Idler : I/O Workload Controlling for Better Responsiveness on Host-Aware Shingled Magnetic Recording Drives

Baoquan Zhang, Ming-Hong Yang, Xuchao Xie, and David H.C. Du, *Fellow, IEEE*

Abstract—Host-Aware/Drive-Managed Shingled Magnetic Recording (SMR) drives can accept non-sequential writes using a buffer called media cache. Data in the media cache will be migrated to its designated location by a cleaning process if the buffer is full (blocking cleaning) or the drive is idle (idle cleaning). However, blocking cleanings can severely extend the I/O response time. Therefore, it is crucial to fully understand the cleaning process and find ways of mitigating the caused performance degradation. In this article we further evaluate the cleaning process and propose a potential remedy scheme called *Idler* on Host-Aware SMR drives. Idler adaptively induces idle cleanings based on dynamic workload characteristics and media cache usages to reduce the severity of blocking cleanings. Our evaluations show that in the workloads with a small non-sequential write ratio (about 10 percent), Idler can reduce the tail response time and the workload finish time by 56–88 and 10–23 percent, respectively, compared with those without such control. With the help of an external write buffer on an SSD, the tail response time of SMR drives with Idler can be closer to that of conventional disk drives.

Index Terms—Storage systems, shingled magnetic recordings, tail response time, workload characterizations

1 Introduction

Shingled Magnetic Recording (SMR) is a technology to increase the areal density of spinning drives by overlapping data tracks [1]. In an SMR drive, overlapped data tracks are grouped into zones, and each zone has a write pointer to indicate the current write location. An update is considered a non-sequential write if it does not begin at the write pointer in a zone. A non-sequential write may destroy the existing content in its adjacent tracks if it writes directly in its location. To avoid impacting the overlapped data, Host-Managed SMR (HM-SMR) drives prohibit non-sequential writes. That means existing applications need to be modified if we intend to use HM-SMR drives or a log-structured file needs to be used to support applications.

Device-Managed SMR (DM-SMR) and Host-Aware SMR (HA-SMR) drives implement a Shingled Translation Layer (STL) to handle non-sequential updates. STL includes an on-device persistent cache called Media Cache and an Extent Mapping Table. A non-sequential write will be buffered in the media cache first, and later migrated to its designated location by a cleaning process. Data can be either stored at the media cache or its designated location. Any data access needs to check with the mapping table to identify its current location. The cleaning process can be triggered by either an idle duration (i.e., a duration without any requests) called idle cleaning, or by depleted STL resources (e.g., media cache free space or free entry slots in the mapping table) called blocking cleaning. HM-SMR drives can only be used in workloads without any non-sequential writes like data archives and backups or with the support of a log-structured file. DM-SMR and HA-SMR drives can be deployed in many more scenarios since they both can handle non-sequential writes with the help of STL [2].

In the modern storage infrastructure, the tail response time of a system becomes significant. Applications require a low and predictable tail I/O response time [3], [4], [5], [6]. The 99.9th percentile (P99.9) response time is used as a useful statistical metric to evaluate the tail response time [4], [5], [6], [7], [8], [9]. However, HA/DM-SMR drives suffer from a long tail response time caused by the blocking cleaning. Evaluations [2], [10] show that during a blocking cleaning, an SMR drive spends up to seconds to serve a single I/O request.

Although workloads has burstiness, an SMR drive cannot rely on idle cleaning only to clean the media cache. An idle cleaning requires a relatively long idle duration to trigger and will be interrupted by any newly arrived requests. Therefore, the data cleaning efficiency of idle cleaning is degraded and blocking cleaning is still triggered. Moreover, the beginning time and the lasting duration of a blocking cleaning are unpredictable. The unpredictable performance with a long tail response time limits the usage of SMR drives to sequential workloads. Therefore, in this paper, we further evaluate the cleaning process of HA-SMR drives and explore ways of mitigating the caused performance degradation to extend the application scenarios of SMR drives.

- *B. Zhang, M.-H. Yang, and D.H.C. Du are with the Department of Computer Science, University of Minnesota–Twin Cities, Minneapolis, MN 55455. E-mail: {zhan4281, yang5445}@umn.edu, du@cs.umn.edu.*
- *X. Xie is with the College of Computer, National University of Defense Technology, Changsha 410073, China. E-mail: xiexuchao@nudt.edu.cn.*

Manuscript received 5 Apr. 2019; revised 15 Nov. 2019; accepted 8 Jan. 2020. Date of publication 15 Jan. 2020; date of current version 8 May 2020.
(Corresponding author: Baoquan Zhang.)
Recommended for acceptance by H. Jiang.
Digital Object Identifier no. 10.1109/TC.2020.2966194

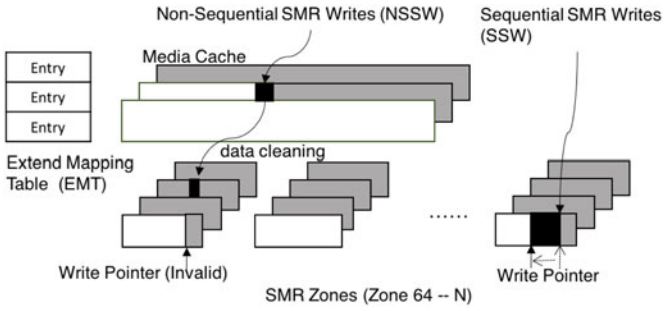


Fig. 1. Internal structure of HA-SMR drives.

We choose HA-SMR drives to evaluate since HA-SMR drives provide interfaces with Zoned Block Command (ZBC) [11] to check the current states of its internal structures. The internal information is helpful for our design and evaluations. By conducting evaluations, we find an idle cleaning triggered by a minimal idle duration has limited impacts on the drive performance since it stops cleaning once new requests are issued. While a blocking cleaning considerably degrades the drive performance and increases the I/O response time significantly.

Inspired by the different influences on the drive performance from idle cleanings and blocking cleanings, we propose a scheme called *Idler* to decrease the long tail response time of an HA-SMR drive caused by the blocking cleaning. Idler considers both the resource utilization of the media cache and the current workload characteristics to induce idle durations adaptively. With the induced idle duration, data in the media cache can be cleaned in a controllable way by idle cleanings without increasing the I/O response time significantly. It can also be implemented in the firmware of DM-SMR drives. We evaluate Idler with real-world workloads. Our evaluations show that Idler can avoid blocking cleanings in workloads with a non-sequential write ratio of 10 percent. The tail response time and the workload finish time are reduced by 56–88 and 10–23 percent respectively compared with those without such control. With the help of an external write buffer on an SSD, the tail response time of Idler can be further reduced. However, the performance of Idler is still worse than that of a conventional disk drive.

In the rest parts of the paper, Section 2 evaluates the data cleaning process. Section 3 discusses the performance improvements of artificially triggered idle cleanings. Then we present the proposed Idler in Section 4.2. We evaluate Idler in Section 5. Section 6 discusses the existing work. Section 7 offers conclusions and the future work.

2 Media Cache Cleaning and Evaluation

We evaluate the media cache cleaning using HA-SMR drives whose internal structure is shown in Fig. 1. It includes STL and SMR zones. An SMR zone is a collection of overlapped data tracks [11]. STL consists of a media cache and a mapping table. The buffered data in the media cache are log-structured since the media cache also consists of shingled tracks. Each shingled zone has a writing position called a write pointer. Any write beginning with the position pointed by the write pointer is considered as a *Sequential SMR Write* and can be issued directly to its designated location. The position of the write pointer will also be updated accordingly after a sequential write. A write with its targeted address not beginning at the write pointer is called a *Non-Sequential SMR Write*. A non-sequential write will be re-directed to media cache by STL. Its designated zone will be changed from a sequential state to a non-sequential state. The write pointer is no longer valid in a non-sequential zone. All the following write requests to a non-sequential zone will be buffered in the media cache [2].

STL utilizes a cleaning process to migrate the buffered data from the media cache to its designated locations in a Read-Modify-Write fashion [10]. A zone will be switched back to a sequential state after being cleaned. The cleaning process will be triggered once one of the resources (either free space in the media cache or free entry slots in the mapping table) is depleted [10].

In the rest of this section, we investigate how both the blocking cleaning and the idle cleaning affect the drive performance. We use *fio* to benchmark a raw HA-SMR drive (Seagate ST8000AS0022, 8TB Capacity) with direct I/O. In this drive, the size of the media cache is 28GB, and the maximum number of mapping entries is 182,000.

Fig. 2 shows the influences on the I/O response time by idle and blocking cleanings respectively. In this experiment, we use a single I/O thread. Since an ongoing cleaning increases the I/O response time and decreases the count of non-sequential zones, we use the change of the I/O response time (right y-axis) and the count of non-sequential zones (C_{NSZ}, left y-axis) to indicate when the cleaning starts. C_{NSZ} can be checked by Zoned Block Commands [11].

Fig. 2a shows the impact of idle cleanings on the I/O response time. We issue a set of non-sequential SMR writes (NSSWs) and then idle for a duration. Once we detect an idle cleaning starts, i.e., the number of non-sequential zone decreases at around 100s on x-axis, we issue more requests including non-sequential SMR writes, sequential SMR writes (SSWs), and reads. The I/O response time of the following requests is not increased. It indicates an idle cleaning has

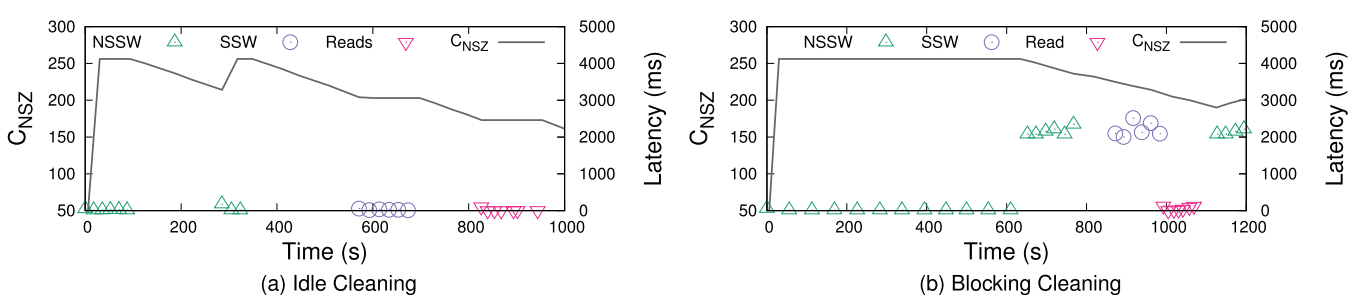


Fig. 2. Service latencies of an HA-SMR drive.

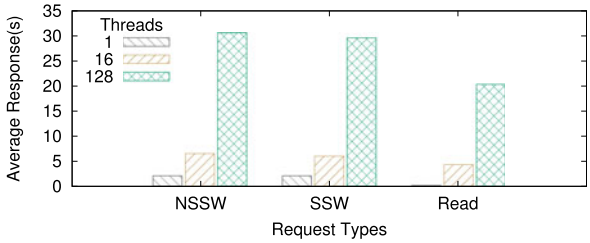


Fig. 3. The response time with IO dependencies.

limited impacts on the I/O response time since it will stop if any I/O requests are issued. For the blocking cleaning (Fig. 2b), we issue a set of NSSWs to trigger a blocking cleaning. At about 650s, a blocking cleaning is triggered since the I/O response time is increased. Then we issue more SSWs, Reads, and NSSWs. The response time of the following requests is increased due to blocking cleaning. The response time of all requests after a blocking cleaning triggered is longer than that in Fig. 2a.

I/O dependencies influence the I/O response time since an I/O request may wait in the I/O queue before submitted to the device. When there is only 1 I/O thread, i.e., most of the I/O requests are dependent, subsequent requests may need the results of the previous requests. The waiting time in the I/O queue will be short because a new I/O request is only generated after the last request finishes. However, the drive may have multiple I/O threads. I/O requests from various applications are independent and submitted to the I/O queue concurrently. An SMR drive has only one disk head to serve I/O requests one by one. The I/O wait time of concurrent I/O requests will be increased since a request has to wait until the drive finishes all of the previous requests. If a blocking cleaning is ongoing on an SMR drive, the I/O wait time of a later request will be increased significantly due to the enlarged I/O response time of the previous requests. Therefore, in the next experiment, we study the influences from a blocking cleaning on the I/O response time under different I/O dependencies.

Fig. 3 shows the average I/O response time under different I/O dependencies when a blocking cleaning is ongoing. We emulate the I/O dependencies by issuing requests with varying numbers of threads. We assume only the I/O requests in the same thread are dependent. In this experiment, we first trigger a blocking cleaning by issuing enough non-sequential writes. Then we start different numbers of threads to issue I/O requests including non-sequential SMR writes (NSSW), sequential SMR writes and reads. From the results, we find that when the number of threads increases, the I/O response time are enlarged more severely. The reason is that with more independent threads, more I/O requests submitted to I/O queue concurrently. With the increased response time of previous requests, the wait time of the later I/O requests are largely increased. As a result, the total I/O response time will be increased to an extremely large value with multiple independent threads due to the enlarged I/O wait time.

An idle cleaning utilizes the idle duration to clean the media cache. It is critical to find its minimum trigger time. In this experiment, we choose 8 zones and issue one non-sequential write to each with a given idle time between two subsequent requests. After 8 non-sequential writes, we check the number of non-sequential zones right away. We repeat the experiment three times for the same idle duration.

TABLE 1
Time Durations to Trigger an Idle Cleaning

IO size	Idle Time	Non-sequential Zone
4 KB	240 ms	8, 8, 8
	245 ms	8, 8, 8
	250 ms	7, 6, 6
	260 ms	0, 0, 0

Table 1 shows the number of non-sequential zones. It should be 8 if an idle cleaning is not triggered by this idle time. We only show the results of 4 KB IO size here even though we perform the experiments with 4 KB, 64 KB, 256 KB, 512 KB, and 1 MB I/O sizes since they have similar results. When the idle time reaches 250 ms, the number of non-sequential zones is less than 8 in all tests indicating an idle cleaning is triggered by a 250 ms idle time. Note that it takes some time to clean even one zone. The real minimum triggering time may be a bit shorter than 250 ms.

3 Improving Drive Performance With Artificially Triggered Idle Cleanings (AT-IC)

3.1 AT-IC

A blocking cleaning can largely increase the I/O response time. Although an idle cleaning has less impact on drive performance, it needs some minimal trigger time and stops with any newly arrived I/O requests. Idle cleanings may work well in a workload with burst I/O requests and relatively long idle time between bursts. However, in a workload with a short idle time smaller than minimal triggered time, idle cleanings might not even be triggered.

Inspired by the different performance influences from idle cleanings and blocking cleanings, we explore a potential concept of reducing the large tail response time caused by blocking cleanings using Artificially Trigger Idle Cleanings (AT-IC). AT-IC mitigates the performance degradation caused by a blocking cleaning with a way of invoking idle cleanings. AT-IC artificially creates idle durations when executing a workload. The I/O requests issued during an induced idle duration are stalled and served in the next execution duration. With these artificially created idle durations, idle cleanings are invoked to conduct cleaning and will not be interrupted for a fixed idle duration. With the invoked idle cleanings, a drive can clean the data in the media cache in a controllable manner which does not significantly enlarge the I/O response time.

The benefits of AT-IC is highly related to idle configurations and the characteristics of a workload. For a given workload, two configuration parameters, idle ratio and idle length, have the most significant impacts. The idle ratio means the proportion of total idle time in the duration of executing the whole workload. The idle length means the duration of each idle period. A higher idle ratio means we create longer idle time such that an idle cleaning can clean more buffered data. However, it also means that the total execution time will be longer. If the idle ratio is fixed, an

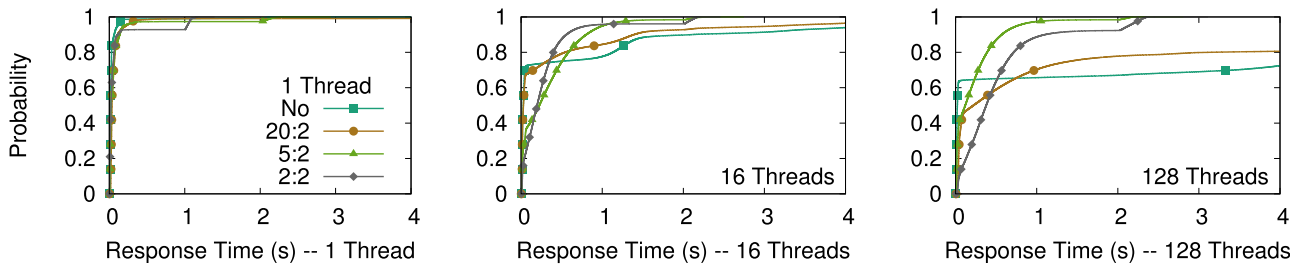


Fig. 4. CDF of the response time with different *idle ratios*.

idle cleaning can be triggered frequently with a smaller idle length or less frequently with a larger idle duration. Since I/O requests are delayed during an idle duration, with a shorter and frequent idle length, I/O requests will have a shorter response time. However, triggering idle cleanings more frequently with a smaller idle duration is inefficient since an idle cleaning needs some minimal idle duration, 250 ms based on our previous evaluations, to trigger.

Besides, characteristics of a workload including request arrival speeds, non-sequential write ratios, I/O request sizes, and the number of non-sequential zones may also influence the performance of AT-IC. The request arrival speed represents the intensity of a workload. It determines the number of requests delayed during an idle duration. A higher arrival speed makes AT-IC delays more I/O requests during a given idle duration leading to a larger I/O response time. The non-sequential write ratio represents the consuming speed of media cache resources. A higher non-sequential write ratio means we may fill up the media cache more frequently. The I/O request size may also influence the time duration to fill up the media cache. With a larger I/O request size, the media cache is filled up in a shorter duration, and blocking cleaning may be triggered by the depleted free space. While with a smaller I/O request size and a lower non-sequential write ratio, the media cache can serve for a longer time before a blocking cleaning is triggered. However, a high non-sequential write ratio and a smaller I/O request size may trigger a blocking cleaning due to the depleted mapping entries. At last, a lower number of non-sequential zones represents a shorter time to clean the media cache by either blocking cleanings or idle cleanings.

AT-IC can reduce the performance degradation caused by blocking cleanings. It makes the I/O response time more predictable and provides much better performance than that without such control. However, all of the above factors influence the effects of AT-IC. Therefore in the next subsection, we validate the performance improvement achieved by AT-IC and discuss the impacts from the idle configuration and workload characteristics.

3.2 AT-IC Validation

We implement AT-IC as a block I/O replay engine with libaio [12]. AT-IC issues block I/O requests respecting the time intervals between I/O requests. Beside, AT-IC can also periodically add a user-defined idle duration.

3.2.1 Idle Configurations

Since SMR drives are generally deployed with workloads of low non-sequential write ratios, we first evaluate AT-IC with workloads of low ratios of non-sequential writes. We vary the two parameters in the idle configuration: idle ratios and idle lengths. We synthesize traces consisting of I/O requests of 1 MB size. Here we use a large I/O size since a larger request size fills up the media cache sooner. And it will negatively influence the performance of AT-IC. However, it has less impact on the total cleaning time since it is the distance between the update offset and the write pointer position that dominates the cleaning time [2]. The request intervals are evenly distributed in the range from 10 ms to 100 ms. 10 percent of the requests are non-sequential writes with a total of 35 GB non-sequential writing size that spread over 1,024 zones. Another 10 percent of requests are sequential writes, and the remaining 80 percent of requests are reads. To study the conditions of different I/O dependencies, we partition the same trace into different numbers of subtraces (1, 16, 128). The I/O requests will be distributed evenly among subtraces based on their timestamps. Finally, we replay every subtrace with a separate thread. Only the requests within the same thread are dependent.

Idle Ratio. In this experiment, we intend to discover the influence of various idle ratios on the performance of AT-IC. AT-IC replays a workload with different idle ratios while keeping the same idle length. Here we set the idle length to 2s since it is close to the largest service latency when a blocking cleaning is ongoing on a drive. Fig. 4 shows the Cumulative Distribution Function (CDF) probability of the I/O response time with different idle ratios. The line marked with "No" means we replay the trace with an idle ratio of 0 percent. The others marked with the $Execution : Idle$ like $< 20 : 2 >$ means the drive serves requests for 20s and then idle for 2s and the idle ratio is about 2/22=9.09 percent. We measure I/O response time with different idle ratios under different numbers of I/O threads (1, 16, 128). We also display the tail (P99.9) response time and the workload finish time in Fig. 5.

With only 1 thread (Fig. 4), all requests are dependent. The distribution of the response time of AT-IC is similar with that of "No". The reason is that the idle length, 2s, is close to the largest response time when a blocking cleaning is ongoing. The current idle length blocks the subsequent request for a time duration same with blocking cleanings. Better results can be achieved with a shorter idle length which will be discussed later. However, the P99.9 response time with 1 thread in Fig. 5a shows that with a too-small idle ratio, .e.g $20 : 2$, AT-IC even increases the tail response time since it can not avoid blocking cleanings and introduces extra delay by the idle duration.

When the number of I/O threads increases (Column 2 and 3 in Fig. 4), i.e., more independent requests are issued concurrently, AT-IC can achieve a better tail response time

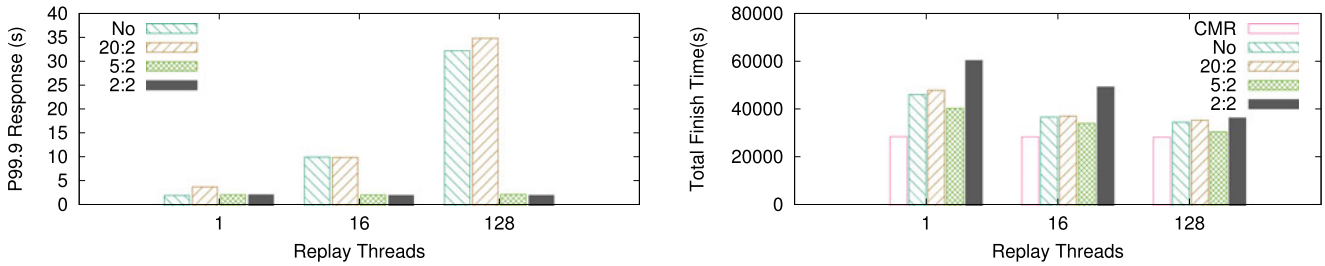


Fig. 5. P99.9 response time and total finish time of different *idle ratios*.

than no controls. The P99.9 response time of AT-IC with $< 5:2 >$ and $< 2:2 >$ is reduced by over 90 percent with 16 and 128 threads respectively compared to the results of no controls since blocking cleanings are not triggered when using AT-IC. Even though the concurrent requests wait in the I/O queue during an idle duration, they are satisfied in a short time after the idle duration. While if a blocking cleaning is triggered, the drive spends up to seconds to serve a request. The I/O response time will be increased to a huge value due to the extended waiting time, as we discussed in Section 2. The results of $< 20:2 >$ shows that with a too-small idle ratio, AT-IC only has limited benefits since it does not clean enough data. As a result, BC is still triggered.

The median (P50) response time of AT-IC is larger than that of the no control in all cases. The reason is that with fixed configurations, AT-IC is triggered from the beginning of the workload although there are still plenty available resources in the media cache. For no control, the response time will be only enlarged during a blocking cleaning. The number of requests with enlarged response time is smaller than that of AT-ICs.

Fig. 5b shows workload completion time. The results of $< 20:2 >$ indicate that with a too-small idle ratio, AT-IC can not finish the workload earlier. While as the idle ratio increasing ($< 5:2 >$), the workload completion time can be reduced by 10–15 percent compared to those without any controls. The reason is that AT-IC avoids the large response time increased by the blocking cleaning. However, if the idle ratio is too large, e.g., $2:2$, the total completion time will be enlarged due to the excessive idle duration, even though the blocking cleaning can be avoided.

Idle Length. In this experiment, we discuss the performance influences from different idle lengths while keeping the same idle ratio (about 28.75 percent). Fig. 6 shows the distribution of the response time. Fig. 7 displays the P99.9 response time and the workload finish time. Comparing the results of $< 2.5:1 >$ and $< 5:2 >$, we find that by shortening the idle length, the tail response time of AT-IC can be further reduced. However, if the idle length is too short, AT-IC may have no effects. The distribution of the response time of $< 1.25:0.5 >$ is close to that of no control. Since an idle cleaning needs about 250 ms to trigger, almost half of the idle time is wasted without cleaning data. The total finish time of $< 1.25:0.5 >$ is also larger than those of $< 2.5:1 >$ and $< 5:2 >$ as shown in Fig. 7b.

3.2.2 Workload Characteristics

We have investigated the following factors of a workload including request arrival rates (Arrival Interval), ratios of non-sequential SMR writes (NSSW ratio), counts of non-sequential zones (NSZ) with I/O sizes of 1 MB and 8 KB respectively. We only show the results of 1 MB I/O size in Table 2. The evaluations with 8 KB I/O size have similar results since the cleaning efficiency of the media cache is determined by the distance between the update offset and the write pointer location [2]. Since we have discussed the detailed results in Section 3.2.1, in this evaluation, we demonstrate the tail (P99.9) response time and the completion time of the workload (Comp. Time). In Table 2, we synthesize workloads with 35 GB total NSSW which is enough to fill up the media cache (28 GB). For each workload, we execute the workload with AT-IC ($< Execution : Idle >=< 5:2 >$) using independent 16 threads. We compare the P99.9 response time and workload finish time of AT-IC with those of no control (No).

Request Arrival Speed. We use the time interval between requests (Arrival Interval) in Table 2 to represent the request arrival rate. In this experiment, the non-sequential writes spread over 1,024 zones. We set the NSSW ratio to 10 percent (10 percent sequential writes and 80 percent read) and vary the Arrival Interval among 30 ms, 50 ms, and 100 ms. A smaller Arrival Interval means a higher request arrival rate, i.e., the workload is more intense.

In the results of different Arrival Intervals, both of the tail response time and completion time for no control become larger for the smaller Arrival Intervals. In a more intense

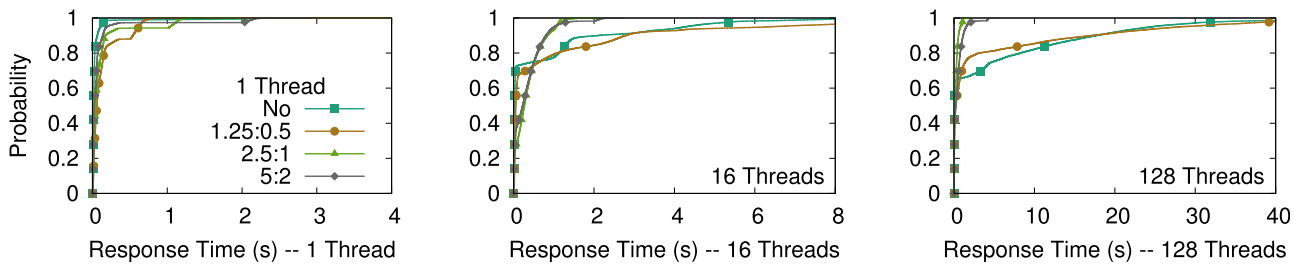
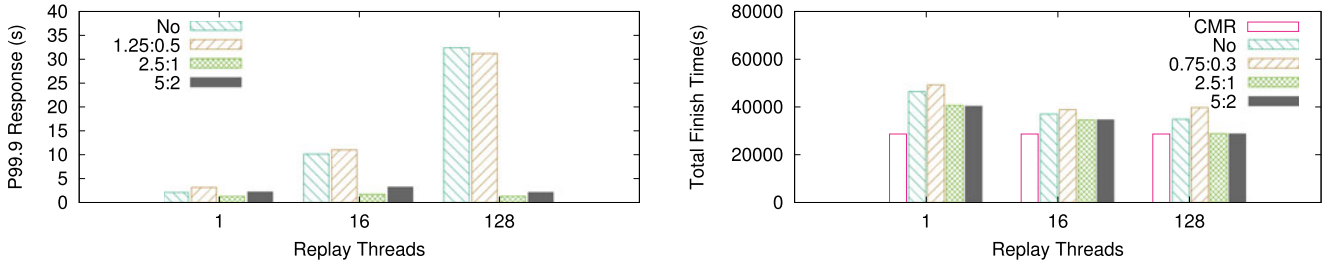


Fig. 6. CDF of the response time with different *idle lengths*.



(a) P99.9 response time (b) Total finish time

Fig. 7. P99.9 response time and total finish time of different *idle lengths*.

workload, e.g., 20 ms request arrival interval, the filling speed of media cache is faster than the cleaning speed of AT-IC. Therefore, AT-IC can not avoid blocking cleaning.

AT-IC can achieve better performance with larger Arrival Intervals. When the Arrival Interval becomes 50 ms, AT-IC can reduce the tail I/O response time and the total completion time by 72.2 and 8.1 percent respectively. When the Arrival Interval is 100 ms, AT-IC still reduces the P99.9 response time of no control by 47.1 percent. Therefore, we argue that workload intensity (Arrival Interval) can significantly influence the effect of AT-IC. AT-IC works better in light-weight workloads. With a very intensive workload, AT-IC may need to be reconfigured for a longer idle duration or a higher idle ratio to clean the media cache more efficiently.

NSSW Ratio. In the "NSSW Ratio" of Table 2, we generate workloads with different NSSW ratios (10, 20, and 40 percent). We keep the same ratios of SSW (10 percent) and Read (80, 70, and 50 percent) with 100 ms request arrival intervals. NSSW spread among 1,024 zones. In the results without control, a higher NSSW ratio results in a worse drive performance since more write requests are blocked during a BC. The service latency of a write is increased more than that of a read based on our previous evaluations (Fig. 2b). The blocked I/O requests accumulate a larger I/O response time with more non-sequential writes.

By comparing the results of no control to that of AT-IC, we find that with 10 and 20 percent NSSW ratios, AT-IC can reduce the tail response time by 50.9 and 55.3 percent respectively. The workload completion time of AT-IC is also closer to that of a CMR drive comparing with the completion time without controls. However, with a large NSSW ratio like 40 percent, AT-IC with the current configuration cannot achieve enough benefits since it can not avoid BCs. Therefore, we confirm that NSSW ratio significantly influences the AT-IC performance. AT-IC may need to be configured with a longer idle duration for workloads with higher NSSW ratios to achieve higher cleaning efficiency.

TABLE 2
Workload Characteristics (1 MB I/O Size)

Arrival Interval				
Value	P99.9 (s)		Comp. Time (s)	
	No	AT-IC	No	AT-IC
20ms	12.99	14.37	7617	8832
50ms	9.22	2.56	24559	22546
100ms	4.32	2.12	40375	38861
NSSW Ratio				
Value	P99.9 (s)		Comp. Time (s)	
	No	AT-IC	No	AT-IC
10%	4.32	2.12	40375	38861
20%	6.78	3.03	22112	19356
40%	12.32	10.22	11018	11128
NSZ Count				
Value	P99.9 (s)		Comp. Time (s)	
	No	AT-IC	No	AT-IC
8	1.1	2.23	36037	38066
1024	4.32	2.12	40375	38861
2048	4.66	2.57	41003	39230

Counts of Non-Sequential Zones. In the "NSZ Count" in Table 2, we keep the workloads with the same NSSW ratio (10 percent), and request arrival speed (100 ms). However, we spread the non-sequential writes among different numbers of zones (8, 1,024, and 2,048). When the NSZ count is 8, a blocking cleaning can be finished in a short time. Therefore, the drive does not have exceptionally long I/O response times. When the non-sequential writes spread among more zones, a higher I/O response time is produced because a blocking cleaning will last longer. The drive will suffer a bad performance for a longer duration. I/O Requests thereby accumulate a longer I/O response time.

By comparing the results of AT-IC to those without control, we find that AT-IC is able to reduce the I/O response time when the requests are distributed among more zones. When the count of non-sequential zones is very small, e.g., 8, the P99.9 response time of AT-IC is more than twice of that without controls since the response time is enlarged by the induced idle duration. Therefore, for workloads with a small count of non-sequential zones, AT-IC may not be helpful. A blocking cleaning can finish in a very short time without influencing the drive performance significantly.

Conclusions. We have validated that AT-IC can significantly reduce the tail response time of HA-SMR drives. Both idle configuration and workload characteristics significantly influence the effects of AT-IC. Carefully configuring the idle length and idle ratio is required if AT-IC wants to achieve excellent performance benefits. However, the characteristics in a real workload will dynamically change from time to time. They are harder to handle by AT-IC with a fixed idle configuration. Besides, as we discussed, AT-IC with fixed configurations has a long median (P50) response time, indicating AT-IC blocks more requests than no controls. The reason is that AT-IC does not consider the current utilization of the media cache and starts idle duration from the beginning of the workloads. Therefore, in the next section, we will discuss how to improve the drive performance by using artificially triggered idle cleanings with adaptive idle configurations considering both the utilization of the media cache and workload characteristics.

4 IDLER: ARTIFICIALLY TRIGGERING CLEANINGS WITH ADAPTIVE IDLE DURATIONS

4.1 Basic Principle

Idler can trigger idle cleaning using adaptive idle durations based on the current media cache resource utilization as well as real-time workload characteristics. Typically, Idler needs to decide when to trigger an idle duration and how long the idle duration should be. Based on the previous evaluations in Section 3.2.2, Idler considers the following parameters: $\boldsymbol{U}$ is the media cache resource utilization including free space and mapping slots. We set two thresholds, U_{low} and U_{high} ($0 < U_{low} < U_{high} < 100\%$). $\boldsymbol{T}_r$ is the average time interval between consecutive requests. $\boldsymbol{T}_{lat}$ is the average request latency. $\boldsymbol{V}_m$ and $\boldsymbol{V}_c$ represent the consuming and cleaning speeds of media cache resources respectively. $\boldsymbol{L}$ is the length of the duration to trigger idle cleaning.

These parameters are continuously monitored, and the conditions are periodically evaluated. Both media cache space utilization and the usage of mapping entry slots are monitored and evaluated with two thresholds U_{low} and U_{high}. Since blocking cleanings can be triggered either by depleted space or mapping entry slots, U will consider the resource, space or mapping entry slots, closer to be depleted to represent the media cache utilization. When $U < U_{low}$, Idler will not trigger an idle duration since a good amount of resources are still available in media cache. When $U_{low} < U < U_{high}$, we start to trigger an idle duration. The idle duration length L is decided based on Equation (1). Since media cache still has a certain amount of resources, Idler will not trigger a long idle duration to avoid introducing a large I/O response time. We calculate the average latency of the requests T_{lat} and compare T_{lat} with the average time interval between requests T_r. If $T_r \leq T_{lat}$, it means the requests are issued relatively intensively. In this case, Idler will not trigger an idle duration ($L = 0$) since any idle duration may block a large number of requests. When $T_r > T_{lat}$, there are some time intervals between requests. In this condition, Idler triggers the idle duration with a minimal length to limit the maximal I/O response time. We set the minimum length to 0.3s here to make sure that idle cleanings will be triggered.

$$L = \begin{cases} 0, & T_r \leq T_{lat} \\ 0.3s, & T_r > T_{lat} \end{cases}. \tag{1}$$

When $U > U_{high}$, Idler has to clean the media cache more aggressively. In this condition, L is determined by Equation (2). If the workload is not intensive ($T_r > T_{lat}$), Idler will trigger an idle duration with a maximal length to achieve a higher cleaning efficiency. We set the maximal length to 2s which is equal to the largest service latency when a blocking cleaning is triggered to limit the maximal I/O response time. If $T_r \leq T_{lat}$, it means the current workload is intensive. A large number of requests will be delayed during an idle duration. To clean the media cache more efficiently and minimize the I/O response time, Idler will further compare the consuming speed V_m to the cleaning speed V_c of the media cache resource. If $V_m > V_c$, it means U will keep increasing. Idler has to increase L to achieve better cleaning efficiency. When $V_m \geq V_c$, Idler can decrease U with the current L. Therefore, Idler reduce the L to reduce the I/O response time further.

$$L = \begin{cases} L - 0.1s, & T_r \leq T_{lat} \;\&\; V_m < V_c \\ L + 0.1s, & T_r \leq T_{lat} \;\&\; V_m > V_c \\ 2s, & T_r > T_{lat} \end{cases}. \tag{2}$$

4.2 Idler Implementation

Media Cache Resource Monitoring. Monitoring media cache resources is challenging. There are no available interfaces to check the media cache resource utilization directly. The only useful interface is the *report_zone* interface in libzbc [11], [13]. It can check the states of zones (sequential or non-sequential) and the write pointer location of each zone. The *report_zone* is an intrusive command incurring a relatively large performance overhead. Therefore, it cannot be called frequently. However, we do not need the precise values of these parameters.

To reduce the performance overhead, we only estimate the media cache resource utilization U instead of getting an accurate value. Idler maintains a ghost media cache which records the total amount of consumed resources (both space and mapping entries). It also remembers the media cache resources used by each zone and the write pointer location of each zone. For non-sequential writes directed to media cache, Idler keeps updating U. For sequential writes, Idler will also update the location of write pointers. Since data is buffered in media cache with a log structure [10], repetitive updates to the same address will also be counted.

We periodically update the information in the ghost media cache by checking the current zone states using *report_zone* (every 10 minutes). By comparing the set of non-sequential zones in ghost media cache with that from the drive media cache, we identify the cleaned zones and the cleaning speed (V_c) during the last period. U can be updated by eliminating the resources used by the cleaned zones.

Workload Characterization. Idler calculates request intervals T_r, the average latency of requests T_{lat} and the consuming speed of the media cache resource V_m during the execution duration. After an idle duration, both of the blocked requests and newly issued requests will be served during the execution duration. Since the blocked requests are issued in an intensive way without following their original time intervals, the calculations of T_r and V_m are only performed after we finish all of the delayed requests. For the un-delayed requests, Idler samples 10 consecutive requests to calculate T_r, T_{lat} and V_m. T_r and T_{lat} will be counted for each new request, while V_m will be counted for only non-sequential writes.

Idle Frequency. The idle frequency is determined by both idle duration and execution duration. In Idler, the execution duration is not fixed. The length of an execution duration is long enough to serve all blocked requests from the previous idle duration plus some number of new requests such that enough new requests can be sampled to determine the next idle duration. If Idler decides not to trigger an idle duration, a new execution duration will start right away.

4.3 Integrating Idler With a Write Buffer

In many scenarios, storage systems may include a high-speed write buffer to speed up the writes. The difference

between an external write buffer and the on-device media cache is that writing to the external buffer will not interrupt the data cleaning on SMR drives. Although writes to and reads from an write buffer may create idle durations, the triggered idle cleaning may not clean enough data in media cache since uncontrolled buffer evictions and buffer-missed reads will interrupt the idle cleanings. Idler can also be helpful when it is integrated with a write buffer to control the page evictions and buffer-missed reads. Besides, this write buffer also provides opportunities to reduce the I/O response time introduced by the triggered idle durations since it can temporarily hold writing requests during the idle duration. The data temporarily buffered at the write buffer will eventually migrate to SMR drives.

Designing an efficient write buffer for SMR drives is not the primary contribution of this paper. Existing studies have discussed multiple designs to write buffers for SMR drives [2], [14], [15], [16]. Our goal here is to study how Idler designed for SMR drives can work together with a write buffer in SSD to further improve the performance of SMR drives. To simplify our discussion, we make the following assumptions for a write buffer.

The write buffer is a page-based write buffer on SSD. Here we set the page size to 4 KB. A read request will check the buffer first and then go to SMR drives if they cannot be satisfied from the buffer. We assume a simple eviction policy–the Zone-based Least Recently Used (ZLRU). ZLRU remembers the access time for all of the zones who have data buffered. The victim zone will be selected by LRU. All of the buffered data from the same zone will be evicted together. We have two data paths when evicting data from the write buffer to SMR drives. If we have more than 400 pages (i.e., 1.6 MB) to be cleaned for a zone, we write the data back to its designated zone directly by read-merge-writes. Otherwise, we evict the data to the media cache. We choose 400 pages as a threshold since we assume the average service latency for a 4 KB write is about 10 ms, and it takes about 4–6s to read a zone out, reset the write pointer and write data back.

The write buffer consists of a free page pool and a page map. We load the map into memory and persist it in a reserved region on SSD. The write buffer allocates new pages from the free page pool to buffer new writes. For SMR drives, a victim zone list is maintained based on ZLRU. If not enough free pages are available, page evictions will be triggered. The reclaimed pages will be returned to the free page pool.

We integrate Idler with a write buffer in the following way. During an idle duration, the write buffer will temporarily hold both non-sequential writes and sequential writes if it has free space. Reads will not be delayed if they can be read from the buffer. At the meantime, we set up a wait queue to receive requests during an idle duration under two conditions. First, if the buffer is full and the data to be written are not in the buffer, the writes will be put into the wait queue. Second, reads will also be put into the queue if they cannot be satisfied from the buffer. In the next execution duration, we will issue the delayed reads and SSWs in the wait queue to SMR drives. The delayed NSSWs will be buffered in the write buffer after zone evictions.

The basic principle discussed in Section 4.1 is still applied. That is, both workload characteristics and media cache resource utilization will be considered when deciding the length of an idle duration. We still use the ghost media cache, as discussed in Section 4.2, to estimate the media cache resource utilization. However, instead of increasing media cache resource utilization for each non-sequential writes, we update the used media cache resource information during the eviction. If the data is evicted, we increase the media cache resource utilization accordingly in the ghost media cache. We still periodically (every 10 minutes) check the zone states by $report\_zone$ interface and adjust media cache resource utilization(U) and the cleaning speed (V_c).

We characterize the workload during the execution duration. The average request interval and the average request latency are counted only for the buffer-missed requests (including non-sequential writes, sequential writes, and reads) since buffer-missed requests will probably be delayed during an idle duration. When calculating the consuming speed of the media cache resource V_m, we only count the non-sequential writes that do not have a hit in the buffer. The non-sequential writes will not increase the media cache utilization if they can be served from the buffer.

TABLE 3
Characteristics of Traces

trace	write counts & sizes	zones	write%
proj_1	2,496,935 & 25.576 GB	1,475	10.56%
prn_1	2,769,610 & 30.785 GB	1,222	24.66%
usr_1	3,857,714 & 56.127 GB	1,886	8.52%
usr_2	1,994,612 & 26.469 GB	926	18.87%
proj_2	3,624,878 & 168.686 GB	1,474	12.39%

5 Idler Evaluation

We implement Idler as a block I/O replay engine and evaluate it by executing real-world traces [17]. To emulate the different I/O dependencies, we still partition the traces into different numbers of sub-traces and replay with separate threads. Again the requests in the same partitions are dependent. Precisely finding I/O dependencies from block traces is hard without extra information. HFReplayer [18] can approximately estimate I/O dependencies in block traces based on the latency information. Therefore, we use a similar algorithm when partitioning a trace.

When executing the traces, we respect the request intervals between the requests to emulate their characteristics. We run the traces on an HA-SMR drive using No Control (regular operation without idle durations), AT-IC, and Idler respectively. After we try multiple times, we configure AT-IC with $< 5 : 2 >$ since it provides a better result. For Idler, we set $< U_{low}, U_{high} >$ to $< 0.1, 0.8 >$ to allow the media cache to buffer enough data before an idle cleanings is triggered. And after U_{high} is reached, the media cache still has available resources. We also execute the traces on a Conventional Magnetic Recording (CMR) drive and compare the workload completion time.

The characteristics of the workloads are shown in Table 3 including the counts of writes, the total write sizes, counts of non-sequential zones, and the write ratio. The traces are captured from scenarios of user home directories (usr_1, usr_2), project directories (proj_1, proj_2) and print servers

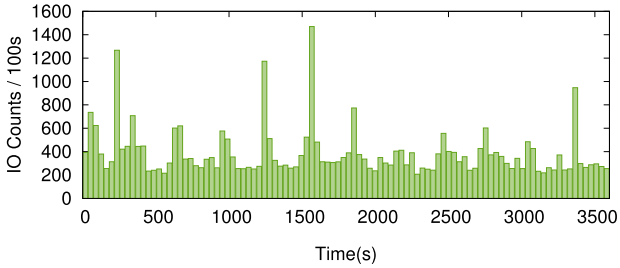


Fig. 8. The variation of the request intensity in proj_1.

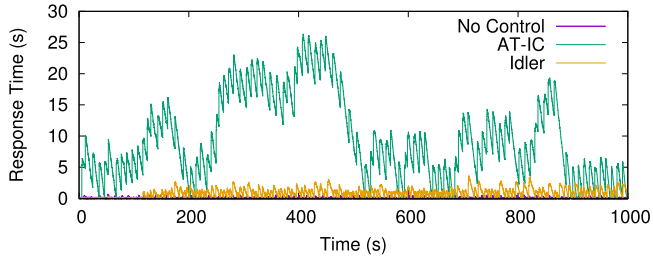


Fig. 9. Response time of proj_1 *before BC starts* with 128 threads.

(prn_1) [19]. I/O response time is significant for those scenarios. Traces are captured on a CMR drive. Almost all of the writes are non-sequential. Blocking cleanings are triggered by depleting the mapping entry slots, although the writing sizes of traces are not significantly larger than the size of media cache. During a workload, requests are not issued with a unified rate. Fig. 8 uses the trace proj_1 as an example to show the variation of I/O counts for every 100 seconds within an hour. The intensity of the workload varies a lot with time. We partition the trace into 1, 16 and 128 sub-traces to emulate different conditions of I/O dependencies.

Fig. 9 shows the response time of three designs in non-cleaning condition (the first 1,000s) of the workload under 128 I/O threads. During the first 1,000s, No Control has the shortest response time since there are no artificially triggered idle durations delaying the subsequent requests. However, with a fixed configuration, the response time of AT-IC varies a lot. When the workload is I/O intensive, the predefined execution time may not be able to finish all of the blocked requests from the previous idle duration. Thus, AT-IC accumulates a long response time. The proposed Idler can adjust its idle length and execution length dynamically. When the workload is intensive, Idler will keep the idle duration short if the media cache still has a lot of available resources. Even with a longer idle duration, Idler can make sure that the next execution duration will finish all of the blocked requests from the previous idle duration. Therefore, Idler can keep the response time in a small value (mostly under a second) even though they are still longer than those of No Control when no blocking cleaning is triggered.

Fig. 10 shows the CDF plot of the response time. We also compare the P99.9 response time and workload finish time in Fig. 11. The results of no control indicates that the blocking cleaning is triggered although the workload may have some burstiness. The idle cleanings triggered by the natural idle duration cannot clean the media cache efficiently. Idler is able to provide better response time than AT-IC and no control since Idler only triggers minimal required idle durations to avoid blocking cleanings with limited performance overheads. Under a single thread, Idler reduces the P99.9 response time and workload finish time by 56 and 23 percent compared with those of without a control. The improvement is more significant if having more independent I/O requests for Idler. The P99.9 response time is reduced by 83.2 and 88.8 percent with Idler compared to the results without control under 16 and 128 threads respectively. However, the tail response time of Idler is still not comparable with that of a CMR drive, even though the total completing time of the workload with Idler is closer, about 8–10 percent longer, to that of a CMR drive. Results in Fig. 11b indicate Idler can finish the workload in a shorter time. Since we issue the requests according to their

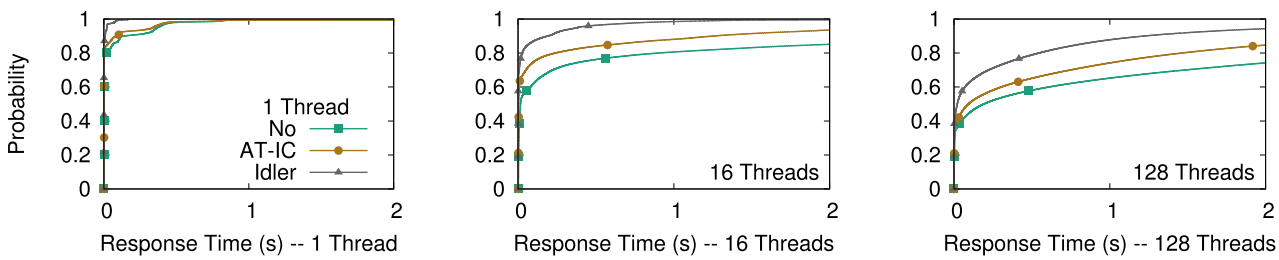


Fig. 10. CDF of the response time with *proj_1*.

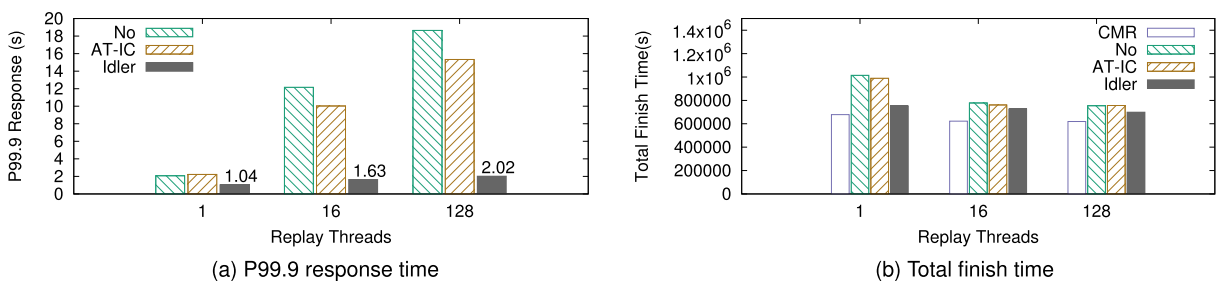


Fig. 11. P99.9 response time and total finish time of *proj_1*.

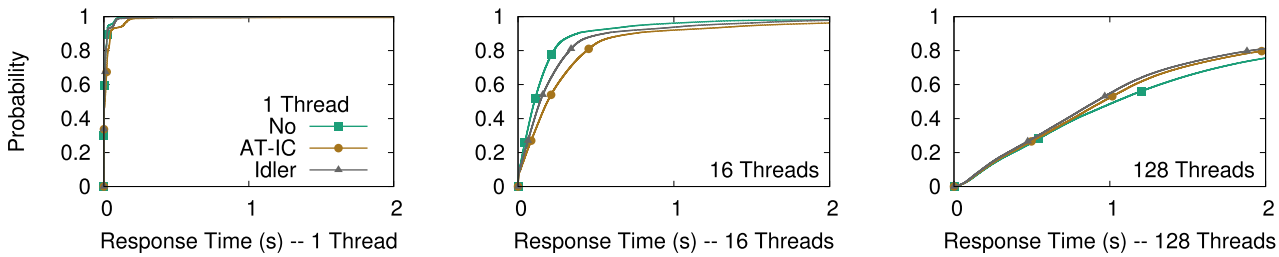


Fig. 12. CDF of the response time with *prn_1*.

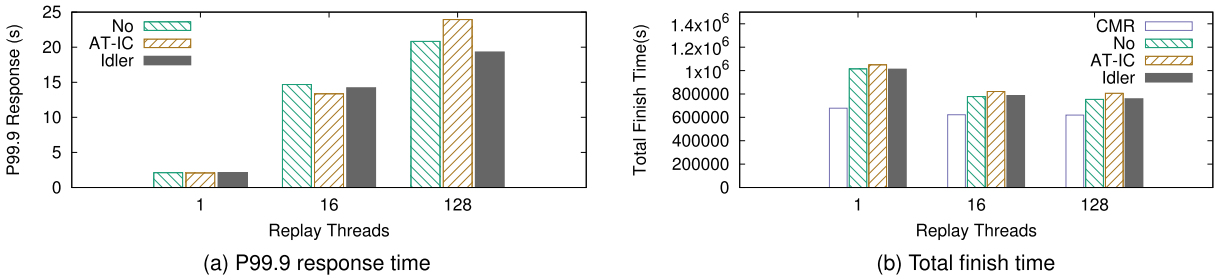


Fig. 13. P99.9 response time and total finish time of *prn_1*.

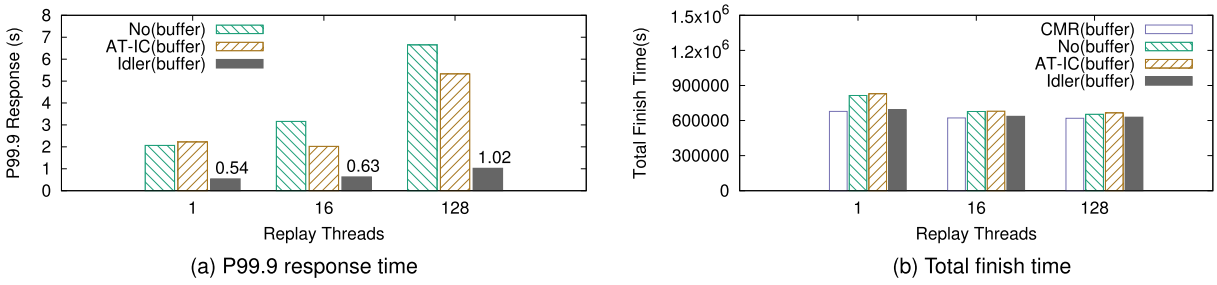


Fig. 14. P99.9 response time and total finish time of *proj_1* with a buffer.

timestamp when replay the trace and no blocking cleaning is triggered, the total finish time of Idler under different numbers of I/O threads is close to each other.

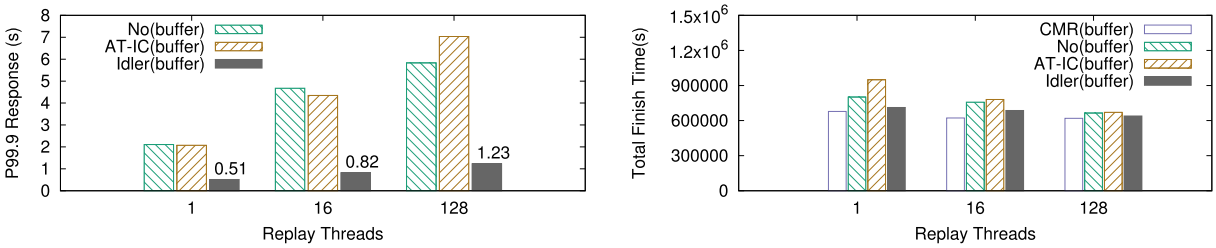
Idler needs extra memory and computing resources for buffering the delayed requests during an idle duration and updating the ghost cache information. Therefore, we measure the memory and CPU overheads of no control, AT-IC, and Idler while replaying the trace proj_1 with different numbers of threads. The maximum memory consumption is similar under different I/O threads which are about 3.9 MB (no control), 4.2 MB (AT-IC) and 5.1 MB (Idler). Idler does not introduce a significant memory overhead since it keeps the idle duration small and will not buffer too many requests. However, Idler needs some memory space to store the ghost cache information. Idler does not introduce a significant CPU overhead either since updating ghost cache information for each request only modifies several variables and *report_zone* is called infrequently. The differences in CPU utilization among no control, AT-IC, and Idler under different I/O threads are less than 10 percent.

However, the results of Figs. 12 and 13 shows that when the workload has a higher ratio of non-sequential writes (24.66 percent), both AT-IC and Idler can not help to reduce the response time. The distributions of the response time of AT-IC and Idler are similar to those of no controls. Both of the P99.9 response time and workload completion time are close to those of no controls. Therefore, we recommend that Idler can be deployed in the workloads with low ratios of non-sequential writes under 10 percent.

We further evaluate Idler with three more traces (usr_1, usr_2, proj_2) with 16 I/O threads. The characteristics of those workloads are shown in Table 3. Fig. 16 shows the tail response time of these three traces. The results indicate that Idler can significantly reduce the tail response time in the workloads with a small non-sequential write ratio (under 10 percent) compared with the results with no control.

Idler can reduce the response time compared to that without any control in workloads with low non-sequential write ratios. However, the response time of Idler can still be large. The P99.9 response time can be up to seconds depending on the configurations of maximal idle length. Besides, for workloads with high non-sequential write ratios, Idler can not avoid the blocking cleaning. Therefore, we further integrate the Idler with a write buffer on SSD. The buffer can absorb the non-sequential writes and temporarily hold the requests during an idle duration. To demonstrate the effectiveness, we only use a small buffer (64 MB). The performance results will have no significant differences if we deploy the buffer in host memory. When executing workloads on the CMR drive with a write buffer, we use regular LRU instead of ZLRU evictions.

Figs. 14 and 15 shows the P99.9 response time and the total finish time of Idler with a write buffer. For "No" and



(a) P99.9 response time (b) Total finish time

Fig. 15. P99.9 response time and total finish time of *prn_1* with a buffer.

AT-IC, a write buffer cannot reduce the tail I/O response time since blocking cleaning is triggered. The page evictions and buffer-missed reads are still delayed. The results of Idler indicate that with the help of a small write buffer, the P99.9 response time can be further reduced compared to those without a write buffer in Fig. 11a. When the ratio of non-sequential writes is small (Fig. 14), the P99.9 response time can be shorter than a second. In the workloads with 10 percent ratio of non-sequential writes, the P99.9 response time is reduced by 71.3–87.7 percent. Meanwhile, Fig. 15 shows that with a write buffer, Idler can reduce the P99.9 response time by 70–85.2 percent in a workload with a higher ratio of non-sequential writes. The reason is that the write buffer absorbs a good amount of non-sequential traffics, Idler only triggers short idle durations.

6 Related Work

Most of the existing researches including file systems [20], [21], key-value stores [22], [23] and other storage systems [24], [25], [26], [27] target at HM-SMR drives. However, applications or systems have to be modified significantly or even re-designed since HM-SMR drives do not accept non-sequential workloads. Other studies also intend to help HM-SMR drive handle non-sequential workloads [28].

For DM/HA-SMR drives, existing studies focus on performance modeling and evaluations of the drives. Skylight [10] and Wu *et al*. [2] conduct evaluations on DM- and HA-SMR drives respectively to find the internal structure, cleaning policies, etc. Mhalagi develops a simulator to simulate the drive performance [29]. Shafaei also accurately models the performance of DM-SMR drives [30]. Aghayev involves ext4 in DM-SMR drives [31], [32].

Besides, there are other designs for HA-SMR drives to avoid the long response time. H-Buffer [2] and VPC [3] intend to improve the performance of HA-SMR drives by utilizing a log-structured buffer/cache on shingled zones in additions to the Media Cache. SMRC is another design using an SSD as the upper-layer cache for SMR drives [16]. Idler can also be integrated with them.

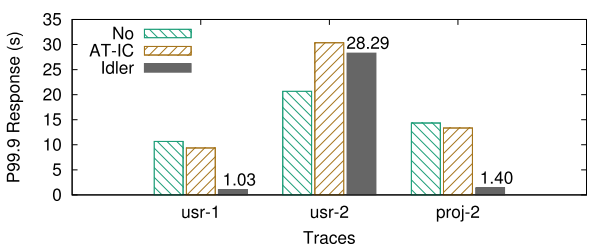


Fig. 16. P99.9 response time of different traces.

7 Conclusion and Future Work

In this paper, we evaluate the cleaning process of HA-SMR drives. We identify how data cleaning affects the drive performance. We propose a scheme called Idler. Our evaluations show that Idler can avoid the extremely long and unpredictable I/O response time in workloads with an NSSW ratio smaller than 10 percent. In the future, we would like to deploy Idler with some applications like database and key-value store to verify the performance improvement.

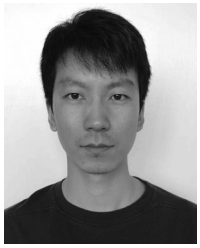
Acknowledgments

This work was partially supported by NSF I/UCRC Center for Research in Intelligent Storage and the following NSF awards 1439662, 1525617, 1536447, 1708886, 1763008, and 1812537.

References

[1] T. Feldman and G. Gibson, "Shingled magnetic recording areal density increase requires new data management," *USENIX; login: Mag.*, vol. 38, no. 3, pp. 22–30, 2013.
[2] F. Wu, M.-C. Yang, Z. Fan, B. Zhang, X. Ge, and D. H. Du, "Evaluating host aware SMR drives," in *Proc. 8th USENIX Workshop Hot Top. Storage File Syst.*, 2016, pp. 31–35.
[3] M.-C. Yang, Y.-H. Chang, F. Wu, T.-W. Kuo, and D. H. Du, "On improving the write responsiveness for host-aware SMR drives," *IEEE Trans. Comput.*, vol. 68, no. 1, pp. 111–124, Jan. 2019.
[4] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble, "Tales of the tail: Hardware, OS, and application-level sources of tail latency," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–14.
[5] M. E. Haque *et al.*, "Few-to-many: Incremental parallelism for reducing tail latency in interactive services," *ACM SIGPLAN Notices*, vol. 50, pp. 161–175, 2015.
[6] M. Hao, G. Soundararajan, D. R. Kenchammana-Hosekote, A. A. Chien, and H. S. Gunawi, "The tail at store: A revelation from millions of hours of disk and SSD deployments," in *Proc. 14th USENIX Conf. File Storage Technol.*, 2016, pp. 263–276.
[7] N. Li, H. Jiang, D. Feng, and Z. Shi, "PSLO: Enforcing the X^{th} percentile latency and throughput SLOs for consolidated VM storage," in *Proc. 11th Eur. Conf. Comput. Syst.*, 2016, Art. no. 28.
[8] S. Kim, Y. He, S.-W. Hwang, S. Elnikety, and S. Choi, "Delayed-dynamic-selective (DDS) prediction for reducing extreme tail latency in web search," in *Proc. 8th ACM Int. Conf. Web Search Data Mining*, 2015, pp. 7–16.
[9] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "PriorityMeister: Tail latency QoS for shared networked storage," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–14.
[10] A. Aghayev, M. Shafaei, and P. Desnoyers, "Skylight-A window on shingled disk operation," *ACM Trans. Storage*, vol. 11, no. 4, 2015, Art. no. 16.
[11] T. T. Committee, "Information technology—zoned block commands (ZBC)," 2014. [Online]. Available: http://www.t10.org/ftp/zbcr01.pdf
[12] Linux, "Kernel asynchronous I/O (AIO) support for Linux," 2004. [Online]. Available: http://lse.sourceforge.net/io/aio.html

[13] A. M. Damien Le Moal, "ZBC device manipulation library," 2015. [Online]. Available: https://github.com/hgst/libzbc
[14] C.-I. Lin, D. Park, W. He, and D. H. Du, "H-SWD: Incorporating hot data identification into shingled write disks," in *Proc. IEEE 20th Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2012, pp. 321–330.
[15] C. Wang, D. Wang, Y. Chai, and D. Sun, "Larger cheaper but faster: SSD-SMR hybrid storage boosted by a new SMR-oriented cache framework," in *Proc. 33rd Int. Conf. Massive Storage Syst. Technol.*, 2017.
[16] X. Xie, L. Xiao, X. Ge, and Q. Li, "SMRC: An endurable SSD cache for host-aware shingled magnetic recording drives," *IEEE Access*, vol. 6, pp. 20 916–20 928, 2018.
[17] M. R. Cambridge, "MSR Cambridge traces," 2008. [Online]. Available: http://iotta.snia.org/traces/388
[18] A. Haghdoost, W. He, J. Fredin, and D. H. Du, "On the accuracy and scalability of intensive I/O workload replay," in *Proc. 15th USENIX Conf. File Storage Technol.*, 2017, pp. 315–328.
[19] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *ACM Trans. Storage*, vol. 4, no. 3, 2008, Art. no. 10.
[20] A. Palmer, "SMRFFS-EXT4," 2015. [Online]. Available: https://github.com/Seagate/SMR_FS-EXT4
[21] C. Jin, W.-Y. Xi, Z.-Y. Ching, F. Huo, and C.-T. Lim, "HiSMRfs: A high performance file system for shingled storage array," in *Proc. 30th Symp. Mass Storage Syst. Technol.*, 2014, pp. 1–6.
[22] R. Pitchumani, J. Hughes, and E. L. Miller, "SMRDB: Key-value data store for shingled magnetic recording disks," in *Proc. 8th ACM Int. Syst. Storage Conf.*, 2015, Art. no. 18.
[23] T. Yao *et al.*,"GearDB: A GC-free key-value store on HM-SMR drives with gear compaction," in *Proc. 17th USENIX Conf. File Storage Technol.*, 2019, pp. 159–171.
[24] W. He and D. H. Du, "SMaRT: An approach to shingled magnetic recording translation," in *Proc. 15th USENIX Conf. File Storage Technol.*, 2017, pp. 121–134.
[25] C.-F. Wu, M.-C. Yang, and Y.-H. Chang, "Improving runtime performance of deduplication system with host-managed SMR storage drives," in *Proc. 55th Annu. Des. Autom. Conf.*, 2018, Art. no. 57.
[26] D. Park, Z. Fan, Y. J. Nam, andD. H. Du,"A lookahead read cache: Improving read performance for deduplication backup storage," *J. Comput. Sci. Technol.*, vol. 32, no. 1, pp. 26–40, 2017.
[27] P. Macko *et al.*, "SMORE: A cold data object store for SMR drives (extended version)," in *Proc. 34th Symp. Mass Storage Syst. Technol.*, 2017.
[28] A. Manzanares, N. Watkins, C. Guyot, D. Le Moal, C. Maltzahn, and Z. Bandic, "ZEA, A data management approach for SMR," in *Proc. 8th USENIX Workshop Hot Top. Storage File Syst.*, 2016, pp. 26–30.
[29] S. Mhalagi, L. Duan, and P. Rad, "Designing and evaluating hybrid storage for high performance cloud computing," in *Proc. Annu. IEEE Int. Syst. Conf.*, 2018, pp. 1–8.
[30] M. Shafaei, M. H. Hajkazemi, P. Desnoyers, and A. Aghayev," Modeling SMR drive performance," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 44, no. 1, pp. 389–390, 2016.
[31] A. Aghayev, Y. Theodore, G. Gibson, and P. Desnoyers, "Evolving ext4 for shingled disks," in *Proc. 15th USENIX Conf. File Storage Technol.*, 2017, pp. 105–120.
[32] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: Current status and future plans," in *Proc. Linux Symp.*, 2007, vol. 2, pp. 21–33.



Baoquan Zhang is working toward the PhD degree in computer science at the University of Minnesota Twin Cities (UMN), Minneapolis, Minnesota advised by professor David H.C. Du. Currently, He is working on Memory/Storage Systems, Key-Value Stores, RAID Systems and so on. As a research assistant (June 2013—April 2015), He worked in Cloud Storage and Distributed Systems at Research Institute of Information Technology, Tsinghua University, Beijing, China.

Ming-Hong Yang received the BS and MS degrees in computer science from National Tsing Hua University, Taiwan, in 2008 and 2010, respectively. He is currently working toward the PhD degree in computer science at the University of Minnesota, Twin Cities, Minneapolis, Minnesota. His research interest includes system design for emerging storage devices and technologies such as SMR drives and storage class memories, caching policy design, protocol design for wireless networks, and software-defined networks.

Xuechao Xie received the PhD degree in computer science from the National University of Defense Technology (NUDT), China, in 2018. Currently he is an assistant professor with the College of Computer, NUDT, China. His current research interests include high performance computing, file and storage systems with non-volatile memory, solid-state drives, and shingled magnetic recording drives.

David H. C. Du is currently the Qwest chair professor in computer science and engineering at the University of Minnesota Twin Cities, Minneapolis, Minnesota and the director of the NSF Center for Research in Intelligent Storage (CRIS). His current research interest includes intelligent and large storage systems. He serves on editorial boards of several international journals. He was a program director (IPA) at National Science Foundation (NSF) CISE/CNS Division from 2006 to 2008. He has served as conference chair, program committee chair, and general chair for several major conferences in database, security and parallel processing, He is fellow of the IEEE (since 1998).

Information Assurance Through Redundant Design: A Novel TNU Error-Resilient Latch for Harsh Radiation Environment

Aibin Yan, Yuanjie Hu, Jie Cui, Zhili Chen, Zhengfeng Huang, Tianming Ni, Patrick Girard, *Fellow, IEEE*, and Xiaoqing Wen, *Fellow, IEEE*

Abstract—In nano-scale CMOS technologies, storage cells such as latches are becoming increasingly sensitive to triple-node-upset (TNU) errors caused by harsh radiation effects. In the context of information assurance through redundant design, this article proposes a novel low-cost and TNU on-line self-recoverable latch design which is robust against harsh radiation effects. The latch mainly consists of a series of mutually interlocked 3-input Muller C-elements (CEs) that forms a circular structure. The output of any CE in the latch respectively feeds back to one input of some specified downstream CEs, making the latch completely self-recoverable from any possible TNU, i.e., the latch is completely TNU-resilient. Simulation results demonstrate the complete TNU-resiliency of the proposed latch. In addition, due to the use of fewer transistors and a high-speed path, the proposed latch reduces the delay-power-area product by approximately 91 percent compared with the state-of-the-art TNU hardened latch (TNUHL), which cannot provide a complete TNU-resiliency.

Index Terms—Redundant design, latch design, self-recoverability, resiliency, radiation effect, triple-node upset

1 Introduction

In nano-scale CMOS technologies, aggressive shrinking of transistor feature sizes drastically increases the susceptibility of digital devices to soft errors, resulting in information corruptions, execution errors, or even system crashes. Soft errors are mainly caused by alpha particles from package radioactive decay, neutrons from cosmic rays, and high-energy electrons [1], [2]. Circuits and systems used in harsh radiative (e.g., aerospace) environment are subject to soft errors, which are among the main sources of reliability decrease for safety-critical applications [3]. Moreover, as a result of the aggressive shrinking of transistor feature sizes, soft errors at the earth level have also become a matter of concern. Therefore, it is crucial to design soft-error-tolerant or even on-line self-recoverable storage modules in order to construct highly reliable circuits and systems for dependable computing in harsh radiative environment.

For the radiation-hardening-by-design of basic storage modules, many designs of memory cells [4], [5], [6], flip-flops [1], [7], [26], and latches [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25] have been proposed. For such storage modules, the state of a single node can be changed due to a striking-particle causing a soft error, which is called *single-node upset* (*SNU*). Various latch designs for tolerating SNUs have been proposed in the literature [8], [9], [23]. However, in modern CMOS manufacturing technologies, aggressive shrinking of transistor feature sizes may cause a single-event charge to simultaneously affect multiple nodes, thus causing *multiple-node upset* (*MNU*) [10], [11], [12], [13], [14], [15], [16], [19], [20], [24], [25]. The most common MNUs under consideration are *double-node upset* (*DNU*) [10], [11], [12], [13], [14], [15], [19], [24] and *triple-node upset* (*TNU*) [16], [20], [25]. Since MNUs can increase the soft error rate of storage modules and cannot be completely mitigated by the methods targeting only SNUs, the threat of MNUs to electronic circuits and systems is becoming increasingly serious. For designers and manufacturers of highly reliable circuits and systems, MNUs, especially TNUs, have become a serious concern in advanced CMOS technologies [16], [20], [25].

Recently, several latch designs for tolerating DNUs and/or TNUs have been proposed, making them more reliable against MNUs. However, these latch designs still suffer from some limitations. First, in some DNU hardened latches [19], there is still at least one node-pair that can be flipped if the striking-particle has a high energy. In other words, these latches are not sufficiently DNU hardened. Thus they

- *A. Yan, Y. Hu, J. Cui, and Z. Chen are with the Anhui Engineering Laboratory of IoT Security Technologies, and the School of Computer Science and Technology, Anhui University, Hefei, Anhui 230601, China. E-mail: {abyan, cuijie}@mail.ustc.edu.cn, 1045954240@qq.com, zlchen@ahu.edu.cn.*
- *Z. Huang is with the School of Electronic Science and Applied Physics, Hefei University of Technology, Hefei, Anhui 230009, China. E-mail: huangzhengfeng@139.com.*
- *T. Ni is with the College of Electrical Engineering, Anhui Polytechnic University, Wuhu, Anhui 241000, China. E-mail: timmyni@mail.hfut.edu.cn.*
- *P. Girard is with the Laboratory of Informatics, Robotics and Microelectronics of Montpellier, University of Montpellier/CNRS, 34095 Montpellier, France. E-mail: girard@lirmm.fr.*
- *X. Wen is with the Graduate School of Computer Science and Systems Engineering, Kyushu Institute of Technology, Fukuoka 8208502, Japan. E-mail: wen@cse.kyutech.ac.jp.*

Manuscript received 12 Apr. 2019; revised 18 Nov. 2019; accepted 8 Jan. 2020. Date of publication 15 Jan. 2020; date of current version 8 May 2020. (Corresponding author: Tianming Ni.) Recommended for acceptance by N. Cicero. Digital Object Identifier no. 10.1109/TC.2020.2966200

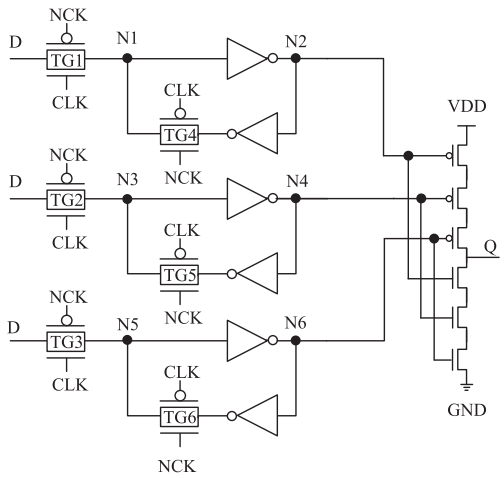


Fig. 1. A TMR based latch design.

cannot provide efficient/complete self-recoverability from DNUs caused by a high-energy striking-particle. Second, some DNU hardened latches can provide a complete DNU self-recoverability. However, they cannot provide a complete TNU tolerance [12], [13], [14], [15]. Third, the TNU hardened latch designs in [16], [20], [25] are completely TNU tolerant. However, they cannot provide a complete TNU self-recoverability. To the best of our knowledge, there is no latch design that can provide such a complete TNU self-recoverability against harsh radiation effects.

In this paper, a novel *Low Cost and TNU-self-Recoverable Latch* (*LCTNURL*) design against harsh radiation effects is proposed for dependable computing. The design mainly consists of many mutually interlocked 3-input *Muller C-elements* (*CEs*), forming a circularly linked schematic. For any CE, its output triply feeds back to one input of some specified downstream CEs, making the latch completely self-recoverable from any possible TNU, i.e., the latch can provide a complete TNU self-recoverability in addition to a complete TNU-tolerance, due to the special working principles of these interlocked 3-input CEs. Simulation results demonstrate the complete TNU self-recoverability and cost-effectiveness of the proposed LCTNURL latch. Moreover, compared with the *TNU hardened latch* (*TNUHL*) that cannot provide a complete TNU self-recoverability [16], the proposed LCTNURL latch reduces the delay-power-area product by approximately 91 percent, owing to the use of fewer transistors and a high-speed path. Note that, as self-recoverability means resiliency, we will use resiliency for brevity in the rest of this paper. Clearly, similar latches have been reported previously [20], [27]. However, to provide TNU-resiliency, more CEs with more inputs are needed.

The contribution of this paper can be summarized as follows. (1) We propose a novel latch with complete TNU (including DNU and SNU) resiliency, and the latch features a low transmission delay and a low delay-power-area product. (2) We propose new node-upset models that consider not only SNU, but also DNU and TNU, for fault-injections to validate the performance of hardened latch designs. (3) We consider joint SNU, DNU, and TNU resiliency requirements and propose new definitions of "resiliency", "immunity", "tolerance", and "mitigation" for a better classification of all hardened latch designs.

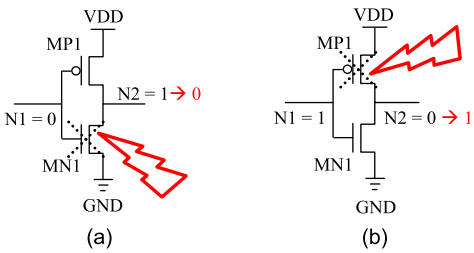


Fig. 2. SNU model. (a) A particle strikes MN1. (b) A particle strikes MP1.

The rest of the paper is organized as follows. Section 2 introduces some background, such as node-upset models, resiliency requirements, CEs, latch-reliability definitions, and latch classifications. Section 3 describes the schematic, working principles, and validations of the proposed TNU-resilient latch. Section 4 provides a comprehensive evaluation of the proposed latch and a comparison with state-of-the-art latches. Section 5 concludes the paper.

2 Preliminaries

2.1 Node-Upset Models

Fig. 1 shows the schematic of a *Triple-Module-Redundancy* (*TMR*) based latch design. It can be seen that the TMR based latch consists of three identical standard latches (on the left side) and one 3-input CE (on the right side). The CE is used as a voter and its functioning will be detailed in Section 2.3. As we will see, the latch tolerates any possible SNU in hold mode (i.e., when the *system clock* (*CLK*) is low and the *negative system clock* (*NCK*) is high). It should be noted that, if a particle strikes two or more transistors that are OFF when the TMR based latch is in hold mode, not only SNUs but also DNUs and TNUs have to be considered, especially in the worst case where the striking-particle-energy is high. Let us now introduce the node-upset models used for SNUs, DNUs and TNUs.

The SNU model as shown in Fig. 2 is well-known [28]. Here, the dotted-cross-marked transistors are OFF, and the lighting marks indicate that the OFF-state transistors are respectively struck by a high-energy particle. In Fig. 2a, $N1 = 0$, thus MN1 is OFF. When a particle strikes MN1, this transistor will be ON and the value of N2 will flip from 1 to 0. In Fig. 2b, $N1 = 1$, thus MP1 is OFF. When a particle strikes MP1, this transistor will be ON and the value of N2 will flip from 0 to 1. Note that, if an ON transistor is struck by a high-energy particle, no soft error will occur.

Based on the SNU model, DNU and TNU models can be created. Fig. 3 shows the DNU model when the TMR based

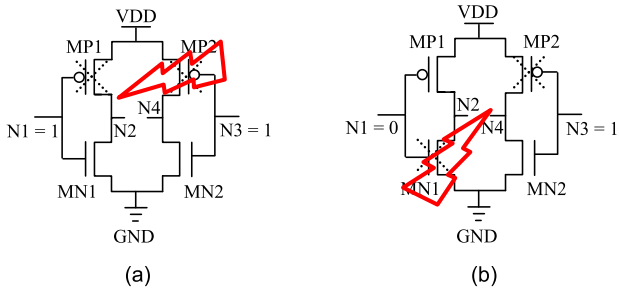


Fig. 3. DNU model. (a) A particle strikes MP2, and MP1 also collects the charge due to charge sharing. (b) A particle strikes MN1, and MP2 also collects the charge due to charge sharing.

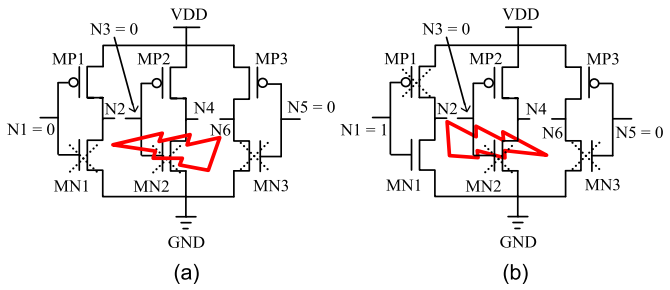


Fig. 4. TNU model. (a) A particle strikes MN2, and MN1 and MN3 also collect the charge due to charge sharing. (b) A particle strikes MN2, and MP1 and MN3 also collect the charge due to charge sharing.

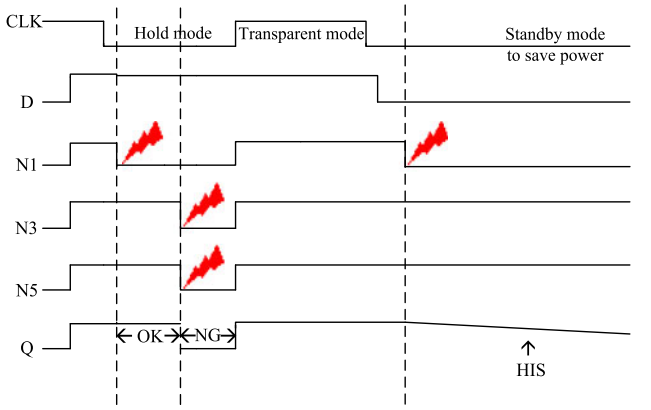


Fig. 5. Waveform diagram of i) an accumulated-SNU-and-DNU induced TNU, and ii) an SNU inducing a high impedance state, for the TMR based latch in Fig. 1.

latch is struck by a high-energy particle in hold mode. In Fig. 3a, $N1 = 1$ and $N3 = 1$, thus MP1 and MP2 are OFF. When the particle strikes MP2, due to charge sharing, MP1 may also collect the charge produced by the striking-particle, making both MP1 and MP2 to become ON. As a result, the values of N2 and N4 are flipped from 0 to 1. In Fig. 3b, $N1 = 0$ and $N3 = 1$, thus MN1 and MP2 are OFF. When the particle strikes MN1, due to charge sharing, MP2 may also collect the charge produced by the striking-particle, making both MN1 and MP2 to become ON. As a result, the value of N2 is flipped from 1 to 0 and the value of N4 is flipped from 0 to 1. Note that, similar scenarios can occur when $N3 = 0$.

The TNU model is shown in Fig. 4 when the TMR based latch is struck by a high-energy particle in hold mode. In Fig. 4a, $N1 = 0$, $N3 = 0$, and $N5 = 0$, thus MN1, MN2, and MN3 are OFF. When the particle strikes MN2, due to charge sharing, MN1 and MN3 may also collect the charge produced by the striking-particle, making MN1, MN2, and MN3 to become simultaneously ON. As a result, the values of N2, N4, and N6 are flipped from 1 to 0. In Fig. 4b, $N1 = 1$, $N3 = 0$, and $N5 = 0$, thus MP1, MN2, and MN3 are OFF. When the particle strikes MN2, due to charge sharing, MP1 and MN3 may also collect the charge produced by the striking-particle, making MP1, MN2, and MN3 to become simultaneously ON. As a result, the value of N2 is flipped from 0 to 1 and the values of N4 and N6 are flipped from 1 to 0. Note that, similar scenarios can occur when $N3 = 1$ or $N5 = 1$.

2.2 Resiliency Requirements

Fig. 5 shows the waveform diagram of an accumulated-SNU-and-DNU induced TNU, and the SNU induced *high impedance state* (*HIS*), for the TMR based latch shown in Fig. 1. It can be seen that, when N1 suffers from an SNU, it cannot self-recover from the SNU. Since the other nodes in the TMR based latch are not affected, the CE can intercept the SNU, i.e., Q is still correct as it is indicated with "OK" in Fig. 5. Subsequently, the node-pair <N3,N5> is affected by a DNU. Note that, a DNU can be represented using two simultaneous SNUs as indicated in Fig. 5. It is obvious that the SNU occurring at N1 and the DNU occurring at <N3, N5> are accumulated in the TMR based latch, leading to a TNU, since not only N1 but also <N3,N5> cannot self-recover from these soft errors. As a result, Q flips from 1 to 0 until it is refreshed by the input node D when the latch is switched into transparent mode (i.e., when CLK is high and NCK is low). This is indicated by "NG" in Fig. 5. However, if the TMR based latch can provide an SNU or DNU resiliency, the SNU and DNU will not accumulate into a TNU. Moreover, if <N1,N3,N4> is directly affected by a TNU, the TMR based latch cannot tolerate the TNU since the latch cannot provide a TNU resiliency.

It can be seen from the right part of Fig. 5 that, if the latch enters in a standby mode (i.e., CLK keeps low and NCK keeps high for a long period of time) to save power dissipation, and if N1 suffers again from an SNU, N1 will flip and will not be able to self-recover. At this time, although the other nodes of the latch are correct, Q will enter into an HIS state due to the data-retaining property of the CE (which will be described in the next sub-section). In other words, Q is floating to an undetermined value due to current-discharge as time passes. However, if the TMR based latch can provide SNU, DNU, and/or TNU resiliencies, the inputs of the CE will take the same correct value, making the CE correct output value to avoid the HIS. Therefore, node-upset resiliency is highly required for robust latch designs, since SNUs, DNUs, and even TNUs can accumulate to MNUs.

2.3 Muller C-elements

Many latch designs hardened against SNUs, DNUs, and/or TNUs have been proposed in recent years. In these designs, CEs are widely used as main components. Fig. 6 shows the schematics of widely used CEs in radiation hardened latch designs. The working principle of a CE is that it behaves as an inverter when its inputs have identical values. However, if its inputs have different values, its output will enter into an HIS, thus keeping the previous value during a certain amount of time. Note that, if this situation lasts for an extended period of time, its output will be floating into an undetermined value due to current discharge. Let us consider a 3-input CE as an example. Table 1 shows the truth table that is consistent with the above description of a CE behaviour. Moreover, the CEs are also controllable by the CLK and the NCK signals as shown in Fig. 6b and 6(d). For these *clock-gating* (*CG*) based CEs, when $CLK = 0$ and $NCK = 1$, they behave as normal CEs as those shown in Fig. 6a and 6c. When $CLK = 1$ and $NCK = 0$, they are in sleep mode, and this is used to avoid current competition on their outputs to reduce power dissipation. Some latch designs also use 4-input CEs that can be easily created according to the construction rules of the CEs shown in Fig. 6.

TABLE 1
Truth Table for 3-Input Muller C-Element

I1	I2	I3	Out
0	0	0	1
0	0	1	
0	1	0	
0	1	1	
1	0	0	**
1	0	1	
1	1	0	
1	1	1	0

** *It means that "Out" can retain the previous correct value for a certain amount of time. However, after a given period, "Out" will enter into an HIS, floating into an undetermined value due to current discharge.*

2.4 Latch-Reliability Definitions

To better classify conventional latch designs hardened against SNUs, DNUs, and/or TNUs, some definitions must be first introduced. Consider a Latch *L* with *m* single-nodes including the output-node Q (i.e., nodes $N1$, $N2$, $N3$, ..., and Nm, in which, *Nm* denotes Q). When the latch works in hold mode during one clock-cycle, the following latch-reliability definitions can be given. It should be noted that, the *m* single-nodes do not include the input-node D, since D is disconnected from the latch in hold mode. In transparent mode, node upsets in latches can be removed by D through transmission gates. Thus it is unnecessary to consider node upsets in transparent mode.

Definition 1 ("resiliency"). *Latch* L *is SNU-resilient, if and only if the following Striking-Condition 1 and Retaining-Condition 1 are true. Latch* L *is DNU-resilient, if and only if the following Striking-Condition 2 and Retaining-Condition 1 are true. Latch* L *is TNU-resilient, if and only if the following Striking-Condition 3 and Retaining-Condition 1 are true.*

Definition 2 ("immunity"). *Latch* L *is SNU-immune, if and only if the following Striking-Condition 1 and Retaining-Condition 2 are true. Latch* L *is DNU-immune, if and only if the following Striking-Condition 2 and Retaining-Condition 2 are true. Latch* L *is TNU-immune, if and only if the following Striking-Condition 3 and Retaining-Condition 2 are true.*

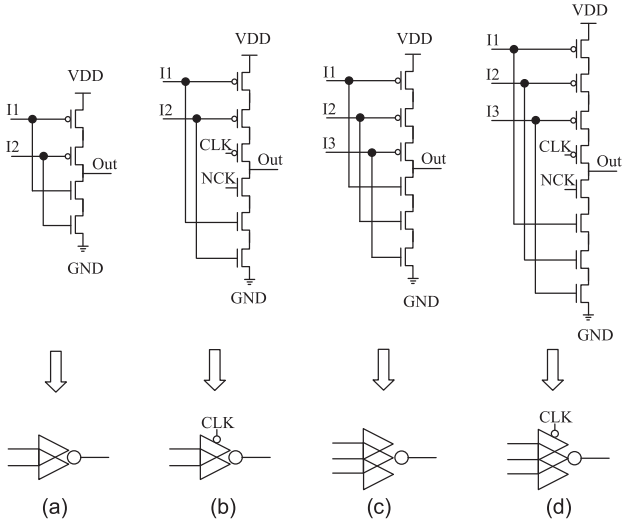


Fig. 6. Schematics of widely used Muller C-elements in radiation hardened latch designs. (a) 2-input. (b) Clock-gating based 2-input. (c) 3-input. (d) Clock-gating based 3-input.

Definition 3 ("mitigation"). *Latch* L *is SNU-mitigated, if and only if the following Striking-Condition 1 is true, and the Retaining-Condition 3 is not always true, which means that Latch* L *is only partially SNU-tolerant. Latch* L *is DNU-mitigated, if and only if the following Striking-Condition 2 is true, and the Retaining-Condition 3 is not always true, which means that Latch* L *is only partially DNU-tolerant. Latch* L *is TNU-mitigated, if and only if the following Striking-Condition 3 is true, and the Retaining-Condition 3 is not always true, which means that Latch* L *is only partially TNU-tolerant.*

Note that, the definition of "immunity" is also applicable to that of "tolerance", since we consider that the Retaining-Condition 3 should not occur if a latch is identified as SNU, DNU, and/or TNU tolerant.

Striking-Condition 1

A high-energy particle strikes any single-node $Ni\,(i = 1, 2, 3, \ldots, m)$, causing an SNU.

Striking-Condition 2

A high-energy particle strikes any node-pair $< Ni, Nj >$ $(i, j = 1, 2, 3, \ldots, m$, and $i \neq j)$, causing a DNU.

Striking-Condition 3

A high-energy particle strikes any triple-node $< Ni, Nj, Nk >$ $(i, j, k = 1, 2, 3, \ldots, m$, and $i \neq j \neq k)$, causing a TNU.

Retaining-Condition 1

Any node $Ni\,(i = 1, 2, 3, \ldots, m)$ remains or self-recovers to its previous correct value.

Retaining-Condition 2

Node *Nm* remains or self-recovers to its previous correct value; however, there is at least one internal node $Ni\,(i = 1, 2, 3, \ldots, m-1)$ that retains a flipped value.

Retaining-Condition 3

Node *Nm* retains a flipped value.

2.5 Latch Classification

According to the above reliability definitions, conventional hardened latch designs can be easily classified as follows. For SNU hardening, the *self-Recoverable, Frequency-aware and Cost-effective* (*RFC*) latch in [9], *High-performance, Low-cost, Robust and CG based* (*HLR-CG1*) and *HLR-CG2* latches in [21], and the *Interlocking Soft Error Hardened Latch* (*ISEHL*) design in [22] are SNU-resilient due to the use of specific feedback loops for keeping values. Thus, they are also SNU-immune/tolerant and SNU mitigated. The *Low Cost and Highly Reliable proposed by NAN et al.* (*LCHRNAN*) latch in [8], the design in [17], the HLR latch in [21], and the *FEedback Redundancy based SNU-Tolerant* (*FERST*) latch in [23] are SNU-immune/tolerant and SNU mitigated but not SNU-resilient. Indeed, they suffer from the HIS problem since they use a CE at the output stage for tolerating SNUs. They also suffer from the node-upset accumulation issue.

