages of using a micro-batch system are most apparent for stateless operations. With more complex pipelines, customized implementations can give event-driven frameworks a large latency advantage. Due to its micro-batch architecture, Structured Streaming can handle very high throughput at the cost of high latency. Under tight latency SLAs, Flink sustains the highest throughput. Additionally, Flink shows the least performance degradation when confronted with periodic bursts of data. When a burst of data needs to be processed right after startup, however, micro-batch systems catch up faster while event-driven systems output the first events sooner.

Index Terms—Apache spark, structured streaming, apache flink, apache kafka, kafka streams, distributed computing, stream processing frameworks, benchmarking, big data

1 Introduction

In response to the increasing need for fast, reliable and accurate answers to data questions, many stream processing frameworks were developed. Each use case, however, has its own performance requirements and data characteristics. Several benchmarks were conducted in an effort to shed light on which frameworks perform best in which scenario, e.g., [1], [2], [3], [4], [5]. Previous work often benchmarked Flink and Spark Streaming. We extend this work by comparing later releases of these frameworks with two emerging frameworks being Structured Streaming, which has shown promising results [6], and Kafka Streams, which has been adopted by large IT companies such as Zalando and Pinterest. We take a two-pronged approach of comparing stream processing frameworks on different workloads and scenarios. In the first place, we assess the performance of different frameworks on similar operations. We look at relationships between metrics such as latency, throughput and resource utilization for different processing scenarios. The parameter configurations of a framework have a large influence on the performance in a certain processing scenario. Therefore, we tune parameters per workload. Additionally, we evaluate the performance impact of adding different, complex operations to the workflow of each framework. Understanding the impact of an operation on the performance of a pipeline, is a valuable insight in the design phase of the processing flow. By combining these two approaches, we can form a more nuanced evaluation of which framework is most suitable for which use case. In short, we make the following contributions:

1) Open-source benchmark and extensive analysis of emerging frameworks, Structured Streaming and Kafka Streams, besides newer releases of Flink and Spark Streaming.
2) Accurate latency measurement of pipelines of different complexities for all frameworks by capturing time on a single Kafka broker.
3) Thorough analysis of relationships between latency, throughput and resource consumption for each of the frameworks and under different processing pipelines, workloads and throughput levels.
4) Dedicated parameter tuning per workload for more nuanced performance evaluations with detailed reasoning behind each of the settings.
5) Inclusion of built-in and customized stateful operators.
6) Guidelines for choosing the right tool for a use case by taking into account inter-framework differences, different pipeline complexities, implementation flexibility and data characteristics.

The code for this benchmark can be found at https://github.com/Klarrio/open-stream-processing-benchmark. The rest of this paper is organized as follows. The next section gives an overview of the related work that formed a basis for this study. In Section 3 we describe the setup that was used to conduct this benchmark. Section 4 dives deeper into the configurations used for the different frameworks. A discussion of the results follows in Section 5, followed by general conclusions. Finally, we outline some of the limitations and opportunities for further research. Additional information on how this benchmark was conducted can be found in the Supplemental File, which can be found on the Computer

• *The authors are with Ghent University, 9000 Ghent, Belgium. E-mail: {Giselle.vanDongen, Dirk.VandenPoel}@ugent.be.*

Manuscript received 16 Aug. 2019; revised 10 Jan. 2020; accepted 18 Feb. 2020. Date of publication 5 Mar. 2020; date of current version 30 Mar. 2020. (Corresponding author: Giselle van Dongen.) Recommended for acceptance by Henri E. Bal. Digital Object Identifier no. 10.1109/TPDS.2020.2978480

1045-9219 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2020.2978480.

2 Related Work

Several benchmarking studies have been conducted over the last years. In this section, we outline the most important differences. A tabular overview has been given in Section 1 of the Supplemental File, available online. Most of these benchmarks were implemented on one or a combination of the following frameworks: Spark Streaming, Storm and Flink. In [4] and [5], Spark Streaming outperforms Storm in peak throughput measurements and resource utilization but Storm attains lower latencies. The Yahoo benchmark [3] confirmed that under high throughput Storm struggles. Karakaya *et al.* [2] extended the Yahoo Benchmark and found that Flink 0.10.1 outperforms Spark Streaming 1.5.0 and Storm 0.10.0 for data-intensive applications. However, in applications where high latency is acceptable, Spark outperforms Flink. Karimov *et al.* [1] confirmed that the general performance of Spark Streaming 2.0.1 and Flink 1.1.3 is better than that of Storm. In this work, we include the latest versions of Spark Streaming 2.4.1 and Flink 1.9.1 due to their superior results. Additionally, we include Spark's new Structured Streaming 2.4.1 API and Kafka Streams 2.1.0, which have not been thoroughly benchmarked before. Micro-batch systems as well as event-driven frameworks are represented in this benchmark. Both Spark frameworks are micro-batch systems [7], meaning that they buffer events on the receiver-side and process them in small batches. Kafka Streams and Flink work event-driven and only apply buffering when events need to be send over the network.

Previous work mainly focused on benchmarking end-to-end applications in areas such as ETL, descriptive statistics and IO, e.g., [2], [3], [8], [9]. Some work has been conducted on benchmarking single operations such as parsing [8], filtering [4], [8], [9] and windowed aggregations [1]. The metrics for these single operations were, however, recorded as if it were end-to-end applications by including ingesting, buffering, networking, serialization, etc. In this work, we also measure the impact of adding a certain operation to a pipeline. We ensured that each of these operations constitutes a seperate stage in the DAG of the job. We based the design of our pipeline on the Yahoo benchmark [3], which has also been adopted by [2]. Besides a tumbling window, we also include a sliding window for the latency measurement workload. For these types of stateful transformations, some frameworks offer more flexibility via low-level APIs, custom triggers and operator access to managed state. In this work we implemented the tumbling and sliding window with built-in as well as low-level customized operators for Flink, Kafka Streams and Structured Streaming to investigate the advantages of using this flexibility.

Two of the most important performance indicators in stream processing systems are latency and throughput, which have been included in most previous benchmarks, e.g., [1], [3], [8], [9]. We use the same methodology to measure latency of single operations as in the initial proposal for this study [10]. The ability of a framework to process events faster and in bigger quantities with a similar setup, leads to cost reductions and a greater ability to overcome bursts of data. Most research, however, merely analyses the maximum throughput that can be sustained for a prolonged period of time, e.g., [1]. In this benchmark, we analyze throughput in two additional scenarios. First, we investigate the behavior of the framework under a bursty workload. Second, we monitor the behavior of the framework when it needs to process a big burst of data right after startup. Some papers already studied the behavior of stream processing frameworks under bursty data. Shukla and Simmhan [8], and later RIoTBench [9], studied the behavior of Storm under bi-modal data rates with peaks in mornings and evenings, while Karimov *et al.* [1] studied the effects of a fluctuating workload for Spark, Flink and Storm. Spark and Flink showed similar behavior in the case of windowed aggregations, while Flink handled windowed joins better than Spark. Storm was the most susceptible to bottlenecks due to data bursts. Most other literature studied constant rate workloads under different data scales, e.g., [2], [3], [4], [5]. To be able to generate these different loads, we use temporal and spatial scaling similar to [8] and [9].

StreamBench [4] and Qian *et al.* [5] studied the effects of a failing node on throughput and latency. Their results indicated no significant impact for Spark Streaming while Storm experienced a large performance decrease. In this work, in contrast, we investigate the behavior of processing a big burst of data at startup, mimicking the behavior triggered when the job would be down for a period of time.

This benchmark uses Apache Kafka [11] as distributed publish-subscribe messaging system to decouple input generators from processors. In the Kafka cluster, streams of data are made available on partitioned topics. The majority of past benchmarking literature uses Apache Kafka for this purpose as this is representative of real-world usage, e.g., [2], [3], [8], [9], [10]. Furthermore, frameworks, e.g., [6], [12], require a replayable, durable data source such as Kafka to give exactly-once processing guarantees.

Finally, the configuration parameters of a framework have large effects on latency and throughput. Previous research often did not sufficiently document the configuration settings or used default values for all workloads. In this work, framework parameters are tuned separately for each workload to get a more accurate view of the performance of a framework on a certain metric or in a certain scenario. A thorough discussion follows in Section 4, supported by extra material in the Supplemental File, available online.

3 Benchmark Design

In this section, we describe the benchmark setup. First, we introduce the input data source. Next, we elaborate on the operations that were included and the metrics that are monitored. Finally, we describe the different workloads and the architecture on which the processing jobs are run.

3.1 Data

We perform the benchmark on data from the IoT domain because of its increasing popularity and its increasing amount of use cases. We use traffic sensor data from the Netherlands provided by NDW (Nationale Databank Wegverkeersgegevens). This data contains measurements of the number of cars (flow) and their average speed at around

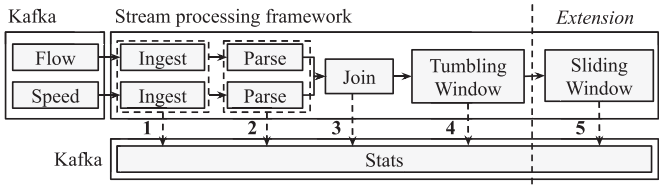


Fig. 1. Processing flow based on [10].

60,000 measurement points in the Netherlands and this for every minute of every lane of the road. The data publisher publishes a subset of this data set as a constant rate data stream on the Kafka input topics. For periodic burst workloads, every ten seconds an enlarged batch is published. The volume of data can be inflated by a configurable factor by using spatial scaling, similar to [9]. By changing the characteristics of the data, we mimic different datasets, thereby generalizing the benchmark to other use cases.

3.2 Processing Pipeline

A stream processing benchmark should be able to test the impact of different types of operations. Therefore, we run each workload for different pipeline complexities. We do this by using an extensible processing pipeline (Fig. 1) with operations similar to [3] and [10].

1) Ingest: Reading data from the Kafka flow and speed topics: no transformations are done on the data.
2) Parse: Parsing the JSON flow and speed messages.
3) Stream-stream join: inner join of flow and speed messages with the same timestamp, measurement point and lane.
4) Tumbling window: Adding up the number of cars that passed by in the last second and averaging their speed for all the lanes of the road.
5) Extension: Sliding window: Only executed for the latency measurement workload. Computes the relative change in flow and speed for each measurement location over the last two and three seconds. The window length is three seconds and the slide interval is one second.

Initially, we execute the pipeline up to and including the ingest stage. We use the performance of this stage as a baseline. In the second run, we add the parse stage to the pipeline. This mimics ETL jobs with simple data transformations. Afterwards we add a join operation, common in data enrichment scenarios. Finally, we add window stages and customized stateful operations for testing more complex analytics capabilities. Intermediate stages do not publish their outputs to Kafka.

All stateful operations (joining and windowing) are done on event time. For this, we use the Kafka log append timestamp of the input observation. This is possible since the input stream is never out of order. Spark Streaming does not offer event time processing characteristics, therefore some extra logic is required to handle windows accurately in the case of bursty data.

Event-driven frameworks do not apply buffering on the receiver side and can therefore, have significant latency advantages. This advantage disappears when built-in stateful operators such as tumbling windows are used. Since a flexible API is an important asset when optimizing pipeline performance, we included multiple implementations of the stateful operators. For the joining stage of the processing pipeline, we use the most appropriate join semantic available in the framework. For Flink and Kafka Streams, we use an inner interval join [13]. This type of join permits joining events with timestamps that lie in a relative time interval to each other, which is precisely what we want to do. This type of join has a much lower latency than the tumbling window implementation that is typical for micro-batch systems since it can output events directly after receiving the entire pair for the join and it doesn't have to wait for the tumbling of the window.

The Flink API offers flexibility for stateful operations such as the ability to use different state backends [14], to define custom window triggers [12] or to use low-level as well as high-level APIs to do stateful operations [15]. We implement the tumbling window stage with the default event time trigger as well as with a custom trigger that triggers computation when enough data has arrived to do the computation. Furthermore, the sliding window stage was implemented with a built-in sliding window, as well as with the low-level processor API which gives direct access to managed keyed state and timers [15]. Instead of using a window buffer to compute the change in speed and flow over the last seconds, we manage our own list state to do the computation and generate output as soon as an observation enters the processor function. This has the effect that each event can be processed instantly.

Kafka Streams offers two levels of APIs: a high-level Domain Specific Language (DSL) and a low-level Streams Processor API. In this benchmark, we implemented the pipeline with the Kafka Streams DSL for the stateless stages. For the stateful stages we studied two implementations: one using the DSL windowing functionality and the other using the low-level processor API. The processor API allows us to interact with state stores and build customized processing logic. To implement the aggregation and relative change phases, we call the processor API from the DSL by supplying a transformer as described in the documentation [16]. For both stages, we use a persistent key value store backed by RocksDB to store state. RocksDB is the default state backend in Kafka Streams. By using customized low-level implementations we can make sure events are sent out as soon as possible, reaching lower latencies.

Finally, we also include an alternative implementation for Structured Streaming. When we use built-in aggregations such as tumbling window, we experienced issues with the propagation of watermarks leading to very high latencies and making it impossible to chain aggregations, as discussed in Section 5.1. Therefore, we also implemented these stages with mapping functions with custom state management. In this approach, we have fine-grained control over state and the publishing of output so we do not rely on the propagation of watermarks.

The operations discussed in this section cover the main building blocks of stream processing pipelines for most use cases, as can be inferred from the documentation of these frameworks [14], [16], [17], [18]: basic operations, joining, windowing and processing functions. We believe that by including these operations, we can provide a general performance assessment of these frameworks.

3.3 Metrics

We collect four metrics for every run: the latency of each message, the throughput per second and the memory and CPU utilization of the workers. For each run, except for the single burst workload, we ignore the first five minutes of data since these initial minutes often show increased latency and CPU due to the start up.

3.3.1 Latency

Latency is the amount of time the processing framework requires to generate an output message from one or multiple input messages. This can be difficult to measure since distributed systems do not have a global notion of time. The approach we follow to counter this is described in Section 3.4.1. We unify the way latency is measured across frameworks by subtracting the Kafka log append timestamps of the output and input message. We do not rely on the internal metrics of the frameworks since this does not guarantee us a uniform definition of latency. Furthermore, our definition of latency includes the time the event resided on Kafka before the framework started processing the observation. When backpressure kicks in, this difference between event time latency and processing-time latency can become significant [1]. In the case where multiple input messages are required for one output message, we use the timestamp of the latest input message that was required to do the computation, as proposed by [1].

3.3.2 Throughput

We define throughput as the number of messages a framework processes per second. We measure throughput in three scenarios: peak sustainable throughput under constant load, burst throughput at startup and throughput under periodic bursts. Peak sustainable throughput expresses the maximum throughput that a framework can sustain over an extended period of time without becoming unstable. We consider a system to be stable when the latency and queue size on the input buffer do not continuously increase [1] and when average CPU levels are within reasonable bounds, i.e., lower than 80 percent. Since this benchmark focuses on (near) real-time processing, we do not allow median latencies to go above 10 seconds. In addition, we measure the burst throughput at startup to capture the speed at which a framework can make up for an incurred lag. Finally, we compare input and output throughput for workloads with periodic bursts.

3.3.3 Memory Utilization

We collect the heap memory usage of the JVMs that run the processing jobs by scraping the JMX metrics exposed by the jobs. The effect on CPU usage of enabling JMX is negligible. Heap memory is used, by all frameworks, to store state and do computations. However, memory management is intrinsically different for each framework, as will be evident from the results.

3.3.4 CPU Utilization

Finally, we look at CPU utilization or the ability to evenly spread work across the different slaves as well as the processing power required to do specific operations. We consider average CPU levels above 80 percent as unhealthy due to limited resources that are available to the JVM to maintain the running processes when unforeseen bursts of data occur. We monitor CPU at the container level using cAdvisor [19].

3.4 Workloads

Stream processing frameworks need to be able to handle different data characteristics and processing scenarios: bursty workloads, constant rate workloads with different throughput levels, catching up after downtime. We define specific workloads for each of these scenarios. Furthermore, we define separate optimized workloads for measuring latency and throughput since they influence each other, e.g., via buffer sizes and timeouts [12].

3.4.1 Workload for Latency Measurement

The first workload is the one designed for measuring latency in optimal circumstances. Latency is measured by subtracting the Kafka log append time of the output and input message. Kafka is a distributed service and these timestamps are appended by the leading broker of the topic partition when the message arrives. The leading brokers for the input and output partitions can reside on different machines. Time-of-day clocks can demonstrate noticeable time differences across machines. This makes computations with timestamps from different brokers inaccurate. In an experiment with five brokers, we clocked negative latencies of up to 55 ms for up to ten percent of the events for the earlier stages of the pipeline. Network Time Protocol (NTP) can be used to synchronize time-of-day clocks but still has a minimum error of 35 ms [20]. Due to this, time-of-day clocks are not suitable to measure latencies of less than 10 ms. To counter this, we use a single Kafka broker and we compare the timestamps that were attached to the input and output message by that single Kafka broker [10]. This ensures that all time recordings are done on the same machine and therefore, based on the same clock. Log append time, i.e., the wall clock time of the broker, is accurate up to a millisecond level, which is sufficient for our measurements [21].

This workload is used to determine the lowest obtainable latency. To eliminate the influence throughput has on latency, we use a very low throughput to prevent stressing the framework and the single Kafka broker. In the next workload we gradually increase the load to see how latency holds up as throughput grows. Data flows in at a constant rate of 400 messages per second throughout the entire run. This equals a volume of around 135 MB in one run of thirty minutes. We run this workload for different pipeline complexities to be able to see its impact on latency and we add an additional sliding window stage to the pipeline.

3.4.2 Workload for Sustainable Throughput Measurement

Whereas the previous workload focuses on determining the lowest obtainable latency, this workload investigates how this latency evolves when the framework is used for more realistic and considerable loads and to determine the peak sustainable throughput. Additionally, we use this workload to investigate the behavior of the frameworks under a constant data input rate which is common for use cases such as

monitoring systems (e.g., application logs), preventive maintenance workloads or connected devices (e.g., solar panels).

We determine the peak sustainable throughput by running the workload repeatedly for increasing data scales and monitoring latency, throughput and CPU. For latency, we check whether the median latency does not increase monotonically throughout the run and remains below 10 seconds. When the volume of data is higher than the framework can process, latency increases constantly throughout the workload due to processing delays. Furthermore, we also check whether the framework has processed all events by the time the stream ends. In this work, we assume that once the input stream halts, the framework needs to be able to finish processing in less than ten seconds, otherwise it was either lagging behind on the input data stream or batching at intervals higher than 10 seconds. Finally, we monitor whether the average CPU utilization of the framework containers does not exceed 80 percent.

3.4.3 Workload With Burst at Startup

The third workload measures the peak burst throughput that a framework can handle right after startup. This mimics the effect of a job trying to catch up with an incurred delay. For this, we start up the data publisher before the job has been started and let it publish around 6,000 events per second, which is a throughput level that all frameworks are able to handle but still imposes a considerable load. Five minutes later, we bring up the processing job and start processing the data on the Kafka topics from the earliest offset. This equals around 1 800 000 events or 350 MB, with linearly increasing event times. Once the processing job is running, we reduce the throughput to around 400 messages per second to let event time progress but not put any more load on the framework. We let this processing job run for ten minutes. Afterwards, we evaluate the throughput at which the framework was able to process the initial burst of data.

3.4.4 Workload With Periodic Bursts

Finally, we want to monitor the ability of the framework to overcome bursts in the input stream. This mimics use cases such as processing data from a group of coordinated sensors or from connected devices that upload bursts of data every few hours or use cases which need to be able to sustain peaks of data such as web log processing. We have a constant stream of a very low volume and every ten seconds there is a large burst of data. Each burst of data contains approximately 38 000 events, which is approximately 7.5 MB. The publishing of one burst takes around 170 to 180 ms. We look at how long latency persists at an inflated level after a large burst and at the effects on CPU and memory utilization.

3.5 Architecture

To make our benchmark architecture mimic a real-world production analytics stack, we use AWS EC2. We set up a CloudFormation stack which runs DC/OS on nine m5.2xlarge instances (one master, nine slaves). Each of these instances has 8 vCPU, 32 GB RAM, a network bandwith of up to 10 Gbps and a dedicated EBS bandwidth of up to 3500 Mbps. Each instance has an EBS volume attached to it with 150 GB of disk space. We use DC/OS as an abstraction layer between the benchmark components and the EC2 instances on which they run. All benchmark components run in Docker containers on DC/OS. To mitigate the potential performance differences between EC2 instances, we run each benchmark run five times on different clusters and select the run with median performance.

As a message broker, we use Apache Kafka. For most runs, the cluster consists of five brokers (2 vCPU, 6 GB RAM) which is the same number as the number of workers for each of the frameworks. As described in Section 3.4.1, we use a single Kafka broker (6.5 vCPU, 24 GB RAM) for the latency measurement workload. We use Kafka Manager for managing the cluster and creating the input and output topics, each of these with 20 partitions since each framework cluster has 20 processing cores. Messages are not replicated to avoid creating too much load on the Kafka brokers. The input topics for the speed and flow messages are partitioned on the ID since this is the key for joining and aggregating. Additionally, all brokers are configured to use LogAppendTime for latency measurements. The roundtrip latency of the Kafka cluster is around 1 ms, tested by publishing and directly consuming the message again. We use cAdvisor, backed by InfluxDB, to monitor and store CPU and networking metrics of the framework containers. Concurrently, the processing jobs expose heap usage metrics on an endpoint via JMX, which is then scraped and stored by a JMX collector. An architecture diagram and detailed information on the distribution of resources over all services has been listed in Section 3 of the Supplemental File, available online. Without the framework clusters, the DC/OS cluster has 45 percent CPU allocation and 33 percent memory allocation. The framework cluster adds another 28 percent CPU allocation and 35 percent memory allocation.

4 Frameworks

In this section, we discuss each of the frameworks and their configurations, as listed in Table 1. We only discuss the configuration parameters for which we do not use the default values. Parameter tuning is done per workload to get a more accurate measurement of performance. Configurations were chosen based on the documentation of the frameworks, expert advice, and an empirical study, which can be further consulted in the Supplemental File in Section 4, available online.

Some settings are equal for all frameworks. For the frameworks that use a master-slave architecture, i.e., Flink and both Spark frameworks, we deploy a standalone cluster with one master and five workers in Docker containers. Kafka Streams runs with five instances. Each framework gets the same amount of resources for their master (2 vCPU, 8 GB RAM) and slaves (4 vCPU, 20 GB RAM each). Parallelism is set to 20 since we have 20 Kafka partitions and 20 cores per framework.

For event-driven frameworks we use event time as time characteristic. These frameworks use watermarks to handle out-of-order data [12]. We choose an out-of-order bound of 50 ms since we do not have out-of-order data that needs to be handled but we do need to take into account the possible time desynchronization between the brokers. Assume that, in the case of a constant rate of data, two events that need to

TABLE 1
Framework Configuration Parameters

a. Apache Flink (v1.9.1)

Parameter	Value	Default
JobManager count	1	/
TaskManager count	5	/
JobManager CPU	2	/
JobManager heap / memory	8 GB / 8 GB	/
TaskManager CPU	4	/
TaskManager heap / memory	18 GB / 20 GB	/
Number of task slots	4	1
Default parallelism	20	1
Time characteristic	event time	processing time
State backend	FileSystem	InMemory
Buffer timeout	100 ms (L: 0 ms)	100 ms
Checkpoint interval	10 000 ms	None
Watermark interval	50 ms	/
Out-of-orderness bound	50 ms	/
Object reuse	enabled	disabled

b. Apache Kafka Streams (v2.1.0)

Parameter	Value	Default
Instances count	5	/
Number of threads / CPUs	4	/
Java heap / memory	18 GB / 20 GB	/
Kafka topic parallelism	20	1
Commit interval	1000 ms	30 000 ms
Message timestamp type	LogAppendTime	CreateTime
Linger ms	100 (L: 0)	100
Grace period		
join & tumbling window	5s (PB, ST: 30s)	/
sliding window	50 ms	/
Retention time	interval + grace	1 day
Max task idle ms	0 (SB: 300 000)	0
Producer compression	lz4 (L: none)	none
Producer batch size	200 KB (L: 16 KB)	16 KB

c. Spark Streaming and Structured Streaming (v2.4.1)

Parameter	Value	Default
Master count	1	/
Worker count	5	/
Master CPUs / memory	2 / 8 GB	/
Worker CPUs / memory	4 / 20 GB	/
Driver cores / heap	2 / 6 GB	/
Executor cores / heap	4 / 17 GB	/
Number of executors	5	/
Default parallelism	20	parallelism of parent or cores
SQL shuffle partitions	5 (str.), 20 (sp.)	200
Serializer	kryo	java
Locality wait	100 ms	3 s
Garbage collector	G1GC	Parallel GC
Initiating heap occupancy	35%	45%
Parallel GC threads	4	/
Concurrent GC threads	2	/
Max GC pause ms	200	200
Micro-batch interval		
Initial stages	200 ms	/
Analytics stages	1 s (ST: 3 s, 5 s)	/
Trigger interval (str.)	0	/
Block interval ms	50 (ST: 150, 250 sp.)	200
Watermark ms (str.)	50	/
minBatchesToRetain (str.)	2	100

((str.) refers to Structured Streaming; (sp.) refers to Spark Streaming) Specific workload parameters are noted by L (latency workload), ST (sustainable throughput workload), SB (single burst workload) and PB (periodic burst workload).

be joined or aggregated arrive at the extreme beginning and end of the interval. If not all brokers have synchronized time and the out-of-orderness bound is set to zero, it is possible that the watermark had already progressed past the log append time of the second event by the time it entered the system. This might happen if one broker is lagging 50 ms behind on another broker in system time. In this case the join will not take place for this pair of events since it will be seen as 50 ms out of order. We assume that the time difference between the brokers is not more than 50 ms and therefore, we choose an out-of-orderness bound of 50 ms. We do not use a watermark larger than 50 ms because this inherently increases the latency [12], [13].

Event-driven frameworks buffer events before they are sent over the network to reduce the load. Buffers are flushed when they are full or when a configurable timeout has passed. This timeout usually has a default value of 100 ms, which is what we use for throughput measurements. For the latency measurement workload, we disable buffering by setting the buffer timeout of Flink and the linger time of Kafka Streams to 0 ms, thereby optimizing for latency. Structured Streaming and Spark Streaming do micro-batching and, therefore, intrinsically buffer events. A more thorough explanation of this is given in the Supplemental File in Sections 4.1.1 and 4.2.1, available online.

4.1 Apache Flink

For Apache Flink [12], the configuration settings are listed in Table 1(a). The five task managers each have 4 CPUs and therefore, four task slots as suggested by [14]. Additionally, they get 20 GB memory of which 18 GB is assigned to the heap and the other 2 GB is left for off-heap allocation to network buffers and the managed memory of the task manager [14].

Three state backends are currently available in Flink: MemoryStateBackend, FileSystemBackend en RocksDBBackend. We use the FileSystemBackend since this is the recommended backend for large state that fits in heap memory. MemoryStateBackend is used for jobs with little state and in development and debugging stages. RocksDB is a state store kept off-heap and is recommended if the state does not fit in the heap memory of the task managers [14]. The FileSystemBackend we use is backed by HDFS and checkpointing is done every ten seconds. Due to Flink's asynchronous checkpointing mechanism, the processing pipeline is not blocked while checkpointing. Flink's mechanism of keeping state reduces the load on the garbage collector, as opposed to the mechanism used by Spark which still heavily relies on JVM memory management, as described in Section 4.4.

We define a watermark interval of 50 ms. This means the current watermark is recomputed every 50 ms. Finally, we enable object reuse since this can significantly increase performance.

4.2 Apache Kafka Streams

In 2016, Apache Kafka released Kafka Streams [13], a client library for processing data residing on a Kafka cluster. Kafka Streams does not make use of a master-slave architecture and does not require a cluster but runs with different threads that rely on the Kafka cluster for data parallelism, coordination and fault tolerance. All Kafka Streams instances share the same consumer group and application id such that they all process a part of the input topic partitions.

Each instance will be running on four threads to optimally use all resources. Kafka Streams stores state on Kafka topics and therefore, does not require a HDFS cluster.

When using the DSL, we need to set the grace period and retention times for the window stages. Since each input record leads to an output update record [13] and only the complete output records are kept, the grace period does not introduce additional latency and has been put at a high number of 5s. The retention time is a lower-level property that sets the time events should be retained, e.g., for interactive queries, and has no influence on the latency. We set this to be equal to the slide interval plus grace period.

We do not use message compression for latency measurements. For measuring throughput and resource consumption, we use slightly different parameters, as recommended by [22]. We keep the linger time at the default 100 ms. We use lz4 message compression to reduce the message sizes and load on the network, thereby increasing throughput. We also increase the size of output batches to reduce the network load.

For processing a single burst, we set the maximum task idle time to 300 000 ms. By doing this, Kafka Streams waits an increased amount of time to buffer incoming events on all topics before it starts processing. This prevents the system from running ahead on some topics and subsequently discarding data on other topics if their event time is too far behind.

4.3 Apache Spark: Spark Streaming

Apache Spark [7] is a cluster computing platform that consists of an extended batch API in combination with a library for micro-batch stream processing called Spark Streaming. The job runs with one driver and five executors. On each worker, we allocate 1 GB of the 20 GB to JVM overhead. The executors use 10 percent of the remaining memory for off-heap storage, which leaves 17 GB for heap memory. For the driver we allocate 6 GB of heap memory. Checkpoints are stored in HDFS.

When executing the initial stages of the pipeline, i.e., ingesting and parsing, the jobs are configured to have a micro-batch interval of 200 ms. In the initial stages, the selectivity of the data is one on one so a lower batch interval leads to lower latencies. For the analytics stages we set the micro-batch interval to the same length as the interval at which sensors send their data and on which we will do our aggregations, which is one second. For the sustainable throughput workload, we set the batch interval to three and five seconds since this increases the peak throughput [7]. Spark Streaming splits incoming data in partitions based on the block interval. Therefore, it is recommended to choose the block interval to be equal to the batch interval divided by the desired parallelism [17].

For reading from Kafka we use the direct stream approach with PreferConsistent as location strategy, meaning the 20 Kafka partitions will be spread evenly across available executors [17]. Often it is recommended to use three times the number of cores as the number of partitions. This, however, only leads to performance improvement in the case of data skew. When keys are not equally distributed, having more partitions than the number of cores allows more flexibility in spreading work over all executors. In this use case, the data is equally spread over all keys. Hence, using the same number of partitions as the number of cores leads to less overhead and better performance.

In all workloads, Spark Streaming checkpoints at the default interval which is a multiple of the batch interval that is at least 10 seconds [17]. We use the kryo serializer for serialization and register all classes since it is faster and more compact than the default java serializer.

4.4 Apache Spark: Structured Streaming

In 2016, Apache Spark released a new stream processing API called Structured Streaming [6] which enables users to program streaming applications in the DataFrame API. The DataFrame API is the dominant batch API, and Structured Streaming brings Apache Spark one step closer to unifying their batch and streaming APIs. Structured Streaming offers a micro-batch execution mode and a continuous processing mode for event-driven processing. We use the micro-batch approach because the continuous processing mode is still experimental and does not support stateful operations at the time of this writing. Additionally, the micro-batch API does not support chaining of built-in aggregations, therefore, we do not execute the sliding window stage. However, we also include an implementation of the final two stages using map functions with custom state management, since this circumvents the issues with built-in aggregations.

For Structured Streaming, we trigger job queries as fast as possible, which is enabled by setting the trigger interval to 0 ms. Queries run in the default append mode, which means only new rows of the streaming DataFrame will be written to the sink. Due to the architectural design of Structured Streaming, checkpointing is done for each batch [6]. This leads to a very high load on HDFS. To be able to run the jobs at full capacity, a significant size increase of the HDFS cluster was necessary since the checkpointing for every micro-batch led to seconds latency increase for the windowing phases.

The default parallelism parameter arranges parallelism on the RDD level. Structured Streaming, however, makes use of DataFrames. The parallelism of Dataframes after shuffling is set by the SQL shuffle partitions parameter. Since all workloads have small state and since opening connections for storing state is expensive, performance improves significantly when the number of SQL shuffle partitions is reduced to 5. When doing this for Spark Streaming, we did not notice a similar performance improvement. Structured Streaming checkpoints the range of offsets processed, after each trigger. In contrast, the default checkpoint interval of Spark Streaming is never less than 10 seconds. Due to the heavier use of checkpointing, Structured Streaming benefits more from reducing the amount of SQL shuffle partitions. We use kryo for fast and compact serialization. We use the G1 garbage collector (GC) with the parameters listed in Table 1 to reduce GC pauses. A more thorough explanation of the reasoning behind this can be found in the Supplemental File, available online. Finally, we set the minimum number of batches that has to be retained to two, which further reduces memory consumption and, therefore, GC pauses.

TABLE 2
Result Overview for Flink (FL), Kafka Streams (K), Spark Streaming (SP) and Structured Streaming (ST)

Workload	Decision factor	FL	K	SP	ST
Latency	stateless stages	4	4	1	1
	built-in stateful API	4	3	4	1
	customized stateful API	4	4		2
Sustainable throughput	unconstrained latency	2	1	3	4
	constrained latency	4	3	2	1
Single burst	initial output	4	3	1	2
	processing time	2	1	2	4
Periodic burst	stateless stages	3	3	1	4
	built-in stateful API	4	1	3	1
	customized stateful API	4	4		2

Scores can only be interpreted relatively to the scores of the other frameworks. Higher scores indicate better performance.

5 Results

In this section, we discuss the results of each of the workloads. First, we discuss the latency and sustainable throughput workloads and afterwards the burst workloads. For all of the results we did multiple runs with similar results and we discarded the first five minutes of the result timeseries to filter out warm-up effects. An overview of the results has been summarized in Table 2 per workload and for each additional decision factor. These decision factors influence the decision for a framework based on job requirements or pipeline complexity. A higher score refers to better performance. The scores do not have an absolute value and should be interpreted relatively to the scores of the other frameworks.

5.1 Workload for Latency Measurement

This first workload was designed to measure latency accurately, as was described in Section 3.4. The latency distribution for each of the pipeline complexities and frameworks has been given in Fig. 2.

By using event-driven processing and no buffering, Flink and Kafka Streams have the lowest latency for ingesting the data, with a median of 0 ms and a p99 of 1–2 ms. When parsing is added to the pipeline, the median latency of both frameworks increases by 1 ms. Flink has a lower p99 latency than Kafka Streams, 2 ms and 7 ms respectively. When we experiment with higher buffer timeouts and linger times, these latencies rise linearly, as documented in the Supplemental File and confirming [12], available online. The latencies for Spark Streaming and Structured Streaming are considerably higher, as expected for a micro-batch approach. For Spark Streaming we chose a micro-batch interval of 200 ms for these stateless stages, giving us median latencies of around 140 ms to 150 ms and p99 latencies of 316 ms to 350 ms. Structured Streaming uses the approach of processing as fast as possible. As a result, the median latency is slightly lower than for Spark Streaming, with 105 ms for ingesting and 148 ms when parsing is added. We notice, however, much more pronounced tail latencies with events requiring up to 930 ms of processing time. This can be an important factor for jobs under tight SLAs.

When we include the joining stage in the processing pipeline, the differences in latencies between the frameworks grow. For Spark Streaming, the micro-batch interval was raised to one second because this is equal to the join interval. Since processing one batch takes longer, Structured Streaming also forms larger micro-batches. The median latency of Spark Streaming was 650 ms after joining. The micro-batch interval is 1,000 ms so the buffering time for an event is on average 500 ms. Since the median latency is 650 ms, we infer that it takes an additional 150 ms to process all events of the batch. This is confirmed by the form of the distribution, which is rather uniform between 170 ms and 1,140 ms. Speed and flow observations of one measurement point are sent immediately after each other. The lower latencies come from pairs of events that arrived at the end of the tumbling window while the higher latencies come from pairs of events

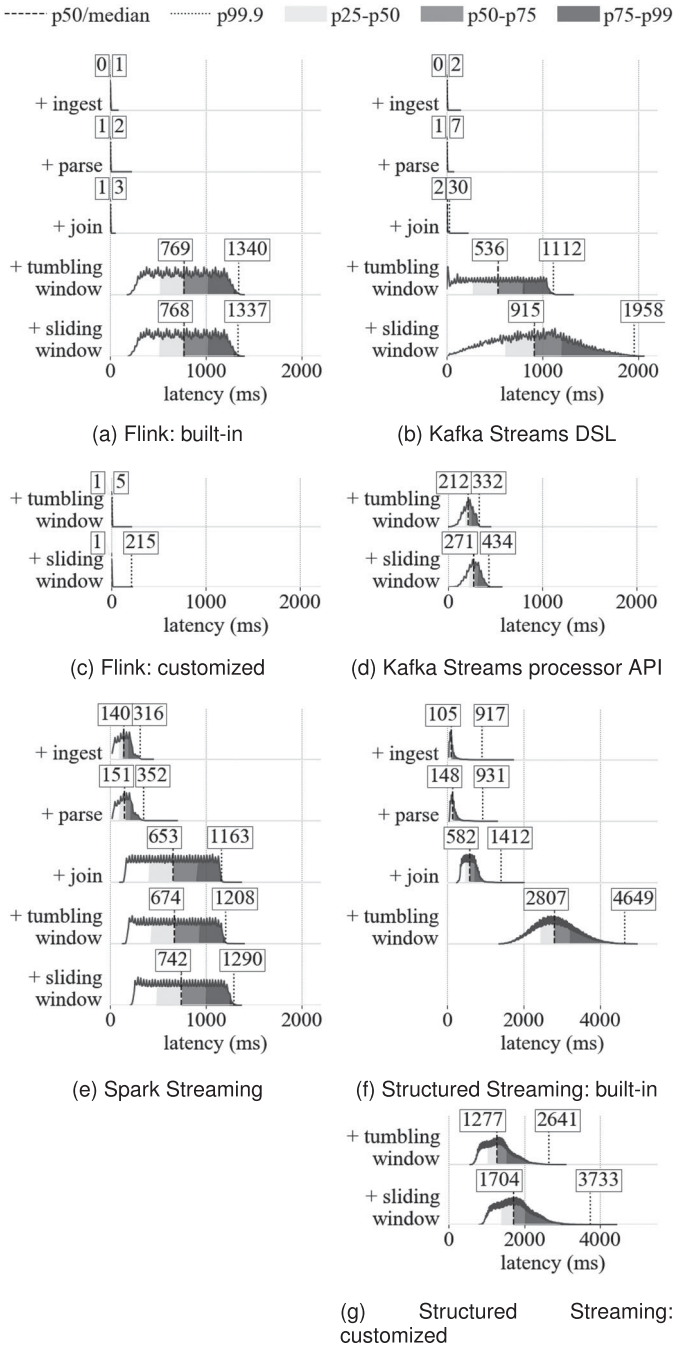


Fig. 2. Latency distribution for all frameworks. The stage until which the pipeline was executed is shown on the y-axis. The chart for Structured Streaming has a different x-axis scale than the other charts.

that arrived at the beginning of the window. Processing a batch of events, therefore, took between 140–170 ms. Furthermore, we see that the variance of the latency increased after adding the join. This is due to the fact that the events are joined using a tumbling window. The p99.999 latency is 200 ms higher than the p99 latency, meaning there are some minor outbreaks but that in general latency stays within predictable ranges. The latencies for Structured Streaming suffered from a much longer tail with the p99 above 1,400 ms, while the median stays at 582 ms.