For DNU hardening, the *DOuble-Node Upset Tolerant* (*DONUT*), *Non-Temporally Hardened LaTCH* (*NTHLTCH*), *Highly Robust Double-Node Upset Tolerant* (*HRDNUT*), *Double-Node-Upset-Resilient Latch* (*DNURL*), and DeltaDICE latch designs in [12], [13], [14], [15], [24] respectively are DNU-resilient due to the multiple-module construction and/or specific feedback loops for keeping values. Thus,

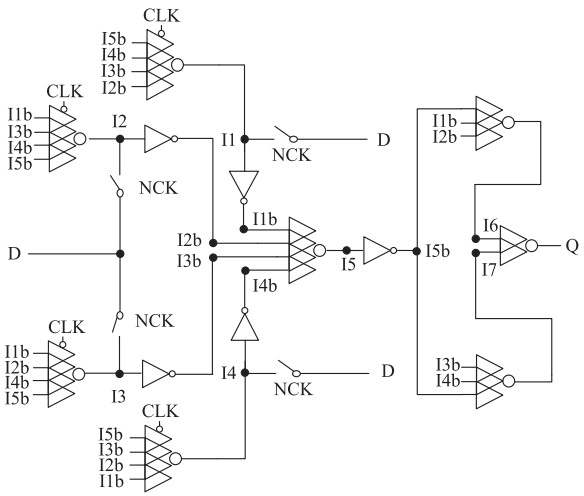


Fig. 7. Schematic of the TNUHL latch design in [16].

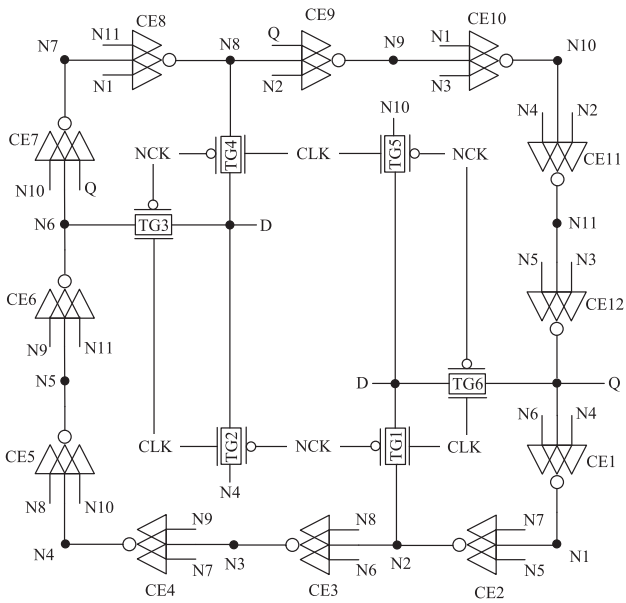


Fig. 8. Proposed LCTNURL latch design.

they are also DNU-immune/tolerant and DNU mitigated. The *Double-Node-Upset Soft-Error Interception Latch* (*DNU-SEIL*) design in [10], the *Low-cost Single Event Double-Upset Tolerant* (*LSEDUT*) latch in [11], and the *Circuit and Layout Combination Technique* (*CLCT*) latch in [19] are DNU-immune/tolerant and DNU mitigated but not DNU-resilient. Indeed, they also suffer from the HIS problem and the node-upset accumulation issue.

For TNU hardening, the *TNU Hardened Latch* (*TNUHL*) design in [16], *Low Cost and TNU completely Tolerant* (*LCTNUT*) latch in [20], and the design in [25] are TNU-immune/tolerant but not TNU-resilient. Indeed, they also suffer from the HIS problem and the node-upset accumulation issue. In addition, the TNUHL latch design shown in Fig. 7 is not cost-effective, especially in terms of transmission delay, since there are too many transistors from D to Q, making so that the latch requires much time to transmit signals from D to Q. To the best of our knowledge, there is no latch design that can provide a TNU-resiliency to avoid the HIS and node-upset accumulation issue. Therefore, we propose a novel TNU-resilient latch design in the following section.

3 Proposed TNU-Resilient Latch Design

3.1 Latch Schematic and Error-free Working Principles

The schematic of the proposed *Low Cost and TNU-self-Recoverable Latch* (*LCTNURL*) design is shown in Fig. 8. The latch only consists of six *transmission gates* (*TGs*) and twelve 3-input CEs as those shown in Fig. 6c. In Fig. 8, D and Q are the input and output nodes, CLK and NCK are the system clock and negative system clock signals, and N1 up to N11 are the internal nodes of the latch. The TGs are used for initializing the latch and the CEs are used for keeping values in the latch. Fig. 9 shows the layout of the proposed LCTNURL latch.

Let us now discuss the error-free working principles of the latch. When $\mathrm{CLK} = 1$ and $\mathrm{NCK} = 0$, the latch works in transparent mode and all of the TGs are ON. At this time, nodes N2, N4, N6, N8, N10, and Q are only driven by D through respective TGs. In other words, Q is only driven by D through one TG, and thus the transmission delay from D to Q is reduced. When $\mathrm{D} = 0$, i.e., $\mathrm{D} = \mathrm{N2} = \mathrm{N4} = \mathrm{N6} = \mathrm{N8} = \mathrm{N10} = \mathrm{Q} = 0$, the pMOS transistors in CE1, CE3, CE5, CE7, CE9, and CE11 are ON, and thus $\mathrm{N1} = \mathrm{N3} = \mathrm{N5} = \mathrm{N7} = \mathrm{N9} = \mathrm{N11} = 1$. In other words, all nodes in the latch can be correctly determined by the initialization in transparent mode. Note that, the similar scenarios can be observed when $\mathrm{D} = 1$.

When $\mathrm{CLK} = 0$ and $\mathrm{NCK} = 1$, the latch works in hold mode and all of the TGs are OFF. Hence, nodes N2, N4, N6, N8, N10, and Q cannot be driven by D through TGs and they still have the previously initialized values. At this time,

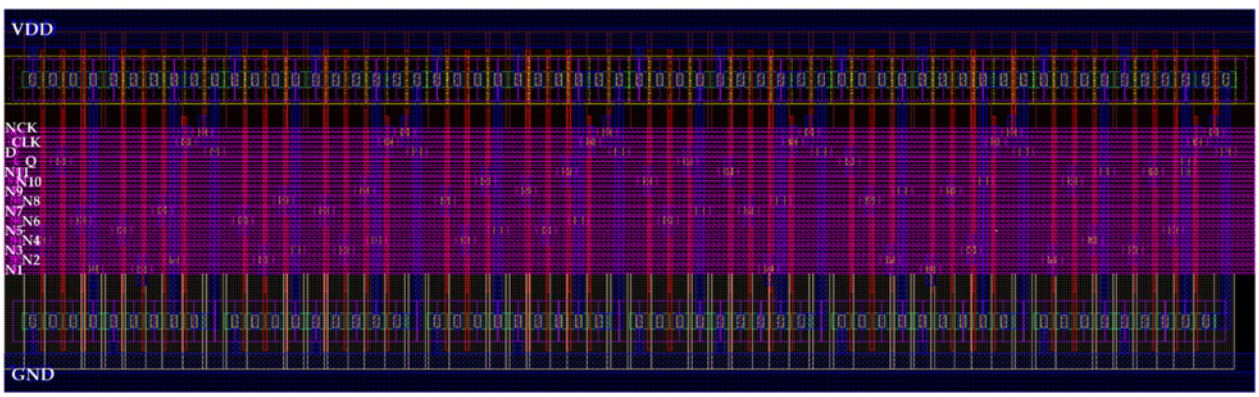


Fig. 9. Layout of the proposed LCTNURL latch design.

the values of N1, N3, N5, N7, N9, and N11 can be determined by N2, N4, N6, N8, N10, and Q through CE1, CE3, CE5, CE7, CE9, and CE11, and the values of N2, N4, N6, N8, N10, and Q can be determined by N1, N3, N5, N7, N9, and N11 through CE2, CE4, CE6, CE8, CE10, and CE12. In this way, many feedback loops are formed, making the latch able to effectively retain values.

3.2 SNU-and-DNU Resiliency Working Principles

Let us now discuss the SNU-and-DNU resiliency working principles of the latch in hold mode. Let us first discuss the SNU-resiliency of the latch. For SNUs, because of the symmetry of the latch, Q, N2, N4, N6, N8, and N10 have the same value and N1, N3, N5, N7, N9, and N11 have another same (inverted) value. Hence, we only need to consider the cases where two indicative single-nodes (e.g., Q and N1) are respectively affected by an SNU. When Q suffers from an SNU, it will be temporarily flipped. According to the feedback rule of CEs, the wrong value of Q will be fed to one of CE1's inputs, one of CE7's inputs, and one of CE9's inputs. However, the other nodes will not be affected, since the error can be intercepted by CE1, CE7, and CE9. Finally, the inputs of CE12 are still correct, and thus Q can self-recover. Note that, the similar scenarios can be observed when N1 suffers from an SNU. In summary, the latch is SNU-resilient.

Let us now discuss the DNU-resiliency of the latch. Let λ represent the node-distance between the output of one CE and the output of an adjacent CE. Since the latch is symmetrically constructed, the node-distances between the output of one CE and the output of any other CE can be λ, 2λ, 3λ, 4λ, 5λ, or 6λ. Therefore, there are only six indicative node-pairs (i.e., <Q, N1>, <Q,N2>, <Q,N3>, <Q,N4>, <Q,N5>, and <Q,N6>) for DNU consideration and all other node-pairs are equivalent to the above node-pairs. In the following, these six possible cases where each node-pair is respectively affected by a DNU will be considered. Note that, when <Q,N2>, <Q,N4>, or <Q,N6> is affected by a DNU, any CE of the latch has the critical property that if one or two of its inputs are affected, its output will not be affected; if its output is affected, its inputs will not be affected. Thus, its output still have (or can self-recover to) its previous correct value.

When <Q,N1> is affected by a DNU, it represents the case where the node-pair with node-distance λ is affected. In this case, CE12 still has the above-mentioned critical property, and thus Q can self-recover to its original correct value. At this time, all inputs of CE1 are correct, and thus N1 can restore back to its original correct value. In other words, <Q,N1> can self-recover from the DNU. It can be seen from the above analysis that the latch is DNU-resilient in this case. The analysis of the resiliency working principles for <Q,N3> and <Q,N5> is similar to that of <Q,N1>. When <Q,N2> is affected by a DNU, it represents the case where the node-pair that has a node-distance of 2λ is affected. In this case, CE12 and CE2 still have the above-mentioned critical property, and thus Q and N2 can self-recover to their original correct values. In other words, <Q, N2> can self-recover from the DNU. It can be seen from the above analysis that the latch is DNU-resilient in this case. The analysis of the resiliency working principles for <Q, N4> and <Q,N6> is similar to that of <Q,N2>.

Based on the above analysis, it can be concluded that the proposed LCTNURL latch is DNU-resilient for all indicative node-pairs.

3.3 TNU Resiliency Working Principles

Let us now discuss the TNU-resiliency of the proposed LCTNURL latch. Since the latch is symmetrically constructed, only the following six indicative cases where each triple-node is affected by a TNU need to be considered.

Case 1: The TNU simultaneously affects <Q,N1> and one of the other nodes of the latch; **Case 2**: The TNU simultaneously affects <Q,N2> and one of the other nodes of the latch except N1; **Case 3**: The TNU simultaneously affects <Q,N3> and one of the other nodes of the latch except N1 and N2; **Case 4**: The TNU simultaneously affects <Q,N4> and one of the other nodes of the latch except N1, N2, and N3; **Case 5**: The TNU simultaneously affects <Q,N5> and one of the other nodes of the latch except N1, N2, N3, and N4; **Case 6**: The TNU simultaneously affects <Q,N6> and one of the other nodes of the latch except N1, N2, N3, N4, and N5. Before analyzing the TNU-resiliency working principles, the following three important properties of the latch are introduced.

Property 1 (Resiliency). *If all inputs of a CE are correct and its output is affected by a TNU, then its output will self-recover from the TNU.*

Property 2 (Valid-retention). *If one or two inputs of a CE are affected by a TNU but its output is not affected, then it will provide the correct output value.*

Property 3 (Corruption). *If all inputs of a CE are affected by a TNU, it will provide a flipped output value.*

Property 4 (Invalid-retention). *If one or two its inputs and its output are simultaneously affected by a TNU, its output will keep the flipped value (at this time, it should be noted that, all its affected inputs should first self-recover to their previous correct values, and then it will output the previous correct value).*

In Case 1, due to the symmetry of the latch, the indicative key triple-nodes are only <Q,N1,N2>, <Q,N1,N3>, <Q,N1, N4>, <Q,N1,N5>, and <Q,N1,N6>. When <Q,N1,N2> is affected by a TNU, CE1 and CE2 satisfy Property 4, CE3, CE7, CE8, CE9, CE10, and CE11 satisfy Property 2, and CE4, CE5, CE6, and CE12 satisfy Property 1. Hence, the CEs except CE1 and CE2 can still output the correct values. It is obvious that, CE3, CE5, and CE11 can still output the correct values, i.e., the inputs of CE12 are still correct, and thus Q can first self-recover to its original correct value. Hence, the inputs of CE1 are still correct, and N1 can self-recover to its original correct value. Similarly, N2 can self-recover to its original correct value. In other words, <Q,N1,N2> can self-recover from the TNU.

When <Q,N1,N3> is affected by a TNU, CE1 and CE12 satisfy Property 4, CE2, CE4, CE7, CE8, CE9, and CE10 satisfy Property 2, and CE3 satisfy Property 1. Hence, the CEs except CE1 and CE12 can still output the correct values. It is obvious that, CE2, CE6, and CE8 can still output the correct values, i.e., the inputs of CE3 are still correct, and thus N3 can first self-recover to its original correct value. Hence, the inputs of CE12 are still correct, and thus Q can self-recover

to its original correct value. Similarly, N1 can self-recover to its original correct value. In other words, <Q,N1,N3> can self-recover from the TNU.

When <Q,N1,N4> is affected by a TNU, CE1 satisfies Property 4, CE2, CE5, CE7, CE8, CE9, CE10, and CE11 satisfy Property 2, and CE4 and CE12 satisfy Property 1. Hence, the CEs except CE1 can still output the correct values. It is obvious that, CE3, CE5, CE7, CE9, and CE11 can still output the correct values, i.e., the inputs of CE12 and CE4 are still correct, and thus Q and N4 can first self-recover to their original correct values. Hence, the inputs of CE1 are still correct, and N1 can self-recover to its original correct value. In other words, <Q,N1,N4> can self-recover from the TNU.

When <Q,N1,N5> is affected by a TNU, CE1 and CE12 satisfy Property 4, CE2, CE6, CE7, CE8, CE9, and CE10 satisfy Property 2, and CE5 satisfy Property 1. Hence, the CEs except CE1 and CE12 can still output the correct values. It is obvious that, CE4, CE8, and CE10 can still output the correct values, i.e., the inputs of CE5 are still correct, and thus N5 can first self-recover to its original correct value. Hence, the inputs of CE12 are still correct, and thus Q can self-recover to its original correct value. Similarly, the inputs of CE1 are still correct, and N1 can self-recover to its original correct value. In other words, <Q,N1,N5> can self-recover from the TNU.

When <Q,N1,N6> is affected by a TNU, CE1 satisfies Property 4, CE2, CE3, CE7, CE8, CE9, and CE10 satisfy Property 2, and CE6 and CE12 satisfy Property 1. Hence, the CEs except CE1 can still output the correct values. It is obvious that, CE3, CE5, CE9, and CE11 will output the correct values, i.e., the inputs of CE12 and CE6 are still correct, and thus Q and N6 can first self-recover to their original correct values. Hence, the inputs of CE1 are still correct, and N1 can self-recover to its original correct value. In other words, <Q,N1, N6> can self-recover from the TNU. It can be seen from the above analysis that the latch is TNU-resilient in Case 1.

Due to the symmetry of the latch, in Case 2, the indicative key triple-nodes are only <Q,N2,N3>, <Q,N2,N4>, <Q,N2, N5>, <Q,N2,N6>, and <Q,N2,N7>; in Case 3, the indicative key triple-nodes are only <Q,N3,N4>, <Q,N3,N5>, <Q,N3, N6>, and <Q,N3,N7>; in Case 4, the indicative key triple-nodes are only <Q,N4,N5>, <Q,N4,N6>, <Q,N4,N7>, and <Q,N4,N8>; in Case 5, the indicative key triple-nodes are only <Q,N5,N6>, <Q,N5,N7>, and <Q,N5,N8>; and in Case 6, the indicative key triple-nodes are only <Q,N6,N7>, <Q,N6,N8>, and <Q,N6,N9>.

In Cases 2 to 6, when these indicative key triple-nodes except <Q,N4,N6> are affected by a TNU, the analysis of the resiliency working principles is similar to that of the indicative key triple-nodes in Case 1. However, for the special indicative key triple-node <Q,N4,N6>, all inputs of CE1 are affected by a TNU. At this time, CE1 satisfies Property 3, CE3, CE5, CE7, CE9, and CE11 satisfy Property 2, and CE4, CE6, and CE12 satisfy Property 1. Hence, the CEs except CE1 can still output the correct values. It is obvious that, CE12, CE4, and CE6 can still output the correct values, i.e., Q, N4, and N6 can self-recover to their original correct values. In other words, <Q,N4,N6> can self-recover from the TNU.

As discussed above, the LCTNURL latch is TNU-resilient for all the triple-nodes in the six indicative cases. For all the other triple-nodes, since the latch is symmetrically constructed, they are equivalent to that of the above six indicative cases, i.e., the latch can self-recover from any possible TNU. Therefore, the proposed LCTNURL latch can provide a complete resiliency for TNUs in addition to SNUs and DNUs resiliencies.

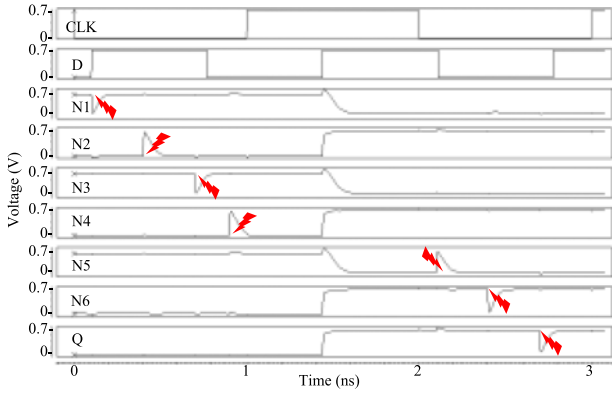


Fig. 10. Simulation results for the complete key SNU injections of the LCTNURL latch design.

3.4 Simulation Results

The LCTNURL latch was implemented in a 16 nm CMOS manufacturing technology and extensive simulations using Synopsys HSPICE were performed. The simulation parameters were the following: the supply voltage was set to 0.7V, the working temperature was set to the room temperature, the PMOS transistors had the ratio $\mathrm{W/L} = 90/16\mathrm{nm}$, and the NMOS transistors had the ratio $\mathrm{W/L} = 22/16\mathrm{nm}$. Note that, the lighting marks in Figs. 10, 11, and 12 denote the injected errors.

Fig. 10 shows simulation results for the complete key SNU injections of the LCTNURL latch. As discussed in Section 3.2, we only need to consider the cases where two indicative single-nodes (e.g., Q and N1) are respectively affected by an SNU. However, we selected more nodes (i.e., Q, N1, N2, N3, N4, N5, and N6) for sufficient SNU injections. As shown in Fig. 10, between 0ns and 1ns (i.e., when $\mathrm{Q} = 0$), SNUs were respectively injected on nodes N1, N2, N3, and N4; between 2ns and 3ns (i.e., when $\mathrm{Q} = 1$), SNUs were respectively injected on nodes N5, N6, and Q. It can be seen from Fig. 10 that the SNU-affected single-nodes can self-recover to correct

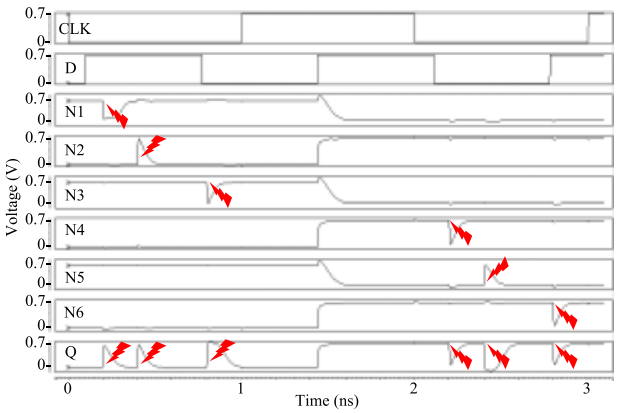


Fig. 11. Simulation results for the complete key DNU injections of the LCTNURL latch design.

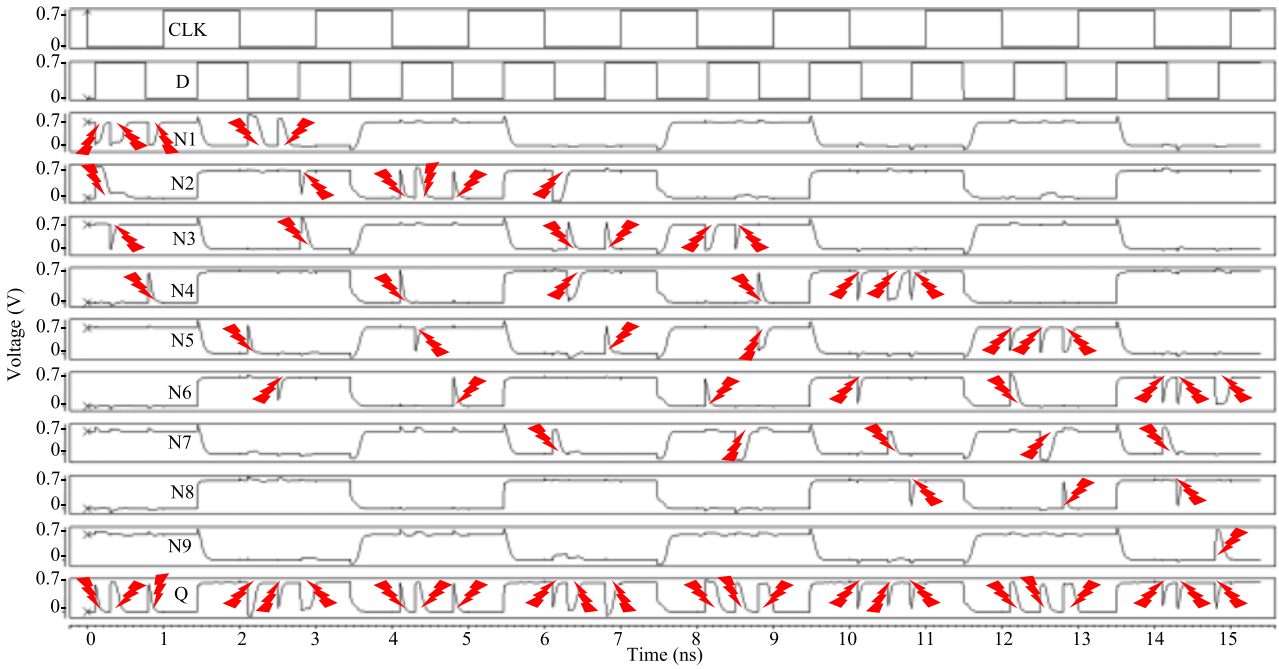


Fig. 12. Simulation results for the complete key TNU injections of the LCTNURL latch design.

values. Note that, for the reverse states of the above nodes, SNU-injections were also performed. As predicted, the above nodes can also self-recover from the injected SNUs. However, for the sake of brevity, details are omitted in Fig. 10.

Fig. 11 shows simulation results for the complete key DNU injections of the LCTNURL latch. As discussed in Section 3.2, we only need to consider the cases where six indicative node-pairs (i.e., $<Q, N1>, <Q, N2>, <Q, N3>, <Q, N4>, <Q, N5>$, and $<Q, N6>$) are respectively affected by a DNU. As shown in Fig. 11, between 0ns and 1ns (i.e., when $Q = 0$), DNUs were respectively injected on node-pairs $<Q, N1>, <Q, N2>$, and $<Q, N3>$; between 2ns and 3ns (i.e., when $Q = 1$), DNUs were respectively injected on node-pairs $<Q, N4>, <Q, N5>$, and $<Q, N6>$. It can be seen from Fig. 11 that the DNU-affected double-nodes can self-recover to correct values. Note that, for the reverse states of the above node-pairs, DNU-injections were also performed. As predicted, the node pairs can also self-recover from the injected DNUs. However, for the sake of brevity, details are omitted in Fig. 11.

In the same way, TNU injections were also performed. Fig. 12 shows simulation results for the complete key TNU injections of the LCTNURL latch. Table 2 shows statistical results of TNU injections according to Fig. 12. In Table 2, "TNUs" denotes the injected TNUs on the key triple-nodes, "Time" denotes the injection time of the TNUs, and "State" denotes the correct state of Q. It can be seen that the TNU-affected nodes can self-recover to correct values. Note that, for the reverse states of the key triple-nodes, TNU-injections were also performed. As predicted, the triple-nodes can also self-recover from the injected TNUs. However, for the sake of brevity, details are omitted in Fig. 12. In summary, simulation results validate the fact that the LCTNURL latch can achieve complete SNU, DNU and TNU resiliencies.

Note that, in all the above SNU/DNU/TNU injection simulations, a controllable double exponential current source model was used as in [9], [15], [20]. The time constant of the rise and fall of the current pulse was set to be 0.1ps and 3ps, respectively. The worst case deposited charge was chosen to be up to 45fC for a single node, which was large enough since the purpose was to validate the circuit operation under extreme SNU/DNU/TNU conditions that might disturb circuit nodes.

4 Evaluation and Comparison

To make a fair comparison, the proposed LCTNURL latch and existing representative hardened latches, such as the LCHRNAN [8], RFC [9], DNUSEIL [10], LSEDUT [11], DONUT [12], NTHLTCH [13], HRDNUT [14], DNURL [15], and TNUHL [16], were designed using the same 16nm CMOS technology. For these latches, the PMOS transistors had the ratio $W/L = 90/16nm$, and the NMOS transistors had the ratio $W/L = 22/16nm$. These parameters are the same than those of the transistors in the proposed LCTNURL latch. The supply voltage was still set to 0.7V and the working temperature was the room temperature.

TABLE 2
Statistic Results for the Complete Key TNU Injections of the LCTNURL Latch Design According to Fig. 12

Time (ns)	TNUs	State	Time (ns)	TNUs	State
0.1	Q, N1, N2	$Q = 0$	8.1	Q, N3, N6	$Q = 0$
0.3	Q, N1, N3	$Q = 0$	8.5	Q, N3, N7	$Q = 0$
0.8	Q, N1, N4	$Q = 0$	8.8	Q, N4, N5	$Q = 0$
2.1	Q, N1, N5	$Q = 1$	10.1	Q, N4, N6	$Q = 1$
2.5	Q, N1, N6	$Q = 1$	10.5	Q, N4, N7	$Q = 1$
2.8	Q, N2, N3	$Q = 1$	10.8	Q, N4, N8	$Q = 1$
4.1	Q, N2, N4	$Q = 0$	12.1	Q, N5, N6	$Q = 0$
4.3	Q, N2, N5	$Q = 0$	12.5	Q, N5, N7	$Q = 0$
4.8	Q, N2, N6	$Q = 0$	12.8	Q, N5, N8	$Q = 0$
6.1	Q, N2, N7	$Q = 1$	14.1	Q, N6, N7	$Q = 1$
6.3	Q, N3, N4	$Q = 1$	14.3	Q, N6, N8	$Q = 1$
6.8	Q, N3, N5	$Q = 1$	14.8	Q, N6, N9	$Q = 1$

TABLE 3
Reliability Comparisons Among the SNU, DNU, and/or TNU Hardened Latch Designs

Latch	SNU Tolerant	SNU Resilient	DNU Tolerant	DNU Resilient	TNU Tolerant	TNU Resilient
LCHRNAN	√	×	×	×	×	×
RFC	√	√	×	×	×	×
DNUSEIL	√	√	√	×	×	×
LSEDUT	√	√	√	×	×	×
DONUT	√	√	√	√	×	×
NTHLTCH	√	√	√	√	×	×
HRDNUT	√	√	√	√	×	×
DNURL	√	√	√	√	×	×
TNUHL	√	√	√	×	√	×
LCTNURL	√	√	√	√	√	√

TABLE 4
Overhead Comparisons Among the SNU, DNU, and/or TNU Hardened Latch Designs

Latch	Ref.	Delay (ps)	Power (μW)	$10^{-4}\times$ Area (nm^2)	$10^{-5}\times$ DPAP
LCHRNAN	[8]	10.28	0.27	2.51	0.70
RFC	[9]	2.16	0.16	2.15	0.07
DNUSEIL	[10]	46.90	0.54	4.48	11.3
LSEDUT	[11]	1.24	0.30	3.69	0.14
DONUT	[12]	14.81	0.71	2.87	3.02
NTHLTCH	[13]	9.85	0.57	5.20	2.92
HRDNUT	[14]	3.86	0.27	3.94	0.41
DNURL	[15]	2.18	0.56	6.99	0.85
TNUHL	[16]	81.04	0.66	7.35	39.31
LCTNURL	Proposed	5.19	0.83	7.53	3.24

Table 3 shows the reliability comparisons among the SNU, DNU, and/or TNU hardened latch designs. It can be seen from Table 3 that the LCHRNAN and RFC latches can only tolerate SNUs, and the RFC latch can provide an SNU resiliency. The DNUSEIL and LSEDUT latches are not only SNU-resilient but also DNU-tolerant. The DONUT, NTHLTCH, HRDNUT, and DNURL latches can provide SNU and DNU tolerance and resiliency. However, all the above latches cannot provide a TNU-resiliency. The TNUHL latch can provide a TNU-tolerance. However, the latch cannot provide DNU and TNU resiliencies. It can be seen from Table 3 that, only the proposed LCTNURL latch can simultaneously provide SNU, DNU, and TNU tolerance and resiliency meaning that the proposed LCTNURL latch can provide the highest reliability against SNUs, DNUs, and TNUs, irrespective of the working environment.

Table 4 shows the overhead comparison results among the SNU, DNU, and/or TNU hardened latch designs. In Table 4, "Delay" denotes D to Q transmission delay, i.e., the average of rise and fall delays from D to Q, "Power" denotes the average power dissipation (dynamic and static), and "Area" denotes the silicon area measured with Eq. (1). In Eq. (1), $n1$ is the number of nMOS transistors, *L*nMOS(*i*) and *W*nMOS(*i*) are the effective length and width of each nMOS transistor, respectively. Similarly, $n2$ is the number of pMOS transistors, *L*pMOS(*i*) and *W*pMOS(*i*) are the effective length and width of each pMOS transistor, respectively. "DPAP" denotes the delay-power-area product calculated with Eq. (2).

$$\mathrm{Area} = \sum_{i=1}^{n1} L_{\mathrm{nMOS}}(i) * W_{\mathrm{nMOS}}(i) + \sum_{i=1}^{n2} L_{\mathrm{pMOS}}(i) * W_{\mathrm{pMOS}}(i) \tag{1}$$

$$DPAP = Delay \times Power \times Area. \tag{2}$$

It can be seen from Table 4 that for the proposed LCTNURL latch and some existing hardened latches such as the RFC, LSEDUT, HRDNUT, and DNURL, their delay is small due to the use of a high-speed path from D to Q. Conversely, the DNUSEIL and TNUHL latches have a larger delay since there are too many transistors from D to Q.

The LCHRNAN, RFC, LSEDUT, and HRDNUT latches have a very low power consumption due to their smaller silicon area and the use of the clock-gating technology for some of them. The DNUSEIL, DONUT, NTHLTCH, and DNURL latches consume more power, and the main reason is that they use extra silicon area to achieve an any-possible-DNU tolerance or resiliency. Similarly, since the TNUHL latch uses extra silicon area to ensure TNU tolerance, it consumes more power. For the same reason, to self-recover from any TNU, the proposed LCTNURL latch also consumes more silicon area, hence leading to a large power dissipation.

Regarding silicon area overhead, compared with the LCHRNAN, RFC, and DONUT latches, the DNUSEIL, NTHLTCH, DNURL, TNUHL and proposed LCTNURL latches require more area since they use redundant transistors to achieve very high reliability.

Regarding DPAP, the DNUSEIL and TNUHL latches have the highest DPAP values, mainly because of their large delay. Conversely, the LCHRNAN, RFC, LSEDUT, HRDNUT and DNURL latches have a lower DPAP since their delay, power, and/or silicon area are smaller. In summary, compared with the only TNU-tolerant latch (TNUHL), the proposed LCTNURL latch is cost effective and achieves a TNU-resiliency at the cost of a slightly silicon area overhead.

We further calculated and analyzed the *percentage of improvement cost* (*PIC*) of the LCTNURL latch compared with the other latches based on the values in Table 4. The PIC of the delay was calculated with Eq. (3), where Delay-proposed is the delay of the proposed LCTNURL latch, and Delaycompared is the delay of the latch compared with the proposed LCTNURL latch. Similarly, the PICs of the power dissipation, silicon area, and DPAP can also be calculated. Positive PICs indicate that the LCTNURL latch consumes less delay, power dissipation, silicon area, and DPAP than the other latches. Negative PICs indicate that the LCTNURL latch has a higher delay, power dissipation, silicon area, and DPAP than the other latches.

$$\mathrm{PICdelay} = \frac{\mathrm{Delaycompared} - \mathrm{Delayproposed}}{\mathrm{Delaycompared}} \times 100\%. \tag{3}$$

Fig. 13 shows the calculated PICs, i.e., the percentages of the improved costs for the proposed LCTNURL latch compared with the other latches. It can be seen from Fig. 13 that, for the delay, compared with the nine target latches, i.e., LCHRNAN, RFC, DNUSEIL, LSEDUT, DONUT, NTHLTCH, HRDNUT, DNURL, and TNUHL, the delay ratio of the proposed LCTNURL latch is 49.51 percent, −140.28 percent, 88.93 percent, −318.55 percent, 64.96 percent, 47.31 percent, −34.46 percent, −138.07 percent, and 93.60 percent,

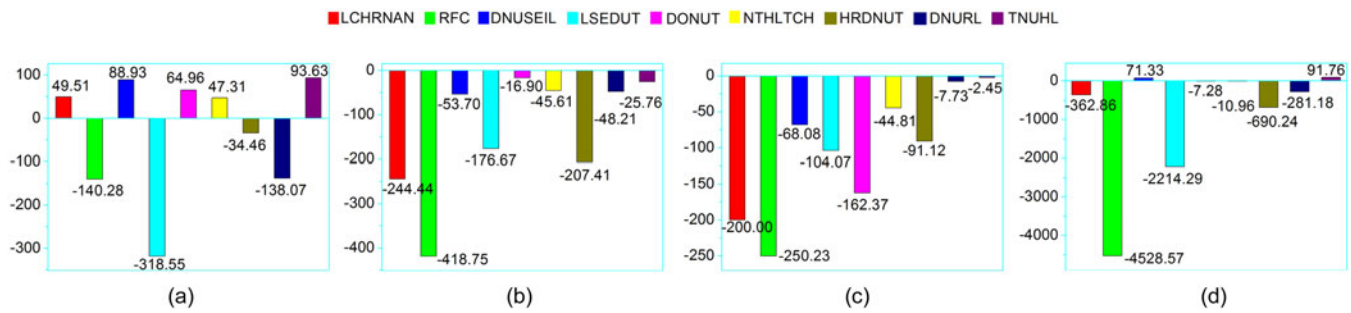


Fig. 13. Percentages (%) of improved costs for the proposed LCTNURL latch. (a) Delay. (b) Power Dissipation. (c) Silicon Area. (d) DPAP.

respectively. There are five positive PICs meaning that the delay of the proposed LCTNURL latch is smaller than that of the five compared latches. Meanwhile, it can be seen from Fig. 13 that, compared with the TNU-tolerant TNUHL latch that cannot provide a TNU-resiliency, the proposed LCTNURL latch can improve the delay by 93.60 percent.

Regarding power dissipation, it can be seen from Fig. 13 that, compared with the nine target latches, the power dissipation ratio of the proposed LCTNURL latch is −207.41 percent, −418.75 percent, −53.70 percent, −176.67 percent, −16.90 percent, −45.61 percent, −207.41 percent, −48.21 percent, and −25.76 percent, respectively. There are no positive PICs meaning that the first TNU-resiliency achievement obtained by the proposed LCTNURL latch is at the cost of power dissipation overhead.

Regarding silicon area, it can be seen from Fig. 13 that, compared with the nine target latches, the silicon area ratio of the proposed LCTNURL latch is −200.00 percent, −250.23 percent, −68.08 percent, −104.07 percent, −162.37 percent, −44.81 percent, −91.12 percent, −7.73 percent, and −2.45 percent, respectively. There is no positive PICs meaning that the first TNU-resiliency achievement obtained by the proposed LCTNURL latch is also done at the cost of a non-negligible silicon area. However, it can be seen from Fig. 13 that, compared with the TNU-tolerant TNUHL latch that cannot provide a TNU-resiliency, the proposed LCTNURL latch only consumes an extra 2.45 percent silicon area.

Regarding DPAP, it can be seen from Fig. 13 that, compared with the nine target latches, the DPAP ratio of the proposed LCTNURL latch is −362.86 percent, −4528.57 percent, 71.33 percent, −2214.29 percent, −7.28 percent, −10.96 percent, −690.24 percent, −271.18 percent, and 91.76 percent, respectively. There are two positive PICs meaning that the DPAP of the proposed LCTNURL latch is smaller than that of the DNU-SEIL and DNUHL latches. However, it can be seen that, compared with the TNU-tolerant TNUHL latch, the proposed LCTNURL latch can improve DPAP up to 91.76 percent. In summary, the above PICs and analysis can validate the cost effectiveness of the proposed LCTNURL latch in terms of delay and comprehensive DPAP, especially when compared with the state-of-the-art TNU-tolerant TNUHL latch that cannot provide TNU-resiliency.

5 Conclusion

In nano-scale CMOS technologies, aggressive shrinking of transistor feature sizes in integrated circuits and systems may lead to the occurrence of multiple-node-upset including *double-node-upset* (*DNU*) and *triple-node-upset* (*TNU*). For the first time, this paper has proposed a novel *Low Cost and TNU-self-Recoverable Latch* (*LCTNURL*) design for dependable computing. The latch mainly consists of twelve 3-input *Muller C-elements* (*CEs*), constructing a circularly linked schematic. For any 3-input CE in the proposed latch, its output triply feeds back to one input of some specified downstream CEs, making the latch completely resilient from any possible SNU, DNU, and TNU. Simulation results have demonstrated the complete SNU, DNU, and TNU-resiliencies of the proposed latch. Moreover, using a high-speed transmission path and fewer transistors, the proposed latch has not only a low delay but also a lower delay-power-area product than the state-of-the-art *TNU hardened latch* (*TNUHL*) that cannot achieve a complete TNU-resiliency.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grants 61974001, 61874156, 61674048, 61604001, 61872001, and 61572031, 61904001, 61834006, Anhui University Doctor Startup Fund (Y040435009), and China Scholarship Council. This research was also supported in part by JSPS Grant-in-Aid for Scientific Research (B) 17H01716.

References

[1] K. Yamada, H. Maruoka, J. Furuta, and K. Kobayashi, "Radiation-hardened flip-flops with low-delay overhead using pMOS pass-transistors to suppress SET pulses in a 65-nm FDSOI process," *IEEE Trans. Nucl. Sci.*, vol. 65, no. 8, pp. 1814–1822, Aug. 2018

[2] M. J. Gadlage, A. H. Roach, A. R. Duncan, A. M. Williams, D. P. Bossev, and M. J. Kay, "Soft errors induced by high-energy electrons," *IEEE Trans. Device Mater. Rel.*, vol. 17, no. 1, pp. 157–162, Mar. 2017

[3] I. A. Danilov *et al.*, "On board electronic devices safety provided by DICE-based Muller C-elements," *Acta Astronautica*, vol. 150, pp. 28–32, 2018

[4] C. Peng *et al.*, "Radiation-hardened 14T SRAM bitcell with speed and power optimized for space application," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 2, pp. 407–415, Feb. 2019

[5] M. N. Sakib, R. Hassan, S. N. Biswas, and S. R. Das, "Memristor-based high-speed memory cell with stable successive read operation," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 37, no. 5, pp. 1037–1049, May 2018

[6] C. Peng *et al.*, "A radiation harden enhanced quatro (RHEQ) SRAM cell," *IEICE Electronics Express*, vol. 14, no. 18, pp. 1–12, 2017

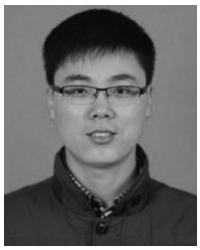
[7] Y. Li *et al.*, "Flip-flop SEUs mitigation through partial hardening of internal latch and adjustment of clock duty cycle," in *Proc. IEEE 21st Int. Symp. Des. Diagnostics Electron. Circuits Syst.*, 2018, pp. 1–6

[8] H. Nan and K. Choi, "Low cost and highly reliable hardened latch design for nanoscale CMOS technology," *Microelectronics Rel.*, vol. 52, no. 6, pp. 1209–1214, 2012

[9] A. Yan *et al.*, "A self-recoverable, frequency-aware and cost-effective robust latch design for nanoscale CMOS technology," *IEICE Trans. Electronics*, vol. 98, no. 12, pp. 1171–1178, 2015
[10] K. Katsarou and Y. Tsiatouhas, "Soft error interception latch: Double node charge sharing SEU tolerant design," *Electronics Lett.*, vol. 51, no. 4, pp. 330–332, 2015
[11] J. Jiang *et al.*, "Low-cost single event double-upset tolerant latch design," *Electronics Lett.*, vol. 54, no. 9, pp. 554–556, 2018.
[12] N. Eftaxiopoulos, N. Axelos, and K. Pekmestzi, "DONUT: A double node upset tolerant latch," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, 2015, pp. 1–6.
[13] Y. Li *et al.*, "Double node upsets hardened latch circuits," *J. Electron. Testing*, vol. 31, pp. 537–548, 2015.
[14] A. Watkins and S. Tragouodas, "A highly robust double node upset tolerant latch," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst.*, 2016, pp. 1–6.
[15] A. Yan, Z. Huang, M. Yi, X. Xu, Y. Ouyang, and H. Liang, "Double-node-upset-resilient latch design for nanoscale CMOS technology," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 6, pp. 1978–1982, Jun. 2017
[16] A. Watkins and S. Tragoudas, "Radiation hardened latch designs for double and triple node upsets," *IEEE Trans. Emerg. Topics Comput.*, to be published, doi: 10.1109/TETC.2017.2776285.
[17] Z. Huang, H. Liang, and S. Hellebrand, "A high performance SEU tolerant latch," *J. Electron. Testing*, vol. 31, no. 4, pp. 349–359, 2015
[18] S. Lin, Y. Kim, and F. Lombardi, "Design and performance evaluation of radiation hardened latches for nanoscale CMOS," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 19, no. 7, pp. 1315–1319, Jul. 2011
[19] X. Hui and Z. Yun, "Circuit and layout combination technique to enhance multiple nodes upset tolerance in latches," *IEICE Electronics Express*, vol. 12, no. 9, pp. 1–7, 2015
[20] A. Yan *et al.*, "Novel low cost, double-and-triple-node-upset-tolerant latch designs for nano-scale CMOS," *IEEE Trans. Emerg. Topics Comput.*, 2018, to be published, doi: 10.1109/TETC.2018.2871861.
[21] H. Nan and K. Choi, "High performance, low cost, and robust soft error tolerant latch designs for nanoscale CMOS technology," *IEEE Trans. Circuits Syst. I*, vol. 59, no. 7, pp. 1445–1457, 2012.
[22] H. Liang *et al.*, "Design of a radiation hardened latch for low-power circuits," in *Proc. IEEE 23rd Asian Test Symp.*, 2014, pp. 1–6.
[23] M. Fazeli *et al.*, "Feedback redundancy: A power efficient SEU-tolerant latch design for deep sub-micron technologies," in *Proc. 37th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, 2007, pp. 1–10.
[24] N. Eftaxiopoulos *et al.*, "Delta DICE: A double node upset resilient latch," in *Proc. IEEE Int. Midwest Symp. Circuits Syst.*, 2015, pp. 1–4.
[25] D. Lin *et al.*, "A novel self-recoverable and triple nodes upset resilience DICE latch," *IEICE Electronics Express*, vol. 15, no. 19, pp. 1–10, 2018
[26] H. Wang *et al.*, "A layout-based rad-hard DICE flip-flop design," *J. Electron. Testing*, vol. 35, pp. 111–117, 2019
[27] J. Jiang, W. Zhu, J. Xiao, and S. Zou, "A novel high-performance low-cost double-upset tolerant latch design," *Electronics*, vol. 2, no. 10, pp. 1–8, 2018
[28] M. Omana, D. Rossi, and C. Metra, "Novel transient fault hardened static latch," in *Proc. Int. Test Conf.*, 2003, pp. 886–892.

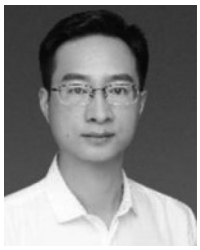
Aibin Yan received the MS degree from the University of Science and Technology of China, Hefei, in 2009, and the PhD degree from the Hefei University of Technology, in 2015. His research interests include soft error rate analysis, and radiation hardening by design for nano-scale ICs.

Yuanjie Hu received the BS degree from Chuzhou Colleague, in 2018. Currently, she is working toward the MS degree in software engineering major in Anhui University. Her research interest includes radiation hardening by design for nano-scale ICs.



Jie Cui received the PhD degree from the University of Science and Technology of China, Hefei, in 2012. Currently, he is a professor with Anhui University, Hefei, Anhui, China. His research interests include IoT security, applied cryptography, software-defined networking, vehicular ad hoc network, and design for fault tolerance.

Zhili Chen received the BS and PhD degrees from the University of Science and Technology of China, Hefei, in 2004 and 2009, respectively. Currently he is a professor with Anhui University. His research interests include privacy-preserving cloud auction, secure spectrum auction, and design for fault tolerance.



Zhengfeng Huang received the PhD degree from the Hefei University of Technology, in 2009. Currently he is a professor since 2018. His research interests include design for soft error tolerance/mitigation. He is a member of Technical Committee on Fault Tolerant Computing which belongs to China Computer Federation. He worked as a visiting scholar at the University of Paderborn, Germany from 2014 to 2015.

Tianming Ni received the PhD degree from the Hefei University of Technology, Hefei, China, in 2018. He joined the College of Electrical Engineering, Anhui Polytechnic University, in 2018. His research interest includes built-in-self-test, design automation of digital systems, design for IC reliability, 3D IC test and fault tolerance.

Patrick Girard received the MSc degree in electrical engineering, and the PhD degree in microelectronics from the University of Montpellier, France, in 1988 and 1992, respectively. He is currently a research director with CNRS (French National Center for Scientific Research) and works in the Laboratory of Informatics, Robotics and Microelectronics of Montpellier (LIRMM) – France. His research interests include all aspects of digital testing and memory testing, with emphasis on critical constraints such as timing and power. Reliability and fault tolerance are also part of his research activities. He is a fellow of the IEEE.

Xiaoqing Wen (fellow member IEEE) received the BE degree from Tsinghua University, China, in 1986, the ME degree from Hiroshima University, Japan, in 1990, and the PhD degree from Osaka University, Japan, in 1993. In 2004, he joined Kyushu the Institute of Technology, Japan, where he is currently a professor. He co-authored and co-edited two books about VLSI Test. He holds 43 US Patents and 14 Japan Patents on VLSI testing. His research interests include VLSI test, diagnosis, and testable design. He is serving as an associate editors for journals such as *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* and *Journal of Electronic Testing: Theory and Applications*

Prune and Plant: Efficient Placement and Parallelism of Virtual Network Functions

Wei Bao, *Member, IEEE*, Dong Yuan, *Member, IEEE*,
Bing Bing Zhou, and Albert Y. Zomaya, *Fellow, IEEE*

Abstract—Network function virtualization (NFV) is a promising solution to realize a variety of network services. By definition, virtual network functions (VNFs) are chained together to realize different services. However, chaining is not an ideal solution as service latency grows linearly with respect to the length of the chain. Motivated by the fact that many VNFs can be parallelized, we investigate parallelism of VNFs for acceleration. The dependency of the VNFs is characterized by a directed acyclic graph (DAG). We aim to deploy the VNFs in the right place and process them in parallel without violating the DAG, to minimize the overall delay. However, directly solving the delay minimization problem is NP-hard, and it may also introduce a large number of duplicated packets to burden the system. To deal with these issues, we propose the Prune and Plant (P&P) scheme with polynomial computational complexity, to reduce the overall delay while limiting the number of duplicated packets. P&P comprises two stages: in the Prune stage, we prune the original DAG into a series-parallel graph (SP-graph), which eliminates NP-hardness while maintaining parallelism of VNFs. In the Plant stage, we find the optimal placement for the VNFs with respect to the SP-graph. By both simulation and prototyping, we demonstrate that P&P significantly outperforms benchmark schemes.

Index Terms—Network function virtualization, network function placement, delay minimization, parallelism

1 Introduction

Network function virtualization (NFV) is a promising solution to realize a new generation of network services. Different from traditional solutions where network functions (NFs) are realized by dedicated network hardware, NFs are now implemented as software components, to be cost-effectively executed in commodity servers. Monetary costs of purchasing and upgrading expensive network hardware can be greatly reduced. In addition, network dynamics can be more easily handled as NFs can be rapidly created, deconstructed, or migrated with great flexibility and scalability.

In order to realize a specific network service, service providers usually let network traffic pass through several NFs in a particular order, forming a network service function chain (SFC) [1]. However, NFs might be deployed in virtual machines/containers hosted by different physical servers, so that a service chain has to traverse through these servers along a path.[1] However, chaining is not an ideal solution [2] as the latency of service chain grows linearly with respect to the length of the chain, which could be large if a number of NFs are involved–it could be unacceptable for applications with tight latency requirements, such as stock market exchanges, online shopping, and on-live video streaming.

The key drawback of the service function chain solution is that it ignores the possibility that NFs with no dependency can be processed in parallel, especially for those NFs only require "read" operations, such as monitor, caching, and inspection. Recent investigation shows that 53.8 percent of NFs are parallelizable [2]. If these parallelizable NFs have to be placed far away from each other, unnecessary yet significant delays are generated if they are processed one by one.

The paper [2] is the pioneer work that proposes and implements NFs in parallel. It shows parallelism achieves substantial delay reduction when the NFs are placed close to each other (e.g., in a same cluster). We envision that NF parallelism has even stronger potential to reduce latency for NFs deployed in a large geographically distributed region. In the example in Fig. 1, if we do not allow parallelism, it takes 30 ms at least to complete the NFs (VPN, caching, monitor, and load balancing). However, since monitor and caching are parallelizable, the overall delay is reduced to 20 ms if they are executed in parallel.

Even though NF parallelism has potential to the reduce delay, we observe that such potential can be fully realized when the NFs are wisely placed at the correct locations. In the same example of Fig. 1, "caching2" is an alternative service node that supports caching (i.e., another place to provide caching service). If we place caching NF there, however, the overall delay is increased to 40 ms, even larger than that without NF parallelism.

1. Service providers may strategically deploy network functions in different geographically distributed nodes, to reduce their operational cost and decrease inter-autonomous system (AS) traffic. For example, NFs related to content delivery are likely to be placed closer to the network edge to reduce the traffic load in the core network.

- *W. Bao, B.B. Zhou, and A.Y. Zomaya are with the School of Computer Science, University of Sydney, Sydney, NSW 2006, Australia. E-mail: {wei.bao, bing.zhou, albert.zomaya}@sydney.edu.au.*
- *D. Yuan is with the School of Electrical and Information Engineering, University of Sydney, Sydney, NSW 2006, Australia. E-mail: dong.yuan@sydney.edu.au.*

Manuscript received 9 Apr. 2019; revised 3 Jan. 2020; accepted 8 Jan. 2020. Date of publication 17 Jan. 2020; date of current version 8 May 2020. (Corresponding author: Wei Bao.)
Recommended for acceptance by A. Louri.
Digital Object Identifier no. 10.1109/TC.2020.2967661

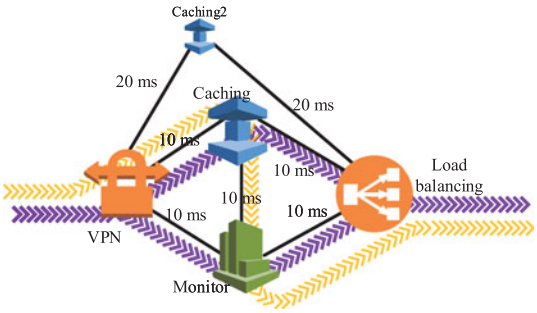


Fig. 1. An example of parallelizable network functions. VPN indicates virtual private networking, and DPI indicates deep packet inspection. The yellow path shows the service function chain, which takes 30 ms. The purple paths show that monitor and caching are conducted in parallel so that the overall delay is reduced to 20 ms. However, if "caching" is placed in "caching2," the overall delay is 40 ms even if parallelism is allowed.

In this paper, we are motivated to investigate the joint NF placement and parallelism problem. The dependency of the NFs is characterized by a directed acyclic graph (DAG). We aim to find the optimal placement to process the NFs in parallel without violating the DAG, to minimize the overall delay. However, we prove that directly solving the problem is NP-hard, due to the complicated coupling of DAG and multiple placement choices for each of the NFs. Another obstacle is that a large number of packet duplications may be generated to burden the network if we directly parallelize NFs without pruning the DAG.

In order to address such issues, we propose the Prune and Plant (P&P) scheme, to solve the problem effectively within polynomial complexity. P&P comprises two stages, Prune and Plant stages. In the Prune stage, we prune the original DAG into a series-parallel graph (SP-graph) so that (1) the original partial orders are not violated, (2) the parallelism of NFs is maintained, (3) the number of packet duplications and network burden are reduced, and (4) the NP-hardness is removed. In the Plant stage, given the pruned SP-graph, we find the optimal placement of NFs, within polynomial computational complexity, so that the overall delay is minimized. P&P scheme reduces overall delay by allowing parallelism of NFs. Meanwhile, it gives polynomial computational complexity as we prune the original DAG into an SP-graph, which avoids the NP-hardness.

Finally, we establish a real-world prototype, in order to test our proposed P&P scheme. Additional simulations are conducted to further test the performance in larger-scale systems. Both the prototype experiment and simulation demonstrate that P&P can bring a substantial performance gain compared with conventional benchmark schemes.

The rest of this paper is organized as follows: In Section 2, we present the system model and design overview of P&P scheme. In Sections 3 and 4, we propose the Prune stage and Plant stage respectively. In Section 5, we evaluate P&P and compare it with benchmark schemes by prototype experiment and simulation. Finally, prior related papers are discussed in Section 6 and the conclusions of this paper are given in Section 7.

2 System Model and Design Overview

2.1 NF Dependency Graph

We consider an NF dependency graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_1, v_2, \ldots, v_V\}$ denotes the set of NFs. $\mathcal{E}$ is a set of directed edges. $|\mathcal{V}| = V$ and $|\mathcal{E}| = E$. A directed edge from u to v $(u, v \in \mathcal{V})$, denoted by (u, v), indicates that NF u must be processed before NF v. u is called a predecessor of v and v is called a successor of u. We assume that there is no loop in $\mathcal{G}$ so that $\mathcal{G}$ is a DAG. In $\mathcal{G}$, we assume that there is one entry vertex v_s and one exit vertex v_d, which correspond to the NF to be processed first and last respectively. In $\mathcal{G}$, v_s is the only NF with 0 indegree and v_d is the only NF with 0 outdegree. Without loss of generality, let $v_1 = v_s$ and $v_V = v_d$.

2.2 Server Network

We also consider a server network $\mathcal{S} = (\mathcal{N}, \mathcal{L})$ as a fully connected undirected graph, where $\mathcal{N} = \{n_1, n_2, \ldots, n_N\}$ denotes the set of service nodes that can accommodate NFs and $\mathcal{L}$ denotes the set of (undirected) links connecting the service nodes. $|\mathcal{N}| = N$ and $|\mathcal{L}| = L$. For presentation convenience, $v \in \mathcal{V}$ is called as a vertex and $n \in \mathcal{N}$ is called a node in the rest of the paper. Given an NF v, it can only be placed at a subset of service nodes (i.e., v is location specific). Let $\mathcal{P}(v)$ denote the set of service nodes that can accommodate v. Let $P(v) = |\mathcal{P}(v)|$ denote the size of $\mathcal{P}(v)$. Let $p_i(v)$ denote the ith feasible service node. $p_i(v)$ is also called the ith placement of v. For example, suppose v can be placed at n_2, n_3, and n_5, then $\mathcal{P}(v) = \{n_2, n_3, n_5\}$, $P(v) = 3$, $p_1(v) = n_2$, $p_2(v) = n_3$, and $p_3(v) = n_5$.

Let $D(v, n)$ denote the processing delay of NF v at service node n. Since different NFs may be processed at different service nodes, we need to transmit packets among them. Let $d(n, n')$ denote the communication delay between n and n'. We allow $d(n, n') \neq d(n', n)$. $d(n, n') = \infty$ if n is not allowed to directly send data to n'. For simplicity, let $d_{ij} = d(n_i, n_j)$ denote the communication delay from the ith to jth service node and $D_{ij} = D(v_i, n_j)$ denote the processing delay of NF v_i at service node n_j.

2.3 NF Placement Problem Formulation

Our aim is to find the optimal placement of each NF (i.e., to find which service node each NF should be placed) so that the overall delay to complete all NFs is minimized. We note that the processing of an NF can be started when its predecessors are all finished, and the NF has received all packets completed by the predecessors. Formally, we formulate the following optimization Problem *P1*

$$\min\ t_N, \tag{1}$$

$$\text{s. t. } t_i - \tau_i \geq \sum_{k=1}^{N} x_{ik} D_{ik}, \forall i = 1, 2, \ldots, V, \tag{2}$$

$$\tau_j - t_i \geq \sum_{k=1}^{N} \sum_{l=1}^{N} x_{ik} x_{jl} d_{kl}, \forall i, j : (v_i, v_j) \in \mathcal{E}, \tag{3}$$

$$x_{ik} \in \{0, 1\}, \forall i, k, \tag{4}$$

$$\sum_{k:n_k \in \mathcal{P}(v_i)} x_{ik} = 1, \tag{5}$$

$$x_{ik} = 0; \forall i, k : n_k \notin \mathcal{P}(v_i), \tag{6}$$

where x_{ik} is a 0-1 variable indicating whether NF v_i is placed at service node n_k. (5) indicates that each NF is placed at exactly one service node. τ_i is the start time of processing NF v_i and t_i is the finish time. (2) indicates that the finish time minus the start time is greater than D_{ik} when NF v_i is processed at n_k. (3) means that we can start to process NF v_j no earlier than the time instant when the processed packets of v_i is received since v_i is a predecessor of v_j. In (3), the right hand side indicates the communication delay when NF v_i is processed at n_k and NF v_j is processed at n_l. (6) means that v_i cannot be placed at a service node not in $\mathcal{P}(v_i)$.

Theorem 1. *Problem* P1 *is NP-hard.*

See the Appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2020.2967661, for the proof. The proof is non-trivial, since it involves sophisticated mapping from 3-CNFSAT to *P1*.

We also note that neither the DAG nor the multiple choices for each NF can individually cause the NP-hardness. Suppose that each NF can only be placed at one service node, then the delay minimization problem is equivalent to a longest path problem in a DAG, which can be solved within polynomial complexity. Suppose the DAG is simplified as a chain, then the delay minimization problem can be solved by the Viterbi algorithm (a special case of our "Plant" stage shown in Section 4). Therefore, the NP-hardness is introduced by the complicated coupling between DAG and multiple choices of NFs.

In this paper, we do not analyze the gap with the optimal solution (e.g., approximation ratio). This is because unless P=NP, any algorithm with polynomial complexity has infinite worst-case performance gap with the optimal solution, which is implied in the proof of Theorem 1. See Appendix 8.5, available in the online supplemental material, for more details.

Note that, the scope of this paper is to investigate how to parallelize network functions rather than how to parallelize flows of different users. In this paper, given one set of NFs, our objective is to investigate how to parallelize them for delay reduction. The scope of this paper is different from one category of papers which study how to handle co-existing flows of different users. In those papers, e.g., [3], [4], [5], [6], [7], multiple flows (usually in form of service chains) are competing for limited network and computing resource and the objective is to wisely accommodate the flows to reduce delay or maximize utility.

2.4 Potential Solutions and Their Limitations

In Theorem 1, we have shown that *P1* is NP-hard, i.e., it cannot be solved within polynomial complexity. We also note that even if *P1* can be solved optimally, it may involve a large number of packet duplications (packet duplications can be realized by real-world NFV systems, such as [8]). This is because if an NF vertex has more than two successors, packets must be duplicated so that the outcome of the NF is delivered to all successors. As shown in Fig. 1, when VPN is processed, the outcome is sent to caching and monitor so that the packet is duplicated 1 time. A packet is duplicated $a-1$ times if an NF has a successors. In total, a packet is duplicated $E-V+1$ times. Packet duplications can cause high network burdens.

To complete all required VFs, we also examine *how many times that a packet is forwarded from one service node to another one* (defined as network burden). For example, in Fig. 1, the yellow path causes 3 times of packet forwarding, but the purple path causes 4 times of packet forwarding. If we directly apply the DAG optimization, there could be as many as E times of packet forwarding.

In order to address the NP-hardness, packet duplications and network burden, an alternative approach is to carry out *topology sorting* on $\mathcal{G}$. Through this approach, the NFs are sorted as a service chain. The optimal NF placement to minimize the overall delay of the chain can be obtained through Viterbi algorithm (a special case of our Plant stage). By doing so, packets are not duplicated as NFs are processed in sequence. Network burden is at most $V-1$. However, as discussed in Section 1, this may introduce longer delays as NFs are not parallelized.

A third approach, namely one-by-one, is to complete the NFs one by one without topology sorting. Through this approach, the objective is to find the optimal placement of NFs and processing orders of NFs to minimize the overall delay. Packets are not duplicated. However, solving this problem is still NP-hard. Consider a special case that all NFs, except the entry and exiting NFs, can be processed in parallel, and each NF can only be placed at one different service node. Then, finding the minimum delay is equivalent to a traveling salesman problem (TSP), which is NP-hard.

2.5 NF Parallelism and Motivation of P&P Scheme

From the above discussion, we notice that solving Problem *P1* is challenging. We need to consider three factors: (1) polynomial computational complexity, (2) a small number of duplicated packets/low network burden, and (3) small delay. Unfortunately, no solution can achieve all of them. In what follows, we aim to balance these factors and propose Prune and Plant scheme. In the P&P scheme, we first prune the original DAG into an SP-graph, by the Graph Pruning (GP) algorithm. Then we find the optimal NF placement with respect to the generated SP-graph, by the Optimal Planting (OP) algorithm. The benefit of P&P is three-fold: (1) Compared with DAG, the partial orders of NFs in the DAG are reasonably simplified so that the problem is tractable within polynomial complexity. (2) Compared with topology sorting and one-by-one, NFs are still processed in parallel to decrease the overall delay. (3) P&P only slightly increases the number of packet duplications and network burden. A detailed comparison of the aforementioned four schemes is shown in Table 1, showing the advantages of P&P.

In the following two sections, we propose the P&P scheme. The Prune stage is proposed in Section 3 and the Plant stage is proposed in Section 4.

3 Stage 1: Graph Pruning

In this section, we prune the DAG $\mathcal{G}$ to generate an SP-graph $\mathcal{G}'$, so that NFs are processed according to the orders in the SP-graph. Different from topology sorting that all NFs are processed in sequence, an SP-graph still allows NFs to be processed in parallel.

TABLE 1
Comparison of Schemes

Scheme	Complexity	Duplication	Burden	Delay
DAG Optimization	NP-hard	$E-V+1$	E	Optimal.
Topology Sorting	$O(VN^3)$	None	$V-1$	Optimal for sorted chain.
One-by-one	NP-hard	None	$V-1$	Optimal for all chains.
P&P	$O(VN^4+E)$	$\leq V-1$	$\leq 2V-2$	Optimal for SP-graph.

Note that, the SP-graph is generated from the original DAG. The orders of NFs in $\mathcal{G}'$ cannot violate the orders in $\mathcal{G}$. In other words, u cannot be processed if an upstreaming NF v is not completed.

Algorithm 1. Graph Pruning (GP) Algorithm

```
1  Load the system input: G; S; d(n,n'), ∀n,n' ∈ N; D(v,n), ∀v ∈ V, n ∈ N; P(v), ∀v ∈ V;
2  global G, G' = ∅;
3  All vertices in G are colored white;
4  GPrune(∅, v_s);
5  return (G');
6
7  function GPrune(UpStream, BranchHead);
8  CurrentBranch⇐{BranchHead};
9  CurrentVer⇐BranchHead;
10 while True do
11   NextList⇐FindZeroIndegree(G\UpStream);
12   NextList⇐NextList⋂WhiteVer;
13   if |NextList| = 1 then
14     NextVer.Black(); /*NextVer is the only element in NextLest*/;
15     UpStream⇐UpStream ⋃ NextVer;
16     CurrentBranch⇐CurrentBranch ⋃ NextVer;
17     DirectEdge(CurrentVer,NextVer);
18     CurrentVer⇐NextVer;
19   else if |NextList| > 1 then
20     VirtualSink⇐NewVer();
21     Branches⇐ ∅;
22     for each NextVer in NextList do
23       NextVer.Grey();
24       DirectLink(CurrentVer, NextVer);
25     end
26     for each NextVer in NextList do
27       NextVer.Black();
28       (NewBranch,Tail)⇐GPrune(Upstream ⋃ NextVer,NextVer);
29       Branches⇐Branches ⋃ NewBranch;
30       DirectLink(Tail, VirtualSink);
31     end
32     CurrentBranch⇐CurrentBranch ⋃ Branches ⋃ VirtualSink;
33     Upstream⇐Upstream ⋃ Branches;
34     CurrentVer⇐VirtualSink;
35   else
36     return (CurrentBranch, CurrentVer);
37   end
38 end
```

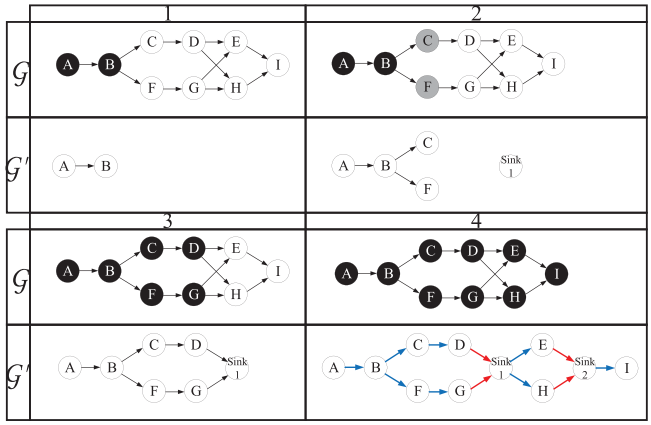


Fig. 2. An example of GP Algorithm.

3.1 GP Algorithm Design

Algorithm 1 shows the Graph Pruning algorithm. It recursively calls the function `GPrune()`. The two arguments of `GPrune()`, `Upstream` and `BranchHead` indicate the set of vertices that have been settled before this level of recursion and the vertex that leads this level of recursion. At the base level, we call `GPrune(∅, $v_s$)` to initiate the procedure. The algorithm keeps $\mathcal{G}$ and $\mathcal{G}'$ as global variables. As an example shown in Fig. 2, the original $\mathcal{G}$ is not changed, but the vertices in $\mathcal{G}$ could be colored during the progress. The generated SP-graph $\mathcal{G}'$ is constructed step by step. New vertices are added gradually in $\mathcal{G}'$.

Each vertex in $\mathcal{G}$ is flagged with one of the following three colors: white, grey, and black. `Ver.Black()` and `Ver.Grey()` flag vertex `Ver` with grey and black respectively. White means that the vertex has never been visited; black means that the vertex has been settled in the SP-graph; and grey means that the vertex has been visited but further processing is required.

In the algorithm, we search for new vertices with zero indegree and put them in `NextList` (Lines 11–12). `CurrentVer` is used to label the vertex in consideration. We are free to process NFs in `NextList` as their predecessors are all completed. If there is only one vertex in `NextList` (Lines 13–18), this NF (i.e., `NextVer`) is the only one to be processed next. Therefore, we can build an edge from `CurrentVer` to `NextVer` in the copied graph $\mathcal{G}'$ and this is done by the function `DirectEdge(CurrentVer, NextVer)` (Line 17). An example is shown in Fig. 2, step 1.

If there are more than one vertices with zero indegree in `NextList` (Lines 19–34), all of them can be processed in parallel, and thus we can initiate new parallel branches for each of them. This is the key step to realize NF parallelism. Then, we recursively call the `GPrune()` to build each of the branches. Please note that we also need to label the vertices in `NextList` in grey color to prevent that they are considered as "new vertices with zero degree" again. In Line 28, we run the function `GPrune()` to search for the branch starting from each `NextVer` in `NextList`. Please note that `GPrune()` returns the branch starting from `NextVer`, as well as the last vertex terminating that branch. The variable `Upstream` records the set of upstreaming nodes that have already been settled before the new level of recursion. In addition, whenever multiple branches have been generated,

we guarantee that all the branches will be linked to a common virtual sink. In Line 20, we generate a virtual sink. Then all the tails of the branches will link to that sink in Line 30. It is worth noting that the virtual sink can be regarded as an empty NF that can be processed anywhere with zero processing delay. (The empty NF can be easily implemented in reality as it only needs to receive and forward packets.) In Fig. 2, the above procedure is shown in steps 2–3. C and F are both selected in the `NextList`. Both of them are colored gray and we then run `GPrune()` on C and F respectively. In the branch starting from C, we find $C \rightarrow D$; in the branch starting from G, we find $F \rightarrow G$. Both D and G are then linked to Sink1.

Finally, `GPrune()` returns in Line 36 as no more NF with zero indegree can be found.

3.2 Complexity Analysis

The computational complexity of GP algorithm is $O(V + E)$. This is because each vertex is visited at most twice when it is colored grey and black. Each edge is visited once when we search for the nodes in `NextList`. Each node in the original $\mathcal{G}$ can generate at most one virtual sink. Therefore, the overall computational complexity of GP is $O(V + E)$.

3.3 Number of Packet Duplications and Network Burden

In order to study the number of packet duplications if $\mathcal{G}'$ is used, we first need to study the number of vertices and edges in $\mathcal{G}'$. In $\mathcal{G}'$, the set of edges is $\mathcal{E}'$. The set of vertices $\mathcal{V}'$ comprises the set of vertices copied from $\mathcal{G}$, denoted by $\mathcal{V}_{copy}$, and the set of virtual sinks, denoted by $\mathcal{V}_{virtual}$. $|\mathcal{V}_{virtual}| \leq |\mathcal{V}_{copy}|$ since each vertex $\mathcal{G}$ can generate at most one virtual sink. For each vertex in $\mathcal{V}_{copy}$, its indegree is 0 or 1. Therefore, in $\mathcal{G}'$, the number of edges ending at one vertex in $\mathcal{V}_{copy}$ is $V - 1$ as shown in the blue color in step 4 in Fig. 2. The number of edges ending at virtual sinks is smaller than the number of edges ending at copied vertices, as shown in the red color in step 4. This is because given a red edge terminating a branch, we can find a blue edge initiating the branch. Therefore, we have $|\mathcal{E}'| \leq 2V - 2$ in $\mathcal{G}'$. Based on the discussion in Section 2.4, the number of duplications is equal to $|\mathcal{E}'| - |\mathcal{V}'| + 1 \leq (2V - 2) - V + 1 = V - 1$. The network burden is equal to the number of edges of $\mathcal{G}'$, which is at most $2V - 2$.

4 Stage 2: Optimal Planting

In this section, we study the optimal placement of NFs given the generated SP-graph $\mathcal{G}'$, to minimize the overall delay with respect to $\mathcal{G}'$.

4.1 Preparations

4.1.1 Redefinition of Link Delays and Processing Delays

For presentation convenience in this section, we define $c(v, u, i, j)$ as the communication delay from the ith placement of NF v to the jth placement of NF u, i.e., $c(v, u, i, j) \triangleq d(p_i(v), p_j(u))$, $1 \leq i \leq P(v), 1 \leq j \leq P(u)$. Let $C(v, i)$ denote the processing delay of NF v at its ith placement, i.e., $C(v, i) \triangleq D(v, p_i(v)), 1 \leq i \leq P(v)$.

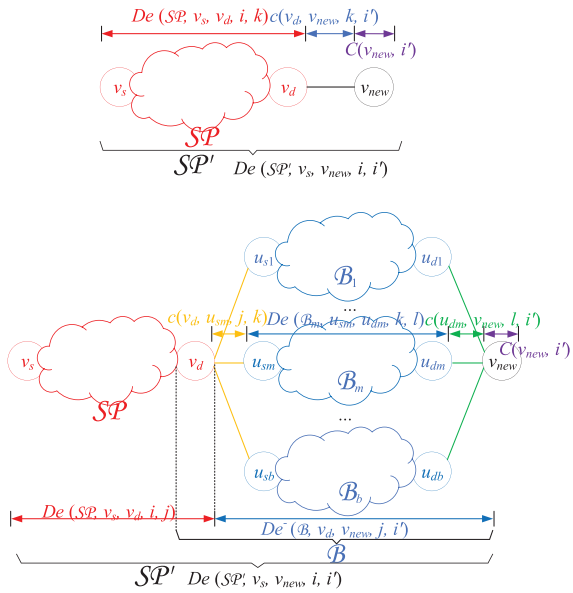


Fig. 3. Top: Situation 1. Bottom: Situation 2.

4.1.2 Accommodation of Virtual Sinks

In Section 3, we have added virtual sinks in $\mathcal{G}'$. These virtual sinks are empty NFs, and can be processed at any service node in $\mathcal{N}$, and their processing delays are all zeros. In other words, $\forall v \in \mathcal{V}_{virtual}$, $\mathcal{P}(v) = \{n_1, n_2, \ldots, n_N\}$ and $C(v, i) = 0$, $\forall 1 \leq i \leq N$.

4.2 Construction of the Bellman Equation

We aim to derive the minimum delay of the placement of SP-graph through a dynamic programming approach. We note that an SP-graph can be decomposed into smaller-scale sub-SP-graphs. We investigate the optimal substructure through the decomposition. Through this way, we are able to characterize the Bellman equation to realize a dynamic programming solution.

Let $De(\mathcal{SP}, v_s, v_d, i, i')$ denote the minimum overall delay of a (sub-)SP-graph $\mathcal{SP}$, with the entry NF v_s and exit NF v_d, and v_s is placed at its ith placement, and v_d is placed at its i'th placement. Here, $1 \leq i \leq P(v_s)$ and $1 \leq i' \leq P(v_d)$. In what follows, we study two situations when additional vertices are added to $\mathcal{SP}$. We aim to characterize the minimum overall delay at the expanded SP-graph through dynamic programming. As shown in Fig. 3, the two situations are:

- *Situation 1*: Appending one vertex.
- *Situation 2*: Appending multiple parallel branches (each branch is an SP-graph).

Since any SP-graph can be constructed recursively by Situations 1 and 2, we are able to characterize the minimum delay of an arbitrary SP-graph by deriving the Bellman equations for Situations 1 and 2.

For briefness, the term "beginning of an NF v" indicates the time instant to start to process v and the term "end of v" indicates the completion time of processing v.

4.2.1 Appending One Vertex

As shown in Fig. 3 (top-left), in Situation 1, another NF v_{new} is appended at the end of $\mathcal{SP}$, forming $\mathcal{SP}'$. Our aim is to

obtain the minimum overall delay of $\mathcal{SP}'$, $De(\mathcal{SP}', v_s, v_{new}, i, i'), \forall i, i'$, given the optimality of $\mathcal{SP}$. This can be characterized by the following Bellman equation

$$De(\mathcal{SP}', v_s, v_{new}, i, i') = \min_{1 \le k \le P(v_d)} De(\mathcal{SP}, v_s, v_d, i, k) + c(v_d, v_{new}, k, i') + C(v_{new}, i'), \forall 1 \le i \le P(v_s), \quad 1 \le i' \le P(v_{new}). \tag{7}$$

This is because the overall delay from the beginning of v_s (ith placement) to the end of v_{new} (i'th placement) is equal to the delay from the beginning of v_s (ith placement) to the end of v_d (kth placement) plus the communication delay from v_d (kth placement) to v_{new} (i'th placement) plus the processing delay at v_{new} (i'th placement). In order to find the minimum overall delay, we search through all possible k and find the best k that achieves minimum delay.

4.2.2 Appending Multiple Parallel Branches

As shown in Fig. 3 (bottom-right), in Situation 2, suppose that $\mathcal{SP}$ is with head v_s and tail v_d. At the end of v_d, $b > 2$ branches are appended. The mth branch itself is an SP-graph, denoted as $\mathcal{B}_m$, with head u_{sm} and tail u_{dm}. The tails of the m branches are all linked to a common vertex v_{new}. Please note that the overall graph is also an SP-graph, denoted by $\mathcal{SP}'$. Our aim is to obtain the minimum overall delay of $\mathcal{SP}'$ given the optimality of $\mathcal{SP}$ and $\mathcal{B}_m, \forall m$. Let $\mathcal{B}$ denote the portion of $\mathcal{SP}'$ from v_d to v_{new}. The overall delay spent at $\mathcal{SP}'$ can be calculated as the sum of $\langle$a$\rangle$ from the beginning of v_s to the end of v_d and $\langle$b$\rangle$ from the end of v_d to the end of v_{new}.

First, we characterize the delay of part $\langle$b$\rangle$. Let $De^-(\mathcal{B}, v_d, v_{new}, j, i')$ denote the minimum delay in this part. Please note that here we use the term De^- instead of De because the processing delay at v_d is not counted in $\langle$b$\rangle$. Given the minimum delays at each branch $\mathcal{B}_m$, we have

$$De^-(\mathcal{B}, v_d, v_{new}, j, i') = \max_{1 \le m \le b} \Big[\min_{1 \le k \le P(u_{sm}), 1 \le l \le P(u_{dm})} \big[c(u_d, u_{sm}, j, k) + De(\mathcal{B}_m, u_{sm}, u_{dm}, k, l) + c(u_{dm}, v_{new}, l, i') \big] \Big] + C(v_{new}, i'). \tag{8}$$

In (8), the operation $\max_m$ is employed since packets will be processed by the b branches in parallel. v_{new} can be started when all of the b branches are completed. The overall delay is the maximum delay of all branches. Within one branch m, the operation $\min_{k,l}$ is employed since the delay from the end of v_d (jth placement) to the start at v_{new} (i'th placement) is equal to the communication delay from v_d (jth placement) to u_{sm} (kth placement), plus the delay from the beginning of u_{sm} (kth placement) to the end of u_{dm} (lth placement), plus the communication delay from u_{dm} (lth placement) to v_{new} (i'th placement). We search through all possible k and l to find the best ones that achieve minimum $De^-(\mathcal{B}, v_d, v_{new}, i, i')$.

Finally, the overall delay of $\mathcal{SP}'$ is computed as the sum of delays of $\langle$a$\rangle$ and $\langle$b$\rangle$. The minimum delay is minimized over all possible j as follows

$$De(\mathcal{SP}', v_s, v_{new}, i, i') = \min_{j \in P(v_d)} \Big[De(\mathcal{SP}, v_s, v_d, i, j) + De^-(\mathcal{B}, v_d, v_{new}, j, i') \Big]. \tag{9}$$

4.3 OP Algorithm Design

In this subsection, we present the algorithm to find the minimum delay in the SP-graph (Optimal Planting algorithm) based on the derived Bellman equations in Situations 1 and 2. It is shown in Algorithm 2 where `OptPlant()` is recursively called.

Algorithm 2. Optimal Planting (OP) Algorithm

1 Load the SP-graph generated in Algorithm 1;
2 **global** $\mathcal{G}'$, communication delays, processing delays;
3 **De** $\Leftarrow$ `OptPlant(`v_s`,` v_d`)`;
4 `MinDe` $\Leftarrow \min$[**De**];
5
6 **function** `OptPlant(Head, Tail)`;
7 `Current`$\Leftarrow$`Head`;
8 $\texttt{De}_{ii} \Leftarrow C(\texttt{Current}, i), \forall 1 \le i \le P(\texttt{Head})$;
9 $\texttt{De}_{ii'} \Leftarrow \infty, \forall i \ne i'$;
10 **if** $\texttt{Outdegree}(\texttt{Current}) = 1$ **then**
11 $\texttt{NewDe}_{ii'} \Leftarrow \min_{1 \le k \le P(\texttt{Current})}[\texttt{De}_{ik} + c(\texttt{Current}, \texttt{Next}, k, i') + C(\texttt{Next}, i')], \forall 1 \le i \le P(\texttt{Head}), 1 \le i' \le P(\texttt{Next})$;
12 Record $\arg\min_k$ in the above expression;
13 **De** $\Leftarrow$ **NewDe**;
14 Label the edge from `Current` to `Next`;
15 `Current`$\Leftarrow$`Next`;
16 **else if** $\texttt{Outdegree(Current)} > 1$ **then**
17 $m \Leftarrow 0$;
18 `Sink`$\Leftarrow$`Sink(Current)`;
19 **for each** `Vertex` **in** `Successor(Current)` **do**
20 $\texttt{Next}_m \Leftarrow$ `Vertex`;
21 Label the edge from `Current` to $\texttt{Next}_m$;
22 $\texttt{Tail}_m \Leftarrow$ `Tail(Vertex)`;
23 $\textbf{BDe}_m \Leftarrow$ `OptPlant(`$\texttt{Next}_m$`,` $\texttt{Tail}_m$`)`;
24 $m \Leftarrow m + 1, b \Leftarrow m$;
25 **end**
26 $\texttt{De}^-_{ji'} \Leftarrow \max_{1 \le m \le b}[\min_{1 \le k \le P(\texttt{Next}_m), 1 \le l \le P(\texttt{Tail}_m)} [c(\texttt{Current}, \texttt{Next}_m, j, k) + \textbf{BDe}_m(k, l) + c(\texttt{Tail}_m, \texttt{Sink}, l, i')]] + C(\texttt{Sink}, i'), \forall 1 \le j \le P(\texttt{Current}), 1 \le i' \le P(\texttt{Sink})$;
27 $\texttt{NewDe}_{ii'} \Leftarrow \min_{1 \le j \le P(\texttt{Current})}[\texttt{De}_{ij} + \texttt{De}^-_{ji'}], \forall 1 \le i \le P(\texttt{Head}), 1 \le i' \le P(\texttt{Sink})$;
28 Record $\arg\min_{k,l}$ and $\arg\min_j$ in the above expressions;
29 **De** $\Leftarrow$ **NewDe**;
30 `Current` $\Leftarrow$ `Sink`;
31 **else**
32 return **De**;
33 **end**

`OptPlant(Head, Tail)` is to find the minimum delay for the (sub-)SP-graph starting from `Head` and ending at `Tail`. We call function `OptPlant(`v_s`,` v_d`)` in the main function. Please note that `OptPlant()` returns a matrix of **De** since De values in (7) and (8) are derived for any placement so that it is regarded as a matrix. The minimum delay is the min of all entries in the matrix.

Within `OptPlant()`, we scan vertices from `Head` to `Tail`. `Current` records where we have already scanned. **De** is the matrix recording the smallest delays from `Head` to `Current`. $\texttt{De}_{ii'}$ is the entry of the ith row and i'th column.

In Lines 10–15, if `Current` has one successor, there is one NF vertex appended so that we need to handle Situation 1. Therefore, we can update the `De` values in Line 11 following the Bellman Equation (7).

In Lines 16–30, if `Current` has multiple successors, `Current` is connected to multiple branches so that we need to handle Situation 2. All successors of `Current` are visited. The mth branch, starting from $\texttt{Next}_m$ and ending at `Tail(`$\texttt{Next}_m$`)`, is a sub-SP-graph, so that its minimum delay can be derived by recursively calling `OptPlant(`$\texttt{Next}_m$`, Tail(`$\texttt{Next}_m$`))`. The minimum delays of the mth branch are stored at matrix $\mathbf{BDe}_m$.

Please note that when we construct the SP-graph $\mathcal{G}'$ in Algorithm 1, given a vertex v with multiple successors, there is a virtual sink generated (Line 20 in Algorithm 1), and the subbranches merge at the virtual sink. The virtual sink generated by v is memorized in `Sink(`v`)` to be used in Algorithm 2. Similarly, given the first vertex u of a branch, the tail of that branch can be memorized by `Tail(`u`)`. Therefore, in Lines 18 and 22, we can directly find the virtual sink and branch tail.

Now that the minimum delays of all branches are derived, we implement Bellman Equations (8) and (9) in Lines 26 and 27 respectively.

Please note that Lines 14 and 21 are not necessarily executed. They are useful for the complexity analysis in Section 4.4.

4.4 Complexity Analysis

In Algorithm 2, we recursively decompose SP-graph in Situations 1 and 2. Suppose that Situation 1 occurs S_1 times and Situation 2 occurs S_2 times. We can characterize S_1 and S_2 by investigating the edges in $\mathcal{G}'$, since each edge is labeled only once in Line 14 or Line 21.

In Algorithm 2, we "label" edges in $\mathcal{G}'$ and each edge is labeled once. Whenever Situation 1 occurs, one edge, from `Current` to `Next`, in $\mathcal{G}'$ is labeled. Therefore, edges are labeled S_1 times in Situation 1. When Situation 2 occurs for the rth time, m_r edges are labeled, and the number m_r is equal to the number branches attached to `Current` (i.e., the number of successors of `Current`). Therefore, edges are labeled $\sum_{r=1}^{S_2} m_r$ times in Situation 2. Finally, $S_1 + \sum_{r=1}^{S_2} m_r = |\mathcal{E}'|$ since an edge in $\mathcal{G}'$ is visited once.

Whenever Situation 1 occurs, Lines 10–15 are executed. The computational complexity of these Lines is $O(N^3)$ since we need to traverse all possible i, k, and i' and all of them are smaller or equal to N. When Situation 2 occurs, Lines 16–30 are run. The computational complexity of Line 26 is $O(m_r N^4)$ for the rth occurrence of Situation 2. This is because we need to traverse all possible j, k, l and i' and all of them are no greater than N. The factor m_r comes from the maximization of m_r branches. The computational complexity of Line 27 is $O(N^3)$, which can be ignored compared with $O(m_r N^4)$.

Therefore, the overall computational complexity is $O(S_1 N^3 + \sum_{r=1}^{S_2} m_r N^4)$. It is in the scale of $O(VN^4)$, since $S_1 + \sum_{r=1}^{S_2} m_r = |\mathcal{E}'|$, and $|\mathcal{E}'| = O(V)$ (See Section 3.3). Please note that if $\mathcal{G}'$ is a chain, only Situation 1 occurs. OP algorithm is equivalent to Viterbi algorithm with computational complexity $O(VN^3)$.

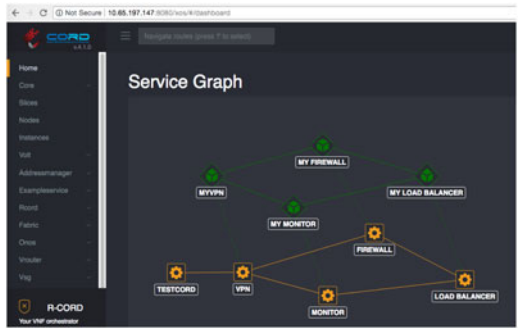


Fig. 4. The portal of XOS.

Finally, by combining the conclusion in Section 3.2, we have the computational complexity of Algorithm 1 as $O(V+E)$, and computational complexity of Algorithm 2 as $O(VN^4)$. Therefore, overall complexity is $O(VN^4 + E)$.

Proposition 1. *The computational complexity of P&P scheme is* $O(VN^4 + E)$.

5 Evaluation

5.1 System Implementation

In order to evaluate our proposed P&P algorithm, we utilize the Open CORD (Central Office Re-architected as a Datacenter https://opencord.org/) project to implement our prototype. Open CORD is a carrier-grade solution that combines software-defined networking (SDN) and NFV built from commodity hardware and open source software (e.g., OpenStack, ONOS, and XOS). We installed and configured the system in a Dell PowerEdge R730 server (Dual Intel Xeon E5-2650 v4 CPU and 64 GB memory) with the reference to CORD 4.1, and realized the VNF parallelism in XOS, i.e., a model-based platform for assembling, controlling, and composing VNFs. The system is composed of five virtual nodes (one head and four compute nodes). The portal of XOS is shown in Fig. 4. In our prototype, we define the VNFs (also called services in XOS) in the head node (shown as the yellow icons in the portal), and create and deploy VM-hosted VNFs instances (shown as the green icons in the portal) in the compute nodes. Hence different VNFs can be executed in parallel in the compute nodes.

5.2 Benchmark Schemes

In this section, we compare our proposed Prune and Plant scheme with six benchmark schemes as follows.

5.2.1 Topology Sort (T-Sort)

We first employ topology sort on the original DAG $\mathcal{G}$ so that a service function chain is generated. Then, we apply Algorithm 2 to derive the placement for the chain. Algorithm 2 realizes the *optimal* placement for the generated chain.

5.2.2 Simple One-by-One (1BY1)

We are also interested in the one-by-one approach introduced in Section 2.4. However, due to its NP-hardness, we implement a simplified version where topology sort is run multiple times to generate multiple possible chains (we randomly select the next vertex in the chain if more than one

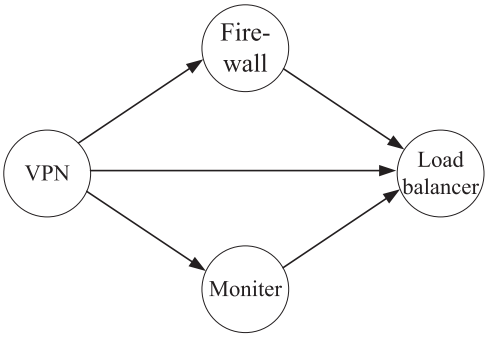


Fig. 5. Dependency DAG in Setting 1.

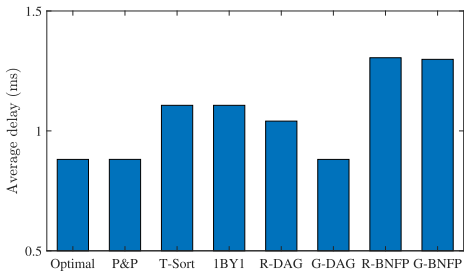


Fig. 6. Delay performance of the experiment.

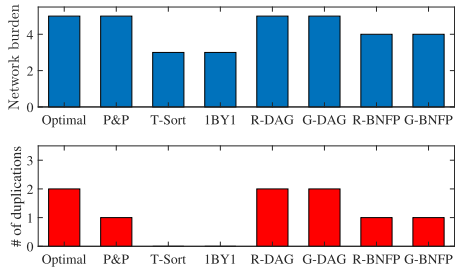


Fig. 7. Network burden and packet duplications of the experiment.

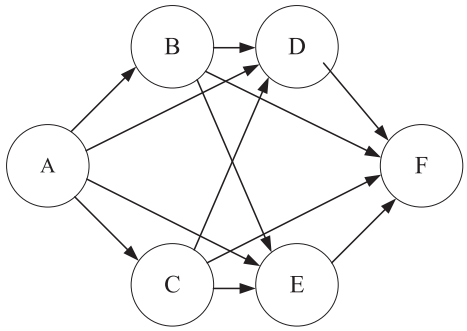


Fig. 8. Dependency DAG in Setting 2.

vertices are feasible). The number of possible chains grows exponentially with respect to the number of vertices so we only generate 10 chains. For each generated chain, we apply Algorithm 2, and then the best chain is used.

5.2.3 Random Placement for DAG (R-DAG)

We directly operate on the original DAG $\mathcal{G}$ without pruning. Each NF is started when it has received all packets completed by the predecessors. Each NF sends processed packets to all successors. However, the searching for the optimal placement is NP-hard (Problem *P1*). Here we randomly choose a service node for each NF.

5.2.4 Greedy Placement for DAG (G-DAG)

It is similar to R-DAG by directly operating on the original DAG $\mathcal{G}$ without pruning. However, when we search for the placement of an NF, the service node which will complete the NF earliest is chosen.

5.2.5 Benchmark Network Function Parallelism With Random and Greedy Placement (R-BNFP and G-BNFP)

We employ the approach to parallelize NFs in Section 4.4 in [2]. However, [2] does not address how to deploy NFs. In R-BNFP, all NFs are randomly placed. In G-BNFP, when we search for the placement of an NF, the service node which will complete the NF earliest will be chosen.

5.3 Experimental Results

We implement four NFs with dependency graph shown in Fig. 5 (Setting 1), and the four NFs can be placed in the four compute nodes in the system, where virtual connections among NFs were established. The delay performance of P&P and benchmark schemes are compared in Fig. 6. We also show the optimal delay performance by exhaustive search. In Fig. 7, we show the network burden and number of packet duplications. (Recall that network burden is defined as the number of times a packet is transmitted between two service nodes.) The two figures show that the delay performance of P&P is very close to the optimal one, while P&P can reduce the number of duplications. They also show that P&P reduces delay compared with T-Sort and 1BY1, as it allows NF parallelism. Its performance is better than R-DAD and is close to G-DAG. (This is because the topology of $\mathcal{G}$ is simple so that greedy placement is already good enough in this scenario.) Compared with G-DAG, P&P decreases network burden and the number of packet duplications by 1. P&P also performs better and R-BNFP and G-BNFP, as both the "Prune" and "Plant" stages will introduce performance gain. In what follows, we carry out simulation to consider a variety of system settings.

5.4 Simulation

In what follows, we also test the performance of P&P under a variety of settings by simulation. The simulation environment is Matlab 2018b running in Dell Optiplex 7,060 with Intel(R) Core(TM) i7-8700 CPU @3.2 GHz 12 cores, and 16 GB RAM. We first test a more complicated network service that consists of 6 NFs, including VPN (A), DPI for protocol noncompliance (B), DPI for intrusion detection (C), monitor (D), caching (E), and load balancing (F), which is referred to as Setting 2. The dependencies of these NFs are characterized by the DAG shown in Fig. 8. The pruned graph by Algorithm 1 is shown in Fig. 9 for reference. The server network comprises 12 service nodes. The 12 service nodes form a fully connected network. We test the performance under different communication delay and processing delay settings. We consider that communication delay between any two service

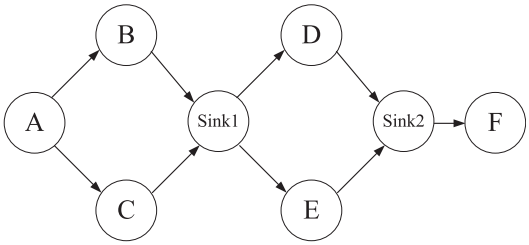


Fig. 9. Pruned graph by Algorithm 1 in Setting 2.

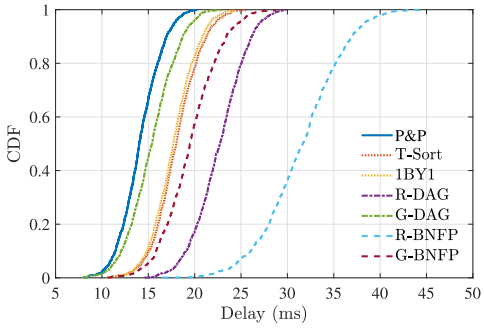


Fig. 10. CDF of delays in Setting 2.

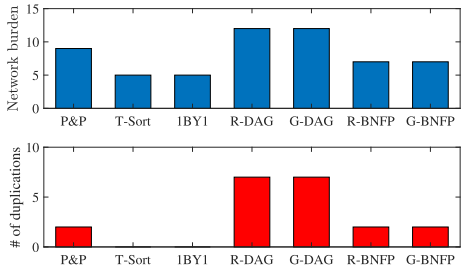


Fig. 11. Network burden and packet duplications in Setting 2.

nodes is uniformly distributed in $[1, 7]$ ms. The processing delay of an NF in a service node is uniformly distributed in [1,3] ms. For each service node n, with probability 0.3, an NF v can be placed in it, i.e., $n \in \mathcal{P}(v)$. The simulation is run for 1000 rounds.

If Fig. 10, we show the empirical cumulative distribution function (CDF) of the delays of P&P and benchmark schemes. In Fig. 11, we show the network burden and number of packet duplications. In this topology, we have $E = 12$ and $V = 6$ in the topology. Following Table. 1, for P&P, the number of duplications and network burden are 2 and 9, which are smaller than $V - 1$ and $2V - 2$ respectively. T-Sort and 1BY1 have 0 duplication and $V - 1 = 5$ network burden. R-DAG and G-DAG have $E - V + 1 = 7$ duplications and $E = 12$ network burden. As a result, we show that P&P has the best delay performance as its CDF is the leftmost one compared with other CDF curves. G-DAG leads to good delay performance (but it is still worse than P&P) since it keeps all possible NF parallelism. However, this is at the cost of much heavier network burden and a larger number of packet duplications compared with P&P. We also notice that G-DAG leads to much better performance compared with R-DAG. Random placement destroys the performance gain brought by parallelism and thus it is critical to place the NFs wisely even if they are already parallelized. For T-Sort and 1BY1, they lead to the minimum network burden and

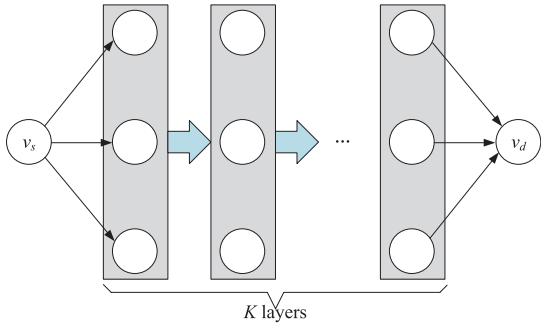


Fig. 12. Dependency DAG in Setting 3.

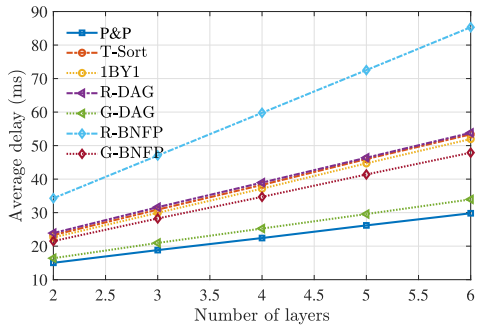


Fig. 13. Average delay under different number of layers.

zero packet duplication as they are operated as service chains, but the delay performance is much worse as they do not allow parallelism. 1BY1 is slightly better than T-Sort as it selects the best chain over multiple random attempts. The "Prune" stage is significant as it decreases the network burden and packet duplications while maintaining the parallelism to reduce delay. Finally, R-BNFP and G-BNFP lead to similar levels of network burdens and packet duplications compared with P&P. However, they do not give good delay performance as placement optimization is not considered. The "Plant" stage is also significant as it minimizes delay, to outperform greedy and random placement.

We also test more complicated network services consisting of a greater number of NFs. We envision that more and more complicated network services that involve more NFs will be developed in the near future to satisfy the exploding service requirements. The dependencies of NFs are shown in Fig. 12 (Setting 3). There are K layers and each layer has three NFs (so that there are $3K + 2$ NFs in total). There is a directed edge from any NF in the ith layer to any NF in the $(i + 1)$th layer and to any NF in the $(i + 2)$th layer. All the other settings are the same as those in Setting 2.

We first study the delay performance, network burden, and number of duplications under different number of layers. The results are shown in Figs. 13, 14, and 15. Please note that T-Sort and 1BY1 (R-DAG and G-DAG; and R-BNFP and G-BNFP) lead to the same network burden and number of duplications so that they are merged in the same curve in Figs. 14 and 15. The figures further confirm the observations under Setting 2. P&P gives the best delay performance: slightly better than G-DAG, and significantly better than all the other schemes. Compared with G-DAG, P&P causes much less network burden and much fewer number of packet duplications. The above observations demonstrate

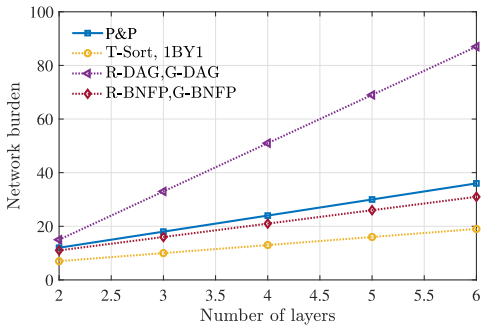


Fig. 14. Network burden under different number of layers.

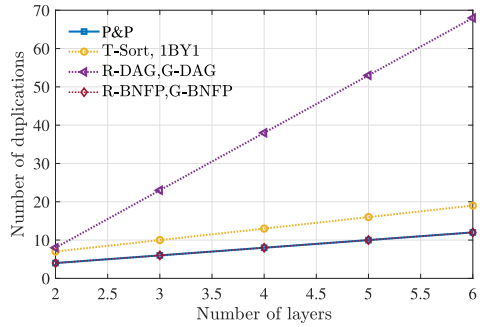


Fig. 15. Number of duplications under different number of layers.

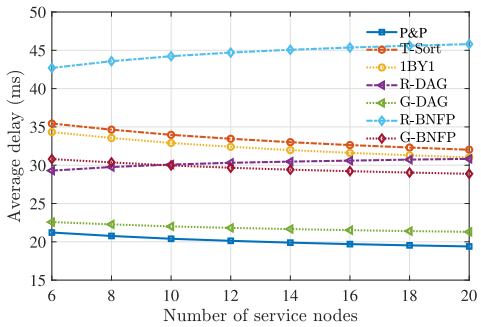


Fig. 16. Average delay under different number of service nodes.

that P&P is a promising solution that can balance the delay performance, network burden, and packet duplications.

Finally, we fix the number of layers K at $K = 3$ but varies the number of service nodes in the system. The service nodes form a fully connected network. Still, all the other settings remain the same. The outcome is shown in Fig. 16. P&P gives the best performance under different number of service nodes. We also note that when the number of service nodes increases, delays of P&P, T-Sort, 1BY1, G-DAG, and G-BNFP are reduced as more service nodes are available, to provide better placement solutions. However, R-DAG, and R-BNFP give worse solutions as the placement is random. It is more likely that the random placement will choose a worse solution given the larger number of service nodes. This is another observation that demonstrates the usefulness of the "Plant" stage to wisely place the NFs.

6 Related Work

By virtualizing NFs that are traditionally implemented in dedicated hardware, NFV has been a promising technology for improving service delivery flexibility and reducing overall costs. However, challenges still exist in network performance guarantee, dynamic virtual function instantiation, and efficient function placement and migration [1], [9]. Successful NFV systems and frameworks have been developed, such as ClickOS [10], StateAlyzr [11], E2 [12], OpenNF [13], ParaBox [14], etc. NFV has been enabled in SDN data plane [15] and the tight coupling of state and processing can be broken by Stateless Network Functions [16], so that NFs can be virtualized, placed, migrated, and orchestrated efficiently and reliably.

Without considering VNF parallelism, performance optimization and enhancement of NFV has attracted much attention [17]. One set of previous work investigated the placement of VNFs [18], [19] to optimize the system performance under certain resource constraints. For example, [3] minimized the traffic delivery distance and the VNF setup cost; [4] minimized the number of serving instances; [5] minimized the resource usage; [6], [20] minimized the total cost of deploying VNF instances in tree-structured networks; [21] maximized the number of admitted requests given a soft real-time guarantee; [22] minimized the overall traffic volume; [23] optimized the resource utilization. [24], [25] minimized the average latency in the presence of satellite links and cloudlets respectively. The papers [7], [26], [27], [28] further jointly considered chaining and placement of VNFs. In addition, dynamic SFC embedding problem is also investigated, when the chain of functions could change during the life of a session [29]. However none of them considered the parallelism of NFs as a mean to improve the system performance.

Parallel processing is an efficient way to accelerate the network performance. NF parallelism is enabled by recent hardware and system realization. ClickNP [30] utilized FPGA to achieve parallelism inside NFs. As a possible hardware infrastructure for NFs, P4 [8] abstracted NFs into multiple stages of tables and enabled parallelism inside an NF or across NFs. In multicast networks, the benefits of parallel NFs are investigated by pioneer research work. For example, [31] optimized service function tree (SFT) embedding to reduce the NFV enabled multicast traffic. In more generic system settings, [2] developed a system that enabled NF parallelism for general dependent NFs. However it did not consider the optimization of parallel NF placement.

The problem considered in this paper is also different from the category of scheduling problems for parallel dependent jobs with multiprocessors [32]. In those problems, each processor can only process one job at a time (no matter whether the job is preemptive or non-preemptive). Also, there is no constraint on where to process a job: it can be processed at any processor. This leads to substantial difference in problem analysis and solution.

7 Conclusion and Discussions

In this paper, we study parallelism and placement of a set of dependent VNFs to realize accelerated network services. However, the problem is NP-hard in nature and it also introduces additional issues such as a large number of duplicated packets and heavy network burden. We propose the Prune and Plant scheme with polynomial computational complexity. It effectively balances (1) the overall delay

performance, and (2) the number of duplicated packets and network burden. In the Prune stage, we prune the original DAG into an SP-graph, which eliminates NP-hardness while maintaining NF parallelism. In the Plant stage, we find the optimal placement of the NFs in the server network, to minimize the overall delay. Finally, prototype based experiments and simulations are conducted to validate our proposed P&P scheme, demonstrating that it significantly outperforms benchmark schemes.

In this paper, our main focus is how to parallelize one set of NFs for one network service. Resource (e.g., bandwidth and computing capacity) is not limited for this single network service. In practical system, however, it will be more interesting to study how to handle parallelizable NFs of multiple network services, competing for the limited resource. This will lead to substantially additional challenge, as NFs may be parallelized in many different ways to adapt the resource limitation. The search space is substantially increased in this situation. We leave this very challenging problem for future study.

Acknowledgments

This work was supported by the Grant from Australian Research Council Discovery Project DP190103710.

References

[1] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 1, pp. 236–262, First Quarter, 2016.

[2] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "NFP: Enabling network function parallelism in NFV," in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2017, pp. 43–56.

[3] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, "Near optimal placement of virtual network functions," in *Proc. IEEE Conf. Comput. Commun.*, 2015, pp. 1346–1354.

[4] Y. Sang, B. Ji, G. R. Gupta, X. Du, and L. Ye, "Provably efficient algorithms for joint placement and allocation of virtual network functions," in *Proc. IEEE Conf. Comput. Commun.*, 2017, pp. 1–9.

[5] H. Feng, J. Llorca, A. M. Tulino, D. Raz, and A. F. Molisch, "Approximation algorithms for the NFV service distribution problem," in *Proc. IEEE Conf. Comput. Commun.*, 2017, pp. 1–9.

[6] Y. Chen, J. Wu, and B. Ji, "Virtual network function deployment in tree-structured networks," in *Proc. IEEE 26th Int. Conf. Netw. Protocols*, 2018, pp. 132–142.

[7] J. Zhang, W. Wu, and J. C. S. Lui, "On the theory of function placement and chaining for network function virtualization," in *Proc. 18th ACM Int. Symp. Mobile Ad Hoc Netw. Comput.*, 2018, pp. 91–100.

[8] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[9] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Commun. Mag.*, vol. 53, no. 2, pp. 90–97, Feb. 2015.

[10] J. Martins *et al.*, "ClickOS and the art of network function virtualization," in *Proc. 11th USENIX Conf. Netw. Syst. Des. Implementation*, 2014, pp. 459–473.

[11] J. Khalid *et al.*, "Paving the way for NFV: Simplifying middlebox modifications using StateAlyzr," in *Proc. 13th USENIX Conf. Netw. Syst. Des. Implementation*, 2016, pp. 239–253.

[12] S. Palkar *et al.*, "E2: A framework for NFV applications," in *Proc. Symp. Operating Syst. Princ.*, 2015, pp. 121–136.

[13] A. Gember-Jacobson *et al.*, "OpenNF: Enabling innovation in network function control," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 163–174.

[14] Y. Zhang *et al.*, "ParaBox: Exploiting parallelism for virtual network functions in service chaining," in *Proc. Symp. SDN Res.*, 2017, pp. 143–149.

[15] H. Mekky, F. Hao, S. Mukherjee, T. V. Lakshman, and Z. Zhang, "Network function virtualization enablement within SDN data plane," in *Proc. IEEE Conf. Comput. Commun.*, May 2017, pp. 1–9.

[16] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *Proc. 14th USENIX Conf. Netw. Syst. Des. Implementation*, 2017, pp. 97–112.

[17] Z. Meng, J. Bi, H. Wang, C. Sun, and H. Hu, "MicroNF: An efficient framework for enabling modularized service chains in NFV," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 8, pp. 1851–1865, Aug. 2019.

[18] J. Liu, Z. Jiang, N. Kato, O. Akashi, and A. Takahara, "Reliability evaluation for NFV deployment of future mobile broadband networks," *IEEE Wireless Commun.*, vol. 23, no. 3, pp. 90–96, Jun. 2016.

[19] Y. Shi, Y. Cao, J. Liu, and N. Kato, "A cross-domain SDN architecture for multi-layered space-terrestrial integrated networks," *IEEE Netw.*, vol. 33, no. 1, pp. 29–35, Jan./Feb. 2019.

[20] B. Ren, D. Guo, Y. Shen, G. Tang, and X. Lin, "Embedding service function tree with minimum cost for NFV-enabled multicast," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 5, pp. 1085–1097, May 2019.

[21] Y. Li, L. T. X. Phan, and B. T. Loo, "Network functions virtualization with soft real-time guarantees," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun.*, 2016, pp. 1–9.

[22] W. Ma, O. Sandoval, J. Beltran, D. Pan, and N. Pissinou, "Traffic aware placement of interdependent NFV middleboxes," in *Proc. IEEE Conf. Comput. Commun.*, May 2017, pp. 1–9.

[23] D. Li, P. Hong, K. Xue, and J. Pei, "Virtual network function placement considering resource optimization and SFC requests in cloud datacenter," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 7, pp. 1664–1677, Jul. 2018.

[24] J. Liu, Y. Shi, L. Zhao, Y. Cao, W. Sun, and N. Kato, "Joint placement of controllers and gateways in SDN-enabled 5G-satellite integrated network," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 2, pp. 221–232, Feb. 2018.

[25] L. Zhao, W. Sun, Y. Shi, and J. Liu, "Optimal placement of cloudlets for access delay minimization in SDN-based Internet of Things networks," *IEEE Internet Things J.*, vol. 5, no. 2, pp. 1334–1344, Apr. 2018.

[26] V. Eramo, E. Miucci, M. Ammar, and F. G. Lavacca, "An approach for service function chain routing and virtual function network instance migration in network function virtualization architectures," *IEEE/ACM Trans. Netw.*, vol. 25, no. 4, pp. 2008–2025, Aug. 2017.

[27] M. Luizelli, D. Raz, and Y. Sa'ar, "Optimizing NFV chain deployment through minimizing the cost of virtual switching," in *Proc. IEEE Conf. Comput. Commun.*, 2018, pp. 2150–2158.

[28] Q. Zhang, Y. Xiao, F. Liu, J. C. S. Lui, J. Guo, and T. Wang, "Joint optimization of chain placement and request scheduling for network function virtualization," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 731–741.

[29] L. Guo, J. Pang, and A. Walid, "Dynamic service function chaining in SDN-enabled networks with middleboxes," in *Proc. IEEE 24th Int. Conf. Netw. Protocols*, 2016, pp. 1–10.

[30] B. Li *et al.*, "ClickNP: Highly flexible and high performance network processing with reconfigurable hardware," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 1–14.

[31] B. Ren, D. Guo, G. Tang, X. Lin, and Y. Qin, "Optimal service function tree embedding for NFV enabled multicast," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 132–142.

[32] V. Sarkar, "Partitioning and scheduling parallel programs for execution on multiprocessors," Ph.D. dissertation, Dept. Electr. Eng., Stanford Univ., Stanford, CA, 1987.

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd Ed., Cambridge, MA, USA: The MIT Press, 2009.

Wei Bao (Member, IEEE) received the BE degree in communications engineering from the Beijing University of Posts and Telecommunications, Beijing, China, in 2009; the MASc degree in electrical and computer engineering from the University of British Columbia, Vancouver, Canada, in 2011; and the PhD degree in electrical and computer engineering from the University of Toronto, Toronto, Canada, in 2016. He is currently a senior lecturer with the School of Computer Science, the University of Sydney, Sydney, Australia. His research covers the area of network science, with particular emphasis on Internet of Things, mobile computing, and edge computing. He received the best paper awards in ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM), in 2013 and 2019 and IEEE International Symposium on Network Computing and Applications (NCA), in 2016.

Dong Yuan (Member, IEEE) received the BEng and MEng degrees from Shandong University, Jinan, China, in 2005 and 2008, respectively, and the PhD degree from Swinburne University of Technology, Melbourne, Australia, in 2012, all in computer science. He is a senior lecturer with the School of Electrical and Information Engineering, the University of Sydney, Sydney, Australia. His research interests include cloud computing, parallel and distributed systems, scheduling and resource management, deep learning, data management and Internet of Things.

Bing Bing Zhou received the graduate degree in electronic engineering from the Nanjing Institute of Technology in China, in 1982, and the PhD degree in computer science from Australian National University, Australia, in 1989. He is an associate professor with the School of Computer Science, the University of Sydney, Australia (2003-present). Currently, he is the theme leader for distributed computing applications in the Centre for Distributed and High Performance Computing, University of Sydney.

Albert Y. Zomaya (Fellow, IEEE) is currently the chair professor of High Performance Computing and Networking with the School of Computer Science, University of Sydney, Sydney, Australia. He is also the director of the Centre for Distributed and High Performance Computing. He has published more than 600 scientific papers and articles and is author, co-author or editor of more than 20 books. He is the founding editor-in-chief of the *IEEE Transactions on Sustainable Computing* and previously he served as editor-in-chief for the *IEEE Transactions on Computers* (2011–2014). Currently, he serves as an associate editor for 22 leading journals, such as, the *ACM Computing Surveys*, *IEEE Transactions on Cloud Computing*, and *ACM Transactions on Internet Technology*. He delivered more than 190 keynote addresses, invited seminars, and media briefings and has been actively involved, in a variety of capacities, in the organization of more than 700 conferences. He is the recipient of the IEEE Technical Committee on Parallel Processing Outstanding Service Award (2011), IEEE Technical Committee on Scalable Computing Medal for Excellence in Scalable Computing (2011), IEEE Computer Society Technical Achievement Award (2014), ACM MSWIM Reginald A. Fessenden Award (2017), and New South Wales Premiers Prize of Excellence in Engineering and Information and Communications Technology (2019). He is a chartered engineer, a fellow of AAAS, IEEE, IET (United Kingdom), an elected member of Academia Europaea, and an IEEE Computer Societys Golden Core member. His research interests include parallel and distributed computing, networking, and complex systems.

Page Reusability-Based Cache Partitioning for Multi-Core Systems

Jiwoong Park, Heonyoung Yeom, and Yongseok Son

Abstract—Most modern multi-core processors provide a shared last level cache (LLC) where data from all cores are placed to improve performance. However, this opens a new challenge for cache management, owing to cache pollution. With cache pollution, data with weak temporal locality can evict other data with strong temporal locality when both are mapped into the same cache set. In this article, we propose page reusability-based cache partitioning (PRCP) for multi-core systems to maximize cache utilization by minimizing cache pollution. To achieve this, PRCP divides pages into two groups: (1) highly-reused pages and (2) lowly-reused pages. The reusability of each page is collected online via periodic page table scans. PRCP then dynamically partitions the shared cache into two corresponding areas using page coloring technique. We have implemented PRCP in Linux kernel and evaluated it using SPEC CPU2006 benchmarks. The results show that our scheme can achieve comparable performance to the optimal offline MRC-guided process-based cache partitioning scheme without a priori knowledge of workloads.

Index Terms—Cache partitioning, page coloring, last-level cache, LRU replacement policy, multi-cores

1 Introduction

Most modern multi-core processors support a shared last level cache (LLC) in which data are placed from all cores [1], [2]. An LLC is one of the most important shared resources, owing to its impact on performance as a consequence of retaining a substantial amount of data on-chip. To manage shared LLCs, modern systems use a variant of the least recently used (LRU) policy. This policy favors cache access recency or temporal locality. Thus, it provides good performance when workloads have high temporal locality. However, when workloads have poor temporal locality or when the working set size of the workloads cannot fit in the cache, cache pollution occurs [3]. With cache pollution, more useful cache lines can be replaced by less useful ones. This pollution can seriously degrade the overall system performance, even when there is only one cache-polluting workload, owing to inter-process interference.

Cache partitioning has been considered one of the solutions to mitigate performance degradation caused by cache pollution; it partitions a shared cache among different pages, objects, processes, or cores. Cache partitioning techniques can fall into two categories [4]: software-based and hardware-based partitioning. All software-based cache partitioning schemes [3], [5], [6], [7], [8], [9], [10], [11], [12] rely on a page coloring technique, which is based on the fact that a single page is mapped into a subset of cache sets in the physically-indexed LLC. On the other hand, hardware-based cache partitioning schemes rely on either a way-based cache partitioning capability of recent commodity processors [2], [13], [14], [15], [16], [17]—e.g., Intel's cache allocation technology (CAT)—or a special hardware feature that is not available in existing hardware [18], [19], [20], [21], [22], [23].

- *J. Park and H. Yeom are with the Department of Computer Science and Engineering, Seoul National University, Seoul 08826, South Korea. E-mail: {pjbear, yeom}@snu.ac.kr.*
- *Y. Son is with the School of Computer Science and Engineering, Chung-Ang University, Seoul 06974, South Korea. E-mail: sysganda@cau.ac.kr.*

Manuscript received 3 Apr. 2019; revised 7 Jan. 2020; accepted 15 Jan. 2020. Date of publication 21 Jan. 2020; date of current version 8 May 2020.
(Corresponding author: Yongseok Son.)
Recommended for acceptance by L. A. Sousa.
Digital Object Identifier no. 10.1109/TC.2020.2968066

Software-based cache partitioning has several advantages over hardware-based one. First of all, there is no hardware requirement. Moreover, it enables finer-grained (page-level) control in terms of which parts of data are isolated from the others, than either process-, or core-level. Therefore, software-based object- or page-level cache partitioning can mitigate intra-application cache interference, which cannot be addressed with hardware process-level way-based cache partitioning solutions. Our study is in line with previous software-based approaches in terms of reducing cache pollution. By contrast, we focus on simplifying the partitioning decisions and maximizing system-level cache efficiency while providing weaker performance isolation among processes.

In this paper, we propose a page reusability-based cache partitioning (PRCP), a software-based approach that maximizes cache utilization. Our key idea is to divide all the processes' working sets into two parts based on the page reusability, and to partition a cache into two corresponding areas. Fig. 1 shows the overview of our approach. PRCP involves four steps as follows. 1) PRCP performs online page-level profiling that involves periodic scans of the running processes' page tables to obtain the reusability of each page. 2) PRCP dynamically classifies all pages of the processes into two groups (viz., low reusability and high reusability) based on the reusability. 3) PRCP divides the LLC into a low reusability area (LRA) and a high reusability area (HRA) through page coloring, keeping the LRA size as small as possible. 4) PRCP then migrates the target pages into the appropriate area, enabling efficient cache partitioning.

We implemented PRCP in Ubuntu 16.04 Linux OS, kernel version 5.0.0, on a real multi-core system with four physical cores. Our experimental results with SPEC CPU2006 benchmarks show that the proposed PRCP improves the application performance by up to 12 percent (6 percent on average) compared to the native Linux kernel. These results demonstrate that our system is comparable to a process-based optimal static cache partitioning system, even though we use a dynamic approach.

The contributions of this study are as follows.

- We built a color-aware page allocator on top of the existing OS page management mechanism in the Linux kernel, avoiding wasted memory resources with consideration of multi-core scalability.
- We designed and implemented PRCP that simplifies the decision regarding the cache partitioning policy, which is the most challenging issue with process-based cache partitioning.
- We evaluated our scheme with SPEC CPU2006 benchmarks on an x86 server. The results indicate that PRCP enhances the overall system performance as much as using optimal MRC-guided process-based static cache partitioning without any a priori knowledge, or hardware support.

2 Background and Related Work

2.1 Page Access Monitoring

Monitoring access for individual pages at kernel level is possible with page table scanning. There is an *accessed bit* in each page table entry (*PTE*). In the x86 architecture, the *accessed bit* is set to 1 by hardware whenever it uses a *PTE* as part of the linear-address translation. In the Linux kernel, a kernel swap daemon called *kswapd* periodically checks this bit and then clears it to see which pages have not been used recently. We can adopt a similar approach to monitoring page accesses by clearing and checking this *accessed bit* with periodic page table scanning.

2.2 Software-Based Cache Partitioning

Software-based cache partitioning scheme rely on page coloring technique to partition the LLC into sets [24]. It exploits the fact that

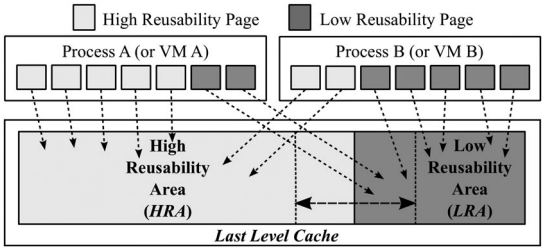


Fig. 1. The overview of PRCP.

the LLC is physically indexed, such that data are mapped into cache sets with respect to their physical addresses; a set of cache is called a *color*. Two pages with different colors cannot be mapped into the same cache area. An operating system can leverage this to restrict the accessible cache area for each process by controlling logical-to-physical page mapping. Fig. 2 shows the physical mapping between physical memory and the LLC in an Xeon E5606 processor with an 8 MB 16-way L3 cache. Note that page coloring technique fundamentally does not work with huge pages [25], because a single huge page, whose size is generally 2 MB, can cover all the sets of cache. However, we do not think that this is the severe limitation of the page coloring-based scheme because the use of huge page is only beneficial for applications that frequently access large amount of memory—those applications do not benefit from cache memory anyway due to the low cache hit ratio. Thus, applying page coloring only on normal pages (4 KB) does not have to compromise application performance.

Process/Core-Level Cache Partitioning: To avoid cache interference among processes(cores), most previous studies [5], [6], [7], [8], [9], [10] have chosen a process/core-based approach where a process or a core can access only a certain portion of the LLC. Tam *et al.* [5] demonstrated the benefits of the statically partitioned shared cache among processes using page coloring.

In contrast, there have been several works [6], [10] that proposed dynamic cache partitioning schemes where the cache partition allocated for each process can be dynamically adjusted. To search for the best partitioning configuration, they used a trial-and-error approach, which has several shortcomings as follows. First, ample time is needed to find the optimal partitioning configuration among the processes. Second, it may not reach the optimal point due to a local maximum scenario. On the other hand, our approach simplifies the cache partitioning policy decision by providing the low-reusability group with the least amount of cache possible.

Some studies [7], [9] have presented hardware performance counter-based online MRCs to find the optimal cache partitioning point among processes. However, these are only applicable to process-level cache partitioning and cannot address cache interference within an application. Zhang *et al.* [8] presented several optimization techniques to reduce overhead from page table scans and page migrations. We can adopt their optimization techniques to minimize monitoring and recoloring overhead.

Our study is in line with the above studies in terms of software cache partitioning, but differs from them in several important ways. First, PRCP is a page-level cache partitioning scheme. As such, PRCP provides finer-grained control over which parts of data are isolated from the others, than object-, process-, or core-level. Second, unlike approaches that enforce strict performance isolation among processes, PRCP provides relaxed performance isolation based on the fact that, for some applications, the cache capacity might be more important than the dedication of cache resources.

Object/Page-Level Cache Partitioning: Soft-OLP [11] is an object-level cache partitioning scheme based on the reuse distance of data objects. In contrast to our scheme that uses online profiling, it requires offline profiling of each application to obtain the per-object reuse distance histograms and inter-object interference histograms.

Soares *et al.* [3] first introduced the concept of a software-only pollute buffer to address intra-process cache pollution. For page-level profiling, they used a *sampled-data address register (SDAR)*, which is only available for PowerPC. In contrast, our scheme does not rely on specific hardware feature. NightWatch [12] is a cache management subsystem that protects the cache from being polluted by *cache polluters*. To search for the potential cache polluters with low overhead, it leverages locality similarity within the same memory chunk and between chunks in the same allocation context. It only requires the hardware performance counter available on commodity processors to predict the reuse distance of the memory chunks.

However, the above methods [3], [12] use a fixed-size pollute buffer, which can result in memory pressure even with sufficient system memory. The memory size for each color is restricted to one-*n*th of the system memory when there are a maximum of *n* colors available in the system. In other words, if the system runs out of the color requested from the process, the process can be under memory pressure despite having enough pages for other colors. Moreover, NightWatch [12] may aggravate this problem because it makes normal chunks share the cache with pollute chunks by allowing the use of all colors.

Complex Hash-Based LLC Addressing: Page coloring was originally devised based on the assumption that the several bits of a physical address can be directly used to locate the specific set of cache. However, modern processors use complex hash functions for mapping physical memory address to cache location due to security reasons as well as reducing conflict misses [26], [27]. Although their hash functions have not been disclosed by vendors, several studies [26], [27] have revealed that they can be obtained by reverse engineering. Some works [28], [29] have implemented page coloring systems with consideration of the complex hash functions and have demonstrated that they could deliver the same capability of performance isolation as the traditional page coloring system. In this paper, we do not consider the complex hash function because the processor in our target system uses a simple hash function. However, we believe that our scheme can be ported to other modern processor-based systems with a similar engineering effort, as done in [28], [29]. We leave for future work exploring the performance impact of complex hashing on our scheme, whose results may vary because both complex hashing and our scheme impact the cache miss behavior.

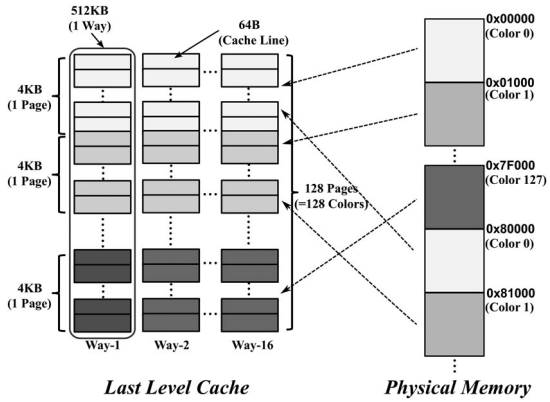


Fig. 2. Physical mapping between physical memory and the last level cache in an Xeon E5606 processor with an 8 MB 16-way L3 cache that uses a simple hash function.

2.3 Hardware-Based Cache Partitioning

Recent multicore server processors provide a way-based cache partitioning feature [2], [13], [14], [15], whose partitioning unit is a subset of ways in LLC. Thus, the partition granularity tends to

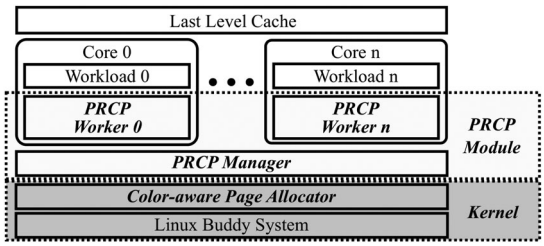


Fig. 3. PRCP system architecture.

be coarser than set-based cache partitioning schemes. Some studies have focused on tackling this issue while utilizing the way-partitioning feature. SWAP [16] have chosen a hybrid approach that combines the hardware-based way-partitioning and page coloring-based set-partitioning to achieve finer-grain partitions. On the other hand, KPart [17] addresses the problem by grouping applications into a few groups and allowing sharing of cache partition among a group of applications. However, because a basic way-partitioning provided by commodity server is also process-level cache partitioning, it inherits the limitation of the process-level approach. For instance, it requires a way of determining the optimal partitioning policy among processes, and remains intra-application cache interference unsolved.

To partition a large shared cache in a more efficient and fine-grained way, many alternative hardware schemes [18], [19], [20], [21], [22], [23] have been proposed. With special hardware supports, their approach can eliminate the huge re-partitioning overhead, which is inevitable for page coloring-based solutions. However, they are not available in existing hardware. By contrast, our solution is a software-only solution that can be readily deployed even in hardware that does not provide basic way-partitioning.

3 Design and Implementation

In this section, we describe the design and implementation of the proposed scheme. As shown in Fig. 3, our scheme consists of three kernel components as follows: (1) a *two-level color-aware page allocator* that serves specific colored page requests, (2) several *PRCP workers* that monitor page accesses, create statistics, and perform page migration, and (3) a *PRCP manager* that coordinates the *PRCP workers* and makes a global decisions.

3.1 Two-Level Color-Aware Page Allocator

There is a physical memory allocator in the Linux kernel for each zone, called the *buddy system*, where free pages are grouped into 1, 2, 4, 8,..., 1024 contiguous pages [30]. Since the *buddy system* is a globally shared kernel component, it is protected by a lock, which may introduce a sequential bottleneck when multiple processes concurrently allocate memory. To avoid lock contention, the Linux kernel additionally maintains a per-CPU page frame cache (PCP) per core. The pages are pulled in and out in a batched manner, from and to the *buddy system*.

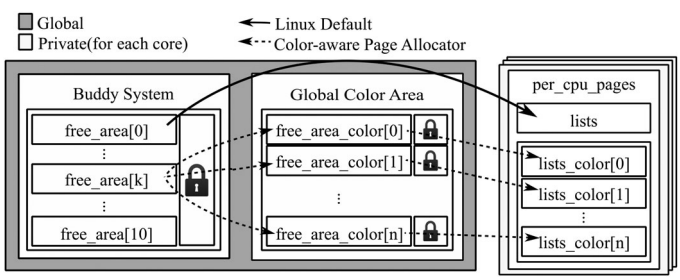


Fig. 4. Two-level color-aware page allocator.

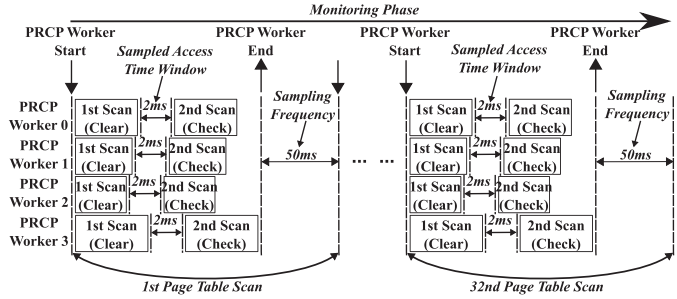


Fig. 5. Monitoring page access.

As shown in Fig. 4, our two-level color-aware page allocator is built on top of the PCP and the *buddy system*. For efficient color refilling, we add a *global color area*, which is shared among cores, between the *buddy system* and the PCP. Basically, we leverage the fact that a sufficiently large block of contiguous pages includes all page colors. For example, a block of 64 contiguous pages has 64 different page colors. If one of the *global color lists* is empty, our allocator requests order-N pages from the *buddy system* and then splits them to refill all color lists at once, rather than requesting single pages repeatedly. Each color list in the PCP can be refilled from the global color lists in a fine-grained manner. When refilling the global color lists from the *buddy system*, the allocator needs a global lock for the zone to protect the race condition. Unlike the global color list, refilling the local color list does not require a global lock for the zone, but rather a global color lock. Thus, distinct color requests from multiple cores can be handled concurrently.

3.2 Monitoring Phase

In the monitoring phase, the *PRCP manager* is responsible for synchronizing the start time of the page table scan work assigned to each *PRCP worker*, ensuring that the page access information is gathered in the same period.

Monitoring Page Access: As a metric for reusability, we use the number of accesses to a page within a time window. Our basic assumption is that rarely accessed pages within a given interval are unlikely to be subsequently reused in cache, owing to temporal locality. Thus, they have low reusability.

Fig. 5 shows the events that occur during the monitoring phase. First, the *PRCP manager* sends the signal to the *PRCP workers* to wake them up. After receiving the signal, the *PRCP workers* start the monitoring phase. During the monitoring phase, the *PRCP workers* repeat 32 monitoring tasks to collect enough data to be statistically meaningful. Each monitoring task involves two page table scans: one for clearing the *accessed bit* in the *PTE* and the other for checking if the bit has been set. After the first page table scan, the *PRCP workers* go to sleep for the *sampled access time window*, during which the target process on the same core can run. After the *sampled access time window* passes, *PRCP workers* are awoken to scan and check whether the page has been accessed during the time window. If so, the *PRCP worker* increments the *access count* for the page, which we add in the kernel page structure to record the access frequency for each page within a monitoring phase. At the end of the monitoring task, each *PRCP worker* goes to the sleep until awoken by the *PRCP manager*. After receiving the notification from the last *PRCP worker* that it has done its job, the *PRCP manager* is woken up and goes to sleep for the *sampling frequency* before starting a next monitoring task. We have empirically determined the best number of repeats of the monitoring task in a phase, the best *sampled access time window*, and the best *sampling frequency* (32, 2 ms, and 50 ms, respectively).

3.3 Decision Phase

In the decision phase, the *PRCP manager* plays a role determining the global policy based on the collected information from *PRCP*

workers. When creating statistics, the *PRCP manager* excludes the information from pages whose access count is zero, because they are not likely to be accessed in the near future thus no impact on cache performance.

Creating Local Statistics: When scanning the pages in the monitoring phase, each *PRCP worker* produces local statistical data from the page access information. The statistics include a *local access count table (LACT)*, a *local page distribution table (LPDT)*, and *local non-accessed low reusability area pages*. The *LACT* is an array that stores the total number of pages for each access count value. For instance, the value of the first index in the array is the number of pages that have been accessed only once during the monitoring phase. The *LPDT* stores the total number of pages that currently belong to the HRA and LRA, respectively. *Local non-accessed LRA pages* refer to the number of non-accessed pages in the LRA. Having produced the local statistical data, the *PRCP worker* informs the *PRCP manager* of its job completion.

Creating Global Statistics: The *PRCP manager* creates the global statistics by collecting the local statistics provided by the *PRCP workers*. The global statistics comprise a *global access count table (GACT)*, a *global page distribution table (GPDT)*, and *global non-accessed LRA pages*. As comprehensive information, these global statistics help to make better decisions.

Threshold Decision: As a threshold to split the working set of processes into two groups, we use the average of non-zero *access counts* of all the pages that belong to the target processes. Therefore, the threshold can be dynamically adjusted according to the workloads currently running. Given the threshold, the pages whose *access count* value is lower than the threshold are considered as *LRA pages*. We have also considered the median of them but have found no significant performance difference. We leave for future work the study of algorithms to find the optimal threshold.

Partitioning Policy Decision: After deciding the threshold, the *PRCP manager* makes a decision how much portion of cache space is allocated for HRA and LRA, respectively. Previous studies based on page coloring [3], [12] do not consider the memory pressure problem, which is described in Section 2.2. They assume that the system has sufficient pages for each color to fulfill memory requests from processes. However, this is not the case when an application having a large memory footprint are restricted to using only a few colors. Since there are a limited number of pages for a specific color, too many page allocation requests for a few colors may lead to out-of-memory errors.

We address this problem by dynamically changing the size of the LRA, namely the number of colors allocated for the LRA. Our principle for the partitioning policy is simple yet efficient: keeping the size of the LRA as small as possible while ensuring that no memory pressure imposed by page coloring occurs. To do so, the *PRCP manager* predicts *worst-case LRA pages*: namely, the number of pages that will be in the LRA in the next epoch in the worst case. This can be calculated by summing the number of non-accessed pages that are currently in the LRA and the number of accessed pages that will be in the LRA after page migration. The former can be obtained from the *global non-accessed LRA pages*, and the latter can be derived in the partitioning policy decision phase. Given the *worst-case LRA pages*, the *PRCP manager* can find the minimum number of colors for the LRA, such that the system can allocate pages without suffering memory pressure, exploiting the number of pages that the system can allocate for a color, which is computed as follows:

$$NR\_PAGES\_PER\_COLOR = (TotalMemoryCapacity/PageSize)/NR\_COLORS. \quad (1)$$

For instance, our target system with total 128 colors, 6 GB memory, and 4 KB-sized pages has 12,288 pages available per color.

For optimal performance, the *PRCP manager* gives the minimum amount of cache to the LRA while allocating the remaining cache to the HRA, such that highly reusable pages utilize the larger cache. After applying the new partitioning policy, the *PRCP manager* forces the *PRCP workers* to start the action phase.

3.4 Action Phase

Page Migration: In the action phase, each *PRCP worker* migrates the misplaced pages to the right area. While scanning the page table of the process, the *PRCP worker* compares the access count of the pages to the threshold to check if they need to be migrated to the opposite area. For example, the pages whose color belongs to the LRA are tested if their *access count* is smaller than the threshold but larger than zero, while the pages whose color belongs to the HRA are tested if their *access count* is larger than the threshold. Only the pages that meet the above migration condition are considered as the candidates for page migration. To avoid too many transitions from the LRA to the HRA and vice versa, we employ the *third-chance* algorithm, which gives pages two more chances before making a transition between two areas. We have also added a new page migration mechanism that allows the migration of a page to a new page with the requested color.

Color Balancing: When the partitioning policy changes, the *PRCP worker* tries to rebalance the colors in a round-robin manner. If the size of the HRA increases and PRCP only migrates pages that are currently wrongly placed, there will be a skewed page distribution for each color in the HRA. In this case, newly allocated colors owing to the increased HRA size are not utilized at all. The same occurs with an increased LRA size. Therefore, all the pages that belong to the LRA or HRA should be rebalanced to take advantage of the increased cache partition size.

3.5 Resting Phase

Generating Dynamic Delay: If all target processes are in a stable state, only few page migrations will occur in future epochs. In this case, the *PRCP manager* generates a longer time delay before starting a new epoch to avoid unnecessary overheads from page monitoring. PRCP considers the current state to be *stable* when the number of pages to be migrated is lower than a *delay threshold*. We heuristically determine this threshold, by calculating the total number of pages of the target processes divided by the number of colors available in the system.

For the *stable* state, the *PRCP manager* exponentially increases the delay time between epochs by multiplying the *delay weight* by two. The *delay time* is computed as *delay time* = *delay unit* $\times$ *delay weight*. The *delay unit* is 3.2 s by default, which is approximately as long as two epochs, except for the resting phase. We set the *default delay weight* and the *maximum delay weight* to 4 and 32, respectively. However, because the workload phase can be changed in the meantime, the *PRCP manager* initializes the *delay weight* to the default value whenever it observes that there have been as many page migrations as the *delay threshold*. After adjusting the delay time, the *PRCP manager* goes to sleep for the *delay time*.

4 Evaluation

4.1 Environmental Setup

Experimental Environment: We have implemented a prototype of the proposed PRCP system for Ubuntu 16.04 Linux OS, kernel version 5.0.0. We conduct a series of experiments to evaluate the effectiveness, performance, and overhead of PRCP on an Xeon E5606 processor (2.13 GHz)-based server machine with four physical cores, 6 GB DDR3 memory, an 8 MB 16-way L3 cache shared by the four cores, and a maximum of 128 colors. Note that we set a relatively small capacity of memory to show that PRCP can overcome the memory pressure problem imposed by page coloring, considering the number of pages available per color in the target machine. We disable transparent huge pages (THP).

TABLE 1
Benchmark Classification

Groups	Benchmarks (Memory Footprint)
I. Cache Fitting	*astar (313 MB), bzip2 (856 MB)*
II. Cache Friendly	*sphinx3 (48 MB), xalancbmk (340 MB)*
III. Cache Polluting	*libquantum (104 MB), leslie3d (129 MB)*

Methodology: Except for the benchmark classification, we concurrently execute a mix of four benchmarks, assigning a different core to each benchmark with help of the *numactl* command. We repeatedly re-run the benchmarks until all four benchmarks are completed at least once, and then we measure the execution time of each benchmark for the first run. Note that by this way, the performance of the benchmark that takes the longest time to finish is always interfered by the other three benchmarks during whole run time. We repeat the evaluation five times and report the average result.

4.2 Benchmark Classification

We have selected SPEC CPU2006 for the evaluation. We use *ref* as input data. For cache sensitivity analysis, we measure the execution time of each benchmark from SPEC CPU2006 when running alone, varying the allocated cache size using the page coloring technique. On the basis of the cache sensitivity, we classify all benchmarks into three groups: *cache friendly*, *cache fitting*, and *cache polluting*. A set of benchmarks are categorized to the *cache polluting* type when their performances are insensitive to the cache size while having a high cache miss rate. By contrast, we categorize a set of benchmarks as the *cache fitting* type when their performance improves as the size of the allocated cache increases but no further gains are possible after a certain point. The *cache friendly* type includes the benchmarks whose performance is sensitive to the cache size; the more cache the process has, the shorter it takes to finish a run. We choose two benchmarks from each group. The results of this grouping are shown in Table 1 while Fig. 6 shows the experimental results of the selected benchmarks, which can be used as offline LLC miss rate curves (MRCs) to help find the best

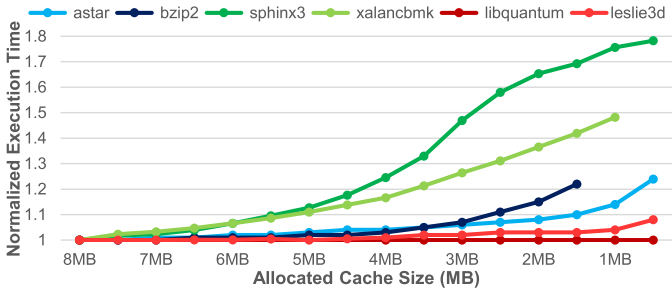


Fig. 6. Normalized execution time of SPEC CPU2006 benchmarks with varying cache size.

static cache partitioning configuration. Note that there are no results for *bzip2* and *xalancbmk* when their allocated cache size is less than 1.5 MB or 1 MB respectively because they failed to run due to the memory pressure caused by page coloring.

4.3 Performance Comparison

To demonstrate the effectiveness of our scheme, we run mixes of four benchmarks from the six benchmarks listed in Table 1 while comparing the performance of the following two cache partitioning approaches: PRCP, which is our approach, and MRC, optimal static cache partitioning driven by offline MRCs, which are obtained in Section 4.2. For process-level static cache partitioning, we have also implemented a kernel mechanism that allows assigning the colors to a process when executing the process. We use the performance of default Linux with non-partitioned LLC as our comparison base. Fig. 7 shows the experimental results.

When there is no *cache polluting* benchmark in a mix of four workloads (Fig. 7a), default Linux performs sufficiently well, even though all processes share the LLC. MRC performs worse than the baseline; the performance of both *bzip2* and *sphinx3* improves by 6 percent while the performance of *astar* and *xalancbmk* decreases by 4 and 9 percent, respectively. In contrast, PRCP preserves the performance of the default Linux. This is because for such cases, the allocated cache capacity becomes more important than the private use

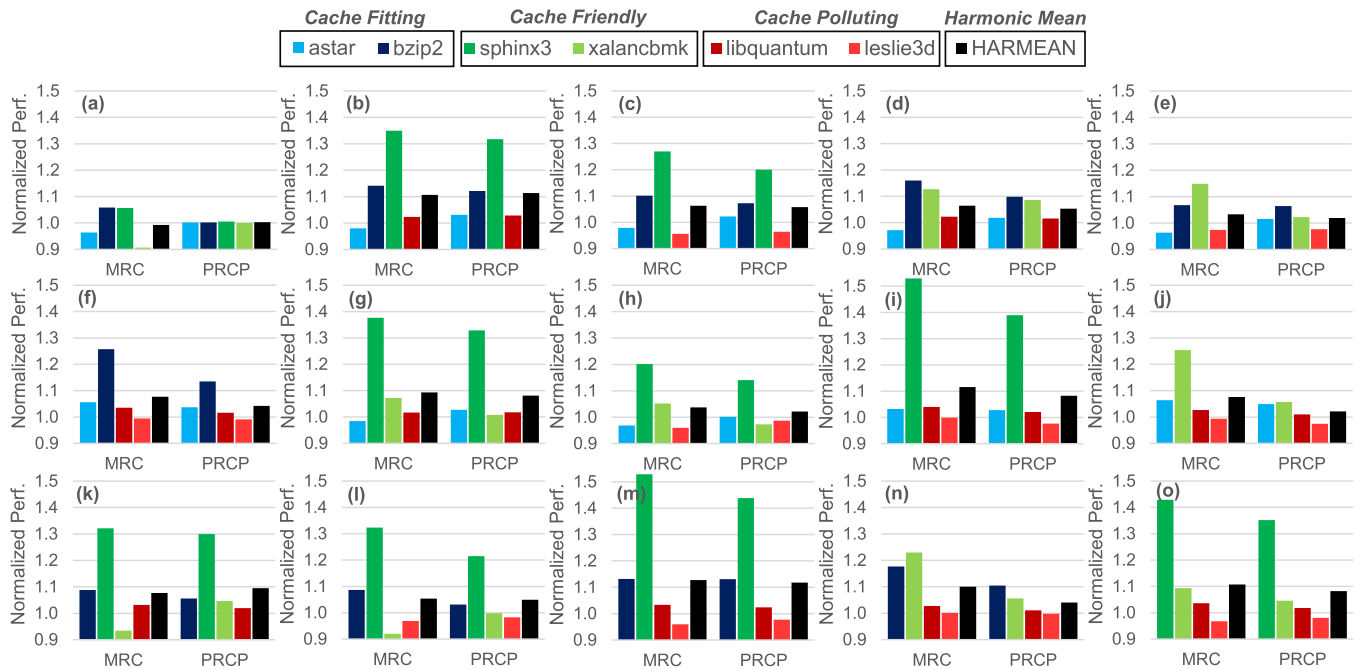


Fig. 7. Comparison of the performance of two cache partitioning schemes, offline MRC-based static cache partitioning (MRC) and our scheme (PRCP). Y-axis is normalized performance to the naive Linux with non-partitioned LLC. Each subfigure represents the performance of a mix of four benchmarks. We also include the harmonic mean of the normalized performance for each mix of four benchmarks.

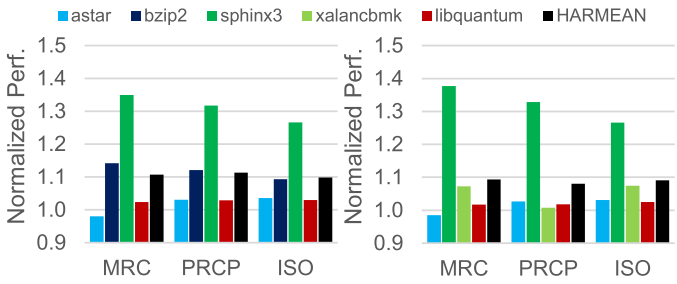


Fig. 8. Comparison of the performance of three cache partitioning schemes, offline MRC-based static cache partitioning (MRC), our scheme (PRCP), and a scheme that isolates the polluting process from others (ISO). Y-axis is normalized performance to the naive Linux with non-partitioned LLC. Each subfigure represents the performance of a mix of four benchmarks. We also include the harmonic mean of the normalized performance for each mix of four benchmarks.

of cache—the total cache capacity is not enough to be partitioned for those *cache friendly* and *cache fitting* workloads. Therefore, strict performance isolation of MRC leads to degraded performance.

One may notice that the performance of some applications (*sphinx3*) tends to improve more than others (*leslie3d*). This is due to the cache sensitivity of the applications. For example, *sphinx3* is the most cache-sensitive among the six benchmarks. Thus, for overall performance, allocating as much cache space to *sphinx3* as possible is the most effective. MRC does so thus it degrades the performance of co-running benchmarks, as shown in Figs. 7a, 7c, 7e, 7h, 7k, 7l, 7m, and 7o. In contrast, PRCP achieves the similar performance improvement in a more balanced way. However, PRCP performs especially worse than MRC for some cases in Figs. 7j and 7n. This is because in the PRCP's point of view, the pages from *xalancbmk* are not distinguished with the pages from *astar* and *bzip2*. We can use the hardware performance counter to supplement the reusability metric (*access count*). We leave this for future work.

As shown in Figs. 7b, 7c, 7d, 7e, 7f, 7g, 7h, 7i, 7k, 7l, 7m, and 7o, PRCP provides the comparable performance to MRC; MRC improves the performance by up to 13 and 7 percent on average while PRCP improves the performance by up to 12 and 6 percent on average. Note that offline MRC-guided static cache partitioning (MRC) do not incur the profiling overhead and the re-partitioning overhead. Even without a priori knowledge, modifications to the application code, or special hardware support, PRCP reaches the optimal performance of process-based static cache partitioning.

Although PRCP leaves the cache contention problem among high-reusability pages in the *HRA* as it stands, PRCP enhances the overall system performance, as much as MRC. This means that exclusively isolating the polluting pages from other pages is enough for improved performance insofar as it simplifies the partitioning policy decision. One may think isolating the polluting applications from others—this can be implemented with Intel's CAT—will lead to the same result. Fig. 8 shows the performance comparison of the scheme (ISO) that isolates the polluting process from other processes with MRC and PRCP. Note that we already know which application is the *cache polluting* type. We can statically partition the cache into two area; 16 colors for *cache polluting* applications and 112 colors for other applications. As shown in the figure, PRCP shows slightly better performance than ISO although PRCP involves page access monitoring and page re-coloring overheads. In reality, this gap becomes wider because even CAT-based solutions require profiling of the workloads and a decision time to partition the cache. Thus, this demonstrate that the page-level cache partitioning is superior to the process-level one.

4.4 Dynamicity of PRCP

To show the dynamicity of PRCP, we have extracted some information from one mix of workloads, namely, page distributions, expected *LRA* size, and the threshold. Table 2 shows the results. Note that the number of pages in Table 2 includes the non-accessed pages within an epoch. At first, the proportion of *LRA* pages to *HRA* pages is nearly 1 to 127, because the number of *LRA* colors is 1 by default. With time, the size of the *LRA* grows and shrinks depending on the threshold and page distribution. The predicted LRA size can differ from the actual LRA size, insofar as we use the *third-chance* algorithm to avoid unnecessary page migrations caused by inaccurate sampling-based page table scans.

Table 2 indicates that *astar* and *bzip2* are both multi-phase workloads, whereas *sphinx3* and *libquantum* are single-phase workloads. Owing to our design choice—whereby makes each epoch starts from the scratch—the termination and restart of workloads, even with multi-phase workloads, do not negatively impact the policy decision of PRCP. However, when restarting the process, all the page information for the *three-chance* algorithm is initialized. Therefore, there are inaccuracies of the predicted *LRU* size for several future epochs. At 1,217 s in Table 2, *sphinx3* ended just before the decision phase; Thus, PRCP made policy decisions based on the statistics of the remaining three processes. By contrast, in the next epoch when *sphinx3* restarted, PRCP worked well again for the four processes.

4.5 Overhead

We investigate the overhead of PRCP, breaking it down into four parts according to the respective phases of PRCP. Table 3 shows the runtime overhead of PRCP for each phase when running a mix of *astar*, *bzip2*, *sphinx3*, and *libquantum*. Note that the results do not include the context switch overhead. However, because the *PRCP manager* and *workers* are unscheduled most of the time, we do not think that the context switch will incur noticeable overhead. The

TABLE 2
Dynamicity of PRCP

Time (sec)	NR HRA Pages					NR LRA Pages					Expected LRA Pages	THLD
	astar	*bzip2*	*sphinx3*	*libq*	Total	*astar*	*bzip2*	*sphinx3*	*libq*	Total		
15	82,704	214,499	10,217	16,447	323,867	660	1,690	84	131	2,565	14,717	7
43	82,053	212,810	10,137	16,317	321,317	1,311	3,379	165	261	5,116	46,905	6
97	6,952	205,257	9,738	15,669	237,616	412	11,875	571	909	13,767	27,276	10
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
314	57,181	24,484	8,826	8,183	98,674	2,598	586	1,567	8,395	13,146	41,356	10
431	6,922	205,652	8,646	7,987	229,207	340	10,111	1,796	8,591	20,838	21,631	14
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1200	6,793	22,729	7,714	7,858	45,094	469	1,522	3,256	8,720	13,967	14,645	13
1217	7,140	23,870		9,717	40,727	122	381		6,861	7,364	8,885	5
1233	7,140	24,675	10,138	8,618	50,571	122	395	164	7,960	8,641	11,192	8
1261	7,115	23,521	10,139	8,283	49,058	147	377	164	8,295	8,983	14,060	19
1314	1,038	212,810	10,147	8,278	232,273	24	3,379	164	8,300	11,867	14,486	9

TABLE 3
Overhead of PRCP

Mon.	Dec.	Act.	Rest.	Total
0.3(0.7)%	0.02(0.04)%	0.04(0.14)%	0.01(0.04)%	0.35(0.92)%

most significant overhead originates from the monitoring phase because in this phase, a sequential page table scan occurs 64 times within about 1.6 seconds. Note that to check the page access, two sequential page table scans are required: one for clearing the *access bit*, and the other for checking it. Overall, the average total overhead costs only 0.35 percent (and up to 0.92 percent) of the execution time of the benchmark.

The overhead is rather trivial compared to the performance improvement that results from PRCP. This is due to the time delay between epochs, which can be at least 12.8 seconds and at most 102.4 seconds long. From these results, we conclude that the overhead required for PRCP is almost negligible.

5 Conclusion

We present PRCP to maximize LLC utilization in a multi-core system. PRCP simplifies the decision regarding the cache partitioning policy by classifying all pages of co-running user processes into two groups based on page reusability. It then gives the low-reusability group the smallest possible portion of the cache via page coloring. PRCP can achieve the performance of optimal process-based static cache partitioning without additional hardware support, application changes, or a priori knowledge of workloads.

Acknowledgments

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (2015M3C4A7065646, 2016M3C4A7952587). This work was also supported by NRF (MSIT) (2018R1C1B5085640).

References

[1] Intel Corporation, "Intel (R) 64 and IA-32 Architectures Software Developer's Manual," 2016. [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm. Accessed: Feb. 12, 2020.
[2] Advanced Micro Devides, "BIOS and Kernel Developer's Guide (BKDG) for AMD family 15h models 00h-0Fh processors," 2013. [Online]. Available: https://www.amd.com/system/files/TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf. Accessed: Feb. 12, 2020.
[3] L. Soares, D. Tam, and M. Stumm, "Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer," in *Proc. 41st Annu. IEEE/ACM Int. Symp. Microarchit.*, 2008, pp. 258–269.
[4] S. Mittal, "A survey of techniques for cache partitioning in multicore processors," *ACM Comput. Surv.*, vol. 50, no. 2, 2017, Art. no. 27.
[5] D. Tam, R. Azimi, L. Soares, and M. Stumm, "Managing shared L2 caches on multicore systems in software," in *Proc. Workshop Interaction Between Operating Syst. Comput. Archit.*, 2007, pp. 26–33.
[6] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *Proc. IEEE 14th Int. Symp. High Perform. Comput. Archit.*, 2008, pp. 367–378.
[7] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm, "RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations," *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 1, pp. 121–132, 2009.
[8] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proc. 4th ACM Eur. Conf. Comput. Syst.*, 2009, pp. 89–102.
[9] L. He, Z. Yu, and H. Jin, "FractalMRC: Online cache miss rate curve prediction on commodity systems," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp.*, 2012, pp. 1341–1351.
[10] Y. Ye, R. West, Z. Cheng, and Y. Li, "COLORIS: A dynamic cache partitioning system using page coloring," in *Proc. 23rd Int. Conf. Parallel Archit. Compilation Techn.*, 2014, pp. 381–392.
[11] Q. Lu, J. Lin, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Soft-OLP: Improving hardware cache performance through software-controlled object-level partitioning," in *Proc. 18th Int. Conf. Parallel Archit. Compilation Techn.*, 2009, pp. 246–257.
[12] R. Guo, X. Liao, H. Jin, J. Yue, and G. Tan, "NightWatch: Integrating lightweight and transparent cache pollution control into dynamic memory allocation systems," in *Proc. USENIX Annu. Techn. Conf.*, 2015, pp. 307–318.
[13] Intel Corporation, "Improving real-time performance by utilizing cache allocation technology," 2015. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-whitepaper.pdf. Accessed: Feb 12, 2020.
[14] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 308–319, 2013.
[15] L. Pons, V. Selfa, J. Sahuquillo, S. Petit, and J. Pons, "Improving system turnaround time with Intel CAT by identifying LLC critical applications," in *Proc. Eur. Conf. Parallel Process.*, 2018, pp. 603–615.
[16] X. Wang, S. Chen, J. Setter, and J. F. Martínez, "SWAP: Effective fine-grain management of shared last-level caches with minimum hardware support," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2017, pp. 121–132.
[17] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "KPart: A hybrid cache partitioning-sharing technique for commodity multicores," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 104–117.
[18] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2006, pp. 423–432.
[19] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," *ACM SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 381–391, 2007.
[20] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and efficient fine-grain cache partitioning," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 3, pp. 57–68, 2011.
[21] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *Proc. 22nd Int. Conf. Parallel Archit. Compilation Techn.*, 2013, pp. 213–224.
[22] R. Wang and L. Chen, "Futility scaling: High-associativity cache partitioning," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2014, pp. 356–367.
[23] P.-A. Tsai, N. Beckmann, and D. Sanchez, "Jenga: Software-defined cache hierarchies," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 652–665.
[24] A. Wolfe, "Software-based cache partitioning for real-time applications," *J. Comput. Softw. Eng.*, Ablex Publishing Corp. Norwood, NJ, USA, vol. 2, no. 3, pp. 315–327, 1994.
[25] J. Corbet, "Transparent huge pages," *LWN.net*. Accessed: Feb. 12, 2020. [Online]. Available: https://lwn.net/Articles/359158/
[26] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering Intel last-level cache complex addressing using performance counters," in *Proc. Int. Symp. Recent Advances Intrusion Detection*, 2015, pp. 48–65.
[27] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Systematic reverse engineering of cache slice selection in Intel processors," in *Proc. Euromicro Conf. Digital Syst. Des.*, 2015, pp. 629–636.
[28] A. Scolari, D. B. Bartolini, and M. D. Santambrogio, "A software cache partitioning system for hash-based caches," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, 2016, Art. no. 57.
[29] J. Ahn, C. Hyun, D. Lee, and S. H. Noh, "Cache-aware block allocation for memory-technology storage targeted file systems," in *Proc. 34th ACM/SIGAPP Symp. Appl. Comput.*, 2019, pp. 1424–1431.
[30] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel: From I/O Ports to Process Management*. Sebastopol, CA, USA: O'Reilly Media, 2005.

XeFlow: Streamlining Inter-Processor Pipeline Execution for the Discrete CPU-GPU Platform

Zhifang Li, Beicheng Peng, and Chuliang Weng

Abstract—Nowadays, GPUs have achieved high throughput computing by running plenty of threads. However, owing to disjoint memory spaces of discrete CPU-GPU systems, exploiting CPU and GPU within a data processing pipeline is a non-trivial issue, which can only be resolved by the coarse-grained workflow of "copy-kernel-copy" or its variants in essence. There is an underlying bottleneck caused by frequent inter-processor invocations for fine-grained batch sizes. This article presents *XeFlow* that enables streamlined execution by leveraging hardware mechanisms inside new generation GPUs. XeFlow significantly reduces costly explicit copy and kernel launching within existing fashions. As an alternative, XeFlow introduces *persistent operators* that continuously process data through *shared topics*, which establish efficient inter-processor data channels via hardware page faults. Compared with the default "copy-kernel-copy" method, XeFlow shows up to $2.4\times \sim 3.1\times$ performance advantages in both coarse-grained and fine-grained pipeline execution. To demonstrate its potentials, this article also evaluates two GPU-accelerated applications, including data encoding and OLAP query.

Index Terms—CPU-GPU programming, heterogeneous memory system, GPU scheduling

1 Introduction

Modern GPUs like the NVIDIA Tesla series, equipped with thousands of cores, have demonstrated the computational power in graphics rendering and deep neural networks. In data processing applications, the CPU-GPU architectures are exploited [1], [2], [3], [4] to accelerate heavy-loaded computations, where the CPU side handles the logic control and offloads computations to the GPU side. However, in these scenarios, input data initially lies on the CPU side. Therefore, inevitable data transfer and synchronization between two sides may weaken the power of using GPU.

Note that two types of mainstream CPU-GPU platforms have different influences. In consumer electronics and embedded systems, "integrated GPUs" like AMD's APU and ARM's Mali are fashionable in the market. This design puts CPU and GPU into a single chip and lets them share a single memory space. Without the concerns of memory coherency, it is easy to adopt existing software techniques like pipeline execution [5] in OpenCL. However, integrated GPUs are unsuitable for computing-intensive workloads due to fewer cores and lower memory bandwidth. Consequently, "discrete GPUs" are the mainstream in data centers and clouds, where the two kinds of processors, CPU and GPU, own discrete memory spaces called *host memory* and *device memory* respectively.

Discrete GPUs are more powerful but lead to challenges in software design. With discrete memory spaces, applications must carefully maintain coherency between two sides by following a workflow of "copy-kernel-copy" (CKC). We take CUDA, the de facto toolkit to program NVIDIA's GPU, as an example to present a CKC workflow. First, the CPU sends a command (e.g., `cudaMemcpy`) that migrates data from host memory to device memory via the PCIe bus and the DMA engine. Then a *kernel* is invoked to run on the GPU, which computes data in device memory with plenty of threads. After waiting for the kernel to finish, the second copy is involved to get results back to the host memory.

In data processing applications like data encoding and analytics, "pipeline" is one of the most common patterns that applies a series of operations on the input data. However, adopting CPU-GPU pipelines to discrete CPU-GPU platforms is not straightforward. Normally, application data is held by the CPU side, and each CKC workflow can only process a limited amount of data. For these reasons, current systems could only support CPU-GPU pipelines at the API level. But in essence, their underlying implementations are still based on the CKC workflows [1], [2], [3], [4], [6]. As shown in Fig. 1a, this approach divides input data into continuous batches with the given size and deals with each batch by a CKC workflow. Due to costly inter-processor invocations—two copies and one kernel launching per batch, this overhead cannot be ignored [7]. CUDA's HyperQ [8] provides multiple command queues to overlap data copy and kernel execution from concurrent CKC workflows.

However, this method does not work well all the time. Here is a trade-off between throughput and latency. With fewer invocations, larger batches provide better throughput but incur a longer latency per batch, where HyperQ could fully overlap concurrent CKC workflows. In contrast,

• *The authors are with East China Normal University, Shanghai 200062, China. E-mail: {zhifangli, beichengpeng}@stu.ecnu.edu.cn, clweng@dase.ecnu.edu.cn.*

Manuscript received 9 June 2019; revised 11 Jan. 2020; accepted 15 Jan. 2020. Date of publication 21 Jan. 2020; date of current version 8 May 2020.
(Corresponding author: Chuliang Weng.)
Recommended for acceptance by J. C. Hoe.
Digital Object Identifier no. 10.1109/TC.2020.2968302

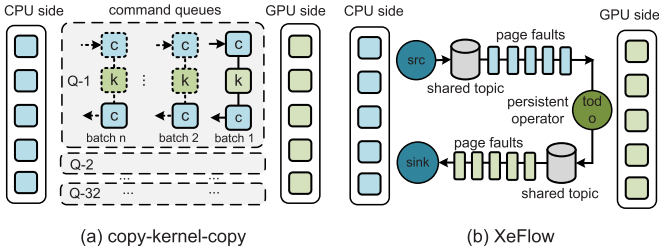


Fig. 1. Two models of inter-processor pipeline execution.

smaller batches shorten this latency but cause underutilization of GPU. Owing to limited tens of command queues (e.g., 32 in NVIDIA GPUs), HyperQ is hard to execute many fine-grained CKC workflows simultaneously. To handle such fine-grained workloads, [9] employs lightweight tasks instead of heavy kernels. Unfortunately, when data lies on the CPU side, this approach still needs explicit copy, and it is incompatible with HyperQ to hide transfer latency.

To the best of our knowledge, the CKC workflow (or its variants) is still the mainstream in CPU-GPU pipelines, even though it leads to performance issues under small batch sizes. Since those software methods based on inter-processor invocations are imperfect, we attempt to explore hardware-based methods. With the advent of new discrete GPUs like NVIDIA's Pascal and Volta, *unified memory (UM)* provides an opportunity to communicate CPU and GPU sides through hardware page fault. Besides, the technique of "persistent thread" [10] enables partial control inside kernel execution.

On the top of UM and persistent thread, we present *XeFlow*, a framework to build data processing pipelines on the discrete CPU-GPU platforms. To deal with issues in fine-grained workloads, we employ a streamlined approach, as shown in Fig. 1b. Our design principle includes two basic ideas. First, we let hardware page faults rather than software invocations handle data transfer and overlapping. Second, we only launch the kernel once and then compute input data throughout the whole running time. To address the challenges encountered in the CPU-GPU pipeline, we make contributions as follows.

Persistent Operator. To execute CPU-GPU pipelines by CKC, each batch of data will launch a kernel for computing. We realize a subtle fact that the same kernel is invoked for different batches repeatedly. Therefore, it makes sense to use a single kernel for the whole input data. From this idea, we introduce the notion of *persistent operator*, which extends persistent thread to reuse the kernel context. The persistent operator wraps user-defined code and lives on the GPU to perform continuous execution, without repetitive kernel launching. Unlike [9] that uses a CPU-managed table to schedule GPU tasks, the persistent operator is controlled by itself, without the assistance of remote CPU.

Shared Topic. Apart from kernel launching, the invocation of explicit copy is another bottleneck. By using page fault of UM to transfer data between processors, we employ *shared topic* to establish an efficient data channel shared by CPU and GPU sides. Our design is inspired by the "subscribe-publish" scheme in the database community, which is used to decouple producers and consumers in a pipeline. Specifically, the persistent operator is notified to read data from the subscribed topic and write computed data to the published topic only when data is ready. It provides an interface based on UM pointers to read or write data directly without explicit copy. If the UM pointer refers to the remote side, the shared topic leverages page faults to deliver desired data and skip unwanted data as well.

Layered Memory Coherency. As shared topics deal with data transfer via UM, the thrashing of page faults between processors is the next troublesome challenge. Besides, the default setting of shared topics does not prevent concurrency hazards. To cope with these issues, we manage UM data according to its access patterns and assign each layer of shared topics with an individual strategy of coherency. By making each side access local data as far as possible, this policy reduces the overhead caused by page faults.

Optimizations of Heterogeneous Workloads. Considering various persistent operators, they can either be computing-intensive or data-intensive, which are bounded by different GPU resources, including computing and transfer bandwidth. To cope with heterogeneous persistent operators, we reorganize the kernel as multiple partitions, where each persistent operator is handled by a partition of threads. By controlling the partitioning of kernel, the kernel could co-utilize two types of resources simultaneously. However, this optimization cannot be achieved easily due to the undocumented implementation of GPU. Therefore, we provide a model to learn the characteristics of GPU, which helps to find an optimal partitioning among persistent operators.

By combining these contributions, XeFlow achieves streamlined pipeline execution on the discrete CPU-GPU systems. In experiments, we design a specialized microbenchmark to evaluate the impact of both CPU-GPU transfer and GPU execution. Compared with the default CKC workflow, experimental results show up to $2.4\times\sim3.1\times$ performance in coarse-grained and fine-grained batch sizes. To show the potentials of XeFlow, we further evaluate two case studies, including data encoding and OLAP query.

We organize the rest of this paper as follows. Section 2 introduces related background. We outline XeFlow's programming model in Section 3. The implementation of shared topics is shown in Section 4. Section 5 explains how to run persistent operators on the GPU. Then Section 6 evaluates the performance of XeFlow through one microbenchmark and two case studies. The related work is discussed in Section 7. Finally, we conclude this paper in Section 8.

2 Background

This section takes CUDA for example to introduce the prior knowledge and challenges in discrete CPU-GPU platforms, which are also similar for other scenarios like OpenCL.

2.1 CPU-GPU Memory Model

CUDA 4.0 with NVIDIA's Fermi GPU (2011) supports 48-bit virtual addressing and memory pinning, which allows GPU to access host memory but not vice versa. CUDA 6.0 with Kepler (2012) first employs imperfect UM that does not allow concurrent access from two sides. In Pascal (2016) or newer GPUs of NVIDIA, CUDA 9.0 supports practicable UM with concurrent access.

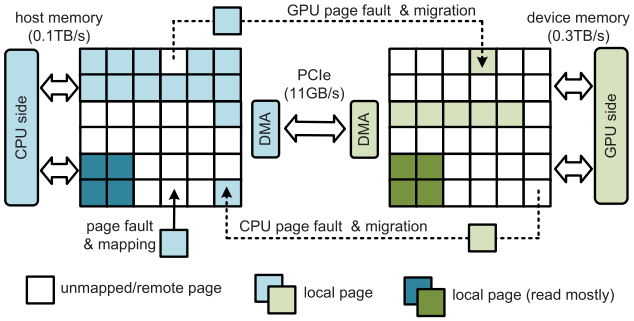


Fig. 2. The paging mechanism of unified memory.

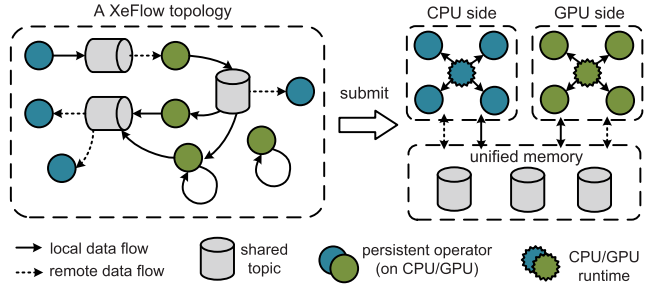


Fig. 3. The overview of XeFlow programming.

As shown in Fig. 2, UM maps host and device memory into a unified virtual space and maintains the coherency via page faults. We could invoke `cudaMallocManaged` to create UM buffers or define UM variables with `__managed__` prefix. By default, a new virtual page is not mapped to a physical page until a page fault is triggered to map it to a local physical page by modifying the page table. Intuitively, the local page can be directly accessed. As for a remote page, a page fault is triggered to migrate it to the local side via the DMA engine and the PCIe bus. Although UM manages data migration automatically, CUDA 9.0 provides optimization "hints" that can affect preferred page location and mapping.

With UM involved, CUDA suggests a simplified version of CKC, namely *kernel-over-unified-memory (KoUM)*. As its name implied, the kernel is directly launched to scan remote data via UM migration. Even though KoUM removes explicit copy before and after kernel execution, recent research efforts [11], [12] have revealed its drawbacks due to the thrashing of page faults (a page fault can spend more than ten microseconds). Moreover, KoUM still invokes a kernel per batch like CKC. Therefore, simply adopting UM cannot solve the performance issue in fine-grained batch sizes.

2.2 Kernel and Persistent Thread

In CUDA programs, CPU plays as a "driver" that controls GPU behavior through commands such as data copy and kernel launching. The kernel computes data in device memory with plenty of threads, which are further divided into *thread blocks*. GPU hardware dispatches these thread blocks to a dozen of *streaming multiprocessors* (SMs) that execute each dispatched thread block in the unit of *thread warp*—a group of threads that run the same instructions. Each thread handles the given data item according to its "thread ID". SM could hide access latency by switching among thread warps when they encounter cache misses or page faults.

As specific hardware detail of GPU hasn't been disclosed, it is hard to control the scheduling of thread warps/blocks. In prior works [9], [13], [14], [15], *persistent thread* [10] inspires a software method to enable partial kernel control. Its basic idea is to run the fixed number of thread blocks that hardware can schedule concurrently. Each thread computes data items according to an indicated range rather than the default "thread ID". This method often works with other techniques like a CPU-managed queue to launch GPU tasks [9]. Since persistent thread does not consider inter-processor communication, it still needs explicit data copy. To run the pipeline without invocations involved, our idea is to combine persistent thread with UM that could migrate data on the fly.

3 XeFlow Programming Model

This section presents the programming model of XeFlow that addresses CPU-GPU pipelines for data processing. XeFlow abstracts each application as a "topology" in Fig. 3. We follow three steps to construct such a topology.

Abstracting Data Dependencies as Shared Topics. Unlike previous works that migrate data with explicit copy [3], [8], [9], XeFlow abstracts the data dependency between CPU and GPU as a shared topic, where producers and consumers of pipeline work together with the subscribe-publish interface. In the shared topic, data is organized as *frames*, a bulk of contiguous pages. The producer side could write data into frames continuously and then publish them to the tail of this shared topic to make data accessible. Subsequently, committed frames will be consumed with an increasing offset. Since two operations are performed by two sides simultaneously, Section 4.2 presents our proposed coherency policy to ensure concurrent safety across processors. As this interface only deliveries a handler rather than the whole frame, the shared topic eliminates buffer copy in the CKC workflow.

Filling Computations into Persistent Operators. When writing a CUDA kernel, the programmer should organize computations with plenty of threads and synchronize kernel execution manually. In XeFlow, we can easily put computations into the persistent operator as a payload, who lives on the CPU or GPU side and computes data from the shared topic continuously. The persistent operator provides a basic `CPU/GPU_Operator` class, where its `OnCompute()` is rewritten with computational code by the programmer. Since XeFlow deals with GPU execution through a managed kernel, user-defined code uses virtualized thread IDs to locate data items, rather than CUDA's thread IDs.

Constructing the Topology with CPU-GPU Pipelines. After shared topics and persistent operators have been defined, we construct them as the executable topology. According to their data dependency, we use a C++11 lambda closure to link each triple of `<previous_topic, operator, next_topic>` as a CPU-GPU pipeline. Taking the most common "ping-pong" topology in Fig. 4 for example, this topology consists of three persistent operators (`Src`, `ToDo`, and `Sink`) and two shared topics (`C2G` and `G2C`).

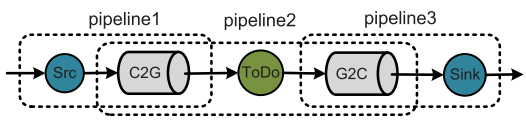


Fig. 4. The ping-pong topology.

These components are connected to three pipelines. The first pipeline $<$`null,Src,C2G`$>$ continuously pushes input data from the CPU side. Meanwhile, the second pipeline $<$`C2G,ToDo,G2C`$>$ pulls input data from the topic `C2G` and pushes GPU-computed value to the next topic `G2C`. Finally, the third pipeline $<$`G2C,Sink,null`$>$ pulls computed data back to the CPU side. The following code template explains how to construct this topology.

```
#include "xeflow.h"
__managed__ Topic C2G, G2C;
class Src: public CPU_Operator{
 __host__ void OnCompute(Context* ctx){
  int* output = (int*)ctx->NextTopic();
  int* end = output+ctx->MaxPosOfNextTopic
  ();
  for(; output < end; output++)
   *output = RandomInt();
}};
class ToDo: public GPU_Operator{
 __device__ int Multi2(int v){ return v*2;}
 __device__ void OnCompute(Context* ctx){
  // get virtualized thread ID
  int tid = ctx->vThreadID();
  int* input = (int*)ctx->PrevTopic();
  int* output = (int*)ctx->NextTopic();
  output[tid] = Multi2(input[tid]);
}};
class Sink: public CPU_Operator{
 __host__ void OnCompute(Context* ctx){
  int* input = (int*)ctx->PrevTopic();
  int* end = ctx->MaxPosOfPrevTopic();
  for(; input < end; input++)
   printf("%d\n", *input);
}};
int main(){
 InitTopic({&C2G, &G2C});
 auto topo=[]__host__ __device__(XeFlow* x){
  x->Link(nullptr, new Src(), &C2G);
  x->Link(&C2G, new ToDo(), &G2C);
  x->Link(&G2C, new Sink(), nullptr);};
 SubmitAndExec(topo);
}
```

After submitting this topology, XeFlow transparently manages its execution without the concerns on data transfer and scheduling. We leave the details of how the topology works in the next two sections.

4 Shared Topic

To cope with data transfer between persistent operators efficiently, the shared topic is organized as three layers. The catalog layer stores metadata defined by the topology. The offset index layer indicates the physical position of data in the bottom frame layer. This section will present the detailed implementation of shared topics.

4.1 Inter-Processor Synchronization

The first step to build shared topics is establishing efficient primitives for inter-processor synchronization. As the GPU side does not run an operating system, existing mechanisms like mutex and semaphore cannot be used for this purpose. Furthermore, CUDA only allows the CPU to synchronize kernel execution by costly invocations, but not vice versa—the GPU side cannot synchronize with CPU by itself. Therefore, duplex and invocation-free mechanisms are urgently required. Here are three methods we use in the shared topic.

Direct Read. In the shared topic, *direct read* is the uppermost approach we use for data synchronization between two sides. As the name implies, there is no explicit copy before kernel execution. This approach lets persistent operators on the CPU or GPU scan frames on the remote side via UM pointers. This method makes use of page faults to transfer accessed pages of frames between processors. However, direct read works correctly only when the remote side does not modify data simultaneously. The following two mechanisms are employed for concurrent occasions.

Inter-Processor Atomic Instruction. Based on the mechanism of page fault, UM provides "inter-processor" atomic instructions like `atomicAdd_system()`, which could atomically update a concurrent UM variable used by CPU and GPU sides. Hardware page faults are triggered to maintain the memory coherency across two sides. In the shared topic, we leverage atomic instructions to establish the basic atomicity of variables, such as pointers and cursors. However, atomic instructions are quite expensive owing to the overhead of costly PCIe round trips between two sides.

Inter-Processor Spinlock. In common designs, mutex is employed to protect data structures shared by multiple users. Owing to the lack of an operating system, GPU does not support typical mutex. As an alternative, we could implement *inter-processor spinlock* based on the above atomic instructions. Before using spinlock, a lock variable is initialized as 0. To acquire this lock, the caller repeats "compare-and-swap" until it swaps 1 to this lock variable successfully. To release the lock, we reset it back to 0. To avoid potential deadlocks, only the leader thread within the thread warp can lock or unlock this variable.

Considering the high concurrency of CPU-GPU systems, frequent inter-processor synchronizations lead to frequent page faults as well. The design of shared topics should consider the impact of involved page faults. This problem is solved in the next subsection.

4.2 Layered Memory Coherency

To minimize the impact of page faults, we optimize the policy of memory coherency inside shared topics. We observed that different parts of shared topics have individual patterns of data access. For this consideration, we organize the shared topic as multiple layers shown in Fig. 5, and apply a specialized strategy on each layer with CUDA's locality hints. We introduce each layer in a top-down order.

Catalog (Read-Mostly). The top layer stores the metadata of shared topics, including frame sizes and handlers to the below layer. During topology execution, both CPU and GPU sides frequently read metadata from the catalog. Default UM will migrate pages back and forth among two sides. Since all shared topics are pre-defined, the catalog is rarely modified during execution. As recognizing this property, we apply the *read-mostly* policy to the catalog, which maintains a local replica of catalog for each side. Consequently, each side could directly read the local replica without page faults involved. To

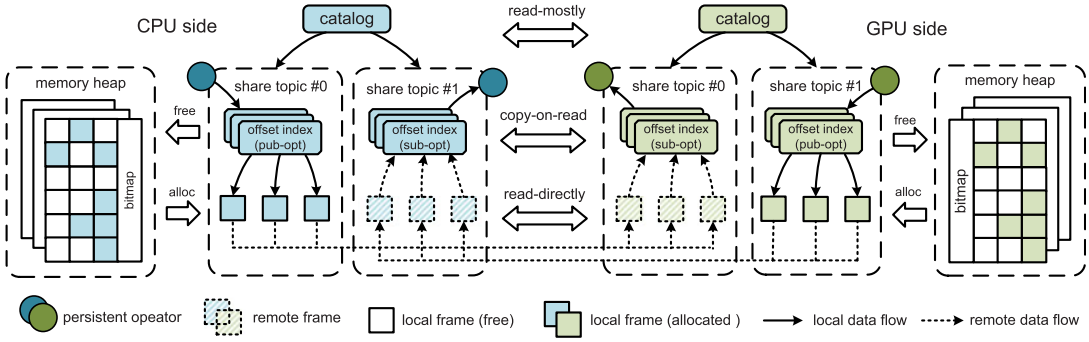


Fig. 5. The architecture of shared topics.

deal with rare updates, two page faults are triggered to modify two replicas, respectively.