For the event-driven processing frameworks, Flink and Kafka Streams, the type of join used is an interval join which can obtain a much lower latency. By joining events that lay within a specified time range from each other, the framework can give an output immediately after both required events arrive and does not wait for an interval to time out. The latency of joining is the lowest for Flink with a median round-trip latency of 1 ms and 99 percent of the events processed under 3 ms. Kafka Streams reaches median latencies of 2 ms and shows much higher tail latencies with a p99 of 30 ms and a p99.999 of 218 ms. We always compute the latency of the last event that was required to do the join. If we would incorporate the latency of the first event, these latencies would be higher, but we would not be expressing processing time.

The last two stages of the processing pipeline are the windowing stages: a tumbling window, as well as a sliding window are implemented and chained together. At the time of writing, Structured Streaming did not support chaining built-in aggregations. Therefore, we do not execute the built-in sliding window stage for Structured Streaming. We can circumvent this issue by using map functions with custom state management and processing time semantics, as discussed later. The performance of the built-in tumbling window behind the join led to a large performance degradation with p99 latencies of 4,649 ms and a median latency of 2,807 ms. There are a few main contributors to this 2,800 ms median latency. The first one is micro-batching. Despite setting the trigger to 0 ms to achieve processing as fast as possible, we still see micro-batches occurring of around 800–1,000 ms with occasional increases due to garbage collection. With a constant stream coming in this means that each event already had an average delay of 400-500 ms before processing has even started. The processing time of the batch in which the event resides then adds another 1,000 ms to this time. Once a record has been joined, its propagation to the tumbling window phase is delayed until event time has passed the window end and watermark interval, which means records are buffered for another micro-batch interval. These three factors together add up to a base latency of 2,800 ms and an even higher p99 latency. As confirmed by the developers of Structured Streaming in [6], the micro-batch mode of Structured Streaming has been developed to sustain high throughput and should not be used for latency-sensitive use cases. By using customized stateful mapping functionality with processing time semantics, we can improve performance since we do not rely on watermark propagation anymore. As can be seen in Fig. 2g, the latency of the tumbling window stage now has a median of 1,277 ms. Adding the sliding window stage, leads to a median latency of 1,704 ms. We still see large tail latencies for both stages, going up to 3,700 ms for the sliding window stage. This can be an issue for use cases with tight latency requirements.

For the older Spark Streaming API, we measure low median latencies of 667 for tumbling window and 765 ms after adding the sliding window. However, we see the 99th percentile latencies increase with complexity. We also notice that the uniform distribution caused by the tumbling window join persists throughout the following stages. The tumbling window as well as sliding window have slide intervals of one second. Therefore, no latency reductions can be obtained by using customized implementations since the latency is directly tight to the fixed micro-batch interval. Spark Streaming becomes a very good contender for these more complex transformations to the built-in high-level APIs of natively faster frameworks Flink and Kafka Streams. For Flink and Kafka Streams we compare two implementations. When using the built-in API, the latency of Kafka Streams increases the least with the addition of the tumbling window, although this behavior is not sustained when the sliding window is added. For all frameworks, we notice growing tail latencies when complexity is added via joins and aggregations. Partly this is due to the checkpointing mechanisms that are required to do stateful stream processing. As state increases, the garbage collection becomes more time consuming and less effective because a larger state needs to be kept and checked at each cycle. Finally, the shuffling and interdependence between the observations to generate the output increases the variance of the latency. Through customizing stateful implementations, we can heavily reduce the latency for some use cases. For Flink, using a custom trigger for the tumbling window reduced the median latency to 1 ms with a p99 of 5 ms. This is much lower than when using the default trigger because the default trigger forces events to be buffered till the end of the window and added an additional delay due to the burst of data that needs to be processed at the end of the interval. By allowing events to be sent out as fast as possible, the latency can be significantly reduced. Similarly, adding a processor function with managed keyed state and timers to do the sliding window keeps the median latency at 1 ms and slightly increases the p99 to 215 ms. This shows that the flexibility of the Flink API enables optimizing the way stateful operations are done by e.g., reducing the time and amount of data that is buffered. For Kafka Streams we also notice large latency reductions (4x) when using the low-level API, however, not as large as those of Flink. Besides, it is important to keep in mind that not all processing flows can be optimized by a customized implementation. In this flow, this was the case because it unnecessarily blocks events until it reaches a window timeout.

We can conclude that event-driven frameworks, such as Flink and Kafka Streams, are preferred when doing simple ETL operations. Built-in windowing operations reduce the difference between micro-batching and event-driven systems, shifting in favor of Spark Streaming. The type of stateful transformation chosen has a large influence on latency. Event-driven frameworks make it possible to do interval joins and stateful operations with custom triggers and state management which can lead to significant latency reductions compared to built-in windowing functionality. Structured Streaming also benefits from a customized implementation

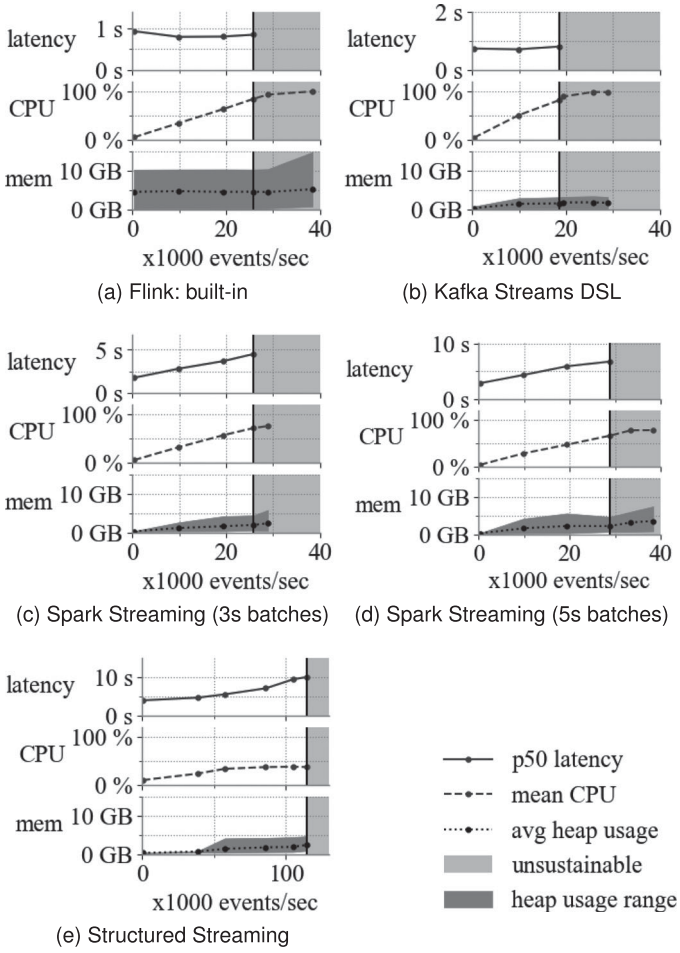


Fig. 3. Sustainable throughput: performance for different throughput levels for the tumbling window stage. Each marker represents a benchmark run for the corresponding throughput level. The light grey zone marks unsustainable levels of throughput. Structured Streaming has different scale on x-axis.

since we do not rely on watermarks anymore for output to be send out. When tight latency SLAs are imposed, the longer tail latencies and larger distribution spread for complex pipelines should be kept in mind.

5.2 Workload for Sustainable Throughput Measurement

The sustainable throughput for each of the frameworks is measured for execution of the pipeline up to and including the tumbling window stage. In Fig. 3, peak sustainable throughput is denoted by the beginning of the grey zone. In this zone, either mean CPU went above a threshold or processing could not keep up with the input stream. Dot markers represent benchmark runs at corresponding throughput levels. The throughput levels bordering this grey zone are often also at risk of becoming unsustainable since the job does not have any buffer for unforeseen bursts, delays, etc. For unsustainable loads, we show CPU and memory metrics but we do not show latency metrics since increasing delays makes it soar to very high levels of which the median has no interpretative value. For most frameworks, an increase in latency signals throughput levels becoming unsustainable. It is important to note that the latency we show here is computed using a multi-broker Kafka cluster and is, therefore, only accurate to a tenth of a second. Hence, we use this latency merely to compare broad trends. Memory usage fluctuations are shown by the minimum, maximum and average values of the run to avoid cluttering the graph. We take a look at the growth in heap usage and the effectiveness of GC cycles to clean up the heap memory as throughput increases.

For Spark Streaming, we increase the batch interval since this increases the peak sustainable throughput significantly. With a batch interval of one second Spark Streaming was not able to handle a peak sustainable throughput of over 20 000 events per second. With a batch interval of three seconds a sustainable peak throughput of 26 000 events per second is reached. When we increase the batch interval to five seconds, it increases to 30 000 events per second which confirms that sustainable throughput can be increased by increasing the batch interval, but this comes with an inherent latency cost. We notice that for the runs with low load the latency is around 300 ms longer than half of the batch interval, implying that the processing of the batch took approximately 300 ms. As throughput increases, the latency steadily increases as well due to the increased processing time for larger batches. Jobs start incurring delays when the processing time becomes larger than the batch interval. At this point, the latency approaches 1.5 times the batch interval since this includes the buffering at the receiver side. Structured Streaming, on the other hand, uses a dynamic batch interval to improve the data processing speed. When the throughput increases, the batch interval and latencies increase as well. This mechanism allows Structured Streaming to handle a throughput of 115 000 events per second with a CPU utilization of 50 percent. When the throughput increases any further the batch interval passes the 10 second median latency threshold we imposed, but still remains sustainable. At this point, we are confronted with a trade-off between latency and throughput. As stated in [6], Structured Streaming has been optimized for throughput. For latency-sensitive use cases, they recommend using the continuous processing mode which is still experimental. Since we focus on benchmarking real-time processing systems, we do not allow latency to go above 10 seconds in this workload. Following this definition, Structured Streaming has a peak sustainable throughput of 115 000 events per second while maintaining a median latency under 10 seconds. The pattern of increasing latencies for increasing throughput is not apparent for Flink and Kafka Streams. The main reason is that Flink and Kafka Streams do event-driven processing with very limited buffering on the receiver side. When the constant rate throughput is at such a level that it causes backpressure on the receiver side, the throughput is already at unsustainable levels. Flink can sustain over 30 000 events per second for a buffer timeout of 100 ms. At this point, CPU levels are over 95 percent. In order to avoid mean CPU utilization above a threshold of 80 percent, we set the peak sustainable throughput at 26 000 events per second. As an experiment we set the buffer timeout to –1, which means that buffers are only flushed when they are full. However, this did not increase throughput further. This is in agreement with the paper presenting Flink [12], which shows a large throughput increase when increasing the timeout from 0 to 50 ms but a limited throughput increase when increasing from 50 ms to 100 ms. When using the custom trigger implementation, the peak sustainable throughput decreased slightly to 25 000 events per

second, as can be consulted in the Supplemental File, available online. The custom trigger implementation is slightly more CPU intensive than the default implementation and CPU is the bottleneck to increasing throughput further.

Kafka Streams DSL shows relatively constant latencies of around 700 ms for all sustainable throughput levels. At a throughput level of 18 500 events per second, average CPU levels reach 83 percent. When the throughput increases further, processing starts lagging behind on the input stream and latencies soar. When using the processor API to implement the tumbling window stage, there was a slight increase in peak sustainable throughput to 20 000 events per second, as can be consulted in the Supplemental File, available online.

For all runs, throughput has a linear relationship with CPU usage. We see that the runs with unsustainable throughput levels, i.e., in the grey zone, have average CPU levels of above 80 percent. Additionally, for most frameworks, an increase in throughput leads to an increase in memory usage due to the larger input and larger state. At sustainable levels of throughput, Flink is the only framework where we do not notice an increase in average heap memory usage when throughput increases. We see memory ramp up to approximately 9 GB. It then drops back to around 100 MB after it is collected. The maximum heap used only starts increasing after throughput reaches unsustainable levels.

As visualized in Table 2, our results confirm that Structured Streaming can reach higher throughput due to micro-batch execution optimization. Spark Streaming also shows a substantial peak throughput increase when the micro-batch interval is raised. We, however, make the large latency cost that comes with this explicit. For use cases with tighter latency requirements, Flink obtains the best results.

5.3 Workload With Burst at Startup

We executed the single burst workload for all frameworks and all pipeline complexities and implementations. Since we notice similar behavior for all complexities, we will focus on the results for the tumbling window stage. Additional results have been put in the Supplemental File in Section 5.3, available online. In Fig. 4, the behavior of the framework during the first minutes after startup is shown. The light grey zone shows where the framework caught up to the incoming stream. In the first row of Fig. 4, we see the output throughput with clear differences across frameworks that are mainly due to a micro-batch or event-driven approach. Kafka Streams and Flink publish the first output 16 seconds after startup. Structured Streaming takes longer with 44 seconds. Spark Streaming requires more than a minute of processing time before the first output is published. However, these initial batches for Spark Streaming, contain around 80 000–110 000 events in six subsequent batches. Structured Streaming shows batch sizes going up to 150 000 events for the first five seconds and manages to catch up the fastest after 53 seconds. Due to its adaptive micro-batch approach it can process large batches of data well and can quickly catch up with the input data, validating the statements made by the developers of Structured Streaming in [6]. Spark Streaming also uses a micro-batch approach, causing a similar large burst in output as observed with Structured Streaming. After 76 seconds most of the data is processed, however batches stay elevated

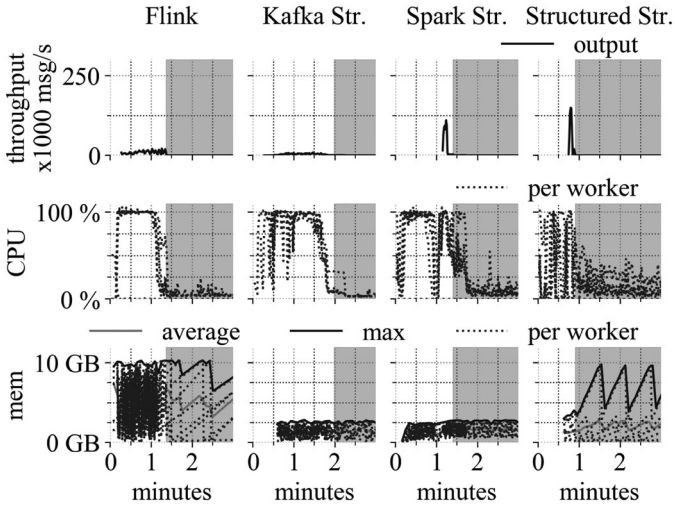


Fig. 4. Single burst workload until tumbling window stage: performance for first three minutes of the run. The grey zone denotes where processing has caught up on the input stream.

at around 2,700 events for another eight seconds afterwards. Flink is able to catch up after 81 seconds by publishing smaller batches over a longer period of time. Kafka Streams requires most time to process all data with 99 seconds and also publishes smaller batches over a longer period of time. To make the built-in window implementation of Kafka Streams process the data in order, the allowed maximum task idle time had to be raised as described in Section 4.2.

During the processing of the burst, all workers of Flink, Kafka Streams and Spark Streaming have 100 percent CPU utilization. The CPU utilization of Spark Streaming drops immediately before the batches are published onto Kafka. For Structured Streaming, we do not reach 100 percent CPU for the initial minutes of processing and we notice one worker having increased CPU utilization compared to the other workers of the cluster. Similar to other workloads, Flink uses up to 9 GB of the heap while Spark Streaming and Kafka Streams stay around 3 GB.

When executing the workload until other stages or for different implementations, we notice similar patterns as can be consulted in the Supplemental File, available online. The customized implementation finishes faster for Kafka Streams and Structured Streaming and reaches a higher peak throughput. The peak throughput for Structured Streaming goes up to even 229 000 events per second.

In conclusion, micro-batch frameworks take in a big load of data on start-up and take a while to give initial outputs. Event-driven frameworks take in smaller loads of data and start delivering output sooner. Structured Streaming consistently catches up the fastest, followed by Flink, Spark Streaming and finally Kafka Streams.

5.4 Workload With Periodic Bursts

We ran the periodic burst workload for several pipeline complexities. The results for the tumbling window stage are shown in Fig. 5 and for the join stage in Fig. 6. Additional results have been put in the Supplemental File in Section 5.4, available online. We zoom in on a one-minute-interval nearing the end of our run to get a clearer view of what happens after a burst of data is ingested. Similar to the workload for

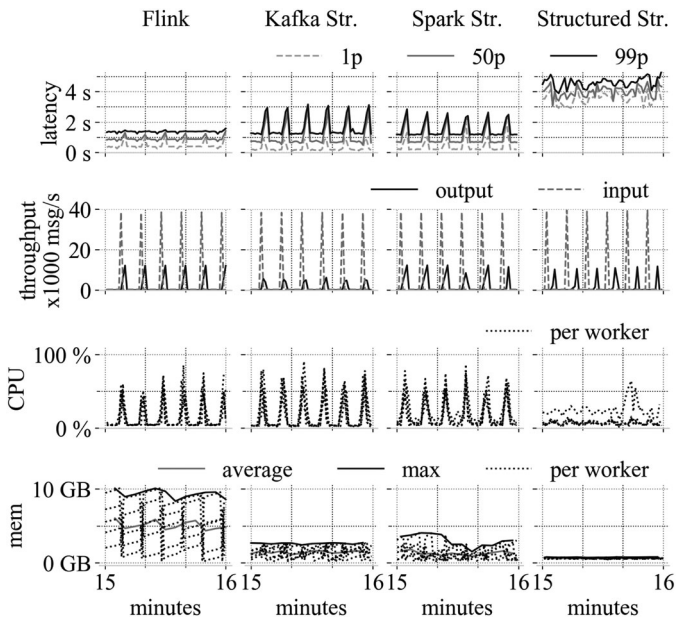


Fig. 5. Periodic burst workload until the tumbling window stage: performance metrics for the last minute of the run.

sustainable throughput, latency measurements are only accurate to tenth of a second.

When we look at the input throughput, we can clearly discriminate the periodic bursts of around 38 000 messages every ten seconds. In between bursts, the input throughput stays at 400 messages per second. For the tumbling window stage, each output event requires on average 3.17 input events. For the join stage, two input events are required. This explains the discrepancy in messages per second between the input and output throughput. The differences in the height of the output throughput rate between frameworks can be explained by the fact that some frameworks ingest the burst at once while others ingest it in several chunks and output the results more gradually.

In Fig. 5, the relationships between the different performance metrics reveal some interesting patterns. For Spark Streaming, we see that after a burst of the input throughput, CPU levels rise and are followed by a burst in the output throughput. The latency increases for two to three seconds after the burst, which can be noted by the width of the peak. The median latency during the low load periods is around 700 ms. However, the first seconds after a burst events have an inflated median latency of 1,370 ms, then it goes up to 2,050 ms after which it falls back to 1,350 ms again for the final batch. The main reason for this is the increased processing time required to do all the computations for this burst of data. Besides, the events that come in right after the burst incur a scheduling delay due to this increased processing time and, therefore, have inflated latencies as well.

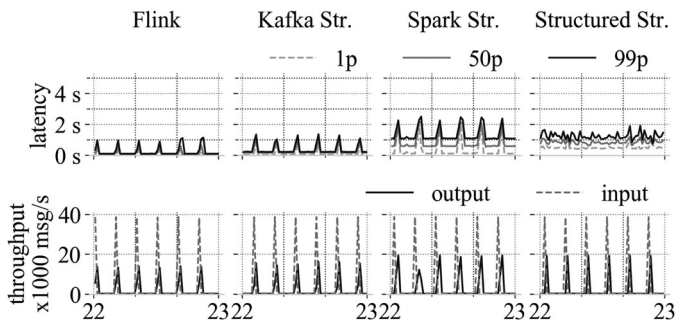
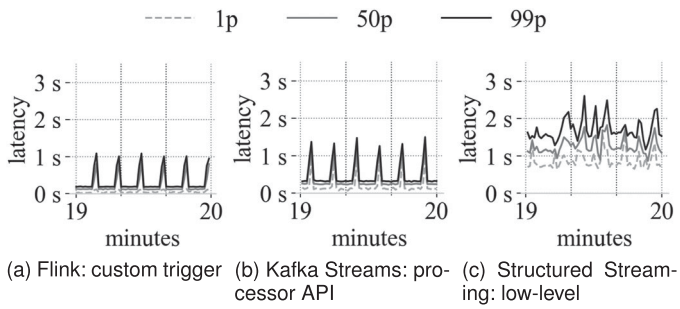


Fig. 6. Periodic burst workload until the join stage: performance metrics for the last minute of the run.



(a) Flink: custom trigger (b) Kafka Streams: processor API (c) Structured Streaming: low-level

Fig. 7. Periodic burst workload for customized tumbling window implementations for Flink, Kafka Streams, and Structured Streaming: latency metrics for one minute of the run.

In Fig. 5 we also show the performance of the default trigger implementation for Flink and the DSL implementation for Kafka Streams. We see that the impact of bursts on the latency is the least for Flink with the median latency rising from around 800 ms to 1,250 ms and the p99 latency staying fairly constant and under 1,500 ms. Due to the lower impact of bursts on the processing time of Flink, the period for which latency remains at higher levels is also shorter. When we execute this workload for earlier stages, such as the parse and join stage (Fig. 6), we see a larger effect on the p50 and p99 latency. This can be explained by the fact that the tumbling window applies buffering which dampens latency differences across events. When using custom triggers, this buffering is reduced as can be seen in Fig. 7. We now see clear reactions to bursts again and a very low latency in between bursts. Furthermore, the p99 latency after a burst is 1,200 ms which is still lower than the almost constant latency of the default trigger implementation. When we use the low-level API of Kafka Streams to implement this stage, the latency of processing a burst is similar to that of Flink and much lower than with the DSL implementation. With the DSL implementation, the median and p99 latency rise considerably after a burst, up to 2,800 ms. For the parsing stage, the latency of Kafka Streams is even slightly lower than that of Flink. In contrast, we can see in Fig. 5 that the latency of Structured Streaming exhibits a very irregular course with the median fluctuating between 3,000 ms and 4,500 ms and no clear reactions to bursts. This is due to the delays in watermark propagation that have been explained in Section 5.1. When we use the customized implementation that does not use watermarks (Fig. 7), the median latency remains under the two seconds. Structured Streaming can sustain the largest throughput due to its dynamic micro-batch approach and it also benefits from that when processing bursts. For the join stage and the parsing stages, we see lower and less variable latencies and no clear reactions to bursts. Finally, we see that particularly for Kafka Streams and Spark Streaming immediately after a burst, the distribution of the latency becomes much more narrow, with median and p99 latencies almost converging.

For CPU utilization, we see that for most frameworks all workers are equally utilized when processing bursts, with almost all lines in Fig. 5 coinciding. During the processing

of a burst we see CPU usage of 60-90 percent for all workers, while the CPU utilization throughout the rest of the run remains around 2-4 percent. Structured Streaming shows different behavior with an average load of 6-10 percent and peaks between 20-40 percent mainly contributed by one worker. This worker had almost constantly higher CPU usage than the other workers in the cluster.

When looking at the heap usage throughout this run, we see that Structured Streaming uses the least memory. For Flink we see memory usage ramp up to 10 GB before a GC is triggered. These GC cycles are synchronized with the frequency of the bursts. After a burst has been processed, GC is initiated. For the other frameworks this behavior is not as explicit. The amount of memory that is used is not necessarily important for framework performance. What is important, is that GCs stay effective and that memory frees up well after each GC, which is the case for all frameworks.

As a conclusion, we can state that using custom implementations can not only reduce the latency of the pipeline in general but also the latency of processing bursts. When we use only high-level APIs, Structured Streaming shows much resiliency against bursts for the parsing and joining pipelines and also uses the least resources. Structured Streaming benefits greatly from its micro-batch execution optimizations when processing bursts, confirming [6]. Due to its dynamic micro-batch interval, it reaches much lower latencies than Spark Streaming for these earlier stages. When the pipeline gets more complex Flink suffers the least from processing bursty data, followed by Spark Streaming. Detailed visualizations of the other workloads has been put in the Supplemental File, available online.

6 Conclusion

In this paper we have presented an open-source benchmark implementation and full analysis of two popular stream processing frameworks, Flink and Spark Streaming, and two emerging frameworks, Structured Streaming and Kafka Streams. Four workloads were designed to cover different processing scenarios and ensure correct measurements of all metrics. For each workload, we offer insights on different processing pipelines of increasing complexity and on the parameters that should be tuned for the specific scenario. Our latency workload provides correct measurements by capturing time metrics on a single machine, i.e., Kafka broker. Furthermore, dedicated workloads were defined to measure peak sustainable throughput and the capacity of the frameworks to overcome bursts or catch up on delay. We discuss the metrics latency, throughput and resource consumption and the trade-offs they offer for each workload. By including built-in as well as customized implementations where possible, we aim to show the advantages of having a flexible API at your disposal.

A concise overview of the results is shown in Table 2. For simple stateless operations such as ingest and parse, we conclude that Flink processes with the lowest latency. For these tasks, the newer Structured Streaming API of Spark shows potential with lower median latencies than Spark Streaming. However, it suffers from longer tail latencies making it less predictable to reach SLAs with tight upper bounds. For the joining stage, event-driven frameworks such as Flink and Kafka Streams have an advantage due to the interval join capability. This type of join can output events with a much lower latency, compared to the tumbling window join available in micro-batch frameworks. The latency differences between event-driven and micro-batch systems disappear when the pipeline is extended with built-in tumbling and sliding windows. For these operations, Spark Streaming shows lower tail latencies and outperforms the other frameworks. However, by using the flexibility frameworks offer for these stateful operations through custom triggers and low-level processing capabilities, pipelines can be optimized to reach lower latency.

A second important metric is throughput. Structured Streaming is able to sustain the highest throughput but this comes at significant latency costs. If latency and throughput are both equally important, Flink becomes the most interesting option. When processing a single large burst at startup, Flink outputs the first events the soonest while Structured Streaming finishes processing the entire burst the earliest. For use cases where the input stream exhibits occasional bursts, the least performance impact is perceived when using Flink as the processing framework. When using the default implementation, the p99 latency of Flink stays constant throughout bursts while for other frameworks significant increases are noted. The latency of processing bursts can again be optimized by using low-level APIs.

Finally, the results of this paper can be used as a guideline when choosing the right tool for a processing job by not only focusing on an inter-framework comparison but also highlighting the effects of different pipeline complexities, implementations and data characteristics. When choosing a framework for a use case, the decision should be made based on the most important requirements. If subsecond latency is critical, even in the case of bursts, it is advised to use highly optimized implementations with an event-driven framework such as Flink. When subsecond latency is not required, Structured Streaming offers a high-level, intuitive API that gives very high throughput with minimal tuning. It is important to keep in mind that at the time of this writing parts of the Structured Streaming API were still experimental and not all pipelines using built-in operations were supported yet. For those use cases, Spark Streaming with a high micro-batch interval offers a good alternative. For ETL jobs on data residing on a Kafka cluster that do not require very high throughput and have reasonable latency requirements, Kafka Streams offers an interesting alternative to Flink with the additional advantage that it does not require a cluster pre-installed.

7 Limitations and Further Research

This benchmark could be extended with joins with static datasets and the stateful operators should be analyzed under different window lengths. As well, other frameworks could be included such as Apex, Beam and Storm. Finally, the fault tolerance and scalability of the frameworks under these workloads are interesting directions for future work.

Acknowledgments

This research was done in close collaboration with Klarrio, a cloud native integrator and software house specialized in

bidirectional ingest and streaming frameworks aimed at IoT & Big Data/Analytics project implementations. For more information please visit https://klarrio.com.

References

[1] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream processing engines," in *Proc. IEEE 34th Int. Conf. Data Eng.*, 2018.

[2] Z. Karakaya, A. Yazici, and M. Alayyoub, "A comparison of stream processing frameworks," in *Proc. Int. Conf. Comput. Appl.*, 2017, pp. 1–12.

[3] S. Chintapalli *et al.*, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2016, pp. 1789–1792.

[4] R. Lu, G. Wu, B. Xie, and J. Hu, "Stream bench: Towards benchmarking modern distributed stream computing frameworks," in *Proc. IEEE/ACM 7th Int. Conf. Utility Cloud Comput.*, 2014, pp. 69–78.

[5] S. Qian, G. Wu, J. Huang, and T. Das, "Benchmarking modern distributed streaming platforms," in *Proc. IEEE Int. Conf. Ind. Technol.*, 2016, pp. 592–598.

[6] M. Armbrust *et al.*, "Structured streaming: A declarative API for real-time applications in apache spark," in *Proc. Int. Conf. Manage. Data*, 2018, pp. 601–613.

[7] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 423–438.

[8] A. Shukla and Y. Simmhan, "Benchmarking distributed stream processing platforms for IoT applications," in *Proc. Technol. Conf. Perform. Eval. Benchmarking*, 2016, pp. 90–106.

[9] A. Shukla, S. Chaturvedi, and Y. Simmhan, "RIoTBench: An IoT benchmark for distributed stream processing systems," *Concurrency Comput. Pract. Experience*, vol. 29, no. 21, 2017, Art. no. e4257.

[10] G. van Dongen, B. Steurtewagen, and D. Van den Poel, "Latency measurement of fine-grained operations in benchmarking distributed stream processing frameworks," in *Proc. IEEE Int. Congress Big Data*, 2018, pp. 247–250.

[11] J. Kreps *et al.*, "Kafka: A distributed messaging system for log processing," in *Proc. NetDB*, 2011, pp. 1–7.

[12] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin IEEE Comput. Soc. Tech. Committee Data Eng.*, vol. 36, no. 4, pp. 28–38, 2015.

[13] M. J. Sax, G. Wang, M. Weidlich, and J.-C. Freytag, "Streams and tables: Two sides of the same coin," in *Proc. Int. Workshop Real-Time Bus. Intell. Analytics*, 2018, Art. no. 1.

[14] Flink Documentation, 2019, Accessed: Dec. 19, 2019. [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-stable/

[15] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in apache flink®: Consistent stateful distributed stream processing," *Proc. VLDB Endowment*, vol. 10, no. 12, pp. 1718–1729, 2017.

[16] Kafka Streams Documentation, 2019, Accessed: Dec. 19, 2019. [Online]. Available: https://kafka.apache.org/documentation/streams/

[17] Apache Spark: Spark programming guide, Accessed: Dec. 19, 2019, 2019. [Online]. Available: https://spark.apache.org/docs/latest/streaming-programming-guide.html

[18] Structured Streaming, 2019, Accessed: Dec. 19, 2019. [Online]. Available: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

[19] cAdvisor, 2019, Accessed: Jun. 04, 2019. [Online]. Available: https://github.com/google/cadvisor

[20] M. Caporaloni and R. Ambrosini, "How closely can a personal computer clock track the UTC timescale via the Internet?" *Eur. J. Phys.*, vol. 23, no. 4, pp. L17–L21, 2002.

[21] Kafka Improvement Proposals: KIP-32 - Add timestamps to Kafka message, 2019, Accessed Jun. 04, 2019. [Online]. Available: https://cwiki.apache.org/confluence/display/KAFKA/KIP-32+-+Add+timestamps+to+Kafka+message

[22] Y. Byzek, "Optimizing your apache kafka deployment," 2017, Accessed: Dec. 18, 2019. [Online]. Available: https://www.confluent.io/wp-content/uploads/Optimizing-Your-Apache-Kafka-Deployment-1.pdf

Giselle van Dongen (Member, IEEE) is working toward the PhD degree at Ghent University, Ghent, Belgium, teaching and benchmarking real-time distributed processing systems such as Spark Streaming, Flink, Kafka Streams, and Storm. Concurrently, she is lead data scientist at Klarrio specialising in real-time data analysis, processing and visualisation.

Dirk Van den Poel (Senior Member, IEEE) received the PhD degree. He is currently a senior full professor of Data Analytics/Big Data at Ghent University, Belgium. He teaches courses such as Big Data, Databases, Social Media and Web Analytics, Analytical Customer Relationship Management, Advanced Predictive Analytics, and Predictive and Prescriptive Analytics. He co-founded the advanced master of science in marketing analysis, the first (predictive) analytics master program in the world as well as the master of science in statistical data analysis and the master of science in business engineering/data analytics.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Location-Aware and Budget-Constrained Service Deployment for Composite Applications in Multi-Cloud Environment

Tao Shi, Hui Ma, Gang Chen, and Sven Hartmann

Abstract—Enterprise application providers are increasingly moving their workloads to the cloud for technical and economic benefits. Multi-cloud environment makes it possible to orchestrate multiple cloud resources. With the increasing number of available cloud resources provided by multiple cloud providers at different locations with different prices, application providers face the challenge to select proper cloud resources to deploy their applications in the form of a workflow of component service units. Existing studies usually consider minimizing execution time or/and deployment cost. From the perspective of application providers, however, they also pay huge attention to application response time, including particularly network latency between deployed services and users. Meanwhile, application deployment is often subject to stringent budgetary control to ensure financial viability. This article studies a new type of composite application deployment problem that jointly considers both the performance optimization and budget control in multi-cloud at the global scale. To find solutions with minimal response time without running into the risk of over-spending, we propose a hybrid GA-based approach, featuring new design of domain-tailored service clustering, repair algorithm, solution representation, population initialization, and genetic operators. Extensive experiments using the real-world dataset demonstrate that our proposed hybrid GA approach outperforms some recently proposed approaches.

Index Terms—Location, budget, service deployment, multi-cloud, composite applications, genetic algorithm, clustering, repair algorithm

1 Introduction

Gartner[1] forecasts that the worldwide cloud services market will exceed 260 billion U.S. dollars in 2020 and maintain rapid growth [1]. According to Forbes [2], 83 percent of enterprise workloads will be moved to cloud by 2020, and half of them will be hosted on public cloud platforms.

The thriving public cloud market drastically affects its participants, i.e., cloud providers and cloud consumers. On the one hand, the competitive providers are progressively increasing investment and launching various services with customized properties, pricing models and service level agreements (SLAs). On the other hand, consumers have more choices to support their production environments and business-critical operations [1].

From the perspective of cloud consumers, the usage of services and resources from multiple cloud providers can provide many benefits, such as access to widely distributed and legislation-compliant resources, avoidance of vendor lock-in, and high application resilience [3]. In recent years, cloud customers are interested in embracing a *multi-cloud* environment that takes no account of volunteer interconnection and sharing of infrastructures among cloud providers [4]. The deployment and management of multiple clouds are usually abstracted by a third-party, *broker* [5], which can provide a single entry point to multiple clouds. Supported by the broker and the multi-cloud platform, the cloud customers are able to transit workload among different cloud providers freely to further increase their business flexibility.

In cloud environments, cloud consumers can be either the end-users who use the cloud services directly or the software/application service providers who have their own consumers [6], e.g., Software-as-a-Service (SaaS) provider, Salesforce Customer Relationship Management (CRM).[2] In this paper, we focus on *application providers* who offer domain specific software/applications to end users by using multi-cloud services/resources made available by cloud providers.

It is typical for an application provider to offer several business applications simultaneously, and the relevant business processes are usually abstracted as workflows [7]. These *composite applications* consist of a set of specific functionalities, i.e., *constituent services*. Different applications often share some common constituent services [8]. To distinguish different services, we use *application services* or *services* to refer constituent services in the remaining of this paper. For example, a tax-setting service can be shared by ordering, selling, and payroll applications.

1. https://www.gartner.com/

2. https://www.salesforce.com/

- *T. Shi, H. Ma, and G. Chen are with the School of Engineering and Computer Science, Victoria University of Wellington, Wellington 6012, New Zealand. E-mail: {tao.shi, hui.ma, aaron.chen}@ecs.vuw.ac.nz.*
- *S. Hartmann is with the Department of Informatics, Clausthal University of Technology, 38678 Clausthal-Zellerfeld, Germany. E-mail: sven.hartmann@tu-clausthal.de.*

Manuscript received 20 Nov. 2019; revised 29 Feb. 2020; accepted 10 Mar. 2020. Date of publication 17 Mar. 2020; date of current version 9 Apr. 2020. (Corresponding author: Tao Shi.)
Recommended for acceptance by R. Yahyapour.
Digital Object Identifier no. 10.1109/TPDS.2020.2981306

1045-9219 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Under the public cloud paradigm, many deployment patterns, such as Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Function-as-a-Service (FaaS), are available for application services. Application providers often prefer IaaS in practice in order to achieve high level of control and flexible management goals [9]. In this paper, we consider the major scenario of deploying application services on a set of virtual machines (VMs) to support composite applications capable of processing tens of thousands or even millions of requests on a daily basis [10].

The deployment of services is crucial for application providers since it affects the Quality of Service (QoS) of composite applications significantly [8], [11]. The problem has received some research attentions in recent years. For example, the service deployment problem is solved by minimizing the *execution time* of composite applications in [8]. In multi-cloud, however, the *network latency* between users/data and services in different locations may significantly affect the application performance [12], [13]. Therefore, to evaluate accurately the performance of composite applications, we must consider the *average response time*, measured from the moment users make application requests to the moment when they receive the corresponding responses. Driven by the goal to minimize the average response time, we investigate the service deployment problem by selecting proper VMs from the global cloud marketplace to jointly support multiple composite applications subject to diverse user distributions and quality requirements.

It is widely known that leading cloud providers usually provide cloud resources, e.g., VM instances, with varied pricing subject to regions.[3] In consideration of both performance and cost, there are some research works searching for the Pareto front, in which each solution represents a unique trade-off deployment plan [14], [15]. However, enterprises are often more interested in budgetary control to ensure that their actual costs adhere closely to their financial plan [16]. In view of this demand, the budget management module is offered by many cloud providers, e.g., Amazon Web Service (AWS).[4] The budgetary control emphasizes on performance optimization within a given budget, which is of immense practical significance for application providers [17]. Therefore, it is necessary and important to formulate the service deployment problem as a *budget-constrained* optimization problem so as to truly meet the business management needs of many enterprises.

Genetic algorithm (GA) is a classic optimization technique to generate high-quality solutions for various combinatorial optimization problems with practical importance [18], [19]. In particular, driven by a population-based solution improvement framework, GA has been widely used to select proper multi-cloud resources for scalable cloud application deployment [20], [21], [22]. However, new technical innovations are required to effectively apply GA to solve our application deployment problem in multi-cloud. This is because: (1) We consider the geographical location of cloud resources. In the global cloud marketplace, a large number of different VM types are provided at different data centres. The large and complex search space can impair the scalability and applicability of the existing GA-based approaches [23]. (2) Many of GA-based algorithms [24], [25] solve constrained optimization problems by introducing penalty functions. However, numerous research works in the literature have shown that repair algorithms have the potential to outperform penalty-based algorithms in GA by adopting problem-specific heuristics [26], [27]. We need to propose a specific repair algorithm to solve our problem effectively, because there are no standard guidelines for the design of repair algorithms [28]. (3) To control the computation time of the repair process, we should also consider the efficiency of our proposed repair algorithm [26].