Offset Index (Copy-On-Read). The next layer is the offset index that helps persistent operators to read the frame with a subscribed offset. This layer stores the index handler of each published offset along with the pointer of its related frame. Unlike the catalog, this layer is frequently modified to register published frames to the latest offset. Therefore, we adopt the *copy-on-read* policy to decrease page faults involved. It uses two replicas optimized for publishing and subscribing. The publish-optimized replica locates on the CPU or GPU, based on the running location of the previous operator in the pipeline, which will publish frames to this replica. This policy restricts the modified ranges within only one side. Similarly, the subscribe-optimized replica locates on the other side for subscribing to decrease remote access.

To illustrate how the offset index works, we take Fig. 6 as an example. When publishing a frame, the persistent operator inserts an index handler into the index page of publish-optimized replica and modifies the range of valid offset. Considering concurrent modifications, the index page is locked. To read a frame with the subscribed offset, the persistent operator first looks at the subscribe-optimized replica on the local side. If the latest index handlers are not found locally, page faults are triggered to copy them from the other replica. As each published frame registers a 64-byte index handler, each page fault will deliver 64 handlers (i.e., 4 KB/64 = 64) of published frames. After that, the position of index handlers in the page can be easily computed from the given offset. If this frame has not been published, subscribing is failed. This operation can be repeated until this frame is published. Since modifications only perform on the publish-optimized replica, subscribing does not need to lock the subscribe-optimized replica. After subscribing is finished, the persistent operator could consume subscribed frames with the index handler.

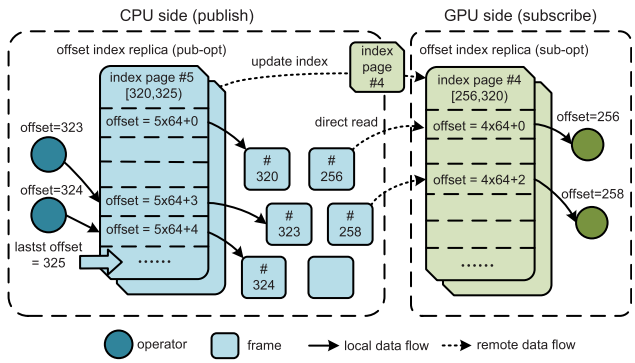


Fig. 6. Two replicas of offset index.

Frame (Read-Directly). The underlying layer stores published frames upon UM data. The CKC model provides a buffer-copy interface, which needs to migrate the buffer between two sides. The overhead cannot be neglected, especially when the buffer size is small. The frame layer provides a read-directly interface. Unlike the above two policies, this layer only maintains a single replica on the published side. When the frame is accessed through UM pointers, GPU's hardware automatically handles transfer and coalesces the requests of contiguous pages. This replica is fixed on the published side to create the opportunity of reducing data to transfer, as presented in the next subsection.

4.3 Skipping Transfer via Fine-Grained Data Structures

Due to limited PCIe bandwidth, CPU-GPU transfer can be the bottleneck in many scenarios. [16] avoids redundant transfer when the areas of accessed data are overlapped and regular. Based on the above read-directly interface, this paper shows another idea of "skipping", where undesired data can be skipped by looking up fine-grained data structures before copying the whole frame. To illustrate its potentials, we take *online analytical processing* (OLAP) [17] as an example, which normally suffers from data transfer. Limited by CKC, GPU-accelerated OLAP query often employs coarse-grained tables. With fine-grained data structures, we bring an opportunity to skip undesired items in the table.

Columnar Paging. CPU-based OLAP systems often organize the table as contiguous rows. To execute an OLAP query by GPU, the entire table is moved to the GPU side reluctantly even if not all columns are desired. Moreover, the row-based format also causes uncoalesced access in device memory. A feasible method is *columnar paging* [18] that arranges values of the same column to the same page in the frame. As direct read allows GPU to access pages via frame pointers, we could easily adapt columnar paging to the shared topic. This optimization enables persistent operators to skip undesired pages and coalesce memory access for each page. Unfortunately, the CKC model is inefficient to manipulate small pages with explicit copy.

Lightweight Indexing. In OLAP, "select" is a standard operator that scans a table to collect rows meeting the given predications. Lightweight indexing brings potential optimizations by dividing each frame as data pages and a few index pages. By looking up index pages in advance, this optimization dramatically reduces the amount of data pages to transfer. For the range predication, index pages could store the minimum and maximum values of each data page. Before scanning a data page, the GPU side could look up the index pages to detect whether it falls into the min-max range. As for the point predication, index pages could store *bloom filter* [19], a hash sketch of data pages to estimate whether a data page contains matched rows.

4.4 User-Space Memory Management

This subsection introduces the memory management strategy in the frame layer. When creating a UM buffer, CUDA's default allocator needs to modify page tables of both CPU and GPU sides. For this reason, allocating each frame on the fly is very costly. Previous research efforts [20], [21] achieve efficient device memory allocation in the user space. We are inspired by this design but extend it to serve both CPU and GPU sides with UM. For each side, XeFlow employs a *memory heap* to allocate memory locally. When the memory heap initializes, UM buffers are created by `cudaMallocManaged`. All pages are pre-mapped to the page table according to the desired memory coherency. With the virtual paging of UM, the capacity of memory heap can oversubscribe total device memory (e.g., 24 GB for NVIDIA P40). Since current UM does not support disk swapping, this capacity cannot exceed host memory.

The memory heap divides each UM buffer into contiguous frames, and a bitmap, where the *ith* bit indicates whether the *ith* frame is free. Each frame contains contiguous 4 KB data pages (i.e., by default, 4 MB frame size). To allocate a new frame, the memory heap tries to flip a bit from 0 to 1 with a linear hash path. In contrast, the memory heap flips the bit back to free a useless frame. Considering concurrent allocation, the modifications of bitmap should use atomic instructions. When one memory heap exhausts all created buffers, it could extend the capacity by creating new buffers or stealing unused buffers from another heap.

5 Runtime System

This section introduces the runtime system to execute persistent operators on the discrete CPU-GPU platform.

5.1 Persistent Operator Execution

For either the CPU or GPU side, the persistent operator has a similar execution flow, as presented in Algorithm 1. The core of persistent operators is a "loop" that executes computations round by round. In each round, it subscribes the input frame and then publishes the computed frame until it encounters the exit flag. Due to the inherent differences between CPU and GPU, lines 6-14 of this algorithm have different implementations for each side.

On the CPU side, it is straightforward to assign CPU threads to compute input frames. In implementation, we can easily use a parallel programming toolkit like Linux Pthread or OpenMP. When the topology contains multiple persistent operators, many works have already studied mature strategies to achieve load balancing. Therefore, this paper mainly focuses on the GPU side.

Algorithm 1. Template for Topology Execution

Input: a topology Γ with a series of piplines like (topic T_{pv}, operator Θ, topic T_{nt})

```
1  for each pipeline (T_pv, Θ, T_nt) ∈ Γ do
2    Initialize a frame handler f_pv for T_pv ;
3    Initialize a frame handler f_nt for T_nt ;
4    Forward (f_pv, Θ, f_nt) to a CPU thread or a GPU kernel ;
5    for round r = 0 ; Θ is not exited; r++ do
6      f_pv ← subscribe a committed frame from T_pv ;
7      for each item t_i ∈ f_pv do
8        t_i^* ← apply computations of Θ on the t_i ;
9        write t_i^* into f_nt by the posistion i ;
10     end
11     if f_nt is fulfilled then
12       publish f_nt to the T_nt ;
13       f_nt ← create an empty frame ;
14     end
15   end
16 end
17 return wait for all Θ in the Γ ;
```

As for GPU execution, the situation gets complicated. Each persistent operator has various demands of GPU resources like computing and bandwidth. As CUDA kernels are scheduled by undocumented hardware, our optimization spaces to co-utilize two resources are restricted by GPU implementation. To overcome this barrier, we divide a single CUDA kernel to execute persistent operators in a managed manner, as shown in Fig. 7.

To exploit parallelism, each thread block of physical kernel only subscribes a part of frames in the shared topic. Each thread block independently runs the above scheduling loop to avoid expensive global synchronizations. In each round, it creates a *work-vector* that indicates the assigned operator ID of each thread warp. This work-vector is stored in GPU's fast on-chip memory to speed up access. Then each thread warp can look up the work-vector and choose the assigned persistent operator to execute in this round.

Considering bothersome logic divergences during GPU execution, all thread warps are assigned to persistent operators in an aligned way. The work-vector also stores frame handlers for each persistent operator, thus multiple operators could subscribe a shared topic via independent offsets. As XeFlow takes over intra-GPU execution, "virtualized"

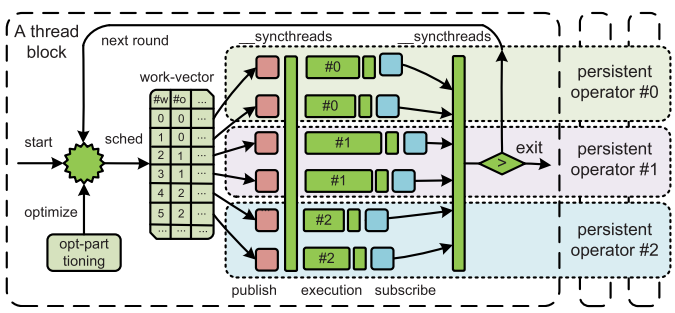


Fig. 7. The virtualized kernel for GPU execution.

thread IDs should be used to replace CUDA's thread IDs for locating data items in the frame. After finishing the execution, the thread block is synchronized to move on to the next round. We conclude the implementation of GPU execution in Algorithm 2.

Algorithm 2. GPU Side Execution

Input: a topology Γ with a series of piplines like (topic T_{pv}, operator Θ, topic T_{nt})

```
Γ* ← extract all GPU operators from Γ ;
for round r = 0 ; Γ* ≠ ϕ ; r++ do
  work-vector ← assign 32 warps to |Γ*| operators;
  do subscribe ;
  __syncthreads ;
  Θ ← work-vector[warp-id].operator-id ;
  if 32 * warp − id ≤ thread − id < 32 * (warp − id + 1) then
    vir-thread-id ← thread-id + r* thread-num ;
    execute Θ by using vir-thread-id as the position ;
  end
  do publish ;
  if Θ is finished then
    delete Θ from Γ* ;
  end
  __syncthreads ;
end
```

5.2 Optimizations for Heterogeneous Workloads

During kernel execution, persistent operators in the different thread warps share GPU hardware via time slicing. When a thread warp begins to transfer data, GPU yields the idle arithmetic units to another thread warp. In essence, different persistent operators prefer various GPU hardware resources. For example, data-intensive ones can be improved by using more thread warps to hide transfer latency, while computing-intensive ones have slight promotion due to limited arithmetic units shared by multiple thread warps.

When creating a work-vector for heterogeneous persistent operators, an optimal partitioning of thread warps will improve the overall performance by co-utilizing different units inside GPU. However, it is almost infeasible to figure out their real resource demands. Current profiling tools like CUDA's `nvprof` are designed for the whole kernel. To estimate persistent operators inside the kernel, we should model the performance of this workload.

We could evaluate the execution time of a workload with n heterogeneous persistent operators as follows. In a scheduling round, we denote R_i as the round time for the *ith* persistent operator that is assigned with N_i thread warps. Intuitively, the overall round time depends on the slowest operator (i.e., $max\{R_i\}$). When a thread block consists of $\mathbf{N}$ thread warps, to get the minimized round time, the optimal resource partitioning $\{N_i\}$ can be given by

$$\underset{\{N_i\}}{\arg\min}\ max\{R_i\} = max\{\ f_i(N_i)\ \} \mid \sum N_i = \mathbf{N}, \tag{1}$$

where $f_i(N_i)$ indicates how *ith* operator is affected by the number of assigned thread warp N_i. By default, we could use a following polynomial function as an approximation.

$$f_i(N_i) = \theta_2 N_i^2 + \theta_1 N_i + \theta_0. \tag{2}$$

Data-intensive operators have lower $[\theta_2, \theta_1, \theta_0]$ because more thread warps will alleviate memory latency. In contrast, $[\theta_2, \theta_1, \theta_0]$ tends to be higher in computing-intensive operators due to limited arithmetic units. For a given operator, we cannot know the value of $[\theta_2, \theta_1, \theta_0]$ in advance. Besides, to find an optimal partitioning for n operators, exhaustive search leads to unscalable $O(\binom{N-1}{n-1})$ complexity.

5.3 Auto-Tuning Tool

To estimate $[\theta_2, \theta_1, \theta_0]$ in a scalable way, we provide a auto-tuning tool based on *polynomial regression*, a machine learning algorithm to learn the black-box GPU. When tuning *ith* operator, we could get the sampled R_i from GPU's timer. By using k samples of (N_i, R_i) under varying N_i, we can learn the estimated $[\hat{\theta_2}, \hat{\theta_1}, \hat{\theta_0}]$ through *batch gradient descent* (BGD). Since BGD calculates these samples in limited passes, it reduces the tuning complexity to linear $O(nk)$, which can be finished within milliseconds (see experiments in Table 2).

After that, we let all $\{R_i\}$ be an equal bias f_0 (e.g., the average of sampled f_i) to minimize $max\{R_i\}$. Then the optimal partitioning $\{\hat{N_i}\}$ can be estimated as

$$\{\hat{N_i}\} = \left\{ \frac{\left(\sqrt{\hat{\theta_1}^2 - 4\hat{\theta_2}(\hat{\theta_0} - f_0)} - \hat{\theta_1}\right)/2\hat{\theta_2}}{\sum\left(\sqrt{\hat{\theta_1}^2 - 4\hat{\theta_2}(\hat{\theta_0} - f_0)} - \hat{\theta_1}\right)/2\hat{\theta_2}} \times \mathbf{N} \right\}, \tag{3}$$

where $(\sqrt{\hat{\theta_1}^2 - 4\hat{\theta_2}(\hat{\theta_0} - f_0)} - \hat{\theta_1})/2\hat{\theta_2}$ is the solution of polynomial function $\hat{\theta_2} N_i^2 + \hat{\theta_1} N_i + \hat{\theta_0} = f_0$.

Note that this strategy is mainly to maximize the overall GPU utilization. The future works could extend this method for other dimensions like latency and energy consumption. To reduce incurred overhead, we could perform tuning only when the uniform partitioning leads to an imperfect deviation of $\{R_i\}$ that exceeds an empirical threshold.

6 Evaluation

This section first evaluates the performance of XeFlow via a microbenchmark. Then we evaluate its application potentials by two case studies.

6.1 Experiment Setup

Our experiments employ a machine equipped with Intel Xeon CPU and NVIDIA P40 GPU based on Pascal architecture. Table 1 lists our software and hardware specifications.

We observed that pipeline-like CPU-GPU programming has been studied by various research works [2], [3], [4]. These implementations can be concluded as the following types.

- *CKC*. This method repeatedly invokes a copy-kernel-copy workflow for each batch of input data.
- *CKC+HQx*. To overlap copy and kernel execution of CKC, this strategy leverages CUDA HyperQ to run x concurrent CKC workflows.
- *CTC*. This strategy uses lightweight GPU tasks to substitute CUDA kernels (Pagoda [9]) but still incurs two invocations of copy per batch.

TABLE 1
Software and Hardware Specifications

OS	CentOS 7
Toolchain	CUDA/NVCC 9.1
CPU side	Intel Xeon Silver 4110 @ 2.10 GHz x 2 32 hyper-threading cores, 11 MB L3 cache
GPU side	150 GB host memory NVIDIA P40 @ 1.53 GHz x 1 3840 CUDA cores, 1,024 threads per block 24 GB device memory, 346 GB/s bandwidth 11 GB/s PCIe 3.0 x 16, 2 DMA copy engines

- *KoUM.* This strategy removes two copies from CKC by using UM pointers as the kernel input. However, it still launches repetitive kernels for each batch.

To evaluate more general cases, we do not directly compare XeFlow with various systems. As an alternative, we compare their underlying implementations based on CUDA.

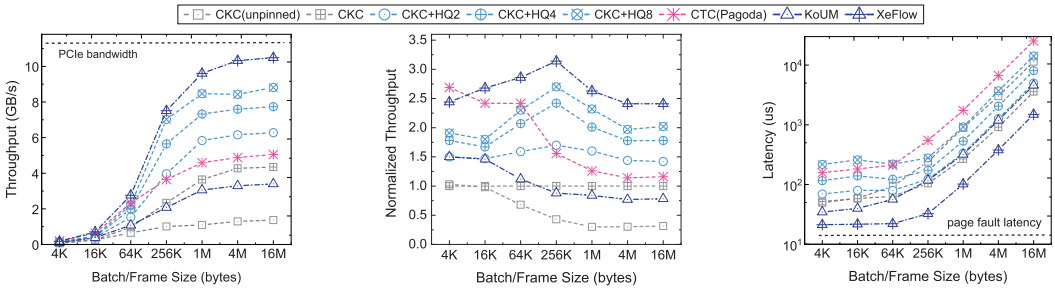
6.2 Performance Evaluation via Microbenchmark

Since this paper addresses CPU-GPU pipelines for data processing, both CPU-GPU transfer and GPU computing will impact the performance. Therefore, we design a specialized microbenchmark based on the ping-pong topology (see Section 3), which could evaluate two factors together. There are two types of variants. One is "$C(GC)_n$" where $n+1$ CPU operators (i.e., C) and n GPU operators (i.e., G) are interleaved in the topology. The other is "$C(G)_nC$" that contains two CPU operators and n adjacent GPU operators.

Overall Performance. For the most common CGC (i.e., $C(GC)_1$), Fig. 8 compares the overall performance of various implementations. Before running the microbenchmark, we divide 10 GB data based on the demanded batch sizes (or frame sizes for XeFlow). The experimental results are analyzed below. (1) As data transfer is handled by CPU rather than the DMA engine, CKC with unpinned memory causes a suboptimal performance. (2) With pinned memory involved, CKC is improved but suffers from non-overlapped data copy and kernel execution. (3) When x concurrent CKC workflows (x = 2/4/8) are running to overlap copy and execution, CKC+HQx optimizes the throughput under coarse-grained batch sizes. (4) By replacing kernels with lightweight tasks, CTC (i.e., Pagoda) shows impressive throughput under fine-grained batch sizes. However, CTC still invokes explicit copy and cannot work with HyperQ to hide copy latency. These reasons lead to performance deterioration under coarse-grained batch sizes. (5) Even though KoUM discards explicit copy, it still suffers from low throughput due to frequent page faults and kernel launching. (6) XeFlow not only minimizes runtime invocations but also reduces page faults through layered memory coherency. Compared with CKC, XeFlow achieves up to $2.4\times \sim 3.1\times$ throughput and nearly a third of latency.

Computational Workloads. In real applications, GPU is employed to deal with heavy-load computations. To further evaluate these scenarios, we fill the GPU operator in the ping-pong topology with a group of workloads (three computing-intensive and three data-intensive) based on Parboil [22], a widely-used benchmark for GPU computing. Experimental results in Fig. 9 also present noticeable improvements like the previous experiments. We notice the differences between the two types of workloads. For computing-intensive workloads in Fig. 9a, 9b, and 9c, the improvements decrease significantly when the batch size grows. This phenomenon implies that computing instead of CPU-GPU transfer becomes the bottleneck under coarse-grained batch sizes. In contrast, as for data-intensive workloads in Fig. 9d, 9e, and 9f, data transfer is still the primary bottleneck even for the cases with larger batch sizes.

Detailed Profiling. To prove our analysis of performance, we use CUDA's `nvprof` to get detailed profiling parameters. Fig. 10a shows the number of involved inter-processor invocations to process per GB data. CKC introduces three invocations per batch (i.e., copy+kernel+copy). CTC reduces it to two invocations of copy, and KoUM reduces it to one invocation of kernel launching. For this reason, the above methods still lead to performance issues under fine-grained batch sizes. In contrast, XeFlow only launches a single kernel to server persistent operators in the topology. Fig. 10b shows the number of involved page faults during execution. KoUM and unoptimized XeFlow cause frequent page faults. To evaluate the promotion of layered memory coherency, we gradually add the optimized policy of the catalog, index, and frame layer to unoptimized XeFlow. Experimental results in Fig. 10b and 10c explain three interesting facts as below. (1) For small batch sizes, the optimization of layered coherency is imperative. Otherwise, XeFlow is even worse than KoUM. Compared with unoptimized XeFlow, three optimized layers provide about 21, 52, and 334 percent improvements. (2) For larger batch sizes, about 79 percent promotion is contributed by the optimized frame layer because this layer stores large amounts of data consumed by the GPU side. (3) When the batch size is very large, the



(a) Throughput. (b) Speedup over CKC. (c) Latency per Frame.

Fig. 8. The overall performance with varying batch/frame sizes.

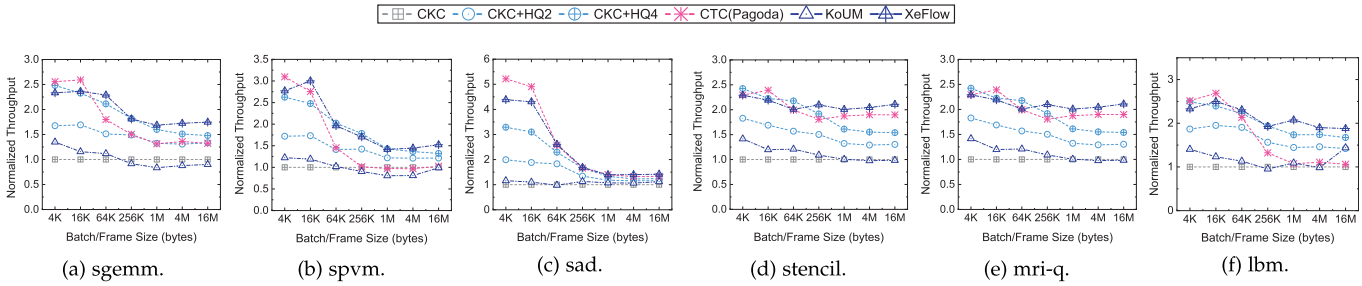


Fig. 9. The performance of computational workloads.

relative speedups decrease slightly because hardware can merge the requests of contiguous page faults.

Scheduling Overhead Analysis. To deeply understand the cost of using persistent operators, we collect the execution time in the scheduling round by CUDA clock. Fig. 11 shows the distribution of each step. When persistent operators are scheduled in each round, a tiny work-vector will be calculated, which only incurs a slight $0.02\%\sim2.5\%$ overhead of runtime. As for very small frames, publishing and subscribing take most of the execution time. Due to more expensive locking, publishing incurs larger overhead, while subscribing only needs to fetch an index handler. However, under larger frames, operator execution can utilize more GPU resources but also enlarges the response latency of each frame. To make a trade-off between utilization and latency, the programmer needs to set appropriate frame size.

Scalability. This experiment evaluates whether XeFlow could scale with more operators. With a varying GPU operator number n (1 to 4), we evaluate $C(GC)_n$ and $C(G)_nC$ topologies under fine-grained (16 KB) and coarse-grained (4 MB) batch/frame sizes. We adopt a uniform partitioning of thread warps for n GPU operators. We do not evaluate CKC and KoUM for their relatively poor performance. Fig. 12 demonstrates the throughput with varying n. Under the fine-grained batch, CKC+HQ4 and Pagoda introduce a noticeable degradation for $C(GC)_n$ because this topology imposes $n+1$ CPU-GPU copies per batch. Differently, $C(G)_nC$ leads to slight degradation for constant two copies per batch. By mitigating the overhead of invocations, XeFlow only shows a slight $1.7\%\sim3.6\%$ degradation for $C(GC)_n$ and $C(G)_nC$. For the larger frames, the main bottleneck turns into the limited PCIe bandwidth shared by n operators. Even though all three methods degrade with the increasing n, XeFlow is still better than CKC+HQ4 and Pagoda due to its fewer invocations. We also observed that XeFlow and CKC+HQ4 have higher throughput than Pagoda that does not optimize these coarse-grained cases.

Heterogeneous Workloads. When a workload contains heterogeneous persistent operators (e.g., data-intensive and computing-intensive), the partitioning of thread warps is crucial to performance. To study its impact, we construct such a heterogeneous workload, where each GPU operator performs x times *fused multiply-add (FMA)* for each input item. By changing the value of x, we can simulate either data-intensive or computing-intensive cases. First, we run XeFlow's auto-tuning tool to learn from FMA operators with varying x (short for FMAx). To simplify the analysis, we only compare two FMAx operators in each experiment. Technically, it can be applied to n operators (see Section 5.2). As shown in Table 2, this microsecond tuning time can be neglected by long-running applications. According to learned models, we estimate heterogeneous workloads under varying partitioning cases of 32 thread warps (from

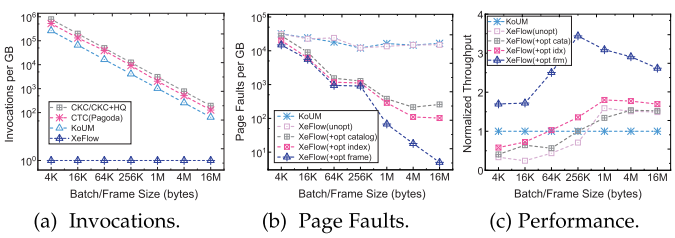


Fig. 10. Detailed profiling.

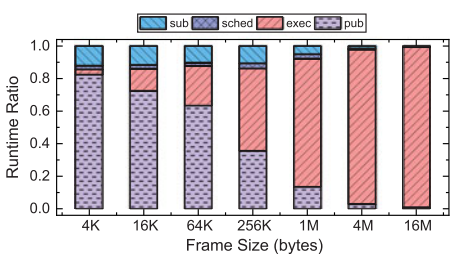


Fig. 11. The runtime distribution of XeFlow.

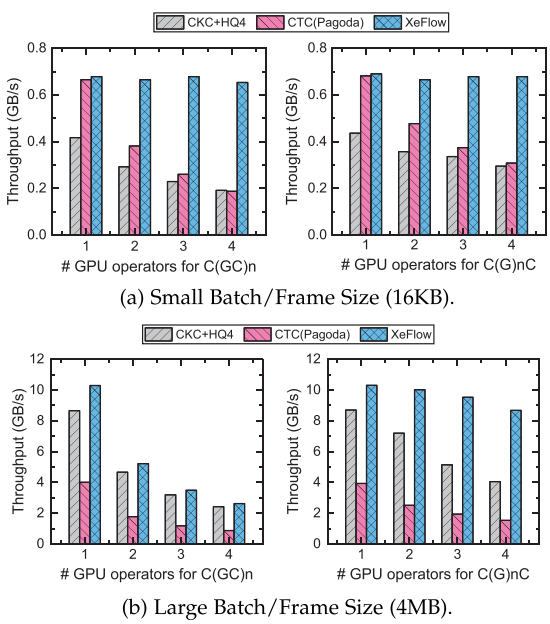
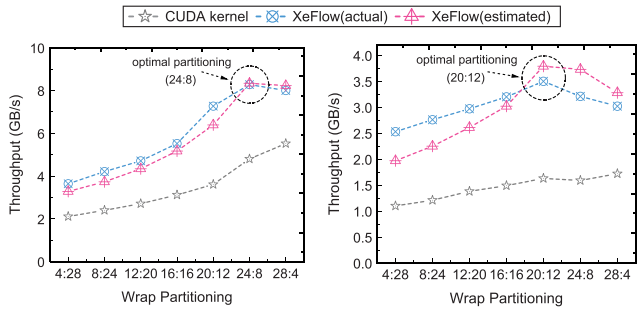


Fig. 12. The scalability of different topologies.

TABLE 2
The Tuning Time (milliseconds) for Varying Workloads and the Numbers of Samples

# samples	FMA4	FMA8	FMA16	FMA32	FMA64
800	2.64	3.19	5.72	6.88	9.26
1600	8.95	9.38	10.71	12.66	17.69
3200	17.82	18.74	20.28	25.54	35.20
6400	33.68	35.79	42.37	53.91	81.91
12800	68.85	74.55	78.47	95.64	149.79



(a) FMA4 + FMA16 (b) FMA16 + FMA32

Fig. 13. The performance of heterogeneous workloads.

4:28 to 28:4). Then, we run concurrent CUDA kernels and XeFlow to get the actual performance. Fig. 13 shows that estimated throughput has a similar curve like the actual value. Two optimal partitioning cases, 24:8 in Fig. 13a and 20:12 in Fig. 13b, show up to $4.1\times$ and $3.2\times$ over concurrent CUDA kernels, along with $2.0\times$ and $1.4\times$ speedups over the worst partitioning case of XeFlow itself.

6.3 Application Evaluation: Data Encoding

The first application is "data encoding" that is ubiquitously used in data centers. For example, erasure coding helps to tolerate storage error, and AES encryption is employed to ensure data privacy. Owing to massive calculations, CPU-only designs may incur poor performance. To accelerate such workloads with XeFlow, we develop such a prototype topology illustrated in Fig. 16 that implements data encoding on the GPU operator. To avoid the impact of different encoding implementation, we use hand-written code rather than third-party libraries. Besides, we pre-load data to the ramdisk to bypass the impact of disk I/O.

We simulate erasure encoding with the RAID-6 algorithm in the GPU operator. Fig. 14a and 14b show the performance of erasure encoding. For very fine-grained input (e.g., 4 KB), the CPU-only version that directly accesses host memory achieves better performance than all GPU versions. XeFlow achieves $2.2\times/1.8\times/3.1\times$ throughput than CKC/CKC+4HQ/KoUM when the batch/frame size is 64 KB. For coarse-grained input, XeFlow outperforms CTC that cannot overlap transfer with computing. XeFlow provides about $1/5\sim1/2$ latency than other GPU versions or even the CPU version when GPU acceleration dominates the cost of CPU-GPU data transfer.

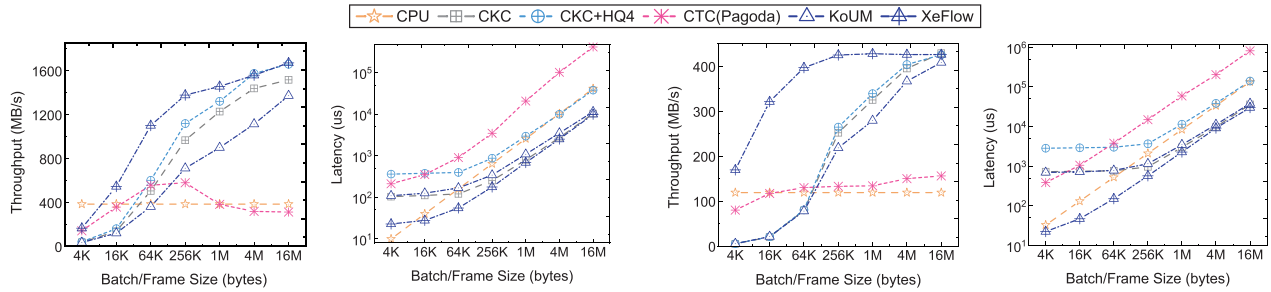
We further simulate AES encryption with *electronic code book (ECB)* mode in the GPU operator. Fig. 14c and 14d present the performance of AES encryption, where XeFlow also shows significant improvements over other methods. This experiment has a similar trend like erasure coding but has lower throughput due to more computations in AES encryption.

6.4 Application Evaluation: OLAP Query

Another application is "OLAP query" that scans the relational table by an SQL clause that consists of *select* and *filter* operators. Recent works [23], [24] employ GPU to accelerate OLAP query with the CKC workflow that needs to load the table into device memory before executing the query as a kernel. Unlike data encoding, OLAP query workloads tend to be more data-intensive. It implies that CPU-GPU transfer is the bottleneck. Thanks to fine-grained data structures, including columnar paging and lightweight indexing, XeFlow enables to skip transfer of undesired pages (see Section 4.3).

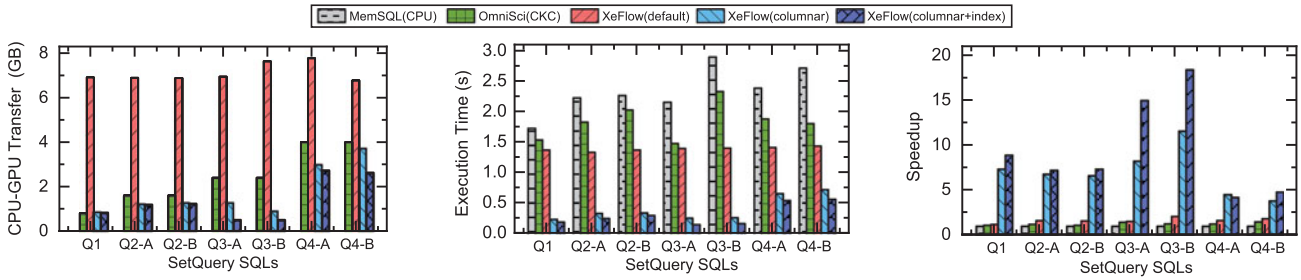
To evaluate the impact of these optimizations, we implement seven SQLs with XeFlow based on *SetQuery* [25], a synthetical OLAP benchmark. The evaluated table (i.e., 6.5 GB) consists of ten million items in 13 scalar columns. Before experiments, this table is also loaded into a ramdisk to bypass disk I/O during runtime. We also analyze other OLAP systems, including GPU-accelerated OmniSci [23] and CPU-only MemSQL [26].

Fig. 15a presents the amount of transfer between CPU and GPU sides during query execution. Technically, CPU-only MemSQL does not involve any CPU-GPU transfer. For Q1, Q2-A/B, Q3-A/B that only utilize a few columns, XeFlow reduces up to $80\%\sim85\%$ amount of CPU-GPU transfer by skipping undesired columns. Q4-A/B bring fewer $45\%\sim60\%$ reductions because more columns are scanned by the GPU side. With lightweight indexing, XeFlow shows further reduces $5\%\sim61\%$, especially for Q3-A/B with a smaller data selectivity. Restricted by coarse-grained copy, OmniSci cannot adopt these fine-grained data structures. Fig. 15b shows that the execution time follows an inverse



(a) Erasure Coding Throughput. (b) Erasure Coding Latency. (c) AES Encryption Throughput. (d) AES Encryption Latency.

Fig. 14. The performance of encoding.



(a) CPU-GPU Transfer. (b) Execution Time. (c) Normalized Speedup.

Fig. 15. The performance of OLAP query.

trend of CPU-GPU transfer. When CPU-GPU transfer dominates GPU acceleration, GPU-accelerated OmniSci is not better than CPU-only MemSQL. Thanks to fully overlapped table transfer, default XeFlow has $8\%\sim64\%$ promotion over OmniSci based on CKC. With the combination of columnar paging and lightweight indexing, Fig. 15c illustrates that XeFlow achieves up to $4.8\times\sim18.5\times$ speedup over MemSQL and $3.1\times\sim10.3\times$ speedup over OmniSci.

6.5 Discussion

Even though XeFlow helps to upgrade current applications with GPU acceleration, it requires programmers to reorganize the application as a series of pipelines for data processing. Since current XeFlow does not support control operators like "if-else" and reduce operators like "aggregation", we still cannot adopt XeFlow to the application based on a more complex topology. To support these structures, the runtime system of XeFlow should be modified. Each control or reduce operator will incur a global synchronization before starting the next operator, which cannot be executed in a streamlined fashion.

7 Related Work

Considering the rapid development of hardware, the architecture of CPU-GPU applications is still an open issue. This section briefly concludes the relevant research.

Data Transfer within Discrete CPU-GPU Systems. [7] models the performance of CPU-GPU applications and presents the significant impact of data transfer. To simplify data management in CPU-GPU systems, ADSM [27] enables the CPU side to access GPU data transparently, but not vice versa. GPUfs [28] enables GPU to manipulate file data with POSIX-like API. Gensys [29] allows GPU to control memory allocation and file access by system calls. Gdev [30] supports oversubscribed device memory by swapping data between CPU and GPU sides. GPUnet [31] abstracts transfer primitives in the CPU-GPU system. Note that the above methods employ software invocations to perform inter-processor copy, while XeFlow solves this issue via hardware page fault with optimized coherency. HiWayLib [16] avoids redundant transfer when the areas of accessed data are overlapped and regular, while XeFlow skips the transfer of data that will not be computed by GPU.

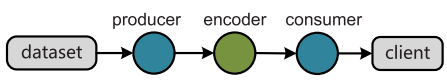


Fig. 16. The topology of data encoding accelerated by GPU.

CPU-GPU Programming Model. CUDA 10.0 provides the notion of "graph" to launch multiple GPU kernels with a single call. However, it does not change the CKC workflow when input data lies in the host memory. PTask [2] supports POSIX pipe with CPU and GPU, and Dandelion [4] uses it to execute compiled programs. XKaapi [3] runs CPU-GPU pipelines with work stealing. Although these works use pipeline at the API level, their underlying execution still employs the CKC workflow, which is inefficient to deal with small batches. [32] enables pipeline execution on the integrated GPU by hardware mechanisms. [5], [33] execute query pipelines on the integrated GPU, while our paper employs fine-grained data structures to address the bottleneck of PCIe bandwidth on the discrete GPUs. Pagoda [9] employs lightweight tasks to substitute heavy-weight kernels but does not optimize data copy especially for large batches. By minimizing runtime invocations with persistent operators and shared topics, XeFlow ensures a good performance under both fine-grained and coarse-grained workloads.

The Trend of CPU-GPU Fusion. Nowadays, the fusion of CPU and GPU has been a trend for both industry and research. UM is a key technology for CPU-GPU fusion. [34], [35] focus on the hardware design of virtual GPU memory, which are not adopted to the current products. Before NVIDIA's Pascal GPUs providing real UM, prior works of virtual GPU memory [11], [12], [36] are limited by hardware supports. RSVM [37] and ActivePointer [38] achieve software virtual memory as an alternative when GPU hardware does not support virtual memory. [39] provides basic synchronization primitives in the GPU, we extend them for the discrete CPU-GPU system and amortizes the overhead of synchronization with optimized coherency. By modifying GPU memory designs, [40] allows CPU to get a subset of results from GPU before the kernel finishing. [41], [42] maintain inter-processor coherency through hardware protocols but need to modify the design of CPU and GPU. Therefore, they can only run on the simulators, while XeFlow provides a software solution for the current products.

GPU Kernel Scheduling. [43], [44] study how to integrate multiple operators into a single kernel during the compilation time. Whippletree [13] implements intra-GPU pipelines through the task queue and persistent thread but does not take CPU-GPU pipelines into considerations. VersaPipe [14] analyses multiple types of GPU execution models, including persistent thread that enables further control within GPU. Kernelet [15] partitions GPU kernels for multiple tasks based on the characteristics of memory access. However, Kernelet

requires prior assumptions of black-box GPU hardware. Poise [45] uses machine learning to optimize thread organizations of kernel but does not consider the operators within a virtualized kernel. Inspired by these works, XeFlow divides thread warps in the GPU kernel for heterogeneous operators according to their resource demands.

8 Conclusion

This paper has presented XeFlow that exploits pipeline execution on the discrete CPU-GPU platforms. Limited by hardware, prior CPU-GPU systems still follow the CKC workflow or its variants, which are only optimized for coarse-grained scenarios. With the notion of persistent operator and shared topic, XeFlow minimizes inter-processor invocations during runtime. XeFlow also alleviates the cost of involved page faults through layered memory coherency. As for heterogeneous workloads, we optimize GPU scheduling to co-utilize computing and bandwidth resources. In evaluation, XeFlow achieves significant speedup for both coarse-grained and fine-grained cases. This paper explores a novel architecture for CPU-GPU software and encourages future works on the new GPU hardware.

Acknowledgments

This research was supported by National Key Research and Development Program of China (No. 2018YFB1003400), and National Natural Science Foundation of China (No. 61772204 and 61732014).

References

[1] P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki, "HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines," *Proc. VLDB Endowment*, vol. 12, pp. 544–556, 2019.

[2] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: Operating system abstractions to manage GPUs as compute devices," in *Proc. 23rd ACM Symp. Operating Syst. Princ.*, 2011, pp. 233–248.

[3] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin, "XKaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 1299–1308.

[4] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly, "Dandelion: A compiler and runtime for heterogeneous systems," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 49–68.

[5] J. Paul, J. He, and B. He, "GPL: A GPU-based pipelined query processing engine," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1935–1950.

[6] A. Sabne, P. Sakdhnagool, and R. Eigenmann, "Scaling large-data computations on multi-GPU accelerators," in *Proc. 27th Int. ACM Conf. Supercomputing*, 2013, pp. 443–454.

[7] 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.

[8] H. Li, D. Yu, A. Kumar, and Y.-C. Tu, "Performance modeling in CUDA streams–A means for high-throughput data processing," in *Proc. IEEE Int. Conf. Big Data*, 2014, pp. 301–310.

[9] T. T. Yeh, A. Sabne, P. Sakdhnagool, R. Eigenmann, and T. G. Rogers, "Pagoda: Fine-grained GPU resource virtualization for narrow tasks," in *Proc. 22nd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2017, pp. 221–234.

[10] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style GPU programming for GPGPU workloads," in *Proc. Innovative Parallel Comput.*, 2012, pp. 1–14.

[11] W. Li, G. Jin, X. Cui, and S. See, "An evaluation of unified memory technology on NVIDIA GPUs," in *Proc. 15th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2015, pp. 1092–1098.

[12] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in CUDA," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2014, pp. 1–6.

[13] M. Steinberger, M. Kenzel, P. Boechat, B. Kerbl, M. Dokter, and D. Schmalstieg, "Whippletree: Task-based scheduling of dynamic workloads on the GPU," *ACM Trans. Graph.*, vol. 33, 2014, Art. no. 228.

[14] Z. Zheng, C. Oh, J. Zhai, X. Shen, Y. Yi, and W. Chen, "VersaPipe: A versatile programming framework for pipelined computing on GPU," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2017, pp. 587–599.

[15] J. Zhong and B. He, "Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1522–1532, Jun. 2014.

[16] Z. Zheng, C. Oh, J. Zhai, X. Shen, Y. Yi, and W. Chen, "HiWayLib: A software framework for enabling high performance communications for heterogeneous pipeline computations," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2019, pp. 153–166.

[17] A. Berson and S. J. Smith, *Data Warehousing, Data Mining, and OLAP*. New York, NY, USA: McGraw-Hill, 1997.

[18] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, "Weaving relations for cache performance," in *Proc. 27th Int. Conf. Very Large Data Bases*, 2001, pp. 169–180.

[19] G. Lu, Y. J. Nam, and D. H. Du, "BloomStore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash," in *Proc. IEEE 28th Symp. Mass Storage Syst. Technol.*, 2012, pp. 1–11.

[20] A. V. Adinetz and D. Pleiter, "Halloc: A high-throughput dynamic memory allocator for GPGPU architectures," in *Proc. GPU Technol. Conf.*, 2014, pp. 1–59.

[21] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg, "ScatterAlloc: Massively parallel dynamic memory allocation for the GPU," in *Proc. Innovative Parallel Comput.*, 2012, pp. 1–10.

[22] J. A. Stratton *et al.*, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," Uni. Illinois at Champaign, Champaign, IL, Tech. Rep. IMPACT-12–01, 2012.

[23] C. Root and T. Mostak, "MapD: A GPU-powered big data analytics and visualization platform," in *Proc. Int. Conf. Comput. Graph. Interactive Techn.*, 2016, pp. 1–2.

[24] S. Zhang, J. He, B. He, and M. Lu, "OmniDB: Towards portable and efficient query processing on parallel CPU/GPU architectures," in *Proc. IEEE Symp. Very Large-Scale Data Anal. Vis. Endowment*, 2013, pp. 1374–1377.

[25] P. E. O'Neil, "The set query benchmark," The Benchmark Handbook, pp. 1–39, 1993.

[26] 2019. [Online]. Available: https://www.memsql.com/

[27] I. Gelado, J. Cabezas, N. Navarro, J. E. Stone, S. J. Patel, and W. W. Hwu, "An asymmetric distributed shared memory model for heterogeneous parallel systems," in *Proc. 15th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2010, pp. 347–358.

[28] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "GPUfs: Integrating a file system with GPUs," *ACM Trans. Comput. Syst.*, vol. 32, 2014, Art. no. 1.

[29] J. Veselỳ, A. Basu, A. Bhattacharjee, G. H. Loh, M. Oskin, and S. K. Reinhardt, "Generic system calls for GPUs," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit.*, 2018, pp. 843–856.

[30] S. Kato, M. McThrow, C. Maltzahn, and S. A. Brandt, "Gdev: First-class GPU resource management in the operating system," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2012, Art. no. 37.

[31] M. Silberstein *et al.*, "GPUnet: Networking abstractions for GPU programs," *ACM Trans. Comput. Syst.*, vol. 34, 2016, Art. no. 9.

[32] M. S. Orr, B. M. Beckmann, S. K. Reinhardt, and D. A. Wood, "Fine-grain task aggregation and coordination on GPUs," in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit.*, 2014, pp. 181–192.

[33] J. Power, Y. Li, M. D. Hill, J. M. Patel, and D. A. Wood, "Toward GPUs being mainstream in analytic processing: An initial argument using simple scan-aggregate queries," in *Proc. 11th Int. Workshop Data Manage. New Hardware*, 2015, pp. 1–8.

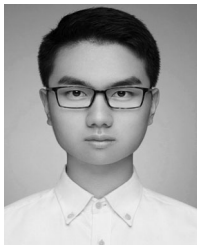
[34] C.-H. Hong, I. Spence, and D. S. Nikolopoulos, "GPU virtualization and scheduling methods: A comprehensive survey," *ACM Comput. Surv.*, vol. 50, 2017, Art. no. 35.

[35] Y. Hao, Z. Fang, G. Reinman, and J. Cong, "Supporting address translation for accelerator-centric architectures," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2017, pp. 37–48.

[36] J. Hestness, S. W. Keckler, and D. A. Wood, "GPU computing pipeline inefficiencies and optimization opportunities in heterogeneous CPU-GPU processors," in *Proc. IEEE Int. Symp. Workload Characterization*, 2015, pp. 87–97.

[37] F. Ji, H. Lin, and X. Ma, "RSVM: A region-based software virtual memory for GPU," in *Proc. 22nd Int. Conf. Parallel Architectures Compilation Techn.*, 2013, pp. 269–278.
[38] S. Shahar, S. Bergman, and M. Silberstein, "ActivePointers: A case for software address translation on GPUs," *ACM SIGOPS Operating Syst. Rev.*, vol. 51, pp. 84–95, 2018.
[39] J. A. Stuart and J. D. Owens, "Efficient synchronization primitives for GPUs," pp. 1–12, 2011, *arXiv:1110.4623*.
[40] D. Lustig and M. Martonosi, "Reducing GPU offload latency via fine-grained CPU-GPU synchronization," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit.*, 2013, pp. 354–365.
[41] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans, and O. Villa, "Combining HW/SW mechanisms to improve NUMA performance of multi-GPU systems," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchit.*, 2018, pp. 339–351.
[42] N. Agarwal, D. Nellans, E. Ebrahimi, T. F. Wenisch, J. Danskin, and S. W. Keckler, "Selective GPU caches to eliminate CPU–GPU HW cache coherence," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 494–506.
[43] C. Gregg, J. Dorn, K. M. Hazelwood, and K. Skadron, "Fine-grained resource sharing for concurrent GPGPU kernels," in *Proc. 4th USENIX Conf. Hot Top. Parallelism*, 2012, Art. no. 10.
[44] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded GPU," in *Proc. IEEE/ACM Int. Conf. Green Comput. Commun. and Int. Conf. Cyber Phys. Soc. Comput.*, 2010, pp. 344–350.
[45] S. Dublish, V. Nagarajan, and N. Topham, "Poise: Balancing thread-level parallelism and memory system performance in GPUs using machine learning," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2019, pp. 492–505.

Zhifang Li received the bachelor's degree from East China Normal University, Shanghai, China, in 2016, where he is currently working toward the PhD degree. His research interests include real-time analytical system, heterogenous computing system, and distributed database system.



Beicheng Peng received the bachelor's degree from East China Normal University, Shanghai, China, in 2018, where he is currently working toward his master's degree. His research interests include massively-parallel computer system, heterogenous computing, and distributed database system.

Chuliang Weng received the PhD degree from Shanghai Jiao Tong University, Shanghai, China, in 2004. He is currently a professor in East China Normal University (ECNU), Shanghai, China. Before joining ECNU, he worked at Huawei Central Research Institute and was an associate professor in Shanghai Jiao Tong University, Shanghai, China. He was also a visiting research scientist in Columbia University, New York City, New York. His research interests include parallel and distributed systems, storage systems, OS, and system security.

Request Flow Coordination for Growing-Scale Solid-State Drives

Ming-Chang Yang, *Member, IEEE*, Yuan-Hao Chang, *Senior Member, IEEE*, Tei-Wei Kuo, *Fellow, IEEE*, and Chun-Feng Wu, *Student Member, IEEE*

Abstract—Performance-intensive applications have led both interface and architecture changes of high-end, growing-scale solid-state drives (SSDs). However, we observe that most of the time, the actual drive performance could not be easily scaled or boosted up with the increasing of internal resources of growing-scale SSDs due to the potential congestion of I/O requests. Such observation inspires this article to look for a request flow coordination design to appropriately control and throttle the I/O request over the increasingly-complicated SSD internal organization with manageable coordination overhead. The main objective is to avoid overloading or congesting any sub-module of growing-scale SSDs by making good use of the abundant internal resources, so as to effectively improve the drive performance in terms of the request-response time. The capability of the proposed design was evaluated with realistic and intensive I/O workloads, and the results are very encouraging.

Index Terms—Solid-state drive (SSD), flash memory, request flow coordination, growing-scale

1 Introduction

More and more performance-intensive applications have begun to employ flash-memory based solid-state drives (SSDs) through high speed interfaces such as PCI Express [20] and NVM Express [17]. In order to better bridge the performance gap between the high speed interfaces and flash-memory, contemporary high-end SSDs have also started to adopt a new internal architecture to interconnect hundreds to thousands of flash chips with multiple I/O channels and multiple control units [5], [12], [24]. Thanks to this new many-chip and many-core architecture, the capacity of SSDs is getting easier and easier to be scaled up by introducing more flash-memory chips and control units into a single SSD device in a modular way, while the management complexity is still manageable. Nevertheless, under our investigation, the performance of SSD on serving I/O requests could not be easily scaled or boosted up as well, in terms of the request response time, even when there are more and more internal resources (i.e., flash chips and control units). This is because it is hard to always fully utilize the abundant internal resources to service I/O requests in parallel merely based on some static request direction or data partitioning policies [23]. In other words, how to appropriately control and throttle the I/O request flows over the increasingly complicated SSD internal architecture becomes the key to optimize the overall SSD performance. This emerging design issue in growing-scale SSDs strongly motivates this work to look for a new design, which can take full advantage of the abundant internal resources to service I/O requests and avoid congesting any sub-module of growing-scale SSDs.

As shown in Fig. 1, a traditional flash-based SSD is usually attached to the host system through serial bus interfaces (e.g., SATA [22]), and incorporates a controller to manage the internal flash chips. In general, each chip is composed of multiple dies (e.g., four dies per chip), each die is further composed of few planes (e.g., two planes per die), and each plane might consist of thousands of blocks and each block is usually of hundreds of pages (e.g., 128 or 256 pages per block) [12]. Notably, each plane usually contains a "page register" (which is usually made of faster memory technology such as SRAM) to temporarily keep the data which will be written to or read from the demanded flash page, so that the controller can simultaneously access multiple pages on multiple dies in a parallel manner. On the other hand, flash-memory can only be read or written in the unit of flash pages while all flash pages of the same flash block need to be erased at a time; meanwhile, every flash block can only endure a limited number of erase operations, or the internal memory cells might not be able to correctly retain bits of data. Therefore, SSDs usually implement an embedded software module, called *flash translation layer (FTL)* [4], [6], [15], [21], to perform different management routines (such as address translation, garbage collection, and wear leveling), so as to overcome various hardware constraints or limitations of flash-memory chips and to ultimately improve the device performance and reliability.

- *M.-C. Yang is with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, NT, Hong Kong. E-mail: mcyang@cse.cuhk.edu.hk.*
- *Y.-H. Chang is with the Institute of Information Science, Academia Sinica, Taipei 115, Taiwan (R.O.C.). E-mail: johnson@iis.sinica.edu.tw.*
- *T.-W. Kuo is with the College of Engineering, City University of Hong Kong, Hong Kong, and also with the Department of Computer Science and Information Engineering, National Taiwan University, Taipei 10617, Taiwan. E-mail: ktw@csie.ntu.edu.tw.*
- *C.-F. Wu is with the Department of Computer Science and Information Engineering, National Taiwan University, Taipei 106, Taiwan. E-mail: cfwu@iis.sinica.edu.tw.*

Manuscript received 1 July 2019; revised 25 Nov. 2019; accepted 8 Jan. 2020. Date of publication 21 Jan. 2020; date of current version 8 May 2020. (Corresponding author: Ming-Chang Yang.) Recommended for acceptance by H. Jiang. Digital Object Identifier no. 10.1109/TC.2020.2968439

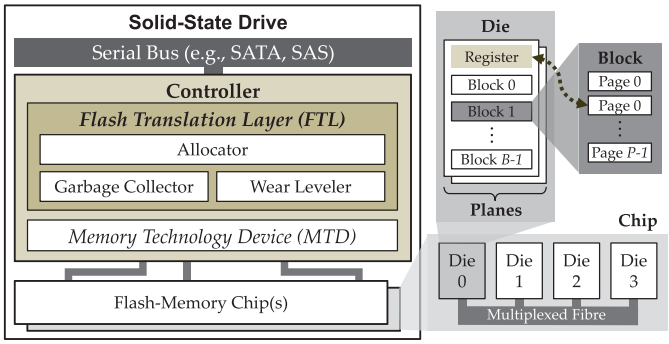


Fig. 1. Traditional architecture of flash-based SSDs.

In the past decades, the flash management designs have drawn the most of attentions since its core design, the address translation strategy, could significantly affect the device performance. To achieve a good balance between the required on-device DRAM space (for maintaining the management information) and the access performance, many FTL designs have been proposed [4], [6], [15], [21]. In particular, DFTL [6], which maintains the page-level mapping information and dynamically loads/stores the required translation information between flash memory and the RAM space on demand, is a currently widely-adopted FTL design because of its simplicity and efficiency. Additionally, various garbage collection strategies [13], [26], [27] and wear leveling designs [2], [18] have also been widely investigated to improve the device performance and extend the device lifetime, respectively. More recently, as the number of flash chips keeps increasing in an SSD, many excellent researches tried to fully take advantage of the internal access parallelism to improve the device performance [3], [8], [14], [30]. Even though some proposed to redirect reads to improve the read performance for SSDs [16] and some proposed to maximize resource utilization and achieve high performance for many-chip SSDs [11], these approaches are mainly designed for the SSDs adopting the traditional architecture, while the special features of the new architecture of high-end SSDs are out of their major design considerations.

On the other hand, request redirection has been applied to many usage scenarios in storage systems. The existing designs perform request redirection based on different criteria or constraints. For example, in [19], write requests on "spun-down disks" will be temporarily redirected to persistent storage elsewhere in the data center for further reduction of power consumption. As another example in [28], [29], write requests on the "degraded RAID set" (i.e., the RAID set is performing the background tasks) will be outsourced to a surrogate RAID set for the alleviation of user performance degradation. However, most of the existing request redirection/out-sourcing solutions are mainly designed for disk-based storage systems, while this paper is one of the pioneers to investigate how to enable request redirection in the new architecture of high-end SSDs.

Different from the existing designs for either traditional SSDs or disk-based storage systems, this paper is strongly inspired by the interface and architecture changes of the high-performance and growing-scale SSDs [5], [12], [24]. We are especially interested in how to achieve good design scalability to facilitate the device development when the scale of SSDs keeps growing. That is, instead of redesigning yet another new architecture, we tend to embrace the new many-chip and many-core SSD architecture [5], [12], [24]. In the meantime, to improve the SSD performance, in terms of the request response time, in this paper, a new *request flow coordination* design is presented to appropriately control and throttle the I/O request flows over the increasingly complicated SSD internal organization with scalable and manageable coordination overhead. Our key idea is to avoid overloading any sub-module of the SSD by making a good use of the abundant internal resources, so that the request response time can be effectively reduced and the overall SSD performance can be significantly improved. The technical novelty and contributions of this paper are twofold. First, our design accurately monitors and estimates the "loading" of a module by leveraging the device-level queueing capability of the contemporary flash module controller. Second, our design realizes efficient request redirection among flash modules by deploying the multiple control units in the multi-module architecture as a "master-slave" architecture. Our evaluation results show that the proposed design could achieve significant write performance improvement with nearly no read performance degradation and very limited request redirection overhead. Specifically, under the evaluated I/O traces with 16 times of workload intensity, the proposed design could significantly improve the write performance by at least 67 percent when the number of flash modules was doubled, while only about extra 1.5 percent performance delay was introduced on performing request redirections.

The rest of this paper is organized as follows: Section 2 presents the architecture changes of high-performance SSDs and the motivation regarding this work. In Section 3, the request flow coordination design is introduced with the adaptive request redirection policy and the discussions of potential design challenges and our practical solutions. Then Section 4 presents the evaluation results, and Section 5 concludes this work.

2 Motivation: Architecture Changes of SSDs

Recently, more and more data-intensive and high performance computing applications have begun to employ the high-end SSDs through high speed interfaces like PCI Express [20] or NVM Express [17], so as to break the performance bottleneck of the conventional storage interface. In addition, in order to bridge the performance gap between the high speed interface [10] (e.g., 16 GB/sec) and flash-memory [31] (e.g., 40 $\sim$ 400 MB/sec), modern SSDs have begun to interconnect hundreds to thousands of flash chips to form a single device to increase the potential maximal access bandwidth [11]. Furthermore, as the new many-chip architecture presented by Marvell [24], these massive flash chips amount are interconnected with multiple I/O channels and multiple control units, so that neither the data transmission buses nor the control units would become the performance bottleneck [11].

When we look closer, as illustrated in Fig. 2, in this new "many-chip and many-core" architecture [5], [12], [24], each control unit is interconnected to the high speed data bus through a dedicated I/O channel, and each controller implements the individual FTL routines to manage the flash chips

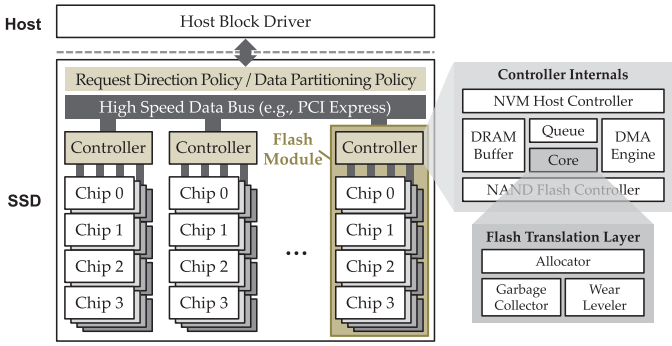


Fig. 2. New "many-chip and many-core" architecture of growing-scale SSDs.

attached to it. Please note that, to simplify the following discussion, in the rest of this paper, a "flash module" will denote a control unit and the corresponding flash chips attached to it. Thus, the many-chip and many-core architecture will be simply referred to as the "multi-module architecture". On the other hand, because of the increasingly complicated internal organization of SSDs, this new architecture requires a *static request direction policy or data partitioning policy [23], [25]* to efficiently forward the received I/O requests to the in-charge flash module without causing performance delay/latency on forwarding and directing I/O requests. For example, one simple yet efficient way to define the request direction policy is to partition the entire logical storage space into multiple successive and fixed-sized pieces based on the total number of flash modules, where each flash module only needs to service the I/O requests belonging to its pre-defined range of logical space.

Moreover, this new multi-module architecture ensures very good design scalability[1] in three aspects. First, under this architecture, the storage capacity and the maximal bandwidth of a single device can be easily scaled up by simply including more flash modules. Second, even when the storage capacity keeps increasing (by employing more flash modules), the management complexity is still highly manageable and scalable. The rationale behind this is that each module is only required to serve the requests belonging to its corresponding logical space, and each module can utilize the dedicated hardware resources (e.g., the control unit) to perform its own FTL routines over the attached flash chips. Third, the internal, high access parallelism of each flash module can be still efficiently exploited based on many excellent studies about "RAID[2] inside SSD" [8].

Nevertheless, based on our investigation, even though a higher bandwidth can be provided, the actual SSD performance might not be easily scaled up (i.e., speeded up) as well, even when a larger number of modules is adopted. This is because *there might be no any specific "optimal" request direction policy, which could evenly distribute the I/O requests among flash modules for all kinds of access patterns so as to completely prevent some certain modules from becoming the performance bottleneck*. In particular, as shown in Table 2 (please find in Section 4.1), in real server workloads, different storage volumes of the same service are indeed composed of workloads of quite different access patterns. Furthermore, when the number of modules keeps increasing to form an ultra-scale SSD, it would be even harder to balance the loadings among the enormous number of modules by merely adopting some simple request direction policies. As a potential consequence, even if the number of modules keeps increasing, the actual performance of the multi-module architecture could not be accordingly improved in practice, since the modules receiving the most of I/O loadings would inevitably become the performance bottleneck of an SSD.

According to the aforementioned discussion, we are largely interested in how to efficiently coordinate the I/O requests over the increasingly complicated internal organization so as to effectively scale up the overall performance for the growing-scale SSDs. In particular, our objective is to optimize the device performance by proposing a *request flow coordination* design, on top of the multi-module architecture, so that the proposed design can still embrace the good design scalability of the multi-module architecture. The main contribution of the proposed design is to properly redirect I/O requests among flash modules to avoid overloading any flash module, so that the write request response time can be significantly improved. Notably, different from the existing data partitioning and load balancing designs for multiple disks or parallel disk systems [23], our design is a device-level solution for growing-scale SSDs. That is, our design is able to exploit the internal hardware structures and internal flash management information to essentially improve the performance for SSDs. The major technical challenge is on how to realize the request redirection with manageable overhead. Specifically, it is especially important to figure out an efficient way to monitor and quantize the runtime loadings of flash modules, so as to effectively avoid overloading any flash module by timely redirecting write requests among flash modules. In addition, since the write requests could be possibly redirected to any module in the SSD, how to efficiently retrieve the requested data without exhaustedly searching over all modules would also largely affect the device performance.

3 Request Flow Coordination

3.1 Overview

In this section, we present a *request flow coordination* design to improve the performance for the growing-scale SSDs by adaptively redirecting the I/O requests among flash modules. As shown in Fig. 3, the proposed design re-deploys the multiple control units in the multi-module architecture as a "master-slave" architecture. That is, the proposed design treats the controllers of flash modules as slave controllers", and employs an extra controller as the "master controller" to coordinate the slave controllers. Notably, to keep the good design scalability enabled by the existing multi-module architecture, the master controller should direct the requests to the corresponding flash modules according to the predefined request direction policy; meanwhile, the slave controller of each flash module should be in charge of servicing the I/O requests belonged to its pre-assigned logical space and performing FTL routines to manage the attached flash chips. Therefore, even if the proposed design is adopted, the total storage capacity can be still easily scaled up by incorporating more flash modules and the management complexity

1. Scalability is the capability to handle a growing amount of work, or the potential to be enlarged in order to accommodate that growth.
2. RAID is short for Redundant Array of Independent Disks.

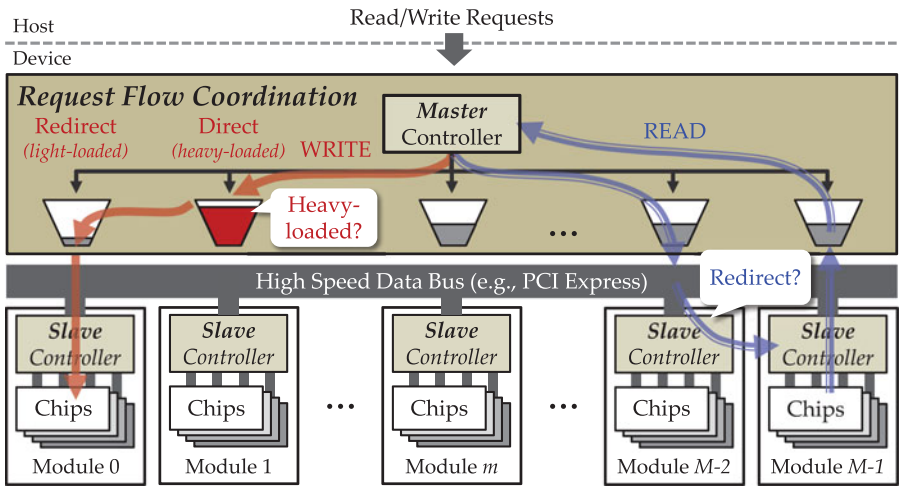


Fig. 3. An overview of the proposed request flow coordination design.

is still highly manageable and scalable, as compared to the existing multi-module architecture.

In order to further make a good use of the plentiful internal resources, the proposed design defines new routines for the master controller to coordinate I/O request flows among slave flash modules through *request redirection*:

- *Performance Monitoring:* The master controller should monitor and quantize the run-time performance status (i.e., the current loading) over all underlying flash modules, so as to timely detect the "heavy-loaded" flash modules that might result in the performance bottleneck.
- *Write Request Redirection:* To prevent any flash module from becoming the performance bottleneck due to the "instant congestion" of I/O requests on a certain flash module, the master controller should adaptively redirect some write requests from those "heavy-loaded" modules to some proper "light-loaded" flash modules based on the monitored run-time performance status of flash modules.
- *Read Request Redirection:* When the requested data of a read request has already been redirected to other modules, the master controller should efficiently communicate with slave modules to locate the requested data without exhaustedly searching over all flash modules and degrading the read performance.
- *Redirected Request Recall:* However, when the requested data are redirected, the response time of a read request will be inevitably lengthened. Thus, the master controller should also minimize the overall redirection overhead imposed on slave modules by properly recalling the redirected write requests when the original module is no longer heavy-loaded without introducing additional data migration overhead.

On the other hand, in order to efficiently perform read/write request redirection, the proposed design requires slave controllers to maintain additional "redirection information". Notably, according to the source and destination of the write request redirection, the (slave) flash modules can be further classified into two categories: The *direct module* is the flash module which the request is belonged to, based on the pre-defined request direction policy. In contrast, the *redirect module* is the flash module which the request is redirected to. Thus, besides of conventional FTL routines, the proposed design also defines what kind of new redirection information should be additionally maintained by the direct modules and redirect modules as follows:

- *Direct Module:* When a write request is redirected, the corresponding direct module should keep track of the destination of the redirect module for every redirected write request, so as to help the master controller to efficiently locate the redirected data for future read accesses.
- *Redirect Module:* The slave controller of the redirect module should also maintain additional mapping information for all redirected data, so as to ensure the redirected data can be correctly retrieved from the right physical pages of the module.

With the help of these newly defined routines and the additional redirection information, the proposed design can effectively utilize all flash modules to handle the received I/O requests to reduce the request response time and improve the device performance. Furthermore, we argue that when the workload intensity or the number of slave modules keeps increasing, the proposed two-level, master-slave architecture can be easily scaled up to three-level (and even multi-level) architectures by incorporating more master controllers. By doing so, each master controller only needs to handle the I/O requests belonging to a limited range of logical address space; meanwhile, none of the master controllers will become the potential performance bottleneck. In the following section, Section 3.2 will first illustrate the *adaptive redirection policy* of the proposed design in details. Then, Section 3.3 shall point out the technical challenges in realizing the proposed design with practical solutions. Finally, Section 3.4 will give property analysis of the proposed design in terms of the request coordination overhead and the response time reduction.

3.2 Adaptive Request Redirection Policy

This section presents the core design philosophy of the proposed request flow coordination: a *write request redirection policy* to adaptively redirect write requests among slave modules to reduce the request response time, and a *read*

request redirection policy to efficiently retrieve the requested data without exhaustedly querying over all slave modules.

Algorithm 1 illustrates how the WRITE REQUEST REDIRECTION POLICY could properly redirect write requests among slave modules to avoid overloading some certain salve modules. First, when a write request req is received, the master controller will find the corresponding direct module $module_{direct}$, based on the pre-defined request direction policy (Line 1). Notably, as mentioned in Section 2, the pre-defined direction policy is used to define how to forward read/write requests to the corresponding flash modules. Then, instead of directly forwarding the req to the corresponding $module_{direct}$, the master controller will check the current performance status of $module_{direct}$ so as to avoid overloading it (Line 5). If the $module_{direct}$ turns out to be comparatively heavy-loaded than other modules (Lines 5–8), the master controller will look for a comparatively light-loaded module $module_{redirect}$ as the redirect module, and then redirect the req to $module_{redirect}$. Notably, when selecting a light-loaded module, the master controller should also consider the logical storage utilization of modules, because a module can only store a fixed amount of logical data. If req is redirected, the master controller should also ask the $module_{direct}$ to update the redirection information to record that the req has been redirected to $module_{redirect}$ (Line 8). By doing so, when future read requests try to retrieve data of req, the $module_{direct}$ can help the master controller to efficiently redirect the read requests to the corresponding $module_{redirect}$ without searching over all modules. Otherwise, if the $module_{direct}$ is not currently (or no longer) heavy-loaded, the req should be directly forwarded to the corresponding $module_{direct}$ to avoid introducing too much redirection traffics (Lines 9–10).

Algorithm 1. WRITE REQUEST REDIRECTION POLICY

Input: req: The received write request.
1 Find the corresponding $module_{direct}$ of req based on the pre-defined direction policy;
2 **if** *req was previously redirected* **then**
3 Look up the previously redirected $module_{redirect}$;
4 Recall req from $module_{redirect}$;
5 **if** $module_{direct}$ is comparatively heavy-loaded **then**
6 Look for a comparatively light-loaded $module_{redirect}$ among other flash modules;
7 Redirect req to $module_{redirect}$;
8 Update the redirection information of $module_{direct}$;
9 **else**
10 Direct req to $module_{direct}$;

Notably, if the req was previously redirected into other modules, we should always "recall' the previously-redirected data from the previously redirected module $module_{redirect}$ so as to minimize the overall redirection traffics (Lines 2–4). Meanwhile, when the new version of data of the redirected request is received (as a new write request), it is also the best timing to recall the redirected request since no additional data migration overhead will be introduced. Specifically, the new version of data of req will be either directed into $module_{direct}$ or redirected into another $module_{redirect}$ based on the current loading of the $module_{direct}$, while the old version of data remained in the previous $module_{redirect}$ could be directly discarded without migration. Moreover, it is also worthy to note that recalling a redirected request could reduce not only the redirection management overhead but also the garbage collection overhead for the previously-redirected module $module_{redirect}$, since the garbage collection overhead has been proved to be positively correlated to the total amount of valid data stored in an SSD [7]. Thus, recalling and discarding the previously redirected data could indirectly improve the garbage collection performance for the previously-redirected module $module_{redirect}$.

On the other hand, Algorithm 2 depicts the READ REQUEST REDIRECTION POLICY to elaborate how to efficiently handle read requests, especially when some requested data might have been already redirected into some other modules rather than its direct module. First, when a read request req is received, the master controller should find the corresponding direct module $module_{direct}$ as what the WRITE REQUEST REDIRECTION POLICY does in the beginning (Line 1). Then, the master controller will communicate with the slave controller of the direct module $module_{direct}$ to check whether the requested data can be directly retrieved from it (Line 2). If not (Lines 2–4), the master controller will further ask the $module_{direct}$ about what is the redirect module $module_{redirect}$ where the requested data have been currently redirected to (Line 3). Then, the master controller can forward/redirect the req to $module_{redirect}$ to efficiently retrieve the requested data, rather than querying over all flash modules (Line 4). Otherwise, if the requested data stay in its pre-defined direct module, the master controller will retrieve the requested data directly from the corresponding $module_{direct}$ (Lines 5–6).

Algorithm 2. READ REQUEST REDIRECTION POLICY

Input: req: The received read request.
1 Find the corresponding $module_{direct}$ of req based on the pre-defined direction policy;
2 **if** *the requested data of req is currently not in* $module_{direct}$ **then**
3 Look up the $module_{redirect}$ of req by asking $module_{direct}$;
4 Retrieve the requested data of req from $module_{redirect}$;
5 **else**
6 Retrieve the requested data of req from $module_{direct}$;

Notably, *the two proposed policies could ensure good design scalability, since both policies would at most involve two flash modules (i.e., $module_{direct}$ and $module_{redirect}$) to redirect a read/write request, no matter how many flash modules are interconnected in the SSD*. The trick behind this property is that even if the master controller might redirect a write request to any slave module, the direct module will keep track of the redirect module(s) for all data of its pre-defined logical space, so that the master controller can efficiently locate the requested data by only querying the corresponding "direct module".

3.3 Challenges and Implementations

3.3.1 Challenges for Master Controller

The two most critical design challenges for the master controller are about 1) how to represent and monitor the current "loading" of a slave module, and 2) how to determine a slave module is currently "heavy-loaded" or "light-loaded".

First, in order to accurately represent and estimate the loading for flash modules, the proposed design takes

advantage of the device-level queueing capability of the contemporary flash module controller. As shown in Fig. 2, the controller of flash modules usually incorporates a device-level queue (a.k.a. native command queuing [9]) to buffer I/O requests, and is thereby able to reschedule the queued I/O requests to fully utilize the internal parallelism and optimize the performance [11]. Therefore, a practical and effective way to represent the current loading of flash modules is to sum up the total amount of the queued flash commands in the command queue (i.e., the total amount of flash page reads and writes, and flash block erases). Alternatively, in order to have a more accurate loading estimation, we can calculate the actual *blocking time* for each slave module, by summing up the time latencies introduced by the queued flash commands, to see how long we should actually wait before the new I/O request could be serviced. Notably, since the read/write/erase flash command has quite different time latencies, in our implementation, we adopt the actual blocking time to more accurately represent the loading of a flash module. In addition, the loading of flash modules can be immediately updated whenever the command queue is refreshed to keep the most accurate estimation; instead, the loading of flash modules can be also periodically updated to reduce the overheads in monitoring the current loading, even though the loading/performance monitoring is in fact a light-weight process in terms of execution time, compared with the flash read/write/erase commands.

More importantly, if every slave controller can dynamically monitor and report its loading to the master controller, the master controller can easily have good loading controls over all modules and efficiently determine whether a slave flash module is currently heavy-loaded or not. For example, the master controller can calculate an "average loading" over all modules, so that modules whose loadings are heavier (resp. to lighter) than the average loading can be regarded as heavy-loaded (resp. to light-loaded) modules. Moreover, to avoid unnecessary redirection traffics when all modules are equally busy, we can further calculate the difference of blocking time between the direct module and the module having the minimal blocking time. If this time difference is large enough (e.g., larger than one standard deviation), we can regard the direct module as a "sufficient heavy-loaded" module, so that it is beneficial to perform request redirection. Section 3.4.1 will give an analysis on the overhead of the proposed design to better understand when is beneficial to trigger the proposed request redirection for having response time reduction. Moreover, a heap-like data structure can be also used to quickly find out the lightest-loaded module (i.e., the module having the shortest blocking time) to have a good candidate of redirect module.

3.3.2 Challenges for Slave Controller

In order to effectively service the directed and redirected write requests, every slave controller needs to maintain two types of mapping tables: *direct table* and *redirect table*. In general, very similar to the common address translation table of various FTL designs [1], [6], the functionality of both types of tables is to translate the logical block addresses (LBAs) of data to the corresponding physical block addresses (PBAs) of flash-memory chips. First, the direct table is designed to keep track of all data whose logical block addresses belong

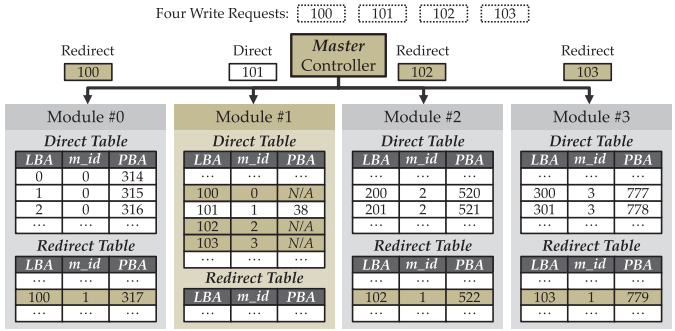


Fig. 4. Examples of direct and redirect tables.

to the pre-defined logical space of a slave flash module, no matter the data are directly stored in this module or have been redirected to other modules. In other words, the direct table provides the address translation functionality to record the physical block addresses for the directed data; meanwhile, the direct table also records the redirect module information for the redirected data. Therefore, when a read request is received, the master controller can always query the corresponding direct module to look up its direct table to know where (specifically, which module) to retrieve the requested data, rather than exhaustedly searching over all modules. In contrast, the redirect table is used to maintain the mapping information for the data redirected from other flash modules to the flash-memory space of this module, so as to provide the address translation information for the redirected data.

Fig. 4 shows an illustrative example to elaborate how the direct and redirect tables facilitate the read/write request redirection. In this example, both tables are implemented as the *page-level mapping tables*, which are widely adopted used in the popular page-level FTL and DFTL [1], [6] because of its better translation performance. Notably, the required memory/storage space to maintain both tables is limited, because a module can only store a limited amount of user data. Besides, every entry of both tables consists of three fields: *LBA*, *m_id*, and *PBA*. The fields *LBA* and *PBA* in both tables represent the logical block addresses of data and the physical address of the flash-memory chips, respectively. However, the field *m_id* should be interpreted differently in the two tables. In direct tables, the field *m_id* is used to record the module identification (id) where the corresponding data currently reside; while in redirect tables, the field *m_id* is used to record the module id where the corresponding data are redirected from.

As the example illustrated in Fig. 4, four write requests are received to write data of LBAs 100 to 103 in the module #1 (because these LBAs belong to module #1 based on the pre-defined direction policy). Assume only data of LBA 101 are directly written in the module #1, while data of LBAs 100, 102, and 103 are redirected into modules #0, #2, and #3, respectively. Under this scenario, the slave controller of the module #1 will update the entries of LBAs 100 to 103 in the direct table to record 1) the data of LBA 101 are stored in the storage space of PBA 38 and 2) the data of LBAs 100, 102, and 103 are redirected into modules #0, #2, and #3, respectively (by updating the fields *m_id*). In the meantime, the slave controllers of the module #0, #2, and #3 also need to update their redirect tables to record that the data of

LBAs 100, 102, and 103 are stored in its own storage space of PBAs 317, 522, and 779, respectively. Therefore, when these data are requested, data of LBA 101 can be directly retrieved from its direct module (i.e., module #1) by looking up the direct table; while read requests to data of LBAs 100, 102, and 103 will be efficiently retrieved from the storage space of PBAs 317, 522, and 779 in modules #0, #2, and #3, respectively.

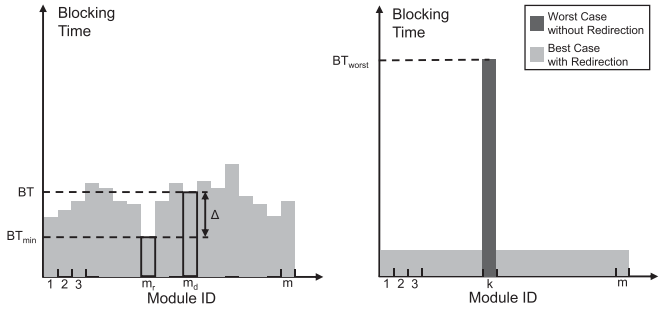
In practice, both direct table and the redirect table should be persisted in flash memory, and only the required address translation and redirection information will be dynamically loaded/stored between the limited on-device DRAM space and the flash memory. We propose to implement the direct table as *linear table*, which is widely adopted in popular page-level FTL and DFTL [1], [6], since the linear table can provide the most efficient table lookup performance and an efficient way for table load/store [6]. In addition, since our design attempts to minimize the overall redirection traffics, most of data will be directed into its direct module and the most of entries in the direct table will be efficiently used. By contrast, it would be very inefficient to implement redirect table as linear table, since most of table entries in the redirect table might be unused due to the limited redirection traffics. Therefore, instead of using linear table, we propose to implement the redirect table as *tree structure*, such as *B+ tree*, so as to avoid wasting flash memory space to maintain a huge amount of unused entries while still provide efficient lookup performance. Specifically, assuming a flash page can keep E redirection entries, the size of a leaf node in the B+ tree can be set to E so that the redirection information can be efficiently loaded/stored between DRAM and flash pages. Furthermore, implementing redirection table as B+ tree only demands for very limited DRAM space to keep the tree structure (i.e., indexing) of B+ tree, while all the redirection information pointed by the leaf nodes can be persisted in flash memory.

3.4 Property Analysis

This section is going to analyse the overhead of the proposed request flow coordination design so as to better determine when is beneficial to trigger the proposed request redirection (please see Section 3.4.1). In addition, this section will also discuss the potential of the proposed design on reducing the request response time, and then demonstrate that our design would have the greater potential to reduce the response time as the number of modules keeps increasing (please see Section 3.4.2). To ease the following discussions, we assume that the out-of-order execution is not considered in this analysis and a write request can be redirected to any other slave modules without losing of generality.

3.4.1 Request Coordination Overhead

This section will give an analysis on the overhead of the proposed design to better understand when is beneficial to trigger the proposed request redirection for having response time reduction. Given a multi-module SSD composed of m slave modules, Fig. 5a shows a possible distribution of the blocking time of m modules at a certain moment, where the x-axis denotes the module identification (id) and the y-axis denotes the current blocking time of each module. Notably,



(a) Analysis of Worst-Case Request Coordination Overhead (b) Analysis of Best-Case Response Time Reduction

Fig. 5. Illustrated analyses of worst-case request coordination overhead and best-case response time reduction.

here the blocking time of a module is defined as the total time latency introduced by all of the queued flash commands in that module.

Then, we consider that a new write request is received, and the module m_d of blocking time BT is the corresponding direct module; meanwhile, the module m_r is selected as the candidate of the redirect module since it has the smallest blocking time BT_{min}. To determine whether to perform the request redirection or not, we need to first evaluate whether we can have positive reduction on the request response time. In general, the request response time can be composed of 1) the blocking time of the target module, 2) the expected look-up time of mapping table, and 3) the request execution time. Thus, when this new write request is serviced by a module (e.g., m_d) without performing the request redirection, the response time, RT_{no}, can be derived as:

$$RT_{no} = BT + [P_{no} \times T_c + (1 - P_{no}) \times (T_r + T_w)] + E,$$

where the blocking time of the direct module m_d and the request execution time are denoted as BT and E, respectively. The expected look-up time of mapping table can be further decomposed into the time (i.e., T_c) to access the mapping information in the mapping cache of the controller with the probability P_{no} and the time to load and store a mapping page between the mapping cache and the flash memory with the probability $1 - P_{no}$, where the time to read and write a flash page are T_r and T_w, respectively. Similarly, if this new write request is redirected to a remote module (e.g., m_r), the response time, RT_{re}, can be derived as:

$$RT_{re} = BT_{min} + 2 \times [P_{re} \times T_c + (1 - P_{re}) \times (T_r + T_w)] + E,$$

where the blocking time of the redirect module m_r is BT_{min}, the request execution time is E, and the probability to access the mapping information in the mapping cache is P_{re}. However, please note that, the expected look-up time of mapping table in this case should be doubled, since performing a request redirection needs to look up the direct mapping table in the direct module and the redirect mapping table in the redirect module.

Obviously, it is beneficial to perform the request redirection, if the response time without request redirection RT_{no} is longer than the response time with request redirection RT_{re}. Here, we can further deduce that it is beneficial to perform the request redirection only if:

$$\begin{aligned}\Delta &= BT - BT_{min} \\ &> 2 \times [P_{re} \times T_c + (1 - P_{re}) \times (T_r + T_w)] \\ &\quad - [P_{no} \times T_c + (1 - P_{no}) \times (T_r + T_w)],\end{aligned}$$

where the blocking time difference between the direct module and redirect module is denoted as Δ in Fig. 5a, and the right-hand side of this inequality can be further considered as the request coordination overhead introduced by our design. Moreover, the worst case of the request coordination overhead could be further derived as $2 \times (T_r + T_w)$ when the probability P_{re} equals to 0 and P_{no} equals to 1, since the time to read and write a flash page (i.e., T_r and T_w) is much longer than the time to access the cache mapping T_c. Nevertheless, please note that, this is just the theoretical worst case of the request coordination overhead. In practice, our evaluations show that only about extra 1.5 percent performance delay was introduced on performing request redirections (please see Section 4.2.3 for more details).

In summary, if the blocking time difference between the direct module and redirect module is larger than the worst case of the request coordination overhead (i.e., $2 \times (T_r + T_w)$), it will always be beneficial to perform the proposed request redirection to effectively reduce the request response time.

3.4.2 Response Time Reduction

This section further discusses the potential of the proposed design on reducing the request response time. To show the greatest potential of the proposed design, we consider a special case, as shown in Fig. 5b, in which the proposed design could gain the largest reduction on the request response time. Given a multi-module SSD composed of m slave modules, the dark grey bars in Fig. 5b show the worst-case distribution of blocking time of m modules at a certain moment, when the proposed request redirection is not applied. Notably, without adopting the proposed request redirection, the worst-case distribution of blocking time may happen when all the requests are queued in a certain module (e.g., module k) so that the module k will have the worst-case blocking time BT_{worst} while the blocking time of any other module equals to 0. In contrast, the light grey bars in Fig. 5b show the best-case distribution of blocking time of m modules that could be achieved by the proposed request redirection policy. That is, if the proposed design could completely and evenly redirect the requests among all flash modules, all the modules would share the worst-case blocking time BT_{worst} and the largest reduction of response time for module k can be derived as:

$$BT_{worst} \times (m - 1)/m,$$

where we assume, under this best case, the request coordination overhead could be ignored since the mapping information could be found in the mapping cache in all slave modules. Furthermore, from this result, we can further infer that when a larger number of modules are equipped, our design would have the greater potential to reduce the response time. This is because a larger m would lead to a larger fraction number of $(m - 1)/m$. In summary, the largest reduction of response time could be achieved by the proposed design if all the internal resources could be fully utilized; moreover, the more modules are adopted, the more response time could be reduced by the proposed design.

TABLE 1
The Setting of a Simulated Flash Module [8]

# of channels	2 (per module)	Total Capacity	64 GB
# of chips	4 (per channel)	Over-provision	Extra 10%
# of dies	4 (per chip)	Queue depth	64
# of planes	2 (per die)	Buffer size	1 MB

4 Performance Evaluation

4.1 Evaluation Setup

In this section, we evaluate the capability of the proposed adaptive request redirection policy (denoted as `Redirection`) with realistic and intensive I/O workloads. In Sections 4.2.1 and 4.2.2, we first investigate the capability of the proposed design in improving the SSD performance in terms of the average response time (ART) (i.e., the average time to complete I/O requests). Specifically, Section 4.2.1 shows the performance changes when the workload intensity kept increasing by speeding up the workloads by different times (i.e., `1x`, `2x`, `4x`, `8x`, and `16x`). Then, Section 4.2.2 shows the performance comparisons when the number of flash modules was doubled. Finally, in Section 4.2.3, we will present the potential performance overhead of the proposed `Redirection` on performing the request redirection.

To implement the multi-module architecture, we extend and revise the public flash simulator, namely `SSDSim` [8], to interconnect multiple control units with multiple flash chips through high-speed interface. Thus, multiple flash modules can be simulated and implemented in a single flash-based SSD. Besides, as shown in Table 1, each simulated flash module is composed of 8 flash chips and the storage capacity of a single simulated flash module is 64 GB (however, please note that the total storage capacity of the simulated SSD is of hundreds of GB, depending on the total number of simulated flash modules); meanwhile, the controller of every simulated flash module can buffer at most 64 read/write requests, and has 1 MB on-device DRAM buffer for maintaining the management information and/or caching pages of read/write data. Notably, in order to have the best internal access parallelism inside a flash module, we adopt all of the suggested settings presented in [8]. In addition, we implemented the most widely-adopted FTL design, i.e., the demand-based FTL (`DFTL`) [6], to manage the simulated flash modules. Notably, `DFTL` on-demand loads/stores the mapping information between the on-device RAM buffer and flash memory to effectively reduce the RAM space consumption with page-level translation performance. Our implementation of `DFTL` follows the suggested settings in [6] to have 1024 cached mapping entries and manage the cached entries by using the segmented LRU policy.

Besides, we use public and realistic I/O workloads, collected from the servers of Microsoft Research Cambridge [19], to evaluate the performance behavior of the simulated multi-module SSD. In particular, three different types of server applications, i.e., the research project servers (`rsrch`), the user file servers (`usr`), and the source controller servers (`src`), are used to evaluate the proposed design. In addition,

TABLE 2
The Statistics of the Evaluated Workloads

rsrch	Volume #0	Volume #1	Volume #2
# Write Requests	1,300,030	13,738	71,223
# Read Requests	133,625	42	136,364
Written Data Size	10.81 (GB)	0.15 (GB)	0.28 (GB)
Read Data Size	133,625 (B)	42 (B)	136,364 (B)
usr	Volume #0	Volume #1	Volume #2
# Write Requests	1,333,406	3,857,714	1,994,612
# Read Requests	904,483	41,426,266	8,575,434
Written Data Size	13.07 (GB)	56.12 (GB)	26.46 (GB)
Read Data Size	35.30 (GB)	2,079.22 (GB)	415.28 (GB)
src	Volume #0	Volume #1	Volume #2
# Write Requests	1,300,030	13,738	71,223
# Read Requests	133,625	42	136,364
Written Data Size	9.33 (GB)	0.17 (GB)	39.28 (GB)
Read Data Size	1.36 (GB)	37.01 (GB)	22.78 (GB)

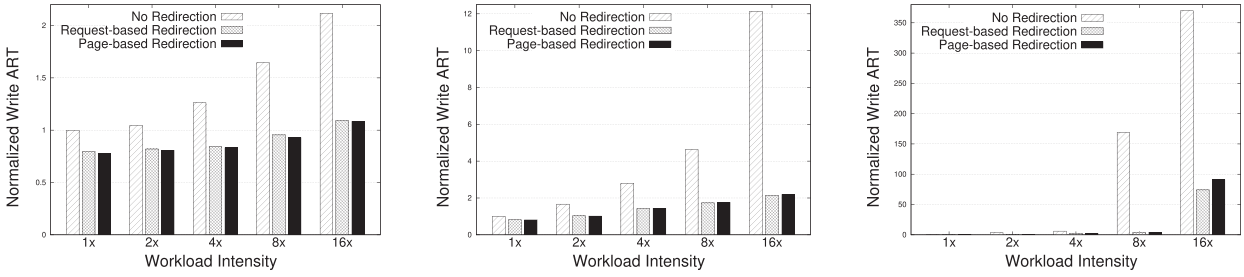
the three types of considered server workloads are composed of three volumes of data, so that the evaluated SSD deployed three flash modules to store the data belonging the corresponding volume, respectively. In other words, a predefined request direction policy [23] is used to forward/direct the read/write request to the corresponding flash module according to the volume identifications of requests. The statistics data of the evaluated workloads are summarized in Table 2. As observed, in real server workloads, different storage volumes of the same service comprise of workloads of quite different access patterns. In the following evaluation results, `No Redirection` denotes the proposed `Redirection` policy was not applied and the requests were just simply forwarded based on the volume identifications; while `Request-based Redirection` and `Page-based Redirection` denote the proposed redirection policy was adopted in the unit of requests or flash pages, respectively.

4.2 Evaluation Results

4.2.1 Various Workload Intensity

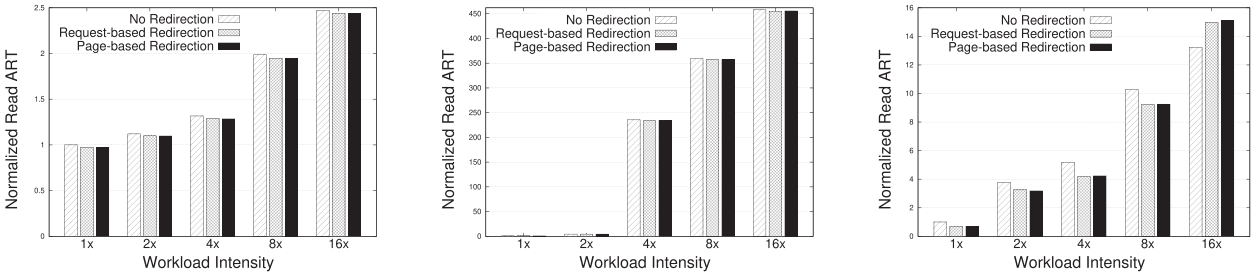
Figs. 6 and 7 show the performance results with and without the proposed `Redirection` policy under various workload intensity. In both figures, the x-axis denotes different workload intensity by speeding up the evaluated workloads by different times (i.e., `1x`, `2x`, `4x`, `8x`, and `16x`), so that we can investigate the performance scalability of the proposed `Redirection` scheme in handling more intensive workloads. On the other hand, in both figures, the y-axis denotes the average response time (ART) normalized to the results of `No Redirection` under `1x` workload intensity, so as to demonstrate the performance changes without losing of generality.

First of all, Fig. 6 clearly shows that, under all evaluated workloads, the proposed `Redirection` policy was effective in improving the write performance for the multi-module SSDs. When the three workloads were replayed as its original timestamps (i.e., `1x` workload intensity), the proposed `Redirection` policy could improve the write average response time by up to 21, 19, and 68 percent under the `rsrch`, `usr`, and `src` server workloads, respectively, as compared with `No Redirection` scheme. This is because the proposed design would adaptively redirect I/O requests among all flash modules, so that no any flash module would be overloaded and become the performance bottleneck. Moreover, as the workload intensity kept increasing (i.e., from `1x` to `16x`), it can be clearly observed that the performance results of `No Redirection` would suffer from severer and severer performance degradations (please see Fig. 6c). Differently, the proposed design could still lead to



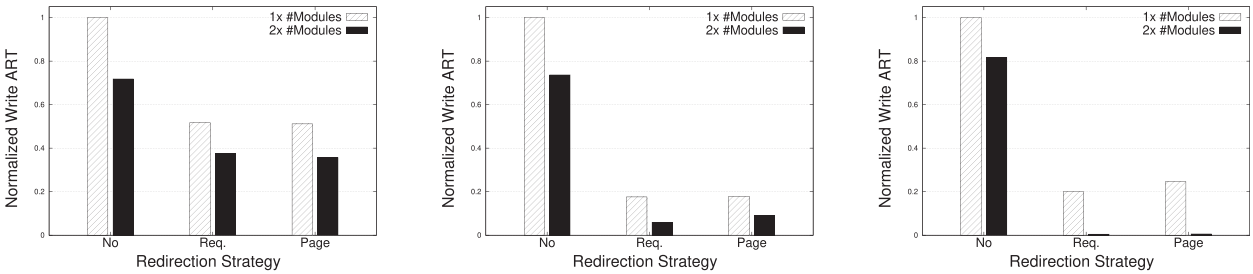
(a) `rsrch` workload (b) `usr` workload (c) `src` workload

Fig. 6. Normalized write average response time.



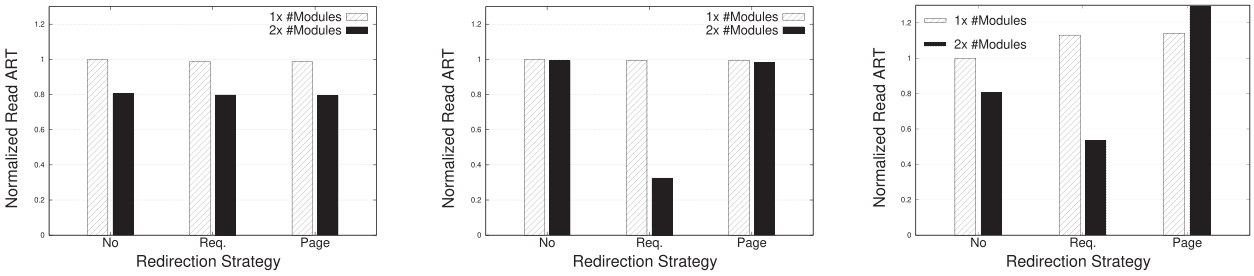
(a) `rsrch` workload (b) `usr` workload (c) `src` workload

Fig. 7. Normalized read average response time.



(a) rsrch (b) usr (c) src

Fig. 8. Write performance under double number of flash modules (16x workload intensity).



(a) rsrch (b) usr (c) src

Fig. 9. Read performance under double number of flash modules (16x workload intensity).

quite stable write performance, even when the workload intensity was increased by 16 times. Such results reveal that the proposed Redirection design has good performance scalability in terms of the workload intensity, since the proposed policy can always make good use of the abundant internal resources to handle even more intensive workloads.

On the other hand, Fig. 7 shows that the proposed design could lead to comparable read performance as No Redirection scheme under the evaluated workloads. The reasons for this phenomenon are twofold. First, when the proposed policy tried to retrieve the redirected data, the master controller would efficiently communicate with slave modules to locate the requested data without exhaustedly searching all flash modules. Second, the proposed design would properly recall the redirected requests so as to minimize the overall redirection traffics. As a result, the read performance would not be seriously degraded due to the request redirection. Specifically, under rsrch and usr workloads, the proposed Redirection policy could even improve the read average response time of No Redirection scheme by around 1 to 3 percent in most cases. In addition, under the write-intensive src workload, the proposed Redirection policy could even have potentials to improve the read performance of No Redirection scheme by up to 30 percent when the workload intensity is less than 8x; while when the workload intensity was 16x, the proposed Redirection policy could result in about 10 percent slower read performance than No Redirection scheme. This is because the proposed policy aims at mitigating the run-time heavy write traffic by dynamically redirecting write requests among modules, even though this design could potentially introduce some delays to the read requests.

It is also interesting to see that there were only limited performance differences (no matter read or write performance), when the proposed redirection policy was performed based on different units. Specifically, form Figs. 6 and 7, the performance results between performing data redirection in the unit of requests or flash pages would only have no more than 2 percent differences, and neither the two evaluated units could always lead to the best performance results.

4.2.2 Doubled Number of Flash Modules

To further validate the performance scalability of the proposed Redirection policy, Figs. 8 and 9 show the performance results with and without the proposed design under different numbers of flash modules. Notably, in order to provide a fair comparison, we doubled the number of flash modules for the simulated SSD, but reduced the storage capacity of each module to the half of the storage capacity of the originally simulated modules. In the meantime, the terms 1x # Modules and 2x # Modules are used to indicate whether the number of flash modules was doubled or not (i.e., 2x # Modules denotes a doubled number of flash modules were used). In other words, the total storage capacity of the 1x # Modules and 2x # Modules configurations are just the same. Moreover, in both figures, the x-axis denotes different redirection strategies (i.e., No Redirection, Request-based Redirection, and Page-based Redirection), and the y-axis denotes the average response time (ART) normalized to the results of No Redirection. In addition, in both figures, the workload intensity was set as 16x without losing of generality.

Fig. 8 clearly shows that, with a doubled number of flash modules, the write performance could be improved under all evaluated workloads, no matter whether the proposed policy is applied or not. This is reasonable because a double number of flash modules provides a twofold number of control units and a double size of on-device buffer area to handle I/O requests more efficiently. However, if we look closer, the No Redirection scheme with 2x # Modules could only

improve the write performance by at most 29 percent than that of No Redirection scheme with 1x # Modules; however, the proposed Redirection policy could significantly improve the write performance by at least 67 percent when the number of flash modules was doubled. This result directly demonstrates that, even though the doubled number of flash modules were used, the write performance could not be easily scaled up by merely based on the pre-defined request direction policy.

On the other hand, as shown in Fig. 9, the proposed Redirection policy could not only ensure comparable read performance as No Redirection scheme when the number of flash modules was doubled, but also have the potentials to greatly improve the read performance by at least 47 percent than that of No Redirection scheme with 1x # Modules, under usr and src workloads. The rationale behind this is that, as shown in Fig. 8, the proposed Redirection policy could greatly improve the write performance, especially under usr and src workloads. That is, our design could avoid jamming any certain module with lots of read and write requests, so that the time to complete write and read requests could be both improved. In summary, as the scale of the SSD keeps increasing, the proposed design can not only achieve better write performance, but also have potentials to improve the read performance.

4.2.3 Load/Store Overhead of Address Translation and Redirection Information

In our implementations, the controllers of flash modules adopted the DFTL design and thereby needed to on-demand load/store both *address translation information* and *redirection information* between the DRAM space and flash memory. That is, in practice, the proposed design might have some performance overheads on loading/storing both types of information between the on-device DRAM space and the flash memory, when the DRAM space is not large enough to completely contain all of the required address translation and redirection information.

In particular, our implementation of DFTL follows the suggested settings in [6] to have 1024 cached mapping entries and manage the cached entries by using the segmented LRU policy. That is, both address translation information and redirection information will compete for the 1024 cached mapping entries. To explicitly show the performance delay caused by the load/store of both types of information, we compare the performance results, in terms of average response time (ART), when 1024 and an unlimited number of cached mapping entries are provided, respectively. Notably, there will be *no* performance overheads on loading/storing the information when an unlimited number of cached mapping entries is provided. Moreover, the evaluated workloads were replayed as its original workload intensity without losing of generality. According to our evaluations, the performance degradation caused by the load/store of both types of information was limited. Specifically, the proposed design would introduce nearly *no* performance degradation under the rsrch server workloads; additionally, the proposed design would introduce roughly 1.8 and 7.1 percent performance delay on loading/storing both types of information under the write-intensive usr and src server workloads, respectively.

5 Conclusion

Performance-intensive applications have led the interface and architecture changes of high-end SSDs to efficiently manage the growing scale of flash memory space. To improve the performance of the multi-module SSDs, this paper proposes a *request flow coordination* flow to dynamically redirect I/O requests among flash modules to avoid overloading any flash module. Consequently, the SSD performance could be effectively improved when the number of modules of the device is increased. Our evaluation results show that the proposed design could achieve significant write performance improvement with nearly no read performance degradation and very limited coordination overhead. Specifically, under the evaluated I/O traces with 16 times of workload intensity, the proposed design could significantly improve the write performance by at least 67 percent when the number of flash modules was doubled, while only about extra 1.5 percent performance delay was introduced on performing request redirections. In our future work, we shall further investigate how to redirect I/O requests to not only benefit the write performance but also to improve the read performance with limited runtime data migration overhead.

Acknowledgments

This work was supported in part by The Research Grants Council of Hong Kong SAR (Project No. CUHK24209618) and Tencent Technology.

References

[1] A. Ban, "Flash file system," US Patent 5404485A, Apr. 1995.
[2] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo, "Endurance enhancement of flash-memory storage systems: An efficient static wear leveling design," in *Proc. 44th Annu. Des. Autom. Conf.*, 2007, pp. 212–217.
[3] F. Chen, B. Hou, and R. Lee, "Internal parallelism of flash memory-based solid-state drives," *Trans. Storage*, vol. 12, no. 3, pp. 13:1–13:39, May 2016.
[4] F. Chen, T. Luo, and X. Zhang, "CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives," in *Proc. 9th USENIX Conf. File Storage Technol.*, 2011, pp. 77–90.
[5] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart SSDs: Opportunities and challenges," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 1221–1230.
[6] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. 14th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2009, pp. 229–240.
[7] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *Proc. Israeli Exp. Syst. Conf.*, 2009, pp. 10:1–10:9.
[8] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren, "Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance," *IEEE Trans. Comput.*, vol. 62, no. 6, pp. 1141–1155, Jun. 2013.
[9] Intel Corporation and Seagate Technology, "Serial ATA Native Command Queuing: An Exciting New Performance Feature for Serial ATA," Jul. 2013.
[10] M. Jung, "Exploring design challenges in getting solid state drives closer to CPU," *IEEE Trans. Comput.*, vol. 65, no. 4, pp. 1103–1115, Apr. 2016.
[11] M. Jung and M. T. Kandemir, "Sprinkler: Maximizing resource utilization in many-chip solid state disks," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit.*, 2014, pp. 524–535.
[12] M. Jung and M. Kandemir, "Revisiting widely held SSD expectations and rethinking system-level implications," in *Proc. ACM SIGMETRICS/Int. Conf. Meas. Modeling Comput. Syst.*, 2013, pp. 203–216.

[13] M. Jung, R. Prabhakar, and M. T. Kandemir, "Taking garbage collection overheads off the critical path in SSDs," in *Proc. 13th Int. Middleware Conf.*, 2012, pp. 164–186.
[14] M. Jung, E. H. Wilson, and M. Kandemir, "Physically addressed queueing (PAQ): Improving parallelism in solid state disks," in *Proc. 39th Annu. Int. Symp. Comput. Archit.*, 2012, pp. 404–415.
[15] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Trans. Embedded Comput. Syst.*, vol. 6, no. 3, Jul. 2007.
[16] J. Liang, Y. Xu, D. Sun, and S. Wu, "Improving read performance of SSDs via balanced redirected read," in *Proc. IEEE Int. Conf. Netw. Archit. Storage*, 2016, pp. 1–10.
[17] K. T. Malladi, M. Awasthi, and H. Zheng, "FlexDrive: A framework to explore NVMe storage solutions," in *Proc. IEEE 18th Int. Conf. High Perform. Comput. Commun.; IEEE 14th Int. Conf. Smart City; IEEE 2nd Int. Conf. Data Sci. Syst.*, 2016, pp. 1115–1122.
[18] M. Murugan and D. H. C. Du, "Rejuvenator: A static wear leveling algorithm for NAND flash memory with minimized overhead," in *Proc. IEEE 27th Symp. Mass Storage Syst. Technol.*, 2011, pp. 1–12.
[19] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *Trans. Storage*, vol. 4, no. 3, pp. 10:1–10:23, Nov. 2008.
[20] PCI-SIG, "PCI Express Base Specification Revision 4.0," 2016.
[21] Z. Qin, Y. Wang, D. Liu, Z. Shao, and Y. Guan, "MNFTl: An efficient flash translation layer for MLC NAND flash memory storage systems," in *Proc. 48th Des. Autom. Conf.*, 2011, pp. 17–22.
[22] SATA-IO, "Serial ATA Revision 3.2," 2015.
[23] P. Scheuermann, G. Weikum, and P. Zabback, "Data partitioning and load balancing in parallel disk systems," *VLDB J.*, vol. 7, no. 1, pp. 48–66, Feb. 1998.
[24] S. Kung, "Native PCIe SSD Controllers: A Next-Generation Enterprise Architecture for Scalable I/O performance," Jan. 2012.
[25] J.-Y. Shin, Z.-L. Xia, N.-Y. Xu, R. Gao, X.-F. Cai, S. Maeng, and F.-H. Hsu, "FTL design exploration in reconfigurable high-performance SSD for server applications," in *Proc. 23rd Int. Conf. Supercomputing*, 2009, pp. 338–349.
[26] C. Tsao, Y. Chang, M. Yang, and P. Huang, "Efficient victim block selection for flash storage devices," *IEEE Trans. Comput.*, vol. 64, no. 12, pp. 3444–3460, Dec. 2015.
[27] C. Wang, Q. Wei, M. Xue, J. Yang, and C. Chen, "Data-centric garbage collection for NAND flash devices," in *Proc. IEEE Non-Volatile Memory Syst. Appl. Symp.*, 2015, pp. 1–6.
[28] S. Wu, H. Jiang, D. Feng, L. Tian, and B. Mao, "Improving availability of raid-structured storage systems by workload outsourcing," *IEEE Trans. Comput.*, vol. 60, no. 1, pp. 64–79, Jan. 2011.
[29] S. Wu, H. Jiang, L. Tian, and B. Mao, "Workout: I/O workload outsourcing for boosting RAID reconstruction performance," in *Proc. 7th USENIX Conf. File Storage Technol.*, 2009, pp. 239–252.
[30] W. Xie, Y. Chen, and P. C. Roth, "Parallel-DFTL: A flash translation layer that exploits internal parallelism in solid state drives," in *Proc. IEEE Int. Conf. Netw. Archit. Storage*, 2016, pp. 1–10.
[31] R. Yamashita *et al.*, "11.1 a 512Gb 3b/cell flash memory on 64-word-line-layer BiCS technology," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2017, pp. 196–197.

Ming-Chang Yang (Member, IEEE) received the BS degree from the Department of Computer Science, National Chiao-Tung University, Hsinchu, Taiwan, in 2010, and the master's and PhD degrees under the supervision of professor Tei-Wei Kuo from Graduate Institute of Networking and Multimedia, National Taiwan University, Taipei, Taiwan, in 2012 and 2016, respectively. Currently he is an assistant professor with the Department of Computer Science and Engineering, The Chinese University of Hong Kong. His primary research interests including emerging non-volatile memory and storage technologies, memory and storage systems, and the next-generation memory/storage architecture designs.

Yuan-Hao Chang (Senior Member, IEEE) received the PhD degree from the Department of Networking and Multimedia of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, in 2009. Currently he is an assistant research fellow with the Institute of Information Science, Academia Sinica, Taipei, Taiwan, since August 2011. From February 2010 to July 2011, he was an assistant professor with the Department of Electronic Engineering, National Taipei University of Technology, Taipei, Taiwan. His research interests include storage systems, embedded systems, and computer system architecture.

Tei-Wei Kuo (Fellow, IEEE) received the BS and PhD degrees in computer science from the National Taiwan University and the University of Texas at Austin in 1986 and 1994, respectively. Currently he is the dean of College of Engineering, City University of Hong Kong, Hong Kong, and a distinguished professor of the Department of Computer Science and Information Engineering, National Taiwan University, where he served as the department chairman from August 2005 to July 2008. He served as a distinguished research fellow and the director of the Research Center for Information Technology Innovation, Academia Sinica, between January 20, 2015, and July 31, 2016. His expertise is embedded systems, non-volatile memory system and software, and real-time systems. He is an ACM Fellow.

Chun-Feng Wu (Student Member, IEEE) received the BS degree from the Department of Computer Science and Information Engineering, National Central University, in 2014, and the MS degree from the Department of Computer Secience, National Tsing-Hua University, in 2016. He is currently working toward the PhD degree in the Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan. Meanwhile, he serves in R&D alternative service at Institute of Information Science, Academia Sinica, Taipei, Taiwan. His primary research interests include memory/storage systems, embedded systems, operating systems and the next-generation memory/storage architecture designs.

Hierarchical Orchestration of Disaggregated Memory

Wenqi Cao and Ling Liu, *Fellow, IEEE*

Abstract—This article presents *XMemPod*, a hierarchical disaggregated memory orchestration system. *XMemPod* virtualizes cluster wide memory to scale large memory workloads in virtualized clouds. It makes three novel contributions: (1) *XMemPod* offers efficient, transparent, and dynamic sharing of available memory that is disaggregated across VMs on the same host or in the cluster. (2) *XMemPod* provides a hierarchical memory expansion framework, which enables memory-intensive workloads on a VM to expand its memory demand over virtualized host memory first, and remote memory next, before resorting to external disk. (3) *XMemPod* provides a suite of optimization techniques to further improve the utilization and access latency of disaggregated memory. *XMemPod* is deployed on a virtualized RDMA cluster without any modifications to user applications and the OSes. Evaluated with multiple workloads on unmodified Spark, Apache Hadoop, Memcached, Redis and VoltDB, using *XMemPod*, throughputs of these applications improve by 11x to 612x over conventional Linux, and by 1.7x to 14x over the existing representative remote memory paging systems, and yet the total amount of network traffic consumed by *XMemPod* is only 24 percent of the existing approaches.

Index Terms—Memory disaggregation, virtualization, operating system

1 Introduction

Big data and latency-demanding applications [7], [8], [9], [53], [58] are typically deployed using the application deployment models, comprised of virtual machines (VMs), containers, and/or executors/JVMs. These applications enjoy high throughput and low latency if they are served entirely from memory. However, actual estimation and memory allocation are difficult. When these applications cannot fit their working sets in real memory of their VMs/containers/executors, they suffer large performance loss due to excess page faults and thrashing. Even when available (unused) memory is present in other VMs/containers/executors on the same host or a remote node, these applications are unable to share those unused host/remote memory (Section 2).

Existing research studied the above problems from two orthogonal dimensions: (i) estimating working set size for accurate resource allocation and (ii) increasing effective resource capacity for executors. Accurate memory allocation is hard as peak memory variations happen under different application types, workload inputs, data characteristics and traffic patterns. Applications often overestimate their requirements or attempt to allocate for peak usage [17], [27], resulting in severely unbalanced memory usage across executors, underutilization on the host and across the cluster [48]. Instead, proposals for increasing effective memory capacity promote the allocation of global memory resource shared by all machines (VMs/containers) to increase their effective memory capacities. These proposals promote new architectures and new hardware design for memory disaggregation [10], [22], or new programming models [42], [47]. But they lack of desired transparency at OS, network stack, or application level, hindering their practical applicability. Recent efforts represented by Accelio nbdX [1], including Infiniswap [24] built on top of nbdX, exploit RDMA networks for remote memory paging with transparency. However, they solely rely on RDMA pre-registered memory allocation for remote memory sharing and cannot extend RDMA for sharing available physical memory on the same node across VMs or containers.

This paper describes *XMemPod*, a hierarchical disaggregated memory orchestration system, which accelerates big-data and machine learning (ML) workloads by virtualizing two types of cluster-wide external memory: (1) The available memory from other VMs on the same node due to inter-VM memory usage imbalance, which provides memory-speed access; and (2) The available remote memory in the cluster through RDMA registered memory allocation, which provides network-speed access. *XMemPod* is designed to dynamically orchestrate terabytes of cluster wide memory into multiple memory pods of these two types. By leveraging guest indirection, *XMemPod* transparently virtualizes cluster-wide available memory into the physical address space of each guest OS instance on demand (Section 3).

XMemPod is inspired by existing proposals from two orthogonal efforts (Section 6): exploiting non-intrusive sharing of memory between host and its guest VMs [30], [56], [57] and exploiting the disk-network latency gap via available remote memory [29], [60]. However, unlike existing proposals for dynamic memory balancing, *XMemPod* does not pay any CPU overhead and time delay for memory scanning to detect memory imbalance and does not rely on

• *The authors are with the School of Computer Science, Georgia Institute of Technology, Atlanta, GA 30332. E-mail: {wcao39, lingliu}@cc.gatech.edu.*

Manuscript received 26 Apr. 2019; revised 8 Jan. 2020; accepted 15 Jan. 2020. Date of publication 21 Jan. 2020; date of current version 8 May 2020.
(Corresponding author: Wenqi Cao.)
Recommended for acceptance by J. Shalf.
Digital Object Identifier no. 10.1109/TC.2020.2968525

estimation of the working set size of each VM/container/executor to respond to memory pressure, since accurate estimation of working set size is difficult under changing workloads [11], [16]. Moreover, existing solutions for remote memory paging [1], [24] solely rely on RDMA memory registration for network speed memory sharing with remote node (s), and cannot be used to leverage available memory in other VMs on the same node for memory speed sharing across VMs on the same host.

XMemPod addresses the above challenges by making the two types of disaggregated memory pods available to unmodified VMs for use via a low-latency in-memory block device. These memory pods serve transient overflow data or memory paging events for the VM that requires larger physical memory than its allocation. *XMemPod* does so without any modifications to its applications or guest OS.

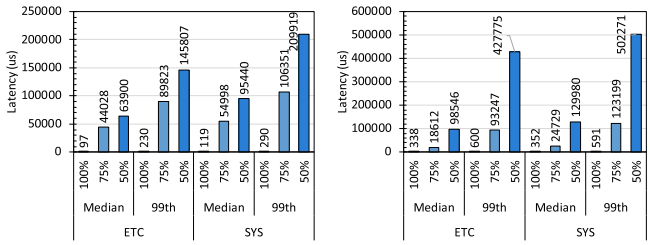
XMemPod meets the memory demand of a VM by prioritizing the use of available memory from other VMs on the same host via the memory-speed I/O device. Upon insufficient available memory on a local host, *XMemPod* resorts to remote memory pods via the RDMA port as the next tier network-speed I/O device. To scale the management cost of large size clusters, XMemPod reliably partitions the cluster-wide external memory into non-intrusive, elastic remote memory sharing groups through decentralized coordination.

XMemPod is implemented on Linux kernel 4.1.0 and KVM [35], deployed on a 56 Gbps, 32-machine RDMA cluster in Emulab [5] with 2 terabyte (TB) collective memory. We evaluated it using multiple unmodified memory intensive applications: Spark [58], [59], Apache Hadoop [53], Memcached [7], Redis [8], VoltDB [9], and open benchmarks: HiBench [31], SparkBench [37] and YCSB [20]. Using *XMemPod*, throughputs of these applications improve by 11x to 612x, median and tail latencies improve by 174x to 702x and 218x to 630x respectively, over conventional OS swap facility. Compared with the existing approaches such as nbdX or Infiniswap, using *XMemPod*, throughputs improve between 1.7x and 14x, median latencies and tail latencies improve between 1.8x and 12x, and 3.3x and 15x respectively (Section 5).

There are situations that *XMemPod* does not yet handle effectively, including the simple first-come-first-serve policy for using the host coordinated shared memory on each node, which could be improved by using a more sophisticated policy, such as threshold-caped. Also we were unable to obtain larger RDMA clusters to perform stress test on scalability.

2 Motivation

The Effect of Transient Memory Usage Variation. Memory utilization imbalance and temporal usage variations are frequently observed in virtualized clouds [17], [23], and production datacenters [24], [53]. One study on a google 12K-machine cluster running a mixture of long and short-lived workloads reported around 50 percent memory utilization, stating "the gap between resource requests and average usage accounted for most of this difference" [48], [49]. Another study on two production datacenters (3,000-machine Facebook analytic cluster and 12,500-machine google cluster) reports severe imbalance in memory utilization for more than 70 percent of the time across machines [24].



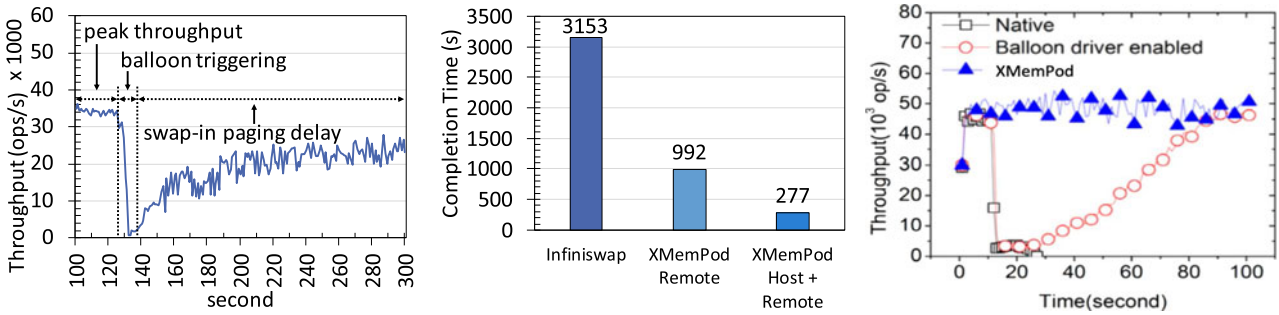
(a) Redis (b) VoltDB

Fig. 1. The effect of transient memory usage variation.

To gain an in-depth understanding of the potential opportunities of exploiting available memory of other VMs on the same node or in the cluster for performance speed-up of VMs under transient memory pressure, we run our micro benchmark on two bigdata applications: Redis [8], an in-memory NoSQL store, and VoltDB [9], a translytical in-memory database. We measure per server performance for each application with two benchmark workloads: ETC and SYS (for detailed setup see Section 5), using the raw dataset of 20 GB from YCSB [20] per server.

The peak memory that can fit the full working set is measured for both Redis and VoltDB, which are 29 GB and 30 GB respectively. We run each application on VM with three different memory capacity configurations – 100, 75, and 50. x% indicates that the VM is configured to run an application and can hold x% of its working set in memory. When $x = 100\%$, no paging occurs, but when $x < 100\%$, paging happens, as shown in Fig. 1. We highlight two observations. *First*, compared to 100 percent configuration, when 25 percent of the working set does not fit in memory (75 percent configuration), the median latency is worsened by 462x and 70x for Redia and VoltDB respectively. For 50 percent configuration (another 25 percent reduction), median latencies are worsened by 802x and 369x for Redis and VoltDB respectively. *Second*, For 75 percent configuration, 99th percentile latencies of Redis and VoltDB are worsened by 391x and 208x respectively, compared to 100 percent configuration. For 50 percent configuration, 99th percentile latencies are degraded by 724x and 850x for Redis and VoltDB respectively.

Dynamic Memory Balancing With the Balloon Driver. To cope with such harsh performance degradation when the working sets of applications do not fit in memory, memory ballooning [56] was proposed to move unused memory between VMs. However, solely relying on ballooning, applications under memory pressure still suffer hefty performance degradation due to three types of delays: the timing delay of scheduling ballooning, the balloon driver delay for moving sufficient memory, and the time delay for applications to return to their peak performance. Fig. 2a shows the three types of delays, the performance deterioration incurred, and the adverse effect on application performance, when the working set of Redis (ETC workload) does not fit in memory (50 percent configuration). We observed that the throughput of Redis deteriorates sharply during the 15 seconds from 123rd to 138th second, due to the excessive paging-out events, the timing delay of scheduling ballooning, and the balloon driver delay for moving sufficient memory. Upon installing sufficient ballooning memory at the 140th second, Redis throughput starts to recover but at a very slow pace. This is



(a) Ballooning Delays (b) Comparison of Solutions (c) XMemPod

Fig. 2. Alternative solution approaches.

because even with additional memory, Redis cannot immediately regain its peak performance due to the slow per page swap-in operation upon each page fault. It takes approximately 140 seconds to fully utilize the newly ballooned memory. Furthermore, ballooning incurs the overheads of scheduling and moving (deflating and inflating) memory from one VM to another on the same host, and cannot leverage the available remote memory in the cluster.

Differentiating Local versus Remote External Memory. In virtualized clouds, multiple VMs run on the same host machine, each having its own memory allocation (typically equal amount). When a VM reaches its memory limit, it has two alternative low-latency memory expansion opportunities before paging to local disk swap partition. They are available (unused) memory on local host and available (unused) remote memory in the cluster. Note that even though SSD is much faster than HDD, it is still two orders of magnitude slower than fast interconnect like InfiniBand based RDMA [54]. At the same time, RDMA interconnect is several times slower than memory-speed I/O on local node [45], making latency-sensitive prioritization critical for efficient disaggregated memory orchestration.

XMemPod implements a virtual block device manager, which provides a hierarchical disaggregated memory orchestration service to each VM by leveraging indirection, and interfacing with the host shared memory first, the remote memory next, and resorting only to the disk I/O for exception handling in the presence of insufficient remote memory. Fig. 2b shows the comparison of three alternative design choices by running Redis ETC workload under 50 percent configuration: (a) using existing solutions (e.g., Infiniswap) for remote memory sharing via RDMA, (b) *XMemPod* remote, which optimizes the remote RDMA sharing through bandwidth-aware batch swap-in and swap-out, and (c) hierarchical *XMemPod*, which leverages available memory on the host first and then available remote memory in the cluster, in addition to RDMA bandwidth optimizations (Section 4). Using hierarchical *XMemPod* in case (c), Redis completion time is 11.3x times faster than existing remote memory solutions in case (a), and 3.2x faster than *XMemPod* RDMA in case (b).

Fig. 2c shows the performance comparison with XMemPod turned on. We observe that with hierarchical orchestration of disaggregated memory, XMemPod can significantly improve the runtime performance of Redis when the working set of its data cannot fit fully into the main memory allocated at the configuration time. We would like to note that the comparison with existing systems is fair for two reasons. First, it is fair if the available host memory for XMemPod can be directly distributed to the VMs on the same host. Second, all solutions have the same amount of host memory. The main difference is that the conventional approach to VM management is to equally allocate the available host memory to all VMs on the host. In contrast, XMemPod takes back some proportion of the allocated VM memory to create a pool of host coordinated shared memory, say 1/10 or 1/3 of the 30 GB allocated memory to each VM on a host. XMemPod utilizes this pool of shared memory to compensate those VMs that are under transient memory pressure through providing host and remote memory paging service.

3 *XMemPod* Overview

XMemPod consists of four core components: the disaggregated memory paging service manager (PSM), the shared memory manager (SSM), the remote memory manager (RMM) and the disk swap partition manager (DSM), as shown in Fig. 3. They coordinate closely to provide efficient virtualization of cluster wide memory and hierarchical orchestration of disaggregated memory sharing capabilities. Client module and server module are the two main functional components of *XMemPod*. A VM that is low on memory will run the *XMemPod* client module to acquire external disaggregated memory service. Every node in the cluster can run the *XMemPod* server

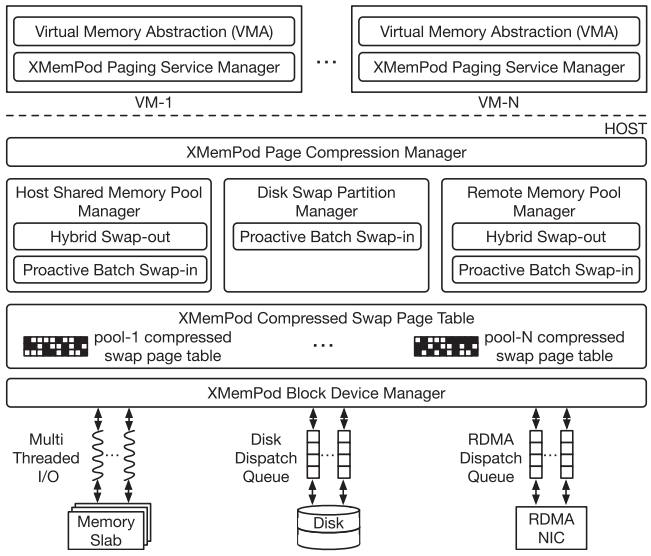


Fig. 3. *XMemPod* System architecture.

module to provide external memory expansion and sharing in response to the requests from *XMemPod* clients in a cluster. *XMemPod* is currently implemented on a VM based applications deployment platform, which is common in production datacenters with virtualized clouds [17], [49]. *XMemPod* implements remote memory paging using an Infiniband RDMA network, though it makes no assumption on specific RDMA network technology.

The Memory Paging Service Manager is the client module implemented on top of the *kswapd*, a default kernel daemon, responsible for memory paging. The PSM intercepts swap-out and swap-in operations and redirects them to *XMemPod*. The PSM provides a virtualized block device interface to the applications on a guest VM. This virtualized block device can be configured as a low latency primary swap device. Extensions to this virtualized block device is to configure it as a low latency volatile file system for storing large datasets, or a memory-mapped I/O space for large memory application. For the guest VM, the block device is simply a fast I/O partition (e.g., a linear I/O space) with an order of magnitude smaller access latency than disk. *XMemPod* implements the host-guest shared memory swap partition with both ramdisk based option and shared memory based option. The former is mounting a host resident ramdisk to the guest VM, and the latter uses a shared memory pool provided by the host, which is mapped into the host swap partition address space on guest VM. Comparing to the ramdisk solution, the shared memory approach delivers 1.7x better performance via a loadable kernel module for Linux 4.1.0 for all workloads we have tested. By default, *XMemPod* uses the shared memory solution.

The Shared Memory Manager (SMM) uses a POSIX shared memory region as the basis for sharing memory between host and VMs. On host, *XMemPod* is implemented and runs as a user space program, which manages host and remote shared memory block devices and disk block device, each device exposes a conventional block device I/O interface to the guest OS's VMA manager, which treats the block device as a fixed size swap partition. No modification is required to the host OS or any of its existing kernel modules, including KVM. The entire host shared memory area is divided into multiple elastic memory *pods*, one per VM on the host, provided that a shared memory pod is only established for a guest when paging initially occurs, and the shared memory pod is revoked from the guest when the execution of its applications terminates, or when the host shared memory manager terminates its service to the guest VM. Upon creating a shared memory pod, its entire address space is logically partitioned into slabs of fix size, according to `Host-SlabSize` in initial configuration. The shared memory pod manager maintains the mapping between the page offset in the guest swap partition and the address in its corresponding shared memory swap partition, and it opportunistically adjusts the size of each shared memory pod to decrement or increment one slab, based on the lower and upper pod utilization thresholds, defined by `sm-pod-min` and `sm-pod-max`. There are three types of pages in the host shared memory: *active* pages, those being used, *free* pages, those allocated to the pod but not yet being used, and *idle* pages, those that do not belong to any pod.

The Remote Memory Manager is designed with similar principles. First, the setting of `Remote-SlabSize`, `Remote-pod-min`, and `Remote-pod-max` can be different from those set for host shared memory configuration. Second, slabs are the units for balancing memory utilization across remote machines and providing low latency mapping to remote memory. All pages in a slab are mapped to the same remote memory. However, different slabs in the same remote pod can be mapped to multiple remote servers' memory. Third, we implement remote memory management on top of nbdX [1], which exploits the advantages of multi-queue implementation in the block layer and the Accelio acceleration facilities to provide fast IO to a remote shared memory device. When a paging request is redirected from a guest VM to the remote memory manager, it first selects one primary remote server and one secondary remote server from the `remote-count` candidates maintained by *XMemPod*, and uses the corresponding remote memory pods to expose the I/O interface of remote shared memory and use VirtIO to the guest (VM1) for read and write requests. For each remote server, a remote pod connector is setup to keep the RDMA connection with the remote server. On a remote node, when its NIC receives paging requests, it accesses the pages by mapping the target offset and size into the corresponding remote RDMA registered memory region. To guarantee the data to be delivered without corruption, Reliable Connected Queue Pair (RC QP) is configured on both sides. For swap-in request (read), the SMM passes to RMM the metadata from the compressed disaggregated memory page table (CSPT), including compressed page size, location of the compressed page in the remote memory pod(s), the instruction on proactive swap-in (batch read), RMM checks metadata parameters and performs swap-in operations. RMM also broadcasts periodic resource utilization messages which the client modules use to discover the available remote memory servers, such as their memory availability and load as well as page transfers to its clients. When a sever reaches its remote memory capacity `Remote-pod-max`, it will decline to serve any new swap-out (write) requests from its clients.

The Disk Swap Partition Manager is called to serve a swap-out request if there is insufficient remote memory in the cluster or if time-out occurs from an existing swap-out operation. DSM will resort to conventional OS swap facility to perform the swap. To ensure the availability of swap-out pages, *XMemPod* ensures the triple modular redundancy by maintaining three copies of each swap-out page in three different remote nodes. Although in the first implementation of the XMemPod, a swap-out event is committed only when at least three remote servers have committed the swap-out event. We do maintain triple modular redundancy for each swap page with the eventual consistency guarantee.

Connection Management. *XMemPod* uses one-sided RDMA READ/WRITE operations for data plane activities and RDMA SEND/RECEIVE operations for control plane activities. For each individual connection, two channels are established: (i) RDMA channel is used for maintaining the network connection, data transfer, and data corruption. (ii) *XMemPod* channel maintains status of remote MemPod by interacting with remote agent and uses it for placement and eviction algorithms.

Remote Memory Balancing. Several remote memory selection algorithms are implemented to minimize memory imbalance across machines and used as configuration

parameters, including round robin (RR), weighted RR, random, or power of two choices [24], [50], with RR as the default. *XMemPod* proactively allocates memory slabs of size `Remote-SlabSize` and registers them as memory regions for RDMA operations on remote servers, which reduces the cost of initialization process. Remote idle memory is monitored and when it drops below the `Remote-low` threshold, remote memory slabs will be deregistered preemptively through our remote slab eviction handler based on `Remote-SlabSize`, and updates the respective compressed swap page table (CSPT) corresponding to the deregistered slabs (see next section for detail on CSPT). At the same time, new remote memory servers will be selected to host the evicted pages for maintaining the dual replica of swap-out pages.

Fault Tolerance. *XMemPod* runs its client module of the disaggregated memory paging service on a guest VM, and its server modules, such as SMM, RMM, DSM on the host in every node of the cluster. If the corresponding guest VM or the host machine fails, or the local disk fails during paging, such as server/VM crash, *XMemPod* provides the same failure semantics as the guest OS swap facility today. For remote memory paging, it is important to handle unexpected failure scenarios due to network connection (link) failure or remote server failures. First, *XMemPod* does not require central coordination for remote memory paging, thus, single point of failure and frequent message synchronization are avoided. Second, each remote swap-out operation is replicated to at least two remote nodes and also maintained in the local disk swap of the guest VM. Third, each remote paging operation is treated as an atomic transaction, all or nothing, which is recorded in the corresponding entry of the CSPT, thus removing the inconsistency due to remote connection failure or remote server failure. Although in the first prototype, only dual remote servers are used, to support stronger fault tolerance, *XMemPod* may support paging to more than two remote servers and maintains the metadata in the corresponding CSPT for each swap-out request. This not only further increases the fault tolerance in the presence of network connection failure and unreachable server induced failure, but may also simplify the remote slab eviction handling.

Consistency. For each VM, we maintain a CSPT, which works as a log table to track of where a swap-out page is. We set the *CSPT_entry* using a location flag. When it is set to `LOCAL`, the page is stored in the host-VM shared memory pod, when it is set to `REMOTE`, the `REMOTE` tag will provide two location entries: the primary remote MemPod and the secondary remote MemPod. Otherwise, the local disk partition is used, indicating insufficient remote memory in the cluster or failure to reach remote servers before time-out. For swap-out request, *XMemPod* synchronously writes data into RDMA channel of primary remote MemPod and secondary remote MemPod. Once the RDMA WRITE operation of two remote MemPod completes, and its corresponding *CSPT_entry* is updated, the swap-out operation is committed. To avoid data corruption during transferring process, RC QP is configured for each RDMA channel, which guarantees that messages are delivered from a requester to a responder at most once and in order without correction. Requester considers a message operation complete once there is an ack from the responder that the message was read/written to its memory. Responder considers a message operation complete once the message was read/written to its memory. For swap-in request, if the *CSPT_entry* is set to `REMOTE`, *XMemPod* will put one RDMA READ operation on the primary MemPod address into the RDMA dispatch queue. When the READ completes, *XMemPod* responds by marking the commit of the read operation. Otherwise, *XMemPod* reads it from the secondary MemPod, or the local disk partition. We consider two failure cases.

1) *Local VM Failure/Exit*. When the PSM is unreachable, *XMemPod* considers the scenario as local VM failure or exit. Upon detecting this failure, the garbage collection is triggered at both local host-VM shared MemPod and two remote MemPods. It marks all slabs for this VM as unmapped and removes corresponding CSPT. The garbage collection for local disk is performed and cleared by guest OS.

2) *Host Failure*. For each server module of *XMemPod*, its RMM periodically checks the reachability of each remote MemPod agent in its remote server group, and considers unreachability of a remote agent as the remote host failure scenario. Upon detecting this failure, *XMemPod* updates the corresponding *CSPT_entry*(s) as unavailable. For a failed host, the remote agent unmaps and deregisters all slabs that are used as remote MemPod slabs for page requests read from and write to this failed host. The broken RDMA and *XMemPod* channel will be removed from the agent as well.

Scalability. A fundamental challenge for providing cluster-wide memory virtualization is to scale the system to terabytes of collective memory in a cluster. However, to track where each swap-out page is located in the cluster, the server module corresponding to the VM needs to maintain the metadata such as server ID, MemPod ID and offset in the CSPT entry of this page in either the host-VM shared memory or in the RDMA registered memory region for remote memory sharing. Consider the scenario that a simple in-memory hash table is used to store the location of each of its pages in the cluster, and each page is 4 KB in size. If we use 8 bytes to store each location identifier metadata, then we would need up to 5 GB host-VM shared memory to store the hash table for the 2 terabytes of cluster-wide memory. For 10 TB, it is 25 GB. Maintaining a CSPT of such size for each VM will incur prohibitively high cost as the cluster-wide memory scales up. To address this scalability challenge, *XMemPod* adopts group sharing model, where servers in a cluster are partitioned into groups of similar sizes. Servers within a group can share remote memory with one another. Each server can access the up-to-date memory sharing state of all other members of its group via its group leader. A leader election protocol [32] periodically elects the one with maximum available memory as the leader of a group. If the leader node crashes (handshake time-out), a new leader election process will be triggered. Also, leaders of all remote memory sharing groups form the top tier grouping service, which supports dynamic re-grouping upon request from their member client(s).

XMemPod is deployed successfully on small size RDMA clusters (32 machines) with up to 8 VMs each, a total of up to 256 VMs. As the number of VMs increases on a host, there is small overhead involved in dynamic allocation/

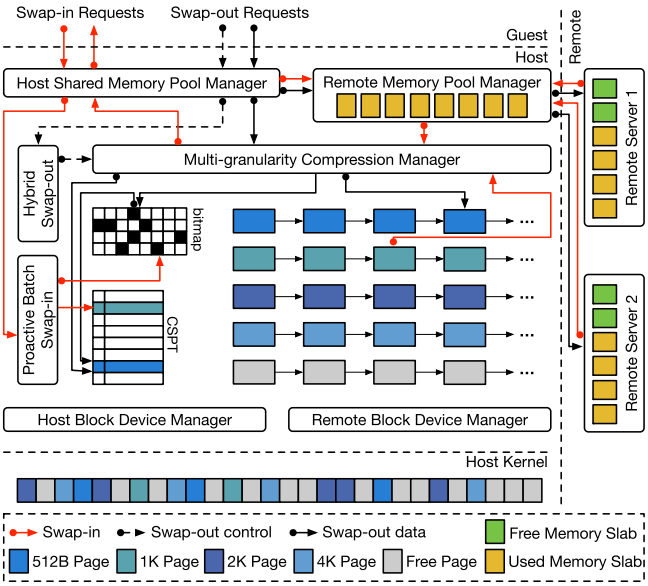


Fig. 4. *XMemPod* optimizations.

deallocation of shared memory pools in unit of slab. Right sizing of slabs is beneficial. Larger slabs can decrease the utilization efficiency of memory sharing. Also, the current sharing strategy for host and remote shared memory is greedy and demand driven.

4 *XMemPod* Optimizations

We introduce three additional optimizations to further improve the efficiency of *XMemPod*. Fig. 4 shows how they are orchestrated in concert for both local host and remote memory sharing.

4.1 Compressed Swap Page Table: CSPT

XMemPod compresses all swap-out pages before placing them into the proper swap partition. All swap-in pages are decompressed first before sent back to the application's working memory. Based on the notion that the compression efficacy mainly depends on two factors: compressibility of data and choice of data granularity [36], [52], *XMemPod* deploys a multi-granularity compression algorithm, which is optimized for achieving the best compressibility: e.g., a 4 KB page can be best compressed to one of the four different sizes: (0,512 B], (512,1 KB], (1 KB,2 KB], and (2 KB,4 KB]. LZ4 [19] is used as the default. Other multi-granularity compression algorithms include LZO-1X [43].

To keep track of compressed swap pages, four compressed page placement queues are created: for 512 B queue, 8 compressed page slots are created for one shared memory page, each holds a compressed page of size in (0,512 B]. Similarly, 4 compressed page slots for 1 KB queue, each slot holds a compressed page of size in (512 B, 1 KB], 2 slots for 2 KB queue in (1 KB, 2 KB], and 1 slot for 4 KB queue in (2 KB, 4 KB]. *XMemPod* creates and maintains a compressed disaggregated memory page table (CSPT) to keep track of each swap-out page and its metadata. With CSPT, the swap-in operation can easily lookup and locate the compressed page from the respective swap partition, and send it back to the guest in decompressed format. A hash-table based index structure [34], [44] is used to optimize read/write operations on the CSPT. When a swap-out operation is intercepted, a *CSPT_entry* is created for the swapped out page and links it to the Hashtable and a bitmap variable field is used to indicate the location of the swap partition where the compressed page currently resides. Other metadata recorded in the *CSPT_entry* includes the address of this compressed page, the length of compressed page which is required for decompression.

4.2 Hybrid Swap-Out

Upon receiving a swap-out request, the PSM intercepts the page and calls the *swap_writepage()* operation. The operation begins by compressing the page into a temporary buffer in order to know which one of the four compressed page-groups this compressed page should belong to. Based on the compressed page size, *XMemPod* chooses the shared memory page in the most suitable compressed page group, locates the free slot with its offset, and places the compressed page in the slot. Next, the PSM checks with SMM to determine if there is sufficient room in the host-VM shared memory pod to serve this swap-out request. When there is insufficient host memory, instead of putting the *most recent pages to remote* (MRPR), we advocates the hybrid swap-out optimization by placing *least recent pages to remote* (LRPR), which maximizes the utilization of host-VM shared memory to serve the most recent paging traffic, and selects remote memory as the next preferred tier for hybrid swap-out. Similarly, when there is insufficient remote memory in the cluster, the hybrid swap-out procedure will split the older pages by their LRU stamps and move the older ones to disk swap partition and make room in the remote memory pod for relatively more recent pages. In both cases, pages remain compressed during hybrid swap-out process and updates are performed on the compressed disaggregated memory page table and the hashtable index for remote memory accordingly.

4.3 Proactive Swap-in

We observe that when the working set of application is big, the page table grows and the PTE lookup cost increases. For frequent swap-in events, this cost can be non-trivial. *XMemPod* accelerates swap-in operations by piggyback of some metadata: when a page is swapped out, *XMemPod* keeps the address of the PTE related to this page as a piece of metadata together with the swapped-out page, and stores the metadata in the corresponding entry of the CSPT for this swap-out page. For each swap-out page, this metadata only takes up 4 bytes. The cost of keeping this metadata in the swap area is only about 1/1000 of the total size of the swapped-out pages. Using this metadata, when a page is swapped into the working memory of the guest, *XMemPod* is able to quickly locate the PTE that needs to be updated by referring to this metadata without the need to scan the page table. Thus, the time spent on accessing the PTE of a swapped page will not increase as the size of the system wide page table grows. This advantage is applied to proactive swap-in from host to guest, from remote to guest, or from remote to host.

When swap-in a page from remote memory pods, the SSM will check if the free pages in the shared memory pod

TABLE 1
Applications Used in Experiments

Workload	Suite/Application	Dataset
PageRank	Spark GraphX	1 million pages
LogisticRegression	Spark Mllib	7.5 million samples
TunkRank	PowerGraph	30 million vertices
Kmean	PowerGraph	7 million samples
SVM	Liblinear	45 thousand samples
YCSB-Memcached	Memcached	20 million records
YCSB-Redis	Redis	20 million records
YCSB-VoltDB	VoltDB	20 million records
Canneal	PARSEC	2.5 million records
Apache Solr	Cloudsuite	12 GB index

TABLE 2
Machine Learning Workload Performance Comparison of Linux, nbdX and *XMemPod* – Completion Time (second)

		Page Rank	Logistic Regression	Tunk Rank	Kmean	SVM
100%	Linux	226	520	319	216	536
	nbdX	227	519	318	213	540
	XMemPod	228	512	319	215	538
75%	Linux	19800	12240	2830	847	1512
	nbdX	302	869	798	451	874
	XMemPod	**238 (83x)**	**545(22x)**	**352(8x)**	**246(3x)**	**585 (3x)**
50%	Linux	25200	18360	4273	22320	15671
	nbdX	1263	1298	954	712	1152
	XMemPod	**297(85x)**	**694(26x)**	**403(11x)**	**284 (79x)**	**691 (23x)**

are above the pre-defined batch swap-in threshold, and if yes, the remote memory pod manager will trigger proactive batch swap-in, which swaps in the requested page together with those nearby based on temporal or spatial locality. The batch swap-in threshold is a XMemPod configuration parameter that is initialized at the system startup time and can be modified by the shared memory manager based on the available free memory at runtime. This batch threshold parameter is defined by a pair of values, the first value specifies the number of pages to be included in the proactive batch when a page is being swapped in, and the second value specifies which pages are chosen with respect to the swap-in page in terms of the temporal or spatial locality. In the first prototype implementation of XMemPod, we only implement spatial locality. When the #page threshold is defined to be k, only those k pages that are near by the page being swapped-in will be included in the batch.

5 Evaluation

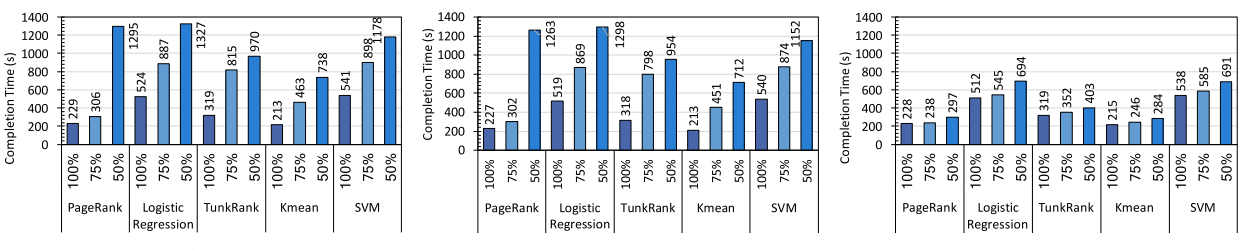
We evaluate *XMemPod* using ten popular memory-intensive applications, with first five machine learning and next five big data applications, listed in Table 1, and compare *XMemPod* with Linux (conventional OS swap facility), Accelio nbdX [1] and Infiniswap [24].

Experiments are performed on a 32-machine, 56 Gbps Infiniband cluster. Each machine has 32 core E5-2650v2 CPU, 64 GB memory, 2 TB SATA 7.2K rpm hard drives, and running KVM 1.2.0 with QEMU 2.0.0 as virtualization platform. We use Linux 4.1.0 and Ubuntu 14.04 for both the guest and host system. For most of the experiments unless otherwise stated, we run 80 VMs on a 32-machine RDMA cluster and created an equal number of VMs for each application workload. We started with the 100 percent configuration by creating VMs with large enough memory to fit entire workload in memory. We measured the peak memory consumption, and then ran 75 and 50 percent configurations by creating VMs with enough memory to fit these fractions of the peak memory consumption. The working sets for the ten applications range from 25 GB to 30 GB and their input dataset sizes range from 12 GB to 20 GB per VM. Unless explicitly stated, we use the default configurations: `sm-SlabSize=512 MB`, `Remote-SlabSize=1 GB`, hybrid swap-out threshold 80 percent, proactive swap-in threshold 50 percent, and remote server connection timeout is `1s`.

For fair comparison, the setups for nbdX, Infiniswap and *XMemPod* use the RDMA registered memory region of the same size for remote memory sharing on each node. In comparison, the Linux setup does not reserve RDMA buffer for remote memory and utilizes the physical memory on each node for its hosted VMs. For the VMs on a local node, *XMemPod* consolidates the total amount of allocated memory per VM to create the host-coordinated shared memory pods at the cost of assigning smaller memory to each of its VMs.

5.1 Impact on Applications

Machine Learning Applications Performance. We first measure and compare the performance of the three systems on PageRank, LogisticRegression, TunkRank, Kmean, and SVM as shown in Table 2. We also compare Infiniswap, *XMemPod Remote*, and *XMemPod*, as shown in Figs. 5a, 5b, and 5c. We highlight three observations: First, for 75 percent configuration, *XMemPod* improves application completion time by 24x on average and up to 83x over Linux, and improves over Infiniswap and nbdX by 2.3x and 2.2x respectively. Second, for 50 percent configuration, *XMemPod* improves application completion time by 45x on average and up to



(a) Infiniswap (b) XMemPod Remote (c) XMemPod

Fig. 5. Machine learning workload performance comparison of infiniswap, *XMemPod Remote* and *XMemPod*.

TABLE 3
Big Data Workload Performance Comparison of Linux, nbdX and *XMemPod* – Completion Time (second)

		Memcached		Redis		VoltDB		Canneal	Solr
		ETC	SYS	ETC	SYS	ETC	SYS		
100%	Linux	397	424	250	310	858	928	113	60
	nbdX	398	423	251	311	858	925	113	60
	XMemPod	397	424	250	310	857	926	112	60
75%	Linux	23640	23781	112359	128205	44943	62305	3623	2216
	nbdX	1630	1669	1475	1585	1545	1619	251	839
	XMemPod	**451(52x)**	**466(51x)**	**265(424x)**	**321(399x)**	**904(50x)**	**993(63x)**	**117(31x)**	**65(34x)**
50%	Linux	84745	85470	158730	217391	235394	333333	6806	3649
	nbdX	2415	2493	2918	3346	1790	1807	479	961
	XMemPod	**494(172x)**	**509(168x)**	**277(573x)**	**355(612x)**	**979(240x)**	**1057(315x)**	**167(41x)**	**72(51x)**

85x over Linux, and improves over Infiniswap by 4.4x (best case) and 2.6x on average, improves over nbdX by 4.3x (best case) and 2.5x on average. Third, For 75 percent configuration, using *XMemPod*, machine learning applications experience only on average 1.1x increase in completion time, comparing to 1.8x using nbdX, 1.9x using Infiniswap and 25x using Linux. For 50 percent configuration, using *XMemPod*, machine learning applications experience only on average 1.3x increase in completion time instead of 2.5x using nbdX, 3.4x using Infiniswap and 59x using Linux.

Big Data Applications Performance. We next measure and compare the three systems on five big data applications: Memcached [7], Redis [8], VoltDB [9], Canneal [3], and Apache solr [2]. For Memcached, Redis and VoltDB, we use the dataset published by Facebook [14] and choose ETC and SYS as workloads to explore different rates of SET operations. ETC is read-intensive workload, which has 5 percent SET and 95 percent GET, and SYS has 25 percent SET and 75 percent GET, which is popularly used as write-intensive workload [14], [24]. Table 3 shows the completion time for vanilla Linux, nbdX, and *XMemPod*. Figs. 6a, 6b, and 6c show the comparison of Infiniswap, *XMemPod Remote*, and *XMemPod*.

We observe three interesting results: (1) For 75 percent configuration, *XMemPod* improves application completion time over Linux by 138x on average and 423x in the best case, and improves over Infiniswap and nbdX by 13.2x and 12.9x in the best case, and 4.7x and 4.5x on average, respectively. (2) For 50 percent configuration, *XMemPod* improves application completion time over Linux by 612x in the best case and 271x on average. It improves application completion time by 13.8x and 13.3x in the best case and 6.6x and 6.2x on average over Infiniswap and nbdX respectively. (3) Applications using *XMemPod* benefit from more steady performance, but experience high degree of fluctuations in performance when using Infiniswap or nbdX.

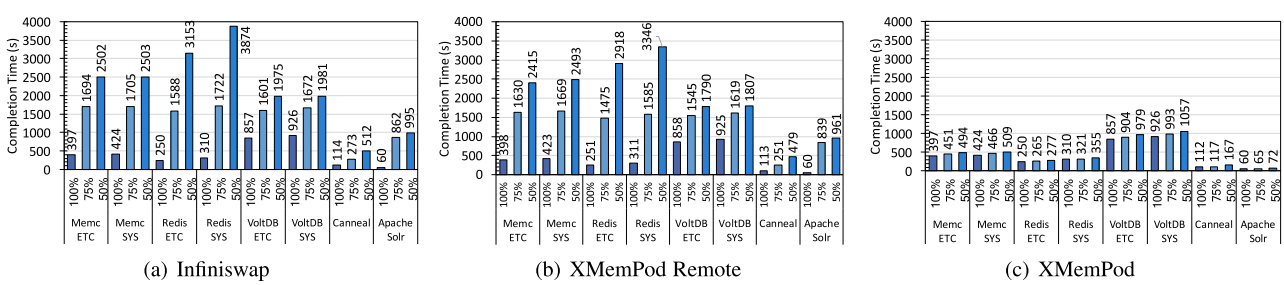
For 50 percent configuration, using *XMemPod*, big data applications experience on average 1.2x increase in completion time, compared to 7.9x using Infiniswap, 7.3x using nbdX and 313x using Linux.

Given that tail-latency is considered a more accurate indicator of performance degradation for bigdata workloads, in addition to median latency, we also measured the 99th percentile response latency for Memcached, Redis and VoltDB respectively for all four systems, as shown in Fig. 7. Using *XMemPod*, the 99th percentile latencies for all applications are much closer to the performance of 100 percent configuration when 50 percent of their working sets do not fit in memory. For example, for 50 percent configuration, using *XMemPod*, the 99th tail latencies of Memcached, Redis, and VoltDB are increased by 42, 16, and 33 percent respectively. In contrast, the 99th tail latencies are increased by 19.7x, 16.0x, and 3.4x respectively by using Infiniswap, and by 16.9x, 13.9x, and 2.5x respectively by using nbdX.

5.2 Cluster Utilization

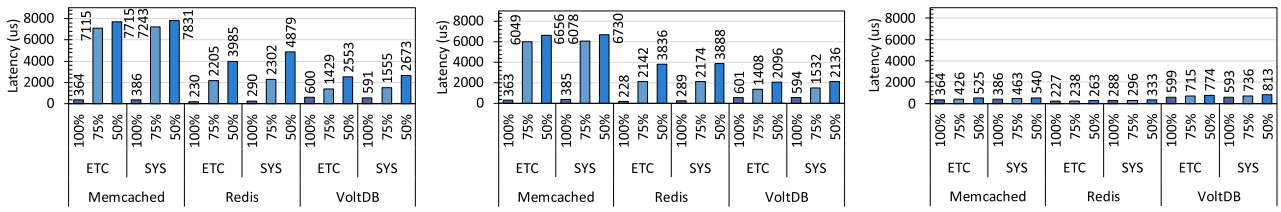
Fig. 8 shows the amount of send/receive data over the RDMA cluster of 32 machines. The total amount of network traffic over RDMA in the case of Infiniswap is 524 GB, while *XMemPod* is 128 GB, which is only 24 percent of Infiniswap. Similar comparison is observed for nbdX.

Fig. 9 compares the overall data distribution with respect to data stored in local host versus data stored in remote nodes for *XMemPod* and Infiniswap. It is interesting to observe that for Infiniswap, the ratio of the average of data stored in local versus remote is 65 percent versus 35 percent.



(a) Infiniswap (b) XMemPod Remote (c) XMemPod

Fig. 6. Big data workload performance comparison of infiniswap, *XMemPod Remote* and *XMemPod*.



(a) Infiniswap (b) XMemPod Remote (c) XMemPod

Fig. 7. Big data workload 99th percentile latency comparison of infiniswap, *XMemPod Remote* and *XMemPod*.

In contrast, for *XMemPod*, the ratio of average of data stored in local versus remote is 90 percent versus 10 percent.

5.3 Host/Remote Memory Distribution

We evaluate the impact of hierarchical memory orchestration of *XMemPod* on Redis, Memcached, VoltVB by varying its host and remote memory distribution using 50 percent configuration. XM-SM denotes 100 percent of paging events are handled in host shared memory. XM-RDMA denotes 100 percent in remote memory via RDMA network, and zero host shared memory is reserved by *XMemPod*. The remaining three host-remote distribution split ratios are XM-9:1, XM-7:3, and XM-5:5. XM-9:1 denotes that 90 percent of paging traffic is served in host-VM shared memory pod and 10 percent of it is sent to the remote memory pods. We compare the performance of *XMemPod* using varying distribution configurations with Linux, Infiniswap and nbdX.

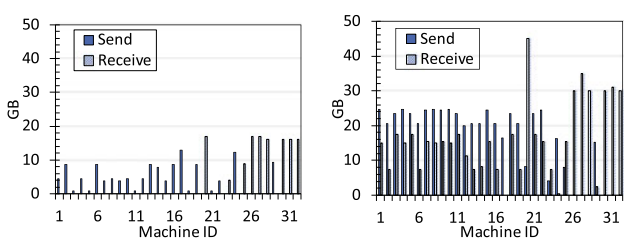
Fig. 10 shows the results. We highlight four observations. *First*, Fig. 10a shows that using XM-SM, throughputs of Redis, Memcached, and VoltDB increase by up to 571x, 171x, and 240x respectively, compared with Linux, increase by 11.4x, 5.1x, and 2.0x compared with Infiniswap, and increase by 10.5x, 4.9x, and 1.8x compared with nbdX. *Second*, using XM-RDMA, throughput of Redis, Memcached and VoltDB increase by 3.2x, 1.8x, and 1.6x respectively, compared to Infiniswap, and increase by 2.9x, 1.8x, and 1.5x respectively, compared to nbdX. *Third*, Fig. 10a shows that the throughputs of all three applications reduce, as the percentage of remote memory increases for *XMemPod* from XM-SM, XM-9:1, XM-7:3, XM-5:5 to XM-RDMA. Figs. 10b and 10c show that the median and 99th percentile latencies using *XMemPod* are significantly shorter for Redis, Memcached, and VoltDB compared to Linux, Infiniswap and nbdX for all distribution settings of host-remote memory sharing. Even with remote only XM-RDMA, the 99th percentile latencies of Redis, Memcached and VoltDB improve by 1.8x, 1.7x, and 3.1x, comparing to Infiniswap, and by 1.7x, 1.5x, and 2.6x, comparing to nbdX.

5.4 Impact of Remote Memory Sharing

In this section, we deploy *XMemPod* on a 32-machine RDMA cluster on Emulab [5]. We turn off the host-VM shared memory option and only rely on remote memory for serving paging requests under 50 percent configuration. We randomly choose a server with two VMs running Memcached and Redis ETC workload respectively, and configure their remote memory slabs to be distributed across multiple remote servers. We measure the performance of remote paging to 2 to 8 servers, each of which runs a VM for Logistic Regression workload. We report two sets of experiments due to space constraint.

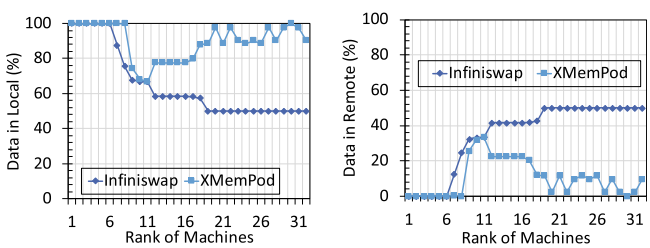
Remote Paging to Multiple Servers. Fig. 11 measures the impact of remote paging on both host and remote application performance. We make three observations. *First*, on host server, using *XMemPod*, all applications complete significantly faster than the scenario without *XMemPod*. Furthermore, *XMemPod* with 2 remote server case improves the completion time of Memcached and Redis by 70x and 179x respectively, over the scenario without *XMemPod*. *Second*, the performance of both Redis and Memcached is stable as we increase the number of servers for serving remote paging. This is because each server only equips with one Infiniband card in the Emulab RDMA cluster, the performance of remote read/write operation is limited by one network card. We expect that *XMemPod* will benefit from parallel remote I/O if multiple Infiniband cards are equipped. *Third*, Logistic Regression on each of the remote servers serving remote paging traffic from Redis and Memcached exhibits stable performance in all cases, which is expected because *XMemPod* uses one-sided RDMA operation to read/write remotely, which completely bypasses remote CPUs.

Impact of Eviction on Application Performance. Fig. 12 measures the impact of remote memory eviction operations on the performance of both host and remote applications. We highlight two observations. First, eviction increases the completion time of Memcached and Redis by 1.3 and 1.4 percent. A primary factor is caused by the process of



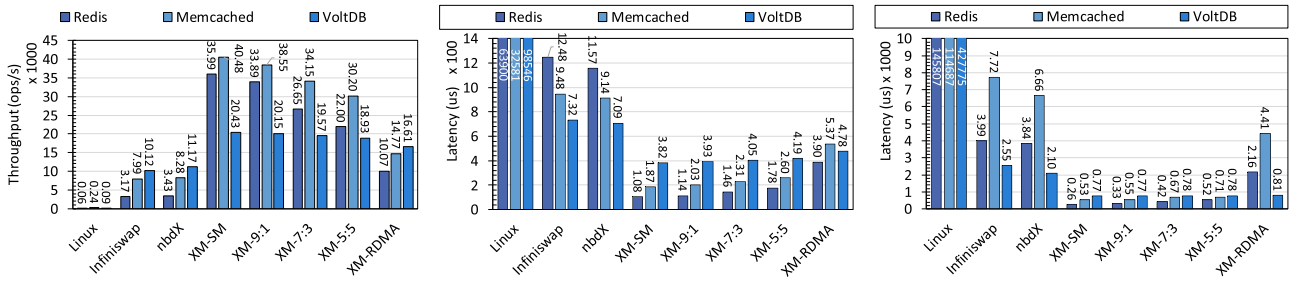
(a) XMemPod Send/Receive Data (b) Infiniswap Send/Receive Data

Fig. 8. Network traffic of individual machines.



(a) Data Stored in Local (%) (b) Data Stored in Remote (%)

Fig. 9. Data distribution of individual machines.



(a) Throughput (b) Median Latency (c) 99th Latency

Fig. 10. Varying host and remote memory sharing distribution (different latency scale in Fig. 10b and 10c).

EVICT event handling, updating CSPT (Section 4.1), and redirecting request to the other remote server that stores the copy of requested memory slab. Second, eviction has no performance impact on Logistic Regression on remote server, as shown in Fig. 12b. The main cost of evicting slab is memory region deregistration. For 1 GB slab, the average cost of it is 273 microseconds. When we increase to 4 GB, 8 GB or 16 GB of slabs, the average cost for memory region deregistration is 1,121, 2,157, and 4,356 microseconds.

6 Related Work

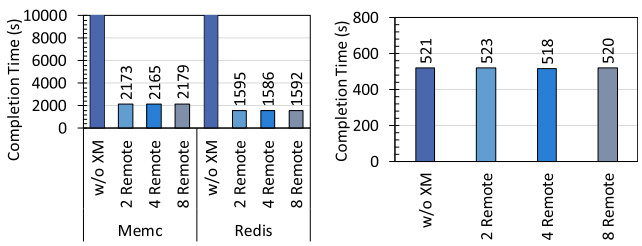
Dynamic Memory Balancing. Existing research on scheduling and consolidating physical memory among VMs centered on dynamic memory balancing, with Ballooning [56] as the most representative. Proposals [26], [62] devoted to periodic estimation of VM working set size. But, accurate working set prediction is hard under changing workloads [30].

Memory Page Compression and Deduplication. Existing proposals for page deduplication [27] aim at identifying duplicate memory pages. Some open source efforts provide transcendent memory (tmem) interface for paging [6] or a compressed RAM cache [33]. Proposals for addressing double paging [11] and memory compression [46], [51] can be leveraged in *XMemPod* to further improve host/remote memory sharing.

Distributed Shared Memory/Disaggregated Memory. Distributed shared memory was studied extensively [12], [15]. However, DSM suffers poor performance due to high communication overhead. Disaggregated memory has attracted much attention recently [13]. Disaggregated remote memory has attracted much attention over the past decade [18], [22], [24], [28], [38], [39], [40]. Most proposals rely on new hardware architecture, new network protocols to cut down the communication cost. Recently, some proposals show the benefit of leveraging RDMA technology [25], [41], such as remote storage for key-value stores[21], distributed objects [55], swap pages [24], object replication [61], and disaggregated datacenters in a box [4]. Most of these efforts lack of desired transparency. Infiniswap [24] is developed on top of Accelio nbdX [1], and both use RDMA read/write interface for fast and reliable access to remote memory regions registered for RDMA networking. But, existing solutions via RDMA access to remote memory cannot be used to implement host-coordinated shared memory sharing among VMs on the same host. To the best of our knowledge, *XMemPod* is the first to implement local memory sharing between VMs on the same host and to promote a hierarchical memory sharing framework with host-coordinated memory sharing first, followed by RDMA-coordinated remote memory sharing. Our experiments show that the *XMemPod* approach to hierarchical orchestration of disaggregated memory is highly effective.

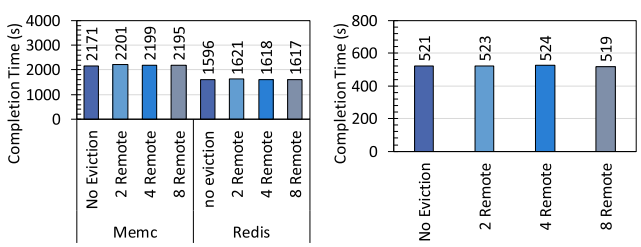
7 Conclusion

We have presented *XMemPod*, a hierarchical disaggregated memory orchestration framework for virtualization of cluster-wide memory. *XMemPod* is the first to implement memory-speed access to available memory in other VMs on the same host, and to promote a hierarchical memory sharing framework with host-coordinated memory sharing first, followed by RDMA-coordinated remote memory sharing. *XMemPod* is deployed on a virtualized RDMA cluster without any modifications to user applications and the host OSes. It works as a loadable kernel module to dynamically extend running OS to support *XMemPod* functionality. It can also be unloaded to free the memory and other resources when the functionality is no longer required. It offers orders of magnitude performance gain for bigdata and ML applications. *XMemPod* incurs no functional nor performance impact on guest VMs that do not use it or that incurs zero paging. Evaluated on unmodified Spark, Apache Hadoop, Memcached, Redis and VoltDB, *XMemPod* improves throughputs of these



(a) Host - Memcached and Redis (b) Remote - Logistic Regression

Fig. 11. *XMemPod* with multiple remote servers.



(a) Host - Memcached and Redis (b) Remote - Logistic Regression

Fig. 12. Eviction impact on performance.

applications over Linux by 11x to 612x, improves median and tail latencies by 174x to 702x and 218x to 630x respectively. Compared to the state of art remote memory paging systems (Infiniswap and nbdX), *XMemPod* improves throughputs by up to 14x, median latencies and tail latencies by up to 12x and 15x respectively. *XMemPod* is released at https://github.com/git-disl/XMemPod.

Acknowledgments

This research has been partially supported by the National Science Foundation under Grants IIS-0905493, CNS-1115375, NSF 1547102, SaTC 1564097, and Intel ISTC on Cloud Computing. Any opinions, findings, and conclusions or recommendations expressed in this article are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Accelio based network block device, 2015. [Online]. Available: https://github.com/accelio/NBDX
[2] Apache Solr, 2005. [Online]. Available: http://lucene.apache.org/solr
[3] Canneal, 2008. [Online]. Available: http://parsec.cs.princeton.edu/overview.htm
[4] dReDBox: Disaggregated data center in a box, 2018. [Online]. Available: http://www.dredbox.eu/publications/
[5] Emulab, 2008. [Online]. Available: https://www.emulab.net
[6] Frontswap, 2012. [Online]. Available: https://www.kernel.org/doc/Documentation/vm/frontswap.txt
[7] Memcached, a distributed memory object caching system, 2003. [Online]. Available: https://memcached.org
[8] Redis, an in-memory data structure store, 2009. [Online]. Available: https://redis.io
[9] VoltDB, a translytical in-memory database, 2010. [Online]. Available: https://github.com/VoltDB/voltdb
[10] M. K. Aguilera *et al.*, "Remote memory in the age of fast networks," in *Proc. Symp. Cloud Comput.*, 2017, pp. 121–127.
[11] N. Amit, D. Tsafrir, and A. Schuster, "VSwapper: A memory swapper for virtualized environments," *ACM SIGPLAN Notices*, vol. 49, pp. 349–366, 2014.
[12] C. Amza *et al.*, "TreadMarks: Shared memory computing on networks of workstations," *J. Comput.*, vol. 29, no. 2, pp. 18–28, Feb. 1996.
[13] K. Asanovic and D. Patterson, "FireBox: A hardware building block for 2020 warehouse-scale computers," in *Proc. USENIX Conf. File Storage Technol.*, 2014, pp. 1–46.
[14] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 40, pp. 53–64, 2012.
[15] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed shared memory based on type-specific memory coherence," in *Proc. 2nd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 1990, pp. 168–176.
[16] R. Birke, L. Y. Chen, and E. Smirni, "Data centers in the wild: A large performance study," IBM Research, Zurich, Switzerland, 2012.
[17] P. Bodík, I. Menache, M. Chowdhury, P. Mani, D. A. Maltz, and I. Stoica, "Surviving failures in bandwidth-constrained datacenters," in *Proc. ACM SIGCOMM Conf. Appl. Technol. Archit. Protocols Comput. Commun.*, 2012, pp. 431–442.
[18] W. Cao and L. Liu, "Dynamic and transparent memory sharing for accelerating big data analytics workloads in virtualized cloud," in *Proc. IEEE Int. Conf. Big Data*, 2018, pp. 191–200.
[19] Y. Collet, "LZ4–Extremely fast compression," Tech. Rep., 2015.
[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.
[21] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "FaRM: Fast remote memory," in *Proc. 11th USENIX Conf. Netw. Syst. Des. Implementation*, 2014, pp. 401–414.
[22] P. X. Gao *et al.*, "Network requirements for resource disaggregation," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 249–264.
[23] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 323–336.
[24] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with INFINISWAP," in *Proc. 14th USENIX Conf. Netw. Syst. Des. Implementation*, 2017, pp. 649–667.
[25] C. Guo *et al.*, "RDMA over commodity ethernet at scale," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 202–215.
[26] F. Guo, "Understanding memory resource management in vmware vsphere 5.0," VMware, Inc, Palo Alto, CA, 2011.
[27] D. Gupta *et al.*, "Difference engine: Harnessing memory redundancy in virtual machines," *Commun. ACM*, vol. 53, pp. 85–93, 2010.
[28] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker, "Network support for resource disaggregation in next-generation datacenters," in *Proc. 12th ACM Workshop Hot Top. Netw.*, 2013, pp. 1–7.
[29] M. Hao, G. Soundararajan, D. R. Kenchammana-Hosekote, A. A. Chien, and H. S. Gunawi, "The tail at store: A revelation from millions of hours of disk and SSD deployments," in *Proc. 14th USENIX Conf. File Storage Technol.*, 2016, pp. 263–276.
[30] M. R. Hines, A. Gordon, M. Silva, D. Da Silva, K. Ryu, and M. Ben-Yehuda, "Applications know best: Performance-driven memory overcommit with Ginkgo," in *Proc. IEEE 3rd Int. Conf. Cloud Comput. Technol. Sci.*, 2011, pp. 130–137.
[31] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *Proc. IEEE 26th Int. Conf. Data Eng. Workshops*, 2010, pp. 41–51.
[32] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," in *Proc. USENIX Conf. USENIX Annu. Techn. Conf.*, 2010, Art. no. 11.
[33] S. Jennings, "The zswap compressed swap cache," in *LWN.net*, 2013. [Online]. Available: https://lwn.net/Articles/537422/
[34] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, "Raising the bar for using GPUs in software packet processing," in *Proc. 12th USENIX Conf. Netw. Syst. Des. Implementation*, 2015, pp. 409–423.
[35] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kVM: The Linux virtual machine monitor," in *Proc. Linux Symp.*, 2007, pp. 1–6.
[36] M. Kjelso, M. Gooch, and S. Jones, "Empirical study of memory-data: Characteristics and compressibility," *IEE Proc.-Comput. Digital Techn.*, vol. 145, no. 1, pp. 63–67, Jan. 1998.
[37] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "SparkBench: A comprehensive benchmarking suite for in memory data analytic platform spark," in *Proc. 12th ACM Int. Conf. Comput. Frontiers*, 2015, pp. 1–8.
[38] S. Liang, R. Noronha, and D. K. Panda, "Swapping to remote memory over InfiniBand: An approach using a high performance network block device," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2005, pp. 1–10.
[39] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *Proc. Annu. Int. Symp. Comput. Architecture*, 2009, pp. 267–278.
[40] K. Lim *et al.*, "System-level implications of disaggregated memory," in *Proc. IEEE Int. Symp. High-Perform. Comp Archit.*, 2012, pp. 1–12.
[41] F. Mietke, R. Baumgartl, R. Rex, T. Mehlan, T. Hoefler, and W. Rehm, "Analysis of the memory registration process in the mellanox InfiniBand software stack. 8 2006," in *Proc. Eur. Conf. Parallel Process.*, 2006, pp. 124–133.
[42] J. Nelson *et al.*, "Latency-tolerant software distributed shared memory," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2015, pp. 291–305.
[43] M. Oberhumer, "LZO real-time data compression library," User Manual for LZO, 2008. [Online]. Available: http://www.oberhumer. com/opensource/lzo/
[44] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, pp. 122–144, 2004.
[45] O. Paz, "InfiniBand essentials every HPC expert must know," in *Retrieved August*, 2014. [Online]. Available: http://www.hpcadvisorycouncil.com/events/2014/swiss-workshop/presos/Day_1/1_Mellanox.pdf
[46] G. Pekhimenko *et al.*, "Linearly compressed pages: A low-complexity, low-latency main memory compression framework," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2013, pp. 172–184.

[47] R. Power and J. Li, "Piccolo: Building fast, distributed programs with partitioned tables," in *Proc. 9th USENIX Conf. Operating Syst. Des. Implementation*, 2010, pp. 293–306.
[48] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proc. 3rd ACM Symp. Cloud Comput.*, 2012, pp. 1–13.
[49] C. A. Reiss, "Understanding memory configurations for in-memory analytics," PhD thesis, Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, Berkeley, CA, 2016.
[50] A. W. Richa, M. Mitzenmacher, and R. Sitaraman, "The power of two random choices: A survey of techniques and results," *Combinatorial Optim.*, pp. 255–304, 2001.
[51] S. Roy, R. Kumar, and M. Prvulovic, "Improving system performance with compressed memory," in *Proc. 15th Int. Parallel Distrib. Process. Symp.*, 2001, pp. 7 pp.-.
[52] R. B. Tremaine, T. B. Smith, M. Wazlowski, D. Har, K.-K. Mak, and S. Arramreddy, "Pinnacle: IBM MXT in a memory controller chip," *IEEE Micro*, vol. 21, no. 2, pp. 56–68, Mar./Apr. 2001.
[53] V. K. Vavilapalli *et al.*, "Apache hadoop YARN: Yet another resource negotiator," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, pp. 1–16.
[54] J. Vienne, J. Chen, M. Wasi-Ur-Rahman, N. S. Islam, H. Subramoni, and D. K. Panda, "Performance analysis and evaluation of InfiniBand FDR and 40GigE RoCE on HPC and cloud computing systems," in *Proc. IEEE 20th Annu. Symp. High-Perform. Interconnects*, 2012, pp. 48–55.
[55] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, "A note on distributed computing," in *Proc. Int. Workshop Mobile Object Syst.*, 1996, pp. 49–64.
[56] C. A. Waldspurger, "Memory resource management in VMware ESX server," *ACM SIGOPS Operating Syst. Rev.*, vol. 36, pp. 181–194, 2002.
[57] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner, "Memory buddies: Exploiting page sharing for smart colocation in virtualized data centers," *ACM SIGOPS Operating Syst. Rev.*, vol. 43, pp. 31–40, 2009.
[58] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation*, 2012, Art. no. 2.
[59] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Top. Cloud Comput.*, 2010, Art. no. 10.
[60] P. Zhang, X. Li, R. Chu, and H. Wang, "HybridSwap: A scalable and synthetic framework for guest swapping on virtualization platform," in *Proc. IEEE Conf. Comput. Commun.*, 2015, pp. 864–872.
[61] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, "Mojim: A reliable and highly-available non-volatile memory system," *ACM SIGPLAN Notices*, vol. 50, pp. 3–18, 2015.
[62] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," *ACM SIGOPS Operating Syst. Rev.*, vol. 39, pp. 177–188, 2004.

Wenqi Cao received the master's degree from the Computer Science and Engineering Department, Pennsylvania State University, State College, Pennsylvania, and has five years work experience as software engineer working for SIEMENS, HP, and Microsoft. He is currently working toward the PhD degree in computer science and is a research assistant with the School of Computer Science, Georgia Institute of Technology, Atlanta, Georgia. His research interest includes cloud computing, distributed system, and operating system.

Ling Liu (Fellow, IEEE) is a professor with the School of Computer Science, Georgia Institute of Technology, Atlanta, Georgia. She directs the research programs in Distributed Data Intensive Systems Lab (DiSL), examining various aspects of large scale data intensive systems, including performance, availability, security and privacy. She is an internationally recognized expert in the areas of big data systems and analytics, database systems, distributed computing, internet data management, and service oriented computing. She has published more than 300 international journal and conference articles and is a recipient of the best paper award from a number of top venues.

Accurate Cost Estimation of Memory Systems Utilizing Machine Learning and Solutions from Computer Vision for Design Automation

Lorenzo Servadei, Edoardo Mosca, Elena Zennaro, Keerthikumara Devarajegowda, Michael Werner, Wolfgang Ecker, *Member, IEEE*, and Robert Wille, *Senior Member, IEEE*

Abstract—Hardware/software co-designs are usually defined at high levels of abstractions at the beginning of the design process in order to provide a variety of options on how to realize a system. This allows for design exploration which relies on knowing the costs of different design configurations (with respect to hardware usage and firmware metrics). To this end, methods for cost estimation are frequently applied in industrial practice. However, currently used methods oversimplify the problem and ignore important features, leading to estimates which are far off from real values. In this article, we address this problem for memory systems. To this end, we borrow and re-adapt solutions based on *Machine Learning* (ML) which have been found suitable for problems from the domain of *Computer Vision* (CV). Based on that, an approach is proposed which outperforms existing methods for cost estimation. Experimental evaluations within an industrial context show that, while the accuracy of the state-of-the-art approach is frequently off by more than 20 percent for area estimation and more than 15 percent for firmware estimation, the method proposed in this article comes rather close to the actual values (just 5–7 percent off for both area and firmware). Furthermore, our approach outperforms existing methods for scalability, generalization, and decrease in manual effort.

Index Terms—Machine learning, design automation, Hardware-software co-design, deep learning

1 Introduction

Increasing the productivity in industrial hardware/software co-designs is a central issue, as it allows for an efficient development flow as well as a reduction of the design costs.

Automating design tasks early in the development process enable designers to efficiently construct the complex modern chips. Hardware (HW) and Firmware (FW) are 2 major constituents of any modern System-on-Chip (SoC) and, several techniques have been proposed for automating hardware and firmware components [9], [12], [27]. These techniques adapt model-driven approaches in order to specify the system at an higher abstraction level. This in turn allows systems to be represented in abstract structures with required attributes and excluding implementation details. The designer then has the flexibility to create multiple design configurations and realize the system with the desired configuration.

In order to automate the hardware components, a generation flow that focuses on the design-centric models is developed [11]. Here the generation is conceptually divided into multiple layers, which includes model-to-model transformations. The most abstract models are close to the specifications of the system and the least abstract models are close to the system realization in a specific platform (ex: RTL in VHDL/Verilog/SystemVerilog). In order to enhance the configuration capability, the system can be built in a top-down fashion using the generation flow. For the same, a system can be considered as a component tree with the top level component instantiating its child components (with additional program logic) and so on. Individual components are automated considering all required/specified configurations and hence, the system is an assembly of components and sub-components with high degree of configuration capability.

This enables design exploration at early stages of the design process, since plenty of options of how to realize a system can be evaluated prior to its actual implementation. By this, the designer can make sure that a system is realized in the needed customized fashion, and also satisfies certain cost constraints (e.g., with respect to area, number of firmware instructions, etc.). In order to evaluate the different possible design configurations, methods for *cost estimation* are essential.They take properties from the design configuration and try to extrapolate the cost of each implementation based on the configuration. This information is then utilized by the designer to eventually decide which configuration shall be realized. Within this process, the quality of the cost estimation obviously is a crucial criteria–misleading cost estimates will eventually lead to the implementation of design configurations which likely is not satisfactory. In this work, we consider this issue for the design of memory systems.

- *L. Servadei, E. Zennaro, K. Devarajegowda, M. Werner, and W. Ecker are with Digital Enabling Systems, Infineon Technologies AG, 85579 Neubiberg, Germany. E-mail: {lorenzo.servadei, elena.zennaro, keerthikumara.devarajegowda, michael.werner, wolfgang.ecker}@infineon.com.*
- *E. Mosca is with Mathematics, Technische Universitat Munchen, 80333 Munchen, Germany. E-mail: edoardo.mosca@tum.de.*
- *R. Wille is with the Institute for Integrated Circuits, Johannes Kepler Universitat Linz, 4040 Linz, Austria. E-mail: robert.wille@jku.at.*

Manuscript received 15 Feb. 2019; revised 31 Dec. 2019; accepted 8 Jan. 2020. Date of publication 23 Jan. 2020; date of current version 8 May 2020. (Corresponding author: Lorenzo Servadei.) Recommended for acceptance by S. Ha. Digital Object Identifier no. 10.1109/TC.2020.2968888

However, the cost estimation methods that are currently used in industrial practice [23], [29], [35] often oversimplify the problem and neglect important features that strongly influence costs. Consequently, they frequently lead to cost estimates that are far off from the real values (this is discussed in more detail later in Section 2). At the same time, we observe that explicitly recognizing those features and "hard-code" an algorithm, considering all features in order to derive a more accurate estimation, is a cumbersome task. Hence, we investigated an alternative solution which borrows and re-adapts concepts based on *Machine Learning* (ML, [15]) which have been found suitable for problems in the domain of *Computer Vision* (CV, [3]).