To address each of the challenges above, we have developed new algorithmic techniques under the GA framework for effective cloud application deployments. Specifically, we propose a service clustering mechanism based on the dependency among services to reduce the complexity/size of the search space, thereby ensuring the efficiency and effectiveness of our GA-based algorithm. Besides, we propose a problem-specific repair algorithm to progressively downgrade the service hosting VM types so as to meet the budget requirements at the expense of low performance deterioration. The repair algorithm is also designed to achieve a desirable trade-off between performance and computation cost by transforming the over-budget solutions chosen according to an adaptive bound into constraint-compliant solutions. The main contributions of this paper are summarized as follows:

First, we formally define the composite application deployment problem in multi-cloud as a constrained optimization problem with the goal to minimize response time subject to a budget constraint. To the best of our knowledge, this is the first study in the literature on the application deployment problem with realistic consideration of both the performance and the budget impact in multi-cloud at the global scale.

Second, we propose a novel GA-based approach to the budget-aware composite application deployment problem, featuring a newly designed and domain-tailored service clustering method, repair algorithm, solution representation, population initialization, and genetic operators.

Finally, to evaluate the proposed approach, we collect the information about VM capacity and pricing offered by the global top three cloud providers, i.e., Amazon, Microsoft, and Alibaba. Based on the collected information, extensive experiments have been conducted to deploy a large group of composite applications with different service diversities and budget factors. Our approach is compared to several state-of-the-art algorithms, i.e., BHEFT [29] and penalty-based GAs [24], [25] for constrained application deployment problems. The experimental results indicate that our approach significantly outperforms the existing methods, achieving up to about 8 percent performance improvement in terms of response time, and 100 percent budget satisfaction in the meantime.

The remainder of this paper is organised as follows. Section 2 discusses the related work about application deployment and the existing approaches for the budget-constraint optimization problem. Section 3 defines the problem, including the multi-cloud application deployment system. Section 4 presents the details of the hybrid GA-based approach.

3. https://aws.amazon.com/ec2/pricing/on-demand/
4. https://aws.amazon.com/aws-cost-management/aws-budgets/

Section 5 describes the design of experiments and analyses the simulation results. Section 6 concludes the paper.

2 Related Work

This section introduces the related works about application deployment and existing approaches for the budget-constrained optimization problem in cloud. The main challenges that need to be addressed in our problem are also highlighted.

2.1 Service Deployment for Composite Applications

In recent years, cloud computing is becoming a booming hosting paradigm for delivering business applications. How to deploy application services with optimal QoS becomes a critical issue [30], [31]. Several research works have studied the service deployment/placement problem. For example, Wen *et al.* studied application deployment on federated clouds to minimise the monetary cost while simultaneously meeting the security and reliability requirements in [30]. The work in [31] proposed a game theoretic method to optimize both the overall cost and execution time of cloud services. In our previous work [32], we minimize the product of deployment cost and response time of business applications by a GA-based algorithm. From the perspective of application providers, however, application deployment is often subject to stringent budgetary control to ensure financial viability. Therefore, we formulate the application deployment problem as a budget-constrained optimization problem in this paper.

Based on Service-Oriented Architecture (SOA) [33], there is an ongoing trend to develop composite applications with shared services [8], especially for professional service providers. In [8], Huang *et al.* described the inter-dependence and independence among services as a Service Relationship Graph (SRG). Then the problem was solved by transforming it into a minimum k-cut problem and deploying a set of services onto the given VMs. In our work, we focus on deploying composite applications in multi-cloud environment. Because there are various VM types available in numerous data centres, we design a clustering method to group services according to their inter-dependencies, which can reduce the search space effectively.

Business applications involve a potentially large number of user requests, and different applications must cater for varied user distributions. In practice, the user-centric information can be obtained easily by application providers, and utilized to determine the average processing time of services after service deployment. We model the processing of ongoing application requests through a queuing model to precisely capture the performance of an application, which is seldom considered by the existing research works on application deployment in cloud [10].

The service deployment problem shares some similarities with the VM placement problem, which has also received some research attention in the context of multi-cloud [12], [13], [34]. In [12], the VM placement problem was explored to cope not only with the deployment cost, but also the network latency between users and cloud data centres around the world, in the view of the significant impact of network latency on the performance of cloud services. From the perspective of cloud consumers, our previous work [13] improved the large-neighbour-search-based approaches developed in [12] by a GA-based framework. However, different from the problem of deploying composite applications with inter-dependent services in this paper, the VM placement problem in [12], [13] assumes all VMs and services as being independent, which may not be always true, especially when services are used to fulfill specific functionalities in workflows.

The review of the above mentioned papers reveals a need for the study on the service deployment problem considering response time of composite applications to accurately measure their real performance in multi-cloud.

2.2 Budget-Constrained Workflow Scheduling

Selecting the proper cloud services/resources for application tasks is another commonly studied problem, which is also known as the workflow scheduling problem [14]. Most of the existing works focus on scheduling *scientific* workflows with data-intensive or computation-intensive tasks [24]. Compared with the time-consuming execution of these tasks and data transfer between tasks, the network latency between users and cloud resources is trivial for scheduling scientific workflows. Some works on workflow scheduling explicitly recognized the budget control as the key constraint [25], [29], [35], [36]. Being associated with strong commercial concerns, budgetary control is much more important for deploying business workflows [16].

In [29], Budget-constrained Heterogeneous Earliest Finish Time (BHEFT) was proposed as an extension of the Heterogeneous Earliest Finish Time (HEFT) [37] algorithm for budget-constrained workflow scheduling. BHEFT first calculates the priorities of all tasks in applications according to their average execution time and data transfer time among different resources, then allocates the total budget for each task with regard to spare application budget, current task budget, and adjustment factor in order of the priority. If no adequate solution that satisfies the budget constraint can be discovered, we consider the result as mapping failure.

Another category of approaches for budget-constrained workflow scheduling is based on the back-tracking heuristic, e.g., [35], [36]. These algorithms first generate a good initial schedule on the workflow in term of one optimization objective, e.g., the minimal execution time, then the chosen tasks are gradually migrated to downgraded resources in an attempt to meet the budgetary requirement. However, these search-based strategies can easily get trapped in local optima and exhibits inferior performance [38].

Meta-heuristic methods also have been used for budget-constrained workflow scheduling, e.g., the GA-based approach in [25]. A typical GA requires a genetic representation of the solutions with the help of some encoding methods. Each encoded solution, i.e., chromosome, is associated with a fitness value based on a fitness function, and the fitness value of each individual solution is an indication of its chances of survival and reproduction in the next generation. During the process of evolution, three basic GA operations, i.e., selection, crossover, and mutation, will be performed repeatedly until any given stopping condition is met. To deal with the budget constraint, Yu *et al.* applied a penalized fitness function to evolve the budget-compliant schedule in [25]. Similarly, a coevolutionary GA with adaptive penalty function CGA^2 was

proposed in [24] to handle deadline-constrained workflow scheduling, i.e., to minimize total cost and meet the deadline. The crossover and mutation rates were co-evolved in CGA^2 to help convergence.

Except penalty functions, many other constraint-handling techniques have been proposed for evolutionary algorithms (EAs), such as special representations and operators, separation of constraints and objectives, and repair algorithms [27], [39]. Particularly, the repair algorithms aim to transform an infeasible solution into a feasible one during the evolutionary process. The problem-dependent repair algorithms have been widely demonstrated with strong ability to handle constraints both effectively and efficiently on a diverse range of constrained-combinatorial optimization problems [27]. However, the computation time required for repairing the solution is usually higher than other techniques [26]. Therefore, an efficient repair algorithm is needed to control the repair time. In our work, we apply the proportion of feasible individuals in the current population as a self-adaptive factor to dynamically change the repair upper bound.

From our survey of the existing approaches, we choose BHEFT [29] and the penalty-based GA [25] to benchmark the performance of our approach. As the leading heuristic and meta-heuristic methods for budget-constrained workflow scheduling [38], the two approaches can be easily adapted for our problem. We also experimentally compare our new algorithm with CGA^2 [24], which is a representative algorithm with the state-of-the-art performance on deadline-constrained workflow scheduling problems.

TABLE 1
Mathematical Notations

Notation	Definition
W_a	The ath composite application
s_l	The lth application service in composite applications
S_a	The set of services in the ath composite application
E_a	The set of directed edges in the ath composite application
e_{mn}	The directed edge from the mth service to the nth service
$start_a$	The starting service of application W_a
end_a	The ending service of application W_a
U_k	The kth user centre
V_i	The ith type of VM
A_j	The jth data centre location
v_{ij}	The cheapest VM instance that belongs to the ith type of VM in the jth data centre location
c_{ij}	The cost of v_{ij}
λ_{ak}	The request rate of W_a from the kth user centre
θ_a	The workload of application W_a
β_l	The workload of application service s_l
α_l	Processing capacity of the lth service
pt_l	The average processing time of the lth service
dt_l	Data access delay of the lth service
It_{ak}	Network latency from the kth user centre to starting service of the ath application
Ot_{ak}	Network latency from ending service of the ath application to the kth user centre
ut_a	Average network latency between user centres and the ath application
st_{mn}	Network latency from s_m to s_n
b	The specified budget by the application provider
k	The budget factor

3 Problem Description

In this paper, we study the composite application deployment problem in multi-cloud, which selects VMs for a set of composite applications to minimize the application response time within a given budget. In this section we first define composite applications and introduce multi-cloud application deployment systems, then formulate the composite application deployment problem in multi-cloud. The key notations to be used for problem definition are listed in Table 1.

3.1 Composite Application

An application provider provides a set of composite applications $\mathcal{W} = \{W_0, \ldots, W_a, \ldots, W_{|\mathcal{W}|-1}\}$ that jointly use a collection of application services $\mathcal{S} = \{s_0, \ldots, s_l, \ldots, s_{|\mathcal{S}|-1}\}$. We represent each application W_a as a Directed Acyclic Graph (DAG), denoted by $G(S_a, E_a)$. $S_a \subseteq \mathcal{S}$ is a set of nodes representing services. Every service s_l might be shared by two or more composite applications. E_a is a set of directed edges where $e_{mn} \in E_a$ connects s_m with s_n. Based on e_{mn}, s_m is called the parent service of s_n and s_n is the child service of s_m. In application W_a, service $s_l \in S_a$ may have more than one parent services and child services, we denote the set of the parent services of s_l as $Pre(s_l)$, and the set of the child services of s_l as $Succ(s_l)$. If a service s_l requires access to any dataset during processing, we denote this service as $\dot{s_l}$. Each application W_a has exactly one dummy starting service that has no parent services, and one dummy ending service that has no child services, which are denoted as $start_a$ and end_a respectively. Three example composite applications are shown in Fig. 1. In the example, service s_1 is shared by applications W_0 and W_1, while $\dot{s_5}$ involves all three applications and is also a data-access service (represented in blue circles).

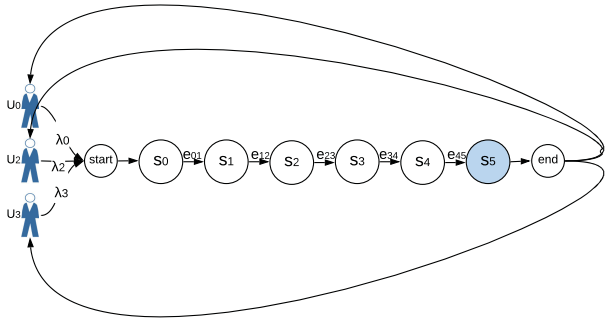
We refer to [15] and consider a set of user centres $\mathcal{U} = \{U_0, \ldots, U_k, \ldots, U_{|\mathcal{U}|-1}\}$, which represent the centres of the global user groups. We assume that new application requests from user centres are generated according to a Poisson distribution, following the common practice in many previous works [40], [41]. The request rates change periodically, e.g., on an hourly basis. Within each hour, the request rate λ_{ak} denotes the request frequency from a user centre U_k to the starting service of the application W_a. In this regard, the total workload of W_a can be determined periodically as

$$\theta_a = \sum_{k=0}^{|\mathcal{U}|-1} \lambda_{ak}. \tag{1}$$

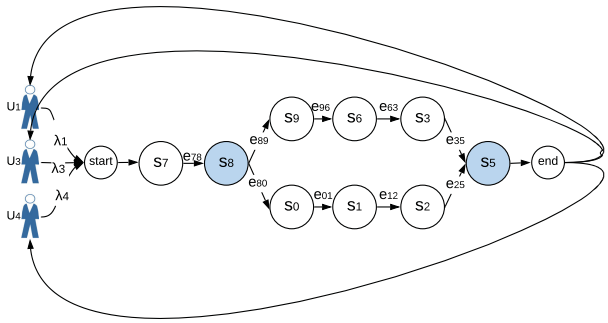
3.2 Multi-Cloud Application Deployment System

Application providers create their composite applications by composing a set of application services with specific functionalities. These services can be deployed onto a set of VMs in multi-cloud with the help of a broker (shown in Fig. 2).

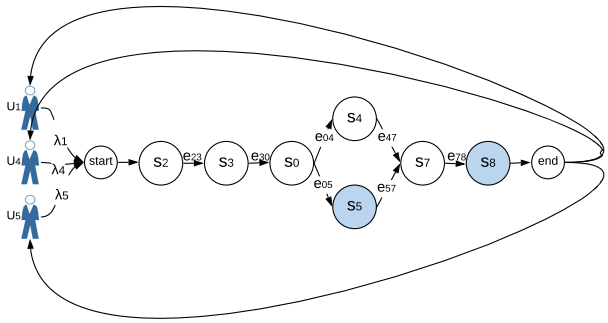
We consider a set of VM types $\mathcal{V} = \{V_0, \ldots, V_i, \ldots, V_{|\mathcal{V}|-1}\}$ and a collection of data centre locations $\mathcal{A} = \{A_0, \ldots, A_j, \ldots, A_{|\mathcal{A}|-1}\}$. Although individual cloud providers cannot support all VM types in each location, application providers still can find the cheapest VM type among all available VM types $V_i \in \mathcal{V}$ in the data centre location $A_j \in \mathcal{A}$. We use v_{ij} to denote this VM type and c_{ij} to denote its cost.



(a) W_0



(b) W_1



(c) W_2

Fig. 1. Three example composite applications for deployment.

We assume each application service is deployed to one VM instance. This is a common assumption in the literature [8], [30] and widely exercised in the industry [42]. We further assume that one VM instance is only allocated to one application service in order to minimize interferences between any two services. Each service instance maintains an independent queue for pending requests arriving from all user centres or its parent services in all deployed applications. Following [8], the average resource consumption per request over a long sequence of requests for the same service is considered highly stable, while different services can incur different resource requirements. We follow [43] and model the operation of each individual service instance as an $M/M/1$ queue. Because an application service can be shared by multiple applications, the aggregated workload of service s_l can be calculated by

$$\beta_l = \sum_{a=0}^{|\mathcal{W}|-1} \theta_a \qquad \text{s.t.} \quad s_l \in S_a. \tag{2}$$

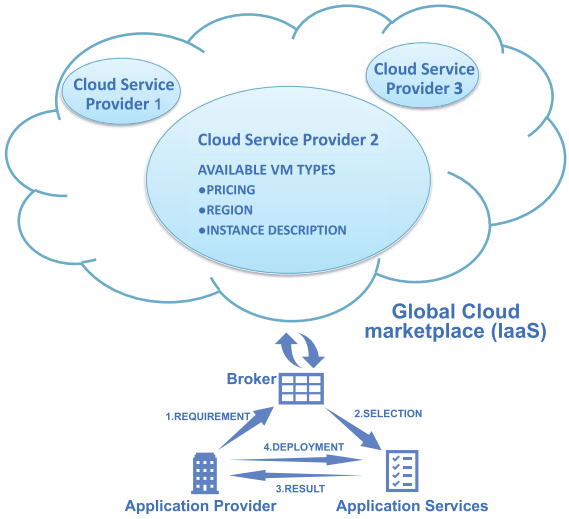


Fig. 2. A simplified multi-cloud broker architecture based on [20].

For data-access service $\dot{s}_l$, we use dt_l to denote the data access time between service $\dot{s}_l$ and the closest dataset deployed either by the application provider or a third party. We assume that request processing starts from data access, i.e., once the requested data becomes available, the service deployed to a VM instance will perform any necessary processing on the data. Considering the data access, the adjusted capacity of a VM instance with capacity α_l can be computed by

$$\widetilde{\alpha_l} = \frac{1}{\frac{1}{\alpha_l} + dt_l}. \tag{3}$$

According to Little's Law [44], the average request processing time of service s_l is

$$pt_l = \begin{cases} \frac{1}{\widetilde{\alpha_l} - \beta_l} & \text{if } \dot{s}_l \\ \frac{1}{\alpha_l - \beta_l} & \text{otherwise.} \end{cases} \tag{4}$$

We do not take the data access cost into account in the overall budget constraint, because the cost is usually negligible compared with the service deployment cost.[5] We also have an operation constraint, i.e., $\alpha_l(\widetilde{\alpha_l}) > \beta_l$, that guarantees that the workload of the VM instance will not exceed its capacity. In the remaining of this paper, we use *capacity feasible* VMs to refer the VM types whose capacities are greater than their workloads.

We define the average response time of an application ART_a as the total time on average required for W_a involving a workflow of service instances to process all user requests. Since a user centre $U_k \in \mathcal{U}$ and a data centre location $A_j \in \mathcal{A}$ can span large geographic areas, we must take network latency into account when computing ART_a. The *average* network latency ut_a between user centres and the application W_a can be determined by

$$ut_a = \frac{\sum_{k=0}^{|\mathcal{U}|-1} \lambda_{ak}(It_{ak} + Ot_{ak})}{\theta_a}, \tag{5}$$

5. https://azure.microsoft.com/en-us/pricing/details/bandwidth/

where It_{ak} is the network latency between a user centre U_k and the starting service of W_a, while Ot_{ak} is the network latency between the ending service of W_a and user centres U_k. Besides, we denote the network latency between service s_m and its child service s_n as st_{mn}. The aforementioned three types of network latency, i.e., dt_l, ut_a, and st_{mn}, are usually determined by several non-changeable factors in the communication network, such as the physical distance among datasets, user centres, and services. Accordingly, the ART_a can be calculated as follows:

$$ART_a = ut_a + MS(W_a), \tag{6}$$

where function MS calculates the response time within the workflow W_a (also called *makespan*). We define two functions EST (Earliest Start Time) and FT (Finish Time) for each service to calculate the application makespan. The computing process begins with the starting service, and the MS is the finish time of the ending service

$$\begin{aligned} &EST(start_a) = 0, \\ &FT(s_l) = EST(s_l) + pt_l, \\ &EST(s_l) = \max_{s_m \in Pre(s_l)} \{FT(s_m) + st_{ml}\}, \\ &MS(W_a) = FT(end_a). \end{aligned} \tag{7}$$

For example, based on the above formulas, we can compute ART of W_2 in Fig. 1 as follow:

$$\begin{aligned} ART_2 = {} & ut_2 + pt_2 + st_{23} + pt_3 + st_{30} + pt_0 \\ & + \max\{(st_{04} + pt_4 + st_{47}), (st_{05} + pt_5 + st_{57})\} \\ & + pt_7 + st_{78} + pt_8. \end{aligned}$$

3.3 Service Deployment Problem

We use a service deployment one-dimensional vector $X = \{v_{ij}^l\}_{l=0}^{|\mathcal{S}|-1}$ to represent the application deployment plan, where v_{ij}^l indicates that a service s_l is deployed to one instance of VM type V_i in the data centre location A_j. That is, the deployment cost of s_l, i.e., DC_l, is c_{ij}. The total deployment cost (TDC) and total response time (TRT) can be calculated as follows:

$$TDC = \sum_{l=0}^{|\mathcal{S}|-1} DC_l \tag{8}$$

$$TRT = \frac{\sum_{a=0}^{|\mathcal{W}|-1} \theta_a \cdot ART_a}{\sum_{a=0}^{|\mathcal{W}|-1} \theta_a}. \tag{9}$$

Therefore, the budget-constrained application deployment problem is to minimize TRT of all the composite applications, as defined in formula (9), subject to the budgetary constraint

$$\begin{aligned} \min_{X} \quad & TRT \\ \text{subject to} \quad & TDC \leq b, \end{aligned} \tag{10}$$

where b is the budget specified by the application provider. We use *budget feasible* or *feasible* solutions to refer solutions satisfying budgetary constraints in the remaining of this paper. We refer to [35] and define b in Eq. (11):

$$b = C_{cheapest} + k \cdot (C_{highest} - C_{cheapest}), \tag{11}$$

where $C_{highest}$ and $C_{cheapest}$ are the total costs of deployments by using either the most expensive VM type or the cheapest capacity feasible VM types for all the services respectively. $k \in [0, 1]$ is the budget factor to determine how tight the budgetary control is. The larger k is, the more budget the application provider has.

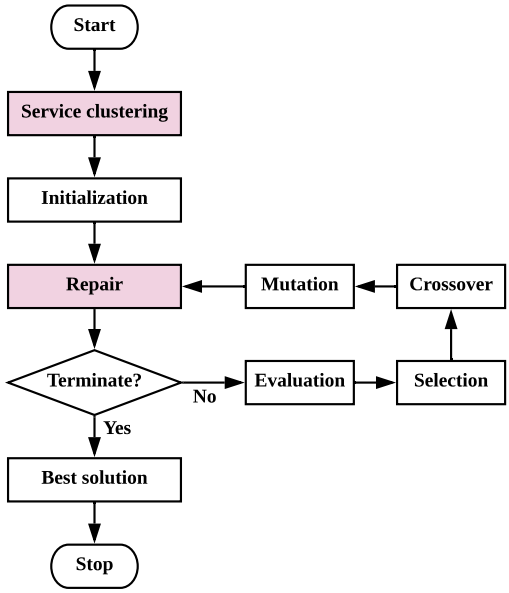


Fig. 3. The framework of the hybrid GA-based approach.

4 A Hybrid GA-Based Approach

In multi-cloud environment, there are a large number of different VM types available at different data centre locations, resulting in extremely large search spaces (theoretical size is $(|\mathcal{V}| \cdot |\mathcal{A}|)^{|\mathcal{S}|}$). We first propose a clustering method to group relevant services and deploy the services in the same cluster to the same location. This clustering technique can substantially reduce the size of the search space. In recent years, repair algorithms have become relatively promising solutions to handle constraints in EAs [27]. These techniques demonstrate promising performance for various combinatorial optimization problems thanks to problem-specific heuristics [26]. However, when using repair algorithms to solve constrained optimization problems, computation time is one of the critical issues. Consequently, we propose a repair algorithm to downgrade the non-critical services for the over-budget solutions within an adaptive bound to improve efficiency without hurting performance.

Fig. 3 shows the overall process of the proposed hybrid GA-based approach. After service clustering and encoding, an initial population is generated based on the greedy and the BHEFT algorithm [29]. In each generation, our algorithm first repairs the population within an adaptive upper bound, then evaluates the fitness of the repaired solutions. Finally, the problem-specific selection, crossover and mutation operations are performed to generate a new population. The population is evolved iteratively until the predefined

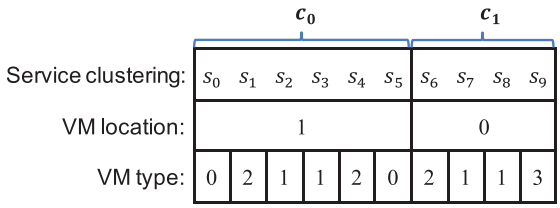


Fig. 4. An example chromosome for encoding deployment solutions.

maximum number of generations is reached. Finally, the best chromosome is decoded to derive the final application deployment plan.

In the following subsections, we will introduce in detail the service clustering, encoding/decoding scheme, fitness evaluation, problem-specific population initialization, repair algorithm, and genetic operators.

4.1 Service Clustering

Regarding the VM location selection problem, if the application services are considered separately, the size of the search space is $|\mathcal{A}|^{|\mathcal{S}|}$ (maximum value is 15^{40} in our experiments). It is essential to reduce the search space in order to guarantee algorithm scalability. Inspired by [45] that recommends execution of pipeline tasks on the same site to save data transfer time, we decide to group *dependent services* and deploy the services in the same cluster to the same location to reduce network latency. A straightforward approach to determine the dependency of a service on any application is to compare the request rates experienced by the same service from different applications.

In practice, we regard each application as the centroid of a separate service cluster. A nearest centroid classifier [46] can hence be applied to classify individual services according to their application request rates. Concretely, if a service is shared by multiple applications, we assign it to the cluster c_a where application W_a has the highest request rate. By this means, we can guarantee that the services in the same cluster will communicate more frequently than services from different clusters. For the service that has the equal request rate with respect to more than one applications, we simply assign it to the cluster with the highest number of services. By doing so, the communication among a larger number of services can be easily localized to the same data centre, reducing the networking overhead.

Algorithm 1. Service Clustering Algorithm

Input: $\mathcal{W}$, $\mathcal{S}$
Output: clustered services
1: **for all** $s_l \in \mathcal{S}$ **do**
2: find the highest request rate $\theta_{highest}$ among applications $\mathcal{W}'$ that include s_l
3: **if** $\mathcal{W}' = \{W_a\}$ **then**
4: assign s_l to the cluster c_a
5: **else**
6: find the application $W_a \in \mathcal{W}'$ with the most services and assign s_l to the cluster c_a
7: **end if**
8: **end for**

After service clustering, the size of the search space is reduced to $|\mathcal{A}|^{|\mathcal{W}|}$. In other words, the size of search space after service clustering is $|\mathcal{A}|^{|\mathcal{S}|-|\mathcal{W}|}$ times less than that of the original problem without clustering. More details about the clustering algorithm can be found in Algorithm 1. Assume that the request rates of the three applications in Fig. 1 are 20, 15, and 10 per second, the clustering algorithm will group 10 services into two clusters, i.e., c_0 and c_1 as demonstrated in Fig. 4.

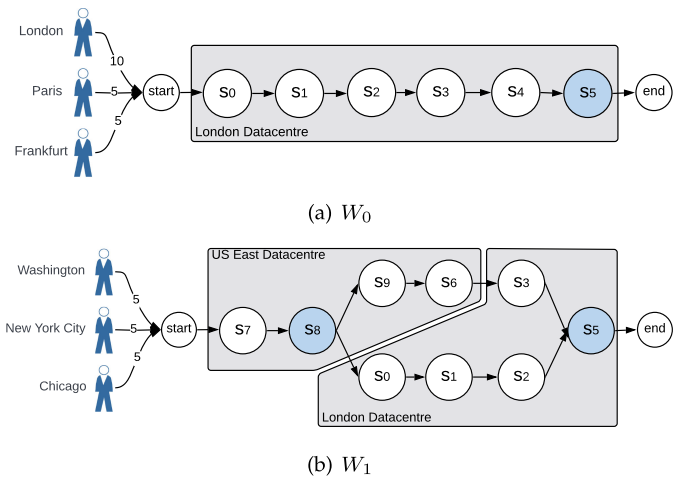


Fig. 5. A VM location deployment example with request distribution.

It is important to note that service clustering can effectively reduce network latency across all applications. Refer to the example shown in Fig. 1. For application W_0, all its services in c_0 will be deployed to the same data centre, e.g., AWS London shown in Fig. 5a. In this case, the time required for inter-service communication is_0 is negligible. According to the application request distribution in Fig. 5a, the average network latency between users and W_0 can be calculated according to Eq. (5)

$$\begin{aligned} ut_0 &= (10 \cdot (2+2) + 5 \cdot (7+7) + 5 \cdot (17+17))/20 \\ &= 14(\text{ms}). \end{aligned}$$

On the other hand, for other applications such as W_1 with its services grouped into more than one clusters, we can still effectively control the inter-service network latency. In particular, if all the users of W_1 are in America as shown in Fig. 5b, and we still allocate cluster c_0 to AWS London but cluster c_1 to AWS US East, then the average network latency between users and W_1 can be calculated by

$$\begin{aligned} ut_1 &= (5 \cdot (12+92) + 5 \cdot (12+92) + 5 \cdot (8+88))/15 \\ &\doteq 101.33(\text{ms}). \end{aligned}$$

The inter-service network latency is_1 is about 80 ms (from US East to London). This is clearly better than the alternative situation if we allocate all services of W_1 to AWS US East. In that case, we can ignore the inter-service network latency of W_0 and W_1, i.e., $\Delta is_0 = is'_0 - is_0 = 0$ ms and $\Delta is_1 = is'_1 - is_1 = -80$ ms, however, the average network latency between users and the two applications will change to

$$\begin{aligned} ut'_0 &= (10 \cdot (82+82) + 5 \cdot (87+87) + 5 \cdot (97+97))/20 \\ &= 174(\text{ms}) \\ ut'_1 &= (5 \cdot (12+12) + 5 \cdot (12+12) + 5 \cdot (8+8))/15 \\ &\doteq 21.33(\text{ms}). \end{aligned}$$

That is, $\Delta ut_0 = 160$ ms and $\Delta ut_1 = -80$ ms, and $\Delta ART_0 = \Delta ut_0 + \Delta is_0 = 160$ ms and $\Delta ART_1 = \Delta ut_1 + \Delta is_1 = -160$ ms. Note that application W_0 is much more frequently requested by end users than application W_1. This results in longer TRT than the original deployment plan according to Eq. (9).

4.2 Chromosome Representation

We use chromosomes to encode service deployment solutions. We split a chromosome into two strings, i.e., *VM location* and *VM type*, to represent the mappings from services to data centre locations and VM types respectively. Based on the clustering result, the services in the same cluster are treated as a single deployment unit in any *VM location* string.

Fig. 4 gives the encoding of a possible deployment for the applications in Fig. 1, this encoded solution (aka. individual in the GA population) gives a complete deployment plan, according to which service s_0 is deployed to a VM instance of type V_0 at location A_1, and s_1 is deployed to the VM instance of V_2 at the same location A_1, and so on.

4.3 Fitness Evaluation

For each chromosome, we evaluate its fitness by calculating TRT based on Eq. (9). We also calculate TDC to determine the necessity of applying the repair process to enforce budgetary control. Our proposed repair algorithm is described in Section 4.5.

4.4 Initial Population

Generally, GAs generate the initial population randomly to ensure its diversity. To improve the solution quality and convergence speed, a portion of the initial population will be determined based on the greedy heuristic (VM locations selection) and BHEFT [29] (VM types selection).

For the *VM location* string, we apply a greedy method to find the data centre location with minimal network latency between the users and service clusters. The greedy heuristic is presented in Algorithm 2. For each candidate data centre location, we calculate the network latency between it and all user centres (step 7). If there are data access services in this cluster, the data access delay is also added (step 12). Finally, the location of each service cluster with the minimal network latency is chosen to form the initial *VM location* string for subsequent evolution. In Fig. 4., for example, two clusters are deployed in data centre locations A_1 and A_0 respectively.

Next, we initialize the VM types based on BHEFT [29], which is an extension of the well-known DAG scheduling heuristic HEFT [37] by considering the budget constraint. BHEFT first distributes the total budget to tasks (application service in our problem) in the order of their priorities, then selects an appropriate service (VM instance in our problem) for tasks with regards to their separate budget.

For *each* composite application, we follow BHEFT to recursively calculate the rank of every service in this application, as defined by

$$rank_l = \overline{pt_l} + \max_{s_m \in Succ(s_l)} (st_{lm} + rank_m), \tag{12}$$

where $\overline{pt_l}$ is the average processing time for s_l on all capacity feasible VM types in the selected VM location, and st_{lm} is the network latency from s_l to s_m based on the selected VM locations. The calculation begins with the ending service whose rank equals to 0 (dummy service).

Algorithm 2. Greedy Heuristic for VM Locations Selection

Input: $\mathcal{A}$, $\mathcal{W}$, $\mathcal{U}$, λ_{ak}, It_{ak}, Ot_{ak}, service cluster set $\mathcal{C}$
Output: service cluster deployment location

```
1: for all c ∈ C do
2:   for all data centre location A_j do
3:     latency_j ← 0
4:     for all s_l in cluster c do
5:       if s_l connects to starting service or ending service of
         W_a directly then
6:         for all U_k ∈ U do
7:           latency_j ← latency_j + It_ak(Ot_ak) · λ_ak
8:         end for
9:       end if
10:      if s_l requires data access during processing then
11:        for all W_a include s_l do
12:          latency ← latency + dt_l · θ_a
13:        end for
14:      end if
15:    end for
16:  end for
17:  Return the data centre location with the minimal latency
18: end for
```

Because the services are shared by multiple composite applications, for each service we sum up all its rank values from corresponding applications as its comprehensive priority. Then we deploy the services in the order of the priority. The best possible VM instance is selected for each service based on spare application budget (SAB) and current service budget (CSB) as follows:

$$\begin{aligned} SAB_l &= b - \textstyle\sum_{i=0}^{l-1} DC_i - \sum_{j=l}^{|\mathcal{S}|-1} \overline{DC_j} \\ CSB_l &= \overline{DC_l} + SAB_l \cdot AF_l, \end{aligned} \tag{13}$$

where DC_i is the cost of the deployed service s_i, and $\overline{DC_j}$ is the average cost for the undeployed service s_j on all its capacity feasible VM types. The adjustment factor (AF) acts as a weight to tune the impact of SAB_l on CSB_l. Based on [29], AF is defined as the ratio between the average cost of the current service to the sum of the average costs of the undeployed services as follows:

$$AF_l = \begin{cases} \dfrac{\overline{DC_l}}{\sum_{j=l}^{|\mathcal{S}|-1} \overline{DC_j}} & \text{if } SAB_l \geq 0 \\ 0 & \text{otherwise.} \end{cases} \tag{14}$$

The rest of the population will be initialized randomly. On the one hand, we randomly select the VM locations, but ensuring that services in the same cluster are located in the same data center. On the other hand, services are deployed to *available* and capacity feasible VM types randomly, where we can find them in the corresponding location and their capacities are greater than their workload.

4.5 Repair Algorithm

The solutions randomly generated at the initial stage and evolved during generations may violate the given budget constraint. Therefore, to make the infeasible solutions

constraint-compliant we use a repair operator. In particular, our repair operator aims to reduce the total deployment cost (TDC) step by step to be within the given budget.

To reduce the TDC of the given set of applications, for some of their services we can either select the same type of VM but at a different location with a cheaper price or select a smaller VM at the same location. For example, we can change the location of a service from London to US East, which offers cheaper services than at London, or downgrade the VM type to one with lower capacity, e.g., from AWS m5.large to m5. samll. Based on the coding scheme introduced in Section 4.2, however, the change of VM locations will affect all the services in the same cluster and cause huge fluctuation of the total response time (TRT). Consequently, we design our repair operator to downgrade VM types for some selected services while simultaneously evolving VM locations selection through GA operations.

Our repair operator is based on the concept of critical-path [37], which determines the overall response time of an application. Inspired by the work in [37], which proposes critical-path based heuristics to optimize the makespan of workflows, we propose to downgrade VM types for services that are not on any critical path of composite applications to reduce TDC.

The tasks on a critical path are known as critical tasks, whose execution time determines the makespan of workflows. Inspired by the idea, we use *non-critical service* to denote a service that is not on any critical path of the given composite application. To reduce the overall cost without increasing TRT, our repair operator first identifies non-critical services and then downgrades VM types with a lower but feasible capacity to save operational cost. For example, for a non-critical service deployed to AWS m5.xlarge, our repair operator changes the VM type to m5.large, without or with little impact on TRT. By identifying and downgrading non-critical services iteratively, the repair operator improves over-budget solutions until the TDC falls within the budget.

Algorithm 3. Repair Algorithm

Input:current population P, b, $\mathcal{V}$, $\mathcal{S}$
Output:repaired population P'

```
Calculate C_u based on Eq. (15) and determine the best feasible solution best in terms of TRT
for all ind ∈ P do
  if TDC_ind > C_u then
    Replace ind with best
  else
    while TDC_ind > b do
      for all s_l ∈ S do
        Identify the non-critical service and downgrade its VM type with a lower but feasible capacity
      end for
    end while
    Return the repaired P'
  end if
end for
```

Note that repairing infeasible application deployment solutions can be computationally expensive due to number of possible ways to repair, and therefore impacts the overall efficiency of our GA-based approach. Replacing infeasible solutions with existing feasible solutions is an efficient strategy. However, this approach can decrease the diversity of the population and therefore affect the algorithm effectiveness. Therefore, we should repair a sufficient number of infeasible solutions to maintain the diversity of the populations of GA. To achieve a desirable trade-off between computation time and algorithm effectiveness, we set an adaptive upper bound C_u on TDC. With C_u changing dynamically during the evolution process, only the newly evolved solutions with its TDC falling in between a given budget b and C_u will be repaired. The adaptive C_u is calculated as follows:

$$\begin{aligned} C_u &= b + (C_{max} - b) \cdot (1 - r_f) \\ r_f &= \frac{N_f}{N}, \end{aligned} \tag{15}$$

where C_{max} is the largest cost with respect to any solution in the current population, N_f is the number of feasible solutions in the current population, and N is the population size. In this way, C_u decreases whenever the proportion of feasible solutions r_f is increased.

If we use the probability density f_{TDC} to illustrate the distribution of the TDC among all solutions in the population, r_f can be presented by the cumulative distribution function $F_{TDC}(x)$ as $F(b)$. Similarly, the proportion of feasible solutions after repair can be denoted as $F(C_u)$. We assume that f_X is monotonically decreasing within the interval $[b, C_{max}]$, therefore $F(b)$ is convex between b and C_{max}. Then we can easily get

$$\frac{F(C_u) - F(b)}{C_u - b} \geq \frac{F(C_{max}) - F(b)}{C_{max} - b},$$

where $F(C_{max}) = 1$. According to Eq. (15), we can conclude as follows:

$$\begin{aligned} F(C_u) - F(b) &\geq \frac{1 - F(b)}{C_{max} - b} \cdot (C_{max} - b) \cdot (1 - F(b)) \\ F(C_u) - F(b) &\geq (1 - F(b))^2 \\ F(C_u) &\geq (F(b) - 0.5)^2 + 0.75, \end{aligned}$$

where $F(C_u) \in [0.75, 1]$, that is, the repair algorithm with adaptive bound will guarantee at least 75 percent feasible solutions in the current population. The proportion can usually ensure good performance in GAs [47]. For the solutions with a cost higher than C_u in each generation, we simply replace them with the best feasible solution in terms of TRT in the current population. The repair algorithm is described in Algorithm 3.