In this work,[1] we report on these investigations as well as propose the correspondingly resulting methodology. More precisely, we observe that, for typical CV problems such as the determination of the age of a person depicted in an image, ML indeed properly recognizes the respective features and is capable of determining rather accurate estimates. Further, we show that age determination in CV and cost estimation of a memory system share some similarities and can actually be addressed by the same scheme. Based on that, we propose an alternative solution for cost estimation which adapts the concepts from the CV domain.

Experimental evaluations conducted within an industrial environment show that the resulting approach clearly outperforms the state-of-the-art used thus far. While a state-of-the-art approach used until now frequently is off by more than 20 percent for area estimation and more than 15 percent for firmware estimation, the method proposed in this work comes rather close to the actual values (just 5-7 percent off for both area and firmware). Moreover, the proposed method offers higher scalability and flexibility since further features can be directly and automatically learned through the ML training process, and do not need a further hand-tailored pre-processing.

In the following, the proposed approach is described as follows: Section 2, briefly reviews cost estimation for memory systems as conducted thus far–including a discussion of the shortcomings of today's state-of-the-art methods. Motivated by this, we outline and describe the proposed idea in Section 3 followed by a description of the implementation of the methods for accurate cost estimation in Section 4. Finally, Section 5 summarizes the results of the conducted evaluation before the paper is concluded in Section 6.

2 Cost Estimation for Memory Systems

In this section, we illustrate the problems and challenges of cost estimation for memory systems in early stages of the design flow. To this end, we first briefly introduce the hardware generation method used in our industrial environment flow as described in [30]. Afterwards, we present the specification of *Register Interface* (RI) components inspired from [10] which serve as running example in the remainder of this work. We opt for the implementation of this component because of its major importance in the industrial design case. In fact, RIs are almost ubiquitous in SoCs and essential

1. A preliminary work-in-progress report has been published before in [21].

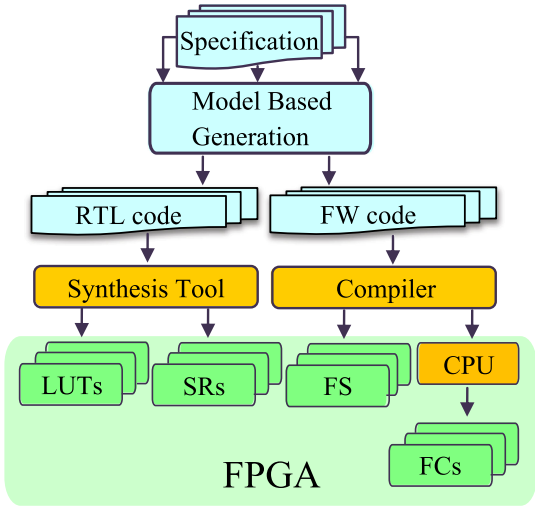


Fig. 1. Model based hardware generation flow.

for HW/FW configurability, as described in [10]. Using this example, today's shortcomings of cost estimation are illustrated–providing the motivation of this work.

In the hardware generation flow of our industrial environment illustrated in Fig. 1, we aim to instantiate hardware designs complying to given specifications. Hence, through a model-based generation approach (thoroughly described in [7], [30]), the designer instantiates a particular configuration of the component through an instance of the *meta-model* following the design requirements. This configuration would then trigger the generation of RTL as well as firmware code. Finally, the cost with respect to hardware and firmware metrics is obtained from the synthesis/compilation of the previously generated code. The obtained cost would then guide the designer towards a suitable implementation of hardware together with firmware on an FPGA board. This same procedure applies to a set of hardware components, including RIs.

RIs are common components in an SoC and provide the control and status mechanism between the core and peripheral devices. Accordingly, RI components may be required in different *configurations* (e.g., depending on the respectively applied peripherals) which is why the initial specification is usually provided generically in terms of a meta-model such as follows:

Example 1. Fig. 2 shows the meta-model of the considered RI component.[2] Here, we describe sub-components that describe the HW and FW structure of the RI. The sub-component *Interface* specifies the general features of the RI such as the *DataWidth* or the *AddressWidth*. The single registers are defined using instances of the sub-component *Unit*, which has a *Name*, a *Size*, and an *Address*. The accessibility to each bitwise position in the registers/*Unit*s is defined by dedicated bitfields, which is why each *Unit* has one or more instances of a corresponding sub-component *Contained* (specifying the start *Position* of a

2. Note that, for sake of clarity, we focus only on the sub-components which are relevant for the remaining discussions. A complete description of this meta-model is available in an online appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2020.2968888 [11].

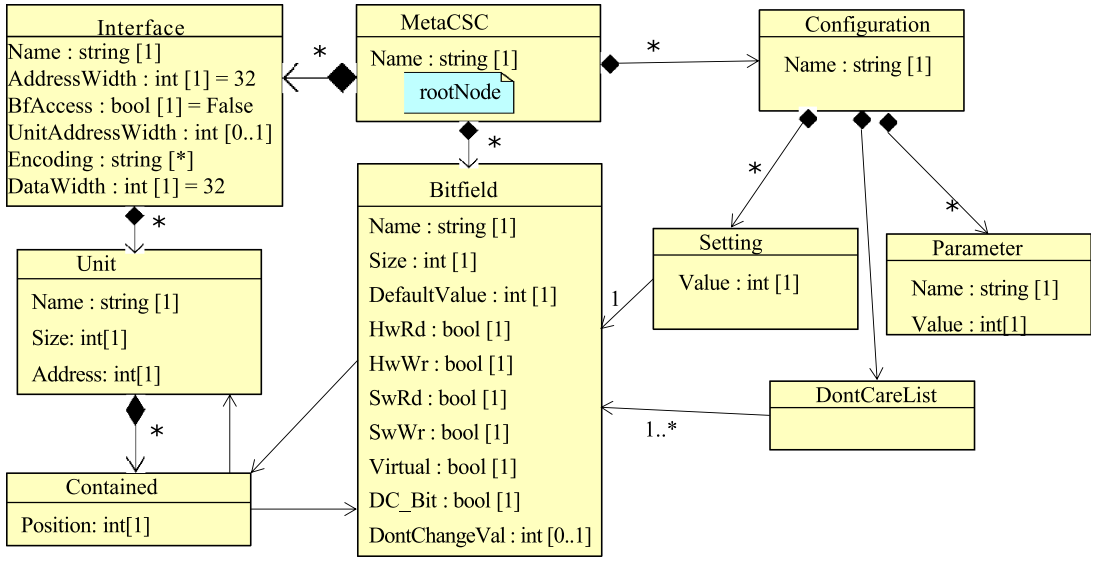


Fig. 2. Meta-model of the *Register Interface* (RI).

bitfield) and the sub-component *Bitfield* (specifying e.g., the *Size* and the allowed access; e.g., *HwRd* and *SwWr* regulating the property being read or written respectively by HW and FW).[3] Additionally, the corresponding FW (which eventually has to obey these accessibility settings) is described using the additional sub-components *Configuration*, which enlist the bitfields involved in the reading/writing operations, and *Setting*, which specify the default/reset value for the bitfields. Last, the *DontCareList* sub-component contains a reference to the bitfields not considered in the FW writing operation, and *Parameter* which is specified depending on the desired instances of the *Bitfield*s.

Using this meta-model, the designer can now instantiate various configurations of an RI component. In the following, we call these instances *design configurations*. Appropriate configurations have to be chosen such that the intended behavior is realized (e.g., enabling proper access to peripheral devices), while satisfying specified cost constraints. For the latter, it is important to have an accurate estimation of the costs for a correspondingly considered configuration. In the following, as shown in Fig. 1, we thereby consider:

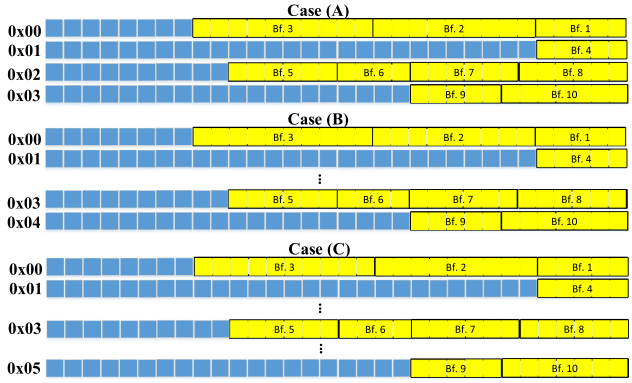
- the area, i.e. the number of *Configurable Logic Blocks* (CLBs) in terms of *Logic Units* (LUTs) and *Slice Registers* (SRs) which are needed to realize the configuration on an FPGA board,
- the size of the generated binary FW code (FS) as well as the number of cycles of a pipelined CPU which are needed to execute the FW program (FCs).

However, in early stages of the design flow, the designer has no fully-fledged implementation of the respective design configurations at hand (they are still to be implemented). Accordingly, the resulting costs need to be estimated in order to evaluate different design choices and, eventually, decide which indeed shall be realized. High-Level Synthesis Tools can be utilized for this purpose [26], and have as well been employed for HW/SW co-design of SoC FPGAs [32]. In our industrial context, with the purpose of learning a fast and accurate estimation for a whole set of designs, we utilize ML approaches. To this end, several methods using ML algorithms for *cost estimation* have been proposed in the past. They use e.g., coarse-grained inputs (e.g., means and aggregate values) as proposed in [23], [29], [31], [35] or high level and application specific ones, as seen in [18]. More precisely, in [35], the features used for CLBs estimation are aggregated for each design generation, resulting in a coarse grained feature space used for the prediction of the SoC area objective. The work [29] instead evaluates the power consumption of different algorithms using high-level features taken from single algorithmic blocks, such as *average working set size* and *total operations*. These are used for predicting power and performance of an FPGA-based soft processor. In [23], different features (here presented as tunable knobs) are evaluated as input for the multi-objective estimation problem, e.g., the *Num. Work Items per group*, the *Num. Work Items per group* and *Num. Private Variables*. Last, in [18], the estimation of the SoC area reduction is computed processing high level configuration data in an application specific manner, thus lacking in flexibility, generalization and not considering multiple objectives (e.g., both software and hardware related).

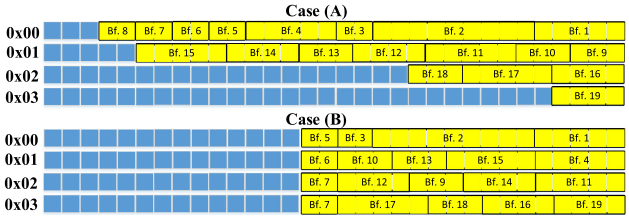
In fact, spatial information such as the spatial position of the bitfields and units inside the memory system have a significant impact on the eventual costs of the estimation. Each bitfield property is set in a configuration context–representing features as a total or mean of values. If provided on a too high (imprecise) level, this may lead to a diminishing of specific information that affect the real cost. At the same time, the complexity of considering those spatial information increases with the size of the considered systems–yielding either a significantly increasing manual effort for the retrieval of feature values or even making it even impossible at all.

Example 2. Consider the instances of the RI meta-model from Fig. 2 as shown in Fig. 3. Here, Fig. 3a shows three different instances which, at a first glance, look identical (e.g., in all three cases the bitfields are the same, and are instantiated in an identical fashion). In fact, the only difference is

3. Note that the structure shown in Fig. 2 allows for different *Unit*s to employ the same *Bitfield*.



(a) Different addresses



(b) Different bitfields mapping

Fig. 3. Design configurations from the RI meta-model.

that the units have different *Address*es. At a first glance, this should not cause significant differences in the costs. Accordingly, existing methods for cost estimation methods yield an estimation of 110 LUTs, 36 SRs, a FS of 2.6 Kb, andc564 FCs for *all* cases. However, if we evaluate the HW Area and retrieve the FW metrics from the actual implementation of those instances,[4] we obtain different values for each case. In fact, as discussed above, the spatial information of each unit indeed has an impact which is why e.g., the realization of Case (a) from Fig. 3a eventually costs 83 LUTs, 42 SRs, a FS of 2.2 Kb, and 524 FCs. The realization of Case (b) from Fig. 3a eventually costs 117 LUTs, 42 SRs, a FS of 2.2 Kb, and 524 FCs. Finally, in Case (c) of Fig. 3a, the costs are 156 LUTs, 42 SRs, a FS of 2.2 Kb, and 524 FCs.

These differences between estimations and real costs become even more substantial if variations on the spatial position of bitfields inside the registers occur as illustrated e.g., by the different cases shown in Fig. 3b. Here, exactly the same bitfields are applied in both cases–only their distribution amongst the units/registers, i.e. their spatial distribution, is different. Accordingly, state-of-the-art cost estimation methods extrapolate the *same* costs for *both* cases, namely 89 LUTs, 97 SRs, a FS of 2.0 Kb, and 476 FCs. However, as discussed above, also here the spatial position of the bitfields has an impact on costs; for this reason, the actual costs for Case (a) are 97 LUTs, 89 SRs, FS of 2.2 Kb, and 576 FCs, while for Case (b), they are 79 LUTs, 89 SRs, FS of 1.4 Kb, and 388 FCs.

These examples illustrate that both, with respect to actual but also relative values between the cases, previously proposed cost estimations are far off and actually lead to rather misleading results–a serious problem when it comes to finally decide which design configuration shall be realized. At the same time, this motivates the development of more precise cost estimation methods which additionally consider the further characteristics discussed above. However, considering all possible features relevant for cost estimation is a cumbersome task which cannot easily be incorporated e.g., by simply adding additional features. Because of that, we are proposing a complementary approach which is described in the next section.

3 Proposed Solution

In this work, we address the problem of cost estimation sketched above. To this end, we borrow concepts from the recently developed fields of *Machine Learning* as well as *Computer Vision*. We show that existing methods, e.g., to process images and videos based on ML, can actually easily be adapted for the purpose of cost estimation. In the following, we describe the proposed methodology as follows: We first review data preparation and basic approaches of ML methods used today for CV tasks. Afterwards, we describe how the features which affect the costs of a memory system can be represented in a similar fashion to pictures for CV algorithms. Eventually, the resulting representation is applied to (existing) ML approaches–providing accurate estimates.

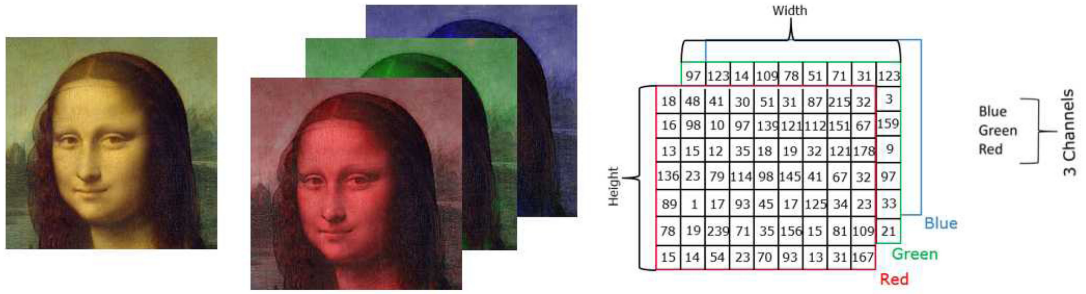
3.1 Machine Learning for Computer Vision

Computer Vision (CV, [3]) is an interdisciplinary area which is mainly concerned with the computational analysis and understanding of single images or sequence of them (i.e. videos). Typically CV requires a particular processing of the image signal, so that specific tasks can be performed (e.g., object detection, classification, age estimation, etc.). Originally, the state-of-the-art methods to accomplish such tasks heavily relied on manual labor, i.e. features were often extracted manually from image pixels before they were processed further [28]. Nowadays, through the availability of more sophisticated ML algorithms, pixel-based representations are often chosen as a direct input to the ML algorithms [5], [15]. In fact, through an extended use of *Neural Networks* (NNs, [5], [15]) (a set of ML algorithms which is loosely inspired by biological neurons), high accuracy in CV tasks can be reached by taking raw images as input and compute features automatically within the learning phase of the ML process [5], [15].[5] For this reason, the representation of images becomes of central importance in CV.

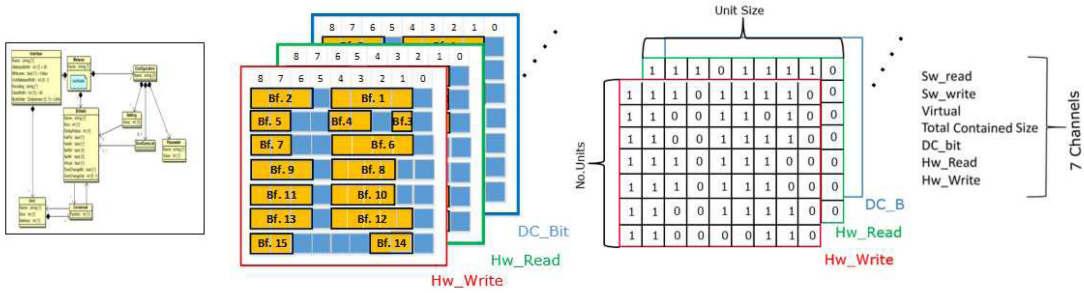
In the following, we utilize the common representation of images in terms of a function $f(h, w, c)$, where h, w, c are coordinates of a 3D matrix $\mathcal{I}^{H \times W \times C}$. Here, H is the height of the image, W the width of the image, and C the number of so-called *channels* of this image. All entries of the 3D matrix (and, hence, all functional values of $f(h, w, c)$) define the *intensity value* v for the respective position and channel. In other words, for a gray scale image, the matrix has $C = 1$ channel where each matrix entry represents the respective gray scale at the corresponding position. Similarly, for an

4. In the (industrial) setting considered here, we used the commercial *Vivado Design Suite* by Xilinx to explicitly implement the given instances.

5. Note that ML and NNs have also succesfully been implemented in domains such as classification [19], text understanding [36], and human activity recognition [8].



(a) Images RGB



(b) Hardware Images

Fig. 4. Representation for CV and cost estimation tasks.

RGB image, the 3D matrix has $C = 3$ channels (one for each of the colors red, green, and blue) where each 3D matrix entries represents the respective portion of these three colors at the corresponding position. For RGB images, the value v is usually within $v \in [0, 255]$ for each one of the three channels.

Example 3. Consider the image shown on the left-hand side of Fig. 4a. Following a typical RGB structure, this image can be defined as the combination of the corresponding red, green, and blue portions as sketched in the center of Fig. 4a. This in turn can be represented in terms of a function $f(h, w, c)$ with a total of $C = 3$ channels, i.e. as a 3D matrix $\mathcal{I}^{H \times W \times 3}$, where H is the pixel-wise height of the RGB image and W is the pixel-wise width. Each one of the channels is therefore structured as a $\mathcal{O}^{H \times W}$ 2D matrix with one pixel value v at each coordinate.

Using this representation, CV tasks such as object detection, classification, segmentation can now be conducted using ML algorithms [5], [15]. For example, let's take a CV task where, given a RGB image of a person, we want to estimate his/her age. To this end, several parts of the human body shown in the image may provide important information about the age, e.g., wrinkles in the skin, color of the hair, etc. However, it obviously is rather difficult to explicitly "hard-code" a computer program which reliably reveals these useful features (e.g., because of the variety in the images set: some images might be zoomed, some badly/differently illuminated, some might be occluded, etc.). Here, ML can provide valuable support.

More precisely, the task is handled as a regression problem (which is suitable, since the desired output should be a number indicating the person's age) where first a *training set* of images is provided for which the respective age of the person is known. Using this training set the ML algorithm can "learn", by minimizing a function that describes the difference between the age that our method predicts and the real age that is included as a label in the training data, all the features mentioned above not in an explicit (i.e. "hard-coded") fashion. In other words, the ML algorithm learns through associations with the image and the respectively given age (to this end, meaningful patterns in the image, the distribution of pixels, and the linked labels guide the ML algorithm to outpoint the estimated age of the person). The pixels of the image are thereby algorithmically processed in order to establish a consistent association of the given image and the given age, as shown in [34].

3.2 Corresponding Representation for a Memory System

Now, recall that, as discussed and illustrated in Section 2, current methods for cost estimation suffer from the fact that they do not properly consider all features that might affect the costs. At the same time, "hard-coding" a computer program which reliably reveals these useful features is hard as well (because of the same reasons why it is hard in CV to determine the age of a person). However, the same method applied for age determination in the domain of CV can also be applied in the domain of cost estimation for memory systems. To this end, we just need corresponding representations for the domain considered here. This is introduced in this section.

More precisely, rather than a RGB image (e.g., providing the pixel intensity v for each coordinate and channel), a similar data-structure for the considered memory system (e.g., providing bitfield positions, bitfield properties, etc.) is required. With such a representation, the same solutions can be used as already successfully utilized in the CV domain. In the following, we describe the proposed data-structures–one for the hardware and one for the firmware.

The hardware data-structure (called *HW image* in the following) represents the respective properties of the memory

system in terms of a function $j(q, l, b)$ where q, l, b are coordinates of a 3D matrix $\mathcal{A}^{Q\times L\times B}$. In contrast to CV, Q now defines the number of units/registers in the memory system, while L defines the respective bit-width and B defines the number of bitfield properties considered. Instead of three channels as in the CV domain (for red, green, and blue), we now use a total of seven channels to represent the HW features. The channels indicate whether the corresponding position in the memory system contains a bitfield (*Bfs*), and if the bitfield allows for a hardware write (*HwWr*), hardware read (*HwRd*), software write (*SwWr*), software read (*SwRd*), a don't change bit (*DC_Bit*), or a virtual bit (*Virtual*). Since these properties either hold for a position in a bitfield or not, they can easily be structured in a binary representation, i.e. each entry of the 2D representation matrix is now a binary value where 1 denotes that the position of the bitfield has the considered property, whereas the value 0 is inserted elsewhere. This results in a *feature representation*, which preserves the spatial structure of the RI for a specific bitfield property and, at the same time, allows a simple addition or removal of features to the hardware data-structure. Furthermore, the data-structure captures aggregate values and statistics among features, which do not need to be explicitly specified. All the above-mentioned traits address the desired characteristics for the cost estimation of memory systems.

Example 4. Consider again the instance of a memory system as shown in the top of Fig. 3a and discussed in Example 2. Those properties are usually represented in terms of a .uml file as sketched in left-hand side of Fig. 4b. Overall, this yields properties with respect to hardware write, hardware read, software write, etc. as sketched in the center of Fig. 4b. Those are eventually represented in terms of matrices as sketched in the right-hand side of Fig. 4b.

Using this data-structure, the HW costs of a memory system can be determined similarly to the age determination in the CV-domain discussed before. That is, first a training set of memory system instances is provided for which the respective real area costs are available. This is used to "learn" the respective features relations and an accurate information processing. Afterwards, proper estimations for the instances for which we actually want to determine the costs can be obtained.

Next, in order to obtain the cost for firmware operations, the corresponding data-structure becomes a bit more challenging. Here, the sequence of reading/writing operations specifies which bitfield/s should be read or written, from/in which register, at each point of the sequence. This is particularly interesting for a FW cost estimation of the RI, as the type of writing operation performed (i.e., the no. of instructions needed) depends on the affected register configuration. Furthermore, distinct types of writing operations have a different impact on the FW size and firmware cycles for a particular design.

In order to outline a corresponding data-structure for the firmware, we get inspiration once again from the CV domain. The evaluation of the firmware cost can be thought of in fact as the same estimation of the age of a person through a multi-channel video. This actually gives a further dimension, so that we can observe different angles on the wrinkles or a person's walking posture over time. In order to adapt this approach for firmware cost estimation, we transform once again the writing and reading operations, together with other RI properties, into binary representations. These correspond to the features representations at each point of the sequence (i.e. frames).

This results in a firmware data-structure (called *FW sequence* in the following) covering the properties of the memory system in terms of a function $z(q, l, d, p)$, where q, l, d, p are coordinates of a 4D matrix $\mathcal{F}^{Q\times L\times D\times P}$. Again, Q defines the number of units/registers in the memory system, while L defines the respective bit-width and D defines the number of considered bitfield properties. That is, in the case of the firmware structure, we have six different channels where each channel is denoted by $\mathcal{F}_d$ with $d \in [1, 6]$. Besides that, P now additionally defines the number of frames and, by this, incorporates the time aspect ($p \in [1, P]$). Following this, each binary feature representation $\mathcal{F}_1^p$ contains the value 1 in the bitwise position of registers containing bitfields to be written; the same approach is applied to the reading operation ($\mathcal{F}_2^p$). Both of these pieces of information are contained in the property *Configuration* of the RI meta-model. A further representation is added for the property *DontCareList* ($\mathcal{F}_3^p$). Once the writing and reading operations, as well as the *DontCareList*, are structured in the corresponding binary representations, they are stacked at the same sequence frame $\mathcal{F}^p$. This composes the dynamic part of the data-structure for the firmware estimation. Successively, three further representations are added to the same frame, corresponding to a static representation of the *HwRd*-feature ($\mathcal{F}_4^p$) and the *DC_Bit*-feature ($\mathcal{F}_5^p$) as well as a representation of the total bitfields ($\mathcal{F}_6^p$) of the design. These values do not change from frame to frame and, hence, are implemented as repeated static representations at each point $\mathcal{F}^p$ of the sequence.

Using this representation, an ML algorithm is equipped with all the information related to firmware and can conduct the cost estimation in a similar fashion as sketched above for the hardware image.

4 Implementation

In this section, we provide the technical details about a possible implementation of the concepts introduced above. To this end, we first present how ML algorithms for CV are utilized to the proposed data-structure so that they can be utilized for the considered cost estimation problem. Afterwards, we provide details on how we optimize the non-trainable parameters of the utilized networks to further improve the obtained accuracy.

4.1 Implementation of the Machine Learning Algorithms

In order to describe the implementation of the methods proposed above, we distinguish between the estimation of the HW costs (LUTs and SRs) and the estimation of the FW costs (FCs and FS). For this reason, we consider two multiple outputs regressions as supervised learning problems, i.e., where the ML algorithm is guided through the learning phase by the real values of the HW Area and of the FW Metrics. The task of the regressions is then to predict the *outputs*,

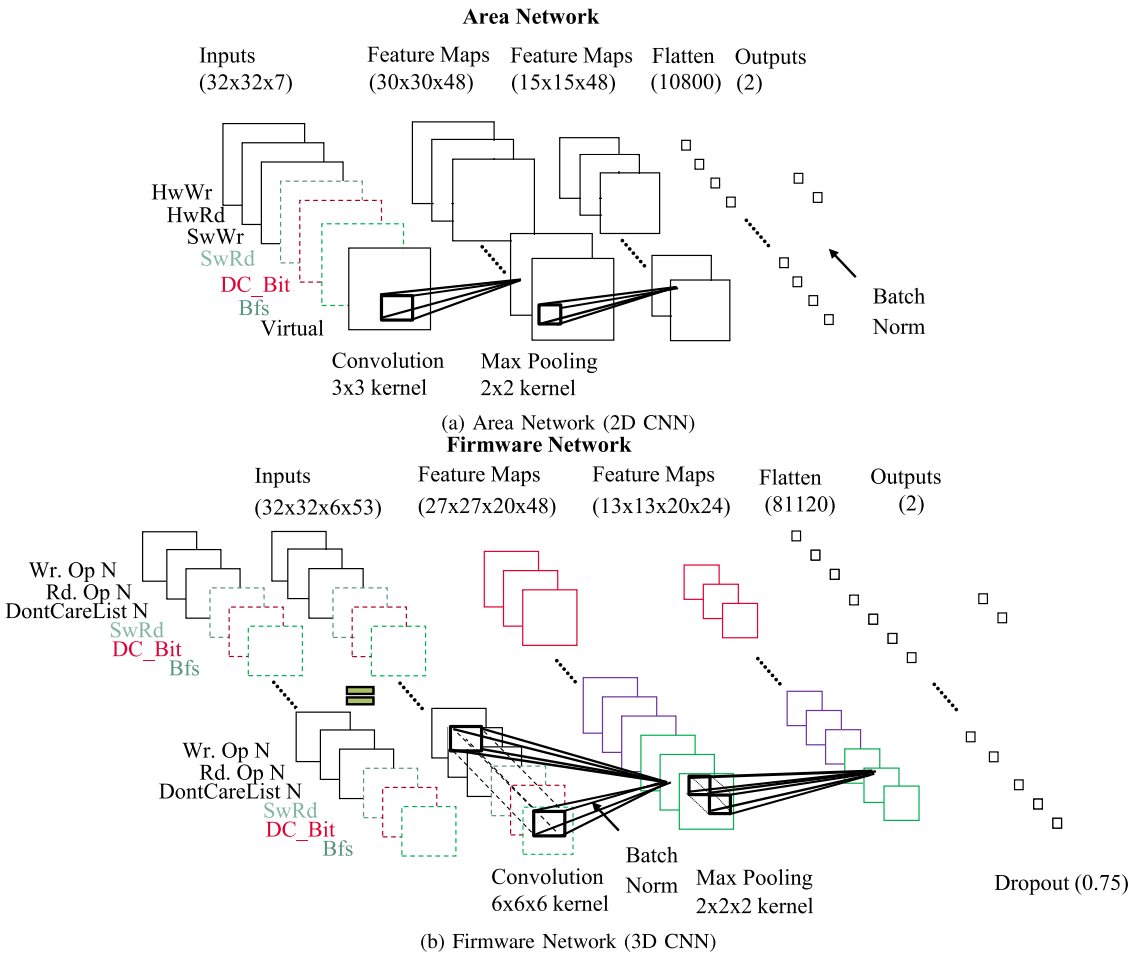


Fig. 5. Architectures of the used *Convolutional Neural Networks* (CNNs).

indicated as $\mathbf{y}_1, \ldots, \mathbf{y}_4$, from the *inputs*, which correspond to the values contained in the proposed data-structures. Accordingly, LUTs (represented by $\mathbf{y}_1$) and Slice Registers (represented by $\mathbf{y}_2$) of the RI are the outputs which indicate the HW Area. We call the prediction towards these outputs *Area Regression* (AR). The FW Cycles (represented by $\mathbf{y}_3$) and FW Size (represented by $\mathbf{y}_4$) instead are the selected FW Metrics. We call the corresponding forecast as *FW Metrics Regression* (FMR). AR and FMR are obtained, as mentioned before, through distinct ML models.

For both regressions, a set of training data $\{(\mathcal{X}^1, \mathcal{Y}^1), \ldots, (\mathcal{X}^N, \mathcal{Y}^N)\}$ with N being the size of the training set is used.

In the case of the AR, the 3D matrix of feature measurements is of size of $\mathcal{A}$, that is $\mathcal{X}^i_{AR} \in \mathbb{N}^{Q \times L \times B}$, whereas in the FMR, the 4D matrix is of size of $\mathcal{F}$, that is $\mathcal{X}^i_{FWR} \in \mathbb{N}^{Q \times L \times D \times P}$.

In the following paragraphs we describe the structure of the ML algorithms for AR (i.e., *Area Network*) and for the FMR (i.e., *Firmware Network*).

4.1.1 The Area Network

For the AR, we consider as an input the HW Images, and a 2D Convolutional Neural Network (CNN) [15]. We select a 2D CNN because of its capability to capture spatially related information. In our case, the HW images are processed along the Q and L dimensions of each feature representation. This allows us to accurately and easily elaborate spatial relations and, hence, to overcome the issues discussed in Section 2–yielding the desired accuracy for a cost estimation of memory systems. The architecture of the used 2D CNN is sketched in Fig. 5a.

4.1.2 The Firmware Network

With regards to the FMR, we consider the FW sequence as input, and a 3D CNN [33] for the estimation. This type of NN can take into account a further dimension of the inputs with respect to the 2D CNN. The FW sequence is processed through local regions of the dimensions Q, L as well as D, where the different feature representations are positioned. This is performed on each single frame $\mathcal{F}^p$ of FW sequence. Selecting this architecture, we can explore the spatial as well as the sequential information over the FW operations, for obtaining the FW cost estimation. Fig. 5b shows a sketch of the implemented 3D CNN's structure.

4.2 Optimization of the Utilized Networks

Once the ML algorithms and inputs are selected for the AR and FMR, further optimization potential exists in properly setting the non-trainable parameters of the network, i.e., the network *hyper-parameters*. This optimization step aims to select the best configuration possible for minimizing the Network estimation error and improve the accuracy for both the Area and the Firmware Network [6]. For doing so, the hyper-parameters optimization algorithm selects the

TABLE 1
Optimal Network's Hyper-Parameters

Networks Optimization				
Hyper-parameters	Area		FW Metrics	
	MLP	2D CNN	MLP	3D CNN
Initial LR	/	0.001	/	0.001
LR Scheduling	/	Exp D	/	Exp D
Dropout Rate	/	/	/	0.75
Batch Size	716	64	716	64
Optimizer	L-BFGS	Adam	L-BFGS	Adam
Activations	Relu			
No. of Filters	/	48	/	20
Filter size	/	3 x 3	/	6 x 6 x 6
No. of Layers	2	3	3	3

LR*: Learning Rate,* Const*: Constant Learning Rate,* Exp D*: Exponential Decay of the Learning Rate.*

best possible configuration out of the hyper-parameters space. We introduce an hyper-parameters space for the considered neural networks, where we define an interval from where the algorithm can select a value for any specific hyper-parameter (i.e., for the Firmware Network, we tried range of different *Filter sizes* e.g., 2 x 2 x 2, 3 x 3 x 3, etc., *No. of Layers* e.g., 1, 2, 3, etc. and other hyper-parameters). If none of the hyper-parameters optimized value tends to the extreme of interval, we hold the predifined interval, else we shift it.

As an hyper-parameters optimization algorithm, we adopt the *Tree-structured Parzen Estimator* (TPE) approach [4]. The TPE is a Bayesian model-based optimization that explores the hyper-parameters space in a non-trivial manner by running several possible configurations of the network and selecting the best performing hyper-parameters for our data. Moreover, through the TPE algorithm, we are able to integrate an ablation study in the network optimization algorithm. The ablation study process, also used in ML approaches such as [13] and [14], consists of removing parts from the network (e.g., a specific layer/group of layers) and evaluating the change in performance of the algorithm. Thanks to this procedure, we could reach at the same time high accuracy and a reduction of both networks to their essential components. This leads to an accurate estimation of HW Area/FW Metrics and yet reduces the complexity of the networks.

As a final result of the TPE approach, we can define an optimized architecture for the estimation of the area as well as for the firmware cost. A description of the hyper-parameters and technical details of the 2D/3D CNNs can be found in [15], [33]. The optimal hyper-parameters that we have found thanks to this process are shown in Table 1, as well as in Fig. 5a and in Fig. 5b.

In the following paragraphs, we analyze the structure of the optimized Area and Firmware Networks obtained using the TPE approach.

4.2.1 Optimized Area Network

As previously mentioned, for the AR we use a 2D CNN architecture, sketched in Fig. 5a. Here, each one of the seven channels of the HW Images is locally processed (convolution operation) by 48 different 3×3 kernels, which stride on the HW image and elaborate the spatial positions of properties in the registers. This operation takes place in the *2D Convolutional Layer* of the CNN. Out of this operation, 48 *Feature Maps* of dimensionality 30×30 are generated. These represent the results of the convolution between the HW image and the above mentioned kernels. Successively, a *Max Pooling Layer* performs pooling operations on the Feature Maps, selecting the maximum values out of local regions in the maps. This operation halves the width and height of the *Feature Maps*. Next, the maps are concatenated in a one dimensional vector (*Flatten Layer*) that is fully connected towards the two final outputs of the network. Before the final output, a *Batch Normalization* operation is performed. The dimensions of the layers and components of the network are sketched in Fig. 5a.

4.2.2 Optimized Firmware Network

The architecture of the 3D CNN used for the FMR is sketched in Fig. 5b. Here, the selected kernels have dimension $6 \times 6 \times 6$, and are able to process through a convolution operation all the six channels of the FW sequence at each frame $\mathcal{F}^p$ (this takes place in the *3D Convolutional Layer*). Elaborating the six channels of the FW sequence at each frame through the selected kernels allows to process the spatial configuration of the bitfield properties through time. This operation generates 20 *Feature Maps* of dimension $27 \times 27 \times 48$. The maps are successively taken as input to the *Max Pooling Layer*, which performs a cubic pooling operation, taking maximum values out of $2 \times 2 \times 2$ local regions. Similarly to the Area Network, a *Flatten Layer* concatenates the maps into a one dimensional vector. A *Dropout Layer* randomly masks out some of the neurons of the vector, with a certain dropout probability. The values out of the active neurons of the network will be propagated into a *Batch Norm Layer*, where a batch-dependent normalization is performed. Finally, the two outputs of the Firmware network are fully connected through a *Fully Connected Layer*. The Firmware Network dimensions are sketched in Fig. 5b. As pointed out in Table 1, we use *Adam* [20] as an optimizer and *ReLU* [25] activation functions for both 2D and 3D CNN.

Besides the issues discussed above, the choice of the kernel size plays a key role in the hyper-parameter optimization for both the 2D CNN for the AR and 3D CNN for the FWR. In fact this component is the core of the convolution operation, establishing how the spatial information in the data is taken into account and how to process the bitfield properties in the underlying data-structure (i.e., HW Images or FW Sequence). The size of the kernels is therefore a key hyper-parameter in the realization of both proposed networks. Typically, kernels used in computer vision have equal dimensions along the height and the width of the image (e.g., 3×3 squared kernel, $6 \times 6 \times 6$ cubic kernel) [15]. These proportions are intended for the kernel to respond equally on variations along different dimensions of the input signal (e.g., image). For the FMR nevertheless, this property is not strictly needed. In fact, in the structure of the firmware program, the writing and reading operations on bitfields take into account a single register at a time. This means, configurations of other registers apart from the accessed one (e.g., for writing/reading a bitfield), do not influence the single access operation, and thus the firmware metrics. Therefore, by using a non-cubic $1 \times 6 \times 6$ kernel, we take into account only the bitfield

TABLE 2
Kernel Performance Comparison

Opt FW NNs	K Dim	No. Par	Avg RMSE	Avg R̂2 Sc
1.	6 x 6 x 6	**188,222**	**88.1**	**0.942**
2.	1 x 6 x 6	**45,037**	**90.9**	**0.937**

Opt FW NNs*: Optimized Firmware Networks,* K Dim*: Kernel Dimension,* No. Par*: Amount of trainable parameters in the Network,* Avg RMSE*: provides the average of the* Root of Mean Squared Error *as loss for the Firmware Network,* Avg R̂2 Sc*: provides the average of the explained variance (R̂2 Score).*

properties of one single register at a time. As shown in the Table 2, an optimized network provided with the selected non-cubic kernels and 5 *Feature Maps* avails itself of a reduced number (decrease by factor 4x) of parameters w.r.t. to the best performing network architecture. Nonetheless, its results are only slightly lower in terms of accuracy. An important remark on that is that a decrement in the number of parameters can be related to lower power consumption, higher speed and efficiency for the training and inference phase of the network [16], [22].

5 Evaluation

The proposed data-structures representing hardware and firmware features (i.e. the HW image and FW sequence) have been realized and applied to the proposed implementations of the ML algorithms as described above. Afterwards, we evaluated the corresponding results, i.e. the obtained numbers using the ML approach inspired by CV has been compared to the results obtained by the state-of-the-art cost estimation. This latter method is currently used within our industrial environment at Infineon Technologies AG [35]. In this section, we summarize the results as well as the drawn conclusions. To this end, we first provide some more details about the correspondingly used environment as well as the considered metrics (which follow basic evaluation schemes from the ML domain). Afterwards, the obtained results are presented and conclusions are drawn.

5.1 Used Learning Environment

Our evaluations have been performed with the following system configurations: as software development environment, we chose Python v3.6, Tensorflow-GPU v1.0.1, and Hyperopt. For optimizing the algorithm on the Nvidia GPU, we used the CUDA Toolkit 9.0 and cuDNN v7.0. With respect to hardware, we considered a Nvidia Tesla P100 for training the ML algorithms, an Intel Core i7-8700K CPU, and DIMM 16 GB DDR4-3000 module of RAM. For evaluating LUTs and SRs, we retrieved reports from the Vivado Synthesis on an Arty-7 FPGA board from Xilinx,[6] while for measurement of the FCs we used a RISC-V CPU implementing 32 bit Base Integer Instruction Set architecture [1].

For the purpose of evaluating the proposed approach, we considered a large dataset of valid design configurations for memory systems, as described in details in [21]. We first uniformly sample the amount of considered bitfields in each design configuration within a range, such the *No. of Bitfields* $\in [25, 50]$. This constitutes a predefined design space taken from design experience. Then, we uniformly sample the attributes of each bitfield including *Size* and properties (i.e., *HwWr*, *HwRd*, etc.). Last, we dispose the bitfields in a random order and we insert them in a fitting *No. of Units*. With this procedure we are able to sample out of different *Bitfields Configurations*. The dataset contains variations over spatial distribution of bitfields–similar to the ones shown in Fig. 3. In total, the complete dataset is composed of 1,024 generated design configurations. We chose this amount after observing that the generation of further samples would not improve the accuracy of the tested ML algorithms.

After generating the design configurations, we retrieved the objectives measurements for each one of the design instances. We first synthesized the instances for obtaining the number of LUTs and SRs of each design. After that, we ran on each one of the memory system a firmware program, where the operations of the program are specified in the design configuration. From this, we eventually retrieved the Firmware Size and Firmware Cycles.

After obtaining the dataset, we divided this into a training set (70 percent of the considered design configurations) used for the training phase of the ML algorithms and a test set (20 percent of the considered design configurations) with which we evaluated the trained algorithms. Besides that, also a validation set (10 percent of the considered design configurations) has been used for fine-tuning the hyper-parameters of the ML algorithms. Using most of the data for training (e.g., 60-80 percent) is a common practice in CV. We did not consider different data split proportions after reporting an accuracy difference $<1\%$ between training and test data. We can therefore conclude that the proposed method does not overfit training data and is able to generalize to configurations that it has not seen yet. In fact, we report high performance also on the validation and the test set, consistently with the guidelines pointed out in [15].

Following the representations introduced in Section 3, the proposed approach utilizes $\mathcal{X}^i_{AR} \in \mathbb{N}^{32\times32\times7}$ feature values to represent HW Images for *Area Regression* and $\mathcal{X}^i_{FMR} \in \mathbb{N}^{32\times32\times6\times53}$ feature values to represent FW sequences for *FW Metrics Regression*. In contrast, the state-of-the-art approach used thus far only aggregates feature values in form of vectors of dimensions 1×9 as an input for the AR and 1×6 as an input for the FMR. We implement the state-of-the-art approach as a *Multi Layer Perceptron* (MLP, cf. [15]), which is a fully connected network, i.e., non-convolutional, for processing aggregated values. We optimize this architecture through the TPE algorithm. In the case of the AR, the MLP is composed by $1st$ *Layer (11 Neurons)* - $2nd$ *Layer (4 Neurons)* while for the FMR, it is composed by $1st$ *Layer (15 Neurons)* - $2nd$ *Layer (6 Neurons)*. For both MLPs, we use an L-BGFS optimizer, with RELU activation functions and L2 Weight Regularizations [15]. Hyper-parameters choices for the MLPs used are enlisted in Table 1.

5.2 Obtained Results

Table 3 summarizes the respectively obtained results. Here, the top of the table provides the mean values and the range of the design configurations within the considered dataset with respect to LUTs (represented by $\mathbf{y}_1$), SRs (represented

6. Documentation to be found on the website: https://www.xilinx.com/products/boards-and-kits/arty.html

TABLE 3
Dataset and Results

Dataset Mean	$\mu_{\mathbf{y}_1} = 456$ $\mu_{\mathbf{y}_2} = 272$	$\mu_{\mathbf{y}_3} = 1238$ $\mu_{\mathbf{y}_4} = 3516$ b
Dataset Range	$y_1 \in [228, 726]$ $y_2 \in [106, 408]$	$y_3 \in [724, 1656]$ $y_4 \in [828, 5758]$ b

Inputs	HW Images FW Sequences	(1024x32x32x7) (1024x32x32x6x53)
Outputs	HW Images FW Sequences	$\{\mathbf{y}_1, \mathbf{y}_2\}$ $\{\mathbf{y}_3, \mathbf{y}_4\}$

	Area		**Firmware Metrics**	
	S-o-t-a	**Proposed**	**S-o-t-a**	**Proposed**
RMSE	128 $E_{\mathbf{y}_1}$ 67 $E_{\mathbf{y}_2}$	**32** $E_{\mathbf{y}_1}$ **14** $E_{\mathbf{y}_2}$	272 $E_{\mathbf{y}_3}$ 261 b $E_{\mathbf{y}_4}$	**80** $E_{\mathbf{y}_3}$ **96** b $E_{\mathbf{y}_4}$
R^2 Score	0.72 $R^2_{\mathbf{y}_1}$ 0.76 $R^2_{\mathbf{y}_2}$	**0.93** $R^2_{\mathbf{y}_1}$ **0.95** $R^2_{\mathbf{y}_2}$	0.83 $R^2_{\mathbf{y}_3}$ 0.81 $R^2_{\mathbf{y}_4}$	**0.95** $R^2_{\mathbf{y}_3}$ **0.93** $R^2_{\mathbf{y}_4}$

RMSE *provides the* Root of Mean Squared Error *(denoted as* $E_{\mathbf{y}_k}$*) and the* R^2 Score *provides the explained variance (denoted as* $R^2_{\mathbf{y}_k}$*) with respect to the number of LUTs (*$\mathbf{y}_1$*), SRs (*$\mathbf{y}_2$*), FCs (*$\mathbf{y}_3$*), and FS (*$\mathbf{y}_4$*). Here,* $\mathbf{y}_4$ *is indicated in terms of bits (b).*

by $\mathbf{y}_2$), Firmware Cycles (represented by $\mathbf{y}_3$), and Firmware Size (represented by $\mathbf{y}_4$). Additionally, we provide a brief summary of *Inputs* and *Outputs* of our method, together with their dimensionality. Afterwards, the corresponding results are provided for area and firmware when the currently used state-of-the-art method is applied (denoted by *S-o-t-a*) and when the proposed method is applied (denoted by *Proposed*). As results, we provide the error of the respectively estimated values compared to the real values. More precisely, the row *RMSE* indicates the *Root of Mean Squared Error* (RMSE, [15]) on the estimates to the real values (in the form of $E_{\mathbf{y}_k}$). Besides that, the R^2 *Score* provides the explained variance [15] of the estimations and is accordingly indicated as $R^2_{\mathbf{y}_k}$. Recall, both errors are provided for LUTs ($\mathbf{y}_1$), SRs ($\mathbf{y}_2$), FCs ($\mathbf{y}_3$), and FS ($\mathbf{y}_4$).

For example, Table 3 shows that the currently used state-of-the-art method yields LUTs estimates with a mean error of $E_{\mathbf{y}_1} = 128$ compared to the real LUTs values. Additionally taking the variance in the LUTs values of the considered dataset into account, we get $R^2_{\mathbf{y}_1} = 0.72$, i.e. the accuracy of this method is off by 28 percent on average.

The results clearly show the significantly improved accuracy of the proposed method compared to the state-of-the-art used so far. The mean errors are significantly reduced. The factor of reduction of the mean error varies from a maximum of approx. 5 for the $E_{\mathbf{y}_2}$ to a minimum of approx. 2.5 for the $E_{\mathbf{y}_4}$. Much more important, however, is how close the considered approaches estimate the actual values: While the accuracy of the state-of-the-art approach frequently is off by more than 20 percent for area estimation and more than 15 percent for firmware estimation, the elaborated consideration of spatial information of the proposed method comes rather close to the actual values (just 5-7 percent off for both area and firmware). A visualization of the error distribution is presented in Fig. 6. Here, we show the predictions of our method on unseen (i.e., test) data versus the ground truth cost retrieved with Xilinx Vivado synthesis. These results refer to the objective y_1, on which our model performs at its worst among the 4 objectives (error standard deviation of 20.4, pointed out by the red lines), as shown in Table 3. One can see how every data-point is close to the diagonal, i.e., very accurately predicted. The confidence interval between the two lines created with the standard

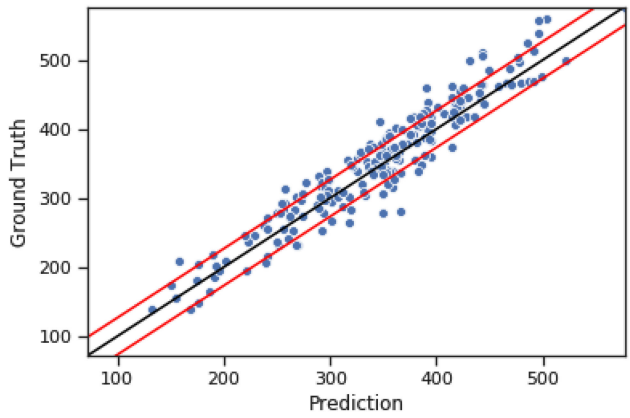


Fig. 6. LUTs (y_1) prediction vs. ground truth.

deviation of the errors is very narrow with just a few points outside of it. This shows the robustness of the proposed method everywhere and so, not overfitting a particular subset of configurations. Furthermore, the method proposed in this work offers higher flexibility (e.g., could easily support additional features such as further types of writing operations, conditional logic, etc.) since they are directly and automatically learned in the training process from easy-to-implement feature representations. At the same time, our method is more scalable when compared to the state-of-the-art since in CNNs the parameters depend less on the input size than in fully connected networks such as MLP, as shown in [15]. Therefore processing bigger inputs, i.e., RI with more units/bitfields, will lead to a much minor increase in the computational effort. The state-of-the-art is inferior when it comes to automatic learning since it requires still manual effort in the pre-process of data aggregation and feature representation. Our model also provides a solution to this last point.

In order to compare our proposed 3D CNN for FMR to other popular neural networks architectures for sequential information processing, we benchmarked it against an LSTM model, first proposed in [17], and against an LSTM model with attention, initially proposed in [2]. Table 4 shows the performance comparison of different types of models for *R^2 Score*, *Training Time*, and number of *Trainable Parameters*, which relates to (but do not exclusively define) the complexity of the model. We first report, for the sake of completeness, the 2D version of the CNN (i.e., *Proposed Method* for AR). To benchmark our 3D CNN model for FMR, we build an LSTM Network with two LSTM layers and two Fully Connected layers. Additionally, we implemented an LSTM with Attention with only one LSTM layer, one Attention layer, and two

TABLE 4
Performance Comparison

	R^2 Score	Tr. Time	Tr. Params
2D CNN (Proposed AR)	0.9407	129.25 s	24,640
3D CNN (Proposed FMR)	0.942	747.65 s	188,222
LSTM (FMR)	0.8654	587.02 s	1,629,106
LSTM + Attention (FMR)	0.8306	1023.28 s	847,888

Tr. Time *provides the training time of the benchmarked neural networks, while* Tr. Params *provides the* No. of Trainable Parameters *of the considered neural networks models.*

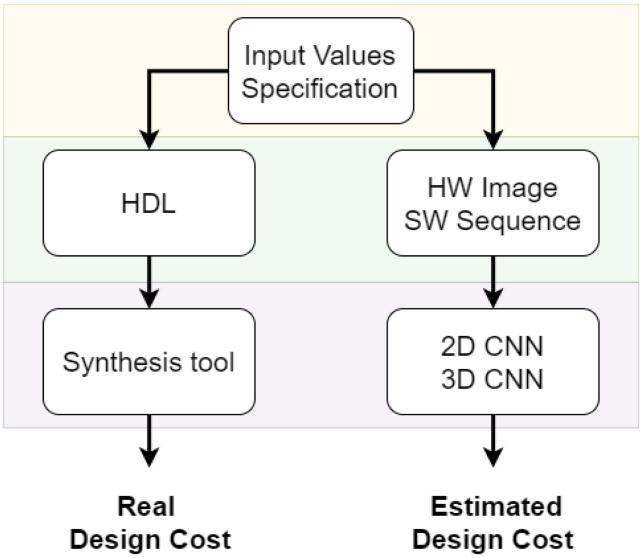


Fig. 7. Flow of the proposed method (right branch) compared to the current industrial flow (left branch).

Fully Connected layers. The 3D CNN has a minor number of parameters to the LSTM model and LSTM with Attention model but has a better accuracy than both.

6 Conclusion and Future Works

In this work, we proposed a cost estimation method for memory systems that explicitly takes spatial information into account in order to derive much more accurate values. To this end, we observed that problems from the domain of *Computer Vision*–in particular age determination of persons depicted in images–are rather similar to cost estimation of memory systems. Accordingly, we re-adapted corresponding solutions based on *Machine Learning* which have been found suitable for computer vision problems for the purpose of cost estimation. The overall flow of the proposed method is sketched in Fig. 7 together with the one currently used in our industrial environment. Experimental evaluations within an industrial context showed that, while the accuracy of the state-of-the-art approach is far off from the actual values, the method proposed in this work comes rather close to them. We believe that learning the parameters directly on the data is the key to make our method suitable for similar estimation tasks. Moreover, we used well establish tools to conduct an extensive study on how to choose the optimal hyper-parameters and we finally chose the configuration that achieved the highest performance. Nevertheless, we have shown that using a non-cubic kernel $1 \times 6 \times 6$ for the FMR reported quasi-optimal performance while reducing considerably the computational effort needed. Furthermore, the proposed approach can easily be extended by further features. Future work will focus on exploring this flexibility as well as the the resulting complexity–approaching a hardware/software trade-off analysis. At the same time, in order to develop a ML model that is measuring uncertainty in the design cost prediction, we are exploring Bayesian Machine Learning [24]. In this way, important information about the confidence of the model on the prediction can be shared with the designer, increasing his/her understanding of the model cost estimation.

Acknowledgments

This work was partially supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

References

[1] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," Univ. California, Berkeley, Berkeley, CA, Tech. Rep. UCB/EECS-2014–146, Aug. 2014.
[2] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proc. Int. Conf. Learn. Representations*, 2015.
[3] D. H. Ballard and C. M. Brown, *Computer Vision*, 1st ed., Upper Saddle River, NJ, USA: Prentice Hall, 1982.
[4] J. Bergstra and R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Proc. 24th Int. Conf. Neural Inf. Process. Syst.*, 2011, pp. 2546–2554.
[5] R. Cipolla *et al.*, *Machine Learning for Computer Vision*, vol. 5, Berlin, Germany: Springer, 2013.
[6] M. Claesen and B. De Moor, "Hyperparameter search in machine learning," in *Proc. 11th Metaheuristics Int. Conf.*, 2015.
[7] K. Devarajegowda, J. Schreiner, R. Findenig, and W. Ecker, "Python based framework for HDSLs with an underlying formal semantics: (invited paper)," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2017, pp. 1019–1025.
[8] T. Dobhal, V. Shitole, G. Thomas, and G. Navada, "Human activity recognition using binary motion image and deep learning," *Procedia Comput. Sci.*, vol. 58, pp. 178–185, 2015.
[9] S. H. M. Durand and V. Bonato, "A tool to support Bluespec SystemVerilog coding based on UML diagrams," in *Proc. 38th Annu. Conf. IEEE Ind. Electron. Soc.*, Oct. 2012, pp. 4670–4675.
[10] W. Ecker, W. Mueller, and R. Doemer, *Hardware-Dependent Software: Principles and Practice*, 1st ed., Berlin, Germany: Springer, 2009.
[11] W. Ecker and J. Schreiner, "Metamodeling and code generation in the hardware/software interface domain," in *Handbook of Hardware/Software Codesign*, Berlin, Germany: Springer, Nov. 2017, pp. 1051–1091.
[12] T. Farkas, C. Neumann, and A. Hinnerichs, "An integrative approach for embedded software design with UML and simulink," in *Proc. 33rd Annu. IEEE Int. Comput. Softw. Appl. Conf.*, 2009, vol. 2, pp. 516–521.
[13] C. Fawcett and H. H. Hoos, "Analysing differences between algorithm configurations through ablation," *J. Heuristics*, vol. 22, no. 4, pp. 431–458, 2016.
[14] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2014, pp. 580–587.
[15] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org
[16] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proc. 28th Int. Conf. Neural Inf. Process. Syst.*, 2015, pp. 1135–1143.
[17] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.
[18] H. Hsiao and J. H. Anderson, "Sensei: An area-reduction advisor for FPGA high-level synthesis," in *Proc. Des. Autom. Test Europe Conf. Exhib.*, 2018, pp. 25–30.
[19] Y. Kim, Y. Jernite, D. Sontag, and A. M. Rush, "Character-aware neural language models," in *Proc. 30th AAAI Conf. Artif. Intell.*, 2016, pp. 2741–2749.
[20] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. Int. Conf. Learn. Representations*, 2015.
[21] K. Devarajegowda, M. Manzinger, W. Ecker, L. Servadei, E. Zennaro, and R. Wille, "Accurate cost estimation of memory systems inspired by machine learning for computer vision," in *Proc. Des. Autom. Test Europe Conf. Exhib.*, 2018, pp. 1277–1280.
[22] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural networks with few multiplications," in *Proc. Int. Conf. Learn. Representations*, 2016.
[23] P. Meng, A. Althoff, Q. Gautier, and R. Kastner, "Adaptive threshold non-pareto elimination: Re-thinking machine learning for system level design space exploration on FPGAs," in *Proc. Conf. Des. Autom. Test Europe*, 2016, pp. 918–923.

[24] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. Cambridge, MA, USA: MIT Press, 2012.
[25] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proc. 27th Int. Conf. Mach. Learn.*, 2010, pp. 807–814.
[26] R. Nane *et al.*, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016.
[27] G. Nicolescu and P. J. Mosterman, *Model-Based Design for Embedded Systems*. Boca Raton, FL, USA: CRC Press, 2009.
[28] M. Nixon and A. S. Aguado, *Feature Extraction & Image Processing for Computer Vision*., 3rd ed., Orlando, FL, USA: Academic Press, Inc., 2012.
[29] A. Powell, C. Savvas-Bouganis, and P. Y. K. Cheung, "High-level power and performance estimation of FPGA-based soft processors and its application to design space exploration," *J. Syst. Archit.*, vol. 59, no. 10, pp. 1144–1156, Nov. 2013.
[30] J. Schreiner and W. Ecker, "Digital hardware design based on metamodels and model transformations," in *Proc. 24th IFIP WG 10.5/IEEE Int. Conf. Very Large Scale Integr.*, 2016, pp. 83–107.
[31] L. Servadei, E. Zennaro, T. Fritz, K. Devarajegowda, W. Ecker, and R. Wille, "Using machine learning for predicting area and firmware metrics of hardware designs from abstract specifications," *Microprocessors Microsystems*, vol. 71, 2019, Art. no. 102853.
[32] F.-J. Streit, M. Letras, M. Schid, J. Falk, S. Wildermann, and J. Teich, "High-level synthesis for hardware/software co-design of distributed smart camera systems," in *Proc. 11th Int. Conf. Distrib. Smart Cameras*, 2017, pp. 174–179.
[33] S. Ji, W. Xu, M. Yang, and K. Yu, "3D convolutional neural networks for human action recognition," in *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 1, pp. 221–231, 2012.
[34] D. Yi, Z. Lei, and S. Z. Li, "Age estimation by multi-scale convolutional network," in *Proc. Asian Conf. Comput. Vis.*, 2014, pp. 144–158.
[35] E. Zennaro, L. Servadei, K. Devarajegowda, and W. Ecker, "A machine learning approach for area prediction of hardware designs from abstract specifications," in *Proc. 21st Euromicro Conf. Digital Syst. Des.*, 2018, pp. 413–420.
[36] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," in *Adv. Neural Inf. Process. Syst.*, pp. 649–657, 2015.

Lorenzo Servadei is working toward the PhD degree at Infineon Technologies AG, in collaboration with the Johannes Kepler University Linz, Linz, Austria. His research interest includes hardware optimization with machine learning. He is currently lecturing on machine learning with the Technical University of Munich, Munich, Germany.

Edoardo Mosca is working toward the master's degree in mathematics in Data Science, Technical University of Munich, Germany. Currently he is working at Infineon Technologies AG in Munich as machine learning researcher and teaches machine learning at the Technical University of Munich, Munich, Germany.

Elena Zennaro received the bachelor's degree in information engineering and the master's degree in automation engineering from the University of Padova, Italy, respectively, in 2015 and 2017. She is currently working at Infineon Technologies AG in Munich, as machine learning engineer in the Design Department.

Keerthikumara Dewarajegowda received the master's degree from the University of Kaiserslautern, Kaiserslautern, Germany, in 2016, where he is currently working toward the PhD degree. His research interests include formal verification methods for modern digital designs and design automation methods.

Michael Werner received the MSc degree in electrical engineering and information technology from the Technical University of Munich, Germany, in 2017. He is currently working toward the PhD research scholar at Infineon Technologies AG in cooperation with the Technical University of Munich, Munich, Germany. His research interest includes model-driven Firmware Development.

Wolfgang Ecker (Member, IEEE) is senior principal engineer at Infineon and a professor at Technical University of Munich, Munich, Germany. He is (co-)author of more than 200 papers on modelling and design automation, received five best paper awards, was granted with the German EDA Achievement Award. He is a member of Acatech, the German Academy of Science and Engineering. He leads the Infineon Deep Learning internal think tank. In addition, he is member of the AI commission of inquiry of the German Government.

Robert Wille (Senior Member, IEEE) is a full professor with the Johannes Kepler University Linz, Linz, Austria. His research interests include the design automation for conventional and emerging technologies. In this field, he has coauthored more than 300 papers, got frequently awarded, and served the community various capacities.

AxMAP: Making Approximate Adders Aware of Input Patterns

Morteza Rezaalipour, Mohammad Rezaalipour, Masoud Dehyadegari, and Mahdi Nazm Bojnordi

Abstract—Making approximate computing specific to user requirements is crucial to system performance, energy-efficiency, and reliability. However, developing hardware for such optimization becomes a significant challenge due to the high cost of examining all potential choices while exploring a large design space. One determinant aspect of exploring a design space is the efficiency of evaluating error metrics, such as the Mean Error Distance (MED) and the Error Probability (EP), for each possible choice within the search space. Since computing these error-metrics is quite time-consuming, efficient calculation approaches are essential. This article proposes a novel formal approach to accurately compute the EP and MED of approximate adders for any input pattern at a linear time and space complexity. Our experimental results indicate that the proposed approach can accurately compute the error-metrics of large approximate adders at a 150 times faster speed compared to the Monte Carlo sampling methods. We then develop AxMAP, a design tool based on the proposed error-metrics computation that generates energy-efficient approximate adders for any given input pattern. When applied to image processing applications, AxMAP produces more than 150 different designs for adders that achieve superior performance and energy-efficiency compared to the existing state-of-the-art approximate adders.

Index Terms—Approximate computing, adders, mean error distance, error probability, circuit synthesis

1 Introduction

Approximate computing trades accuracy for power, area, and delay of modern-day applications such as machine learning and image processing [1]. Digital adders are the key component of a wide range of error-resilient applications. In general, Approximate adders have become the main focus of numerous recent [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]. Approximate adders have been classified to *Low Power Approximate Adder* (LPAA) [2], [5], [7], [8], [10], [11], [13], and *Low Latency Approximate Adder* (LLAA) [3], [4], [6], [9], [12].

Prior work has extensively used various error-metrics such as the Mean Error Distance (MED) and Error Probability (EP) to assess the error characteristics of approximate adders [14], [15], [16], [17]. Computing these metrics is quite time consuming as they often require obtaining the results of every possible input pattern, which is 2^{2n} instances for an n-bit adder. The time complexity of computing MED and EP for an n-bit adder is of the order of $O(2^{2n})$. However, an efficient and accurate computation of error-metrics is necessary to develop design automation tools that can synthesize approximate adders specific to user requirements [18], [19].

- *M. Rezaalipour and M. Dehyadegari are with the K. N. Toosi University of Technology, Tehran 19967, Iran. E-mail: mrezaalipour@email.kntu.ac.ir, dehyadegari@kntu.ac.ir.*
- *M. Rezaalipour is also with the Software Institute, Faculty of Informatics, Università della Svizzera italiana (USI), 6900 Lugano, Switzerland. E-mail: mohammad.rezaalipour@usi.ch.*
- *M.N. Bojnordi is with the School of Computing, University of Utah, Salt Lake City, UT 84112. E-mail: bojnordi@cs.utah.edu.*

Manuscript received 8 Mar. 2019; revised 6 Jan. 2020; accepted 15 Jan. 2020. Date of publication 23 Jan. 2020; date of current version 8 May 2020. (Corresponding author: Masoud Dehyadegari.) Recommended for acceptance by L. A. Sousa. Digital Object Identifier no. 10.1109/TC.2020.2968905

For instance Tajasob *et al.* [2] propose a vast design space with billions of billion approximate adders each having specific circuit and error characteristics. Exploring such huge space to find matches to users needs requires efficient methods of computing error-metrics.

Several analytical methods have been presented in the literature to compute error-metrics [1], [15], [20], [21], [22], [23]. Although they have been successful in addressing this problem partially, there are limitations for each. Some studies, such as the work of Liu *et al.* [15], avoid exhaustive and time consuming calculation of error metrics to some extent by employing Monte Carlo sampling methods. However, the results estimated by Monte Carlo methods are inexact. Moreover, these methods are time-consuming and largely impractical as the number of samples increases. Mazahir *et al.* [1] also mention limitations of Monte Carlo simulations for computing error characteristics of approximate adders.

Li and Zhou [23] propose a framework that computes MEDs of LLAAs accurately while the results produced for EPs are approximated. Their framework only works with uniform input patterns. The method presented by Liu *et al.* [15] estimates MED for different input patterns. However, this framework does not compute EP and employs different approaches for different types of adders. Mazahir *et al.* [1] propose to address the issue with a more general method for computing EPs and error distributions of LLAAs. It also employs error distributions to compute other metrics such as MED. This method works with different input patterns; however, it only produces the results for uniform input patterns accurately. Wu *et al.* [21] propose a method to compute EPs and MEDs of LLAAs which faster than the work by Mazahir *et al.* [1]. Both methods provide estimates of error-metrics for non-uniform input patterns.

It is worth stating that none of the studies mentioned above can calculate MED and EP for LPAAs. Ayub *et al.* [22] propose a method to obtain EPs of LPAAs for different input patterns. This method does not compute MED. Another method presented by Roy *et al.* [20] only computes MEDs of LPAAs, and does not work with different input patterns. Also, the complexity of the method presented in [20] is of the order of $O(2^m)$, where m is the length of the approximate portion.

In this paper, we present a novel formal approach that accurately and efficiently computes EPs and MEDs for LPAAs, for different input patterns, in linear time and space complexities. This approach employs a model so-called the *quad-tree representation* [2] to provide generic MED and EP formulae that can be instantiated for any LPAA. The experimental results demonstrate the validity of our approach and indicate that it can compute MEDs, and EPs of large approximate adders more than 150 times faster than Monte Carlo sampling methods. We develop a tool called AxMAP that employs the proposed approach and automatically generate approximate adders. By exploring the design space, AxMAP finds a suitable adder for given input patterns, under specified circuit and error characteristics. Using this tool, more than 150 adders have been produced for an image processing problem each providing better trade-offs among power, area, delay, and MED compared to the existing state-of-the-art energy-efficient approximate adders. To show the generality of the results, we evaluate the generated adders over a wider range of input images.

2 Background and Observations

2.1 Preliminaries

The proposed approach in this paper is based on two primary concepts: fast and accurate computation of error-metrics and design space exploration using quad-tree representation [2]. This section provides an overview of these concepts.

2.1.1 Error-Metrics

Liang *et al.* [16], [17] propose *Error Distance* (ED) as a metric to characterize the reliability of adders for a specific input pair. For a given n-bit approximate adder and two binary inputs $a = (a_{n-1}a_{n-2}\dots a_0)$ and $b = (b_{n-1}b_{n-2}\dots b_0)$, the error distance is defined as $ED_j = \hat{R}_j - R_j$, where R_j is the correct result (i.e, the result of an accurate adder for a and b), $\hat{R}_j$ is the value that the given approximate adder produces, and j is the index of the current input pair, which we compute as $j = (a_0b_0a_1b_1\dots a_{n-2}b_{n-2}a_{n-1}b_{n-1})_{10}$.

Based on the metric ED, two other well-known error-metrics are defined which are the *Mean Error Distance* and the *Error probability* [16]. EP represents the ratio of the number of input pairs whose ED is not zero, to the total number of input pairs (i.e., 2^{2n} for an n-bit adder). MED is the mean value of all EDs produced for an n-bit approximate adder, and it is shown in Eq. (1). The term Q_j represents the probability of input pair j, and for the uniform input pattern, since input pairs occur with the same probability (i.e., $1/2^{2n}$), Q_j can be replaced by $1/2^{2n}$.

$$MED = \sum_{j=0}^{2^{2n-1}} |ED_j| Q_j. \tag{1}$$

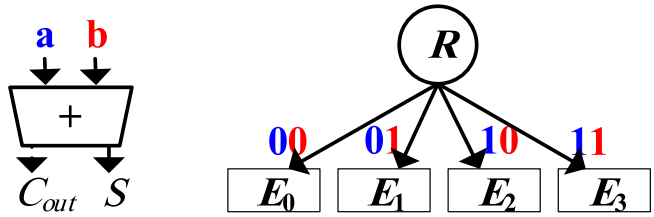


Fig. 1. Quad-tree representation of disjoint single-bit building blocks.

2.1.2 Quad-Tree Representation

The so-called quad-tree representation [2] is a generic form based on ED to model building blocks of a family of LPAAs known as disjoint approximate adders. The quad-tree representation has been employed to produce energy-efficient adders that demonstrate better trade-offs between circuit and error characteristics compared to the state-of-the-art adders. An n-bit disjoint approximate adder comprises a sequence of k-bit approximate sub-adders (i.e., disjoint building blocks) each of which produces a subset of the output bit positions. The disjoint building blocks in a sequence do not pass or receive carry signals, except for the last building block which produces the carry-out signal (C_{out}) of the final result. The state-of-the-art Lower-Part-OR adder (LOA) [13] is a disjoint approximate adder which is designed based on disjoint single-bit building blocks (i.e., $k = 1$).

Fig. 1 illustrates the quad-tree representation of disjoint single-bit building blocks. In this model, the node at level 0 is the root of the tree, which is labeled as R in Fig. 1, and each node at the last level (e.g., the node labeled as E_0) is called *leaf*. A path starting from the root to a leaf demonstrates a specific input pair. Since disjoint single-bit building blocks do not receive carry-in signals (C_{in}), there are only four possible input pairs to them. Thus, there are four edges in their quad-tree representation, each having its corresponding input pair written on it in Fig. 1. The parameter E_i ($0 \le i \le 3$) in each leaf, represents the ED computed for the input pair of the path to that leaf.

Since the last building block of each disjoint approximate adder produces a carry-out signal, it has a 2-bit wide output comprising the sum signal (S) and C_{out}. Thus, there are four possible values for each E_i depending on the exact output for its corresponding input pair. For instance, for input pair $(a, b = 0, 0)$, E_0 can be either of the values 0, +1, +2, and +3. As a result, there are 4^4 possible sequences of leaves, each of which represents a disjoint single-bit building block with the carry-out signal, regardless of hardware implementation.

The output of building blocks that do not have carry-out signals are 1-bit wide comprising only the sum signal (i.e., S). As a result, for these building blocks, there are only two possible values for each E_i. For example, for input pair $(a, b = 1, 1)$, E_3 can be either -1 or -2, which indicates the existence of 2^4 possible disjoint single-bit building blocks of this type.

The quad-tree representation is capable of modeling disjoint multi-bit building blocks as well (i.e., $k > 1$). However, due to space limitations, it is not reviewed here, and for further details, interested readers are directed to the article prepared by Tajasob *et al.* [2].

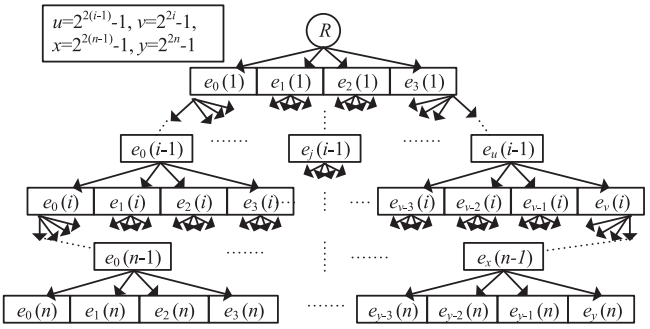


Fig. 2. Quad-tree representation of an n-bit disjoint approximate adder comprising single-bit building blocks.

2.2 Observations

2.2.1 First Observation: MED Calculation

The primary purpose of the quad-tree representation is to model disjoint building blocks based on their EDs. However, after a thorough examination of the quad-tree representation, we realized that it could also represent disjoint approximate adders as a framework for analytical computation of error-metrics for approximate adders.

Fig. 2 illustrates the quad-tree representation of an n-bit adder comprising disjoint single-bit building blocks. In this model, i ($i < n$) refers to the levels of the tree, and in each level, j ($j < 2^{2i}$) is the index of nodes. Considering $a = (a_{n-1}a_{n-2}\dots a_0)$ and $b = (b_{n-1}b_{n-2}\dots b_0)$ as the binary input numbers to the adder, j is computed as $j = (a_0b_0a_1b_1\dots a_{n-2}b_{n-2}a_{n-1}b_{n-1})_{10}$. The depth of this quad-tree is equal to the length of the adder, which is n. Node j at level i, referred to as $e_j(i)$, contains the ED of an i-bit sub-adder comprising the first i least significant bits (i.e., from bit-position 0 to i-1). Thus, for adders comprising disjoint single-bit building blocks, $e_0(1)$ to $e_3(1)$ are equal to E_0 to E_3. An edge starting from a parent node at level $i-1$ to one of its children at level i represents a_ib_i, the ith bits of the two binary input numbers a and b.

Based on the presented model in Fig. 2, each leaf at level n contains the ED value for one of the possible input pairs to the n-bit adder. As a result, for uniformly distributed input patterns, the MED of the family of adders represented in this figure can be computed by Eq. (2).

$$MED(n) = \frac{|e_0(n)| + |e_1(n)| + \ldots + |e_{2^{2n}-1}(n)|}{2^{2n}}. \tag{2}$$

Since there are 2^{2n} addition operations in Eq. (2), its time complexity is of the order of $O(2^{2n})$, which needs to be reduced. On the other hand, considering the characteristics of the quad-tree in Fig. 2, the value in each four siblings at level i can be expressed based the value of their parents at level $i-1$, using the following equation:

$$e_{4j+l}(i) = |e_j(i-1) + 2^{i-1} \times E_l|, \tag{3}$$

where $l \in \{0,1,2,3\}$, the parameter j represents the indices of leaves at level $i-1$, and $0 \le j \le 2^{2(i-1)} - 1$. Employing Eq. (3), we can restate Eq. (2) as the following:

$$\begin{aligned} &MED(n) = \\ &\frac{1}{2^{2n}} \times (|e_0(n-1) + 2^{n-1} \times E_0| + |e_0(n-1) + 2^{n-1} \times E_1| \\ &+ |e_0(n-1) + 2^{n-1} \times E_2| + |e_0(n-1) + 2^{n-1} \times E_3| + \ldots \\ &+ |e_{2^{2(n-1)}-1}(n-1) + 2^{n-1} \times E_0| + |e_{2^{2(n-1)}-1}(n-1) + 2^{n-1} \times E_1| \\ &+ |e_{2^{2(n-1)}-1}(n-1) + 2^{n-1} \times E_2| \\ &+ |e_{2^{2(n-1)}-1}(n-1) + 2^{n-1} \times E_3|), \end{aligned} \tag{4}$$

which still requires 2^{2n} addition operations. In the following, we first present and prove Lemma 1, and then, we use it to decrease the number of addition operations in Eq. (4).

Lemma 1. *In an n-bit disjoint approximate adder, comprising identical disjoint single-bit building blocks, if the EDs of the employed single-bit building blocks are all positive, the absolute value operator in Eq. (3) is removed, and it is restated as follows:*

$$e_{4(j-1)+l}(i) = e_j(i-1) + 2^{i-1} \times E_l.$$

Proof. As mentioned in Section 2.1.2, a disjoint single-bit building block possesses four EDs referred to as E_0, E_1, E_2, and E_3, which are equivalent to $e_0(1)$, $e_1(1)$, $e_2(1)$, and $e_3(1)$, respectively. When EDs of the building block are all positive, for $i = 2$ (i.e., at level 2), both the terms $e_j(i-1)$ and E_l, where $j, l \in \{0,1,2,3\}$, are positive as well. In this case, the terms within the absolute value operator in Eq. (3) are positive, and thus, the absolute value operator can be removed.

Lets assume that $e_j(i-1)$, which represents the content of each leaf at level $i-1$, is always positive. In this case, since E_0 through E_3 (i.e., E_l) are also positive, the 2^{2i} leaves at level i which are represented by $e_{4j+l}(i)$ in Eq. (3), are positive, regardless of the absolute value operator. Thus, the absolute value operator can be removed. Consequently, by the principle of *Mathematical Induction* [24], the absolute value operator in Eq. (3) can be removed for all i ($2 \le i \le n$). □

Based on Lemma 1, when E_0 through E_3 are all positive, we can remove the absolute value operators in Eq. (4) which leads to the following equation:

$$\begin{aligned} MED(n) &= \frac{1}{2^{2n}} \times [4 \times (e_0(n-1) + \ldots + e_{2^{2(n-1)}-1}(n-1))] \\ &\quad + \frac{1}{2^{2n}} \times [2^{2(n-1)} \times 2^{n-1} \times (E_0 + E_1 + E_2 + E_3)] \\ &= MED(n-1) + 2^{n-3} \times (E_0 + E_1 + E_2 + E_3). \end{aligned} \tag{5}$$

Eq. (5) computes MED recursively, using n addition operations, which makes it a lot more efficient compared to Eq. (4). Besides, owing to the fact that Eq. (5) is a *first-order linear* recurrence relation [24], it can be solved using approaches such as the *iteration method*. Solving Eq. (5) results in Eq. (6), which is of the order of $O(1)$.

$$MED(n) = \frac{\sum_{l=0}^{3} E_l}{4} + (2^{n-2} - \frac{1}{2}) \times \sum_{l=0}^{3} E_l. \tag{6}$$

Since we have used Lemma 1 to obtain Eq. (6), this equation can only be used when EDs of building blocks are

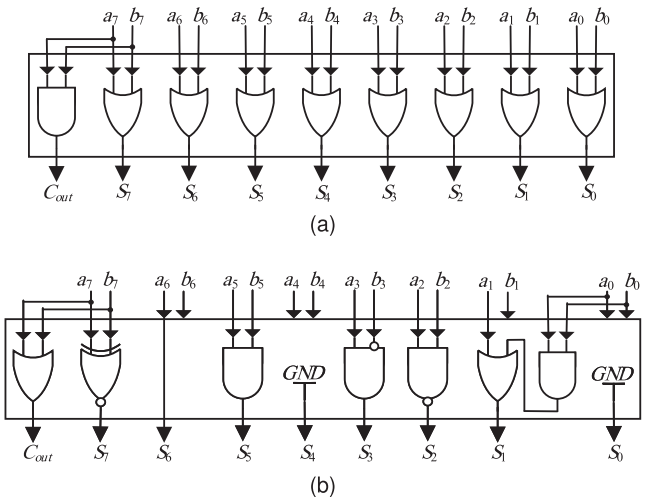


Fig. 3. Gate-level implementation of 8-bit LOA and the proposed 8-bit approximate adder. (a) LOA; (b) the proposed adder.

positive. However, we can follow a similar approach to demonstrate that Eq. (6) can also be used in situations where all EDs are negative, with the only difference that the resulting MED values will also be negative. As a result, we can state that when the EDs of the employed single-bit building blocks are all of the same sign (i.e., all positive or all negative), MED can be computed using Eq. (7).

$$MED(n) = |\frac{\sum_{l=0}^{3} E_l}{4} + (2^{n-2} - \frac{1}{2}) \times \sum_{l=0}^{3} E_l|. \quad (7)$$

2.2.2 Second Observation: Importance of Input Patterns

When it comes to selecting an optimal or near optimal adder design for a specific problem (e.g., a DSP application), the design performing the best trade-off within the range of application dictated constraints is always desired. In approximate computing, the compromise is between error criterion (e.g., MED) and circuit characteristics (i.e., power and energy consumption, area occupation, and circuit delay). Error characteristics should be calculated in a way that describes the error behavior of the selected designs, based on applications input data set. In other words, input patterns should be taken into consideration while selecting a design for a specific application. In the following, we first describe MED computation based a specified input pattern and then present an example to demonstrate the impact of input patterns on the problem of finding optimal adders for a given application.

Consider $P(x)$ as the probability of x to be 1, where x can be any single bit of the two binary numbers $a = (a_{n-1}\, a_{n-2} \ldots a_0)$ and $b = (b_{n-1} b_{n-2} \ldots b_0)$, which are the inputs to an n-bit adder. Consequently, $\overline{P(x)} = 1 - P(x)$ is the probability of x to be 0. Based on this notation, every input pattern can be demonstrated by two sequences $Pat(a) = (P(a_{n-1}), P(a_{n-2}), \ldots, P(a_0))$ and $Pat(b) = (P(b_{n-1}), P(b_{n-2}), \ldots, P(b_0))$. All together, these two sequences indicate the probability of each single bit of the two inputs a and b to be 1. For instance, the two sequences $Pat(a) = (0.1, 0.3, 0.6)$ and $Pat(b) = (0.7, 0.4, 0.9)$ demonstrate the pattern of two inputs to a 3-bit adder, according to which $P(a_0) = 0.6$. To compute MED for a given input pattern, we can use Eq. (1), where the probability of each input pair (i.e., Q_j) is computed by Eq. (8).

$$Q_j = \prod_{q=0}^{n-1} F(a_q) \times F(b_q)$$

$$F(x) = \begin{cases} P(x) & x = 1 \\ \overline{P(x)} & x = 0 \end{cases}. \quad (8)$$

Fig. 3 shows the two 8-bit energy and area-efficient approximate adders to clarify the importance of input patterns. We consider *Lena* and *F16* images as inputs to the image addition benchmark. Table 1 shows the MED of the two adders for uniformly distributed inputs and their power, area, and delay by using Verilog description and synthesis in Synopsys Design Compiler with a NanGate FreePDK45 nm library [25].

If we consider the MED as our criterion, LOA is selected as the best choice for the image addition. However, the results in Table 2 and Fig. 4, demonstrate the superior performance of the proposed adder in both mathematical and subjective metrics, respectively. Although the proposed adder has higher MED than LOA for uniformly distributed inputs, it has better MED for the applied input images. The MED of LOA is 58.99 for the input patterns of Lena and F16, but the MED of the proposed adder is 35.22 for the same input images.

2.2.3 Third Observation: MED Calculation for Different Input Patterns

In this subsection, we demonstrate how to obtain MED of approximate adders for a given pair input patterns by utilizing the quad-tree representation.

TABLE 1
Comparison of Circuit and Error Characteristics Between LOA and the Proposed Adder

Design	Dynamic Power (μW)	Static Power (μW)	Area (μm^2)	Delay (fs)	MED (Uniform)
LOA	2.76	0.226	10.108	0.06	47.875
Proposed	2.63	0.195	7.714	0.09	49.32

TABLE 2
Comparison of Output Quality and MED of LOA and the Proposed Adder in an Image Addition Application

Design	PSNR	MSE	MAE	MED (Lena and F16)
LOA	9.705	6958	71.39	58.99
Proposed	15.291	1922	34.25	35.22

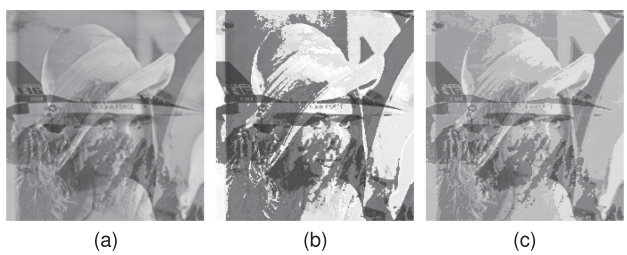


Fig. 4. The results of performing the image addition operation. (a) Accurate 8-bit adder; (b) 8-bit LOA; (c) proposed 8-bit adder in Fig. 3b.

Based on the quad-tree representation in Fig. 2, leaf j at the last level (i.e, level n) contains the ED of the jth input pair to the adder, where $j = (a_0b_0a_1b_1 \dots a_{n-2}b_{n-2}a_{n-1}b_{n-1})_{10}$. Considering this fact and using Eq. (8), for a given input pattern provided by two sequences $Pat(a) = (P(a_{n-1}), P(a_{n-2}), \dots, P(a_0))$ and $Pat(b) = (P(b_{n-1}), P(b_{n-2}), \dots, P(b_0))$, MED can be computed using Eq. (9).

$$\begin{aligned} MED(n) = {} & [\overline{P(a_0)}\ \overline{P(b_0)} \dots \overline{P(a_{n-1})}\ \overline{P(b_{n-1})}] \times e_0(n) + \dots \\ & + [P(a_0)P(b_0) \dots P(a_{n-1})P(b_{n-1})] \times e_{2^{2n}-1}(n). \end{aligned} \tag{9}$$

Based on Eq. (3) and Lemma 1, we can restate Eq. (9) as

$$\begin{aligned} MED(n) = {} & [\overline{P(a_0)}\ \overline{P(b_0)} \dots \overline{P(a_{n-2})}\ \overline{P(b_{n-2})}] \times [e_0(n-1) \\ & \times (\overline{P(a_{n-1})}\ \overline{P(b_{n-1})} + \overline{P(a_n)}P(b_n) + P(a_{n-1})\overline{P(b_{n-1})} \\ & + P(a_{n-1})P(b_{n-1})) + 2^{n-1} \times (\overline{P(a_{n-1})}\ \overline{P(b_{n-1})}E_0 \\ & + \overline{P(a_{n-1})}P(b_{n-1})E_1 + P(a_{n-1})\overline{P(b_{n-1})}E_2 \\ & + P(a_{n-1})P(b_{n-1})E_3)] + \dots \\ & + [\overline{P(a_0)}\ \overline{P(b_0)} \dots \overline{P(a_{n-2})}\ \overline{P(b_{n-2})}] \times [e_{2^{2(n-1)}-1}(n-1) \\ & \times (\overline{P(a_{n-1})}\ \overline{P(b_{n-1})} + \overline{P(a_{n-1})}P(b_{n-1}) + P(a_{n-1})\overline{P(b_{n-1})} \\ & + P(a_{n-1})P(b_{n-1}))2^{n-1} \times (\overline{P(a_{n-1})}\ \overline{P(b_{n-1})}E_0 \\ & + \overline{P(a_{n-1})}P(b_{n-1})E_1 + P(a_n)\overline{P(b_{n-1})}E_2 \\ & + P(a_{n-1})P(b_{n-1})E_3)], \end{aligned}$$

and since $\forall i \in [0, n-1] : (\overline{P(a_i)}\ \overline{P(b_i)} + \overline{P(a_i)}P(b_i) + P(a_i)\overline{P(b_i)} + P(a_i)P(b_i) = 1)$, it is simplified to the following form:

$$\begin{aligned} MED(n) = {} & [\overline{P(a_0)}\ \overline{P(b_0)} \dots \overline{P(a_{n-2})}\ \overline{P(b_{n-2})}e_0(n-2)] \\ & + [\overline{P(a_0)}\ \overline{P(b_0)} \dots \overline{P(a_{n-2})}\ \overline{P(b_{n-2})} \times 2^{n-1} \\ & \times (\overline{P(a_{n-1})}\ \overline{P(b_{n-1})}E_0 + \overline{P(a_{n-1})}P(b_{n-1})E_1 \\ & + P(a_{n-1})\overline{P(b_{n-1})}E_2 + P(a_{n-1})P(b_{n-1})E_3)] + \dots \\ & + [P(a_0)P(b_0) \dots P(a_{n-2})P(b_{n-2})e_{2^{2(n-1)}-1}(n-1)] \\ & + [P(a_0)P(b_0) \dots P(a_{n-2})P(b_{n-2}) \times 2^{n-1} \times (\overline{P(a_{n-1})} \\ & \overline{P(b_{n-1})}E_0 + \overline{P(a_{n-1})}P(b_{n-1})E_1 + P(a_{n-1})\overline{P(b_{n-1})}E_2 \\ & + P(a_{n-1})P(b_{n-1})E_3)]. \end{aligned} \tag{10}$$

Considering the characteristics of the quad-tree representation shown in Fig. 2, level $n-1$ represents a disjoint approximate adder with the length of $n-1$, which takes $a = (a_{n-2} \dots a_1a_0)$ and $b = (b_{n-2} \dots b_1b_0)$ as inputs. Thus, similar to Eq. (9), the MED of the adder at level $n-1$ can be computed by the following equation:

$$\begin{aligned} MED(n-1) = {} & [\overline{P(a_0)}\ \overline{P(b_0)} \dots \overline{P(a_{n-2})}\ \overline{P(b_{n-2})}e_0(n-1)] \\ & + \dots \\ & + [P(a_0)P(b_0) \dots P(a_{n-2})P(b_{n-2})e_{2^{2(n-1)}-1}(n-1)]. \end{aligned} \tag{11}$$

On the other hand, for the adder at level $n-1$ we have

$$\begin{aligned} & \overline{P(a_0)}\ \overline{P(b_0)} \dots \overline{P(a_{n-2})}\ \overline{P(b_{n-2})} \\ & + \overline{P(a_0)}\ \overline{P(b_0)} \dots \overline{P(a_{n-2})}P(b_{n-2}) + \dots \\ & + P(a_0)P(b_0) \dots P(a_{n-2})\overline{P(b_{n-2})} \\ & + P(a_0)P(b_0) \dots P(a_{n-2})P(b_{n-2}) = 1. \end{aligned} \tag{12}$$

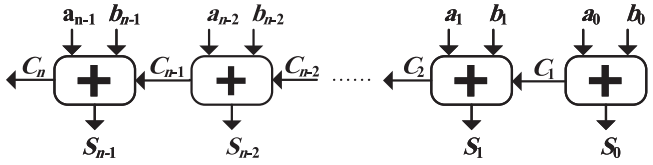


Fig. 5. General structure of LPAAs.

Based on Eqs. (11) and (12), we can simplify Eq. (10) to the following form:

$$\begin{aligned} MED(n) = {} & MED(n-1) + 2^{n-1} \times [\overline{P(a_{n-1})}\ \overline{P(b_{n-1})}E_0 \\ & + \overline{P(a_{n-1})}P(b_{n-1})E_1 + P(a_{n-1})\overline{P(b_{n-1})}E_2 \\ & + P(a_{n-1})P(b_{n-1})E_3]. \end{aligned} \tag{13}$$

For a given input pattern, Eq. (13), which is of the order of $O(n)$, recursively computes MED for the family of adders illustrated in Fig. 2. Since we have used Lemma 1 to obtain Eq. (13), it only works when EDs of building blocks are positive. However, it can be shown that Eq. (13) can also be used when EDs of building blocks are all negative. In this case, the resulting MED is also negative, and thus, its absolute value represents MED.

As we can see, the quad-tree representation has the potential to be used as a framework to compute error-metrics such as MED, for both uniform and specified input patterns. However, it has two limitations. First, when the EDs of single-bit building blocks have different signs, MED computation based on the quad-tree representation is not very straightforward, and we had to use Lemma 1 to be able to compute MED for specific cases. Second, the quad-tree representation can only model disjoint approximate adders. Thus, it can not be used to find formulae for error-metrics while adders comprise building blocks with carry signals among them. Therefore, in Section 3, we address both of these problems by extending the quad-tree representation.

3 Proposed Approach

In this section, we first show how to use the quad-tree to model carry propagation and EDs with different signs. Next, we demonstrate an accurate calculation of the MED and EP of approximate adders for any given input pattern. At last, a case study is provided to illustrate an up-close and detailed example of the proposed approach. Fig. 5 shows the most general structure of an LPAA which we consider as our baseline. In Fig. 5, except for the first approximate full adder in the first bit position which does not have a carry-in input, full adders in all other bit positions may have one. Thus, all blocks may generate carry-out signals. Moreover, each of them can have a different configuration compared to the other ones. Also, Due to their different implementation and functionality, there may be EDs with different signs. Considering all possible configurations, including those with different sign EDs, there are 4^8 separate approximate single-bit full adders available. Thus, for an 8-bit LPAA, the design space contains 65536^8 different configurations but Eq. (7) can calculate MED for only a small fraction of this rich design space.

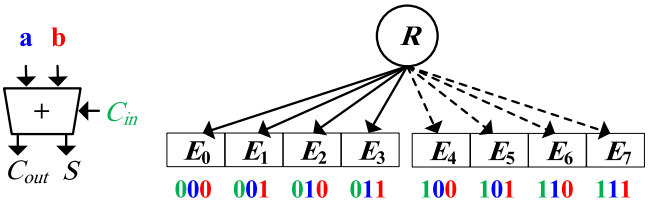


Fig. 6. Extended quad-tree representation of single-bit building blocks with carry signals.

Modeling Carry Propagation. To model adders with carry-out signals, we extend the quad-tree of Fig. 1, as Fig. 6. As shown in Fig. 6, approximate full adders receive three inputs, so they have eight edges. Thus, they should be modeled with eight different EDs (the dashed-lines refer to the case where carry-in = 1).

Depending on the carry-in input, only four of those edges are used at a time. Fig. 7 shows the quad-tree of two consecutive single-bit full adders forming a 2-bit LPAA. Table 3 shows the truth table of the single-bit full adder at bit position 1 and Table 4 shows the truth table of the single-bit full adder at bit position 2.

Table 3 generates a carry-out signal only for the case it receives 11 as its inputs, otherwise, its carry-out is zero. So, if both a_0 and b_0 are 1, then the four selected edges showing the functionality of the second block, are those having carry-in of 1 (i.e., the four bottom rows in Table 4). But, if a_0 and b_0 are not 11, then the other four edges, the top four rows of Table 4, are selected to represent the functionality of this block.

Modeling EDs With Different Signs. Consider the example 2-bit LPAA in Fig. 7. Four of the leaves are negative. Therefore, there are EDs with different signs and LEMMA 1 is no longer applicable. Hence, we cannot use Eq. (7) for MED computation. To overcome this problem, we aim on dividing MED into several groups.

For example, the MED of the 2-bit LPAA can be divided into MED of positive and negative nodes, and each of them can be calculated, separately. Finally, the total MED is the sum of both MED of positive and negative nodes. In a more generic manner, to classify MED into several smaller groups, we need to find out all interactions between two consecutive approximate full adders at bit positions $i-1$ and i (i.e., parent nodes and their connecting edges to their child nodes).

3.1 Node Grouping

Based on the quad-tree representation, at the depth $i-1$, there are $2^{2(i-1)}$ nodes. We classify these nodes into 14 separate groups, depending on carry-out signals, the sign of their EDs, and their interaction with their edges (As shown in Fig. 8).

Therefore, nodes, based on their carry-out signal, can be classified into two groups (i.e., carry-out = 1, and carry-out = 0). Depending on their sign, they can be classified into

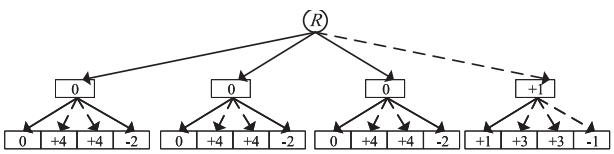


Fig. 7. An example of a 2-bit non-disjoint LPAA uisng modified quad-tree representation.

TABLE 3
Approximate Adder at Bit Position 1

♯	a	b	s	C_{out}	ED
1	0	0	0	0	0
2	0	1	1	0	0
3	1	0	1	0	0
4	1	1	1	1	+1

three groups (i.e., positive, negative, and zero). Also, for the positive and negative cases, they may have the absolute ED value larger, equal, or less than their edges with different sign. Therefore, again, we classify them into three other groups (namely, *Normal*, *Abnormal0*, *Abnormal1*):

1) *Normal*: parents at depth $i-1$, where their absolute EDs are less than 2^{i-1}. In this case, the sign of the child node is always determined by the sign of the edge that connects it to its parent.
2) *Abnormal0*: refers to the parents at depth i-1, with their absolute EDs exactly equal to 2^{i-1}. In this case, based on the amount of the parents edge, the child node is zero.
3) *Abnormal1*: refers to the parents at depth i-1, with their absolute EDs greater than 2^{i-1}. In this case the child nodes sign can be the same as its connecting edges sign, or its parents sign.

In other words, for a single-bit approximate full adder, nodes with the ED value of ±1, are normal. ED value of ±2 refers to abnormal0 nodes and abnormal1 nodes are identified with ED value of ±3.

Since the absolute amount of leaves determine MED, we use this classification and describe MED in a more fine-grained manner by dividing MED into 12 portions (two zero groups are neglected for MED calculation) in a way that the sum of all parts is equal to MED. Table 5, shows the classified MED of an n-bit LPAA and its portions (Indexes "P", "N", and "Z" stand for positive, negative, and zero, respectively. Normal, abnormal0, and abnormal1 are shown with "$Norm$", "$Ab0$", and "$Ab1$").

3.2 Edge Grouping

This process is quite the same as node classification except that the carry-in signal should be taken into considerations too. So, based on receiving carry-in and generating carry-out, there are four groups. Again, we classify the edges based on their sign to be positive, negative, or zero, into three groups. Finally, based on the value of the edge that connects the current child node to its parent, there are three

TABLE 4
Approximate Adder at Bit Position 2

♯	C_{in}	a	b	s	C_{out}	ED
1	0	0	0	0	0	0
2	0	0	1	1	1	+2
3	0	1	0	1	1	+2
4	0	1	1	1	0	-1
5	1	0	0	1	0	0
6	1	0	1	1	1	+1
7	1	1	0	1	1	+1
8	1	1	1	0	1	-1

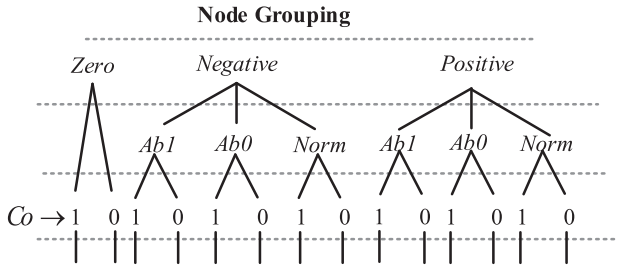


Fig. 8. Node grouping: All 14 groups of nodes.

more groups formed, called Normal, Abnormal0, and Abnormal1 (As shown in Fig. 9). Thus, there are 28 different types of edges available, shown in Table 6 (to save space, we only show normal edges in the table). Since these edges describe the behavior of an approximate adder in the current bit position (i.e., at depth i), the edge variables starts "CR". There are two different kinds of edge variables; (1) CR variables refer to the weighted sum of specific edge errors; and (2) $CRNUM$ variables are equal to the occurrence probability of a specific edge error.

Fig. 10 shows how to calculate all 12 portions of positive CR variables for a given approximate full adder and a given input pattern. The algorithm in Fig. 10 takes the truth table and the probabilities of its input pair as its inputs and outputs all 12 portions of "CR_P". It observes all the eight possible input combinations and their errors. Then, based on their input pattern probability, the amount of errors, carry-in, and carry-out signals, the amount of each of the classes of "CR_P" is calculated. Since the for loop in line 7, iterates 8 times at the most, then all "CR_P" variables are calculated with $O(8)$. The same process should be accomplished for CR_N, too. But for zero nodes CR_Z is always equal to 0.

TABLE 5
Dividing MED into 12 Portions

Portion name	Description
$MED_P(Norm, \overline{Co}, n)$	MED of Positive Normal leaves with Carry-out = 0
$MED_P(Norm, Co, n)$	MED of Positive Normal leaves with Carry-out = 1
$MED_P(Ab0, \overline{Co}, n)$	MED of Positive Abnormal0 leaves with Carry-out = 0
$MED_P(Ab0, Co, n)$	MED of Positive Abnormal0 leaves with Carry-out = 1
$MED_P(Ab1, \overline{Co}, n)$	MED of Positive Abnormal1 leaves with Carry-out = 0
$MED_P(Ab1, Co, n)$	MED of Positive Abnormal1 leaves with Carry-out = 1
$MED_N(Norm, \overline{Co}, n)$	MED of Negative Normal leaves with Carry-out = 0
$MED_N(Norm, Co, n)$	MED of Negative Normal leaves with Carry-out = 1
$MED_N(Ab0, \overline{Co}, n)$	MED of Negative Abnormal0 leaves with Carry-out = 0
$MED_N(Ab0, Co, n)$	MED of Negative Abnormal0 leaves with Carry-out = 1
$MED_N(Ab1, \overline{Co}, n)$	MED of Negative Abnormal1 leaves with Carry-out = 0
$MED_N(Ab1, Co, n)$	MED of Negative Abnormal1 leaves with Carry-out = 1

Having 14 groups for parent nodes and 28 groups for edges, we end up having 392 different types of interactions between a node and its connecting edge. However, there are a lot of invalid interactions among these 392 types. There are three categories for invalid interactions: (1) invalid parents, (2) invalid edges, (3) invalid connections. These invalid interactions and their conditions are as followings:

*1. Invalid Parents.*Positive abnormal0 and abnormal1 parents that do not generate carry-out do not exist. Negative abnormal0 and abnormal1 parents that generate carry-out do not exist either.

2. Invalid Edges.

2.1) A positive normal edge with carry-in of 1 and no carry out does not exist. Similarly, a negative normal edge with no carry-in and carry-out of 1, does not exist.

2.2) There are no positive abnormal0 edges that do not generate a carry-out. Also, there are no positive abnormal1 edges that have no carry-out or receives a carry-in of 1.

2.3) A negative abnormal0 edge that generates a carry-out does not exist. A negative abnormal1 edge that generates carry-out or does not receive a carry-out does not exist.

*3. Invalid Connections.*A parent node that has carry-out of 1 cannot interact with an edge that does not receive a carry-in. Also, a parent node that has no carry-out cannot interact with an edge with carry-in of 1.

After examining all possible 14 $\times$ 28 (392) interactions, we find out that 312 of them are invalid. Therefore, only 80 interactions are possible between a parent not and its edges. For each of these 80 interactions between parents and edges, we must identify the group to which the offspring nodes belong. Then, each one of the 12 portions of MED can be calculated separately. As an example, Table 7 shows the child node classes when normal positive or zero parents interact with different classes of positive edges (Note that invalid connections are removed from the table).

3.3 Case Study

As an example, assume that we want to calculate the desired portion for bit-position i ($MED_P(Ab0, Co, i)$). Therefore, we need all MED and EP portions at bit-position i-1. Also, we need all CR and $CRNUM$ variables for the ith (current) block. The idea is to add the errors of the ith block to the cumulative amount (from bit-positions 0 to i-1) of desired portion of the MED for bit-position i. Thus, we must first identify all interactions, at bit position i-1 that are leading to positive, abnormal nodes with carry-out at bit-position i. Table 8 shows identifies all four interactions that leads to child nodes of group $P(Ab0, Co)$. As an example, the first row of Table 8 indicates that for each $P(Ab0, Co)$ nodes at i-1, there is one edge (i.e., $P(Norm, Cin, Co)$) that leads to a $P(Ab0, Co)$ child node at bit-position i.

Before calculating the desired portion of MED at bit position i, we must know how to compute the contributed EDs from parents and their edges to the desired portion. Based on Eq. (3), the error of a child node can be considered as the absolute summation (or subtraction) between the term $e_j(i-1)$ and the term $2^{i-1} \times E_l$. Our approach for

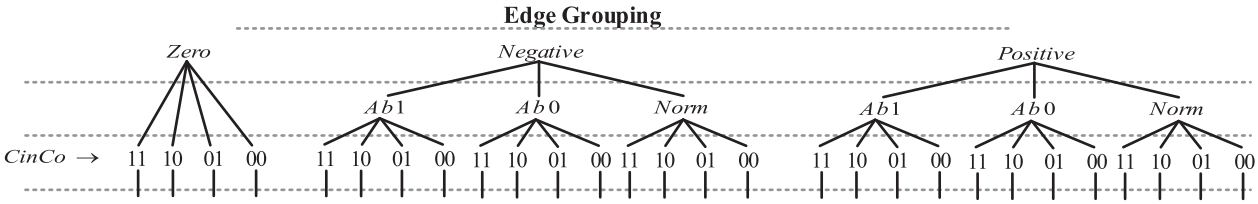


Fig. 9. Edge grouping: All 28 groups of edges.

computing the error of each interaction is to calculate each of the two terms separately, and then add them together.

In Eq. (3), the left hand side term, $e_{4j+l}(i)$ is the total error of a specific interaction, which is also referred to as a child error. In the quad-tree representation, this error is written inside a child node at level i. The term, $e_j(i-1)$ is the error of the parent which also indicates that the errors of the parents are propagated to the lower levels. In other words, this term contains the error that the parent is contributing. The term, $2^{i-1} \times E_l$, is the multiplication between the error of an edge and its corresponding weight. This is because when calculating the variables related to edges (Fig. 10), we neglect their bit-positions. This term contains the error that the edge (or the current block) contributes. After computing each term, if they do not have same signs, the error of the child is equal to the absolute amount subtraction between $e_j(i-1)$ and $2^{i-1} \times E_l$. Otherwise, the error of the child is the absolute summation of the them.

Now for each of the interactions shown in Table 8, we first calculate the parent error. Then, we show how to compute the edge errors, at their bit-position (i.e., the term $2^{i-1} \times E_l$). Form row 1 in Table 8, we understand that trough an interaction between a $P(Ab0, Co, i-1)$ parent and a $P(Norm, Cin, Co)$, a positive abnormal0 child node with a carry-out is formed at level i. Therefore, to calculate the parent errors, we need two values; (1) the sum of all $P(Ab0, Co, i-1)$ parents; (2) and, the occurrence probability of a $P(Norm, Cin, Co)$ edges of the current adder. By multiplying both of them, we can compute the error amount that all the $P(Ab0, Co, i-1)$ parents contribute trough their $P(Norm, Cin, Co)$ edges. The first value is equal to $MED_P(Ab0, Co, i-1)$ and the second value is $CRNUM_P(Norm, Cin, Co)$. We assume that we have all previous MED and EP portions, and the CR variables are all computed in constant time and space. Therefore, the parent errors of the first interaction of Table 8 can be calculated as Eq. (14).

TABLE 6
Classification of Normal Edges

Portion name	Description
$CR_P(Norm, \overline{C_{in}}, \overline{Co})$	Weighted Sum of Positive Normal edges with Carry-in=0 & Carry-Out=0
$CR_P(Norm, \overline{C_{in}}, Co)$	Weighted Sum of Positive Normal edges with Carry-in=0 & Carry-Out=1
$CR_P(Norm, C_{in}, \overline{Co})$	Weighted Sum of Positive Normal edges with Carry-in=1 & Carry-Out=0
$CR_P(Norm, C_{in}, Co)$	Weighted Sum of Positive Normal edges with Carry-in=1 & Carry-Out=1
$CR_N(Norm, \overline{C_{in}}, \overline{Co})$	Weighted Sum of Negative Normal edges with Carry-in=0 & Carry-Out=0
$CR_N(Norm, \overline{C_{in}}, Co)$	Weighted Sum of Negative Normal edges with Carry-in=0 & Carry-Out=1
$CR_N(Norm, C_{in}, \overline{Co})$	Weighted Sum of Negative Normal edges with Carry-in=1 & Carry-Out=0
$CR_N(Norm, C_{in}, Co)$	Weighted Sum of Negative Normal edges with Carry-in=1 & Carry-Out=1

Input: $TruthTable_{Approximate}$, $P(a)$, $P(b)$
Output: All 12 portions of CR_P

$Err = TruthTable_{Approximate} - TruthTable_{Accurate}$
if This adder recieves carry-in signal **then**
  $Combinations \leftarrow 8$
else
  $Combinations \leftarrow 4$
end if
for $i = 1$ **to** $Combinations$ **do**
  if $a == 0$ **and** $b == 0$ **then**
    $Probability \leftarrow \overline{P(a)} \times \overline{P(b)}$
  else if $a == 0$ **and** $b == 1$ **then**
    $Probability \leftarrow \overline{P(a)} \times P(b)$
  else if $a == 1$ **and** $b == 0$ **then**
    $Probability \leftarrow P(a) \times \overline{P(b)}$
  else if $a == 1$ **and** $b == 1$ **then**
    $Probability \leftarrow P(a) \times P(b)$
  end if
  if $Err(i) == +1$ **then**
    if $\overline{Cin}$ **and** $\overline{Co}$ **then**
      $CR_P(Norm, \overline{Cin}, \overline{Co}) += Probability \times |Err(i)|$
    else if $\overline{Cin}$ **and** Co **then**
      $CR_P(Norm, \overline{C_{in}}, Co) += Probability \times |Err(i)|$
    else if Cin **and** $\overline{Co}$ **then**
      $CR_P(Norm, Cin, \overline{Co}) += Probability \times |Err(i)|$
    else if Cin **and** Co **then**
      $CR_P(Norm, Cin, Co) += Probability \times |Err(i)|$
    end if
  else if $Err(i) == +2$ **then**
    if $\overline{Cin}$ **and** $\overline{Co}$ **then**
      $CR_P(Ab0, \overline{Cin}, \overline{Co}) += Probability \times |Err(i)|$
    else if $\overline{Cin}$ **and** Co **then**
      $CR_P(Ab0, \overline{C_{in}}, Co) += Probability \times |Err(i)|$
    else if Cin **and** $\overline{Co}$ **then**
      $CR_P(Ab0, Cin, \overline{Co}) += Probability \times |Err(i)|$
    else if Cin **and** Co **then**
      $CR_P(Ab0, Cin, Co) += Probability \times |Err(i)|$
    end if
  else if $Err(i) == +3$ **then**
    if $\overline{Cin}$ **and** $\overline{Co}$ **then**
      $CR_P(Ab1, \overline{Cin}, \overline{Co}) += Probability \times |Err(i)|$
    else if $\overline{Cin}$ **and** Co **then**
      $CR_P(Ab1, \overline{C_{in}}, Co) += Probability \times |Err(i)|$
    else if Cin **and** $\overline{Co}$ **then**
      $CR_P(Ab1, Cin, \overline{Co}) += Probability \times |Err(i)|$
    else if Cin **and** Co **then**
      $CR_P(Ab1, Cin, Co) += Probability \times |Err(i)|$
    end if
  end if
end for

Fig. 10. Calculation of all 12 portions of CR_P for a given approximate full adder.

TABLE 7
Normal Positive and Zero Parents Interacting With Different Classes of Positive Edges

parent	edge	child
$Pos(Norm, \overline{Co}, i-1)$	$Pos(Norm, \overline{Cin}, \overline{Co})$	$Pos(Norm, \overline{Co}, i)$
	$Pos(Ab0, \overline{Cin}, \overline{Co})$	Invalid Edge
	$Pos(Ab1, \overline{Cin}, \overline{Co})$	Invalid Edge
	$Pos(Norm, \overline{Cin}, Co)$	$Pos(Norm, Co, i)$
	$Pos(Ab0, \overline{Cin}, Co)$	$Pos(Ab1, Co, i)$
	$Pos(Ab1, \overline{Cin}, Co)$	$Pos(Ab1, Co, i)$
$Pos(Norm, Co, i-1)$	$Pos(Norm, \overline{Cin}, \overline{Co})$	Invalid Edge
	$Pos(Ab0, \overline{Cin}, \overline{Co})$	Invalid Edge
	$Pos(Ab1, \overline{Cin}, \overline{Co})$	Invalid Edge
	$Pos(Norm, \overline{Cin}, Co)$	$Pos(Norm, Co, i)$
	$Pos(Ab0, \overline{Cin}, Co)$	$Pos(Ab1, Co, i)$
	$Pos(Ab1, \overline{Cin}, Co)$	Invalid Edge
$Zero(\overline{Co}, i-1)$	$Pos(Norm, \overline{Cin}, \overline{Co})$	$Pos(Norm, \overline{Co}, i)$
	$Pos(Ab0, \overline{Cin}, \overline{Co})$	Invalid Edge
	$Pos(Ab1, \overline{Cin}, \overline{Co})$	Invalid Edge
	$Pos(Norm, \overline{Cin}, Co)$	$Pos(Norm, Co, i)$
	$Pos(Ab0, \overline{Cin}, Co)$	$Pos(Ab0, Co, i)$
	$Pos(Ab1, \overline{Cin}, Co)$	$Pos(Ab1, Co, i)$
$Zero(Co, i-1)$	$Pos(Norm, \overline{Cin}, \overline{Co})$	Invalid Edge
	$Pos(Ab0, \overline{Cin}, \overline{Co})$	Invalid Edge
	$Pos(Ab1, \overline{Cin}, \overline{Co})$	Invalid Edge
	$Pos(Norm, \overline{Cin}, Co)$	$Pos(Norm, Co, i)$
	$Pos(Ab0, \overline{Cin}, Co)$	$Pos(Ab0, Co, i)$
	$Pos(Ab1, \overline{Cin}, Co)$	Invalid Edge

$$MED_P(Ab0, Co, i-1) \times CRNUM_P(Norm, Cin, Co, i). \quad (14)$$

Similarly, the parent errors for row 2 can be calculated as Eq. (15).

$$MED_N(Ab0, \overline{Co}, i-1) \times CRNUM_P(Ab1, \overline{Cin}, Co, i). \quad (15)$$

For rows 3 and 4, the $MED_Z(\overline{Co}, i-1)$ and $MED_Z(Co, i-1)$ are both zero. Therefore, for these two rows the parent errors are zero. To compute the errors caused by edges for the first row, two values must be available; (1) the occurrence probability of all $P(Ab0, Co, i-1)$ parents; (2) and, the sum of errors of $P(Norm, Cin, Co)$ edges of the current block. The product of these two is equivalent to calculating the edge errors for all $P(Ab0, Co, i-1)$ parents. The first value is equal to $EP_P(Ab0, Co, i-1)$, and the second value refers to $CR_P(Norm, Cin, Co)$. Thus, both values are available. Therefore, the edge errors for row 1 of Table 8 are calculated as Eq. (16).

$$EP_P(Ab0, Co, i-1) \times CR_P(Norm, Cin, Co, i) \times 2^{bitposition}. \quad (16)$$

Similarly, Eq. (17) shows the edge errors for row 2 of Table 8.

$$EP_N(Ab0, \overline{Co}, i-1) \times CR_P(Ab1, \overline{Cin}, Co, i) \times 2^{bitposition}. \quad (17)$$

And the edge errors for rows 3 and 4 are calculated as Eqs. (18) and (19).

$$EP_Z(\overline{Co}, i-1) \times CR_P(Ab0, \overline{Cin}, Co, i) \times 2^{bitposition} \quad (18)$$

$$EP_Z(Co, i-1) \times CR_P(Ab0, Cin, Co, i) \times 2^{bitposition}. \quad (19)$$

So far, the total parent errors and total the edge errors of all four interactions have been calculated using Eqs. (14), (15), (16), (17), (18), and (19). For each interaction, the parent and the edge errors must be added/subtracted with/from each other. If they have the same signs, the total error of that interactions is equal to the absolute amount of their summation. Otherwise, they are subtracted from each other. For the first row, both the parent and the edge are positive. Therefore, the total error that row 1 contributes to $MED_P(Ab0, Co, i)$ is equal to absolute summation of Eq. (14) and Eq. (16). For the interaction in the second row of Table 8, the parent and the edge have different signs. Thus, the total error that this interaction is contributing to $MED_P(Ab0, Co, i)$ is equal to the absolute difference of Eqs. (15) and (17). For row 3 and row 4, the parent errors is zero, hence, the contributed errors of these interactions is equal to their corresponding edge errors, Eqs. (18) and (19), respectively. Finally, the $MED_P(Ab0, Co, i)$ is equal to the summation of total error of all four interactions.

In above equations, we assume that we have all MED and EP portions for the previous level i-1, and we demonstrate how to calculate MED at level i. Here we show the calculation of $EP_P(Ab0, Co, i)$ portions for level i. By definition, $EP_P(Ab0, Co, i)$ is the occurrence probability of positive abnormal0 nodes with carry-out at level i. Similar to computation of $MED_P(Ab0, Co, i)$, the first step is to identify all interactions that lead to the desired group of nodes at level i (i.e., positive abnormal0 nodes with carry-out), which is shown in Table 8. Now we start calculating the contribution of each of the four rows/interactions in Table 8 to $EP_P(Ab0, Co, i)$.

For row 1 of Table 8, to compute this portion of EP, we need two values; (1) the occurrence probability of all $P(Ab0, Co)$ parents at level i-1; and, (2) the probability of a $P(Ab0, Co)$ generating a $P(Ab0, Co)$ child. The amount that the first row/interaction is contributing to $EP_P(Ab0, Co, i)$ is equal to the product of these two values. The first value is equal to $EP_P(Ab0, Co, i)$. Based on Table 8, a $P(Ab0, Co)$ parent at level i-1 generates a $P(Ab0, Co)$ node at level i through its $P(Norm, Cin, Co)$ edges. Therefore, the second value is equal to the occurrence probability of $P(Norm, Cin, Co)$ edges of the ith single-bit adder. The amount row 1 contribute to $EP_P(Ab0, Co, i)$ can be calculated as Eq. (20).

$$EP_P(Ab0, Co, i-1) \times CRNUM_P(Norm, Cin, Co, i). \quad (20)$$

TABLE 8
Interactions That Lead to a Positive Abnormal0 Children With Carry-Out of 1

♯	Parent	Edge
1	$Positive(Ab0, Co, i-1)$	$Positive(Norm, Cin, Co)$
2	$Negative(Ab0, \overline{Co}, i-1)$	$Positive(Ab1, \overline{Cin}, Co)$
3	$Zero(\overline{Co}, i-1)$	$Positive(Ab0, \overline{Cin}, Co)$
4	$Zero(Co, i-1)$	$Positive(Ab0, Cin, Co)$

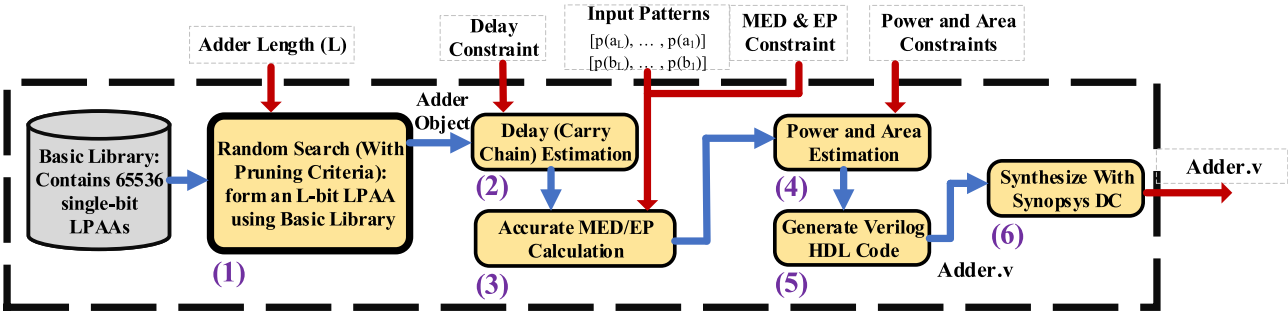


Fig. 11. Block diagram flow of the implemented framework, AxMAP.

Similarly, for rows 2, 3, and 4 of Table 8, we need to multiply the probability of the parent to the probability of its specific edge. Eqs. (21), (22), and (23), shows the amounts rows 2, 3, and 4 contribute to $EP_P(Ab0, Co, i)$, respectively.

$$EP_N(Ab0, \overline{Co}, i-1) \times CRNUM_P(Ab1, \overline{Cin}, Co, i) \quad (21)$$

$$EP_Z(\overline{Co}, i-1) \times CRNUM_P(Ab0, \overline{Cin}, Co, i) \quad (22)$$

$$EP_Z(Co, i-1) \times CRNUM_P(Ab0, Cin, Co, i). \quad (23)$$

By applying the same method to all other 13 portions and adding them together, we can calculate EP with $O(n)$. Thus, for MED calculation, at first, we calculate all portions of EP with $O(n)$. Then, we use EP portions to compute MED, again with $O(n)$.

4 Evaluations

In this section we evaluate the proposed framework for generating approximate circuits automatically. Four experiments have been conducted to thoroughly demonstrate the benefits of the proposed framework. The first experiment shows the speed-up of our proposed method for MED and EP calculation over existing Monte Carlo sampling methods. The second experiment demonstrates the benefit of fast error metric calculation in automatic generation tools dealing with a huge design space. Since our framework can explore the design space much faster than the other ones employing the existing Monte Carlo sampling methods.

In the third experiments, LPAAs are automatically generated in an input pattern-aware manner for a given image processing application that achieves better results in circuit and image processing metrics than the state-of-the-art approximate adders. To demonstrate the generality of the results, the fourth experiment evaluates the generated adders of the third experiment over a wider range of input images. Also, the adders are evaluated in an edge detection application. To conduct fair comparisons between the baseline and AxMAP, all programming and simulations are carried out in MATLAB. Circuit metrics are evaluated using Verilog description alongside with the well-known Synopsys Design Compiler synthesis tool using the NanGate FreePDK45nm library [25].

4.1 AxMAP Methodology

To demonstrate the superiority of the proposed formulae over the MC sampling method, we develop AxMAP that automatically generates efficient application-specific and input-aware approximate adders. AxMAP serves as a framework for evaluating the potentials of input-aware error calculation in the quality of result (QoR). The AxMAP tool takes five input parameters. Power, area, and delay are considered as the designers circuit budget constraints. The tolerable MED over a given pair of input patterns is considered as the error constraint. AxMAP explores the design space and generates approximate adders using the random search algorithm. In each iteration, the selected design satisfying the constraints is considered as a valid output.

Fig. 11 shows the input parameters and the proposed basic building blocks for AxMAP. The input parameters to AxMAP are as follow:

1) *Adder Length (L)*. The bit-width of adders to be generated by AxMAP.
2) *Delay Constraint*. A natural number referring to the maximum length of the carry chain allowed for the adders generated by AxMAP.
3) *Input Patterns*. Two L-bit sequences indicating the bit probability distributions of the two inputs of the adders generated by AxMAP.
4) *MED and EP constraints*. Two real numbers specifying the maximum amount of error allowed for a specific application or input patterns.
5) *Power and area constraints*. Two real numbers representing the maximum amount of the power consumption and area occupation of the adders generated by AxMAP.

As illustrated in Fig. 11, AxMAP employs six basic building blocks and a *Basic Library* that contains all possible (i.e., 65536) single-bit approximate adders. During the first stage, AxMAP performs a *Random Search* for selecting L single-bit samples from the Basic Library to form an L-bit adder object. *Delay Estimation* is performed in the second stage, where the longest carry chain of the adder object is compared with the input delay constraint. If the latency of carry chain is less than or equal to the input delay constraint, the adder object is passed to the next stage. Otherwise, the tool restarts from the first stage, and it attempts to form another L-bit adder object. We consider a user-defined limit for the total number of restarts in AxMAP. The limit is necessary because of two reasons: (1) an exhaustive search amongst all potential designs is extremely time-consuming and (2) exceedingly optimistic bounds for the power, area, delay, and MED requirements of a design may be impractical within the available technology nodes, thereby making the search impossible to converge. In all our experiments, this

TABLE 9
Comparing the Speed-Up Runtime Execution of AxMAP Over the Baseline for 8bit to 128bit Wide LPAAs

Length	8	16	24	32	48	64	128
AxMAP (s)	4.09	9.06	10.79	15.39.	22.99	30.35	62.12
Baseline (min)	13.45	24.7	32.3	45	65	77	170
Speed-up	197	163	183	175	169	151	164

restart limit is set to 100,000 that forces AxMAP to conclude after 100,000 iterations.

An *Accurate MED/EP Calculation* is carried out during the third stage. The MED (or EP) of the adder object is calculated on a given pair of input patterns. Then, the calculated MED is compared with the MED constraint input. If the calculated MED is not less than the MED constraint, AxMAP returns the first stage. Otherwise, it proceeds to the fourth stage for *Power and Area Estimation* by extracting the type and number of gates used for the adder object. If the power or area is more than what user demands, AxMAP returns to the first stage for a new design attempt. The fifth stage is employed to *Generate Verilog HDL Code* for the adder object using a gate-level circuit description stored in a file called adder.v. In the sixth stage, the generated Verilog file is passed to the *Design Compiler* for synthesis and a more accurate estimation of circuit characteristics. If the generated adder still satisfies all of the input circuit constraints, it will be considered as a valid output adder. Therefore, AxMAP records the generated Verilog file along with its circuit and error characteristics. If the circuit constraints are not satisfied, the tool starts the design process over from the first stage to form a new L-bit LPAA.

4.2 Experiment 1: Speedup Over the Existing Methods

Often, in automatic approximate circuit (e.g., adders) generation tools, calculating error metrics takes much more time than that of the circuit and electrical parameters (i.e., power, area and the delay). Thus, a fast error computation method makes the tool much more efficient. Moreover, as the size of generated adders grow, the MC method needs more samples to estimate the error behavior which slows down the tool. Besides, the results that are calculated using the MC are just estimates while accurate evaluations are indeed more reliable.

For the first experiment, we generate seven hundred approximate adders with variable configurations and bit-widths, from 8 to 128-bit (one hundred for each bit-width) wide for AxMAP and the baseline (the baseline is similar to AxMAP but it employs the MC method for MED calculations). The MED of each one of the adders is computed with the MC (with 10,000 samples) and the proposed formula over uniform and a randomly generated pair of input patterns. Table 9 shows that AxMAP, while being entirely accurate, is able to calculate the MED over 150x faster than the baseline.

To further clarify the performance of the proposed approach, Fig. 12 present its computational cost scalability. We generate more than 150 approximate adders with different bit-width (ten different adders for each bit-width). Then, we calculate their MED and report their corresponding average runtime. Fig. 12a shows the average runtime of MED calculation of one adder in the range 4 to 12 bits, with

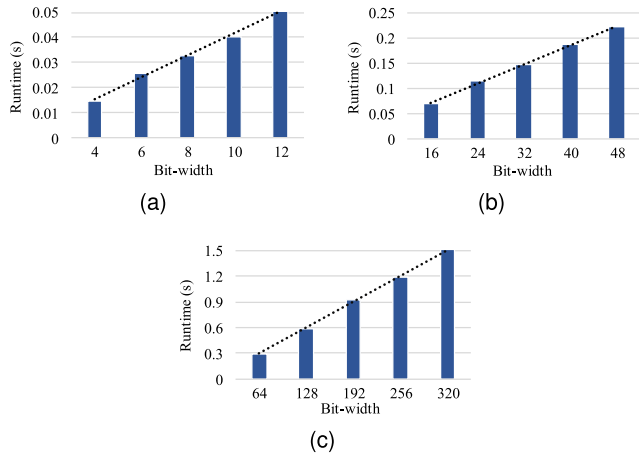


Fig. 12. Cost-scalability analysis. (a) 4 to 12 bits; (b) 16 to 48 bits; (c) 64 to 320 bits.

an increment step of 2. As shown in the figure, the runtime of the proposed approach grows linearly with respect to the size of the adder, which supports the claim about the linear time complexity of MED calculation. As shown in Fig. 12a, a 2-bit growth in the length of adders adds almost 0.01 seconds to the MED computation runtime. Figs. 12b and 12c demonstrate the MED runtime of adders in range 16 to 48 and 64 to 320, with steps of 16 and 64, respectively. Based on Figs. 12b and 12c, a 16-bit and a 64-bit growth adds 0.05 and 0.3 seconds to the runtime, respectively.

4.3 Experiment 2: More Efficient Design Space Exploration

As mentioned in Section 3, the design space of approximate adders is enormous which makes automatic design space exploration tools more crucial than ever. Moreover, an automatic circuit generation tool equipped with a fast quantitative error analysis method can find the desired circuit much faster than the regular ones relying on the MC method-based error analysis. Thus, to illustrate the efficiency that the proposed method grants the automatic adder generation tool (AxMAP), this experiments focuses on exploring the design space of approximate adders to find suitable and efficient approximate adders regarding MED-Power and MED-Area trade-off.

In [26] which is the most complete study in terms of comparing and evaluating the well-known approximate adders, LOA is considered to be one of the best designs, especially when it comes to hardware efficiency (i.e., power consumption and die area occupation). In [27] by A. Najafi *et al.* which is another comprehensive work in terms of comparing various approximate adders from a refreshing view point, again, LOA is chosen as the best design. Therefore, in our experiments, we choose a set of circuit (power and area) and error characteristics slightly lower than LOA as the constraints of AxMAP. In this case, the outputs of AxMAP surely outperform LOA at least in two or three of the metrics. During the configuration process of AxMAP, we consider the delay constraint equal to delay of a single-bit full adder. Therefore, in the delay-related worst case scenario, the output design delay is at least slightly less than the delay of a single-bit full adder. In less than 12 hours, in a

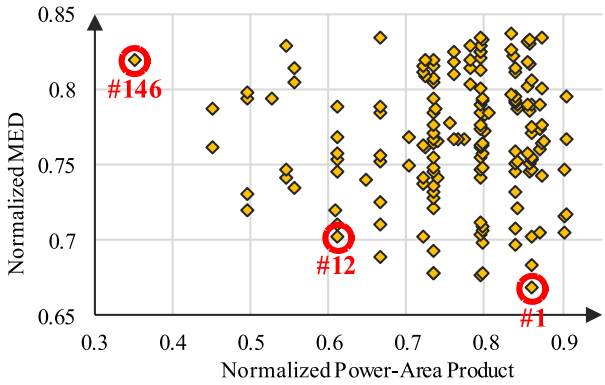


Fig. 13. MED power-area product of generated adders normalized based on those of LOA for uniform input patterns.

Corei5 2.5 GHz processor and 8 GB of RAM, for uniform input patterns, AxMAP was able to generate more than 150 8-bit approximate adders that perform better than LOA in terms of MED, area, and power consumption, only using random searching, which clearly shows the benefit of the proposed fast error analysis method alongside the fact that the design space is filled with useful approximate adders that has never been found.

Fig. 13 demonstrates the MED and power-area product diagram of all generated adders of the second experiment. Both MED and power-area product are normalized based on those of LOA. Compared to LOA, all adders demonstrate better performance in terms of MED and power-area product. Three Pareto optimal structures, with IDs #1, #12, and #146, are chosen (distinguished with a red dashed oval in Fig. 13) as the best designs. Fig. 14 shows the gate-level implementations of these designs (since this experiment was performed over uniform inputs the chosen designs names start with U_ followed by their IDs). Out of three chosen designs, U_1 has the best accuracy with 33 percent better MED than LOA; and U_146 is the most power-area efficient demonstrating 65 percent improvements when compared with LOA. U_12 is a trade-off point between U_1 and U_146; and it shows 30 percent better MED, and 39 percent less power-area, compared to LOA.

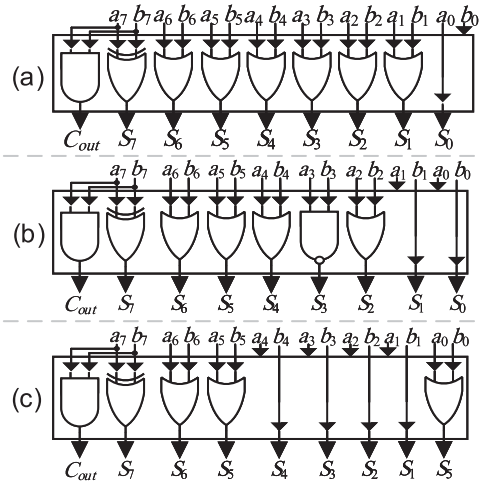


Fig. 14. Gate-level implementation of the chosen designs. a) U_1; b) U_12; c) U_146.

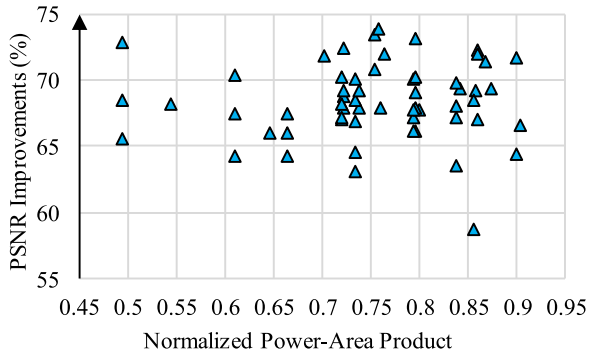


Fig. 15. Normalized MED-power-area.

4.4 Experiment 3: Input Pattern-Aware Automatic Generation of LPAAs for Image Addition

Usually, the benefits of employing approximate units are investigated on a real-life benchmark, e.g., an image processing application, such as many of the previous well-known papers including [10], [15], and [8]. Moreover, based on the motivational example given in Section 2.2.2, when dealing with different applications, the impact of input patterns should be taken into considerations. Therefore, the third experiment aims at using the first two benefits of the proposed method (i.e., speed up gains and efficient design space exploration) to generate energy-area-efficient approximate adders in an image addition problem. In this experiment, more than two hundred different images were randomly selected from ImageNet [28] database with the keywords "Nature", "Natural", "Plants", "Flowers", "Food", "Vegetables", and "Salad".

Similar to the second experiment, power and area constraints of AxMAP are set to an amount slightly lower than those of LOA. The delay is set the delay of a single-bit full adder. The pair of input patterns is set to the average pair of input patterns of all two hundred images from the selected images of ImageNet database. This average pair of input patterns is called "Centroid" (for this experiment, only two images ,with input patterns similar to the Centroid, were selected for all adders). So, the outputs which perform the best on the addition of the specified input pattern candidate of the two hundred images is chosen, unlike the second experiment in which AxMAP was looking for near optimal adders on uniform input patterns. After about 12 hours, AxMAP was able to generate more than 50 adders that are entirely superior to LOA regarding both circuit, power consumption, area occupation, and application dictated metrics, i.e., MED, PSNR (Peak Signal to Noise Ratio), MSE (Mean Squared Error), and MAE (Mean Absolute Error) for all the given pair of input patterns.

The trade-off between the normalized power-area products and the PSNR improvements of the generated adders over LOA is depicted in Fig. 15. As shown in the figure, all adders have better performance over LOA, concerning their output PSNRs, ranging from 73 to 81 percent. Also, Fig. 15 shows reductions in power-area products of outputs over LOA, in range 10 to 50 percent. To clarify more, it should be mentioned that in a single metric comparison, each of the adders are superior to LOA in terms of power, area, and MED. Since the output designs have better MED when operating on the given input patterns, it is somehow expected to perform better concerning several image metrics such as the

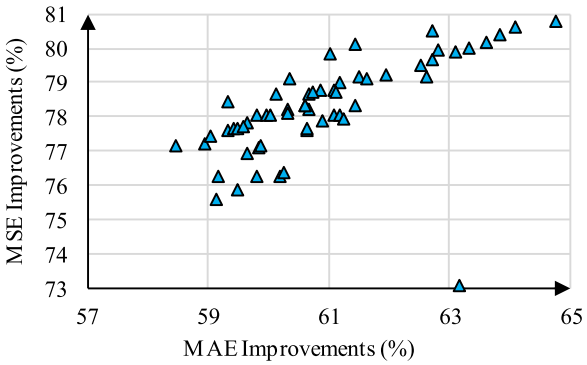


Fig. 16. MSE and MAE improvements.

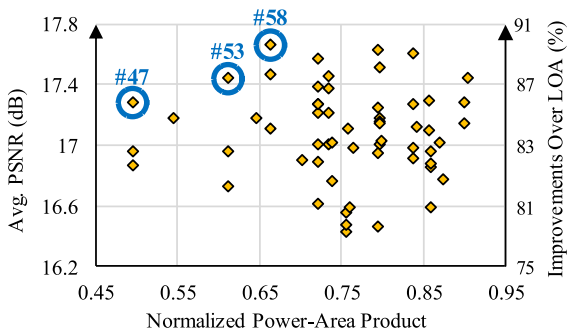


Fig. 17. Average PSNR and normalized power-area product.

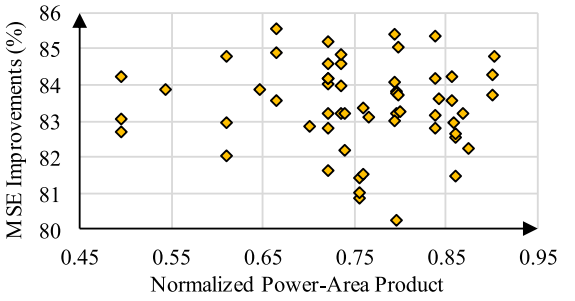


Fig. 18. MSE improvements and power-area product.

MSE and MAE, that their numerical computation is rather MED-like [15]. As can be seen in Fig. 16 the MSE and MAE of the generated adders are at least 73 and 58 percent better than those of LOA, respectively.

4.5 Experiment 4: Testing the Input-Aware Generated Adders on Different Images

The previous experiment showed the benefits generated adders performance regarding circuit characteristics and all the error characteristics for a specified pair of input patterns. However, to show the specialized adders performance in image addition problem, a much more comprehensive comparison has to be accomplished. Therefore, this experiment aims on comparing the image related metrics with those of LOA, when they both employed for a wider range of input images, instead of comparing them for a single image addition problem over the Centroid input pattern. For this experiment, for each of the adders, we randomly choose ten different pair of images from the two hundred images of

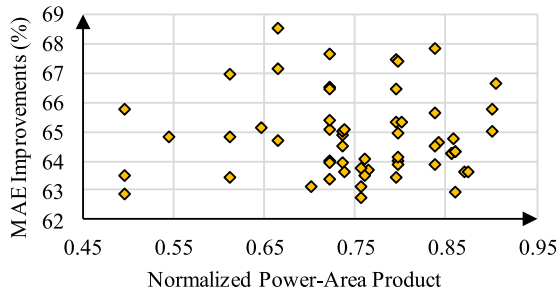


Fig. 19. MAE improvements and power-area product.

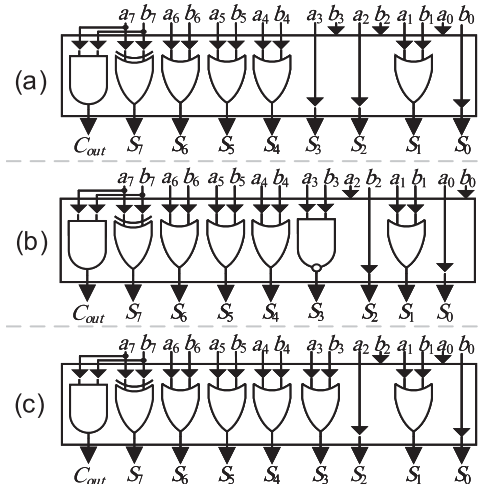


Fig. 20. Gate-level implementation of the chosen designs. (a) N_47; (b) N_53; (c) N_58.

ImageNet database and performed image addition for both the generated adders and LOA.

Fig. 17 shows the PSNR and normalized power-area product diagram of generated adders (more than 50 novel adders). As shown in the figure, in some cases, the generated adders have up to 90 percent better PSNR than LOA, while having 33 percent less power-area product (in the best case, the output PSNR is equal to 10.44 (dB) when employing LOA). Figs. 18 and 19 depicts the MSE/MAE power-area product diagrams for output adders. Based on these to figures, the most power area efficient design has almost 50 percent less power area product than LOA, while improving the MSE and MAE 66 and 84 percent, respectively. As shown in Figs. 20, three Pareto optimal structures, with IDs #47, #53, and #57, are chosen as the best designs.

To further demonstrate the benefits of input-aware circuit generation, the outputs are evaluated for an edge detection (i.e., the Sobel filter) application. We replaced the accurate adders in this filter with the proposed approximate adders (i.e., outputs of AxMAP) and with LOA. For each design ID, we randomly selected an image, and we calculated its corresponding PSNR, MSE, and MAE. Figs. 21a, 21b, and 21c show the PSNR, MSE, and MAE of filtered outputs when implemented using the proposed adders and LOA. Based on Fig. 21a, concerning the PSNR metric, the proposed adders perform better than LOA ranging from 18 percent (for design #18) to 85 percent (for design #25). Based on Figs. 21b and 21c, when the filter is implemented with the proposed adders the MSE and MAE are reduced, compared to the

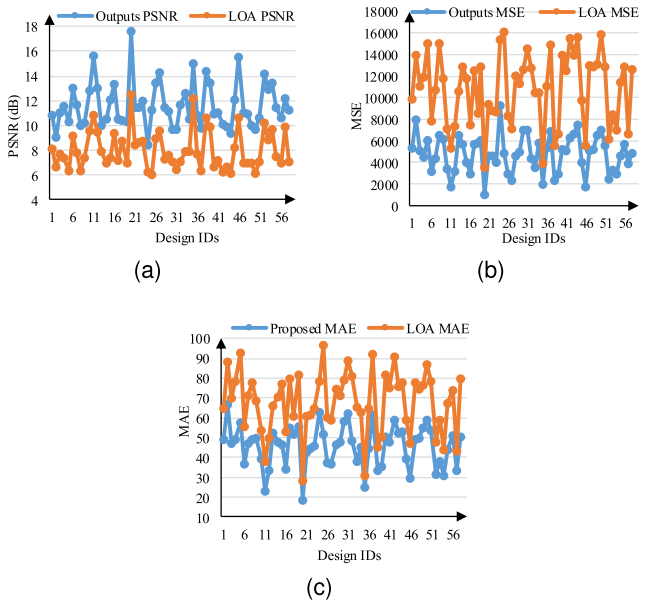


Fig. 21. Comparison of the output designs with LOA in the Sobel filter. (a) PSNR; (b) MSE; (c) MAE.

LOA-based filter. In worst case, for design #18, the MSE and MAE are 31 and 15 percent less than those of LOA-based filter. In the best case, for design #25, the outputs has 69 and 46 percent lower MSE and MAE.

5 Conclusion

This paper examined a new method for calculating the error metrics (MED and EP) alongside a new framework for automatic generation and design space exploration of energy-area-efficient high speed approximate multi-bit adders. The method for error computation demonstrated more than 150x speed up in comparison with the existing Monte Carlo method. Furthermore, the method was able to accurately compute the MED and EP of adders for any given pair of input patterns. For automatic generation of adder circuits, the paper has presented a publicly available[1] toolchain called AxMAP, that randomly explores the design space and outputs the design with higher accuracy and circuit efficiency regarding state-of-the-art designs.

References

[1] S. Mazahir, O. Hasan, R. Hafiz, M. Shafique, and J. Henkel, "Probabilistic error modeling for approximate adders," *IEEE Trans. Comput.*, vol. 69, no. 3, pp. 515–530, Mar. 2017.

[2] S. Tajasob, M. Rezaalipour, M. Dehyadegari, and M. N. Bojnordi, "Designing efficient imprecise adders using multi-bit approximate building blocks," in *Proc. Int. Symp. Low Power Electron. Des.*, 2018, pp. 13:1–13:6.

[3] T. Yang, T. Ukezono, and T. Sato, "A low-power configurable adder for approximate applications," in *Proc. 19th Int. Symp. Qual. Electron. Des.*, 2018, pp. 347–352.

[4] T. Yang, T. Ukezono, and T. Sato, "A low-power yet high-speed configurable adder for approximate computing," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2018, pp. 1–5.

[5] M. Osta, A. Ibrahim, H. Chible, and M. Valle, "Inexact arithmetic circuits for energy efficient IoT sensors data processing," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2018, pp. 1–4.

[6] W. Xu, S. S. Sapatnekar, and J. Hu, "A simple yet efficient accuracy-configurable adder design," *IEEE Trans. Very Large Scale Int. (VLSI) Syst.*, vol. 26, no. 6, pp. 1112–1125, Jun. 2018.

[7] H. A. F. Almurib, T. N. Kumar, and F. Lombardi, "Inexact designs for approximate low power addition by cell replacement," in *Proc. Des. Autom. Test Europe Conf. Exhibit.*, 2016, pp. 660–665.

[8] Z. Yang, J. Han, and F. Lombardi, "Transmission gate-based approximate adders for inexact computing," in *Proc. IEEE/ACM Int. Symp. Nanoscale Archit.*, 2015, pp. 145–150.

[9] M. Shafique, W. Ahmad, R. Hafiz, and J. Henkel, "A low latency generic accuracy configurable adder," in *Proc. 52nd ACM/EDAC/IEEE Des. Autom. Conf.*, 2015, pp. 1–6.

[10] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy, "Low-power digital signal processing using approximate adders," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 32, no. 1, pp. 124–137, Jun. 2013.

[11] Z. Yang, A. Jain, J. Liang, J. Han, and F. Lombardi, "Approximate XOR/XNOR-based adders for inexact computing," in *Proc. 13th IEEE Int. Conf. Nanotechnol.*, 2013, pp. 690–693.

[12] A. B. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *Proc. Des. Autom. Conf.*, 2012, pp. 820–825.

[13] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas, "Bio-inspired imprecise computational blocks for efficient VLSI implementation of soft-computing applications," *IEEE Trans. Circuits Syst. I: Regular Papers*, vol. 57, no. 4, pp. 850–862, Apr. 2010.

[14] S. Dutt, S. Dash, S. Nandi, and G. Trivedi, "Analysis, modeling and optimization of equal segment based approximate adders," *IEEE Trans. Comput.*, vol. 68, no. 3, pp. 314–330, Mar. 2019.

[15] C. Liu, J. Han, and F. Lombardi, "An analytical framework for evaluating the error characteristics of approximate adders," *IEEE Trans. Comput.*, vol. 64, no. 5, pp. 1268–1281, May 2015.

[16] J. Liang, J. Han, and F. Lombardi, "New metrics for the reliability of approximate and probabilistic adders," *IEEE Trans. Comput.*, vol. 62, no. 9, pp. 1760–1771, Sep. 2013.

[17] J. Liang, J. Han, and F. Lombardi, "On the reliable performance of sequential adders for soft computing," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst.*, 2011, pp. 3–10.

[18] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "SALSA: Systematic logic synthesis of approximate circuits," in *Proc. Des. Autom. Conf.*, 2012, pp. 796–801.

[19] Y. Wu and W. Qian, "An efficient method for multi-level approximate logic synthesis under error rate constraint," in *Proc. 53nd ACM/EDAC/IEEE Des. Autom. Conf.*, 2016, pp. 1–6.

[20] A. S. Roy and A. S. Dhar, "A novel approach for fast and accurate mean error distance computation in approximate adders," *IEEE Int. Symp. Circuits Syst.*, pp. 1–5, 2018.

[21] Y. Wu, Y. Li, X. Ge, Y. Gao, and W. Qian, "An efficient method for calculating the error statistics of block-based approximate adders," *IEEE Trans. Comput.*, vol. 68, no. 1, pp. 21–38, Jan. 2019.

[22] M. K. Ayub, O. Hasan, and M. Shafique, "Statistical error analysis for low power approximate adders," in *Proc. 54th ACM/EDAC/IEEE Des. Autom. Conf.*, 2017, pp. 1–6.

[23] L. Li and H. Zhou, "On error modeling and analysis of approximate adders," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2014, pp. 511–518.

[24] R. P. Grimaldi, *Discrete and Combinatorial Mathematics: An Applied Introduction*, 5th ed., London, U.K.: Pearson, 2003.

[25] NanGate, Inc. NanGate FreePDK45 Open Cell Library. 2011. [Online]. Available: https://www.silvaco.com/products/nangate/Library_Creator_Platform/index.html?page_id=2325

[26] H. Jiang, J. Han, and F. Lombardi, "A comparative review and evaluation of approximate adders," in *Proc. 25th Edition Great Lakes Symp. VLSI*, 2015, pp. 343–348.

[27] A. Najafi, M. Weisbrich, G. Payá-Vayá, and A. Garcia-Ortiz, "A fair comparison of adders in stochastic regime," in *Proc. 27th Int. Symp. Power Timing Model. Optim. Simul.*, 2017, pp. 1–6.

[28] J. Deng, W. Dong, R. Socher, L. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.

1. https://github.com/mohrez86/AxMAP

Morteza Rezaalipour received the MSc degree in computer systems architecture engineering from K. N. Toosi University of Technology, Tehran, Iran, in 2019, under the supervision of professor M. Dehyadegari, where he is currently a researcher. His research interests include approximate computing, low-power circuits design, and computer architecture.

Mohammad Rezaalipour received the MSc degree in computer engineering–software from Shahid Beheshti University, Tehran, Iran. He is currently working toward the PhD degree in computer scinece at the Software Institute, Faculty of Informatics, Università della Svizzera italiana (USI). His research interests include automated program repair and software testing.

Masoud Dehyadegari received the PhD degree from the University of Tehran, Tehran, Iran, in 2013 in computer engineering. He is currently an assistant professor of School of Computer Engineering with the K. N. Toosi University of Technology, Tehran, Iran. Form September 2011 until December 2012, he was a visiting scholar in University of Bologna, Italy. His research interests include low-power system design, network-on-chips, and multi-processor system-on-chip.

Mahdi Nazm Bojnordi received the PhD degree in electrical and computer engineering from the University of Rochester, Rochester, New York, in 2016. He is currently an assistant professor with the School of Computing, University of Utah, Salt Lake City, Utah, where he leads the Energy-Efficient Computer Architecture Laboratory. His current research interests include energy-efficient architectures, low-power memory systems, and the application of emerging memory technologies to computer systems. He received the two IEEE micro top picks awards, the HPCA 2016 Distinguished Paper Award, and the Samsung Best Paper Award for his research.

A Deep Reinforcement Learning Based Offloading Game in Edge Computing

Yufeng Zhan, Song Guo, *Fellow, IEEE*, Peng Li, *Member, IEEE*, and Jiang Zhang

Abstract—Edge computing is a new paradigm to provide strong computing capability at the edge of pervasive radio access networks close to users. A critical research challenge of edge computing is to design an efficient offloading strategy to decide which tasks can be offloaded to edge servers with limited resources. Although many research efforts attempt to address this challenge, they need centralized control, which is not practical because users are rational individuals with interests to maximize their benefits. In this article, we study to design a decentralized algorithm for computation offloading, so that users can independently choose their offloading decisions. Game theory has been applied in the algorithm design. Different from existing work, we address the challenge that users may refuse to expose their information about network bandwidth and preference. Therefore, it requires that our solution should make the offloading decision without such knowledge. We formulate the problem as a partially observable Markov decision process (POMDP), which is solved by a policy gradient deep reinforcement learning (DRL) based approach. Extensive simulation results show that our proposal significantly outperforms existing solutions.

Index Terms—Edge computing, computation offloading, Nash equilibrium, partially observable Markov decision process (POMDP), deep reinforcement learning (DRL)

1 Introduction

As MOBILE phones are gaining enormous popularity, more and more mobile applications, such as face recognition, natural language processing and augmented reality, are emerging and attracting great attention [1], [2], [3]. These mobile applications are typically resource-hungry, demanding intensive computation and high energy consumption, which can hardly be supported by mobile phones with limited computation resources and battery life. To overcome this limitation, a novel computing paradigm, called edge computing, has been proposed as a promising solution [4]. A number of modest-size computing servers have been deployed at the edge of pervasive radio access networks close to users, so that users can offload their computing tasks to these servers with low latency.

Although the edge-based computation offloading approach can significantly augment computation capability of users, developing a comprehensive and reliable edge computing system remains challenging. Edge servers have limited hardware resources. If too many users choose to offload their tasks simultaneously, it would exceed the capacity of edge servers, leading to long task response time. Therefore, it is critical to design an efficient offloading strategy to decide which tasks should be offloaded to edge servers. This problem has been recognized as one of the most critical challenges for edge computing, but most existing work needs centralized control to achieve global optimal performance [5], [6]. Unfortunately, it is not practical to force all users to act according to centralized control because they are individuals with rational choices in computation offloading.

Game theory is a powerful framework to analyze the interactions among multiple players who act in their own interests. It can be used to design decentralized mechanisms, such that no player has the incentive to deviate unilaterally. Thanks to its great promises, game theory has been applied for designing offloading algorithm for edge computing by recent research efforts. For example, Chen *et al.* [7], [8] have designed a decentralized computation offloading game for mobile cloud computing. Jošilo *et al.* [9] have proposed selfish decentralized computation offloading in dense wireless networks where each user can offload its computation to multiple wireless base stations. However, existing work can hardly be applied in practice because of two weaknesses. First, they consider a discrete action model that allows users to choose a limited number of actions. Although this model works well in scenarios with a few users, it cannot handle large-scale problems. A straightforward approach is to add more actions in the problem formulation, but it leads to higher algorithm complexity. Second, existing work has a strong assumption that all users should share their information, e.g., quality of network connection and preference on energy efficiency, so that they can make the best offloading decisions. However, users may be unwilling to expose such personal information due to privacy and security concerns.

In this paper, we study to conquer the above weaknesses by designing an algorithm based on game theory enhanced

- *Y. Zhan and S. Guo are with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong. E-mail: zhanyf1989@gmail.com, song.guo@polyu.edu.cn.*
- *P. Li is with the School of Computer Science and Engineering, The University of Aizu, Aizuwakamatsu 965-8580, Japan. E-mail: pengli@u-aizu.ac.jp.*
- *J. Zhang is with the School of Automation, Beijing Institute of Technology, Beijing 100811, China. E-mail: bitzj2015@outlook.com.*

Manuscript received 11 June 2019; revised 7 Dec. 2019; accepted 15 Jan. 2020. Date of publication 23 Jan. 2020; date of current version 8 May 2020. (Corresponding author: Peng Li.) Recommended for acceptance by L. Chen. Digital Object Identifier no. 10.1109/TC.2020.2969148

by deep reinforcement learning (DRL). Specifically, we consider a number of users who can connect an edge server via multiple access points (e.g., base stations or WiFi routers). Each user can arbitrarily divide its task into smaller subtasks and choose to offload a portion of them to the edge server. A challenge arises because of partial offloading. It makes the model more flexible, but users should choose their actions from a continuous space, which is different from discrete models used by existing work that considers simple offloading decisions, e.g., offloading the whole task or not [10].

We first study a simple scenario that users share their information, e.g., network bandwidth and preference, and design an algorithm that can achieve Nash equilibrium. Based on the insight provided by this algorithm, we then extend our work for scenarios without information sharing. The problem is formulated as a multi-agent partially observable Markov decision process (POMDP). To address the challenges of network dynamics and continuous decision space, we propose a decentralized approach based on deep reinforcement learning (D-DRL) with policy gradient and differential neural computer (DNC). Our approach can effectively learn the optimal offloading policy under high network dynamics in a continuous decision space directly from computation offloading game history without any prior knowledge about system models. It has merits over model-based computation offloading game strategies in that it is model-free and provides a general solution to computation offloading problems. Thus, it can be applied to complex and unpredictable situations where it is difficult to obtain precise system models. Moreover, DNC, which is first used in policy gradient DRL, is capable of remembering past information and inferring the hidden states of observations automatically. By incorporating the DNC into our framework, not only the policy optimization process will be accelerated significantly, but also the users can learn policy when the network is time-varying and uncertain.

The main contributions of this paper are summarized as follows:

- We study the task offloading problem in edge computing and formulate it as a decentralized computation offloading game in each time slot by taking into account both communication and computation cost. We solve this problem by proposing an algorithm that can achieve the Nash equilibrium.
- We study the offloading problem without information sharing and formulate it as a multi-agent POMDP. An algorithm based on DRL and DNC has been proposed to solve this challenging problem.
- Simulation results demonstrate the effectiveness of the proposed scheme by comparing it with state-of-the-art.

The remainder of this paper is organized as follows. In Section 2, we discuss the related works. Section 3 presents the problem description. Section 4 gives the algorithm design for scenarios with information sharing. Section 5 provides the detailed multi-agent reinforcement learning approach for scenarios without information sharing. Finally, Section 6 evaluates the system performance by simulation and Section 8 concludes the paper.

2 Related Work

The edge computing paradigm has attracted considerable attention in both academia and industry over the past several years. Nokia introduced the very first real-world edge computing platform in 2013 [11], in which the computing platform called radio application cloud servers is fully integrated with the Flexi Multiradio base stations. Saguna also introduced their fully virtualized edge computing platform Open-RAN, which can provide an open environment for running third-party edge computing applications [12]. Edge computing has been applied into various scenarios [13], [14].

Many existing work has studied the computation offloading problem from the perspective of a single user. Redenko *et al.* [15] have shown that computation offloading can save energy according to their experimental results. In [16], an optimization scheme for energy-efficient application execution has been proposed on the cloud-assisted mobile application platform. Xian *et al.* [17] have proposed an adaptive timeout scheme for computation offloading to improve the energy saving. There are some works that have investigated the computation offloading problem in the multi-user case. Rodrigues *et al.* [6] have proposed a hybrid method for minimizing service delay in edge computing through virtual machine migration and transmission power control. In [20], an iterative algorithm has been proposed to perform the joint optimization of radio and computational resources for multi-cell edge computing under the budget constraints of latency and power. You *et al.* [21] have studied a centralized offloading framework for a multi-user edge computing system based on TDMA and OFDMA aiming to minimize the user's energy consumption. Lin *et al.* [4] have provided a comprehensive survey on computation offloading toward edge computing.

However, all above works need centralized control, ignoring the interactions among multiple users when they independently determine their computation offloading strategies. Some recent works [7], [8], [22], [23], [24], [25] have modeled users as self-interested game players and proposed decentralized schemes to solve the multi-user computation offloading problems. However, they mainly focus on the computation offloading problems under relatively static environment. In real network environment, due to the time-varying wireless networks, the utility of each user is dynamically changing, and thus the solution of the Nash equilibrium in the static game model may not be reached.

Xiao *et al.* [26] have proposed the multi-user computation offloading problem in time-variant wireless networks, and each user needs to compete the computational resource. A Q-learning based approach has been proposed to achieve the Nash equilibrium of the dynamic computation offloading game. However, the users' decision space is discrete in their model and the proposed approach has high complexity in solving large-scale problems.

It is challenging to achieve Nash equilibrium in the stochastic games in the decentralized and dynamic environment. Multi-agent Nash Q-Learning [27] has been proposed for discrete stochastic game. Lillicrap *et al.* [28] have proposed the DDPG approach for the multi-agent Markov decision process, where the environment is fully observable. Zhan *et al.* [29] has

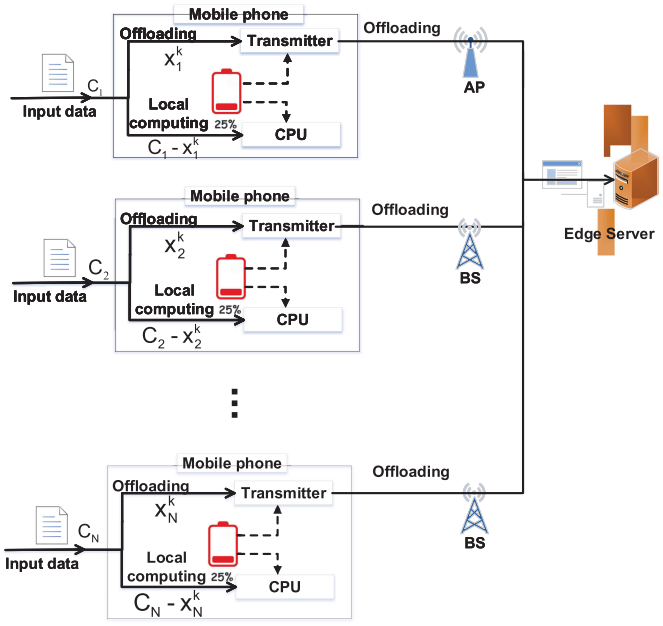


Fig. 1. Multi-user computation offloading with N users in an edge computing environment.

proposed a learning-based incentive mechanism for federated learning, which achieves the Nash equilibrium based on deep reinforcement learning. In contrast to the previous research, our work in this paper formally addresses the problem of partial computation offloading, dynamic environment and incomplete information sharing in edge computing. This is a non-trivial problem due to that each user could only obtain partial observation and thus could not derive the optimal decision.

3 Problem Description

As shown in Fig. 1, we consider a set of $\mathcal{N} = \{1, 2, \ldots, N\}$ users, each of which has a computation-intensive and delay-sensitive task to be executed. There are multiple wireless base stations, through which users can offload their computation tasks to nearby edge servers deployed by the network operator. Following the model in [21], we let C_n denote the size of input data of user n. The user n can offload a part of its computation task to the edge server at time k, which is denoted by x_n^k, and $0 \leq x_n^k \leq C_n$. For example, if user n processes all the computation task locally, then $x_n^k = 0$. On the other hand, we have $x_n^k = C_n$ if the user offloads all its input to the edge server. The network bandwidth assigned to the user n at time k is denoted by $b_n^k \in [\underline{B}_n, \overline{B}_n]$, where $\underline{B}_n$ and $\overline{B}_n$ are the minimum and maximum network bandwidth, respectively. This offloading model can be applied in many applications scenarios, such as malware detection [30]. Malware detection systems explore the feature of the runtime behavior of thousands of applications on mobile devices and involve logging data at each application execution. The application traces have to be scanned in real-time based on the latest malware signature files downloaded from the security database. By offloading the detection tasks to secure servers, edge-based malware detection can reduce the computation of mobile devices. In this scenario, the users can offload any part of the traces to the edge server [21].

The edge server applies algorithms such as deep learning to process the input data. The total amount of input data offloaded from all users to the edge server is $\sum_{n=1}^{N} x_n^k$. Let R denote the total amount of computational resources available at the edge server, which has been normalized to the processing capability of users. The edge server allocates its computation resources to users according to their uploaded data amount, so that user n obtains a portion of $\frac{x_n^k}{\sum_{m=1}^{N} x_m^k} R$ [26], [31]. We use F_n^k to denote the computational capacity of user n in the time slot k. Based on the transmission delay and the processing delay of edge server, the task execution delay of user n in time slot k denoted by t_n^k, is given by

$$t_n^k = \max\left\{\frac{x_n^k}{b_n^k} + \frac{\sum_{m=1}^{N} x_n^k}{R}, \frac{C_n - x_n^k}{F_n^k}\right\}.$$

The transmission cost of user n depends on its transmission bandwidth b_n^k. We let p_n be the transmission cost of unit data for user n. Based on the processing speed of the edge server and the energy cost of the users, the instant utility of user n at time k is defined as

$$u_n(x_n^k, \boldsymbol{x}_{-n}^k) = \frac{\alpha_n x_n^k R}{\sum_{m=1}^{N} x_m^k} - \left(p_n \frac{x_n^k}{b_n^k} + \nu_n(C_n - x_n^k)\right), \tag{1}$$

where $\boldsymbol{x}_{-n}^k = (x_1^k, \ldots, x_{n-1}^k, x_{n+1}^k, \ldots, x_N^k)$ are the decisions of all users except user n. The coefficient α_n is the weight of obtained computational resources at the edge server. To provide rich modeling flexibility and meet user-specific demands, different users could choose different weight parameters in the decision making. For example, when the battery of a user is at a full state, the user would like to put larger weight on the decision making. The ν_n is the energy consumption of processing unit input data by user n. Then, the computational cost for user n to process the local data is $\nu_n(C_n - x_n^k)$ [21].

Given other users' decisions $\boldsymbol{x}_{-n}^k$, the user n would like to make a decision $x_n^k \in [0, C_n]$ to maximize its utility in terms of energy consumption and processing time, i.e.,

$$\begin{aligned} \text{OP}_{\text{User}}: \quad & \max u_n(x_n^k, \boldsymbol{x}_{-n}^k), \\ & s.t.\ x_n^k \in [0, C_n]. \end{aligned} \tag{2}$$

Game theory is a powerful framework to analyze the interactions among multiple users who act in their own interests and design optimal computation offloading scheme, such that no user has the incentive to deviate unilaterally. Our objective is to design an algorithm to achieve the Nash equilibrium that is defined as follows.

Definition 1 (Nash equilibrium). *The strategy set $\boldsymbol{x}^{k,*} = [x_n^{k,*}]_{n\in\mathcal{N}}$ constitutes a Nash equilibrium of the decentralized computation offloading game in time slot k if the following condition is satisfied:*

$$u_n(x_n^{k,*}, \boldsymbol{x}_{-n}^{k,*}) \geq u_n(x_n^k, \boldsymbol{x}_{-n}^{k,*}). \tag{3}$$

The Nash equilibrium has a nice self-stability property such that the user at the equilibrium can achieve a mutually

satisfactory solution and no one has the incentive to deviate. This property is very important to the decentralized computation offloading problem, since the users may act in their own interests.

4 Algorithm With Information Sharing

In this section, we study to design an algorithm for the cases that all mobile users share their complete information including weight α_n, network bandwidth b_n^k and so on. To proceed, we first introduce an important concept called best response [32].

Definition 2. *Given the strategies $\boldsymbol{x}_{-n}^k$ of the other users at time k, user n's strategy $x_n^{k,*}$ is the best response if*

$$u_n(x_n^{k,*}, \boldsymbol{x}_{-n}^k) \geq u_n(x_n^k, \boldsymbol{x}_{-n}^k). \tag{4}$$

According to (3) and (4), all users play the best offloading strategies towards each other at the Nash equilibrium. Based on the concept of best response, we have the following observation.

Lemma 1. *Given $\boldsymbol{x}_{-n}^k$ and the fact that the unit communication cost $\frac{p_n}{b_n}$ is larger than the unit computational cost ν_n, the optimal computation offloading strategy of the nth user is*

$$x_n^{k,*} = \begin{cases} 0, & if\ \ \Delta_1, \\ C_n, & if\ \ \Delta_2, \\ \sqrt{\frac{\alpha_n \sum_{m\neq n}^{N} x_m^k}{\frac{p_n}{b_n^k} - \nu_n}} R - \sum_{m\neq n}^{N} x_m^k, & otherwise, \end{cases} \tag{5}$$

where

$$\Delta_1 : \frac{\alpha_n R}{\frac{p_n}{b_n^k} - \nu_n} < \sum_{m\neq n}^{N} x_m^k, \quad and$$

$$\Delta_2 : {C_n}^2 + \left(\sum_{m\neq n}^{N} x_m^k\right)^2 + \left(2C_n - \frac{\alpha_n R}{\frac{p_n}{b_n^k} - \nu_n}\right) \sum_{m\neq n}^{N} x_m^k < 0,$$

Proof. According to Eqn. (1), the first-order derivative of u_n with respect to x_n^k is

$$\frac{\partial u_n}{\partial x_n^k} = \frac{\alpha_n R}{\sum_{m=1}^{N} x_m^k} - \frac{\alpha_n R x_n^k}{\left(\sum_{m=1}^{N} x_m^k\right)^2} - \frac{p_n}{b_n^k} + \nu_n, \tag{6}$$

and the second-order derivative of u_n with respect to x_n^k is

$$\frac{\partial^2 u_n}{\partial {x_n^k}^2} = \alpha_n \frac{-2R\sum_{m\neq n}^{N} x_m^k}{\left(\sum_{m=1}^{N} x_m^k\right)^3} < 0. \tag{7}$$

Then the utility function u_n is a strictly concave function in x_n^k. Therefore, given any $R > 0$ and any feasible strategy profile of $\boldsymbol{x}_{-n}^k$ of the other users, the optimal computation offloading strategy of user n is unique, if it exits. Setting the first-order derivative of u_n with respect to x_n^k to 0, we have

$$\frac{\partial u_n}{\partial x_n^k} = \frac{\alpha_n R}{\sum_{m=1}^{N} x_m^k} - \frac{\alpha_n R x_n^k}{\left(\sum_{m=1}^{N} x_m^k\right)^2} - \frac{p_n}{b_n^k} + \nu_n = 0. \tag{8}$$

By solving (8), we obtain

$$\tilde{x}_n^k = \sqrt{\frac{\alpha_n \sum_{m\neq n}^{N} x_m^k}{\frac{p_n}{b_n^k} - \nu_n}} R - \sum_{m\neq n}^{N} x_m^k. \tag{9}$$

If $0 < \tilde{x}_n^k < C_n$, we have $x_n^{k,*} = \tilde{x}_n^k$. If $\sqrt{\frac{\alpha_n \sum_{m\neq n}^{N} x_m^k}{\frac{p_n}{b_n^k} - \nu_n}} R - \sum_{m\neq n}^{N} x_m^k < 0$, i.e., Δ_1 holds, we have $x_n^{k,*} = 0$. Otherwise, i.e., Δ_2 holds, we have $x_n^{k,*} = C_n$. □

Based on above Lemma 1, we have the following theorem for users to determine their decisions at Nash equilibrium.

Theorem 1. *With information sharing, the computation offloading strategy of user n at the unique Nash equilibrium satisfies*

$$x_n^{k,*} = \frac{(N-1)R}{\sum_{m=1}^{N} \frac{\frac{p_m}{b_m^k} - \nu_m}{\alpha_m}} \left(1 - \frac{(N-1)\frac{\frac{p_n}{b_n^k} - \nu_n}{\alpha_n}}{\sum_{m=1}^{N} \frac{\frac{p_m}{b_m^k} - \nu_m}{\alpha_m}}\right), \tag{10}$$

if

$$\frac{((N-1)R - C_n)\sum_{m\neq n}^{N} \frac{\frac{p_m}{b_m^k} - \nu_m}{\alpha_m}}{(N-1)(N-2)R + C_n} < \frac{\frac{p_n}{b_n^k} - \nu_n}{\alpha_n} < \frac{\sum_{m\neq n}^{N} \frac{\frac{p_m}{b_m^k} - \nu_m}{\alpha_m}}{N-2}. \tag{11}$$

The utility of user n is

$$u_n(x_n^{k,*}, \boldsymbol{x}_{-n}^{k,*}) = \frac{\alpha_n x_n^{k,*} R}{\sum_{m=1}^{N} x_m^{k,*}} - \left(p_n \frac{x_n^{k,*}}{b_n^k} + \nu_n (C_n - x_n^{k,*})\right),$$

and the computation delay is

$$t_n^k = \max\left\{\frac{x_n^{k,*}}{b_n^k} + \frac{\sum_{m=1}^{N} x_m^{k,*}}{R}, \frac{C_n - x_n^{k,*}}{F_n^k}\right\}.$$

Proof. According Eqn. (5), for $0 < x_n^{k,*} < C_n$, we have

$$\sum_{m=1}^{N} x_m^{k,*} = \sqrt{\frac{\alpha_n \sum_{m\neq n}^{N} x_m^{k,*}}{\frac{p_n}{b_n^k} - \nu_n}} R. \tag{12}$$

Set $\Xi = \sum_{m=1}^{N} x_m^{k,*}$, by Eqn. (12), we have

$$x_n^{k,*} = \Xi - \frac{\Xi^2 \left(\frac{p_n}{b_n^k} - \nu_n\right)}{R\alpha_n}, \tag{13}$$

and therefore

$$\Xi = N\Xi - \Xi^2 \sum_{m=1}^{N} \frac{\frac{p_m}{b_m^k} - \nu_m}{R\alpha_m}. \tag{14}$$

Since $x_n^{k,*} \in (0, C_n)$, hence $\Xi > 0$ has a unique solution for Eqn. (14). Based on Eqn. (14), we obtain the unique solution for Ξ as

$$\Xi = \frac{(N-1)R}{\sum_{m=1}^{N} \frac{\frac{p_m}{b_m^k} - \nu_m}{\alpha_m}}. \tag{15}$$

Plugging the unique Ξ into Eqn. (13), we have the unique solution for user n as

$$x_n^{k,*} = \frac{(N-1)R}{\sum_{m=1}^{N} \frac{\frac{p_m}{b_m^k} - \nu_m}{\alpha_m}} \left(1 - \frac{(N-1)\frac{\frac{p_n}{b_n^k} - \nu_n}{\alpha_n}}{\sum_{m=1}^{N} \frac{\frac{p_m}{b_m^k} - \nu_m}{\alpha_m}}\right). \tag{16}$$

Therefore, the computation offloading game has a unique Nash equilibrium. Since $x_n^{k,*} \in (0, C_n)$, based on Eqn. (16), we can obtain

$$\frac{((N-1)R - C_n)\sum_{m\neq n}^{N} \frac{\frac{p_m}{b_m^k} - \nu_m}{\alpha_m}}{(N-1)(N-2)R + C_n} < \frac{\frac{p_n}{b_n^k} - \nu_n}{\alpha_n} < \frac{\sum_{m\neq n}^{N} \frac{\frac{p_m}{b_m^k} - \nu_m}{\alpha_m}}{(N-2)}.$$

This completes the proof. If (11) holds, we have $x_n^{k,*} \in (0, C_n)$, which satisfies (10). By plugging Eqn. (10), we can obtain the utility $u_n(x_n^{k,*}, \boldsymbol{x}_{-n}^{k,*})$ and computation delay t_n^k of user n under Nash equilibrium. □

Remark 1. According to Theorem 1, the offloading strategy of each user at Nash equilibrium is determined by the computation resource of the edge server, the transmission cost of each user, the radio bandwidth of each user, the local computational cost of each user, and user's preference for delay and energy consumption.

Corollary 1. *With information sharing, if battery power of the nth user is not enough, it will offload less computation to the edge server.*

Proof. As mentioned in Section 3, if user n's battery is not enough, in order to save energy, user n would like to put more weight on energy consumption in the computation offloading decision making (i.e., lower α_n). According to Eqn. (10), we can derive that the first-order derivative of $x_n^{k,*}$ with respect to α_n is

$$\frac{\partial x_n^{k,*}}{\partial \alpha_n} = \frac{(N-2)\frac{\frac{p_n}{b_n^k} - \nu_n}{\alpha_n} \sum_{m=1}^{N} \frac{\frac{p_m}{b_m^k} - \nu_m}{\alpha_m}}{{\alpha_n}^2 (\sum_{m=1}^{N} \frac{\frac{p_m}{b_m^k} - \nu_m}{\alpha_m})^3} + \frac{\sum_{m\neq n}^{N} \frac{\frac{p_m}{b_m^k} - \nu_m}{\alpha_m} - (N-2)\frac{\frac{p_n}{b_n^k} - \nu_n}{\alpha_n}}{{\alpha_n}^2 (\sum_{m=1}^{N} \frac{\frac{p_m}{b_m^k} - \nu_m}{\alpha_m})^3}.$$

Since $x_n^{k,*} > 0$, we can derive $1 - \frac{(N-1)\frac{\frac{p_n}{b_n^k} - \nu_n}{\alpha_n}}{\sum_{m=1}^{N} \frac{\frac{p_m}{b_m^k} - \nu_m}{\alpha_m}} > 0$, i.e., $\sum_{m\neq n}^{N} \frac{\frac{p_m}{b_m^k} - \nu_m}{\alpha_m} - (N-2)\frac{\frac{p_n}{b_n^k} - \nu_n}{\alpha_n} > 0$. Then, we can obtain $\frac{\partial x_n^{k,*}}{\partial \alpha_n} > 0$. Therefore, when user's battery is at a low level, it will reduce the weight α_n and offload less computation to the edge server. □

According to the above analysis, we propose Algorithm 1 that can achieve the Nash equilibrium with complete information sharing. As shown in Algorithm 1, in each time slot, each user first shares its information with other users. After receiving this private information from others, each user decides its offloading strategy according to the optimal computation offloading strategy that can be obtained through Theorem 1.

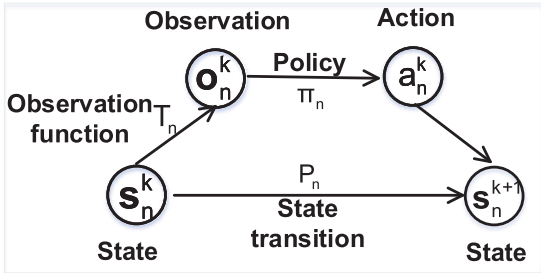


Fig. 2. Partially observable Markov decision process.

Algorithm 1. Algorithm With Complete Information Sharing

1: **for** each time slot k **do**
2: **for** each user $n \in \mathcal{N}$ **do**
3: Publish its private information, i.e., $u_n(x_n^k, \boldsymbol{x}_{-n}^k), p_n, b_n^k, \nu_n, \alpha_n$.
4: Collect information from other users.
5: Calculate the optimal computation offloading strategy according to Theorem 1.
6: **end for**
7: **end for**

5 Algorithm Without Information Sharing

In scenarios with complete information sharing, each user requires the knowledge of other users' information such as the radio bandwidth b_n^k, the decision making weights α_n and so on. However, it is unrealistic to obtain these information in practice because users may refuse to expose these parameters due to the consideration of privacy protection. Furthermore, the physical parameters of users are time-variant, making it challenging for one user to estimate other users' properties accurately.

In this section, we study the offloading algorithm when the properties of other users are unobservable. Specifically, we formulate the dynamic decentralized computation offloading game as a multi-agent partially observable Markov decision process and design a novel dynamic computation offloading algorithm D-DRL for users based on multi-agent DRL approach. With this algorithm, each user can determine the approximately optimal computation offloading strategy directly from game history without any prior information about other users.

5.1 Partially Observable Markov Decision Process

As demonstrated in Fig. 2, a multi-agent POMDP can be represented by $\mathcal{M}_p = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \mathcal{O}, \mathcal{T} \rangle$, which consists of a state space $\mathcal{S} = \{\mathcal{S}_n = \{s_n^k | \forall k \in \mathbb{N}\}\}_{n\in\mathcal{N}}$, an action space $\mathcal{A} = \{\mathcal{A}_n\}_{n\in\mathcal{N}}$, a state transition probability function set $\mathcal{P} = \{P_n : \mathcal{S}_n \times \mathcal{A} \times \mathcal{S}_n \to [0,1]\}_{n\in\mathcal{N}}$, a reward space $\mathcal{R} = \{\mathcal{R}_n\}_{n\in\mathcal{N}}$, an observation space $\mathcal{O}$, and an observation function set $\mathcal{T} = \{T_n : \mathcal{S}_n \times \mathcal{O}_n \to [0,1]\}_{n\in\mathcal{N}}$. In the POMDP, the state space $\mathcal{S}$ is partially known and the decisions are made from observation space $\mathcal{O}$. Users can only observe the past strategy

set $\{\boldsymbol{x}^{k-L}, \boldsymbol{x}^{k-L+1}, \ldots, \boldsymbol{x}^{k-1}\}$ and their own properties. We establish the POMDP for this game as follows.

Observation Space: There is an observation space $\mathcal{O} = \{\mathcal{O}_n\}_{n\in\mathcal{N}}$, where $\mathcal{O}_n = \{\boldsymbol{o}_n^k | \forall k \in \mathbb{N}\}$ and $\boldsymbol{o}_n^k = [\boldsymbol{x}_{-n}^{k-L}, b_n^{k-L+1}, \ldots, \boldsymbol{x}_{-n}^{k-1}, b_n^k]^T$. Note that $\boldsymbol{x}_{-n}^{k-L}$ is randomly generated when $k \leq L$. At the beginning of time slot k, the observation of user n consists of its radio bandwidth at previous $L-1$ time slots and current time, and the size of input data uploaded by other users at previous L time slots.

Action Space: We have $\mathcal{A} = \{\mathcal{A}_n\}_{n\in\mathcal{N}}$, where $\mathcal{A}_n = \{x_n^k | \forall k \in \mathbb{N}\}$. At the beginning of time slot k, the action of user n is the size of input data x_n^k uploaded to the edge server.

Observation Transition: After all users take actions at time k, the observation of user n will transit into $\boldsymbol{o}_n^{k+1}$ satisfying $\boldsymbol{o}_n^{k+1} \sim \int_{\mathcal{S}_n} T_n(\cdot|s_n) P_n(s_n|s_n^k, \{x_m^k\}_{m\in\mathcal{N}}) \mathrm{d}s_n$. It is noteworthy that transition from b_n^k into b_n^{k+1} is a stochastic process. Hence, the whole observation transition process is stochastic.

Reward Space: There is a reward space $\mathcal{R} = \{\mathcal{R}_n\}_{n\in\mathcal{N}}$, where $\mathcal{R}_n = \{r_n^k | \forall k \in \mathbb{N}\}$ and $r_n^k = u_n(x_n^k, \boldsymbol{x}_{-n}^k)$. After all users take actions at time k, each user will calculate its reward according to its utility function and then start next game.