4.6 Genetic Operators

We apply the roulette-wheel selection to choose individuals for breeding. This selection method is the most commonly used selection method for GAs [48]. In this section we present our problem-specific crossover operator and mutation operator as follows.

4.6.1 Crossover

The crossover operator is designed differently for *VM location* and *VM type* strings. For the *VM location* string, we pick up a *cluster* from both parents with equal probability to

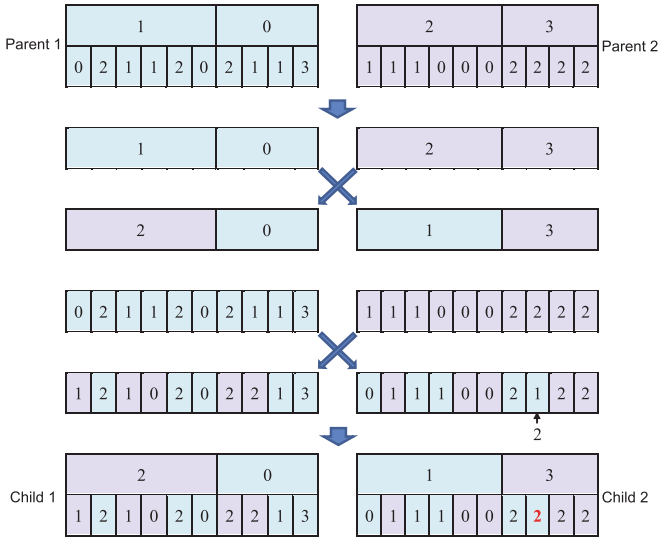


Fig. 6. An example of the crossover operator.

generate offsprings. For the *VM type* string, each *digit* of offsprings is chosen from parents with equal probability. This uniform crossover can be seen as a generalization of the more traditional one- and two-point crossover operators [49]. An example of this operation is given in Fig. 6, in which the offsprings from two crossover operations are combined into new chromosomes. Note that the combination may generate an *unavailable solution* with unavailable VM types in the new location. In this case, we modify the solutions to be available by selecting an available VM types with the most similar capacity. For example, the available V_2 in A_3 (in red) is chosen for the second offspring in Fig. 6.

4.6.2 Mutation

The mutation operator can introduce local improvement to previously evolved solutions by changing an offspring slightly in random. Similar to the crossover operator, our mutation operator mutates the strings of *VM location* and *VM type* separately. For *VM location* string, the mutation operator randomly selects a service cluster and randomly replaces it with a different data centre location. Then the availability of VM types in the new location is checked and modified in a similar fashion as described in Section 4.6.1. For the *VM type* string, a digit is selected randomly and updated with a different, available, and capacity feasible VM type. Fig. 7 demonstrates an example where the first cluster and service s_6 are randomly chosen and replaced.

5 Evaluation

In the absence of a publicly available global multi-cloud testbed, we conduct a series of simulation studies to examine the performance of our proposed hybrid GA-based approach (H-GA for convenience) by comparing it with three leading state-of-the-art algorithms for constrained application deployment problems, i.e., BHEFT [29], the penalty-based GA (P-GA for convenience) [25], and CGA^2 [24]. The P-GA algorithm formulates its penalized fitness function by introducing the extra cost-fitness defined by TDC/b. In other words, when the cost of an application deployment solution exceeds the given budget, the fitness of the solution

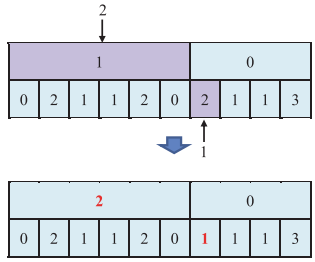


Fig. 7. An example of the mutation operator.

is penalized in terms of this cost-fitness. To adapt CGA^2 to our problem, we change its cost-related fitness value to TRT and time-related constraint violation to TDC.

5.1 Datasets

Based on the latest report regarding the worldwide IaaS public cloud services market share [1], we collect the real VM type descriptions and pricing schemes in May 2019 from three leading cloud providers, i.e., AWS,[6] Microsoft Azure[7] and Alibaba Elastic Compute Service (ECS).[8] 24 different VM types (8 from AWS, 8 from Azure, and 8 from Alibaba) have been included in the experiments. We also consider a total of 15 locations for major AWS, Azure and Alibaba data centres and 8 locations for hosting application-dependent datasets, i.e., San Francisco, Hong Kong, Sao Paulo, Mumbai, London, Tokyo, Frankfurt, and Sydney, which are their main data regions. Furthermore, we adopt 82 user centres from 35 countries on 6 continents in the Sprint IP Network[9] to simulate the global user community.

To evaluate the network latency between users/data and deployed services, and the network latency from parent services to their respective child services, we use real-world observations of network latency from Sprint[10] IP backbone network databases. The real-time delay time between two sites can be obtained easily by clicking the link between them in the SprintLink Network map.

5.2 Simulation Settings

In our experiments, we consider eight business workflow structures corresponding to diverse online activities such as online shopping and travel planning services based on [8] (see Fig. 8). The range of service processing time for single request is also obtained from [8], that is, 5-35 ms running on AWS t2.micro. Three scales of services, i.e., 20 (low), 30 (medium), and 40 (high) different application services are implemented to simulate the different business diversities. For different service diversities, we all generate 8 composite applications by randomly selecting workflow structures and services.

Refer to [50], we apply Facebook subscribers statistics to simulate the distribution of application requests from user centres λ_{ak}. By June 2017, there are approximately 1.33 billion Facebook subscribers from all of 35 countries considered in our simulation.[11] We consider the user scale for composite

6. https://aws.amazon.com/ec2/instance-types/
7. https://azure.microsoft.com/en-us/services/virtual-machines/
8. https://www.alibabacloud.com/product/ecs.
9. https://www.sprint.net/network_maps.php
10. https://www.sprint.net/performance/
11. https://www.internetworldstats.com/facebook.htm

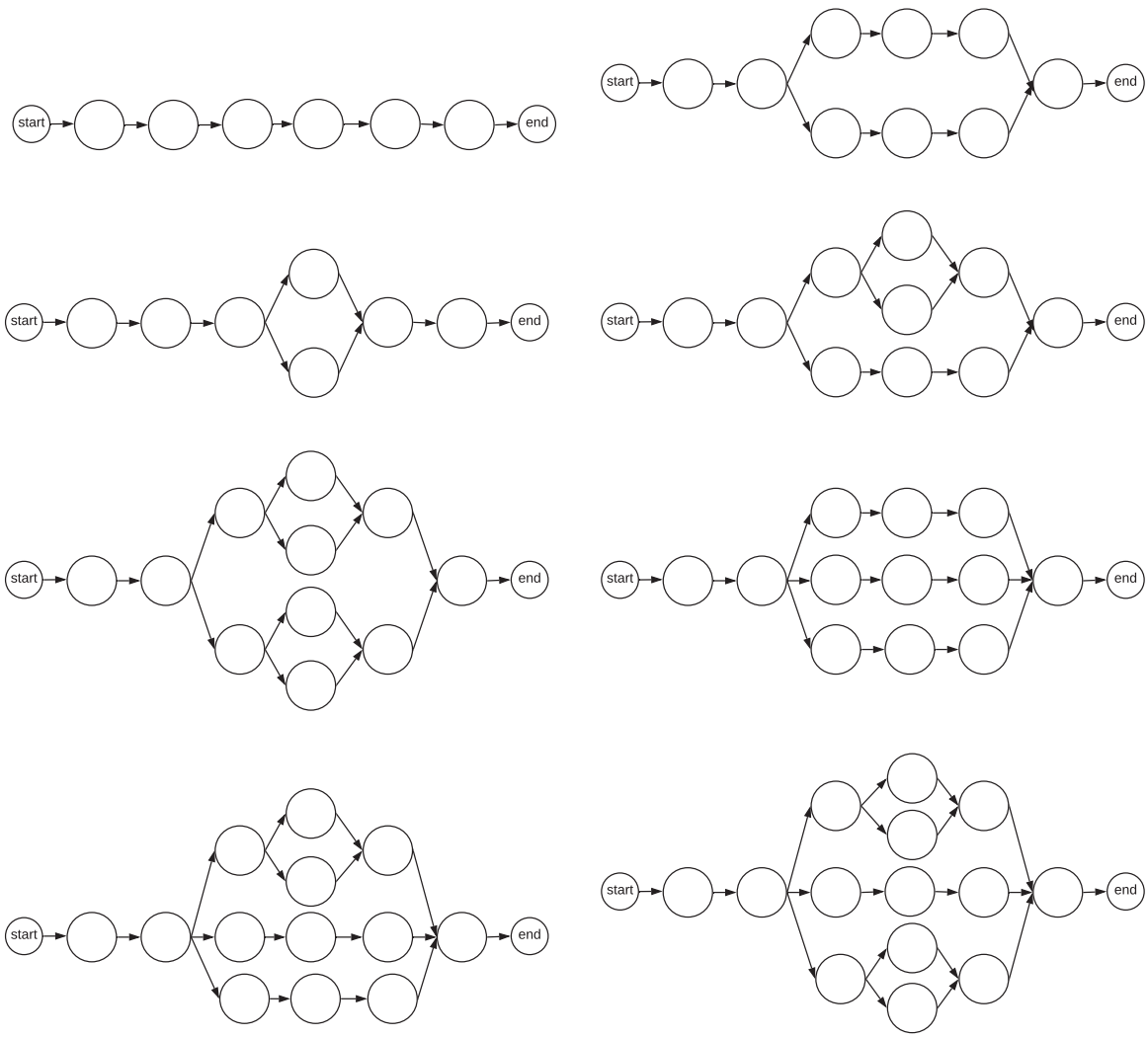


Fig. 8. The business application structures used in the experiments.

applications as 1/1000th of Facebook subscribers to simulate the application providers with million users, e.g., Xero.[12] For each composite application, we select 30 out of all 82 user centres randomly, and each user from the selected user centres makes 25 requests on average daily. Accordingly, the application request rate spans from 52 to 304 requests per second (that is, approximately 4.5-26.3 millions of requests daily). Refer to [38], we implement 6 budget factors, i.e., 0.1, 0.2, 0.3, 0.4, 0.7, and 0.9 for each set of composite applications.

5.3 Algorithm Parameter Settings

Because the BHEFT algorithm [29] does not support VM location selection, we choose VM instances from the most economical data centre to save cost. For P-GA approach, our parameter settings include: population size $N = 1024$, maximum generation $G = 1000$, which are sufficient to perform well. That is, we cannot gain significant improvements on the quality of the final solution by using more generations and larger populations. For CGA^2, we implement the same parameters with [24]: $M_1 = 200, G_1 = 100$ for searching deployment solutions, and $M_2 = 50, G_2 = 20$ for searching suitable crossover and mutation rates. For H-GA approach, our parameter settings include: population size $N = 100$, maximum generation $G = 1000$. Following [24], the proportion of greedy and BHEFT based solutions in the initial population is 20 percent. The crossover rate r_c and mutation rate r_m of P-GA and H-GA are 0.5 and 0.1 respectively following common practice in the literature [51]. To compare the results, we consider the mean and standard deviation of TRT and TDC after running each experiment 30 times. All the experiments are performed on the same computer with Intel Core i7-8700 CPU (3.2 GHz and 15.5 GB of RAM).

5.4 TDC Evaluation

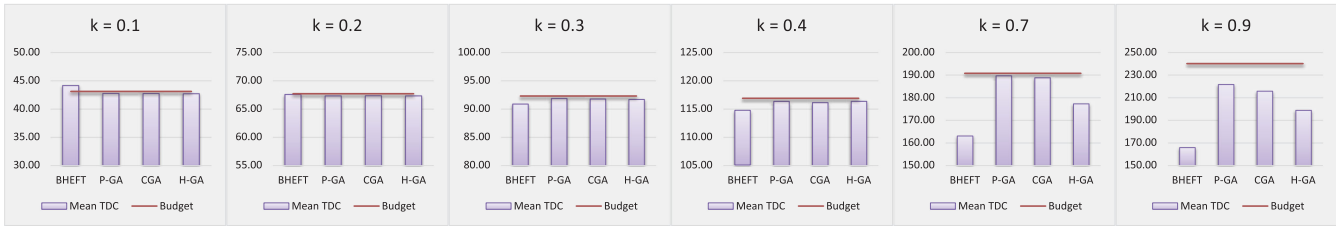
As shown in Fig. 9, we estimate the average TDC for composite applications with different service diversities under different budgets (red lines are the budget constraints).

We first analyze the algorithms in terms of their capabilities of meeting the pre-specified budget requirements. For the tightest budget, i.e., $k = 0.1$, BHEFT cannot generate adequate solutions for all service diversities. While as k increases, BHEFT is capable of deploying applications within budget, even for the composite applications with 40 different services (high diversity in Fig. 9c). Correspondingly, the three

12. https://www.xero.com/nz/

(a) The composite applications with low service diversity.

(b) The composite applications with medium service diversity.



(c) The composite applications with high service diversity.

Fig. 9. Average TDC under different budget constraints.

GA-based algorithms can meet all the budget levels for composite applications with all service diversities.

Regardless of TRT, we can see that the BHEFT approach has the lowest TDC under the majority of budget constraints ($k = 0.3, 0.4, 0.7, 0.9$ for all service diversities). P-GA or CGA^2 has the highest TDC under nearly all budget factors and service diversities. As to our proposed H-GA, it shows better cost savings than the penalty-based GA approaches, especially for the high budget factors, such as $k = 0.7, 0.9$. This is inline with the fact that H-GA deploys a portion of the services to the locations with cheaper VMs described in Section 5.6.

From above results, we can conclude that BHEFT can obtain the lowest TDC if the specific budget is not too low, while H-GA is able to guarantee adequate solutions with relatively low TDC.

5.5 TRT Evaluation

The TRT generated by the above algorithms for composite applications with different service diversities are shown in Fig. 10. Their mean and standard deviation of TRT and TDC with different service diversities and budget factors are presented in Table 2.

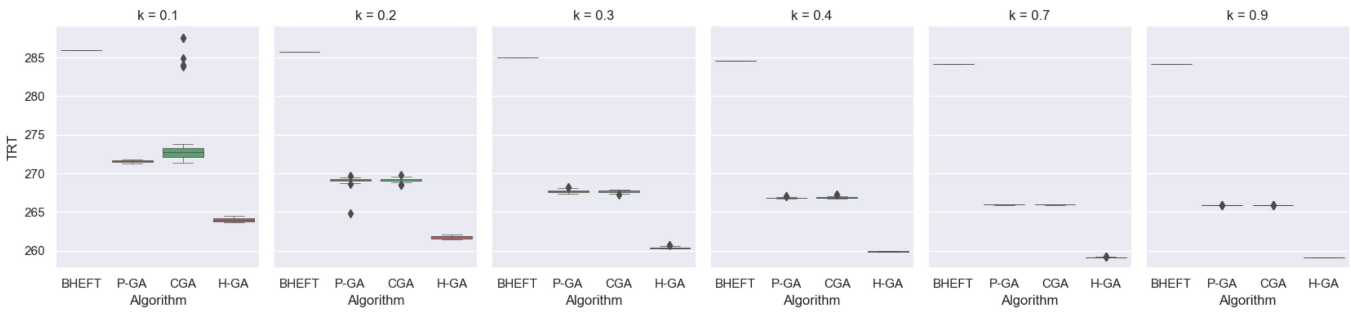
For composite applications with low and medium service diversities, the TRT achieved by the three GA-based algorithms is dramatically shorter than BHEFT at all the budget levels. In these experiments, H-GA outperforms both P-GA and CGA^2 for all cases, which shows that the optimization capacity of our proposed algorithm is better than the existing penalty-based methods. We can also find that CGA^2 has the similar performance as GA-P in the majority of cases. These results suggest that the adaptive penalty function in CGA^2 is not effective for our budget-constrained application deployment problem.

Compared with deploying composite applications with low and medium service diversities, when addressing high service diversity scenario, the performance improvement of P-GA and CGA^2 than BHEFT is very limited, especially when k is low. We attempt to increase the number of generations and fine-tune some parameters, without obtaining noticeably better performance. Correspondingly, H-GA can consistently outperform than BHEFT by optimizing the locations of service clusters.

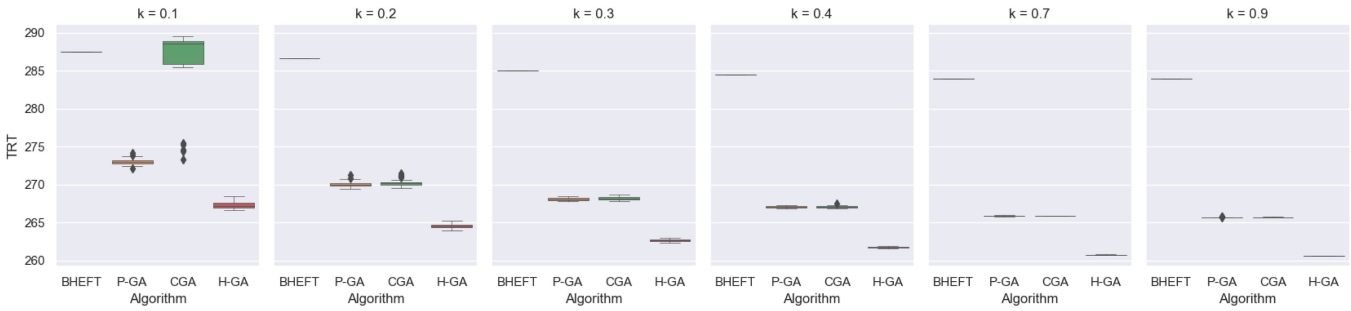
From Table 2, we can calculate the average improvement of H-GA in terms of TRT, 8.5 percent lower than BHEFT, 2.7 percent lower than P-GA, and 2.7 percent lower than CGA^2 for low service diversity, 7.8 percent lower than BHEFT, 2.0 percent lower than P-GA, and 2.8 percent lower than CGA^2 for medium service diversity, and 8.3 percent lower than BHEFT, 7.9 percent lower than P-GA, and 8.0 percent lower than CGA^2 for high service diversity. The observed performance differences between H-GA and three comparing approaches are verified through statistical test (Wilcoxon Rank-Sum test) with significance level of 0.1 for all cases. We also find that H-GA has the smaller standard deviation than the other two GA-based algorithms, confirming that H-GA can consistently find good deployment solutions.

5.6 Further Analysis

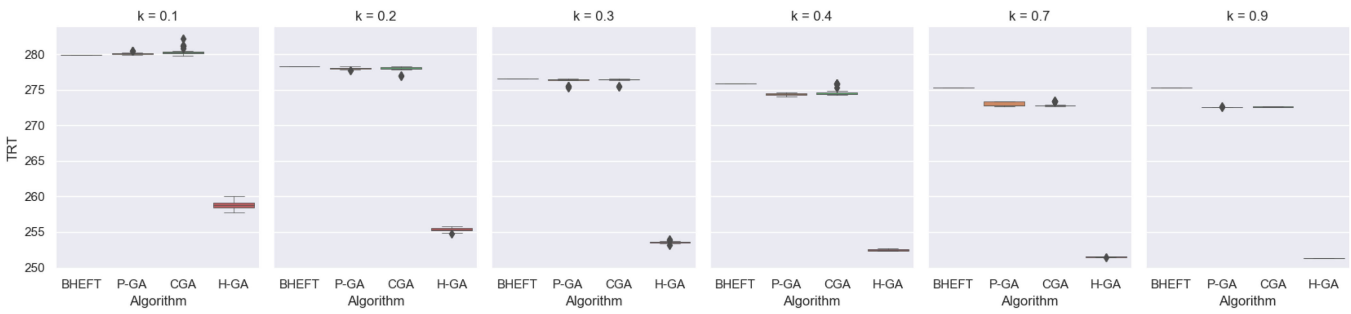
We observe that all the services are deployed to the same location in the final deployment solutions of P-GA and CGA^2,



(a) The composite applications with low service diversity.



(b) The composite applications with medium service diversity.



(c) The composite applications with high service diversity.

Fig. 10. TRT under different budget constraints.

because they cannot investigate inter-dependencies among services and optimize deployment locations only by evolutionary operations, i.e., selection, crossover, and mutation. On the contrary, H-GA tends to deploy the services to different locations (see the number of deployment locations, i.e., NoL in Table 2), and sometimes the locations in the final deployment solutions are different from the locations in the greedy-based initial population. This demonstrates that the service clustering mechanism plays an important role during the evolutionary search for good application deployment solutions.

Take applications with high diversity as example, a total of 40 services are deployed to four locations, i.e., Singapore, Los Angeles, Hong Kong, and London, in the final deployment solution of H-GA. On the one hand, some services are deployed to the cheaper VMs in Los Angeles, which can reduce TDC. On the other hand, selecting the four locations results in different service clusters close to the users, and therefore reduces the network latency of corresponding applications.

We also conduct further experiments to study the impact of request rate distribution on our system performance. For example, we have evaluated H-GA and other competing algorithms based on the real-word request rate distribution for business Web applications in [52]. Our results show that H-GA outperforms other algorithms consistently despite of changing request distributions, achieving up to around 8 percent shorter TRT and 100 percent budget satisfaction.

5.7 Computational Time

For the computational time, BHEFT can generate solutions quickly (within 0.1 seconds for all the test scenarios). The average computation time of H-GA with respect to all service diversities ranges from 75 to 200 seconds, which is slightly less than P-GA. Because CGA^2 involves the co-evolution of crossover and mutation rates, its average computation time is about 10 times longer than H-GA and P-GA (the overall evolution complexity is $O(M_1G_1M_2G_2)$).

To verify the effectiveness of the dynamic upper bound mechanism, we implement the algorithm that repairs all over-budget solutions. The algorithm cannot achieve better performance in terms of TRT, but spends on average 29.66 percent more average computation time (ACT shown in Table 3). The results illustrate that the mechanism of dynamic repair upper bound can trade off performance with computation time well.

TABLE 2
Algorithms Performance Comparison for Deploying Applications With Different Service Diversities and Budget Factors (TRT in ms. and TDC in USD)

Algorithm		BHEFT		P-GA		CGA^2		H-GA		
Diversity	k	TRT	TDC	TRT	TDC	TRT	TDC	TRT	TDC	NoL
Low	0.1	286.01	37.38	271.58±0.14	29.15±0.12	274.73±5.02	29.26±0.17	**263.99±0.23**	**29.25±0.26**	2
	0.2	285.75	42.24	269.01±0.81	42.18±0.24	269.14±0.27	42±0.47	**261.76±0.2**	**42.27±0.28**	
	0.3	285.01	55.04	267.72±0.21	55.28±0.41	267.69±0.18	55.41±0.26	**260.37±0.12**	**55.25±0.4**	
	0.4	284.64	67.33	266.85±0.07	68.37±0.22	266.88±0.13	68.37±0.28	**259.89±0.04**	**68.36±0.28**	
	0.7	284.12	92.16	265.96±0.03	107.38±0.31	265.97±0.02	107.31±0.41	**259.16±0.02**	**106.07±0.79**	
	0.9	284.12	92.16	265.83±0	126.53±0.59	265.83±0	126.42±0.8	**259.14±0**	**111.36±0**	
Medium	0.1	287.52	41.73	273±3.87	36.95±0.37	285.9±5.27	37.01±0.23	**267.26±0.43**	**36.99±0.19**	3
	0.2	286.57	57.47	2160.4±3.15	57.2±0.31	270.28±0.47	57.02±0.62	**264.53±0.29**	**57.23±0.28**	
	0.3	284.98	77.31	2144.82±1.43	77.49±0.33	268.18±0.24	77.58±0.32	**262.66±0.17**	**77.48±0.46**	
	0.4	284.51	97.15	2136.48±0.95	97.85±0.42	267.08±0.14	97.77±0.36	**261.73±0.07**	**97.84±0.39**	
	0.7	283.96	138.24	2127.1±0.29	158.7±0.66	265.89±0.03	158.85±0.44	**260.76±0.02**	**158.17±0.93**	
	0.9	283.96	138.24	2125.53±0.18	186.33±3.6	265.71±0.02	186.01±3.3	**260.65±0**	**174.72±0**	
High	0.1	279.92	44.16	280.1±0.13	42.77±0.17	280.39±0.49	42.75±0.2	**258.81±0.55**	**42.72±0.3**	4
	0.2	278.35	67.58	278.03±0.13	67.33±0.33	277.98±0.3	67.36±0.26	**255.35±0.24**	**67.33±0.25**	
	0.3	276.54	90.88	276.32±0.35	91.87±0.23	276.41±0.25	91.77±0.3	**253.56±0.16**	**91.7±0.38**	
	0.4	275.91	114.69	274.37±0.12	116.36±0.46	274.69±0.51	116.14±0.65	**252.49±0.11**	**116.37±0.52**	
	0.7	275.34	163.07	273.03±0.3	189.71±0.92	272.9±0.25	188.83±2.33	**251.51±0.05**	**177.3±5.51**	
	0.9	275.32	165.89	272.55±0.04	221.68±5.75	272.59±0.05	215.79±7.44	**251.36±0**	**198.86±0**	

Besides, as we can see, for the highest diversity with the tightest budget, H-GA can generate the solution within 13 minutes. That is, if the request rate distribution does not change significantly within an hour (as confirmed by many existing studies [53], [54]), periodical use of H-GA is highly feasible in practice, even considering the extra time required for application migration (previous research shows that service migrations in the cloud can usually be performed within 15 minutes [55]).

From the above discussion, the following conclusions can be made from the experiments: BHEFT can deploy composite applications in multi-cloud quickly but not effectively, also at the risk of over-budget. Comparing with BHEFT, GA-based algorithms are capable of generating adequate solutions and outperform BHEFT in terms of TRT. Our proposed algorithm H-GA also significantly outperforms P-GA and CGA^2. Especially when the budget is flexible (or sufficiently large), H-GA can generate solutions with less TDC and shorter TRT than the other two GA-based algorithms.

TABLE 3
Algorithms Performance Comparison With Different Repair Mechanisms (TRT in ms. and ACT in s.)

		Adaptive bound		Fully reapir		Δ	
Diversity	k	TRT	ACT	TRT	ACT	TRT	ACT
Low	0.1	263.99	247.98	263.96	351.57	0.01%	-29.46%
	0.2	261.76	106.11	261.76	197.09	0.00%	-46.16%
	0.3	260.37	44.43	260.42	98.84	-0.02%	-55.05%
	0.4	259.89	24.60	259.88	49.75	0.00%	-50.55%
	0.7	259.16	21.73	259.16	22.85	0.00%	-4.90%
	0.9	259.14	21.76	259.14	22.87	0.00%	-4.85%
Medium	0.1	267.26	596.21	267.48	713.6	-0.08%	-16.45%
	0.2	264.53	250.41	264.44	394.78	0.04%	-36.57%
	0.3	262.66	77.69	262.80	175.32	-0.05%	-55.69%
	0.4	261.73	28.98	261.69	65.67	0.01%	-55.87%
	0.7	260.76	24.80	260.78	26.04	-0.01%	-4.76%
	0.9	260.65	24.82	260.65	25.99	0.00%	-4.50%
High	0.1	258.81	765.05	258.72	884.65	0.03%	-13.52%
	0.2	255.35	281.10	255.44	454.53	-0.03%	-38.16%
	0.3	253.56	66.72	253.51	172.72	0.02%	-61.37%
	0.4	252.49	26.37	252.51	49.04	-0.01%	-46.23%
	0.7	251.51	25.71	251.51	27.33	0.00%	-5.93%
	0.9	251.36	25.69	251.36	26.72	0.00%	-3.85%

6 Conclusions and Discussion

In this paper, we studied the multi-cloud composite application deployment problem that considers the average response time to reflect the real performance of applications. To address the problem, we propose a novel GA-based algorithm which can select VMs from multi-cloud for constituent services of composite applications within a given cost budget so that the average response time of all composite applications is minimized. Due to the immense search space resulting from the diversity of VM locations and types, we design the service clustering that can effectively utilizes the information hidden in application structures. To improve solution quality and control computation time of the algorithm, we develop a problem-specific repair algorithm to transform a self-adaptive subset of over-budget solutions into budget feasible solutions. Furthermore, our proposed algorithm H-GA applies the two-string decoding and divide-and-conquer genetic operations, which are effective for breeding excellent offsprings. The experiments based on the datasets collected from real world cloud providers, network environment, and business applications show that our algorithm can meet the budget and achieve up to about 8 percent performance improvement in terms of TRT comparing with existing algorithms.

In this paper, we did not consider some new cloud paradigms, e.g., deploying multiple application services to multiple containers in the same VM instance. This is because the potential improvement in VM utilization may not be substantial. Particularly, based on the real-world datasets in our experiments, we found the average VM utilization is close to 90 percent under stringent budget (i.e., $k = 0.1$). Further improving the utilization beyond 90 percent can be difficult,

even after adopting the container technology. Moreover, previous queuing analysis showed that this could potentially lead to high queuing delay [56]. On the other hand, if we apply container technology, we need to handle two nested mappings, i.e., the mappings from services to containers and from containers to VMs. This will significantly increase the solution space, making it more difficult and computationally costly for H-GA to identify quality solutions. Consequently, we will study the container-based service deployment in our future research. Furthermore, we will evaluate H-GA on real-world multi-cloud platforms and develop multi-objective evolutionary algorithms to solve the composite application deployment problem.

Acknowledgments

This work was in part supported by the New Zealand Marsden Fund with the contract number (VUW1510), administrated by the Royal Society of New Zealand.

References

[1] C. Stamford, "Gartner forecasts worldwide public cloud revenue to grow 17% in 2020," 2019. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2019–11-13-gartner-forecasts-worldwide-public-cloud-revenue-to-grow-17-percent-in-2020

[2] L. Columbus, "83% of enterprise workloads will be in the cloud by 2020," 2018. [Online]. Available: https://www.forbes.com/sites/louiscolumbus/2018/01/07/83-of-enterprise-workloads-will-be-in-the-cloud-by-2020/

[3] N. Grozev and R. Buyya, "Inter-cloud architectures and application brokering: Taxonomy and survey," *Softw.: Practice Experience*, vol. 44, no. 3, pp. 369–390, 2014.

[4] D. Petcu, "Consuming resources and services from multiple clouds," *J. Grid Comput.*, vol. 12, no. 2, pp. 321–345, 2014.

[5] A. N. Toosi, R. N. Calheiros, and R. Buyya, "Interconnected cloud computing environments: Challenges, taxonomy, and survey," *ACM Comput. Surv.*, vol. 47, no. 1, pp. 7–53, 2014.

[6] A. J. Ferrer *et al.*, "OPTIMIS: A holistic approach to cloud service provisioning," *Future Gener. Comput. Syst.*, vol. 28, no. 1, pp. 66–77, 2012.

[7] X. Zhu, S. vanden Broucke, G. Zhu, J. Vanthienen, and B. Baesens, "Enabling flexible location-aware business process modeling and execution," *Decis. Support Syst.*, vol. 83, pp. 1–9, 2016.

[8] K. Huang and B. Shen, "Service deployment strategies for efficient execution of composite SaaS applications on cloud platform," *J. Syst. Softw.*, vol. 107, pp. 127–141, 2015.

[9] K. Isenberg, "IaaS vs. CaaS vs. PaaS vs. FaaS: Choosing the right platform," 2017. [Online]. Available: https://dzone.com/articles/iaas-vs-caas-vs-paas-vs-faas-choosing-the-right-platform

[10] R. Xu *et al.*, "A sufficient and necessary temporal violation handling point selection strategy in cloud workflow," *Future Gener. Comput. Syst.*, vol. 86, pp. 464–479, 2018.

[11] L. Wang and K. S. Trivedi, "Architecture-based reliability-sensitive criticality measure for fault-tolerance cloud applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 11, pp. 2408–2421, Nov. 2019.

[12] L. Heilig, R. Buyya, and S. Voß, "Location-aware brokering for consumers in multi-cloud computing environments," *J. Netw. Comput. Appl.*, vol. 95, pp. 79–93, 2017.

[13] T. Shi, H. Ma, and G. Chen, "A genetic-based approach to location-aware cloud service brokering in multi-cloud environment," in *Proc. IEEE Int. Conf. Services Comput.*, 2019, pp. 146–153.

[14] Z. Zhu, G. Zhang, M. Li, and X. Liu, "Evolutionary multi-objective workflow scheduling in cloud," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 5, pp. 1344–1357, May 2016.

[15] B. Tan, H. Ma, Y. Mei, and M. Zhang, "Evolutionary multi-objective optimization for web service location allocation problem," *IEEE Trans. Services Comput.*, to be published, doi: 10.1109/TSC.2018.2793266.

[16] S. Carter, N. J. MacDonald, and D. C. Cheng, *Basic Finance for Marketers*, vol. 1. Rome, Italy: Food & Agriculture Org., 1997.

[17] N. Sabharwal and P. Wali, "Cloud capacity management," in *Cloud Capacity Management*. Berlin, Germany: Springer, 2013, pp. 35–54.

[18] L. Davis, *Handbook of Genetic Algorithms*," New York, NY, USA: Van Nostrand, 1991.

[19] D. Whitley, "A genetic algorithm tutorial," *Statist. Comput.*, vol. 4, no. 2, pp. 65–85, 1994.

[20] L. Heilig, E. Lalla-Ruiz, and S. Voß, "A cloud brokerage approach for solving the resource management problem in multi-cloud environments," *Comput. Ind. Eng.*, vol. 95, pp. 16–26, 2016.

[21] F. Jrad, J. Tao, I. Brandic, and A. Streit, "SLA enactment for large-scale healthcare workflows on multi-cloud," *Future Gener. Comput. Syst.*, vol. 43, pp. 135–148, 2015.

[22] M. Zhang, L. Liu, and S. Liu, "Genetic algorithm based QoS-aware service composition in multi-cloud," in *Proc. IEEE Conf. Collaboration Internet Comput.*, 2015, pp. 113–118.

[23] M. Guzek, P. Bouvry, and E.-G. Talbi, "A survey of evolutionary computation for resource management of processing in cloud computing," *IEEE Comput. Intell. Mag.*, vol. 10, no. 2, pp. 53–67, May 2015.

[24] L. Liu, M. Zhang, R. Buyya, and Q. Fan, "Deadline-constrained coevolutionary genetic algorithm for scientific workflow scheduling in cloud computing," *Concurrency Comput.: Practice Experience*, vol. 29, no. 5, 2017, Art. no. e3942.

[25] J. Yu and R. Buyya, "A budget constrained scheduling of workflow applications on utility grids using genetic algorithms," in *Proc. Workshop Workflows Support Large-Scale Sci.*, 2006, pp. 1–10.

[26] S. Salcedo-Sanz, "A survey of repair methods used as constraint handling techniques in evolutionary algorithms," *Comput. Sci. Rev.*, vol. 3, no. 3, pp. 175–192, 2009.

[27] C. A. Coello Coello, "Constraint-handling techniques used with evolutionary algorithms," in *Proc. Genetic Evol. Comput. Conf. Companion*, 2016, pp. 563–587.

[28] Z. Michalewicz, D. Dasgupta, R. G. Le Riche, and M. Schoenauer, "Evolutionary algorithms for constrained engineering problems," *Comput. Ind. Eng.*, vol. 30, no. 4, pp. 851–870, 1996.

[29] W. Zheng and R. Sakellariou, "Budget-deadline constrained workflow planning for admission control," *J. Grid Comput.*, vol. 11, no. 4, pp. 633–651, 2013.

[30] Z. Wen, J. Cała, P. Watson, and A. Romanovsky, "Cost effective, reliable and secure workflow deployment over federated clouds," *IEEE Trans. Services Comput.*, vol. 10, no. 6, pp. 929–941, Nov./Dec. 2017.

[31] Z. Mao, J. Yang, Y. Shang, C. Liu, and J. Chen, "A game theory of cloud service deployment," in *Proc. IEEE 9th World Congr. Services*, 2013, pp. 436–443.

[32] T. Shi, H. Ma, and G. Chen, "A seeding-based GA for location-aware workflow deployment in multi-cloud environment," in *Proc. IEEE Congr. Evol. Comput.*, 2019, pp. 3364–3371.

[33] Y. Chen and W.-T. Tsai, *Service-Oriented Computing and Web Software Integration: From Principles to Development*. Dubuque, IA, USA: Kendall/Hunt Publishing Co., 2012.

[34] J. Tordsson, R. S. Montero, R. Moreno-Vozmediano, and I. M. Llorente, "Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers," *Future Gener. Comput. Syst.*, vol. 28, no. 2, pp. 358–367, 2012.

[35] R. Sakellariou, H. Zhao, E. Tsiakkouri, and M. D. Dikaiakos, "Scheduling workflows with budget constraints," in *Integrated Research in GRID Computing*. Berlin, Germany: Springer, 2007, pp. 189–202.

[36] L. Zeng, B. Veeravalli, and X. Li, "ScaleStar: Budget conscious scheduling precedence-constrained many-task workflow applications in cloud," in *Proc. IEEE 26th Int. Conf. Adv. Inf. Netw. Appl.*, 2012, pp. 534–541.

[37] 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.

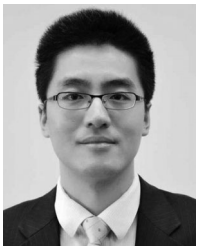
[38] H. Arabnejad and J. G. Barbosa, "A budget constrained scheduling algorithm for workflow applications," *J. Grid Comput.*, vol. 12, no. 4, pp. 665–679, 2014.

[39] E. Mezura-Montes and C. A. C. Coello, "Constraint-handling in nature-inspired numerical optimization: Past, present and future," *Swarm Evol. Comput.*, vol. 1, no. 4, pp. 173–194, 2011.

[40] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "PIAS: Practical information-agnostic flow scheduling for commodity data centers," *IEEE/ACM Trans. Netw.*, vol. 25, no. 4, pp. 1954–1967, Aug. 2017.

[41] J. Guo, F. Liu, T. Wang, and J. C. Lui, "Pricing intra-datacenter networks with over-committed bandwidth guarantee," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2017, pp. 69–81.

[42] Skytap, "Accessing VMs with published services," 2019. [Online]. Available: https://help.skytap.com/Accessing_VMs_with_Published_Services.html
[43] J. Vilaplana, F. Solsona, I. Teixidó, J. Mateo, F. Abella, and J. Rius, "A queuing theory model for cloud computing," *J. Supercomput.*, vol. 69, no. 1, pp. 492–507, 2014.
[44] J. D. Little and S. C. Graves, "Little's law," in *Building Intuition*. Berlin, Germany: Springer, 2008, pp. 81–100.
[45] J. Sahni and D. P. Vidyarthi, "A cost-effective deadline-constrained dynamic scheduling algorithm for scientific workflows in a cloud environment," *IEEE Trans. Cloud Comput.*, vol. 6, no. 1, pp. 2–18, First Quarter 2018.
[46] T. Hastie, R. Tibshirani, J. Friedman, and J. Franklin, "The elements of statistical learning: Data mining, inference and prediction," *Math. Intelligencer*, vol. 27, no. 2, pp. 83–85, 2005.
[47] J. W. Hallam, O. Akman, and F. Akman, "Genetic algorithms with shrinking population size," *Comput. Statist.*, vol. 25, no. 4, pp. 691–705, 2010.
[48] A. Lipowski and D. Lipowska, "Roulette-wheel selection via stochastic acceptance," *Physica A: Statist. Mech. Appl.*, vol. 391, no. 6, pp. 2193–2196, 2012.
[49] C. A. C. Coello, "Use of a self-adaptive penalty approach for engineering optimization problems," *Comput. Ind.*, vol. 41, no. 2, pp. 113–127, 2000.
[50] B. Wickremasinghe, R. N. Calheiros, and R. Buyya, "CloudAnalyst: A CloudSim-based visual modeller for analysing cloud computing environments and applications," in *Proc. 24th IEEE Int. Conf. Adv. Inf. Netw. Appl.*, 2010, pp. 446–452.
[51] P. Chootinan and A. Chen, "Constraint handling in genetic algorithms using a gradient-based repair method," *Comput. Operations Res.*, vol. 33, no. 8, pp. 2263–2281, 2006.
[52] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. 10th ACM SIGCOMM Conf. Internet Meas.*, 2010, pp. 267–280.
[53] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, "Workload prediction using ARIMA model and its impact on cloud applications' QoS," *IEEE Trans. Cloud Comput.*, vol. 3, no. 4, pp. 449–458, Fourth Quarter 2015.
[54] R. N. Calheiros, R. Ranjan, and R. Buyya, "Virtual machine provisioning based on analytical performance and QoS in cloud computing environments," in *Proc. Int. Conf. Parallel Process.*, 2011, pp. 295–304.
[55] M. Mao and M. Humphrey, "A performance study on the VM startup time in the cloud," in *Proc. IEEE 5th Int. Conf. Cloud Comput.*, 2012, pp. 423–430.
[56] N. Tsikoudis, A. Papadogiannakis, and E. P. Markatos, "LEoNIDS: A low-latency and energy-efficient network-level intrusion detection system," *IEEE Trans. Emerg. Topics Comput.*, vol. 4, no. 1, pp. 142–155, First Quarter 2014.



Tao Shi is currently working toward the PhD degree in the School of Engineering and Computer Science, Victoria University of Wellington, Wellington, New Zealand. His main research interests include parallel and distributed computing and evolutionary computation. His research focuses on resource management and combinatorial optimization in clouds.

Hui Ma received the BE degree from Tongji University, Shanghai, China, and the BS (Hons.), MS, and PhD degrees from Massey University, Wellington, New Zealand. She is currently an associate professor in software engineering with the Victoria University of Wellington. Her research interests include service computing, conceptual modelling, database systems, resource allocation in clouds, and evolutionary computation in combinatorial optimization. She has more than 90 publications, including leading journals and conferences in databases, service computing, evolutionary computation, and conceptual modelling, such as the *Journal of Heuristics*, the *Data & Knowledge Engineering* (DKE), the International Conference on Conceptual Modeling (ER), and Genetic and Evolutionary Computation Conference (GECCO), IEEE World Congress on Services, and the *IEEE Transactions on Services Computing* (TSC). She has served as a PC member for more than 70 international conferences, including seven times as a PC chair for conferences such as ER, DEXA, and APCCM. She has also served as local co-chair for Australian AI 2018 and CEC 2019.

Gang Chen received the PhD degree from Nanyang Technological University (NTU), Singapore. He is currently a senior lecturer with the School of Engineering and Computer Science, Victoria University of Wellington, New Zealand. His research interests include reinforcement learning, evolutionary computation and their application to optimization and scheduling problems, resource management, and load balancing in networked systems.

Sven Hartmann received the PhD and DSc degrees, both from the University of Rostock, Rostock, Germany, in 1996 and 2001. From 2002 to 2007, he worked first as an associate professor, then full professor for information systems with Massey University, New Zealand. Since 2008, he is a full professor of computer science and chair for databases and information systems with the Clausthal University of Technology, Germany. There he is also serving as academic dean with the Faculty of Mathematics, Informatics and Mechanical Engineering. He has more than 150 publications. He served as a PC member for more than 80 conferences, including 10 times as PC chair. His research interests include database systems, big data management, conceptual modelling, and combinatorial optimization.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Accelerating Stochastic Gradient Descent Based Matrix Factorization on FPGA

Shijie Zhou, Rajgopal Kannan, and Viktor K. Prasanna, *Fellow, IEEE*

Abstract—Matrix Factorization (MF) based on Stochastic Gradient Descent (SGD) is a powerful machine learning technique to derive hidden features of objects from observations. In this article, we design a highly parallel architecture based on Field-Programmable Gate Array (FPGA) to accelerate the training process of the SGD-based MF algorithm. We identify the challenges for the acceleration and propose novel algorithmic optimizations to overcome them. By transforming the SGD-based MF algorithm into a bipartite graph processing problem, we propose a 3-level hierarchical partitioning scheme that enables conflict-minimizing scheduling and processing of edges to achieve significant speedup. First, we develop a fast heuristic graph partitioning approach to partition the bipartite graph into induced subgraphs; this enables to efficiently use the on-chip memory resources of FPGA for data reuse and completely hide the data communication between FPGA and external memory. Second, we partition all the edges of each subgraph into non-overlapping matchings to extract the maximum parallelism. Third, we propose a batching algorithm to schedule the execution of the edges inside each matching to reduce the memory access conflicts to the on-chip RAMs of FPGA. Compared with non-optimized FPGA-based baseline designs, the proposed optimizations result in up to 60× data dependency reduction, 4.2× bank conflict reduction, and 15.4× speedup. We evaluate the performance of our design using a state-of-the-art FPGA device. Experimental results show that our FPGA accelerator sustains a high computing throughput of up to 217 billion floating-point operations per second (GFLOPS) for training very large real-life sparse matrices. Compared with highly-optimized GPU-based accelerators, our FPGA accelerator achieves up to 12.7× speedup. Based on our optimization methodology, we also implement a software-based design on a multi-core platform, which demonstrates 1.3× speedup compared with the state-of-the-art multi-core implementation.

Index Terms—Machine learning, sparse matrix factorization, training acceleration, bipartite graph representation, FPGA accelerator

1 Introduction

Matrix Factorization (MF), which factors a sparse matrix into two low-rank matrices, is a widely used machine learning technique for many applications such as collaborative filtering [1], topic modeling [10], and text mining [11]. This technique can achieve high prediction accuracy because it is able to derive latent features from observations [1]. Therefore, the model trained by the matrix factorization techique is also called latent factor model [1]. Stochastic Gradient Descent (SGD) is gradient descent optimization technique used to minimize certain objective functions [47]. It is a popular learning algorithm to train the MF model. However, the training process of the SGD-based MF algorithm is very computation-intensive. This is because the MF model needs to be iteratively updated based on the training data for thousands of iterations [1]. When the volume of training data is huge, the training time can become excessively long. Therefore, it is essential to design hardware accelerators to accelerate the training process. However, existing accelerators suffer from the expensive synchronization overhead and the limited parallelism of this algorithm [22]. Accelerating SGD-based MF is still a challenging problem.

Field-Programmable Gate Array (FPGA) is an attractive platform to accelerate computation-intensive applications. FPGA has been widely used to accelerate the inference process of machine learning applications [3], [4], [5], [6], [7], but using FPGA to accelerate the training process remains a challenging research area [8], [9]. In this paper, we design a parallel architecture based on state-of-the-art FPGA to accelerate the training process of the SGD-based MF algorithm. Our design methodology is holistic and generalized: It is motivated by a thorough analysis of the challenges involved in accelerating SGD on FPGAs and designed to be adaptable to accelerating SGD-based MF on a variety of computing platforms. We identify three principal acceleration challenges (1) limited on-chip memory which can limit throughput if the long latencies of external memory accesses are not managed, (2) data dependencies that can prevent concurrent processing within the FPGA pipeline, and (3) access conflicts between different pipelines when accessing dual-port on-chip RAMs. Based on our analysis of these design challenges, we develop a bipartite graph processing approach in which the input training data is first transformed into a bipartite graph representation and followed by a novel 3-level hierarchical partitioning, scheduling and processing of on-chip feature vector data that significantly accelerates the processing of

- *S. Zhou is with Microsoft Corporation, Redmond, WA 98052. E-mail: shijieinusc@gmail.com.*
- *R. Kannan is with US Army Research Lab, Los Angeles, CA 90094. E-mail: Rajgopal.kannan.civ@mail.mil.*
- *V.K. Prasanna is with the Department of Electrical and Computer Engineering, University of Southern California, Los Angeles, CA 90089. E-mail: prasanna@usc.edu.*

Manuscript received 30 Mar. 2019; revised 5 Jan. 2020; accepted 2 Feb. 2020. Date of publication 18 Feb. 2020; date of current version 3 Apr. 2020. (Corresponding author: Viktor K. Prasanna.) Recommended for acceptance by M. D. Santambrogio. Digital Object Identifier no. 10.1109/TPDS.2020.2974744

1045-9219 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

this bipartite graph. In addition, we note that our holistic design approach is general enough to be adapted to accelerating multi-core implementations. Thus we also observe that our optimized multi-core implementation outperforms the state-of-the-art multi-core design by 1.3× due to our optimizations.

We summarize the main contributions of this paper below.

- We develop a highly parallel architecture to accelerate the training process of SGF-based MF. The accelerator consists of parallel processing units concurrently working on distinct input data to sustain high processing throughput. It also has an on-chip buffer to store the feature data to achieve efficient data reuse.
- By using a bipartite graph representation of MF, we propose three novel optimizations to overcome the challenges in acceleration.
  - We first partition the input bipartite graph into induced subgraphs. In lieu of more sophisticated partitioning algorithms with higher preprocessing costs, we develop a simple and fast partitioning heuristic that satisfies a necessary condition for storing the feature vectors of vertices in the on-chip buffer. By overlapping the communication overhead with computation, the accelerator can absorb the long latencies of external memory accesses.
  - To overcome the data dependency challenge, we maximize the available parallelism by partitioning the edges of each induced subgraph into matchings. This reduces the data dependencies among the edges by up to $60\times$.
  - To overcome the bank-conflict challenge, we develop a batching algorithm to partition each matching into batches and schedule the execution of the batches to reduce the conflict accesses to the shared on-chip buffer (i.e., bank conflicts). This optimization results in up to $4.2\times$ bank conflict reduction.
- Experimental results show that the proposed FPGA accelerator sustains a high throughput of up to 217 GFLOPS for training large real-life sparse datasets. Compared with state-of-the-art GPU implementations, our FPGA accelerator achieves up to 12.7× speedup and 24.8× throughput improvement.
- Our proposed optimizations are generic to general purpose processors as well. We adapt our techniques to mutli-core platform and implement a software-based design. The optimized software-based design achieves 1.3× speedup compared with state-of-the-art multi-core implementation with 1.5× fewer cores.

The rest of the paper is organized as follows. Section 2 covers the background and defines the problem; Section 3 introduces the challenges in accelerating the SGD-based MF algorithm; Section 4 presents our proposed optimizations; Section 5 describes the architecture of the accelerator; Section 7 summaries the related work; Section 8 discusses the generality of our design; Section 9 concludes the paper.

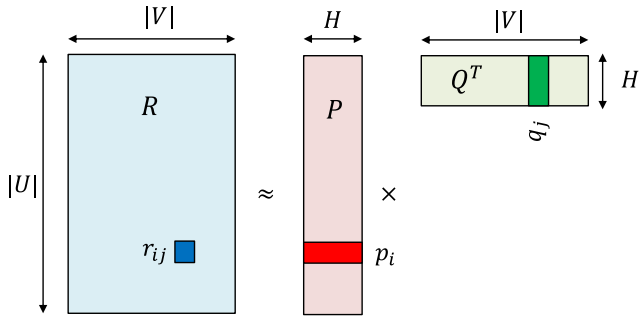


Fig. 1. Input and output of training process.

2 Background

2.1 Matrix Factorization in Machine Learning

MF is a technique to factor a sparse matrix into the product of two low-rank dense matrices. In machine learning applications, MF is used to predict unknown data based on a collection of existing observations that are represented in the matrix format. For example, MF is used to predict customer ratings on products in collaborative filtering [1]. This technique is powerful because it is able to discover some latent features from the observations. An example latent feature of a customer might be his/her income level, which has an impact on his/her interest in products but cannot be directly derived from the observed data (e.g., ratings on products). Hence, the output model of MF is called latent factor model. It consists of two dense matrices that are called (latent) feature matrices. There are three primary approaches used for training the model, including Alternating Least Square (ALS), Stochastic Gradient Descent (SGD), and Gradient Descent (GD) [12], [13], [43].

ALS-based MF alternately fixes one of the two feature matrices. When one feature matrix is fixed, the other feature matrix is recomputed by solving a least-squares problem. ALS-based MF algorithm is inherently parallelizable and requires fewer iterations to converge than the SGD-based MF algorithm. However, it is not scalable to large-scale datasets due to its cubic algorithmic complexity in each iteration [43]. SGD-based MF randomly initializes both the feature matrices and then iteratively improves them. In each iteration, SGD-based MF sequentially processes all the training data. Based on each training data, it updates the feature matrices in the opposite direction of the gradient. GD-based MF also iteratively updates the feature matrices. But it accumulates the intermediate updates and applies the update only after all the training data have been processed in an iteration. Hence, the key difference between SGD-based MF and GD-based MF is that SGD-based MF updates the features matrices once per training data, while GD-based MF updates the features matrices once per iteration. As a result, GD-based MF requires more iterations to converge as well as more training time than SGD-based MF [36], [44]. In this paper, we focus on accelerating SGD-based MF.

2.2 Problem Definition

Without loss of generality, we define the problem based on the context of collaborative filtering for recommender systems [1]. Let U and V denote a set of users and items, $|U|$ and $|V|$ denote the number of users and items, respectively. As shown in Fig. 1, the *input* training dataset is a partially

observed rating matrix $R = \{r_{ij}\}_{|U|\times|V|}$, in which r_{ij} represents the rating of item v_j given by user u_i ($0 \le i < |U|$, $0 \le j < |V|$). The *output* of the training process contains two low-rank matrices, P (a $|U| \times H$ matrix) and Q (a $|V| \times H$ matrix), which are refereed as user feature matrix and item feature matrix, respectively. A typical value of the rank of P and Q (i.e., H) is 32 [15], [16], [17]. The ith row of P (denoted as p_i) constitutes a *feature vector* of user u_i and the jth row of Q (denoted as q_j) constitutes a feature vector of item v_j, respectively. The prediction of the rating of item v_j given by user u_i is the dot product of p_i and q_j

$$\hat{r}_{ij} = p_i \cdot q_j = \sum_{h=0}^{H-1} p_{ih} \cdot q_{jh}. \tag{1}$$

Given an observed rating r_{ij}, the prediction error is computed as $err_{ij} = r_{ij} - \hat{r}_{ij}$. The objective of the training process is to obtain such P and Q that minimize the overall regularized squared error based on all the observed ratings

$$\min_{P,Q} \sum_{u_i \in U, v_j \in V} err_{ij}^2 + \lambda \cdot (||p_i||^2 + ||q_j||^2). \tag{2}$$

In the objective function, λ is a constant used to introduce regularization to prevent overfitting. In order to minimize the objective function, SGD is used to update the feature vectors. As shown in Algorithm 1, SGD randomly initializes all the feature vectors and then updates them by iteratively traversing all the observed ratings. The training process terminates when the overall squared error (i.e., $\sum err_{ij}^2$) converges. By taking an observed rating r_{ij}, p_i and q_j are updated by a magnitude proportional to a constant α (i.e., learning rate) in the opposite direction of the gradient, yielding the following updating equations:

$$p_i^{new} = \beta \cdot p_i^{old} + err_{ij} \cdot \alpha \cdot q_j^{old} \tag{3}$$

$$q_j^{new} = \beta \cdot q_j^{old} + err_{ij} \cdot \alpha \cdot p_i^{old}. \tag{4}$$

In Eqs. (3) and (4), β is a constant whose value is equal to $(1 - \alpha\lambda)$. The algorithm requires to incrementally update the feature vectors *once per rating*. As a result, the ratings of the same item or given by the same user cannot be concurrently processed because they will result in concurrent updates to the same p_i or q_j. Additional details of this algorithm can be found in [1], [2].

Algorithm 1. SGD-Based MF

Train (R)
1: Randomly initialize P and Q
2: **while** Termination_condition = false **do**
3: Overall_squared_error= 0
4: **for** each observed rating $r_{ij} \in R$ **do**
5: Compute $\hat{r}_{ij}$ based on p_i and q_j
6: Compute err_{ij} based on r_{ij} and $\hat{r}_{ij}$
7: Update p_i and q_j based on Eqs. (3) and (4)
8: Overall_squared_error+ $= err_{ij}^2$
9: **end for**
10: **end while**
11: **Return** P and Q

3 Challenges in Acceleration

To achieve efficient acceleration of the SGD-based MF algorithm using FPGA, three challenges need to be carefully addressed.

3.1 Large Data Volume but Limited On-Chip Memory Resources

Since the feature vectors of users and items are repeatedly accessed and updated during the processing, it is desirable to store them in the on-chip memory of FPGA for data reuse. However, for large training dataset that involves a large number of users and items, the feature vectors cannot fit in the on-chip memory. In this scenario, external memory such as DRAM is required to store them. However, accessing feature vectors from external memory can incur long access latencies, which result in massive accelerator pipeline stalls and significant performance deterioration [19], [21]. Therefore, the first challenge is how to use the limited on-chip memory resources to achieve efficient data reuse.

3.2 Limited Parallelism Due to Data Dependencies

SGD is inherently a serial algorithm because it requires to incrementally update the training model once per training data. As a result, the ratings of the same item or given by the same user have data dependencies and cannot be processed in parallel. This is because concurrent processing of such ratings leads to Read-After-Write (RAW) data hazard. We define such data dependency among ratings as *feature vector dependency*. Hence, the second challenge is how to reduce the feature vector dependencies so that the massive parallelism offered by FPGA can be efficiently exploited.

3.3 Concurrent Accesses to Dual-Port On-Chip RAMs

FPGA accelerators usually employ parallel processing units to increase processing throughput [3], [19], [20]. However, the native on-chip RAMs (e.g., block RAM and UltraRAM) of FPGA support only dual-port accesses (one read port and/or one write port) [26], [27], [29]. When multiple processing units concurrently access the same RAM based on distinct memory addresses, these memory accesses have to be serially served. This leads to additional latency to resolve the access conflicts as well as performance deterioration. Therefore, the third challenge is how to schedule the execution to reduce such access conflicts.

4 Graph Representation and Algorithmic Optimizations

In order to overcome the three challenges described in Section 3, we represent the MF problem using a bipartite graph representation (Section 4.1) and propose three novel algorithmic optimizations (Section 4.2).

4.1 Graph Representation

We transform SGD-based MF into a bipartite graph-processing problem so that graph theories can be leveraged to optimize the performance. The input matrix is converted into a weighted bipartite graph G, whose vertices can be divided into two disjoint sets, U (user vertices) and V (item vertices).

TABLE 1
Bipartite Graph Notations for MF

Notation	Description
u_i	the user vertex with index i ($0 \leq i < \|U\|$)
v_j	the item vertex with index j ($0 \leq j < \|V\|$)
p_i	the feature vector of u_i
q_j	the feature vector of v_j
H	the length of each feature vector
$edge_{ij}$	the edge connecting u_i and v_j
r_{ij}	the weight of $edge_{ij}$

Each observed rating in R is represented as an edge connecting a user vertex and an item vertex in G. G is represented using the coordinate (COO) format [22], which is a commonly used graph representation [16], [17], [18], [22], [23], [24]. In this format, each edge (i.e., $edge_{ij}$) is represented as a $<u_i, v_j, r_{ij}>$ tuple, in which u_i and v_j refer to the user and item vertices connected by the edge, and r_{ij} (i.e., edge weight) corresponds to the rating value of v_j given by u_i; all the edges are stored in an edge list E; each user/item vertex maintains a feature vector whose length is H. Table 1 summaries the frequently used graph notations in this paper. Algorithm 2 illustrates the SGD-based MF algorithm based on the bipartite graph representation. All the edges in E are iteratively processed to update the feature vectors of vertices until the overall squared error converges. When the training process terminates, the feature vectors of all the user vertices and item vertices constitute the output feature matrices P and Q, respectively.

Algorithm 2. SGD-Based MF Using Graph Representation

MF_Train ($G(U, V, E)$)
1: **for** each user/item vertex **do**
2: Randomly initialize its feature vector
3: **end for**
4: **while** Termination_condition = false **do**
5: **for** each $edge_{ij} \in E$ **do**
6: Read feature vectors p_i and q_j
7: Compute $\hat{r}_{ij}$ based on Eq. (1)
8: Compute err_{ij} based on r_{ij} and $\hat{r}_{ij}$
9: Update p_i and q_j based on Eqs. (3) and (4)
10: **end for**
11: **end while**
12: **Return** all the feature vectors of vertices

4.2 Algorithmic Optimizations

4.2.1 Optimization 1: Graph Partitioning and Communication Hiding

In order to address the challenge described in Section 3.1, we partition G into induced subgraphs (IS) to achieve two goals: (1) the feature vectors of the vertices in each induced subgraph can fit in the on-chip memory of FPGA; (2) the computation for processing each induced subgraph can completely hide the communication cost.

Let L (N) denote the on-chip storage capacity in terms of the number of feature vectors for user (item) vertices. We partition U into l disjoint vertex subsets $\{U_0, \ldots, U_{l-1}\}$, each of size at most L, where $l = \lceil \frac{|U|}{L} \rceil$. Similarly, V is partitioned into $\{V_0, \ldots, V_{n-1}\}$, each of size at most N, where $n = \lceil \frac{|V|}{N} \rceil$. We will introduce our proposed algorithm to perform the partitioning of U and V in details later in this section. Let E_{xy} denote a subset of E that consists of all the edges connecting the vertices belonging to U_x and V_y in G ($0 \leq x < l, 0 \leq y < n$). U_x, V_y, and E_{xy} form an *Induced Subgraph (IS)* of G [30]. The *necessary condition* for the on-chip buffering of all the feature vectors of each IS is

$$|U_x| \leq L, \forall x \in [0, l) \quad \& \quad |V_y| \leq N, \forall y \in [0, n). \tag{5}$$

Since we ensure that each user (item) vertex subset has no more than L (N) vertices during the partitioning, the necessary condition for the on-chip buffering of the feature vectors is satisfied.

Because there are l user vertex subsets and n item vertex subsets, the total number of induced subgraphs after the partitioning is $l \times n$. Then, in each iteration of the training process, the induced subgraphs are sequentially processed by our FPGA accelerator based on Algorithm 3. Note that during the processing of the edges in E_{xy}, all the feature vectors of the vertices in U_x and V_y have been prefetched and buffered into an on-chip buffer of FPGA; therefore, the processing units of the accelerator can directly access the feature vectors from the on-chip buffer, rather than from the external memory.

Algorithm 3. Scheduling of Induced Subgraph Processing

MF_Train (Induced Subgraphs)
1: **while** Termination_condition = false **do**
2: **for** x from 0 to $l - 1$ **do**
3: Load feature vectors of U_x into on-chip buffer
4: **for** y from 0 to $n - 1$ **do**
5: Load feature vectors of V_y into on-chip buffer
6: Process all the edges $\in E_{xy}$
7: Write feature vectors of V_y into external memory
8: **end for**
9: Write feature vectors of U_x into external memory
10: **end for**
11: **end while**

Using double buffering [3], [31], we pipeline the processing of induced subgraphs to hide the communication cost for data transferring between FPGA and external memory. To illustrate our idea, we define the following terms.

- $|IS_\tau|$: the number of edges in an induced subgraph $IS_\tau, \tau \in [0, l \times n)$
- Ω: the number of edges that can be processed per unit of time
- $P(IS_\tau)$: the computation time to process all the edges of IS_z (i.e., $P(IS_\tau) = \frac{|IS_\tau|}{\Omega}$)
- T_τ^{rd}: the communication time due to reading the feature vectors of IS_τ from external memory
- T_τ^{wr}: the communication time due to writing the feature vectors of IS_τ into external memory
- T_τ: the *intra-subgraph* communication time (i.e., the total time for data transfers, including both reads and writes, occurred during the processing of IS_τ)

As shown in Fig. 2, we pipeline the processing of induced subgraphs by overlapping the computation time $P(IS_\tau)$ of

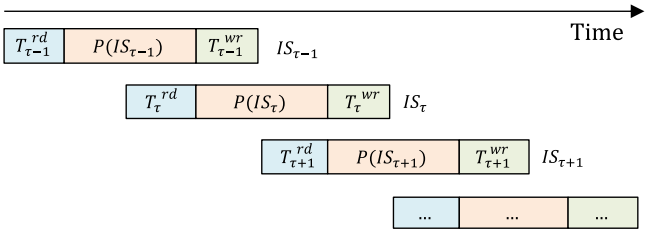


Fig. 2. Pipelined induced subgraph processing.

IS_τ with the *writing* of feature vectors from $IS_{\tau-1}$ and the *reading* of feature vectors from $IS_{\tau+1}$. Therefore, $T_\tau = T_{\tau-1}^{wr} + T_{\tau+1}^{rd}$. Here, T_τ in general may include reads or writes of both user and item feature vectors. We can derive the *sufficient condition* for a *complete* overlap of communication and computation

$$P(IS_\tau) \geq T_{\tau-1}^{wr} + T_{\tau+1}^{rd}, \forall \tau \in [0, l \times n). \tag{6}$$

By replacing the $P(IS_\tau)$ in Inequality (6) with $\frac{|IS_\tau|}{\Omega}$, we obtain

$$|IS_\tau| \geq \Omega \times T_\tau, \forall \tau \in [0, l \times n). \tag{7}$$

Therefore, besides satisfying the necessary condition for on-chip buffering of feature vectors, a desirable graph partitioning approach should also ensure that each obtained IS has sufficient amount of edges for the computation to completely hide the communication. Note that as parallelism (i.e., Ω) increases, it becomes more challenging to satisfy the sufficient condition, especially when the bipartite graph is sparse. Graph partitioning is a classic problem and many sophisticated graph partitioning algorithms have been developed [33]. However, these sophisticated approaches usually introduce significant preprocessing overhead. Our intuition is that it is not necessary to invest significantly in developing complex partitioning algorithms (in terms of preprocessing time), rather any reasonably fast algorithm that satisfies Inequality (5) and (7) is acceptable. A vertex-index-based partitioning approach [32] has been widely used to perform graph partitioning for hardware accelerators [17], [20]. This approach simply assigns a group of vertices with contiguous indices to each vertex subset subject to the subset size condition (i.e., Inequality (5)). Although this approach is fast, it can lead to significant data imbalance such that some induced subgraphs may have very few edges; in this scenario, the communication cost cannot be completely hidden by the computation. Therefore, we propose a new heuristic graph partitioning approach, which is also simple and fast, but leads to a balanced partitioning effect such that each obtained IS has sufficient edges to satisfy Inequality (6) for a sparse bipartite graph.

We define the *subset degree* of a vertex subset as the total number of edges that connect to the vertices in the subset. When we partition U and V into vertex subsets, we attempt to pack vertices into each disjoint vertex subset such that the subset degrees are close to each other. However, the most important criteria is to ensure that the number of vertices in each vertex subset is bound by L and N as defined earlier. Algorithm 4 illustrates our approach to partition U into $U_0, \ldots, U_{l-1}$; V is partitioned based on the same methodology. We first identify the vertex degree of each vertex (i.e., the number of edges connected to the vertex) and sort all the vertices based on the vertex degree in descending order. Then, we greedily assign each vertex into the vertex subset which has the minimum subset degree, until all the vertices are assigned, subject to the subset size condition.

Algorithm 4. Partition U into l Subsets $U_0, \ldots, U_{l-1}$

Let $u_{i \cdot degree}$ denote the number of edges connected to u_i $(0 \leq i < |U|)$
Let $U_{x \cdot size}$ denote the number of vertices in U_x $(0 \leq x < l)$
Let $U_{x \cdot degree}$ denote the subset degree of U_x (i.e., $U_{x \cdot degree} = \sum u_{i \cdot degree}, \forall u_i \in U_x$)
Partition (U, L, l)

```
1: for x from 0 to l − 1 do
2:   U_x ← ?
3:   U_{x·degree} ← 0
4:   U_{x·size} ← 0
5: end for
6: Sort U based on vertex degree in descending order
7: for each u_i ∈ U do
8:   subset_id ← −1
9:   min_degree ← |E|
10:  for x from 0 to l − 1 do
11:    if min_degree > U_{x·degree} and U_{x·size} < L then
12:      subset_id ← x
13:      min_degree ← U_{x·degree}
14:    end if
15:  end for
16:  U_{subset_id} ← U_{subset_id} ∪ u_i
17:  U_{subset_id·degree} ← U_{subset_id·degree} + u_{i·degree}
18:  u_{i·new_user_id} ← subset_id × L + U_{subset_id·size}
19:  U_{subset_id·size} ← U_{subset_id·size} + 1
20: end for
21: Return U_0, ..., U_{l−1}
```

When each vertex is assigned to a vertex subset, we assign a new vertex index to it (Algorithm 4, Line 18), which indicates the vertex subset that it belongs to and its index in the vertex subset. After U and V are partitioned, we reorder the vertices based on the new indices such that the feature vectors of the vertices belonging to the same vertex subset are stored contiguously in external memory. Since user and item vertices are reordered, we also re-index the user and item indices of each edge and partition the edges into induced subgraphs based on the new indices.

4.2.2 Optimization 2: Parallelism Extraction

To address the challenge described in Section 3.2, we further partition the edges in each IS into a list of non-overlapping *matchings*, such that each matching consists of a set of independent edges without any common vertices. Therefore, the edges in the same matching do not have any feature vector dependencies and can be independently processed in parallel.

We partition the edges in each IS into matchings based on the graph theory of edge-coloring [30], which assigns "colors" to the edges of a bipartite graph such that any two adjacent edges do not have the same color. After all the edges have been colored, the edges having the same color form a matching. A classic edge-coloring algorithm has been introduced in [30]. However, this algorithm can result

in small matchings, in which there are very few edges (e.g., only 1 edge). When processing such small matchings, the parallelism provided by the hardware accelerator (i.e., parallel processing units) cannot be fully utilized. Therefore, we propose a new edge-coloring algorithm which avoids small matchings. As shown in Algorithm 5, we maintain all the matchings using a singly linked list (i.e., M_List). During the edge-coloring process, the linked list keeps track of the size of each matching (i.e., the number of edges in the matching), and arranges the matchings based on their sizes in a non-descending order. When coloring an edge, we traverse the linked list and assign the edge to the first matching whose color is appropriate. Therefore, when an edge has multiple color options, the color of the matching that has the minimum number of edges is selected.

Algorithm 5. Partition Edges of an IS into Matchings

Let M_List denote a linked list of matchings
Let M denote the matching being examined
Let $M.size$ denote the number of edges in the matching M
Let $M.color$ denote the color of the matching M
Let M_{next} denote the next matching in M_List linked by M
Partition (IS)
1: Create empty matching M with $M.color$
2: $M_List.addFirst(M)$
3: **for** each edge $e_{ij} \in IS$ **do**
4: $M \leftarrow M_List.head$
5: **while** $M \neq$ Null **do**
6: **if** u_i or v_j has an edge colored by $M.color$ **then**
7: $M \leftarrow M_{next}$
8: **else**
9: Color e_{ij} using $M.color$
10: $M \leftarrow M \cup e_{ij}$
11: $M.size \leftarrow M.size + 1$
12: **while** $\exists M_{next}$ and $M_{next}.size < M.size$ **do**
13: Swap M and M_{next} in M_List
14: **end while**
15: Go to Line 3
16: **end if**
17: **end while**
18: Create empty matching M with $M.color$
19: $M \leftarrow M \cup e_{ij}$
20: $M.size \leftarrow 1$
21: $M_List.addFirst(M)$
22: **end for**
23: **Return** M_List

4.2.3 Optimization 3: Edge Scheduling

The architecture of our accelerator has K parallel processing units sharing an on-chip buffer, which is organized in $2K^*$ ($K^* \geq K$) memory banks with separate banks for users and items (see Section 5.3); therefore, a batch of K edges are fed into the processing units and processed at a time. However, due to the dual-port nature of on-chip RAMs [26], [27], [29], each bank can serve only 1 read access and 1 write access per clock cycle. If there is a bank conflict between two or more accesses within a batch, the memory requests to process the edges have to be serially served. Hence, the latency (in terms of clock cycles) to resolve the bank conflict(s) within a batch is equal to the *maximum* number of accesses to the same bank within the batch. In order to reduce the bank conflicts, we develop a batching algorithm to schedule the processing of edges. As shown in Algorithm 6, the algorithm aims to partition the edges of a matching into batches, with each batch having K edges. We define a threshold value, Δ, to restrict the upper bound of bank conflicts allowed within a batch. Δ is initially set to 0. Then we sequentially traverse the edges and assign an edge into a batch if its addition does not violate the threshold condition or the batch size condition. If there are still unassigned edges after all the edges have been traversed, we increase Δ by 1 and traverse the unassigned edges again. The same procedures repeat until all the edges have been signed into a batch.

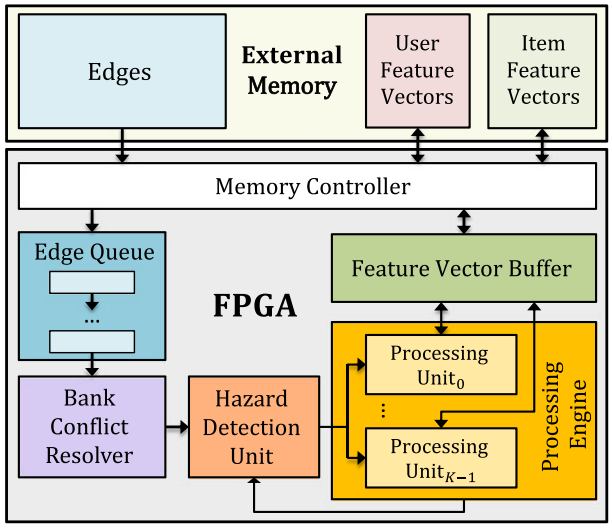


Fig. 3. Overall architecture.

Algorithm 6. Partition Edges of a Matching into Batches

Let K denote the maximum number of edges that can be assigned in a batch
Let $Count_BC(B, e)$ denote a function to count the number of edges in batch B that have bank conflict with edge e
Partition (M)
1: Create $b = \lceil \frac{M.size}{K} \rceil$ empty batches, $B_0, \ldots, B_{b-1}$
2: $\Delta \leftarrow 0$
3: **while** $M \neq ?$ **do**
4: **for** each edge $e \in M$ **do**
5: **for** i from 0 to $b-1$ **do**
6: **if** $B_i.size < K$ **and** $Count_BC(B_i, e) \leq \Delta$ **then**
7: $B_i \leftarrow B_i \cup e$
8: $M \leftarrow M \backslash e$
9: Break
10: **end if**
11: **end for**
12: **end for**
13: $\Delta \leftarrow \Delta + 1$
14: **end while**
15: **Return** $B_0, \ldots, B_{b-1}$

5 FPGA Accelerator Design

5.1 Overall Architecture

The overall architecture of our FPGA accelerator is depicted in Fig. 3. The external memory connected to the FPGA

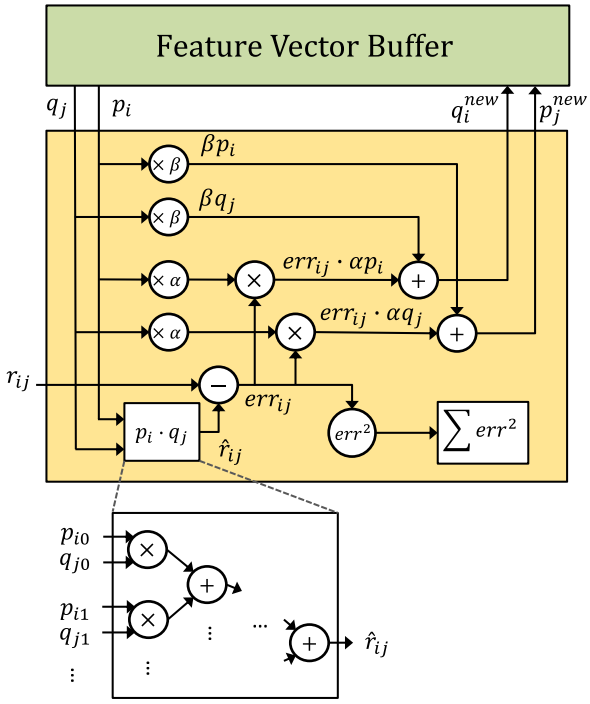


Fig. 4. Architecture of processing unit.

accelerator stores all the edges and the feature vectors of all the user and item vertices. Before an IS is processed, the feature vectors of all the vertices belonging to the IS have been stored in the Feature Vector Buffer (FVB), which is organized as memory banks of UltraRAMs (see Section 5.3). When processing an IS, FPGA fetches the edges from the external memory and stores them into a first-in-first-out Edge Queue (EQ). Whenever the EQ is not full, FPGA prefetches edges from the external memory. A batch of edges are fed into the Bank Conflict Resolver (BCR) (see Section 5.3) at a time and output in one or multiple clock cycles, such that the edges output in the same clock cycle do not result in any bank conflict accesses to the FVB. Then, the edges output by the BCR are checked by the Hazard Detection Unit (HDU) to determine whether they are data-hazard free to be processed. If an edge has no feature vector dependency with any edge being processed in the Processing Engine (PE), it is sent into the PE; otherwise, accelerator stalls occur until the dependency is resolved. The PE consists of K processing units that process distinct edges in parallel. These processing units access the feature vectors of user and item vertices from the FVB.

5.2 Processing Engine

The processing engine (PE) consists of K parallel processing units that concurrently process distinct edges. We show the architecture of each processing unit in Fig. 4. Each input edge is processed based on the follow steps.

1) Based on the user and item vertex indices of the edge, the processing unit reads the feature vectors (i.e., p_i and q_j) from the FVB
2) The prediction $\hat{r}_{ij}$ is computed based on p_i and q_j; meanwhile, p_i and q_j are multiplied with the constants (i.e., α and β) to obtain αp_i, αq_j, βp_i, and βq_j
3) Once the prediction error err_{ij} is obtained, p_i^{new} and q_j^{new} are computed based on Eqs. (3) and (4)
4) The updated feature vectors (i.e., p_i^{new} and q_j^{new}) are written into the FVB

The dot product of p_i and q_j is computed in a binary-reduction-tree fashion, requiring H (i.e., the length of each feature vector) multipliers and $(H-1)$ adders in total. Hence, the processing unit depicted in Fig. 4 contains $7H$ multipliers, $(3H-1)$ adders, 1 subtractor, 1 squarer, and 1 accumulator. This results in a peak computing throughput of $(10H+2)$ floating point operations per clock cycle. The processing unit is fully pipelined to be able to process one edge per clock cycle. We use three pipeline stages to compute each floating point operation. Hence, the pipeline depth of the processing unit is $3(\log H + 4)$.

5.3 Feature Vector Buffer

Since the K processing units of the PE need to concurrently access the Feature Vector Buffer (FVB) to read and write distinct feature vectors, there can be up to $2K$ read requests[1] and $2K$ write requests in each clock cycle. However, the native on-chip RAMs of FPGA, such as BRAMs and UltraRAMs, provide only two ports for reading and/or writing [26], [27], [29]. There are three major approaches to build multiport memory using dual-port on-chip RAMs, including multi-pumping [34], replication [26], and banking [35]. Multi-pumping gains additional ports by running the processing units with $K\times$ lower frequency than the memory. Consequently, this significantly deteriorates the clock rate of the processing units for a large K (e.g., $K = 8$) [25], [26], [27]. Replication-based approaches, such as LVT [25] and XOR [26], create replicas of all the stored data to provide additional ports and keep track of which replica has the most recently updated value for each data element. However, the amount of the RAMs needed in implementing this approach grows quadratically with the number of ports, such that $K \times K$ replicas are required to support K read ports and K write ports. Additionally, the clock rate can degrade below 100 MHz when the width and depth of the memory are large (e.g., 1 Kbit $\times$ 16 K) [27], [28].

In order to support large buffer capacity and sustain high clock rate, we adopt the banking approach [35] to build the multiport FVB. This approach divides the memory into equal sized banks and interleaves these banks to provide higher access bandwidth (i.e., more read and write ports). As shown in Fig. 5, the FVB contains two parts of equal size, one for storing user feature vectors and the other for storing item feature vectors. Each part is divided into K^* banks $(K^* \geq K)$ and each bank is implemented using a dual-port UltraRAM [29]. Therefore, the FVB provides $2K^*$ read ports and $2K^*$ write ports in total. Feature vectors of vertices are stored into the FVB in a modular fashion based on the vertex indices, such that p_i is stored in the $(i\%K^*)$th user bank and q_j is stored in the $(j\%K^*)$th item bank. Hence, the feature vector of any user (item) can be accessed from the FVB based on the user (item) vertex index without complex index-to-address translation.

1. K for user feature vectors and K for item feature vectors

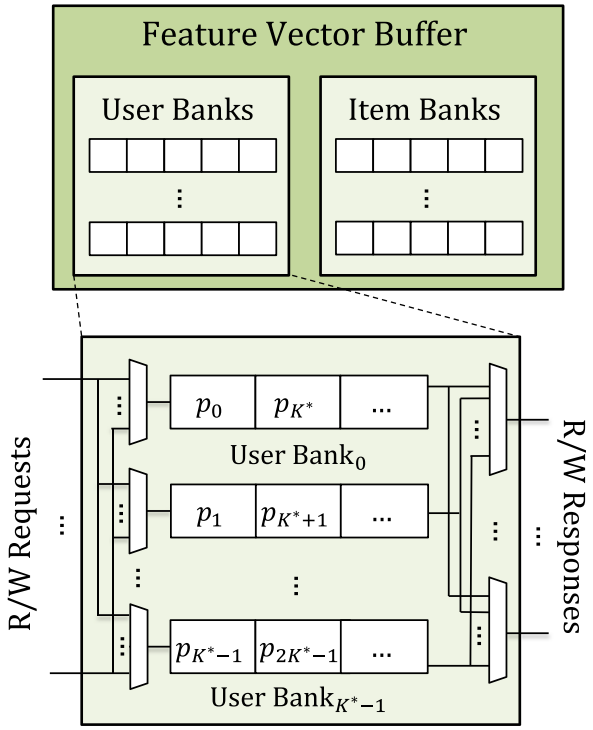


Fig. 5. Multiport FVB based on banking.

However, the banked FVB cannot handle concurrent accesses to the same bank for distinct feature vectors. Such memory accesses are defined as *bank conflict* accesses, and may occur for both user and item feature vectors. To address this issue, we develop a Bank Conflict Resolver (BCR) to avoid any bank conflict accesses. As illustrated in Fig. 6, the BCR fetches a batch of K edges from the Edge Queue (EQ) at a time and employs parallel detectors to detect the potential bank conflicts among the edges. Then, it outputs the edges to the Hazard Detection Unit (HDU) in one or mutliple clock cycles, such that the edges output in the same clock cycle have the feature vectors stored in distinct banks of the FVB. Therefore, concurrent accesses to the same bank will not occur. However, this also leads to additional clock cycles to resolve the bank conflicts within a batch; in the worst case, when all the edges in a batch have conflict with each other, the BCR takes K clock cycles to output all the K edges in the batch.

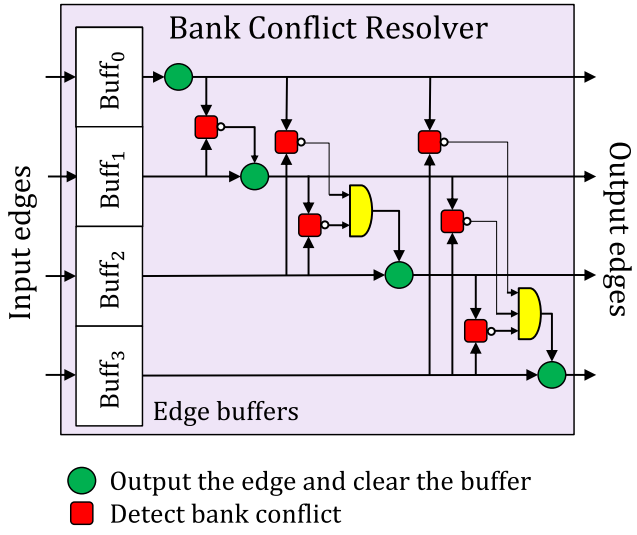


Fig. 6. Architecture of bank conflict resolver for $K = 4$.

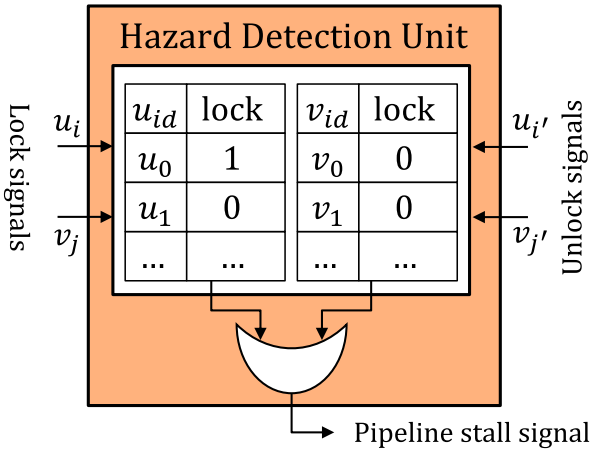


Fig. 7. Architecture of hazard detection unit.

5.4 Hazard Detection Unit

Since the edges of distinct matchings can have common vertices, feature vector dependencies exist among the edges of distinct matchings. Therefore, when the edges output from the Bank Conflict Resolver (BCR) and the edges being processed in the Processing Engine (PE) belong to distinct matchings, read-after-write data hazards may occur. The Hazard Detection Unit (HDU) is responsible for detecting feature vector dependencies and preventing read-after-write data hazards. We design the HDU using BRAMs based on a fine-grained locking mechanism. As shown in Fig. 7, the HDU maintains a 1-bit *lock* for each vertex of the IS being processed. A lock with value 1 means the feature vector of the corresponding vertex is being computed by the PE, and thus cannot be accessed at this time. For each input edge, the HDU checks the locks of both the user and item vertices; if both the locks are 0 (i.e., unlocked), the edge is fed into the PE and the locks are set to 1 (i.e., locked); otherwise, the HDU generates a pipeline stall signal to stall the accelerator until both the locks become 0. Note that when the PE writes an updated feature vector into the FVB, it also sends unlock signals to the HDU to set the lock of the corresponding vertex back to 0. Therefore, deadlock will not occur.

6 Experimental Results

6.1 Experimental Setup

6.1.1 Platforms

We conduct experiments based on a state-of-the-art Virtex UltraScale+ xcvu9pflgb2104 FPGA [38]. This FPGA device has 1,182,240 slice LUTs, 2,364,480 slice registers, 6,840 DSPs, and up to 43.3 MB of on-chip RAMs. Two DDR4 chips are connected to the FPGA as the external memory. Each DRAM has 16 GB capacity and a peak access bandwidth of 19.2 GB/s. Our FPGA designs are implemented in RTL using `Verilog` (available at http://www-scf.usc.edu/~shijiezh/FPGA18); no high-level synthesis tool is used.

In order to show that our proposed optimizations are applicable to general purpose processors as well, we also

TABLE 2
Large Real-Life Matrices Used for Experiments

Dataset	# users ($\|U\|$)	# items ($\|V\|$)	# ratings ($\|E\|$)	Density ($\frac{\|E\|}{\|U\|\times\|V\|}$)
Libim [23]	135 K	168 K	17,359 K	7.6×10^{-4}
Netflix [2]	480 K	17 K	100,480 K	1.2×10^{-2}
Yahoo [24]	1,200 K	137 K	460,380 K	2.8×10^{-3}

TABLE 3
Resource Utilization for Various Number of Processing Units

K	LUT	Register	DSP	On-chip RAM	
				Block RAM	UltraRAM
1	7.0 %	4.6 %	3.8 %	1.2 %	37.5 %
2	14.3 %	8.2 %	7.5 %	1.2 %	37.5 %
4	30.7 %	17.2 %	15.1 %	1.2 %	37.5 %
8	63.9 %	33.4 %	30.2 %	1.2 %	37.5 %

TABLE 4
Clock Rate and Power Consumption for Various Number of Processing Units

K	Clock Rate (MHz)	Power (Watt)
1	171	5.8
2	167	7.3
4	161	12.1
8	150	20.1

implement a software-based design on a multi-core platform (see Section 6.6). The multi-core platform has a 16-core Intel Xeon E5-2650 processor running at 2.6 GHz. Each core has a 32 KB L1 cache and a 256 KB L2 cache. All the cores share a 20 MB L3 cache. The multi-core platform is equipped with 128 GB main memory with a peak access bandwidth of 38.4 GB/s.

6.1.2 Datasets and Machine Learning Parameters

We use several large real-life sparse matrices to evaluate our designs. They have been widely used in related works [12], [16], [22], [36]. Table 2 summaries the characteristics of the datasets. In our experiments, the length of each feature vector is 32 (i.e., $H = 32$) with each element represented using IEEE 754 single precision format. We adopt standard learning rate $\alpha = 0.0001$ and regularization parameter $\lambda = 0.02$ [2]. Note that our focus is accelerating the computations of SGD-based MF; tuning the machine learning parameters to improve the prediction accuracy is out of the scope of this paper.

6.1.3 Performance Metrics

We evaluate the performance of our designs based on the following metrics.

- Resource utilization: the percentages of basic FPGA resources utilized by the accelerator, including slice LUT, register, BRAM, UltraRAM, and DSP.
- Power: the power consumption of the FPGA accelerator, including both the static power and dynamic power.
- Execution time: the elapsed time to complete one iteration of the SGD-based MF algorithm; note that this is independent with the values of α and λ.
- Throughput: the number of floating point operations performed per second (GFLOPS).

TABLE 5
Pre-Processing Time

Dataset	Opt. 1	Opt. 2	Opt. 3	Total
Libim	0.4 sec	4.4 sec	1.6 sec	6.4 sec
Netflix	1.0 sec	10.7 sec	5.4 sec	17.1 sec
Yahoo	5.5 sec	42.3 sec	18.9 sec	66.7 sec

TABLE 6
Training Time

Dataset	Total training time	# iterations to converge	T_{exec} per iteration	Root-Mean-Square-Error
Libim	360.8 sec	11,568	0.03 sec	1.05
Netflix	876.4 sec	5,766	0.15 sec	0.72
Yahoo	2536.5 sec	3,714	0.68 sec	0.94

6.2 Resource Utilization, Clock Rate, and Power Consumption

Tables 3 and 4 show the resource utilization, clock rate, and power consumption of our FPGA accelerator for various number of processing units. The reported results are obtained through post-place-and-route simulations using Xilinx Vivado Design Suite 2018.1 [41]. For $K = 8$, the accelerator uses up to 63.9 percent slice LUTs in the FPGA device. Therefore, we could not increase K further to 16 due to the resource limitations. The feature vector buffer (FVB) is organized in 32 banks and the capacity of the FVB is empirically set to 64K feature vectors (32K for user vertices and 32K for item vertices). We did not increase the capacity of the FVB to 128K because we observed that the clock rate degraded below 100 MHz when 75 percent UltraRAMs of the FPGA device were used.

6.3 Pre-Processing Time and Training Time

Tables 5 and 6 report the pre-processing time and training time, respectively. The pre-processing includes the three proposed optimizations in Section 4, and is performed by the multi-core platform introduced in Section 6.1.1. The training is performed by our FPGA accelerator. Note that the pre-processing is performed only once, while the training is an iterative process. Therefore, the pre-processing time can be amortized and is negligible compared with the training time ($< 3\%$ of the training time). In Table 6, we also report the total number of iterations, the execution time for each iteration, and the Root-Mean-Squared-Error (RMSE) (i.e., $\sqrt{\frac{\sum err^2}{|E|}}$) after the training process is completed.

6.4 Performance versus Parallelism

In this section, we vary the number of processing units (K) from 1 to 8 to explore its impact on pipeline stalls, bank conflicts, and throughput performance.

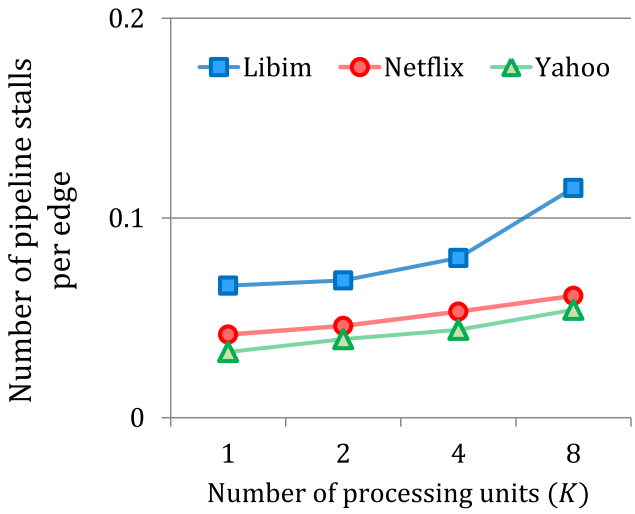


Fig. 8. Number of pipeline stalls per edge for various K.

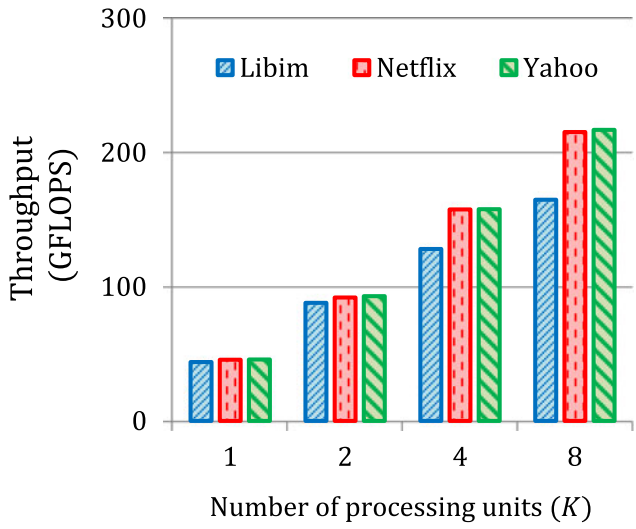


Fig. 10. Throughput for various K.

6.4.1 Pipeline Stalls

Fig. 8 shows *the number of pipeline stalls per edge* for various K. We compute the number of pipeline stalls per edge as the total number of pipeline stalls incurred due to feature vector dependencies divided by the total number of traversed edges; therefore a smaller value corresponds to better performance. We observe that the number of pipeline stalls per edge slightly increases as K increases. This is because when a new matching is to be processed, there can be up to $K \times D$ edges of other matching(s) remaining in the processing units, where D is the pipeline depth of each processing unit. Since the edges belonging to different matchings can have feature vector dependencies, a larger K increases the chances of pipeline stalls. We also observe that the Libim dataset has more pipeline stalls than the other two datasets. The reason is that the Libim dataset is much sparser than the other two datasets (see Table 2), thus resulting in more small matchings that cannot fill up the processing units. When multiple such small matchings are consecutively processed, the pipeline stalls due to feature vector dependencies are more likely to occur.

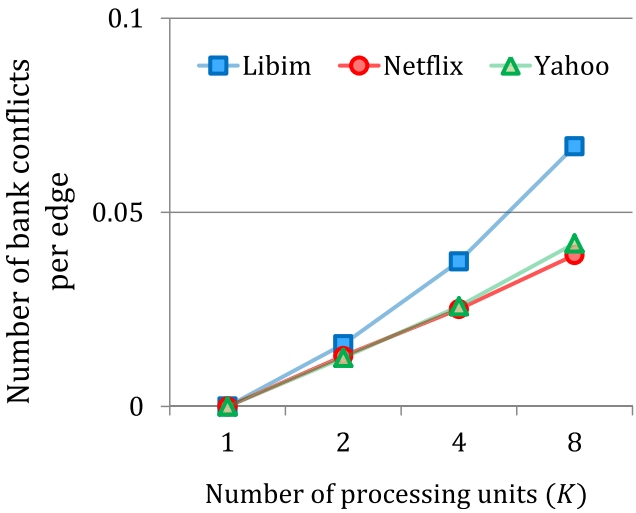


Fig. 9. Number of bank conflicts per edge for various K.

6.4.2 Bank Conflicts

Fig. 9 shows *the number of bank conflicts per edge* for various K, which is computed as the total number of bank conflicts incurred during the processing divided by the total number of traversed edges. This metric indicates the average number of bank conflicts that an edge can result in; thus a smaller value is more desirable. It can be observed that the number of bank conflicts per edge significantly increases as K increases. This is because it becomes more likely that different processing units concurrently access the same bank of the feature vector buffer as K increases. Note that when $K = 1$ (i.e., there is only one processing unit), bank conflict never occurs. Another observation is that the Libim dataset has more bank conflicts per edge than the other two datasets. This is also due to the sparseness of the Libim dataset that leads to a lot of small matchings. When we perform the edge scheduling optimization (i.e., Algorithm 6) for these small matchings, it is very challenging to separate the edges that have bank conflict into distinct batches due to a small number of batches.

6.4.3 Throughput

Fig. 10 shows the throughput performance for various K. We observe that the throughput performance significantly improves as K increases for all the three datasets. For $K = 8$, our FPGA accelerator sustains a high throughput of 165 GFLOPS for Libim, 213 GFLOPS for Netflix, and 217 GFLOPS for Yahoo, respectively. We also observe that the sustained throughput for Libim is lower than Netflix and Yahoo. As explained in Sections 6.4.1 and 6.4.2, this is because the Libim dataset is much sparser than the other two datasets, resulting in more pipeline stalls and bank conflicts per edge.

6.5 Impact of the Optimizations

To show the effectiveness of our proposed three optimizations, we compare our optimized design with several non-optimized FPGA-based baseline designs. All the comparisons are based on $K = 8$.

TABLE 7
Bank Conflict Reduction Due to Optimization 3

Dataset	# bank conflicts per edge		Reduction	T_{exec} per iteration (sec)		Speedup
	Optimized	Baseline		Optimized	Baseline	
Libim	0.067	0.161	2.4×	0.03	0.04	1.3×
Netflix	0.039	0.166	4.2×	0.15	0.23	1.5×
Yahoo	0.042	0.164	3.9×	0.68	1.03	1.5×

TABLE 8
Pipeline Stall Reduction Due to Optimization 2

Dataset	# pipeline stalls per edge		Reduction	T_{exec} per iteration (sec)		Speedup
	Optimized	Baseline		Optimized	Baseline	
Libim	0.115	3.313	28.8×	0.03	0.40	13.3×
Netflix	0.061	3.133	51.3×	0.15	2.19	14.6×
Yahoo	0.054	3.259	60.3×	0.68	10.45	15.4×

6.5.1 Bank Conflict Reduction

We first explore the effectiveness of Optimization 3 (Section 4.2.3) in reducing the number of bank conflicts. Here, the baseline design used for the comparison only performs Optimization 1 and Optimization 2 during the preprocessing. Table 7 summarizes the results of the comparison. We observe that Optimization 3 reduces the number of bank conflicts by 2.4× to 4.2×. As a result, the execution time per iteration is improved by 1.3× to 1.5×.

6.5.2 Data Dependency Reduction

We further explore the impact of Optimization 2 (Section 4.2.2) which aims to reduce the number of pipeline stalls due to feature vector dependencies. For comparison purpose, the baseline design performs Optimization 1 and Optimization 3 only. Table 8 summarizes the effectiveness of this optimization. We observe that the optimized design dramatically reduces the number of bank conflicts by 28.8× to 60.3× and thus achieves 13.3× to 15.4× speedup.

6.5.3 Communication Cost Reduction

Lastly, we study the impact of Optimization 1 (Section 4.2.1) to reduce the communication cost. We define communication cost as the data transfer time between the FPGA and the external memory. The baseline design for the comparison also performs Optimization 2 and Optimization 3; however, when partitioning the input graph into induced subgraphs, the baseline design adopts the widely used vertex-index-based partitioning approach [32] (described in Section 4.2.1) rather than our proposed approach (Algorithm 4). We first compare the partitioning effect of these two approaches. Table 9 lists the maximum size, minimum size, and average size of induced subgraphs with respect to number of edges after the input graph is partitioned. It can be observed that the induced subgraphs obtained by our approach have similar amount of edges, while the induced subgraphs obtained by [32] can vary significantly in size.

Table 10 summarizes the results of the comparison with respect to communication cost. For all the three datasets, the optimized design is able to completely hide the communication cost; while the baseline design can not completely hide the communication cost for Libim and Yahoo datasets. Based on the sufficient condition derived in Section 4.2.1, the required number of edges in each induced subgraph for a complete overlap of computation and communication is 438 K for Libim, 574 K for Netflix, and 574 K for Yahoo, respectively. Since the baseline design has small induced subgraphs which do not satisfy the requirement, the communication cost cannot be completely hidden.

TABLE 9
Comparison of Partitioning Effect

Dataset	Approach	$\|IS\|_{max}$	$\|IS\|_{min}$	$\|IS\|_{avg}$
Libim	This paper	591 K	524 K	579 K
	[32]	703 K	175 K	
Netflix	This paper	6,699 K	6,699 K	6,699 K
	[32]	6,929 K	4,501 K	
Yahoo	This paper	2,487 K	2,288 K	2,465 K
	[32]	3,086 K	288 K	

6.6 Performance of Multi-Core Implementation

Based on our design methodology, we also implement a software-based design on the multi-core platform introduced in Section 6.1.1. To optimize the software-based design, we first partition the input graph into induced subgraphs based on the capacity of the last level cache (i.e., L3 cache); then, each induced subgraph is partitioned into non-overlapping matchings based on Algorithm 5. During the training process, since there is no feature vector dependency

TABLE 10
Communication Cost Reduction

Dataset	Unhidden communication cost per iteration (sec)		T_{exec} per iteration (sec)	
	Opt.	Base.	Opt.	Base.
Libim	0	0.005	0.031	0.036
Netflix	0	0	0.15	0.15
Yahoo	0	0.04	0.68	0.72

TABLE 11
Performance of Our Multi-Core Implementation

Dataset	T_{exec} per iteration (sec)	Throughput (GFLOPS)
Libim	0.35	16.0
Netflix	1.51	21.4
Yahoo	6.55	22.6

among the edges within a matching, each matching can be processed by the 8 cores of the multi-core platform in parallel without any atomic operations. The software-based design is implemented using C language and has been parallelized with 16 threads using multi-threading technique [45]. Table 11 reports the performance of our multi-core implementation for various datasets. We observe that our FPGA accelerator achieves an average speedup of 10× compared with our multi-core implementation.

We further compare the performance of our multi-core implementation with a highly-optimized multi-core implementation, Native [36], which has shown the fastest training speed among existing multi-core implementations [15], [37]. Native implements SGD-based MF on a 24-core Intel E5-2697 processor. It partitions the input training matrix into submatrices, and exploits submatrix-level parallelism to concurrently process the submatrices that do not have feature vector dependencies using distinct CPU cores. However, the submatrices can vary significantly in size, thus resulting in load imbalance among the CPU cores and increasing the synchronization overhead. Table 12 shows the comparison results based on the same dataset (i.e., Netflix). It can be observed that our optimized multi-core implementation achieves 1.3× speedup with 1.5× fewer CPU cores and 2.2× lower memory bandwidth.

6.7 Performance Comparison With State-of-the-Art GPU Implementations

In this section, we compare the performance of our FPGA accelerator with two state-of-the-art GPU implementations [22], [43]. In [22], several scheduling schemes for parallel thread execution on GPU are developed and compared. However, the lock-free static scheduling schemes are not able to efficiently exploit the thousands of cores on the GPU, and the dynamic scheduling schemes require memory locks to handle feature vector dependencies and thus result in significant synchronization overhead. In [43], the GPU design focuses on optimizing the memory performance of GPU by exploiting warp shuffling, memory coalescing, and half-precision (i.e., using 16 bits to represent a floating point number) techniques. Table 13 summaries the comparison results of our FPGA design with [22], [43] for training the same dataset (Netflix). Our design achieves 12.7× and 2.5× speedup compared with [22] and [43], respectively. Note that the speedup is achieved with fewer cores (i.e., processing units), lower clock frequency, and lower memory bandwidth. In addition, the power consumption of our FPGA accelerator is over 10× lower than the thermal design power of the GPU platforms. Therefore, from energy-efficiency perspective (i.e., performance per Watt), our FPGA design achieves even larger improvement.

TABLE 12
Comparison With State-of-the-Art Multi-Core Implementation for Training Netflix Dataset

Approach	Platform		T_{exec} per iteration	Speedup
	# cores	Mem. BW		
[36]	24	86.5 GB/s	2.0 sec	1.3×
This paper	16	38.4 GB/s	1.5 sec	

7 Related Work

7.1 Multi-Core-Based Acceleration

Multi-core-based designs for SGD-based MF usually partition the input matrix into submatrices and use a global table shared by all the CPU threads to schedule the execution of each submatrix [40]. When a thread is idle, it locks the global table and finds an unprocessed submatrix from the table, such that the submatirx has no feature vector dependencies with the submatrices being processed by the other threads; then, the thread updates and unlocks the global table, and sequentially processes all the training data in the submatrix. There are also several multi-core-based graph-processing frameworks that support GD-based MF which can be expressed as a vertex-centric program. Representative examples include GraphLab [15] and GraphMat [16]. However, GD is less efficient than SGD when the volume of training data is large [44]. The reason is that GD needs to generate an update based on the entire training dataset, while SGD generates an update based on a single training data. Nadathur *et al*. observe that GD-based MF runs 40× slower than SGD-based MF for training the Netflix dataset given the same convergence criterion [36].

7.2 GPU-Based Acceleration

GPUs are widely used to accelerate machine learning applications [39]. In [43], ALS-based MF and SGD-based MF are compared based on the GPU platform. The observation is that SGD-based MF results in 4× less training time than ALS-based MF. However, it has been shown that GPUs are not suitable for accelerating SGD-based MF [22], [39], [42]. The main reasons include (1) the fine-grained synchronization of

TABLE 13
Comparison With GPU Implementations for Training Netflix Dataset

Approach	Platform				T_{exec} per iteration	Speedup	Throughput	Improvement
	# cores	Frequency	Mem. BW	Power				
[22]	2880	745 MHz	288 GB/s	235 W	1.90 sec	1.0×	8.6 GFLOPS	1.0×
[43]	3072	1000 MHz	360 GB/s	250 W	0.38 sec	5.0×	171.4 GFLOPS	19.9×
This paper	8	150 MHz	38 GB/s	20 W	0.15 sec	12.7×	213.3 GFLOPS	24.8×

updated feature vectors is very expensive on GPU platforms [39], and (2) the SIMD execution of GPU further inflates the cost of thread divergence when synchronization conflicts occur [22]. In [22], Rashid *et al.* compare several scheduling schemes for parallel execution of SGD on GPU, including lock-based dynamic scheduling schemes and lock-free static scheduling schemes. However, none of these schemes is able to efficiently exploit the GPU acceleration; as a result, the achieved speedup compared with a CPU implementation is very small ($< 1.1\times$). Siede *et al.* [42] investigate the theoretical efficiency of SGD on GPUs, and conclude that fundamental changes in the algorithm are necessary to achieve significant speedup. In [46], a variation of SGD-based MF algorithm called MSGD is proposed. In order to reduce feature vector dependencies and exploit more GPU acceleration, MSGD updates only the feature vector of either user or item per input rating (SGD updates both the feature vectors); the work [46] mathematically proves that such modification does not significantly impact the prediction accuracy.

7.3 FPGA-Based Acceleration

There have not been many efforts to exploit FPGA to accelerate SGD-based MF. In [47], an FPGA accelerator for SGD-based deep learning algorithms is developed. In order to improve hardware efficiency, the accelerator is designed based on a model which allows asynchronous updates (i.e., read-after-write data hazard is acceptable) and uses low-precision computations (i.e., floating-point numbers are represented using low-precision fixed-point data type). However, it is highly likely that applying such model to SGD-based MF will negatively impact the quality of the training output. To the best of our knowledge, our proposed accelerator is the first FPGA-based design to accelerate the training process of SGD-based MF for large input matrices.

8 Generality of Our Design

Our holistic design approach is general enough to be adapted to various machine learning and parallel computing problems. The generality of our design is in four aspects. First, the accelerator can be easily modified to train matrix factorization models that require tuning parameters; for example, the learning rate and regularization parameter can serve as inputs to the accelerator and be adjusted in each iteration based on the achieved accuracy improvement [2]. Second, the on-chip shared memory of GPU platform is also organized as memory banks and suffers bank conflict issue; our approach to reduce bank conflicts is applicable to GPU platform as well, which can enable a more efficient batching of input data and thus improve memory performance. Third, our bipartite graph partitioning approach can be used to partition sparse matrices in such a way that the obtained submatrices have sufficient computations; this enables efficient load balancing when mapping the computations of submatrices to distinct parallel computing units. Lastly, the implementation of our locking mechanism can be directly used to design architecture for many applications that require to incrementally update data during processing; examples include other SGD-based training problems [47] and networking applications [48].

9 Conclusion and Future Work

In this paper, we presented an FPGA-based accelerator to speedup the training process of SGD-based MF. The accelerator consisted of parallel processing units with a shared on-chip feature vector buffer. To optimize the performance, we proposed three novel optimizations by using a bipartite graph representation of MF. Our focus was to obtain simple and fast heuristics based on identifying sufficient conditions for significant acceleration of the SGD-based MF on FPGA. By holistically considering the architectural characteristics of the FPGA platform, the proposed optimizations resulted in a complete overlap of communication and computation, up to 60.3× data dependency reduction, and 4.2× bank conflict reduction. As a result, our FPGA accelerator sustained a high throughput of up to 217 GFLOPS for training large real-life sparse datasets. Compared with the state-of-the-art GPU implementations, our FPGA accelerator achieved up to 12.7× speedup and 24.8× throughput improvement. We also demonstrated that the proposed optimizations were applicable to general purpose processors. Our optimized software-based design achieved 1.3× speedup compared with the state-of-the-art multi-core implementation.

In the future, we will explore multi-FPGA architectures, in which each FPGA device employs our FPGA accelerator, to further reduce the training time and handle even larger input matrices. We are also interested in exploring high-level synthesis tools to generate the architecture of our design and comparing the performance.

Acknowledgments

This work was supported by Intel Strategic Research Alliance funding. This work was also supported in part by the U.S. National Science Foundation under Grants ACI-1339756, CNS-1643351, and OAC-1911229. Equipment Grant from Xilinx, Inc. is gratefully acknowledged.

References

[1] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *IEEE Comput.*, vol. 42, no. 8, pp. 30–37, Aug. 2009.
[2] Netflix update: Try this at home, 2006. [Online.] Available: http://sifter.org/simon/journal/20061211.html
[3] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. Int. Symp. Field-Programmable Gate Arrays February*, 2015, pp. 161–170.
[4] D. H. Noronha, P. Leong, and S. Wilton, "Kibo: An open-source fixed-point tool-kit for training and inference in FPGA-based deep learning networks," in *Proc. Int. Parallel Distrib. Process. Symp. Workshop*, 2018, pp. 178–185.
[5] M. Blott *et al.*, "Scaling neural network performance through customized hardware architectures on reconfigurable logic," in *Proc. Int. Conf. Comput. Des.*, 2017, pp. 419–422.
[6] T. B. Preußer, G. Gambardella, N. J. Fraser, and M. Blott, "Inference of quantized neural networks on heterogeneous all-programmable devices," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2018, pp. 833–838.
[7] R. Zhao *et al.*, "Accelerating binarized convolutional neural networks with software-programmable FPGAs," in *Proc. Int. Symp. Field-Programmable Gate Arrays*, 2017, pp. 15–24.
[8] T. Geng *et al.*, "FPDeep: Acceleration and load balancing of CNN training on FPGA clusters," in *Proc. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2018, pp. 81–84.
[9] S. Zhou, R. Kannan, Y. Min, and V. K. Prasanna, "FASTCF: FPGA-based accelerator for stochastic-gradient-descent-based collaborative filtering," in *Proc. Int. Symp. Field-Programmable Gate Arrays*, 2018, pp. 259–268.

[10] H. F. Yu, C. J. Hsieh, H. Yun, S. Vishwanathan, and I. S. Dhillon, "A scalable asynchronous distributed algorithm for topic modeling," in *Proc. Int. Conf. World Wide Web*, 2015, pp 1340–1350.
[11] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2014, pp. 1532–1543.
[12] M. Zhang, Y. Wu, K. Chen, X. Qian, X. Li, and W. Zheng, "Exploring the hidden dimension in graph processing," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 285–300.
[13] W. Tan, L. Cao, and L. Fong, "Faster and cheaper: Parallelizing large-scale matrix factorization on GPUs," in *Proc. Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2016, pp. 219–230.
[14] H. Yu, C. Hsieh, S. Si, and I. Dhillon, "Parallel matrix factorization for recommender systems," *J. Knowl. Inf. Syst.*, vol. 41, pp. 793–819, 2014.
[15] GraphLab collaborative filtering library, 2006. [Online.] Available: http://select.cs.cmu.edu/code/graphlab/pmf.html
[16] N. Sundaram *et al.*, "GraphMat: High performance graph analytics made productive," in *Proc. VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.
[17] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proc. Int. Symp. Microarchit.*, 2016, pp. 1–13.
[18] S. Zhou, C. Chelmis, and V. K. Prasanna, "Accelerating large-scale single-source shortest path on FPGA," in *Proc. Int. Parallel Distrib. Process. Symp. Workshop*, 2015, pp. 129–136.
[19] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk, "A reconfigurable computing approach for efficient and scalable parallel graph exploration," in *Proc. Int. Conf. Appl.-Specific Syst. Archit. Processors*, 2012, pp. 8–15.
[20] S. Zhou, C. Chelmis, and V. K. Prasanna, "High-throughput and energy-efficient graph processing on FPGA," in *Proc Int. Symp. Field-Programmable Custom Comput. Mach.*, 2016, pp. 103–110.
[21] S. Zhou, C. Chelmis, and V. K. Prasanna, "Optimizing memory performance for FPGA implementation of PageRank," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs*, 2015, pp. 1–6.
[22] R. Kaleem, S. Pai, and K. Pingali, "Stochastic gradient descent on GPUs," in *Proc. Workshop Gen. Purpose Process. Using GPUs*, 2015, pp. 81–89.
[23] L. Brozovsky and V. Petricek, "Recommender system for online dating service," 2007. [Online.] Available: https://pdfs.semanticscholar.org/1a42/f06f368cf9b2ba8565e81d8e048caa5c2c9e.pdf
[24] Ratings and classification data, 2006. [Online.] Available: https://webscope.sandbox.yahoo.com/catalog.php?datatype=r
[25] C. E. LaForest and J. G. Steffan, "Efficient multi-ported memories for FPGAs," in *Proc. Int. Symp. Field Programmable Gate Arrays*, 2010, pp. 41–50.
[26] C. E. Laforest, M. G. Liu, E. R. Rapati, and J. G. Steffan, "Multi-ported memories for FPGAs via XOR," in *Proc. Int. Symp. Field-Programmable Gate Arrays*, 2012, pp. 209–218.
[27] S. N. Shahrouzi and D. G. Perera, "An efficient embedded multi-port memory architecture for next-generation FPGAs," in *Proc. Int. Conf. Appl.-Specific Syst. Archit. Processors*, 2017, pp. 83–90.
[28] S. Zhou, R. Kannan, and V. K. Prasanna, "Accelerating low rank matrix completion on FPGA," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs*, 2017, pp. 1–7.
[29] UltraRAM: Breakthrough embedded memory integration on UltraScale+ Devices, 2016. [Online.] Available: https://www.xilinx.com/support/documentation/white_papers/wp477-ultraram.pdf
[30] R. J. Wilson, "Introduction to graph theory," Oliver and Boyd, ISBN 0-582-24993-7, p. 180, 1996.
[31] E. Nurvitadhi *et al.*, "GraphGen: An FPGA framework for vertex-centric graph computation," in *Proc. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2014, pp. 25–28.
[32] R. Pearce, M. Gokhale, and N. M. Amato, "Faster parallel traversal of scale free graphs at extreme scale with vertex delegates," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 549–559.
[33] R. Chen, J. Shi, B. Zang, and H. Guan, "Bipartite-oriented distributed graph partitioning for big learning," in *Proc. Asia-Pacific Workshop Syst. Article*, 2014, pp. 1–7.
[34] H. E. Yantir, S. Bayar, and A. Yurdakul, "Efficient implementations of multi-pumped multi-port register files in FPGAs," in *Proc. Euromicro Conf. Digit. Syst. Des.*, 2013, pp. 185–192.
[35] J. Wawrzynek, K. Asanovic, J. Lazzaro, and Y. Lee, "Banked multiport memory," 2010. [Online.] Available: https://inst.eecs.berkeley.edu/cs250/fa10/lectures/lec08.pdf
[36] N. Satish *et al.*, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proc. ACM Int. Conf. Manage. Data*, 2014, pp. 979–990.
[37] A. Lenharth, "Parallel programming with the Galois system," 2014. [Online.] Available: http://iss.ices.utexas.edu/projects/galois/downloads/europar2014-tutorial.pdf
[38] Virtex UltraScale+ FPGA data sheet, 2019. [Online.] Available: https://www.xilinx.com/support/documentation/data_sheets/ds923-virtex-ultrascale-plus.pdf
[39] J. Dean *et al.*, "Large scale distributed deep networks," in *Proc. Neural Inf. Process. Syst. Conf.*, 2012, pp. 1232–1240.
[40] W. Chin, Y. Zhuang, Y. Juan, and C. Lin, "A fast parallel stochastic gradient method for matrix factorization in shared memory systems," *ACM Trans. Intell. Syst. Technol.*, vol. 6, no. 1, 2015, Art. no. 2.
[41] Vivado design suite, 2020. [Online.] Available: https://www.xilinx.com/products/design-tools/vivado.html
[42] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "On parallelizability of stochastic gradient descent for speech DNNs," in *Proc. Int. Conf. Acoust. Speech Signal Process.*, 2014, pp. 235–239.
[43] X. Xie, W. Tan, L. L. Fong, and Y. Liang, "CuMF_SGD: Parallelized stochastic gradient descent for matrix factorization on GPUs," in *Proc. Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2017, pp. 79–92.
[44] L. Bottou, "Stochastic gradient descent tricks," in *Neural Networks, Tricks of the Trade*. Berlin, Germany: Springer, 2012, pp. 430–445.
[45] OpenMP, 2018. [Online.] Available: https://computing.llnl.gov/tutorials/openMP/
[46] H. Li, K. Li, J. An, and K. Li, "MSGD: A novel matrix factorization approach for large-scale collaborative filtering recommender systems on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 7, pp. 1530–1544, Jul. 2018.
[47] C. D. Sa, M. Feldman, C. Re, and K. Olukotun, "Understanding and optimizing asynchronous low-precision stochastic gradient descent," in *Proc. Int. Symp. Comput. Archit.*, 2017, pp. 561–574.
[48] D. Tong and V. K. Prasanna, "Sketch acceleration on FPGA and its applications in network anomaly detection," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 4, pp. 929–942, Apr. 2018.

Shijie Zhou received the BS degree in electrical engineering from Zhejiang University, Hangzhou, China, in 2010, and the PhD degree in electrical engineering from the University of Southern California (USC), Los Angeles, California, in 2018. He is currently working with Microsoft as a hardware engineer. His research interests include parallel computing and reconfigurable computing for accelerating graph algorithms. He has published more than 25 papers in peer reviewed conference and journal proceeding. He is a member of the ACM.

Rajgopal Kannan received the BTech degree in computer science and engineering from IIT Bombay, Mumbai, Maharashtra, India, in 1991, and the PhD degree in computer science from the University of Denver, Denver, Colorado, in 1996. He is currently a computer scientist with the Army Research Lab in the Computing Architectures Branch and a research adjunct professor in electrical engineering with the University of Southern California. He was formerly a professor with the Department of Computer Science, Louisiana State University (2000–2015). His academic research was funded by DARPA, NSF, and DOE and he has published more than 150 research papers in international journals and conferences with two patents awarded in the area of network optimization. His research interests include at the intersection of graph analytics, machine learning, and edge computing - enabling application acceleration at the edge on low power devices, for example using Software-Defined Memory for memory bound applications. He is also interested in cyber-physical systems, especially data-driven models, and analytics driving Smartgrid optimization and control.

Viktor K. Prasanna (Fellow, IEEE) received the BS degree in electronics engineering from the Bangalore University, Bengaluru, Karnataka, India, the MS degree from the School of Automation, Indian Institute of Science, and the PhD degree in computer science from Pennsylvania State University, State College, Pennsylvania. He is Charles Lee Powell chair in engineering and is professor of electrical and computer engineering and professor of computer science with the University of Southern California (USC). He is the director of the Center for Energy Informatics (CEI) and the USC-Infosys Center for Advanced Software Technologies (CAST), USC. He is an associate member of the Center for Applied Mathematical Sciences (CAMS) and leads the Parallel Computing/FPGA (fpga.usc.edu) and Data Science (dslab.usc.edu) Labs. The Parallel Computing/FPGA Lab is focused on innovative applications of FPGAs for accelerating computations in diverse areas while the Data Science Lab is exploring novel applications of machine learning in complex engineered systems. He was an associate director of the USC Chevron Center of Excellence for Research and Academic Training on Interactive Smart Oilfield Technologies (Cisoft). His research interests include high-performance computing, parallel and distributed systems, reconfigurable computing, cloud computing, and smart energy systems. He served as the editor-in-chief of the *IEEE Transactions on Computers* during 2003–06. Currently, he is the editor-in-chief of the *Journal of Parallel and Distributed Computing*. He was the founding chair of the IEEE Computer Society Technical Committee on Parallel Processing. He is the steering chair of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) and IEEE International Conference on High Performance Computing (HiPC). He received the W. Wallace McDowell Award from the IEEE Computer Society, in 2015 for his contributions to reconfigurable computing. He received an Outstanding Engineering Alumnus Award from the Pennsylvania State University, in 2009 and the Distinguished Alumnus Award from the University Visveswaraya College of Engineering (UVCE) of Bangalore University, in 2017. He was appointed distinguished professor, Beihang University (BUAA), Beijing, PRC, in 2018. He received a 2019 Distinguished Alumnus Award from the Indian Institute of Science (IISc). His work on regular expression matching received one of the most significant papers in FCCM during its first 20 years award in 2013. He is a fellow of the ACM and the American Association for Advancement of Science (AAAS).

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Bandwidth-Aware Dynamic Prefetch Configuration for IBM POWER8

Carlos Navarro, Josué Feliu, Salvador Petit, *Member, IEEE*, Maria E. Gómez, and Julio Sahuquillo, *Member, IEEE*

Abstract—Advanced hardware prefetch engines are being integrated in current high-performance processors. Prefetching can boost the performance of most applications, however, the induced bandwidth consumption can lead the system to a high contention for main memory bandwidth, which is a scarce resource in current multicores. In such a case, the system performance can be severely damaged. This article characterizes the applications' behavior in an IBM POWER8 machine, which presents many prefetch settings, varying the bandwidth contention. The study reveals that the best prefetch setting for each application depends on the main memory bandwidth availability, that is, it depends on the co-running applications. Based on this study, we propose Bandwidth-Aware Prefetch Configuration (BAPC) a scalable adaptive prefetching algorithm that improves the performance of multi-program workloads. BAPC increases the performance of the applications in a 12, 15, and 16 percent of 6-, 8-, and 10-application workloads over the IBM POWER8 default configuration. In addition, BAPC reduces bandwidth consumption in 39, 42, and 45 percent, respectively.

Index Terms—Prefetch engine, prefetch settings, performance measures

1 Introduction

Hardware data prefetching is an effective technique to hide the long memory latency and can greatly improve the performance of many applications. Much research [1], [2], [3], [4], [5] focusing on monolithic processors has shown the high performance enhancements prefetch approaches can bring. Because of this fact, many works [6], [7], [8], [9], [10], [11] have focused on data prefetching for multicores over the last decade. Research has concentrated on either parallel workloads or multi-program workloads consisting of a set of single-threaded applications, each one running on a different core, and exhibiting a different memory access pattern. To deal with this issue, some approaches, like *sandbox prefetching* [6], propose to implement a set of prefetchers and select the best prefetcher for each core, depending on the running application.

The major shortcoming of prefetching is the introduced interference due to it is a speculative technique. In other words, prefetch requests compete with regular requests for memory resources, which can adversely impact on performance. This issue has been addressed in some approaches [7], [8], [9], [10], [11]. Despite the interference, the important performance gains shown by prefetching in multicores over the last decade have led processors manufacturers to integrate complex prefetch engines in modern processors. These prefetchers, especially in high-performance servers, are deployed with a wide set of prefetch settings aimed at capturing the different access patterns that applications exhibit. Thanks to this hardware support, recent research has focused on commercial machines.

LibPRISM [12] focuses on dynamically selecting at runtime the best prefetcher of an IBM POWER8 to enhance the system performance when running a single parallel workload. In [13] an *adaptive* prefetcher is proposed for the IBM POWER7 processor, where the prefetch *aggressiveness* of each application can be throttled down and up. The study presents results when running just two benchmarks. This approach allows the prefetching to achieve good performance, however, it does not scale with the number of cores. Probably because of this fact, in a posterior work [14] by the same authors, a more scalable approach is devised but at the cost of limiting the prefetching flexibility. Instead of an adaptive prefetching, authors rely on a less efficient but much more simple *on/off* approach.

In order to make a prefetch approach scalable and improve performance when a large number of applications is running on the system, the approach must trade off performance and memory bandwidth consumption. To illustrate this claim, Fig. 1 presents two cases studied on a 10-core IBM POWER8 machine with the prefetching disabled (Off) and the *default prefetch setting* or simply default prefetcher (Def). Figs. 1a and 1b present the IPC of each application running in the mix and in isolation of two workloads consisting of 6 and 10 applications, respectively. The figures also plot the bandwidth (red mark and right y-axis) of the applications when running in the workload with the two studied prefetch configurations. The top-right data in the figure reports the average (geo. mean) IPC of the applications that form the workload and their aggregated bandwidth. As observed, with 6 applications the default prefetch setting improves the performance in almost all the applications, so improving the

• *The authors are with the Departamento de Informática de Sistemas y Computadores, Universitat Politècnica de València, 46022 Valencia, Spain. E-mail: carnase1@inf.upv.es, {jfeliu, spetit, megomez, jsahuqui}@disca.upv.es.*

Manuscript received 8 Aug. 2019; revised 6 Mar. 2020; accepted 17 Mar. 2020. Date of publication 23 Mar. 2020; date of current version 14 Apr. 2020. (Corresponding author: Carlos Navarro Serra.) Recommended for acceptance by M. Guo. Digital Object Identifier no. 10.1109/TPDS.2020.2982392

1045-9219 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

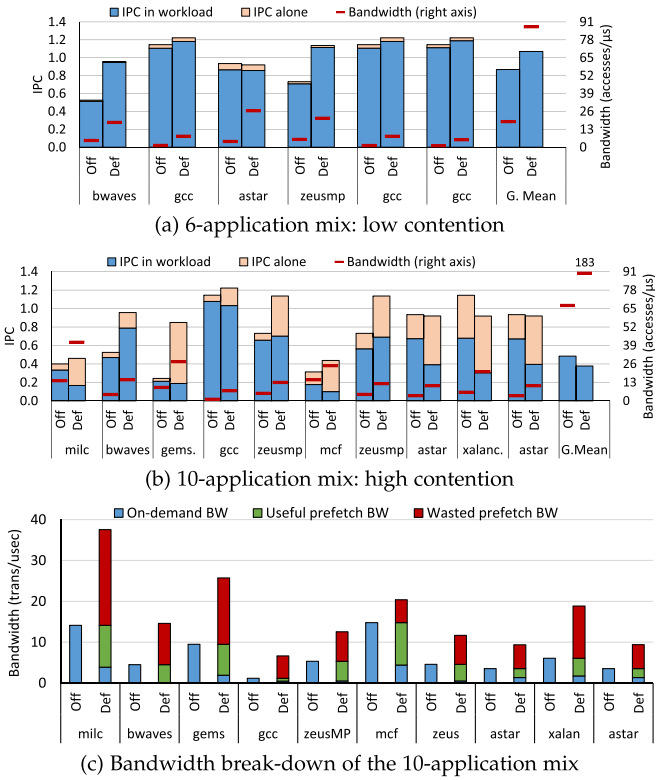


Fig. 1. Example of mixes exhibiting (a) low and (b) high main memory bandwidth contention. The last column reports the average (geo. mean) IPC for the applications of the workload and their aggregated main memory bandwidth, (c) breaks down the bandwidth of the 10-application mix.

overall IPC by 23 percent. In this case, it can be appreciated a relatively low bandwidth consumption, which overall amounts to 18.6 and 87.4 accesses/μs, with the prefetcher disabled and with the default prefetch setting, respectively.

In contrast, when running 10 applications, the default prefetch setting achieves on average worse performance (by 15 percent drop) than no prefetching. Note that most of the performance benefits that prefetching could provide as observed in the individual execution of the applications disappear due to bandwidth contention when applications run in the mix. In this case the default prefetch setting consumes on average more than twice the bandwidth than no prefetching, aggregating 182.8 accesses/μs versus 67.2 accesses/μs with no prefetching. This high amount of aggregated bandwidth consumption of the default prefetch setting increases the inter-application interference making the memory bus a major performance bottleneck.

Fig. 1c breaks down the bandwidth consumed by the applications of the 10-application workload into on-demand bandwidth, useful prefetch bandwidth (i.e., saves memory access latency), and wasted prefetch bandwidth (i.e., due to data not accessed by the applications). This breakdown shows that, despite prefetches have a positive effect on the performance of most applications (on-demand bandwidth is reduced), for many applications more than half of the bandwidth consumed by prefetches is wasted. The wasted bandwidth negatively impacts on the performance of the applications and their co-runners. As the figure shows, in general, the applications with highest bandwidth consumption (e.g., *milc*, *gemsFDTD*) waste more bandwidth and suffer higher performance degradation due to contention. Moreover, contention impacts on the performance of the co-running applications that access main memory, whose IPC when running in the workload is noticeably lower than when running alone with the same prefetch setting.

In this work we propose BAPC (Bandwidth-Aware Prefetch Configuration), an adaptive and scalable bandwidth-aware prefetch configuration approach for the IBM POWER8. BAPC focuses on multi-program workloads composed of single-threaded applications and seeks to dynamically select the best prefetch setting for each application, considering the tradeoff between performance and bandwidth consumption each prefetch configuration presents for each particular application.

The memory bandwidth consumed by an application mainly depends on the performance of the prefetch algorithm for a given application (e.g., coverage, accuracy and timeliness), and on the inter-application interference. LLC interference refers to memory requests (regular and prefetches) competing among them for cache space, thus inaccurate prefetches–specially in aggressive prefetch configurations–can pollute the cache causing the eviction of live cache blocks being used by other applications [15]. To provide a sound understanding of how different levels of interference may affect individual per application performance, this work characterizes, varying the interference level, the impact of distinct prefetch settings on both per-application performance and consumed bandwidth. We found that the *best prefetch setting* for a given application does not depend only on the application itself, but also on the inter-application interference. This claim, illustrated in this work through a rigorous characterization study, differs from the conclusions drawn in [14].

This paper makes three main contributions:

- We characterize applications varying the bandwidth requirements of the competing applications, and we show that the prefetch setting that mostly impacts on performance mainly depends on the bandwidth availability.
- We propose a scalable adaptive prefetch configuration approach that is able to improve the performance of multi-program workloads consisting on a high number of applications.
- The proposed scheme improves performance regardless of the number of running applications and amount of bandwidth the applications request.

The remainder of this paper is organized as follows. Section 2 describes the related work. Section 3 introduces the hardware platform. Section 4 discusses the characterization study. Section 5 presents the BAPC proposal, and Section 6 evaluates its results. Finally, Section 7 presents some concluding remarks.

2 Related Work

Recent research work has focused on prefetching on commercial machines. In [16], authors propose a methodology based on machine learning to select which of the 4 prefetchers available in an Intel Core 2 Quad CPU must be turned on or off. Thus, they select one configuration out of the 16 possible settings. One important difference with respect to

our work is that these processors do not allow graduating the prefetchers, they only allow switching them on and off.

Other works focus on recent IBM POWER processors. Some works [12], [17] have focused on dynamically selecting at run-time the best prefetch setting to enhance the performance of parallel workloads when running a single application on an IBM POWER8. PATer [17] proposes a methodology that dynamically adjusts the prefetch setting in the IBM POWER8. This approach uses performance counters as inputs for machine learning strategies to improve the accuracy of the prefetching models. A background daemon is employed to make use of the prediction models to decide the best prefetch setting for each application running on the processor. Unlike our work, this proposal focuses exclusively on parallel workloads. Finally, LibPRISM [12], also focusing on parallel workloads, manages the SMT level and the prefetch settings in the IBM POWER8.

Other works [13] focus on *adaptive* prefetching to select the best prefetch setting for single-threaded multi-program workloads, similarly to our work. In [13], authors present an adaptive prefetcher capable of boosting the performance by leveraging the prefetch configuration with respect to the default prefetch configuration of the IBM POWER7. However, it is mainly driven by performance so that the prefetch settings are mainly chosen depending on the performance each prefetch setting is able to achieve for the target application. Although this strategy allows the prefetcher to achieve reasonable performance, it fails in that it does not scale with the number of cores, in particular it is evaluated just with two running applications. Mainly due to this fact, in a posterior work [14], these authors face a more scalable approach to configure the prefetch engine in the IBM POWER7 processor with the aim of allocating bandwidth dynamically to the co-running applications. This approach simply activates or deactivates the individual per-core prefetchers of the different threads. Authors claim that they explored other prefetching configurations with an intermediate aggressiveness but did not observe any benefit. Nevertheless, we show that significant performance benefits can be observed, at least in the IMB POWER8. Based on this observation, unlike [14], our approach selects the best prefetch setting instead of only switching the prefetch engine on or off.

Previous or contemporary with research on commercial machines, many research work has been carried out on simulation frameworks focusing on regulating the prefetcher aggressiveness. A representative subset of these works is [18], [19], [8],[20], [21], [22], [23], and [24]. In a similar way, *sandbox prefetching* [6], propose to implement a set of prefetchers and, at run-time, compare the prefetch patterns generated by each prefetcher with the memory accesses of the application to select the prefetcher that better fits the application memory access pattern. In essence, the problem might look similar to the one we address but a comparison cannot be done mainly due to implementation issues. The sandbox prefetching is implemented in a simulation environment. The meachanism implements a set of independent prefetchers and, for each interval, it gathers the memory access patterns and estimates the performance of each prefetcher included in the sandbox in order to choose the best one. This simulation approach considers extra hardware not deployed on current commercial processors. The prefetcher of the IBM POWER8, however, consists of 9 independent *but complementary* fields which amount 25 bits. This gives a huge amount of configurations which makes the sandbox approach impractical. Nevertheless, the key problem from a comparison perspective is that it is not possible to simulate or measure how different prefetch configurations behave in a real processor in the same interval since only one can be active at the same time.

3 Experimental Platform and Prefetcher

The results presented in this work have been carried out on an IBM Power System S812L with a 10-core IBM POWER8 processor working at 3.69 GHz. Each core implements a 64 KB L1 data cache and a 512 KB L2 cache. The processor has a 80MB Last Level Cache (LLC), shared among all the cores, and is equipped with a 32GB off-chip DRAM module. The OS installed in the system is Ubuntu Linux 14.04 with kernel version 4.0.2.

The processor allows the user to configure the prefetcher through the *Data Streams Control Register* (DSCR) [25]. There is one register to define the prefetch setting of each running thread. Each register uses 25 bits to configure the prefetcher along 9 fields.

Because exploring all possible prefetching configurations (2^{25}) could become a tedious and though task, in this work we focus on the parameters with higher impact on the system performance. To this end, we studied each DSCR field in isolation and found that the *load stream disable* (LSD), the *default prefetch depth* (DPDF), and the *prefetch urgency* (URG) are the features that most affect the performance and bandwidth consumption of the studied applications. Other researchers reached similar conclusions working on parallel workloads [12], [17].

The LSD feature allows detecting bursts of loads to prefetch them, and can only be enabled or disabled. We always keep it enabled, since disabling LSD is equivalent to disabling the prefetcher. DPFD and URG are configured with a 3-bit DSCR field each. The DPFD field selects the prefetch depth, which ranges from 2 (shallowest) to 7 (deepest). Setting DPFD to 1 indicates zero depth (i.e., prefetch disabled, regardless of any other DSCR field), while setting it to 0 selects the default depth (4). The URG field indicates how quickly the prefetch depth can be reached. Prefetch urgency can vary from 1 (not urgent) to 7 (most urgent). Analogously to the DPFD field, setting URG to 0 selects the default urgency (4). By default, all the DCSR bits are 0, which enables the LSD characteristic and sets the prefetch urgency and depth to their default values, that is, URG = 4 and DPFD = 4.

4 Characterization Analysis

This section characterizes the applications when running in isolation in terms of performance and bandwidth utilization with different prefetch settings. We study the extreme (i.e., the lowest and the highest) values for both urgency and depth. That is, U1D2, U1D7, U7D2 and U7D7, where the UxDy settings means URG $= x$ and DPFD $= y$. In addition, we also evaluate the default prefetch setting (DEF) and a setting where prefetech is disabled (OFF).

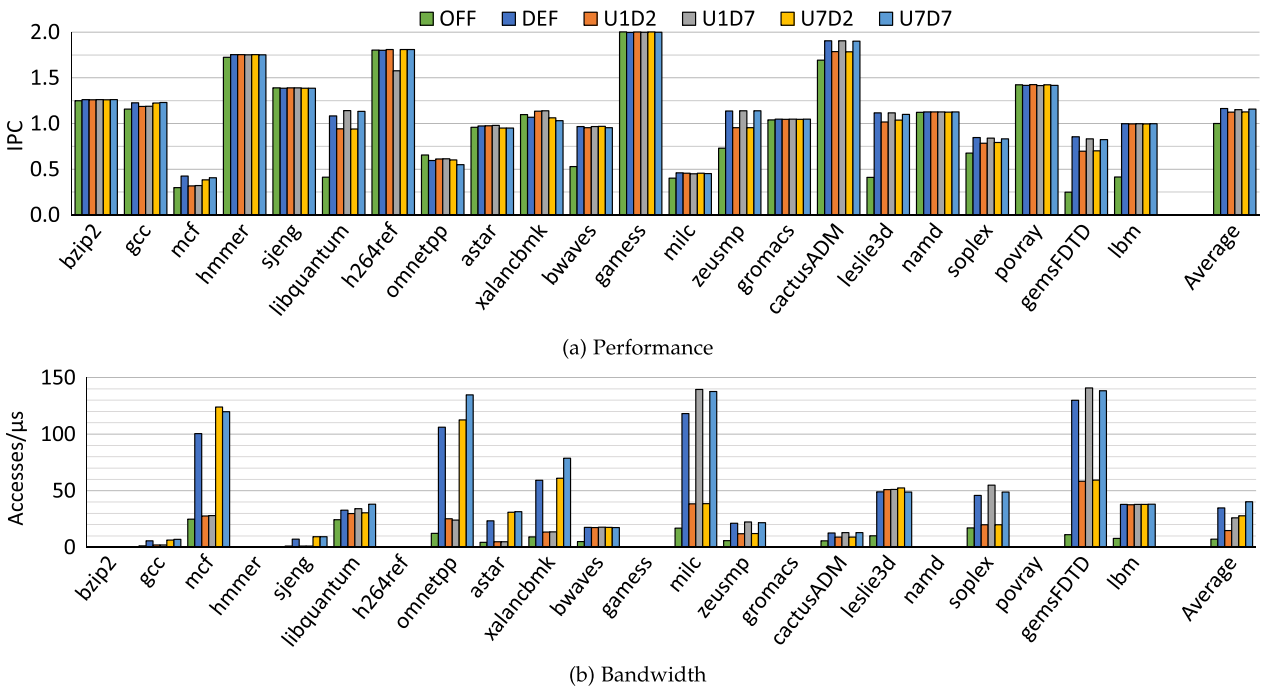


Fig. 2. Impact of the prefetch setting on performance and bandwidth of applications in isolated execution.

4.1 Performance and Memory Bandwidth Utilization With no Inter-Application Bandwidth Contention

Fig. 2a presents the IPC achieved by each application when running alone in the system across the studied prefetch settings. Since, in these experiments, each application runs in isolation, there is no inter-application interference, hence the contention for memory bandwidth is the lowest that applications can experience.

From these results, three main observations can be made. First, although the performance of some applications, such as `gamess` or `povray`, is not affected by prefetching, the performance of many others, such as `libquantum`, `zeusmp`, `cactusADM`, `leslie3d` and `gemsFDTD`, highly improves when prefetching is enabled. The former are typically referred as *prefetch friendly* application and the latter as *prefetch unfriendly*. These results show that it is crucial to keep prefetch enabled for *prefetch friendly* applications. Second, in general, the *prefetch depth* parameter has a higher impact on performance than urgency (e.g., in `libquantum` and `zeusmp`) when the application runs alone in the system. A deeper prefetch depth prefetches more cache lines from the main memory and, when running in isolation, usually improves performance even when the prefetch accuracy reduces since there is enough LLC space and main memory bandwidth to deal with the inaccurate prefetches. Nevertheless, there are applications, such as `gcc` and `mcf`, that are more affected by the urgency parameter. These applications benefit more from speeding up prefetches so that data can move quicker to the upper levels of the cache hierarchy than from issuing more prefetches. Third, the performance achieved by the best performing prefetch configuration hardly outperforms the default configuration, with just a few exceptions in some applications like `libquantum` and `xalancbmk` where the default prefetch setting performs slightly worse.

As shown in the example of Section 1, the main memory bandwidth is a critical resource that must be efficiently handled for performance in current multicores. To address this issue, this section nalyzes the relationship between the prefetch settings and the consumed bandwidth. Fig. 2b shows the bandwidth each application consumes when running in isolation, quantified in main memory accesses per μs, varying the prefetch setting. Three main observations can be made. First, bandwidth strongly varies depending on the prefetch setting, even for those applications whose performance is not significantly affected (e.g., `milc`). As observed, depth is the parameter that impacts the most on the consumed bandwidth. Second, some prefetch settings achieve IPC improvements at the cost of increasing the bandwidth consumption (e.g., DEF, U1D7, and U7D7 in `gemsFDTD`), which in many cases does not justify the performance gains. Third, there are situations where a higher bandwidth consumption translates into performance drops. For instance, in `xalancbmk`, the settings with highest bandwidth consumption (U7D2 and U7D7) offer lower performance than the remaining settings. This is because in these settings prefetches have a high urgency but also very low accuracy, so bringing from memory a high number of non-useful prefetches. It can be appreciated that the default prefetch setting is clearly affected by the discussed shortcomings. For instance, in `milc`, the DEF's bandwidth consumption triples those of U1D2 and U7D2 while achieving similar performance; and in `xalancbmk`, DEF achieves lower performance than U1D2 and U1D7 in spite of its bandwidth consumption is more than 4 times higher.

This section has characterized the behavior of the applications of the SPEC CPU2006 benchmark suite when running in isolation. The study has also been carried out for the SPEC CPU20017 benchmarks, observing neither additional behaviors nor conclusions from the purposes of this work. Thus SPEC CPU2017 characterization results are not shown neither in isolation nor in other studied bandwidth contention scenarios, which are discussed below. Nevertheless, for the sake of completeness, the experimental results (see Section 6)

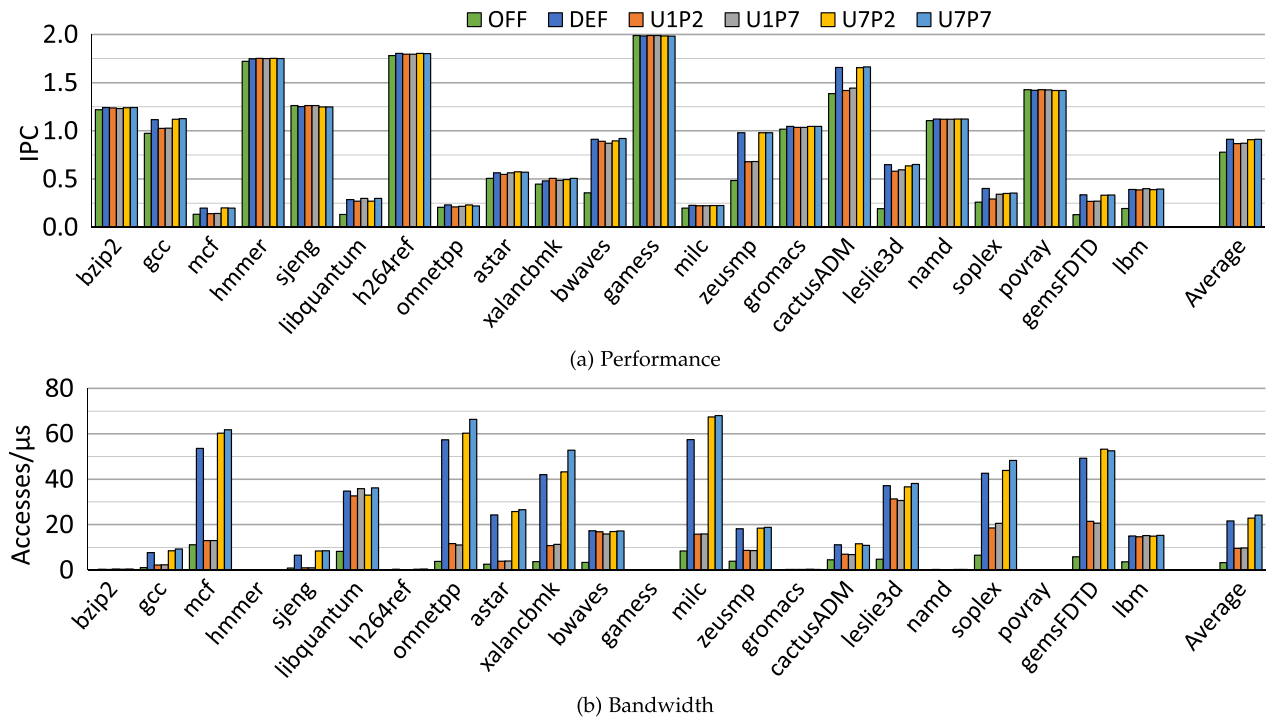


Fig. 3. Impact of the prefetch setting on performance and bandwidth of applications in the highest bandwidth contention.

include applications randomly chosen from both SPEC CPU2006 and CPU2017 suites.

To sum up, this study illustrates that, for some applications and prefetch configurations, an important increase in the bandwidth consumption due to more triggered prefetches only improves performance marginally. In some cases, such as *xalancbmk*, the most aggressive prefetch configurations even degrade performance. Consequently, since the main memory bandwidth is an scarce resource and can quickly become a performance bottleneck, it is important to configure the prefetch aggressiveness based on the application's benefit of prefetch to limit bandwidth consumption.

4.2 Performance and Memory Bandwidth Utilization With Highest Bandwidth Contention

This section evaluates the impact of prefetching on the *highest* main memory bandwidth contention scenario. To this end, each application is executed with 3 instances of the microbenchmark designed in [26] (see Section 4.3.1) that, when running all three together, consume the entire memory bandwidth available in the system. This scenario presents the highest bandwidth contention studied in this paper. It would be possible to create worse scenarios for some applications, varying the writing or reading rates to main memory as well as the stride of the target page. Nevertheless, the analysis of such situations is out of the scope of this paper.

Fig. 3 presents the IPC and memory bandwidth consumption of the applications for the analyzed prefetch settings. As in the previous study, disabling the prefetching can still have a significant adverse impact on the performance of many applications such as `libquantum`, `zeusmp`, `cactusADM` or `leslie3d`. The performance gains in these applications are also related to a higher bandwidth utilization. However, unlike it was previously observed in absence of inter-application interference (Section 4.1), under high bandwidth contention, the prefetch urgency parameter affects the performance and bandwidth consumption more than the prefetch depth. The reason is that when multiple applications co-run and share the main memory bandwidth, increasing the prefetch depth to trigger more prefetches might not always improve performance since it increases bandwidth contention and, in addition, these prefetches compete for LLC storage capacity. In this scenario, the prefetch urgency becomes more important since the higher bandwidth contention translates into a longer memory latency of the prefetches, thus triggering them earlier improves their chances of being timely prefetched.

Taking into account performance and bandwidth consumption, prefetch friendly applications can be classified in two main groups:

- *Prefetch-setting sensitive* applications such as `gcc`, `mcf`, `zeusmp`, or `cactusADM`. The performance of these applications can significantly vary with the prefetch setting. For instance, `zeusmp` improves its IPC from 0.7 to 1 when the prefetch urgency is increased.
- *Prefetch-setting insensitive* applications such as `bwaves` or `leslie3d`. The performance of these applications slightly differs by varying the prefetch setting. However, the associated bandwidth consumption may greatly vary, as observed in `leslie3d`.

In summary, the discussed results show that a prefetch configuration policy should analyze, for each application, both performance and bandwidth metrics with the different prefetch settings. This analysis should guide the prefetching strategy. For instance, the policy should i) disable the prefetch engine for prefetch unfriendly applications, ii) choose the configuration with the lowest bandwidth consumption for prefetch-setting insensitive applications, and iii) find a trade-off

between performance gains and bandwidth consumption for prefetch-setting sensitive applications.

4.3 IPC and Bandwidth Consumption Sensitivity to Bandwidth Contention

So far we have studied the impact of prefetching on individual application performance in two extreme bandwidth scenarios: all the memory bandwidth is available for the studied application, and the application competes for bandwidth with the microbenchmark (i.e., three instances)[1], described below, at its maximum main memory access rate. This section studies the impact of intermediate levels of bandwidth contention on performance and bandwidth consumption of each application varying the prefetch setting. To this end, the microbenchmark is tuned to precisely consume a specified amount of memory bandwidth when it is executed in isolation. This way, when executed together with the studied application, the microbenchmark is used to induce the desired level of inter-application bandwidth contention.

4.3.1 Memory Bandwidth Microbenchmark

Algorithm 1 presents the pseucodode of the microbenchmark. It consists of a main loop (lines 2–9), where a burst of memory requests (lines 3–5) is issued. This burst is followed by a customizable number of NOP operations, which allow us to set the bandwidth consumption of the microbenchmark and study different levels of bandwidth contention. Memory requests access a memory array defined in line 1. The array size should, at least, double the size of the LLC of the processor to ensure that data is evicted from the cache before it is re-accessed, and the variable *stride* should also be configured, depending on the memory architecture, to make sure that two consecutive accesses to the array access different cache lines. In this way, we avoid LLC hits, maximizing the memory bandwidth the microbenchmark consumes.

Algorithm 1. Memory Bandwidth Microbenchmark

```
1: int A[ARRAY_SIZE]
2:   while true do
3:   for (i = 0; i < ARRAY_SIZE; i=i+stride) do
4:     A[i]++
5:   end for
6:   for (i = 0; i < #nops; i++) do
7:     asm("nop")
8:   end for
9: end while
```

We found empirically that the main memory bandwidth saturates when at least three independent instances of the microbenchmark are executed together each one on a different core. Further details on this empirical experiment can be found in Section 1.1 of the Appendix.

This scenario, as mentioned above, is referred to as the highest bandwidth contention scenario. With this setup we have empirically estimated that the maximum bandwidth an application can consume ranges from 180 to 190 accesses per μs. For analysis purposes we consider 190 accesses/μs as the maximum bandwidth utilization.

1. The microbenchmark is always launched in three instances.

To model mid range scenarios providing different amounts of available bandwidth, we have estimated the number of #nops the microbenchmark instances must execute to consume from 90 percent of its maximum bandwidth consumption (i.e., 190 accesses/μs) down to 10 percent in 10 percent steps. From now on we use BCI_XX (Bandwidth Consumed in Isolation) to refer to the configuration in which the microbenchmark instances consume XX% of the overall bandwidth when running in isolation. Notice that this percentage will be effectively reduced when the available bandwidth saturates. In this case, the effective bandwidth consumed by the microbenchmarks will be the difference between the bandwidth consumed by the application and the maximum bandwidth consumption.

4.3.2 Performance and Bandwidth Sensitivity

This section analyzes the behavior of the studied applications in eleven memory contention scenarios. Six prefetch settings have been studied in this work but, in addition to OFF and default, results are only shown for U1D2 and U7D2 prefetch settings because they present similar values to U1D7 and U7D7, respectively; that is, those having the same urgency but the highest depth are presented. The configurations with lowest depth have been chosen because in some cases the consumed bandwidth is lower than when the depth parameter is equal to 7.

Fig. 4 depicts the IPC (y-axis) and bandwidth consumption (x-axis) of the studied applications in four plots, one for each prefetch configuration. Each application is represented with a line with 11 points (or marks) which indicate the level of main memory bandwidth contention introduced by the microbenchmark. The rightmost point represents the values achieved by the application running alone under no contention (i.e., BCI_00), the leftmost represents the performance and bandwidth achieved by the application under maximum contention (i.e., running with BCI_100), and the 9 intermediate points (from right to left) represent the values achieved by the application in mid range bandwidth contention scenarios; that is, from right to left, running with BCI_90 to BCI_10. For instance, the second point (from right to left) of `astar` in Fig. 4a indicates that when this application is under the BCI_10 scenario it achieves an IPC by 0.24 and performs by 22.5 accesses/μs. Section 1.2 of the Appendix complements these figures and shows the bandwidth consumption of each application and the co-running microbenchmarks on each point for a subset of benchmarks with the default prefetch configuration.

In each plot, regardless of the prefetch setting, three main application behaviors can be observed. Below these behaviors are studied for the default prefetch setting depicted in Fig. 4c. First, applications located at the leftmost side (e.g., close to the Y axis) in a single point are insensitive to the bandwidth contention. The reason is that their bandwidth consumption is really low and thus their performance is not affected by main memory bandwidth contention. Second, we can distinguish applications extremely sensitive to the bandwidth contention. In this plot, these applications present from 5 to 35 accesses/μs. Their performance experience a sharp rise with a small increase in the bandwidth consumption. This trend slightly decreases as we move on to

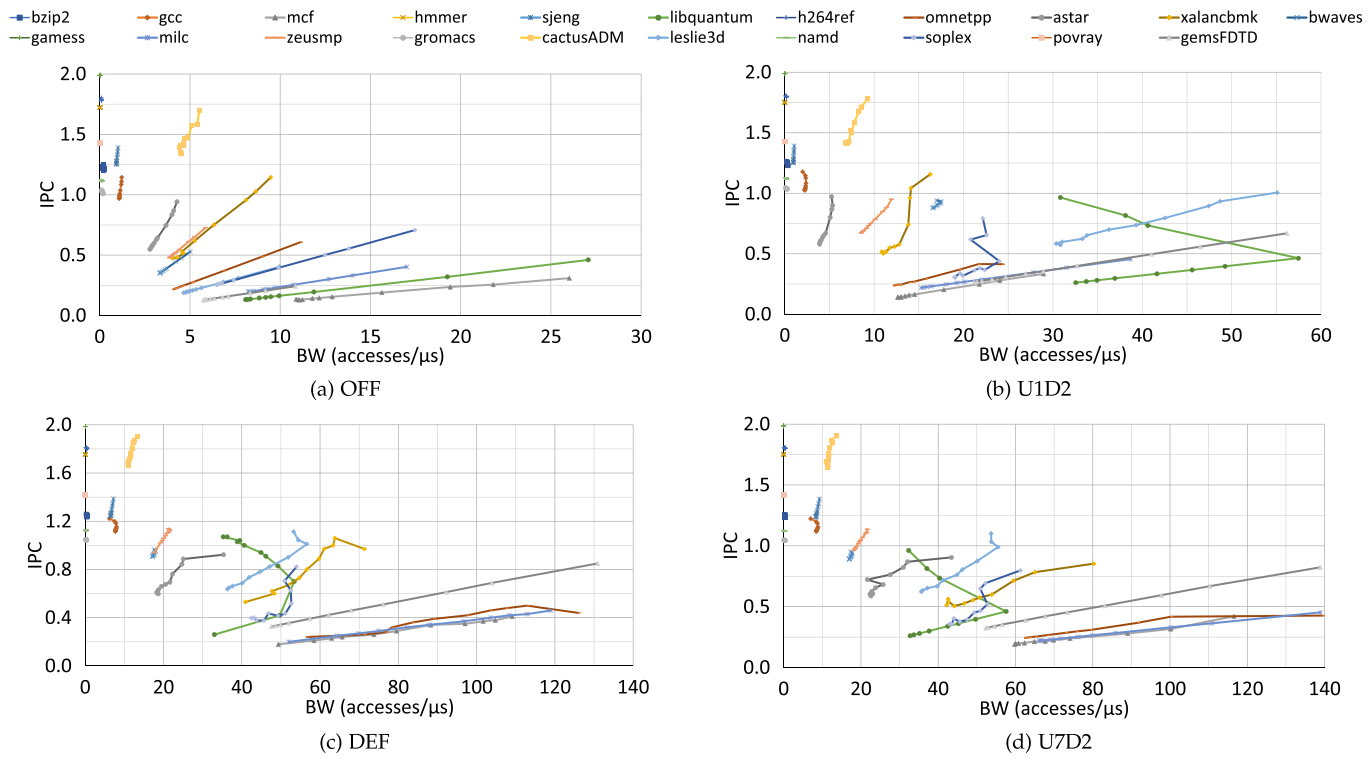


Fig. 4. Performance and bandwidth consumed by the applications for OFF, U1D2, DEF, and U7D2 prefetch settings varying the bandwidth consumption of the microbenchmark.

the right. And third, as we move on by about 50 accesses/μs, increasing the bandwidth consumption still allows improving the performance but to a minor extent. That is, bandwidth consumption must significantly increase to turn into performance gains. For instance, increasing bandwidth consumption from 60 to 130 accesses/μs in `omnetpp` just improves IPC in 0.2. Notice, however, that such a large amount of bandwidth represents almost one third of the total system bandwidth.

Another interesting observation is that the curve of some applications does not always experience a positive slope in terms of bandwidth as bandwidth contention increases, but it first grows and then reduces as the bandwidth consumed by the microbenchmark increases. For instance, *libquantum* shows an IPC by 1.1 in isolation with the default prefetch setting. Then, as the microbenchmarks start consuming bandwidth, *libquantum*'s bandwidth consumption, contrary to expected, increases. The increase in the bandwidth utilization is a result of the cache pollution [27] introduced by the microbenchmarks. This trend is kept until *libquantum* runs in the BCI_30 scenario. From that point on, the bandwidth contention induced by the microbenchmarks causes *libquantum*'s bandwidth utilization to decrease. In summary, this behaviour is mainly caused by the interference between the applications and the microbenchmarks both in the main memory bandwidth and LLC space. Both of them grow with the prefetch aggressiveness. In fact, the plot corresponding to the OFF setting is the only one where all the application's curves present a positive slope.

Comparing the four plots among them, it can be observed that, as expected, the bandwidth consumption of a given application increases as the applied prefetch settings are more aggressive (bottom plots). However, an interesting observation is the magnitude of these values, which widely differ among them depending on the prefetch setting. For instance, the maximum bandwidth an application consumes is by 30, 60, 130 and 140 accesses/μs with the prefetcher disabled (OFF), U1D2, DEF, and U7D2 settings, respectively. This huge difference allows some interesting cross comparisons, for instance, the bandwidth consumption of some applications (e.g., `omnetpp`) under maximum contention with the default prefetch setting is much higher than running the application alone with the prefetcher disabled.

5 BAPC Prefetching Approach

This section presents the prefetch approach proposed in this paper, Bandwidth-Aware Prefetch Configuration (BAPC), which pursues to dynamically adapt the prefetch configuration for each application taking into account the tradeoff between the performance and bandwidth consumption of each prefetch configuration. The results of the characterization study claim that a per-application custom prefetch configuration is needed to regulate the use of prefetching among the co-running applications. This per-application configuration should take into account both performance and bandwidth consumption, since in some cases the performance of a given application can be improved by a given prefetch setting but at expense of a high bandwidth increase. Otherwise, bandwidth consumption can significantly grow, causing important bandwidth contention and leading the system to significant performance losses.

With this aim, in order to quantify the trade-off between performance and bandwidth to select the proper prefetch setting, we define the *P2B ratio* ratio calculated with Equation (1). This metric considers the improvement rate in IPC of a given

prefetch setting over the prefetch disabled, and divides this value by the bandwidth wasting rate introduced by that prefetch setting over no prefetch.

$$P2B\ ratio_{config} = \frac{IPC_{config}/IPC_{OFF}}{BW_{config}/BW_{OFF}}. \quad (1)$$

For instance, a speedup by 1.1 (10 percent performance increase) and bandwidth consumption rate by 2.2 (120 percent bandwidth increase) gives a P2B ratio equal to 0.5. In general, the IPC increase is usually much smaller than the bandwidth increase, as experimental results will show. Once the P2B ratio is quantified, it is compared to a threshold to discern if a given prefetch configuration is eligible for that application or not. We experimentally found that a good trade-off is achieved when $P2B\ ratio \geq 0.25$, which means that a 10 percent performance increase is *worth enough* if it comes with a bandwidth consumption increase lower than 340 percent. Notice that applications consuming a significant amount of bandwidth present and almost flat curve in Fig. 4.

5.1 Static BAPC

With the aim of ascertain the potential of our approach, we have first devised a static version of the mechanism to be applied in high bandwidth contention scenarios. The static approach selects a *customized* prefetch setting for each application of the workload that is used during the whole execution of the application. For each application, the selected setting is the one that better fits the characteristics of the application trading off performance and bandwidth. For this purpose the approach works as follows.

First, for each considered prefetch configuration, the IPC and bandwidth consumption of each application are obtained when running in the highest contention scenario (BCI_100). Second, a list of eligible prefetch settings is obtained. To be included in the list, a configuration must fulfill two conditions: i) its corresponding P2B ratio value must be higher or equal than the P2B threshold (i.e., prefech configurations with P2B ratio below the threshold are discarded), and ii) its IPC must be better than the one obtained with no prefetching. If the list is empty, then there is no prefetch configuration that matches these conditions and, consequently, the prefetching is turned off for the application. On the contrary, if several configurations are in the list, they are arranged in descending IPC order, and the prefetch configuration at the head of the ordered list is chosen.

Table 1 presents the selected configuration obtained with this method for each SPEC CPU2006 and SPEC CPU2017 application. For example, the configuration OFF is chosen for `sjeng` because, as observed in Fig. 3, neither the performance improves due to prefetching (second condition) nor the P2B ratio is higher or equal than 0.25 (first condition). Another example is `zeusmp`, whose selected prefetch configuration is U7D2 because its performance gain is reasonable for the bandwidth consumption increase that the application experiments with this configuration.

By applying a custom prefetch configuration to each application, we ensure a profitable utilization of the available system bandwidth. In contrast, if the default configuration is used in every application, the system will present a much higher contention, which leads to a negative impact in performance. Experimental results obtained when statically assigning the selected prefetch settings to applications are presented in Section 6.

TABLE 1
Selected Configurations for Each Application in SPEC CPU2006 and SPEC CPU2017

Benchmark 2006	Conf.	Benchmark 2017	Conf.
bzip2	U7D2	perlbench_r	U1D2
gcc	U1D2	gcc_r	U7D2
mcf	U7D2	mcf_r	U1D2
hmmer	U1D2	parest_r	U7D2
sjeng	OFF	deepsjeng_r	U1D2
libquantum	DEF	lbm_r	U1D2
h264ref	OFF	x264_r	U7D2
omnetpp	U1D2	omnetpp_r	U1D2
astar	U1D2	cam4_r	U7D2
xalancbmk	OFF	imagick_r	DEF
bwaves	U7D2	bwaves_r	U7D2
gamess	OFF	wrf_r	U7D2
zeusmp	U7D2	leela_r	U1D2
gromacs	DEF	nab_r	U7D2
cactusADM	U7D2	cactuBSSN_r	U7D2
leslie3d	U7D2	roms_r	U7D2
namd	OFF	namd_r	U7D2
soplex	U1D2	xz_r	U1D2
provray	OFF	povray_r	OFF
gemsFDTD	DEF	exchange2_r	U7D2

5.2 Dynamic BAPC

Static BAPC relies on profiled execution information of applications obtained offline. This information is used to select the prefetch setting for each application that is applied during the whole execution. In contrast, dynamic BAPC acts without previous knowledge of applications' behavior, and dynamically selects the target prefetch setting at run-time, so different settings can be chosen for a given application. This way allows BAPC to adapt to the different execution phases each application experiences.

Algorithm 2 presents the pseudo-code of dynamic BAPC. This algorithm can be divided into three main phases: sampling, configuration and execution. These phases are executed sequentially in a loop until the workload execution completes.

In the sampling phase, metrics related to performance and bandwidth consumption are estimated. To do so, the prefetcher is globally disabled to gather IPC_{OFF} and BW_{OFF} for all the applications. Then, the prefetch settings are applied, one by one, to each application while the remaining applications run with the prefetcher turned off. In this way, IPC_{config}, BW_{config}, and $P2B\ ratio_{config}$ can be independently estimated for each application and prefetch configuration.

To reduce the overhead of sampling, the sampling quantum duration is set to 50ms, and the checked prefetch settings are limited to DEF, U1D2 and U7D2, because prefetch depth in general, as discussed in Section 4 does not affect the performance and bandwidth consumption under important bandwidth contention.

The metrics estimated in the sampling phase are employed in the configuration phase to obtain the best prefetch setting for each application. In this phase, a list of candidate settings

is created for each application. A setting is eligible if the measured IPC of the application with that setting is significantly higher than the IPC with prefetcher disabled. This is regulated with the IPC_{factor} parameter, which avoids selecting wrong settings due to estimation errors in the sampling phase. This factor is not necessary in static BAPC since it relies in profiling information obtained offline.

Algorithm 2. Dynamic BAPC

```
1: while there are running applications do
2:   ——— SAMPLING PHASE ———
3:   for all app do Prefetch(app) = OFF
4:   Execute 1 quantum and update IPC_{OFF,app} and BW_{OFF,app} for all apps
5:   for all app do
6:     for all config in {DEF, U1D2, U7D2} do
7:       Prefetch(app) = config
8:       Execute 1 quantum and update IPC_{config,app}, BW_{config,app}, and P2B ratio_{config,app}
9:       Prefetch(app) = OFF
10:    end for
11:  end for
12:  ——— CONFIGURATION PHASE ———
13:  for all app do
14:    config_list_app = {}
15:    for all config in {DEF, U1D2, U7D2} do
16:      if IPC_{config,app} ≥ (IPC_factor × IPC_{OFF,app}) then
17:        Add config to config_list_app
18:      end if
19:    end for
20:    Sort config_list_app in descending IPC order
21:    for all config in config_list_app do
22:      if P2B ratio_{config,app} ≥ P2B ratio_threshold then
23:        Prefetch(app) = config
24:      end if
25:    end for
26:  end for
27:  ——— EXECUTION PHASE ———
28:  Set quantum duration (e.g., 400ms)
29:  for 1 to N quantums do
30:    Execute 1 quantum
31:    Update BW_System
32:    if BW_System ≥ BW_threshold then
33:      min_app = app with lowest P2B ratio_config value
34:      Prefetch(min_app) = OFF
35:    end if
36:  end for
37: end while
```

Once the list of prefetch settings is made for a given application, the prefetch settings in the list are arranged in descending IPC order. Then, the list is looked up to find the configuration with the maximum IPC among those whose P2B ratio value is greater or equal than the threshold. In case of there is not any configuration that fulfills the condition, the prefetch engine is disabled.

The prefetch settings chosen in the configuration phase are applied to the execution phase, which keeps the selected settings for a given number of quanta while monitoring the total accumulated main memory bandwidth consumption. If the bandwidth consumption is over the $BW_{threshold}$, BAPC disables prefetching for the application whose active prefetch setting presents the lowest P2B ratio value. The rationale behind this design choice is that this setting is the least efficient in terms of the performance/bandwidth ratio.

Note that the time taken in the execution phase must be limited in order to capture dynamic phase changes in the application behaviors that affect both performance and bandwidth. In this regard, previous works like [28] have shown that, on average, execution phases last a few seconds without significant changes in the IPC of the application.

6 Performance Evaluation

6.1 Methodology

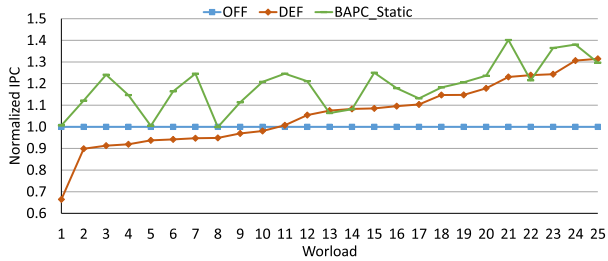
To study the performance and bandwidth utilization under BAPC, we have implemented both the static and dynamic versions in a user-level scheduler. In the dynamic version, this scheduler orchestrates the sampling, configuration and execution phases, collects performance counters to obtain the IPC and main memory bandwidth utilization of the applications, and updates the prefetch configuration for each application based on Algorithm 2. To analyze how BAPC performs compared to other prefetch configurations, we have implemented two static policies in the user-level scheduler: one keeps prefetching disabled, the other keeps it to the default prefetch setting. Note that the later is the default case in a Linux system. In addition, we have also implemented the *Intelligent Bandwidth Shifting* (IBS) mechanism proposed by Jiménez *et al.* [14].

To evaluate the proposal we have designed 3 sets of 25 workload mixes each composed of 6, 8, and 10 applications randomly selected from SPEC CPU2006 and SPEC CPU2017 benchmark suites. In order to give the same weight to all the applications in a mix, we have measured the number of instructions that each application executes in isolation during 120 seconds with the prefetcher turned off. This value is used as the *target* number of instructions of each application. During the execution of a workload mix, when an application completes its target number of instructions, its IPC is obtained and the application relaunched again to keep a constant load during the whole experiment, which finishes when all the applications of the mix complete their target number of instructions. We measure performance by means of the weighted speedup metric [29], which is calculated as the sum of the performance of each application of the workload normalized to its performance when running in isolation with prefetching disabled.

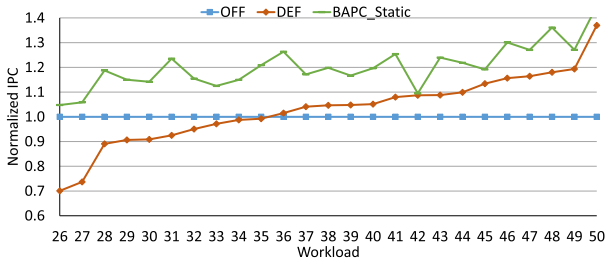
The thresholds and parameters used by the dynamic BAPC approach have been obtained through a wide set of experiments. The presented experimental results have been obtained with $P2B\ ratio_{threshold} = 0.3$, $BW_{threshold} = 185$, $IPC_{factor} = 1.1$, while the execution phase length in quanta (N) and the quantum length are set to 50 and 400 ms, respectively. Sensitivity studies of all these parameters have been carried out. For illustrative purposes, Section 2 of the Appendix presents two sensitivity studies varying the P2B threshold and sampling quantum length.

6.2 Static BAPC

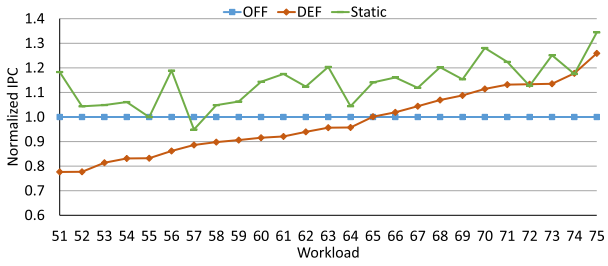
Fig. 5 presents the performance achieved by the default (DEF) and proposed static (BAPC_static) prefetch configuration algorithms normalized to the performance of prefetching



(a) 6-application workloads

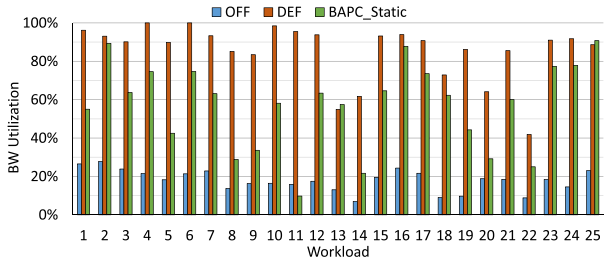


(b) 8-application workloads

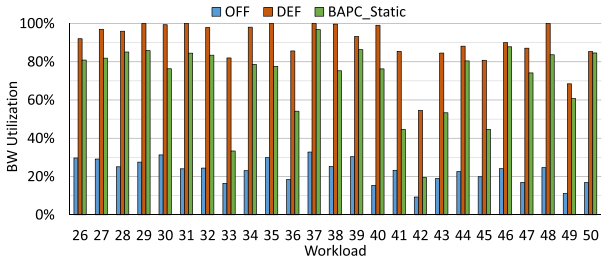


(c) 10-application workloads

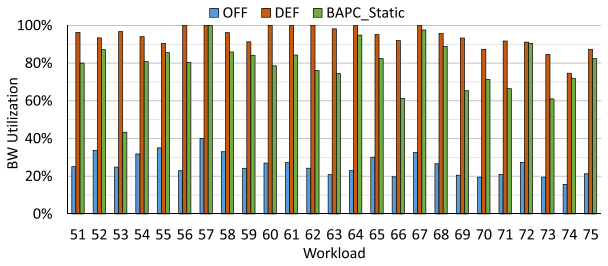
Fig. 5. Performance of DEF and BAPC_Static prefetch configurations normalized to prefetching disabled.



(a) 6-application workloads



(b) 8-application workloads



(c) 10-application workloads

Fig. 6. Bandwidth consumed by BAPC_Static, DEF, and OFF with respect to the maximum available in the system.

disabled (OFF) across 6-, 8-, and 10-application workloads. The 25 workloads of each scenario are presented in increasing speedup order of DEF over no prefetching. As observed, the default prefetch setting does not always achieve higher performance than no prefetching. Moreover, performance losses in some mixes can be as high as 30 percent. These cases appear due to prefetch inaccuracy and the high memory contention that prefetching causes. Because bandwidth contention increases with the number of applications, the number of workloads where DEF is outperformed by OFF tends to grow with the mix size. There are 10, 9, and 14 workloads in the 6-, 8-, and 10-application scenarios respectively, where the OFF setting outperforms DEF. However, despite these cases, DEF achieves better performance than OFF in most workloads, reaching speedups close to or above 40 percent regardless of the number of applications in the mix.

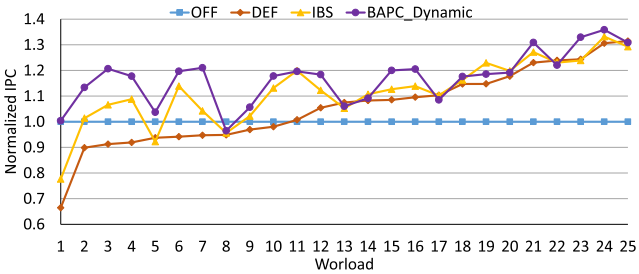
Fig. 6 presents the bandwidth utilization (190 accesses/μs is assumed as the 100 percent) consumed by each workload mix in the studied approaches. The reported bandwidth consumption includes both on demand and prefetch accesses. As observed, the bandwidth utilization greatly varies across the studied approaches. While it approximately ranges from 10 percent to 40 percent when prefetching is disabled, it grows over 90 percent in many workloads with the default configuration. This means that DEF saturates the system bandwidth with prefetches, which translates into system performance degradation when prefetch accuracy and timeliness are low and do not compensate the induced bandwidth contention.

In contrast, the BAPC_static approach reaches an intermediate bandwidth consumption, usually ranging from 40 to 90 percent of the system bandwidth. This is because BAPC_static disables prefetching for prefetch unfriendly applications and selects the best prefetch setting, considering performance and bandwidth, for the remaining, prefetch friendly, applications. This way allows BAPC_static to reduce the bandwidth consumption of those applications that do not benefit from the additional prefetch requests, thus reducing the bandwidth contention and increasing the bandwidth availability for those applications that really need it. These key actions turn into performance gains.

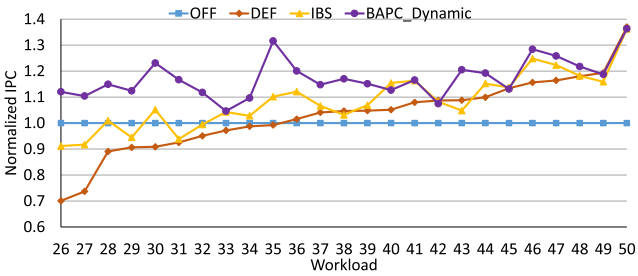
To better figure out how bandwidth contention impacts on performance, notice that BAPC_static greatly outperforms the default prefetch configuration when the bandwidth utilization exceeds 90 percent (e.g., workloads 7, 28, and 51).

6.3 Dynamic BAPC

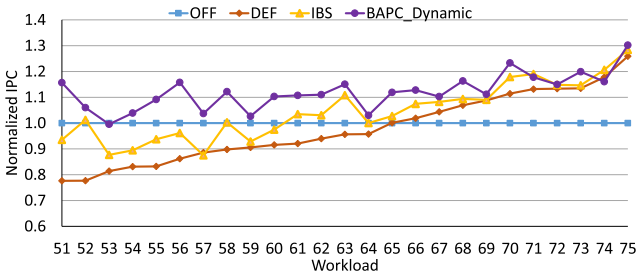
This section evaluates the dynamic version of the proposed mechanism, which, as mentioned above, selects the prefetch settings at run-time for each application depending on the execution phase and the system bandwidth utilization. We compare its performance and bandwidth consumption with no prefetching, the default prefetch configuration, and the IBS mechanism [14]. As our approach, IBS also seeks to save bandwidth and only enables prefetching for those applications that IBS estimates that can benefit from it. However, it



(a) 6 applications workloads

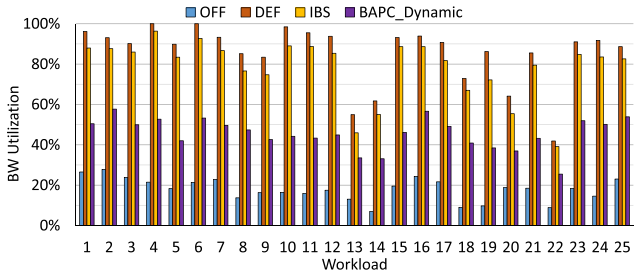


(b) 8 applications workloads

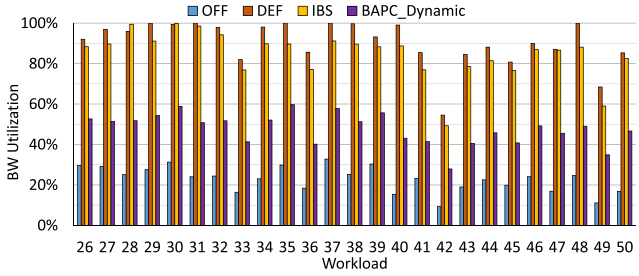


(c) 10 applications workloads

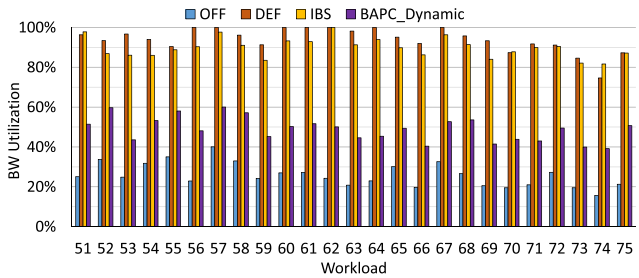
Fig. 7. Performance of DEF, IBS, and BAPC_Dynamic configurations normalized to prefetching disabled.



(a) 6-application workloads



(b) 8-application workloads



(c) 10-application workloads

Fig. 8. Bandwidth consumption of the OFF, DEF, and BAPC_Dynamic prefetch configurations with respect to the maximum bandwidth available in the system.

only considers two prefetch configurations: disabled or enabled at its maximum aggressiveness. Consequently, it does not take advantage of less aggressive prefetch configurations which, as shown in Section 4.2, can achieve most of the performance benefits of the most aggressive configuration while significantly consuming less bandwidth thus reducing the bandwidth contention. This section evaluates the studied schemes under multiprogram workloads. Sections 3 and 4 of the Appendix discuss the accuracy of the performance and bandwidth estimated in the sampling phase, and the per-application performance achieved when the workloads run under dynamic BAPC compared to the default prefetch configuration, respectively.

Fig. 7 shows the normalized IPC of the dynamic approach (BAPC_dynamic), the default configuration, and IBS with respect to no prefetching. Comparing Fig. 7 to Fig. 5 we can observe that the dynamic approach achieves similar results to the static version. BAPC_dynamic achieves a noticeable IPC improvement over OFF, DEF and IBS, with the only exception of just a few workloads where they present barely the same behavior. The performance improvement is on average by 12, 15, and 16 percent over DEF for 6-, 8- and 10-application workload mixes, respectively. This means that BAPC_dynamic is able to detect which is the proper configuration in the distinct execution intervals, making an excellent use of the system bandwidth. IBS outperforms the default prefetch configuration by turning off the prefetching for the applications that do not benefit from it, which reduces bandwidth contention. However, BAPC_dynamic has a finer control of the prefetch aggressiveness and thus it achieves a better bandwidth utilization, which results in significantly higher performance for most of the workloads.

Fig. 8 shows the bandwidth utilization results for the studied prefetch configurations and mechanisms. We can drawn the same conclusions as in the static BAPC approach. In this case, the dynamic approach achieves, on average, 39, 42, and 45 percent bandwidth savings compared to the default configuration for 6-, 8-, and 10-applications.

Finally, notice that the bandwidth utilization of the dynamic approach is significantly lower than that of the static approach. Thus, there is still room to further increase the performance by properly relaxing the thresholds of the algorithm. However, we choose relatively strict thresholds since they provide both high performance and low bandwidth, which makes the approach to stand out in scalability.

6.4 Case Study

In order to better understand the reason why our proposal is able to improve performance and bandwidth, this section analyzes the behavior of each individual application of a given workload. For illustrative purposes we study mix 58. Fig. 9a, 9b, and 9c depict the IPC, bandwidth utilization, and

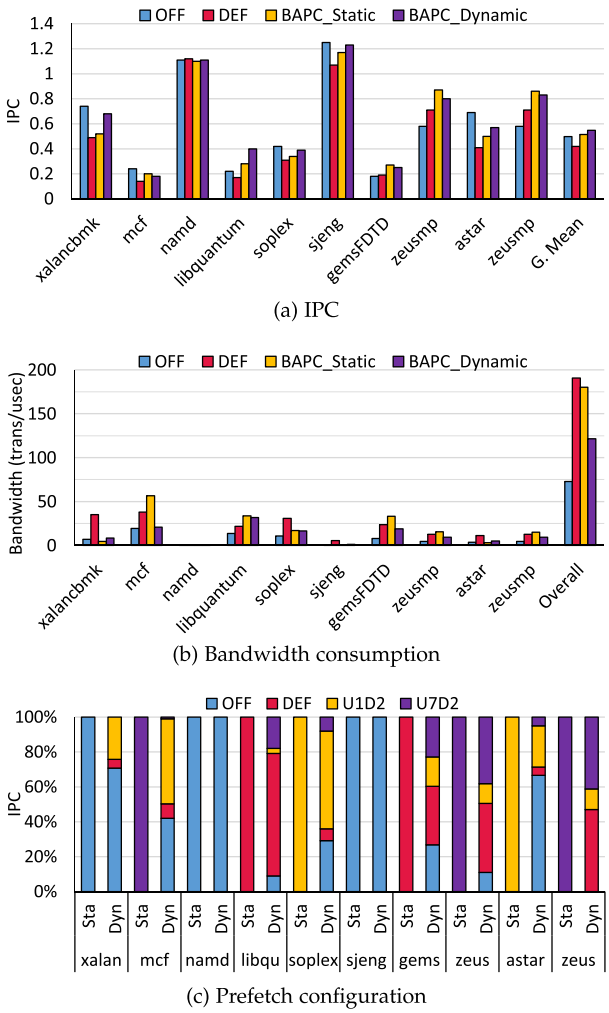


Fig. 9. IPC, bandwidth consumption and prefetch configuration of the applications in workload 58.

prefetch configuration for each application of the workload and the studied prefetch configurations and BAPC mechanisms.

First of all, looking at Fig. 9a, we can observe how the configuration with prefetching disabled outperforms the default prefetch configuration in most benchmarks, which indicates that the default prefetch configuration suffers from bandwidth contention. Fig. 9a confirms this fact since the applications reach an overall bandwidth utilization of 190 trans/μs for the default prefetch configuration, which eliminate the potential benefits that prefetching usually has.

The static BAPC mechanism selects the best prefetch configuration for each application. For some application this turns into higher prefetch aggressiveness and bandwidth utilization (e. g., *mcf*) while for others prefetching is disabled and bandwidth saved (e.g., *xalancbmk*). The static BAPC mechanism improves the performance of the configuration with prefetching disabled but it still suffers from significant bandwidth contention (overall bandwidth utilization of 180 trans/μs). Thus, the performance benefits it has compared to the prefetching disabled are moderate.

The dynamic BAPC mechanism is able to (i) adapt the prefetch configuration to the dynamic phase behavior of the applications and (ii) reduce the prefetch aggressiveness of some applications as soon as it detects that the bandwidth utilization is excessive and could significantly rise bandwidth contention. As Fig. 9c shows, these features make the prefetch configuration of most applications to vary among different prefetch settings. This way turns into important bandwidth savings for several applications that reduce their prefetch aggressiveness for a long time in their execution (e.g., *mcf*, *GemsFDTD*) with minor impact on their performance. Other applications such as *xalancbmk* run with more aggressive prefetch configurations (compared to the static BAPC mechanism) for approximately 30 percent of the execution time, which significantly improves the performance with a small increase in the bandwidth utilization. In summary, adapting the prefetch configuration to the phase behavior of applications while limiting the bandwidth utilization to avoid bandwidth contention allows the dynamic BAPC mechanism to improve the performance of most applications compared to the static BAPC mechanism.

7 Conclusion

Prefetch engines in current high-performance processors include a wide range of prefetch settings, which makes difficult to dynamically configure them at run-time. This work makes two main contributions. First, we characterize the behavior of a wide set of applications varying the bandwidth contention for the most significant prefetch configurations in the IBM POWER8. We found that the best prefetch setting for each individual application, considering both performance and bandwidth consumption, does not depends only on the application itself but also on the co-running applications competing for memory bandwidth. Second, we propose Bandwidth-Aware Prefetch Configuration (BAPC), an scalable and adaptive prefetch algorithm. The dynamic version of BAPC chooses the prefetch setting that better fits the behavior of each application in order to achieve a good trade-off between performance and bandwidth. Experimental results show that BAPC excels in high-contention bandwidth scenarios. Compared to default prefetching, BAPC outperforms by 12, 15, and 16 percent for workload mixes consisting of 6, 8, and 10 applications, while dramatically reducing the bandwidth consumption.

Although the focus of this paper has been on the IBM POWER8, the fundamentals of the characterization study and the devised approach could be generally adapted to other processors implementing programmable prefetchers.

Acknowledgments

This work was supported in part by Ministerio de Ciencia, Innovación y Universidades and the European ERDF under Grant RTI2018-098156-B-C51, Generalitat Valenciana under Grant AICO/2019/317, and Universitat Politènica de València (PAID-06-18) under Grant SP20180140.

References

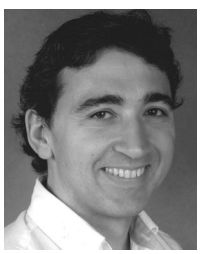
[1] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proc. ACM/IEEE Conf. Supercomputing*, 1991, pp. 176–186.

[2] J. W. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," *ACM SIGMICRO Newsletter*, vol. 23, no. 1–2, pp. 102–110, 1992.

[3] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *Proc. 24th Int. Symp. Comput. Archit.*, 1997, pp. 252–263.
[4] G. B. Kandiraju and A. Sivasubramaniam, "Going the distance for TLB prefetching: An application-driven study," in *Proc. 29th Annu. Int. Symp. Comput. Archit.*, 2002, pp. 195–206.
[5] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *Proc. 10th Int. Symp. High Perform. Comput. Archit.*, 2004, pp. 96–96.
[6] S. H. Pugsley *et al.*, "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit.*, 2014, pp. 626–637.
[7] S. Byna, Y. Chen, and X.-H. Sun, "Taxonomy of data prefetching for multicore processors," in *J. Comput. Sci. Technol.*, vol. 24, no. 3, 2009, pp. 405–417.
[8] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," in *Proc. 42st Annu. IEEE/ACM Int. Symp. Microarchit.*, 2009, pp. 316–326.
[9] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-aware shared resource management for multi-core systems," in *Proc. 38th Annu. Int. Symp. Comput. Archit.*, 2011, Art. no. 141–152.
[10] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-aware DRAM controllers," in *Proc. 41st IEEE/ACM Int. Symp. Microarchit.*, 2008, pp. 200–209.
[11] M. Torrens, "Improving prefetching mechanisms for tiled CMP platforms," PhD. Thesis, Universitat Politècnica de Catalunya, Barcelona, Spain, 2016.
[12] C. Ortega *et al.*, "libPRISM: An intelligent adaptation of prefetch and smt levels," in *Proc. Int. Conf. Supercomputing*, 2017, pp. 28:1–28:10.
[13] V. Jiménez *et al.*, "Adaptive prefetching on POWER7: Improving performance and power consumption," *ACM Trans. Parallel Comput.*, vol. 1, no. 1, pp. 1–25, 2014.
[14] V. Jiménez, A. Buyuktosunoglu, P. Bose, F. P. O'Connell, F. J. Cazorla, and M. Valero, "Increasing multicore system efficiency through intelligent bandwidth shifting," in *Proc. 21st IEEE Int. Symp. High Perform. Comput. Archit.*, 2015, pp. 39–50.
[15] A. Valero, J. Sahuquillo, S. Petit, P. López, and J. Duato, "Combining recency of information with selective random and a victim cache in last-level caches," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 3, pp. 1–20, 2012.
[16] S.-W. Liao, T.-H. Hung, D. Nguyen, C. Chou, C. Tu, and H. Zhou, "Machine learning-based prefetch optimization for data center applications," in *Proc. Conf. High Perform. Comput. Netw. Storage Anal.*, 2009, pp. 1–10.
[17] M. Li *et al.*, "Pater: A hardware prefetching automatic tuner on IBM POWER8 processor," *Comput. Archit. Lett.*, vol. 15, no. 1, pp. 37–40, 2016.
[18] K. Nesbit, A. Dhodapkar, and J. Smith, "AC/DC: An adaptive data cache prefetcher," in *Proc. 13th Int. Conf. Parallel Archit. Compilation Techn.*, 2004, pp. 135–145.
[19] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Proc. 13st Int. Conf. High-Perform. Comput. Archit.*, 2007, pp. 63–74.
[20] W. Heirman, K. D. Bois, Y. Vandriessche, S. Eyerman, and I. Hur, "Near-side prefetch throttling: Adaptive prefetching for high-performance many-core processors," in *Proc. 27th Int. Conf. Parallel Archit. Compilation Techn.*, 2018, pp. 28:1–28:11.
[21] B. Panda and S. Balachandran, "Expert prefetch prediction: An expert predicting the usefulness of hardware prefetchers," *Comput. Archit. Lett.*, vol. 15, no. 1, pp. 13–16, 2016.
[22] X. Zhuang and H. S. Lee, "Reducing cache pollution via dynamic data prefetch filtering," *IEEE Trans. Comput.*, vol. 56, no. 1, pp. 18–31, Jan. 2007.
[23] X. Dang, X. Wang, D. Tong, Z. Xie, L. Li, and K. Wang, "An adaptive filtering mechanism for energy efficient data prefetching," in *Proc. 18th Asia South Pacific Des. Autom. Conf.*, 2013, pp. 332–337.
[24] V. Selfa, J. Sahuquillo, M. E. Gómez, and C. G. Requena, "Efficient selective multicore prefetching under limited memory bandwidth," *J. Parallel Distrib. Comput.*, vol. 120, pp. 32–43, 2018.
[25] B. Hall *et al.*, *Performance Optimization and Tuning Techniques for IBM Power Systems Processors Including IBM POWER8. IBM Redbooks*, 2017. [Online]. Available: https://play.google.com/books/reader?id=7ph0CgAAQBAJ&hl=es&lr=&printsec=frontcover&pg=GBS.PA25.w.0.0.0.1#v=onepage&q&f=false
[26] J. Feliu, S. Petit, J. Sahuquillo, and J. Duato, "Cache-hierarchy contention aware scheduling in CMPs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 3, pp. 581–590, 2014.
[27] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Proc. 13th Int. Symp. High Perform. Comput. Archit.*, 2007, pp. 63–74.
[28] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "Addressing fairness in SMT multicores with a progress-aware scheduler," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2015, pp. 187–196.
[29] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, May/Jun. 2008.

Carlos Navarro received the BS and MS degrees in computer engineering from the Universitat Politècnica de València (UPV), Spain, in 2018 and 2019, respectively. He is currently working toward the PhD degree with the Department of Computer Engineering (DISCA) of the same university. His PhD research focuses on prefetching strategies in commercial multicore processors.

Josué Feliu received the MSc and PhD degrees in computer engineering from the UPV, Spain, in 2012 and 2017, respectively. He is currently working as a postdoctoral researcher with the Department of Computer Engineering of the same university. His research interests include scheduling strategies and performance modeling for multicore and multi-threaded processors. He was awarded the "IEEE TCSC Outstanding PhD Dissertation Award" in 2017.



Salvador Petit (Member, IEEE) received the PhD degree in computer engineering for the UPV, Spain. Since 2009, he has been an associate professor with DISCA Department at UPV, where he has taught several courses on computer organization. He has authored more than 100 refereed conference and journal papers. His current research interests include multithreaded and multicore processors, memory hierarchy design, GPU architecture, and resource management.

Maria E. Gomez received the MS and PhD degrees in computer engineering from the UPV, Spain, in 1996 and 2000, respectively. She joined the DISCA Department at UPV in 1996 where she is currently a full professor. She has published more than 70 conference and journal papers. She has served on program committees for several major conferences. Her research interests are on processor architecture and interconnection networks.

Julio Sahuquillo (Member, IEEE) received the BS, MS, and PhD degrees in computer engineering from the UPV, Spain. He is currently a full professor with DISCA Department at the UPV. He has taught several courses on computer architecture. He has authored more than 150 refereed conference and journal papers. His research interests include processor microarchitecture, memory hierarchy design, GPU architecture, and system resource management.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.