Multi-Agent Learning Objective: We represent the computation offloading policy of user n parameterized by $\boldsymbol{\theta}_n$ as $\pi_{\boldsymbol{\theta}_n}$, which is defined as $\pi_{\boldsymbol{\theta}_n} : \mathcal{O}_n \times \mathcal{A}_n \rightarrow [0, 1]$. Then, the policy optimization problem for user n is derived as follows

$$\begin{aligned} \boldsymbol{\theta}_n^* &= \arg\max_{\boldsymbol{\theta}_n} L_n(\pi_{\boldsymbol{\theta}_n}) \\ &= \arg\max_{\boldsymbol{\theta}_n} \mathbb{E}\Big[V^{\pi_{\boldsymbol{\theta}_n}}\big(\boldsymbol{o}_n^0\big) | \rho_n^0\Big] \\ &= \arg\max_{\boldsymbol{\theta}_n} \mathbb{E}\Big[Q^{\pi_{\boldsymbol{\theta}_n}}\big(\boldsymbol{o}_n^0, x_n^0\big) | \rho_n^0, \pi_{\boldsymbol{\theta}_n}\Big], \end{aligned} \tag{17}$$

where

$$V^{\pi_{\boldsymbol{\theta}_n}}\big(\boldsymbol{o}_n\big) = \mathbb{E}\big[R_n^k | \boldsymbol{o}_n^k = \boldsymbol{o}_n, \Pi, \mathcal{P}, \mathcal{T}\big], \tag{18}$$

$$Q^{\pi_{\boldsymbol{\theta}_n}}(\boldsymbol{o}_n, x_n) = \mathbb{E}\big[R_n^k | \boldsymbol{o}_n^k = \boldsymbol{o}_n, x_n^k = x_n, \Pi, \mathcal{P}, \mathcal{T}\big], \tag{19}$$

$$R_n^k = \sum_{l=k}^{K} \gamma^{l-k} r_n^k. \tag{20}$$

In above equations, $\Pi = \{\pi_{\boldsymbol{\theta}_n}\}_{n\in\mathcal{N}}$ represents the set of all users' policies, $V^{\pi_{\boldsymbol{\theta}_n}}$ is the value function for observation, $Q^{\pi_{\boldsymbol{\theta}_n}}(\boldsymbol{o}_n, x_n)$ is the value function for observation and action, ρ_n^0 is the initial observation probability distribution of user n, R_n^k is the discounted expected future reward for user n at time slot k, and $\gamma \in [0, 1]$ is the discount factor.

Therefore, after formulating the dynamic decentralized computation offloading game as a multi-agent POMDP, the computation offloading strategies of users can be optimized through a multi-agent policy gradient DRL approach under non-cooperative scenarios.

5.2 Algorithm Design

5.2.1 Overview

As shown in Fig. 3, there are N users, each of which has a module called DRL controller that determines the size of input data uploaded to the edge sever. The design of the DRL controller is illustrated in the bottom part of Fig. 3, where there are an actor network, a critic network, a replay buffer, a utility calculation module, a memory module, a value function optimizer, and a policy optimizer. Specifically, the actor network outputs actions according to observation directly through a multiple fully-connected neural network. The critic network first maps the observation into a feature vector, and then reads features from its memory. By combining these features, the final estimated value of observation can be derived and its memory will be renewed through write operations. The replay buffer will store the past D time slots game records for actor and critic update, after which the buffer will be cleared up.

5.2.2 Actor and Critic Networks Design

In [33], MADDPG was proposed for multi-agent reinforcement learning, which first extended actor-critic framework into multi-agent continuous policy optimization. However, they assumed that agents could share their observations and thus designed a centralized critic for agents, which differed from our problem. Considering that the users are unable to share their observations, we design a decentralized actor and decentralized critic for each user. Denote the actor network for approximating the policy of user n as $\pi_{\boldsymbol{\theta}_n}$ and the critic network for approximating the value function of user n as $V_{\boldsymbol{\omega}_n}$, where $\boldsymbol{\theta}_n$ and $\boldsymbol{\omega}_n$ represent the parameters of actor network and critic network, respectively.

More precisely, the actor network $\pi_{\boldsymbol{\theta}_n}$ is designed as a multi-layer fully-connected neural network, which determines the size of input data x_n^k uploaded by user n from its real-time observation $\boldsymbol{o}_n^k$. The critic network $V_{\boldsymbol{\omega}_n}$ is much more complex, which consists of connected-layer and DNC [34]. DNC is a special recurrent neural network with internal memory module, which is capable of learning and remembering the past hidden states of inputs. With DNC, the observation value estimated by $V_{\boldsymbol{\omega}_n}$ depends on the entire observation history. Previous works such as [35] and [36] have founded that recurrent neural networks are effective for addressing POMDP problems. We also demonstrate that critic network with DNC makes the user achieve faster and better convergence to equilibrium through compared experiments.

5.2.3 Policy Optimization Method

We optimize the continuous policy for each user through policy gradient method, for which the objective function is defined in (17). According to the policy gradient theorem proven in [37] and the trust region policy optimization theory proposed in [38], the policy gradient can be calculated as

$$\begin{aligned} \nabla_{\boldsymbol{\theta}_n} L_n &= \mathbb{E}_{\pi_{\boldsymbol{\theta}_n}, \rho_n^1(\boldsymbol{o}_n)}\Big[\nabla_{\boldsymbol{\theta}_n} \log \pi_{\boldsymbol{\theta}_n}\big(\boldsymbol{o}_n, x_n\big) Q^{\pi_{\boldsymbol{\theta}_n}}\big(\boldsymbol{o}_n, x_n\big)\Big] \\ &= \mathbb{E}_{\pi_{\boldsymbol{\theta}_n}, \rho_n^1(\boldsymbol{o}_n)}\Big[\nabla_{\boldsymbol{\theta}_n} \log \pi_{\boldsymbol{\theta}_n}\big(\boldsymbol{o}_n, x_n\big) A^{\pi_{\boldsymbol{\theta}_n}}\big(\boldsymbol{o}_n, x_n\big)\Big] \\ &\approx \mathbb{E}_{\pi_{\hat{\boldsymbol{\theta}}_n}, \rho_n^1(\boldsymbol{o}_n)}\Big[f_n \nabla_{\boldsymbol{\theta}_n} \log \pi_{\boldsymbol{\theta}_n}\big(\boldsymbol{o}_n, x_n\big) A^{\pi_{\boldsymbol{\theta}_n}}\big(\boldsymbol{o}_n, x_n\big)\Big], \end{aligned} \tag{21}$$

where $f_n = \frac{\pi_{\boldsymbol{\theta}_n}(\boldsymbol{o}_n|x_n)}{\pi_{\hat{\boldsymbol{\theta}}_n}(\boldsymbol{o}_n|x_n)}$, $A^{\pi_{\boldsymbol{\theta}_n}}\big(\boldsymbol{o}_n, x_n\big) = Q^{\pi_{\boldsymbol{\theta}_n}}\big(\boldsymbol{o}_n, x_n\big) - V^{\pi_{\boldsymbol{\theta}_n}}\big(\boldsymbol{o}_n\big)$ is the advantage function for observation and action, $\hat{\boldsymbol{\theta}}_n$

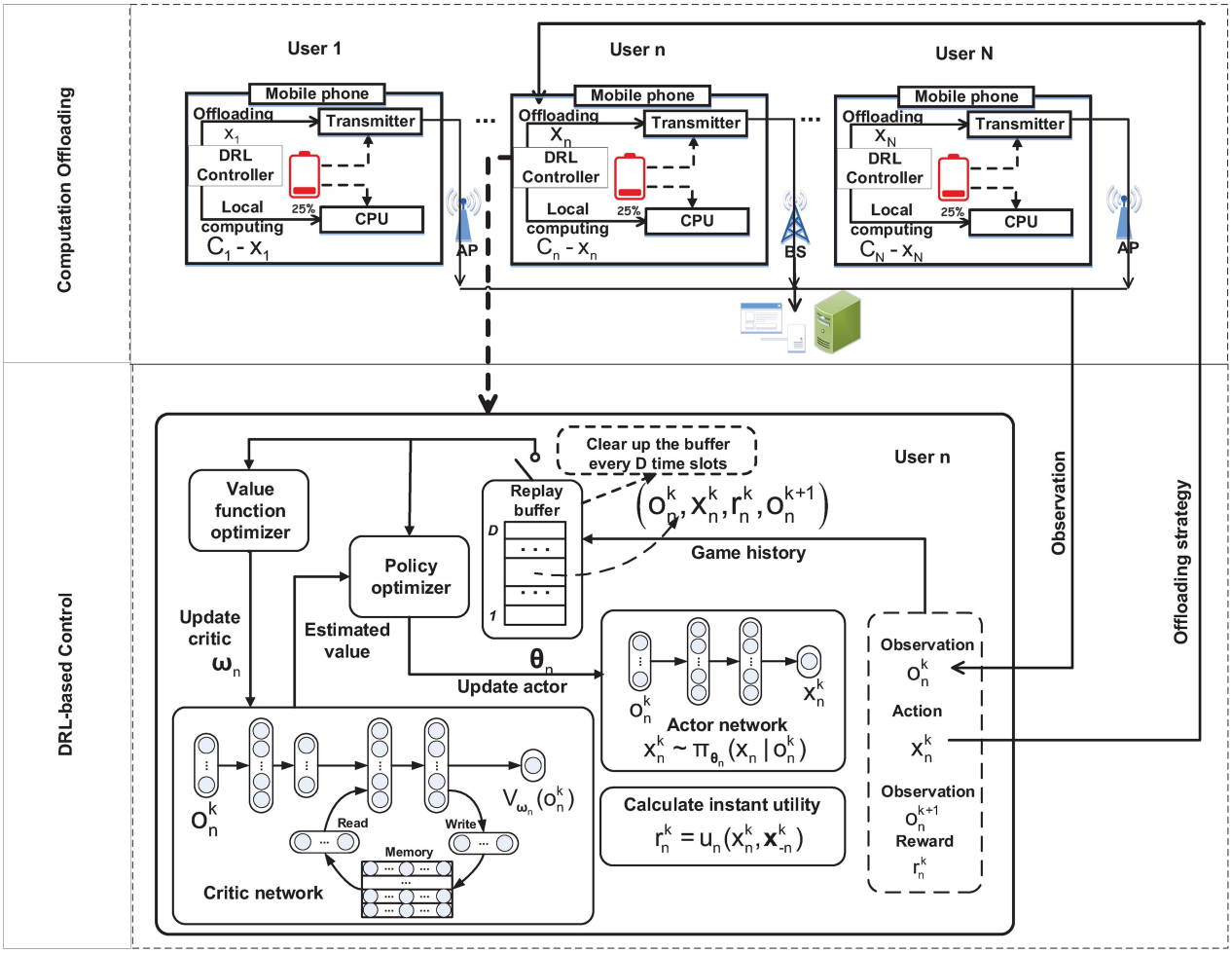


Fig. 3. Dynamic decentralized computation offloading with D-DRL.

represents the parameter of policy for sampling, and $\rho_n^1(\boldsymbol{o}_n)$ is the observation distribution induced by the POMDP. To accelerate the convergence of policy optimization, we adopt the proximal policy optimization (PPO) method proposed in [39], which clips the policy gradient as

$$\nabla_{\boldsymbol{\theta}_n} L_n \approx \mathbb{E}_{\pi_{\hat{\boldsymbol{\theta}}_n}, \rho_n^1(\boldsymbol{o}_n)} \left[\nabla_{\boldsymbol{\theta}_n} \log \pi_{\boldsymbol{\theta}_n}(\boldsymbol{o}_n, x_n) C(\boldsymbol{o}_n, x_n) \right], \tag{22}$$

where

$$C(\boldsymbol{o}_n, x_n) = \min \left[f_n A^{\pi_{\boldsymbol{\theta}_n}}(\boldsymbol{o}_n, x_n), \eta(f_n) A^{\pi_{\boldsymbol{\theta}_n}}(\boldsymbol{o}_n, x_n) \right], \tag{23}$$

$$\eta(x) = \begin{cases} 1+\varepsilon, & x > 1+\varepsilon \\ x, & 1-\varepsilon \le x \le 1+\varepsilon \\ 1-\varepsilon, & x < 1-\varepsilon \end{cases}, \tag{24}$$

and ε is an adjustable parameter.

5.2.4 Updating Laws of Actor and Critic

We define the loss function for updating the critic network of user n as

$$J_n(\boldsymbol{\omega}_n) = \mathbb{E}_{\boldsymbol{o}_n \sim \rho_n^1(\boldsymbol{o}_n)} \Big[-V_{\boldsymbol{\omega}_n}(\boldsymbol{o}_n) + \mathbb{E}_{\boldsymbol{o}_n', x_n} \left[r + V_{\boldsymbol{\omega}_n}(\boldsymbol{o}_n') \right] \Big]^2, \tag{25}$$

where $\boldsymbol{o}_n'$ is the next time observation of $\boldsymbol{o}_n$ for user n. During training process, the estimated gradient about $\boldsymbol{\omega}_n$ is calculated as

$$\nabla_{\boldsymbol{\omega}_n} \hat{J}_n = \frac{1}{D} \sum_{k=0}^{D-1} \left[V_{\boldsymbol{\omega}_n}(\boldsymbol{o}_n^k) - Y_n^k \right] \frac{dV_{\boldsymbol{\omega}_n}(\boldsymbol{o}_n)}{d\boldsymbol{\omega}_n}, \tag{26}$$

where

$$Y_n^k = R_n^k - \gamma^{D-k} R_n(D) + \gamma^{D-k} V_{\boldsymbol{\omega}_n}(\boldsymbol{o}_n(D)), \tag{27}$$

and D is the size of mini-batch for updating critic network. Then, the critic network $V_{\boldsymbol{\omega}_n}$ is updated through mini-batch stochastic gradient descent method as follows

$$\boldsymbol{\omega}_n \leftarrow \boldsymbol{\omega}_n - l_{n,1} \nabla_{\boldsymbol{\omega}_n} \hat{J}_n, \tag{28}$$

where $l_{n,1}$ is the critic learning rate for user n.

Moreover, the estimated gradient about $\boldsymbol{\theta}_n$ is calculated as

$$\nabla_{\boldsymbol{\theta}_n} \hat{L}_n = \frac{1}{D} \sum_{k=0}^{D-1} \nabla_{\boldsymbol{\theta}_n} \log \pi_{\boldsymbol{\theta}_n}(\boldsymbol{o}_n^k, x_n^k) C(\boldsymbol{o}_n^k, x_n^k), \tag{29}$$

where D is the size of mini-batch for updating actor network. And we update the actor network $\pi_{\boldsymbol{\theta}_n}$ by mini-batch stochastic gradient ascent as follows

$$\boldsymbol{\theta}_n \leftarrow \boldsymbol{\theta}_n + l_{n,2} \nabla_{\boldsymbol{\theta}_n} \hat{L}_n, \tag{30}$$

where $l_{n,2}$ is the actor learning rate for user n.

Algorithm 2. D-DRL: Algorithm Without Information Sharing

1: **for** user $n \in \mathcal{N}$ **do**
2: Initialize γ, $l_{n,1}$, $l_{n,2}$, ,$\boldsymbol{\theta}_n$, $\boldsymbol{\omega}_n$, and $\boldsymbol{o}_n^0$.
3: **end for**
4: **for** time slot k in $1, 2, \ldots$ **do**
5: **for** user $n \in \mathcal{N}$ **do**
6: Observe b_n^k and update its observation $\boldsymbol{o}_n^{k-1}$ into $\boldsymbol{o}_n^k$.
7: Store $\{\boldsymbol{o}_n^{k-1}, x_n^{k-1}, \boldsymbol{o}_n^k, r_n^{k-1}\}$ into $\mathcal{D}_n$.
8: Input $\boldsymbol{o}_n^k$ into actor network $\pi_{\boldsymbol{\theta}_n}$ and determine the size of input data x_n^k uploaded to the edge server.
9: Calculate its reward $r_n^k = u_n(x_n^k, \boldsymbol{x}_{-n}^k)$ by (1).
10: **end for**
11: **if** $k\%D == 0$ **then**
12: **for** m in $1, 2, \ldots, M$ **do**
13: **for** user $n \in \mathcal{N}$ **do**
14: Calculate $\nabla_{\boldsymbol{\theta}_n} \hat{L}_n$ and $\nabla_{\boldsymbol{\omega}_n} \hat{J}_n$ via (26) and (29).
15: Update $\boldsymbol{\theta}_n$ and $\boldsymbol{\omega}_n$ through (28) and (30).
16: **end for**
17: **end for**
18: Clear the replay buffer $\mathcal{D}_n$.
19: **end if**
20: **end for**

5.2.5 Algorithm Details

The pseudo codes are shown in Algorithm 2, where each user initializes their observations and the parameters of their actor and critic networks (Line 1-3).

At each time slot k, each user observes its bandwidth and updates its observation (Line 6). Then, each user stores its old observation, strategy, reward and new observation into its experience replay buffer (Line 7). Next, each user takes its current observation as the input of its actor network and then upload a portion of its input data to the edge sever according to the output of the actor network (Line 8). After all users upload their data, each user calculates its instant utility (reward) (Line 9).

Each user updates its actor and critic networks in every D time slots (Line 13). They take the D records in their replay buffers as one mini-batch and estimate the gradients for updating actor and critic networks (Line 14). After that, each user optimizes its actor and critic networks by mini-batch stochastic gradient ascent and descent methods for M times, respectively (Line 15). Finally, they clear their replay buffer (Line 18).

6 Performance Evaluation

6.1 Simulation Settings

Extensive simulations are conducted to evaluate the performance of the proposed algorithms. We consider a set of users, each of which has an amount of input data $C_n \backsim N(1, 0.1)Mb$. These data should be processed in real time and deleted periodically to avoid the storage overflow. We set the user's unit transmission cost as $p_n = 1J/s$, the weight as $\alpha_n = 1$, the user's computation capacity in time slot k as $F_n^k = 0.01Mb/s$ and computation capacity of edge server as $R = 16Mb/s$ by default. Any radio bandwidth that satisfies the constrain in Theorem 1 is reasonable. In our simulation, we set the minimum and maximum network bandwidth as $\frac{1}{12.5}Mbps$ and $\frac{1}{4.5}Mbps$ which satisfies the constrain.

The parameters of the DRL controller are selected through fine-tuning. Specifically, the actor network has two hidden fully-connected layers, each of which contains 200 nodes and 50 nodes, respectively. The critic network also has two hidden fully-connected layers, each of which has 200 nodes and 50 nodes, respectively. The DNC module has three read heads, one write head, and a memory matrix whose size is 10×32. We set $D = 20$, $M = 5$ and $L = 5$ by default. We compared our work with five baseline approaches.[1] In the simulation, the Nash equilibrium (NE) is calculated by Algorithm 1 with complete information sharing. Moreover, as for real world scenario, each user could not get the information of others. Then, the learning-based offloading algorithm has been designed without knowing any prior information to derive the NE.

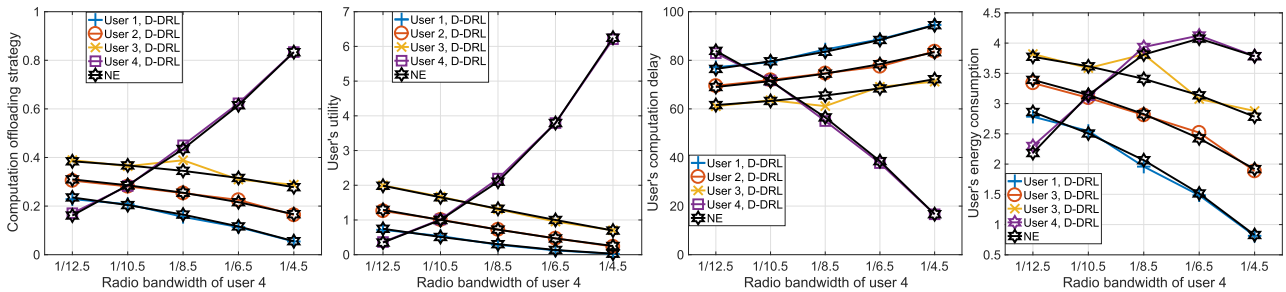
- MAA2C [40]: this is a modified A2C, which is an implementation of A2C for multi-agent environment, called "MAA2C" in this paper.
- MAPPO [41]: this is a modified PPO, which is an implementation of PPO for very large scale competitive multi-agent training, called "MAPPO" in this paper.
- Greedy: it is a heuristic algorithm that greedily chooses a policy with maximum reward from the replay buffer.
- Random: users randomly select the size of input data to offload to the edge server.
- Edge computing by all users (EdgeAll for short): users offload all their computation to the edge server.

6.2 Simulation Results

We first study the convergence of the proposed D-DRL algorithm when there are 4 users. In this simulation, we set $b_1^k = \frac{1}{11.5}$, $b_2^k = \frac{1}{10.5}$, $b_3^k = \frac{1}{9.5}$, $b_4^k = \frac{1}{8.5}$. As shown in Fig. 5, both offloading strategies (i.e., x_n^k) and user utility converge to the NE at about 8000 time slots. The offloading strategies of the four users under Nash equilibrium are 0.165, 0.26, 0.485, and 0.43, respectively.

We then study the influence of network bandwidth by changing user 4's bandwidth (i.e., b_4^k) from $\frac{1}{12.5}$ to $\frac{1}{4.5}$. The bandwidth of other 3 users are set as $b_1^k = \frac{1}{11.5}$, $b_2^k = \frac{1}{10.5}$, and $b_3^k = \frac{1}{9.5}$. As shown in Fig. 4, user 4 increases its size of offloading data and thus obtains a higher utility, under a higher radio bandwidth. For example, user 4 offloads less than 20 percent of input data to the edge when $b_4^k = \frac{1}{12.5}$, leading to a low utility of 0.2. However, when its bandwidth has been improved to $b_4^k = \frac{1}{4.5}$, a higher utility of 6.2 can be obtained by offloading about 81 percent of input data. This is because the unit transmission cost decreases under a larger bandwidth, which motivates users to upload more data to the edge server. In Fig. 4c, we also observe that the user computation delay decreases as the growth of the radio bandwidth. For example, the delay of user 4 decreases by 76.5 percent as the radio bandwidth b_4^k changes from $\frac{1}{12.5}$ to $\frac{1}{4.5}$. This is because

1. The python codes of the numerical experiment are available at https://github.com/bitzj2015/Learning-based-incentive-mechanism-design/tree/master/Edge-DRL



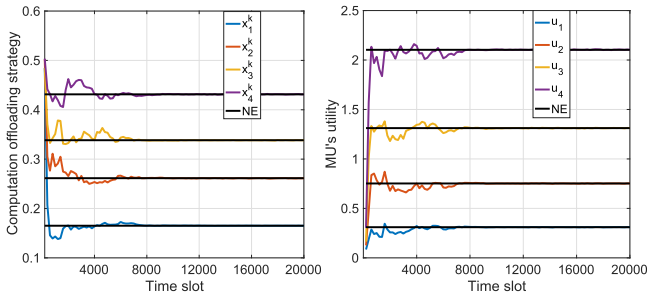
(a) Computation offloading strategy of user. (b) Utility of user. (c) Computation delay of user. (d) Energy consumption of user.

Fig. 4. Performance of D-DRL with four users when varying the radio bandwidth of user 4.

user 4 uploads more data to the edge server during radio bandwidth increasing, leaving less data to be computed locally, thus will reduce the delay of local computation. Fig. 4d shows the energy consumption of each user. We can observe that, user 4 consumes more energy to upload more data to the edge server, in order to decrease the computation delay and thus increase the utility. Since other users offload less data to the edge server with the increase of user 4's radio bandwidth, their energy consumption will reduce. In addition, other users have a performance degradation during the bandwidth increasing of user 4. For example, user 3 has a size of data offloading decreasing by 30 percent, the utility decreasing by 60 percent, and computation delay increasing by 18 percent. This is because other users' competitiveness declined and resulting in computation resource reducing. Based on this analysis, we can derive that when other user's bandwidth increases, user n must upload less data to the edge server.

We study the affect of weight α_n by changing α_4 from 1 to 1.8. The weights of other users are set to $\alpha_1 = 1$, $\alpha_2 = 1$, and $\alpha_3 = 1$. Meanwhile, we fix the bandwidth as $b_1^k = \frac{1}{11.5}$, $b_2^k = \frac{1}{10.5}$, $b_3^k = \frac{1}{9.5}$, and $b_4^k = \frac{1}{8.5}$. As shown in Fig. 6a, we observe that the user 4 with larger weight offloads more data to the edge server, which coincides the statement in Corollary 1. That is because user 4's battery is in high level, as shown in Fig. 6d, it can spend more power to offload more data to the edge server, in order to reduce the computation delay. For example, for user 4, the size of input data offloaded to the edge server increases by 90 percent, while the computation delay decreases by 66 percent, if the weight changes from 1.0 to 1.8.

We compare the performance of five different algorithms under the time-variant network environment, where $b_1^k = \frac{1}{11.5}$, $b_2^k = \frac{1}{10.5}$, $b_3^k = \frac{1}{9.5}$ and b_4^k is randomly chosen among $\{\frac{1}{4.5}, \frac{1}{6.5}, \frac{1}{8.5}, \frac{1}{10.5}, \frac{1}{12.5}\}$. As shown in Fig. 7, our proposed algorithm



(a) Users' offloading strategies. (b) Users' utilities.

Fig. 5. Convergence of D-DRL.

D-DRL significantly exceeds others, by having the highest average utility and the fastest convergence speed. For instance, D-DRL increases the average utility of the users by 2 percent as compared with MAPPO, by 14 percent as compared with MAA2C, by 32.5 percent as compared with Greedy and by 410 percent as compared with Random. Meanwhile, our proposal takes about 8000 time slots to converge to the stable state, and almost no shock. However, MAPPO takes about 14000 time slots and MAA2C takes about 18000 time slots. At the same time, both of them have shocks.

Fig. 8 shows the impact of number of users. In this simulation, in order to show the simulation results more clearly, we set $R = 24Mb$. The average utility of users decreases with the number of users sharing the edge server. That is, although the computation capability of the edge server remains unchanged, it leads to more competitions among users. Therefore, users need to decrease the size of computation to offload to the edge server, resulting in a decline of average utility, as shown in Fig. 8a. At the same time, we can observe that D-DRL has the maximum average utility. When N is greater than 4, MAA2C could not converge to the Nash equilibrium. Fig 8b shows the convergence speed of the three strategies, we see that D-DRL converges to the stable state with the fastest speed.

7 Discussions

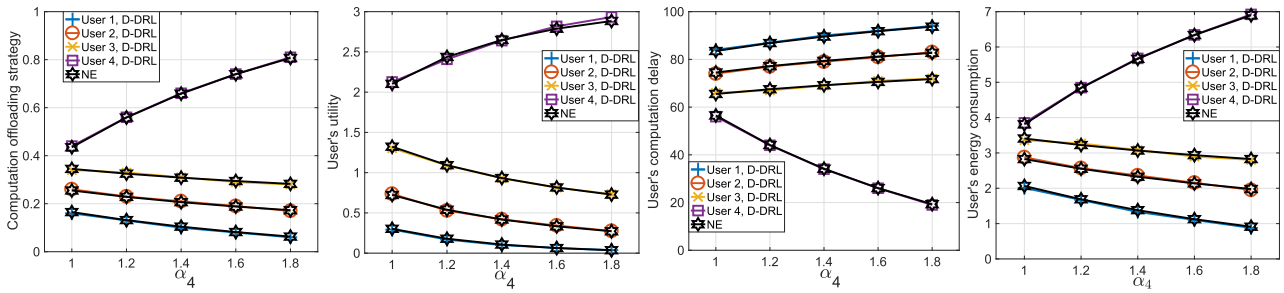
7.1 Time Complexity Analysis

During the D-DRL execution, given the observation information as input, each user utilizes its own actor network π_{θ_n} to generate an action, and thus the computational complexity is merely based on a fully-connected deep neural network. According to [42], the time complexity of a fully-connected deep neural network is determined by the number of multiplication operations, which is $O(\sum_{f=1}^{F} \epsilon_f \cdot \epsilon_{f-1})$, and ϵ_f is the number of neural units in fully-connected layer f. In our design, we use two fully-connected hidden layers in the actor network π_{θ_n} . Meanwhile, since modern mobile devices are becoming stronger and stronger,[2] they can afford the computational overhead incurred by such a kind of actor network.

7.2 Task Offloading Scenario

As for the big data analytics, for example, there are large amounts of pictures to be analyzed. In this case, the volume

2. The Apple has its own Apple A11 Bionic, which is a 64-bit ARM-based SoC. HUAWEI mobile phone has its own state-of-the-art CPU named Kiri970, carried with ARM Mali-G72 MP12 GPU which is an integrated high-end graphics card for ARM-based SoC.



(a) Computation offloading strategy of user. (b) Utility of user. (c) Computation delay of user. (d) Energy consumption of user.

Fig. 6. Performance of D-DRL with four users when varying the weight of user 4.

of pictures can be approximated as a continuous value. Hence, this picture analysis task can be arbitrarily divided into pieces and chosen a portion of them to offload to the edge server. Our model fits this scenario very well. Even for a single picture analysis task, the task is consisted of a large amount of CPU instructions which also can be regarded as a continuous value. Therefore, our model also matches it.

8 Conclusion

In this paper, we consider the dynamic computation offloading decision making problem among users for edge computing under dynamic environment and propose a computation offloading game formulation. We show that the game always admits a unique Nash equilibrium under the certain conditions. We also design a decentralized computation offloading algorithm called "D-DRL" which first combines policy gradient DRL-based approach with DNC that can achieve the optimal offloading strategy. Simulation results demonstrate that the proposed algorithm is much more efficient as compared with the baseline approaches.

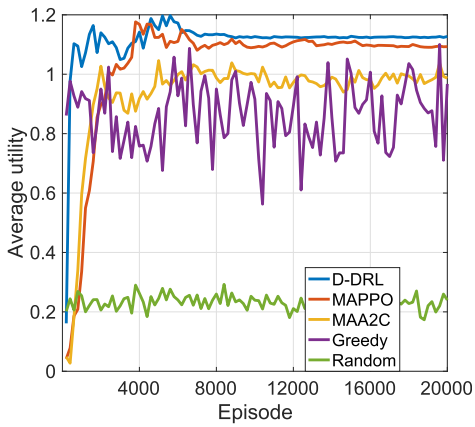
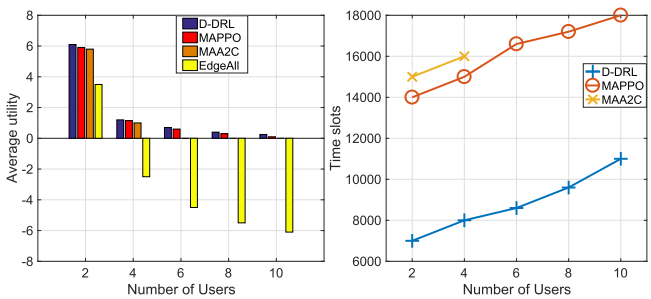


Fig. 7. Average performance with $N = 4$ users in time-variant network environment.



(a) Average utility of users. (b) Average time slots.

Fig. 8. Performance of decentralized computation offloading when varying the number of users.

Acknowledgments

This research was financially supported by the General Research Fund of the Research Grants Council of Hong Kong (PolyU 152221/19E), the National Natural Science Foundation of China (Grant 61872310), and JSPS Grants-in-Aid for Scientific Research JP19K20258.

References

[1] Y. Sun, D. Liang, X. Wang, and X. Tang, "DeepID3: Face recognition with very deep neural networks," 2015, *arXiv:1502.00873*.

[2] T. Soyata, R. Muraleedharan, C. Funai, M. Kwon, and W. Heinzelman, "Cloud-vision: Real-time face recognition using a mobile-cloudlet-cloud acceleration architecture," in *Proc. IEEE Symp. Comput. Commun.*, 2012, pp. 59–66.

[3] A. Maimone, A. Georgiou, and J. S. Kollin, "Holographic near-eye displays for virtual and augmented reality," *ACM Trans. Graph.*, vol. 69, no. 4, 2017, Art. no. 85.

[4] L. Lin, X. Liao, H. Jin, and P. Li, "Computation offloading toward edge computing," *Proc. IEEE*, vol. 107, no. 8, pp. 1584–1607, Aug. 2019.

[5] T. G. Rodrigues, K. Suto, H. Nishiyama, and N. Kato, "Hybrid method for minimizing service delay in edge cloud computing through VM migration and transmission power control," *IEEE Trans. Comput.*, vol. 66, no. 5, pp. 810–819, May 2017.

[6] T. G. Rodrigues, K. Suto, H. Nishiyama, N. Kato, and K. Temma, "Cloudlets activation scheme for scalable mobile edge computing with transmission power control and virtual machine migration," *IEEE Trans. Comput.*, vol. 67, no. 9, pp. 1287–1300, Sep. 2018.

[7] X. Chen, "Decentralized computation offloading game for mobile cloud computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 4, pp. 974–983, Apr. 2015.

[8] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Trans. Netw.*, vol. 24, no. 5, pp. 2795–2808, Oct. 2016.

[9] S. Jošilo and G. Dán, "Selfish decentralized computation offloading for mobile cloud computing in dense wireless networks," *IEEE Trans. Mobile Comput.*, vol. 18, no. 1, pp. 207–220, Jan. 2019.

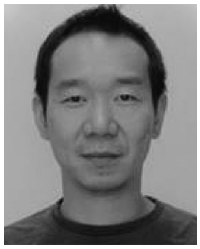
[10] J. Zheng, Y. Cai, Y. Wu, and X. S. Shen, "Dynamic computation offloading for mobile cloud computing: A stochastic game-theoretic approach," *IEEE Trans. Mobile Comput.*, vol. 18, no. 4, pp. 771–786, Apr. 2019.

[11] Intel and N. S. Networks, "Increasing mobile operators value proposition with edge computing," *Tech. Brief*, 2013. [Online]. Available: https://www.intel.co.id/content/dam/www/public/us/en/documents/technology-briefs/edge-computing-tech-brief.pdf

[12] Saguna and Intel, "Using mobile edge computing to improve mobile network performance and profitability," White paper, 2016.

[13] P. Li, X. Wu, W. Shen, W. Tong, and S. Guo, "Collaboration of heterogeneous unmanned vehicles for smart cities," *IEEE Netw.*, vol. 33, no. 4, pp. 133–137, Jul. 2019.
[14] P. Li, T. Miyazaki, K. Wang, S. Guo, and W. Zhuang, "Vehicle-assist resilient information and network system for disaster management," *IEEE Trans. Emerg. Topics Comput.*, vol. 5, no. 3, pp. 438–448, Jul. 2017.
[15] A. Rudenko, P. Reiher, G. J. Popek, and G. H. Kuenning, "Saving portable computer battery power through remote process execution," *ACM SIGMOBILE Mobile Comput. Commun. Rev.*, vol. 2, no. 1, pp. 19–26, 1998.
[16] Y. Wen, W. Zhang, and H. Luo, "Energy-optimal mobile application execution: Taming resource-poor mobile devices with cloud clones," in *Proc. IEEE Int. Conf. Comput. Commun.*, 2012, pp. 2716–2720.
[17] C. Xian, Y.-H. Lu, and Z. Li, "Adaptive computation offloading for energy conservation on battery-powered systems," in *Proc. Int. Conf. Parallel Distrib. Syst.*, 2007, vol. 2, pp. 1–8.
[18] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan, "A framework for partitioning and execution of data stream applications in mobile cloud computing," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 4, pp. 23–32, 2013.
[19] Y. Zhao, S. Zhou, T. Zhao, and Z. Niu, "Energy-efficient task offloading for multiuser mobile cloud computing," in *Proc. IEEE/CIC Int. Conf. Commun. China*, 2015, pp. 1–5.
[20] S. Sardellitti, G. Scutari, and S. Barbarossa, "Joint optimization of radio and computational resources for multicell mobile-edge computing," *IEEE Trans. Signal Inf. Process. Netw.*, vol. 1, no. 2, pp. 89–103, Jun. 2015.
[21] C. You, K. Huang, H. Chae, and B.-H. Kim, "Energy-efficient resource allocation for mobile-edge computation offloading," *IEEE Trans. Wireless Commun.*, vol. 16, no. 3, pp. 1397–1411, Mar. 2017.
[22] Z. Xu, W. Liang, M. Jia, M. Huang, and G. Mao, "Task offloading with network function requirements in a mobile edge-cloud network," *IEEE Trans. Mobile Comput.*, vol. 18, no. 11, pp. 2672–2685, Nov. 2019.
[23] H. Cao and J. Cai, "Distributed multiuser computation offloading for cloudlet-based mobile cloud computing: A game-theoretic machine learning approach," *IEEE Trans. Veh. Technol.*, vol. 67, no. 1, pp. 752–764, Jan. 2018.
[24] T. X. Tran and D. Pompili, "Joint task offloading and resource allocation for multi-server mobile-edge computing networks," *IEEE Trans. Veh. Technol.*, vol. 68, no. 1, pp. 856–868, Jan. 2019.
[25] Y. Liu, C. Xu, Y. Zhan, Z. Liu, J. Guan, and H. Zhang, "Incentive mechanism for computation offloading using edge computing: A stackelberg game approach," *Comput. Netw.*, vol. 129, pp. 399–409, 2017.
[26] L. Xiao, Y. Li, X. Huang, and X. Du, "Cloud-based malware detection game for mobile devices with offloading," *IEEE Trans. Mobile Comput.*, vol. 16, no. 10, pp. 2742–2750, Oct. 2017.
[27] J. Hu and M. P. Wellman, "Nash Q-learning for general-sum stochastic games," *J. Mach. Learn. Res.*, vol. 4, no. Nov, pp. 1039–1069, 2003.
[28] T. P. Lillicrap *et al.*, "Continuous control with deep reinforcement learning," 2015, *arXiv:1509.02971*.
[29] Y. Zhan, P. Li, Z. Qu, D. Zeng, and S. Guo, "A learning-based incentive mechanism for federated learning," *IEEE Internet Things J.*, to be published, doi: 10.1109/JIOT.2020.2967772.
[30] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for Android," in *Proc. 1st ACM Workshop Secur. Privacy Smartphones Mobile Devices*, 2011, pp. 15–26.
[31] X. Wan, G. Sheng, Y. Li, L. Xiao, and X. Du, "Reinforcement learning based mobile offloading for cloud-based malware detection," in *Proc. IEEE Global Commun. Conf.*, 2017, pp. 1–6.
[32] M. J. Osborne and A. Rubinstein, *A Course in Game Theory*. Cambridge, MA, USA: MIT Press, 1994.
[33] R. Lowe, Y. Wu, A. Tamar, J. Harb, O. P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 6379–6390.
[34] A. Graves *et al.*, "Hybrid computing using a neural network with dynamic external memory," *Nature*, vol. 538, no. 7626, 2016, Art. no. 471.
[35] M. Hausknecht and P. Stone, "Deep recurrent Q-learning for partially observable MDPs," in *Proc. AAAI Fall Symp. Ser.*, 2015, pp. 29–37.
[36] N. Heess, J. J. Hunt, T. P. Lillicrap, and D. Silver, "Memory-based control with recurrent neural networks," 2015, *arXiv:1512.04455*.
[37] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proc. 12th Int. Conf. Neural Inf. Process. Syst.*, 2000, pp. 1057–1063.
[38] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *Proc. 31st Int. Conf. Mach. Learn.*, 2015, pp. 1889–1897.
[39] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017, *arXiv: 1707.06347*.
[40] S. Li, S. Bing, and S. Yang, "Distributional advantage actor-critic," *CoRR*, 2018. [Online]. Available: https://arxiv.org/abs/1806.06914
[41] T. Bansal, J. Pachocki, S. Sidor, I. Sutskever, and I. Mordatch, "Emergent complexity via multi-agent competition," 2017, *arXiv: 1710.03748*.
[42] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

Yufeng Zhan received the PhD degree from the School of Automation, Beijing Institute of Technology, China, in 2018. He is currently a postdoc with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong. His research interests include mobile computing, machine learning, and networked control systems.



Song Guo (Fellow, IEEE) received the PhD degree in computer science from the University of Ottawa, Ottawa, Canada. He is currently a full professor with the Department of Computing, The Hong Kong Polytechnic University (PolyU), Hong Kong. Prior to joining PolyU, he was a full professor with the University of Aizu, Japan. His research interests include cloud and green computing, big data, wireless networks, and cyber-physical systems. He has published more than 300 conference and journal papers in these areas and received multiple best paper awards from IEEE/ACM conferences. His research has been sponsored by JSPS, JST, MIC, NSF, NSFC, and industrial companies. He has served as an editor for several journals, including the *IEEE Transactions on Parallel & Distributed Systems*, *IEEE Transactions on Emerging Topics in Computing*, *IEEE Transactions on Green Communications and Networking*, *IEEE Communications Magazine*, and *Wireless Networks*. He has been actively participating in international conferences as general chair and TPC chair. He is a senior member of ACM, and an IEEE Communications Society Distinguished lecturer.

Peng Li (Member, IEEE) received the BS degree from the Huazhong University of Science and Technology, China, in 2007, the MS and PhD degrees from the University of Aizu, Japan, in 2009 and 2012, respectively. He is currently an associate professor with the University of Aizu, Japan. His research interests include cloud computing, Internet-of-Things, big data systems, as well as related wired and wireless networking problems. He has published more than 100 technical papers on prestigious journals and conferences. He won the Young Author Award of IEEE Computer Society Japan Chapter, in 2014. He won the Best Paper Award of IEEE TrustCom 2016. He has supervised students to win the First Prize of IEEE ComSoc Student Competition, in 2016. He has serves as the guest editor of several international journal special issues and he is the editor of the *IEICE Transactions on Communications*.

Jiang Zhang is currently working toward the final year undergraduate degree in the School of Automation, Beijing Institute of Technology, China. His research interests include machine learning and control science and technology.

An Adaptive Thermal Management Framework for Heterogeneous Multi-Core Processors

Young Geun Kim, Minyong Kim, Joonho Kong, and Sung Woo Chung

Abstract—Off-the-shelf embedded systems have adopted heterogeneous multi-core processors which have high-performance big cores and low-power small cores. Though there are two different types of cores in heterogeneous multi-core processors, conventional DVFS (Dynamic Voltage and Frequency Scaling)-based DTM (Dynamic Thermal Management) techniques do not utilize the different types of cores to cool down hot cores. Rather, they primarily reduce the voltage and frequency of the hot cores, leading to performance degradation. In this article, we propose a novel adaptive DTM framework for heterogeneous multi-core processors, which utilizes the big and small cores to prevent performance degradation. Our proposed framework exploits two migration-based DTM techniques: 1) a technique (denoted as $\text{Migration}_{\text{big}\leftrightarrow\text{big}}$) that migrates applications from hot big cores (big cores whose temperature is above a pre-defined threshold) to cold big cores (big cores whose temperature is below the threshold) and 2) a technique (denoted as $\text{Migration}_{\text{big}\leftrightarrow\text{small}}$) that migrates all applications from the big cores to the small cores. In case of thermal emergency of the big cores, our proposed framework checks the number of cold big cores. When there exist available cold big cores, our proposed framework employs $\text{Migration}_{\text{big}\leftrightarrow\text{big}}$ to cool down the hot big cores while not reducing the big core frequency. On the other hand, when there does not exist any available cold big core, our proposed framework employs one between $\text{Migration}_{\text{big}\leftrightarrow\text{small}}$ and a DVFS-based DTM technique, which is expected to result in better performance. In our experiments on an embedded development board, our proposed framework improves the average performance by 8.9 percent, compared to ARM's DVFS-based IPA (Intelligent Power Allocation), satisfying thermal constraints. Our framework also improves the average performance by 10.4 percent, compared to a state-of-the-art predictive DVFS-based DTM technique.

Index Terms—Thermal management, heterogeneous multi-core processor, migration, DVFS, embedded system

1 Introduction

With a rapid advance of process technology, a feature size of MOSFETs in microprocessors has significantly decreased. Thanks to the decreased feature size, more number of transistors have been integrated into the same area, improving performance of the microprocessors with low power consumption [9]. Unfortunately, the increased number of transistors in the same area has led to the increased power density, which in turn increases the temperature even in microprocessors of low-power embedded systems [29].

To manage the temperature of microprocessors, embedded systems usually adopt OS-level DTM (Dynamic Thermal Management) techniques [19], [20]; traditional mechanical techniques, such as air cooling [42], liquid cooling [15], and thermoelectric cooling [23], are not desirable for embedded systems since embedded systems usually have limited space and cost [3], [32]. In case of thermal emergency, the OS-level DTM techniques usually try to operate the microprocessors at lower power states (e.g., reducing voltage/frequency) to reduce the on-chip temperature. Unfortunately, operating the microprocessors at lower power states often degrades performance, causing violations in performance constraints. Those violations often cause critical problems in embedded systems [44]. For example, in vision/image processing systems or smartphones, performance constraint violations often result in QoS (Quality of Service) degradation, leading to user dissatisfaction [41]. To this end, avoiding thermal emergency while satisfying performance constraints is crucial in embedded systems.

Off-the-shelf embedded systems have employed heterogeneous multi-core processors, such as ARM's big.LITTLE [47], which have high-performance big cores and low-power small cores. Though there are two different types of cores in heterogeneous multi-core processors, conventional DVFS (Dynamic Voltage and Frequency Scaling)-based DTM techniques do not primarily exploit unused big cores and the small cores in case of thermal emergency of the big cores[1]. Instead, the DVFS-based DTM techniques first reduce the voltage and frequency of the big cores [43], [46], [50]. Only when the DVFS-based DTM techniques cannot further reduce the frequency of the big cores, they inevitably migrate applications from the big cores to the small cores. Since the DVFS-based DTM techniques do not immediately stop using the hot big cores in case of thermal emergency, they do not reduce the temperature of

- *Y. G. Kim and S. W. Chung are with the Department of Computer Science, Korea University, Seoul 02841, Korea. E-mail: {carrotyone, swchung}@korea.ac.kr.*
- *M. Kim is with the Graduate School of Design, Harvard University, Cambridge, MA 02138. E-mail: mkim05@korea.ac.kr.*
- *J. Kong is with the School of Electronics Engineering, Kyungpook National University, Daegu 41566, Korea. E-mail: joonho.kong@knu.ac.kr.*

Manuscript received 25 Apr. 2019; revised 7 Jan. 2020; accepted 15 Jan. 2020. Date of publication 28 Jan. 2020; date of current version 8 May 2020. (Corresponding author: Joonho Kong.) Recommended for acceptance by J. C. Hoe. Digital Object Identifier no. 10.1109/TC.2020.2970062

1. In heterogeneous multi-core processors, thermal emergency usually occurs in high-performance big cores; it rarely occurs in low-power small cores even at their highest frequency.

the hot big cores rapidly [18]. In this case, applications should run at the reduced big core frequencies until the thermal emergency is resolved, degrading performance.

To avoid performance degradation in case of thermal emergency, in this paper, we propose an adaptive DTM framework for heterogeneous multi-core processors, which utilizes both big and small cores. Along with the conventional DVFS, our proposed framework exploits two migration-based DTM techniques: 1) a technique (denoted as $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$) that migrates applications from hot big cores (big cores whose temperature is above a pre-defined threshold) to cold big cores (big cores whose temperature is below the threshold) and 2) a technique (denoted as $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$) that migrates all applications from the big cores to the small cores [18]. Whenever the temperature of a big core exceeds the pre-defined threshold, our proposed framework checks the number of cold big cores. When there exist available cold big cores, our framework employs $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$ to cool down the hot big cores while not reducing the big core frequency. On the other hand, when there is no available cold big core, our framework estimates performance of $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ and a DVFS-based DTM technique (denoted as DVFS) with running applications using our measurement-based performance estimation method. When the performance of $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ is expected to be higher than that of DVFS, our framework employs $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ to cool down the big cores rapidly. Otherwise, our framework employs DVFS that reduces the big core frequency to sustain temperatures of the big cores under the threshold [17].

In summary, our main contributions are as follows:

1) We propose a novel adaptive DTM framework that exploits migration- as well as DVFS-based DTM techniques in order to avoid performance degradation in case of thermal emergency.
2) For the adaptive DTM framework, we propose two migration-based DTM techniques: $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$ and $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$.
3) We also propose a measurement-based performance estimation method for $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ and DVFS, which is robust to variations of device- or environment-dependent factors (e.g., process variations of processing units, ambient temperature variations, etc.).
4) We implement and evaluate our framework on an embedded development board. In our experiments, our framework improves average performance by 8.9 and 10.4 percent, compared to ARM's DVFS-based IPA [46] and a state-of-the-art predictive DVFS-based DTM technique [43] respectively, satisfying thermal constraints.

The rest of this paper is organized as follows. In Section 2, we introduce related work on OS-level DTM techniques. In Section 3, we propose a novel adaptive DTM framework for heterogeneous multi-core processors. In Section 4, we describe implementation of our proposed framework. In Section 5, we demonstrate our experimental environment and results. Finally, we conclude our work in Section 6.

2 Related Work

2.1 DTM for High-Performance Computer Systems

For high-performance computer systems, many techniques have been proposed to improve performance under the thermal constraints [20]. In OS-level, DTM techniques based on DVFS have been proposed to satisfy thermal constraints [4], [11], [21]. Lee *et al.* proposed a temperature-aware DVFS technique that predicts on-chip temperature with performance counters [21]. Based on the predicted on-chip temperature, their technique adaptively scales the voltage and frequency of cores to prevent thermal problems. Hanumaiah and Vrudhula proposed another temperature-aware DVFS technique for hard real-time applications [11]. Their proposed technique scales the voltage and frequency of cores to prevent thermal problems while satisfying the deadline of the hard real-time applications based on a power/temperature prediction. Cai and Marculescu tried to find an optimal voltage/frequency pair under the thermal constraints, considering temperature inversion effect of FinFET-based multi-core processors [4]. Several OS-level thermal management techniques have been proposed based on task scheduling [5], [6], [10], [27], [36]. Merkel *et al.* tried to balance hot tasks across the cores in symmetric multi-core processors to avoid thermal emergency [27]. Choi *et al.* proposed a thermal-aware task scheduling technique which assigns hot tasks to cores in a balanced manner and delays cool tasks to be executed later than hot tasks [5]. Ge *et al.* proposed another task scheduling-based DTM technique for multi-core processors [10], which proactively balances temperatures of cores by migrating hot applications to cold cores. Coskun *et al.* proposed a task scheduling-based thermal management technique for 3D multi-core processors [6]. Their proposed technique balances the temperature across the cores, considering the location of the cores. Specifically, their technique allocates (or migrates) less CPU-intensive applications to the cores whose temperature is expected to easily increase due to their location. Salami *et al.* tried to adaptively adjust the thermal threshold of task migration-based DTM techniques [36], considering the fact that the thermal characteristics can be changed depending on workloads. However, the above techniques do not consider the heterogeneous multi-core processors, which are widely employed in off-the-shelf embedded devices. Due to the reason, they cannot exploit whole resources available in heterogeneous multi-core processors (e.g., cold big cores and small cores) to reduce the temperature of hot big cores.

2.2 DTM for Embedded Systems

For embedded systems, many OS-level DTM techniques have been proposed for both homogeneous and heterogeneous multi-core processors [19].

2.2.1 DTM for Homogeneous Multi-Core Processors

For homogeneous multi-core processors, Das *et al.* proposed a DVFS- and DPM (Dynamic Power Management)-based thermal management technique to reduce on-chip temperature while satisfying performance constraints [8]. Based on the temperature and CPU utilization, their technique finds a CPU power state (voltage/frequency and the number of active cores) to maintain the temperature under a pre-defined thermal threshold by using a reinforcement learning-based algorithm. Kim *et al.* proposed a temperature-aware DVFS technique [17]. According to Jensen's inequality [31], the average of cubic frequencies is larger than the cube of average frequency when the variance of the frequencies is not zero.

Since dynamic power consumption is roughly proportional to the frequency cubed, the average dynamic power consumption in different frequencies is also larger than the dynamic power consumption with their average frequency. Based on the above insights, their technique tries to maintain frequency and voltage to the average frequency and voltage to save power and enhance performance under thermal constraints. However, the above techniques do not also consider the heterogeneous multi-core processors widely adopted in off-the-shelf embedded systems.

2.2.2 DTM for Heterogeneous Multi-Core Processors

For heterogeneous multi-core processors, Sharifi *et al.* proposed a thermal management technique [39]. Their technique assigns workloads to different types of cores and determines the frequencies of cores in order to reduce the on-chip temperature while satisfying performance constraints at the best-effort. Sharifi *et al.* also proposed TempoMP which contains a thermal model for heterogeneous multi-core processors [38]. The thermal model predicts the on-chip temperature with thermal parameters, such as die thickness, convection capacitance, etc. Based on the predicted temperature and the performance constraints of workloads, it assigns workloads to different types of cores and scales the frequency of cores in order to maintain the temperature under a pre-defined threshold while satisfying performance constraints. Muthukaruppan *et al.* proposed a power management technique for heterogeneous multi-core processors [30]. Their proposed technique consists of the following feedback-based controllers: per-task resource share controller, load balancer, per-cluster DVFS controller, migrator, per-task QoS controller, and chip-level power allocator. The per-task resource share controller adjusts the CPU time slices for each task to satisfy the target performance constraints. In each cluster, the load balancer balances the CPU utilization among the cores and the per-cluster DVFS controller adjusts the frequency of each cluster (big cluster and/or small cluster) to maintain its utilization close to the target utilization. The migrator migrates tasks between different clusters to satisfy their performance constraints. When the power consumption of the clusters exceeds the power budget, the per-task QoS controller and chip-level power allocator reduces the CPU time slices of each task and the clock frequency of each cluster, respectively, so that the total power consumption can be maintained below the power budget. Khdr *et al.* proposed a power density-aware resource management technique for heterogeneous multi-core processors [16]. Their proposed technique derives a uniform power density constraint (a power budget per area) for all cores considering the heat transfer among the cores to avoid thermal violations. Under the derived uniform power density constraint, their technique allocates tasks to different types of cores. After that, their technique adjusts the power density constraint of each type of cores to further improve performance of tasks. For example, if the power density of the small cluster is still lower than the power density constraint even at the maximum frequency, their technique increases the power density constraint of the big cores, so that the frequency of the big cores can be increased. However, the above techniques do not exploit whole available resources (the cold big cores and the small cores) in heterogeneous multi-core processors to cool down hot big cores. Since the techniques do not immediately stop using the hot big cores in case of thermal emergency, they may not reduce the temperature of the hot big cores rapidly. Hence, they often make applications run at the reduced frequencies for a long time, degrading performance.

Recently, ARM introduced a DVFS-based DTM technique for heterogeneous multi-core processors, called IPA (Intelligent Power Allocation) [46]. At runtime, IPA periodically samples the temperatures of CPU cores (including big and small cores) from on-chip thermal sensors. Based on the sampled on-chip temperatures, it calculates the difference (an error value) between a pre-defined threshold[2] and the temperature (or the power consumption) of the hottest core. It then finds the highest frequency that minimizes the error value based on P (a proportional value to the current error value), I (and integral value of the past error values), and D (a derivative value of the error value, which represents the change rate of the error) values. It proactively scales the CPU frequency to the found frequency so that the temperature is converged close to the threshold. It only migrates applications from the big cores to the small cores when the error value of the big cores is expected to be higher than a certain value even at the lowest frequency (and voltage) of the big cores. Baht *et al.* proposed another DVFS-based DTM technique [3] that proactively scales the CPU frequency to the highest one that keeps temperature below a certain threshold based on a temperature prediction; they also used the temperature prediction for power-temperature stability and safety analysis in heterogeneous multi-core processors [2]. Wachter *et al.* also tried to find the highest CPU frequency that keeps temperature below a certain threshold based on the temperature prediction [43]. To improve the accuracy of temperature prediction, they employed a history-based error correction algorithm for their temperature prediction method. However, the above techniques do not also actively exploit whole available resources (the cold big cores and the small cores) in heterogeneous multi-core processors to cool down hot big cores. In addition, if their device- or environment-dependent parameters are not carefully calibrated, they might often provide inappropriate frequency for the applications due to various device- or environment-dependent factors (e.g., process variations of processing units, ambient temperature, etc.) [14]; in case that parameters are not carefully calibrated considering the environment-dependent factors (e.g., even though the ambient temperature can be changed due to the environmental fluctuations, parameters are calibrated for a certain fixed ambient temperature), IPA degrades performance by up to 21.4 percent, compared to the case where device-dependent parameters are well-calibrated [3], [8].

2. Note, for proactive DTM techniques, such as IPA, the thermal threshold is used as the target frequency where the techniques try to keep the temperature below it. On the other hand, for reactive DTM techniques, the thermal threshold is used as the temperature where the techniques start to operate. For both types of DTM techniques, the critical temperature, which is usually much higher than the thermal threshold, can be also used to avoid reliability problems in excessively high temperature ($>= 90\ ^{\circ}C$); when the temperature of a big core exceeds the critical temperature, the techniques immediately reduce the frequency of the CPU cores to the lowest one.

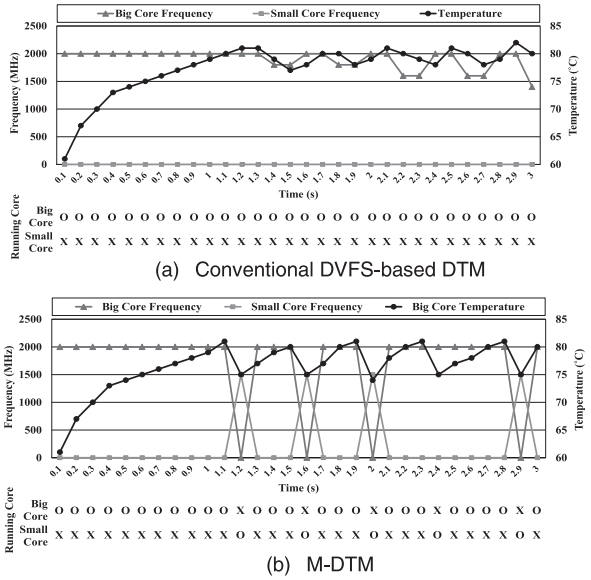


Fig. 1. Runtime behaviors of conventional DVFS-based DTM and M-DTM (matrix01 from the EEMBC benchmark [48] is used for the experiment).

By integrating DVFS and task scheduling, Linux introduced EAS (Energy Aware Scheduling) for heterogeneous multi-core processors [51]. The EAS tries to find an optimal power state (including voltage/frequency and the number of active cores where applications run) using its utilization-based power/performance models, which reduces power consumption while minimizing a performance loss. However, similar to [46], if its device- or environment-dependent parameters are not carefully calibrated, it might fail to find the optimal power state [8].

To improve performance in case of thermal emergency, M-DTM (Migration-based DTM) was proposed [18] for heterogeneous multi-core processors. In case of thermal emergency of the big cores, M-DTM immediately migrates all running applications from the big cores to the small cores. In this way, as shown in Fig. 1, M-DTM cools down the big cores more rapidly, compared to the conventional DVFS-based DTM techniques, providing more time for the applications to run at higher big core frequencies.

In this paper, we extend M-DTM considering the following two aspects. First, when there are available cold big cores, it is possible to utilize the cold big cores to cool down hot big cores (i.e., migrate applications from the hot big cores to the cold big cores) without utilizing small cores or reducing big core frequencies. Second, for applications whose performance is severely degraded on the small cores or whose migration overhead between the big and small cores is huge, the migration to the small cores might result in performance degradation compared to DVFS. Due to this reason, it is necessary to employ a better one between the migration and DVFS in terms of performance, considering application characteristics and migration overheads.

Different from the previous work, we propose a holistic DTM framework which utilizes all available resources (such as idle big cores) in heterogeneous multi-core processors to sustain temperature while improving performance in case of thermal emergency. In the following section, we explain our novel adaptive DTM framework.

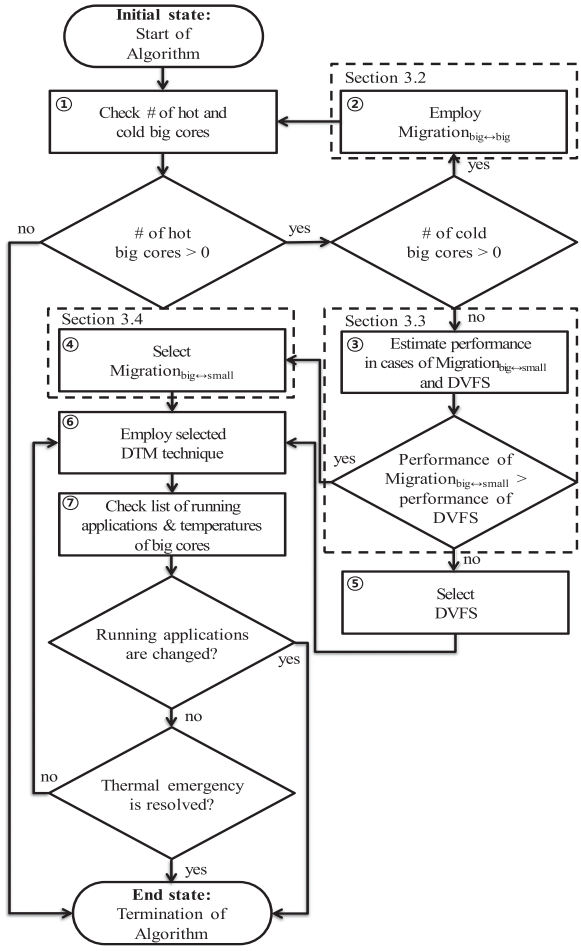


Fig. 2. Algorithm of our proposed adaptive DTM framework.

3 Adaptive Thermal Management Framework for Heterogeneous Multi-Core Processors

3.1 Overview

Our main design philosophy is to utilize all available resources in heterogeneous multi-core processors to sustain temperature while improving performance in case of thermal emergency. To fulfill the design philosophy, our proposed framework adaptively exploits the following three DTM techniques: 1) a technique (denoted as $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$) that migrates applications from hot big cores to cold big cores, 2) a technique (denoted as $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$) that migrates all applications from the big cores to the small cores (M-DTM [18]), and 3) a DVFS-based DTM technique[3] (denoted as DVFS) that reduces the voltage and frequency of the big cores to sustain the temperatures of the big cores under a pre-defined thermal threshold.

Fig. 2 describes an overall algorithm of our framework. Our proposed framework samples on-chip temperatures of the big cores periodically. When the temperature of a big core

3. Our proposed framework uses the "Step Wise" DVFS-based DTM technique, which had been employed in off-the-shelf embedded devices. The technique operates as follows: 1) it periodically measures temperatures of the big cores, 2) when the temperatures of the big cores are above a pre-defined thermal threshold, it reduces the frequency of the big cores by one step, and 3) only when the temperatures of all the big cores are below the threshold, it raises the frequency of the big cores to the highest frequency.

exceeds a pre-defined thermal threshold ($T_{threshold}$), our proposed framework starts the algorithm[4] shown in Fig. 2. At the beginning, the algorithm checks the number of hot and cold big cores (①). If there exist both hot and cold big cores, the algorithm employs $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$ to cool down the hot big cores while not reducing their frequency (②); the details of $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$ are explained in Section 3.2. After employing $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$, the algorithm checks the number of hot and cold big cores again (②→①). If there still exist both hot and cold big cores, the algorithm employs $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$ again. On the other hand, if there does not remain any hot big core (i.e., temperatures of all the big cores are below $T_{threshold}$), the algorithm is terminated, assuming that the thermal emergency of the big cores is resolved.

If all the big cores are hot (i.e., the temperatures of all the big cores are above $T_{threshold}$), the algorithm estimates performance of applications in cases of $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ and DVFS using a measurement-based method (③), which inherently takes into account application characteristics and migration overheads. Based on the estimated performance, it selects one between $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ and DVFS. Details of these procedures are explained in Section 3.3. When the performance of applications with $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ is expected to be higher than that with DVFS, the algorithm selects $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ (④); details of $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ are explained in Section 3.4. Otherwise, the algorithm selects DVFS (⑤). When running applications are changed, the algorithm is terminated, since the best performing DTM technique might be different depending on the running applications (e.g., if running applications are changed from the applications whose performance is not severely degraded on the small cores to other ones whose performance is severely degraded on the small cores, the best performing DTM technique might be changed from $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ to DVFS). On the other hand, if running applications are not changed, it keeps employing the selected DTM technique until the thermal emergency is resolved (⑥ and ⑦). Note, although our framework terminates this algorithm along with the change of applications, it does not incur another thermal violation. The reason is that, since our framework keeps sampling temperatures periodically (i.e., every 100 ms) regardless of the execution of the algorithm, it can immediately restart the algorithm in the next interval.

One of the critical design parameters for our proposed framework is an interval of temperature sampling. Since our framework employs the migration-based DTM techniques depending on the sampled temperatures, it is important to determine the sampling interval considering migration overheads. Generally, the migration overhead among the big cores is very small due to the shared cache; as shown in Table 1, the migration overhead among the big cores is almost unnoticeable, even when we forcibly migrate applications (rspeed01, rgbhpg01, and rgbcmy01 from EEMBC benchmark [48] which have L1 cache-bound working set sizes, a moderate working set sizes, and a relatively large working set sizes, respectively) every 10 ms. Hence, we only consider the migration overhead between the big and small cores, when we determine the sampling interval.

4. As our proposed framework starts to operate when the temperature of a big core exceeds a pre-defined threshold, it is categorized as a reactive DTM technique.

TABLE 1
Execution Time of EEMBC Benchmark Applications w/ and w/o Frequent Migrations Among Big Cores

Application	No Migration	Migration Every 10 ms	Performance Overhead
rspeed01	96.0 s	96.6 s	0.6%
rgbhpg01	96.1 s	96.5 s	0.4%
rgbcmy01	95.9 s	96.1 s	0.2%

To determine the sampling interval considering the migration overhead between the big and small cores, we measure the execution time of an embedded application while employing $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ with various sampling intervals. Fig. 3 shows the execution time of matrix01 (from EEMBC benchmark [48]) with different sampling intervals. As shown in Fig. 3, the execution time of matrix01 is shortest when the sampling interval is 100 ms. In case that the sampling interval is longer than 100 ms, it fails to react fast enough to prevent on-chip temperature from exceeding above the threshold. Even worse, when the temperature exceeds $90\,^{\circ}\mathrm{C}$, a Linux kernel module (or a hardware-level DTM) overrides OS-level DTM techniques and forcibly reduces the frequency of the CPU cores to the lowest one to reduce the temperature, severely degrading performance. On the other hand, when the sampling interval is shorter than 100 ms, its overhead is larger than the performance gain, due to frequent migrations between the big and small cores. Considering the empirical results gained from our experiments, we determine the sampling interval of our proposed framework as 100 ms, which is same as that of DVFS-based DTM techniques widely used for embedded heterogeneous multi-core processors; note, in embedded systems, most DVFS-based techniques (including IPA) use the time granularity of 100 ms [46], considering voltage and frequency transition overheads [33].

3.2 Detailed Algorithm of $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$

In this subsection, we explain the detailed algorithm of $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$ which migrates applications among the big cores. Fig. 4 shows the algorithm of $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$. At first, $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$ identifies a big core whose temperature is above $T_{threshold}$ (①). In addition, it identifies another big core whose temperature is lowest among the big cores (②). After

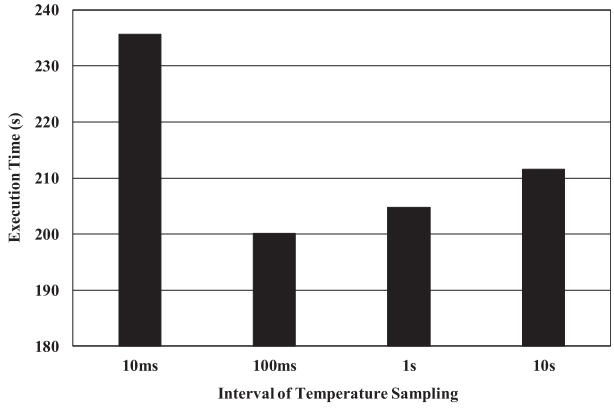


Fig. 3. Execution time of matrix01 with different intervals of temperature sampling for employing $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$.

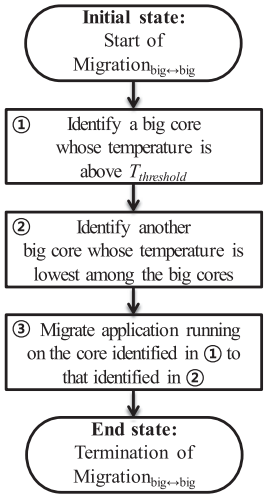


Fig. 4. Algorithm of $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$.

that, $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$ migrates applications (③) running on the big core identified in ① to the big core identified in ②. In this way, $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$ reduces the on-chip temperature of hot big cores without reducing the big core frequency.

3.3 Selection Between $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ and DVFS

To select and employ one between $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ and DVFS when there is no available cold big core, our framework exploits a performance estimation method that executes each technique with running applications once and measures the number of instructions executed per second; it inherently considers application characteristics and migration overheads. Though there have been various methods that estimate performance impact of the migration or DVFS [1], [2], [3], [7], [13], [28], [35], [43], our framework exploits the measurement-based method as it is robust to the variations of device- or environment-dependent factors (e.g., process variations of processing units, ambient temperature variations, etc.); the conventional performance estimation methods are usually prone to being affected by the variations of device- or environment-dependent factors so that relying on them might often fail to provide appropriate amount of CPU resources for applications, degrading performance, as explained in Section 2.2.2.

At the beginning, our performance estimation method employs $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ during a period (denoted as t_{Mig}) while measuring the number of instructions executed per seconds (denoted as N_{Mig}) as (1), using parameters described in Table 2.

$$N_{Mig} = \left(\frac{f_{highest\_small}}{CPI_{highest\_small}(\alpha)} \cdot t_{highest\_small} + \frac{f_{highest\_big}}{CPI_{highest\_big}(\alpha)} \cdot t_{highest\_big} \right) \cdot \frac{1}{t_{Mig}}. \quad (1)$$

Our method measures each parameter, as follows:

1) When $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ migrates all the applications from the big cores to the small cores, the method starts to measure t_{Mig}, $t_{highest\_small}$, and $CPI_{highest\_small}(\alpha)$.

TABLE 2
Parameters Used for Performance Measurement

Parameter	Description
$f_{highest\_big}$	Highest frequency of big cores
$f_{highest\_small}$	Highest frequency of small cores
$t_{highest\_big}$	Time stayed at the highest frequency of big cores
$t_{highest\_small}$	Time stayed at the highest frequency of small cores
$CPI_{highest\_big}(\alpha)$	CPI of an application α at the highest frequency of the big cores
$CPI_{highest\_small}(\alpha)$	CPI of an application α at the highest frequency of the small cores
$f_{0,1,\ldots,n}$	Frequencies of big cores selected by DVFS
$t_{big\_f0,1,\ldots,n}$	Time stayed at each frequency of big cores selected by DVFS
$CPI_{big\_f0,1,\ldots,n}(\alpha)$	CPI of an application α at each frequency of big cores selected by DVFS

2) Since there is no running application on the big cores, the temperature of the big cores decreases below the thermal threshold soon. After that, $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ migrates all the applications back to the big cores. At the same time, the method stops measuring $t_{highest\_small}$ and $CPI_{highest\_small}(\alpha)$, while it starts to measure $t_{highest\_big}$ and $CPI_{highest\_big}(\alpha)$.
3) The method measures t_{Mig}, $t_{highest\_big}$, and $CPI_{highest\_big}(\alpha)$, until the temperatures of the big cores exceed the thermal threshold again.
4) Whenever running applications are changed during the above procedure, our framework (as well as the performance estimation) is immediately terminated assuming that different applications might exhibit different thermal characteristics.

After that, our performance estimation method employs DVFS during another period (denoted as t_{DVFS}) while measuring the number of instructions executed per seconds (denoted as N_{DVFS}) as (2), using parameters described in Table 2.

$$N_{DVFS} = \left(\sum_{k=0}^{n} \frac{f_k \cdot t_{big\_f_k}}{CPI_{big\_f_k}(\alpha)} + \frac{f_{highest\_big} \cdot t_{highest\_big}}{CPI_{highest\_big}(\alpha)} \right) \cdot \frac{1}{t_{DVFS}}. \quad (2)$$

Our method measures each parameter as follows:

1) When DVFS reduces the frequency of the big cores step by step, our method starts to measure t_{DVFS}. In addition, at each frequency step (f_k), the method also measures t_{big_fk} and $CPI_{big\_fk}(\alpha)$.
2) Due to the reduced frequency of the big cores, the temperatures of the big cores decrease below the thermal threshold; DVFS reduces the frequency of the big cores until the temperatures of the big cores are below the thermal threshold. In this case, DVFS raises the frequency of the big cores to the highest frequency. At this time, the method starts to measure $t_{highest\_big}$ and $CPI_{highest\_big}(\alpha)$.
3) The method measures t_{DVFS}, $t_{highest\_big}$, and $CPI_{highest\_big}(\alpha)$, until the temperatures of the big cores exceed the thermal threshold again.
4) Whenever running applications are changed during the above procedure, our framework (as well as the performance estimation) is immediately terminated

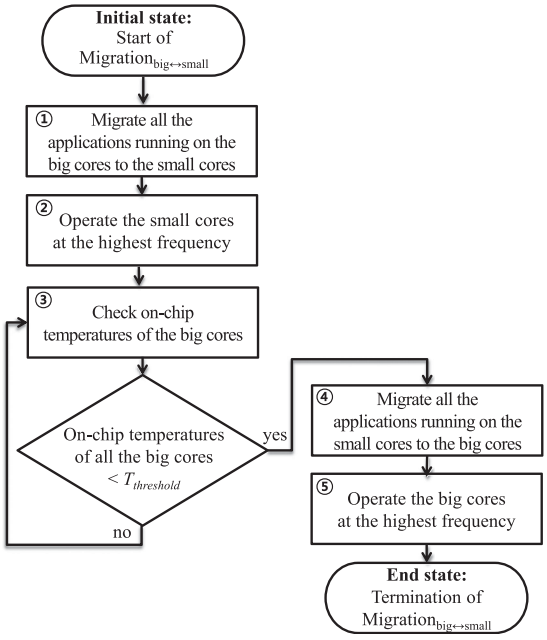


Fig. 5. Algorithm of $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$.

assuming that different applications might exhibit different thermal characteristics.

After measuring N_{Mig} and N_{DVFS} using the method, our proposed framework compares the measured N_{Mig} and N_{DVFS}. In case that N_{Mig} is higher than N_{DVFS}, our proposed framework employs $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$, as explained in Section 3.1; otherwise, it employs DVFS.

To verify the accuracy of the performance estimation method, we compare the estimated performance of the DTM methods for each application to the actual performance. For the applications sets used in our evaluation, the average MAPE (Mean Absolute Percentage Error) between the estimated and the actual number of instructions executed per second is 6.8 percent; though our proposed framework estimates the performance of only a partial interval of applications, the performance of the partial interval represents that of entire execution in embedded applications, since embedded applications usually repeat same operations periodically [34], [48]. To further confirm the usefulness of the performance estimation method, we also compare the DTM technique selected by the performance estimation method to the actual better performing one between $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ and DVFS. In our evaluation, the performance estimation method accurately selects the better performing one for 168 out of 174 cases.

Since our proposed framework estimates performance of the DTM techniques only once as long as the running applications remain same, performance and energy overheads are negligible; for 20 EEMBC benchmark applications, the average performance and EDP overheads of the performance estimation are only 0.74 and 0.76 percent, respectively.

3.4 Detailed Algorithm of $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$

Fig. 5 shows the algorithm of $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$. At the beginning, $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ migrates all the applications running on the big cores to the small cores[5] (①). To minimize the performance loss of applications, $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ scales the frequency of the small cores to the highest frequency (②). Note, through experiments on an off-the-shelf heterogeneous multi-core processor, we found the low-power small cores hardly incur thermal emergency even at the highest frequency.

After migrating applications from the big cores to the small cores, $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ checks the on-chip temperatures of the big cores periodically (③). Once the temperatures of all the big cores decrease below $T_{threshold}$, $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ migrates all the applications running on the small cores back to the big cores (④) and scales the frequency of the big cores to the highest frequency (⑤). Since $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ utilizes the small cores in case of thermal emergency, it reduces the temperatures of the big cores more rapidly compared to the conventional DVFS-based DTM techniques [18]. For this reason, it provides more time for the applications to run at higher big core frequencies, compared to the conventional techniques.

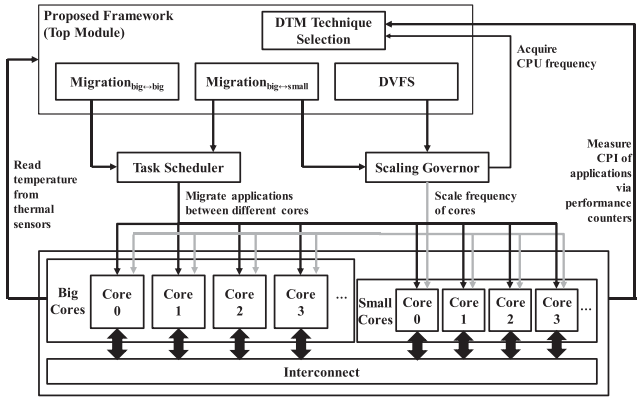


Fig. 6. Implementation overview of our proposed framework.

4 Implementation

We implement our proposed framework in a device driver that manages temperatures of CPU cores (Exynos thermal management unit in our implementation on Odroid-XU3 [49]). There are two prerequisites to implement our proposed framework as follows: 1) at least one thermal sensor is required for each big core in order to identify hot big cores and cold big cores and 2) it should be possible to read CPI and CPU frequency via performance counters and CPU frequency device drivers, respectively, in order to estimate performance of DTM techniques. Note, since most off-the-shelf embedded devices meet the prerequisites, our proposed framework can easily be implemented on the devices without additional hardware.

Fig. 6 shows the implementation overview of our proposed framework. As shown in Fig. 6, our proposed framework (the top module) consists of the following four submodules: 1) $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$, 2) DTM technique selection, 3) $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$, and 4) DVFS. To deal with the

5. After $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ migrates applications from one type of cores to another type of cores, an OS task scheduler balances CPU utilization of applications across the same type of cores [52].

thermal emergency of the big cores, the modules cooperate with a task scheduler and a CPU frequency scaling governor (an OS-level module that scales the frequency of the cores).

At the beginning, the top module periodically reads temperatures of the big cores from on-chip thermal sensors[6]. When the temperature of a big core exceeds $T_{threshold}$, the top module runs $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$ module, as long as there exist both hot and cold big cores. The $\mathrm{migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$ module first obtains the temperatures of the big cores from the top module. Based on the obtained temperatures, the $\mathrm{migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$ module commands the task scheduler to migrate applications from a hot big core to a cold big core. Taking the command, the task scheduler migrates applications through the interconnect.

When the temperatures of all the big cores exceed $T_{threshold}$, the top module runs the DTM technique selection module. The module runs the $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ module and the DVFS module one by one. At the same time, the module acquires CPU frequency from the CPU frequency device driver. It also measures CPI of applications from the performance counter [24], [25], [26]. Based on the acquired and measured values, the module estimates performance of the above two modules and selects one between them, which is expected to result in better performance.

If the DTM technique selection module selects the $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ module, the top module runs the $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ module. Similar to the $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$ module, the $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ module obtains the temperatures of the big cores from the top module. Based on the obtained temperatures, the module commands the task scheduler to migrate applications between the big and small cores. In addition, it commands the CPU frequency scaling governor to scale the frequency of the big and small cores to the highest frequency.

On the other hand, if the DTM technique selection module selects the DVFS module, the DVFS module commands the CPU frequency scaling governor to scale the frequency of the big cores depending on the temperatures of the big cores obtained from the top module.

5 Evaluation

5.1 Experimental Environment

We perform our evaluation on Odroid-XU3 [49], an embedded development board running Android version 4.4.2 (KitKat) and Linux kernel 3.10. Odroid-XU3 has Samsung Exynos 5422 [54], which runs four high-performance big cores and four low-power small cores at the same time. The processor supports 9 frequency steps (from 2.0 GHz to 1.2 GHz, 100 MHz per step) and 6 frequency steps (from 1.5 GHz to 1.0 GHz, 100 MHz per step) for the big cores and small cores, respectively. Note, on Odroid-XU3, four same type cores operate at the same frequency (per-core DVFS is not available).

In our evaluation, we use 20 applications from the EEMBC [48], which is one of the most widely used embedded benchmark suites. The detailed description of each application is shown in Table 3. To evaluate our proposed framework for the case where multiple cores are simultaneously utilized, we conducted our experiments while changing the number of simultaneously running applications from 1 to 4 for each application in the EEMBC. We do not utilize multi-threaded applications in our experiments, since most applications in off-the-shelf embedded devices (e.g., smartphones) have a low thread-level parallelism [12], [37], [40]; as observed in [37], many real-world applications typically utilize only a small number of cores (average TLP of real-world applications is between 1.40 and 2.95). Note running multiple single-threaded applications is not much different from running multi-threaded applications in terms of CPU utilization. We also evaluate our proposed framework with the application sets shown in Table 4, which are composed of different applications. In total, we evaluate our proposed framework with 87 cases for each $T_{threshold}$, as shown in Table 5.

We compare our proposed DTM framework to ARM's DVFS-based IPA which is backported from Linux Kernel 4.2 [46], M-DTM [18], and a state-of-the-art predictive DVFS-based DTM technique [43] (denoted as PD-DTM in the rest of this section), in terms of performance, temperature, and system-wide EDP (Energy Delay Product). The

TABLE 3
EEMBC Benchmark Applications for Evaluation

Application	Description
iirflt01	Infinite impulse response filter algorithm
matrix01	Matrix calculations
pntrch01	Pointer chasing algorithm
puwmod01	Pulse-width modulation algorithm
rspeed01	Road speed calculation algorithm
tblook01	Table lookup algorithm
ttsprk01	Tooth-to-spark test algorithm
cjpeg	Standard compression algorithm for JPEG
djpeg	Standard decompression algorithm for JPEG
rgbcmy01	RGB-to-CMYK conversion algorithm
rgbhpg01	High pass gray-scale filter algorithm
rgbyiq01	RGB-to-YIQ conversion algorithm
ospf	OSPF Dijkstra's algorithm
pktflowb512k	Packet flow algorithm with packet size of 512Kb
routelookup	Route lookup algorithm
bezier01fixed	Bezier curve algorithm with integers
bezier01float	Bezier curve algorithm with floating points
dither01	Floyd-Steinberg error diffusion dithering algorithm
rotate01	Bitmap rotation algorithm
text01	Text parsing algorithm

TABLE 4
Application Sets composed of Different Applications

Set No.	Applications
1	rgbcmy01 and iirflt01
2	text01 and rgbcmy01
3	rspeed01 and bezier01fixed
4	puwmod01, matrix01, and cjpeg
5	ttsprk01, pktflowb512k, and rgbhpg01
6	ospf, djpeg, tblook01, and bezier01float
7	routelookup, rgbyiq01, rotate01, and dither01

6. Though the number of cores/sensors gets larger, it is feasible for our proposed framework to read the temperatures of the cores from the thermal sensors, since the time granularity of sensor reading (the order of us [22]) is much smaller than that of temperature sampling for our proposed framework (100 ms).

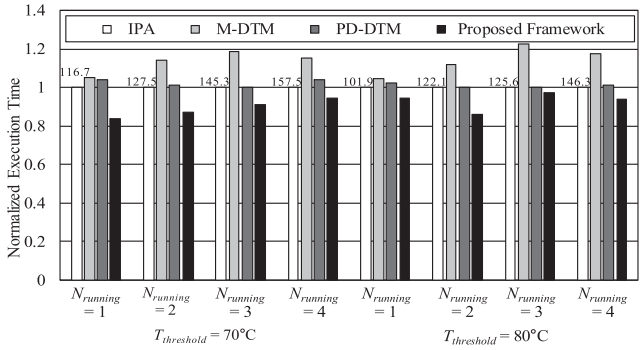


Fig. 7. Normalized execution time of IPA, M-DTM, PD-DTM, and proposed framework (The result is the average of 20 applications from EEMBC. Note the numbers above the white bar stand for the absolute execution time in seconds of the IPA).

detailed operation of IPA is described in Section 2.2.2. For a conservative evaluation, we calibrated device-dependent and environment-dependent parameters of PID controller in IPA, so that the IPA can find the highest frequency that keeps the actual temperature below the thermal threshold; it is difficult to calibrate those parameters due to variations of device- or environment-dependent factors (e.g., process variations of processing units, ambient temperature, etc.). For the PD-DTM [43], we implemented its temperature prediction model along with the error correction algorithm on Odroid-XU3; in our evaluation, the MAE (Mean Absolute Error) of the temperature prediction model of the PD-DTM with its error correction algorithm is 1.23°C.

We keep track of the temperature for each big core by using four on-chip thermal sensors (one thermal sensor for each big core). To measure energy consumption, we use Monsoon Power Meter [53], an external power measurement device. We use $T_{threshold}$ of $80\,^{\circ}\mathrm{C}$ by referring to thermal thresholds of real-world embedded devices. In addition, we use $T_{threshold}$ of $70\,^{\circ}\mathrm{C}$ by referring to thermal thresholds of embedded devices where skin of users directly contacts. For the four techniques, we use the critical temperature (the temperature in which microprocessors incur reliability problems [45]) of $90\,^{\circ}\mathrm{C}$; when the temperature of a big core exceeds the critical temperature, the techniques immediately reduce the frequency of the CPU cores to the lowest frequency. We perform our experiments at room temperature of $20\,^{\circ}\mathrm{C}$. Though a cooling fan is attached on Odroid-XU3, we disable the fan in our evaluation, in order to explore only the impact of OS-level DTM on performance, temperature, and system-wide EDP.

5.2 Experimental Result

5.2.1 Performance

Fig. 7 shows execution time of our proposed framework when $N_{running}$ identical applications are simultaneously running under two different thermal thresholds ($T_{threshold}$). Note the execution time at each $T_{thresold}$ and $N_{running}$ is normalized to that of the IPA.

In case of $T_{threshold} = 70\,^{\circ}\mathrm{C}$, our proposed framework improves performance by 16.2 and 12.8 percent, on average, when $N_{running} = 1$ and $N_{running} = 2$, respectively, compared to the IPA. When the number of running applications is lower than that of big cores, there is a high possibility of having more number of available big cores. In this case, $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$ of our framework utilizes the available big cores to cool down the hot big cores without reducing their frequency. This result is meaningful since real-world embedded devices usually run only one or two computation-intensive applications at the same time [12], [37], [40]. Even when $N_{running} = 3$ and $N_{running} = 4$, our framework improves performance by 8.8 and 5.5 percent (on average), respectively, compared to the IPA. The reason is that, when temperatures of all the big cores are above $T_{threshold}$, our framework employs one between $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ and DVFS, which is expected to perform better.

In case of $T_{threshold} = 80\,^{\circ}\mathrm{C}$, our proposed framework shows slightly lower performance improvement, compared to the case of $T_{threshold} = 70\,^{\circ}\mathrm{C}$. The reason is that, in case of higher $T_{threshold}$, performance of the system is less sensitive to the DTM compared to lower $T_{threshold}$ since DTM operations are not triggered until the temperature reaches the higher $T_{threshold}$. Nevertheless, even in case of $T_{threshold} = 80\,^{\circ}\mathrm{C}$, our framework improves average performance by 5.4 and 13.9 percent when $N_{running} = 1$ and $N_{running} = 2$, respectively, compared to IPA. In addition, when $N_{running} = 3$ and $N_{running} = 4$, our framework also improves average performance by 3.0 and 5.8 percent, respectively, compared to IPA. Note, our proposed framework leads to the most significant performance improvement in the case of $N_{running} = 2$, due to the following reasons: 1) in the case of $N_{running} = 1$, the period of time during which the DTM techniques operate is relatively shorter compared to the other cases ($N_{running} = 2, 3$, and 4), so that there is a less room for performance improvement and 2) in the case of $N_{running} = 2$, there is a high possibility to have more number of available cold big cores compared to the case where $N_{running} = 3$ and $N_{running} = 4$, so that our proposed framework can provide more time for the applications to run at the highest frequency of the big cores through $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$.

Regardless of $T_{threshold}$, our proposed framework outperforms the M-DTM; it enhances the performance by 19.8 percent, on average, compared to the M-DTM. There are two main reasons: 1) our framework first utilizes available cold big cores to cool down the hot big cores and 2) our framework employs DVFS for the applications whose performance is severely degraded on the small cores or whose migration overhead is huge, instead of always migrating applications to the small cores.

Compared to the PD-DTM, our proposed framework enhances performance by 10.4 percent, on average. Since our framework exploits the whole available resources (the cold big cores and the small cores) in heterogeneous multi-core processors to cool down the hot big cores, it provides more time for the applications to run at higher big core frequencies, compared to the PD-DTM. As compared to IPA, the PD-DTM shows worse performance by 1.7 percent (on average), since it reduces the big core frequency slightly more than IPA to avoid thermal emergency based on the predicted temperature; the predictive DVFS of the PD-DTM tries to suppress temperature strictly below the $T_{threshold}$. If we use higher $T_{threshold}$ for the PD-DTM, the PD-DTM may outperform the IPA by providing higher big core frequencies. Note, even in the case where we increase $T_{threshold}$ of the PD-DTM by 5.5°C to make its peak temperature close to

TABLE 5
Categorization of Results Depending on Actually Employed DTM Techniques by Proposed Framework

$T_{threshold}$	70 °C				80 °C			
$N_{running}$	1	2	3	4	1	2	3	4
routelookup	M_{bb}	M_{bs}	M_{bs}	M_{bs}	M_{bb}	M_{bs}	M_{bs}	M_{bs}
rspeed01	M_{bs}	M_{bs}	M_{bs}	M_{bs}	M_{bb}	D	M_{bs}	M_{bs}
dither01	M_{bs}	M_{bs}	M_{bs}	M_{bs}	M_{bb}	D	M_{bs}	M_{bs}
rotate01	D	M_{bs}	M_{bs}	M_{bs}	M_{bb}	D	M_{bs}	M_{bs}
puwmod01	D	M_{bs}	M_{bs}	M_{bs}	M_{bb}	D	M_{bs}	M_{bs}
ospf	D	D	M_{bs}	M_{bs}	M_{bb}	D	M_{bs}	M_{bs}
text01	D	D	M_{bs}	M_{bs}	M_{bb}	D	M_{bs}	M_{bs}
rgbcmy01	D	D	D	D	M_{bb}	D	D	M_{bs}
pktflowb512k	D	D	D	D	M_{bb}	D	D	M_{bs}
bezier01fixed	D	D	D	D	M_{bb}	D	D	M_{bs}
pntrch01	D	D	D	D	M_{bb}	D	D	M_{bs}
ttsprk01	D	D	D	D	M_{bb}	D	D	D
cjpeg	D	D	D	D	M_{bb}	D	D	D
rgbyiq01	D	D	D	D	M_{bb}	D	D	D
iirflt01	D	D	D	D	M_{bb}	D	D	D
tblook01	D	D	D	D	M_{bb}	D	D	D
djpeg	D	D	D	D	M_{bb}	D	D	D
matrix01	D	D	D	D	M_{bb}	D	D	D
rgbhpg01	D	D	D	D	M_{bb}	D	D	D
bezier01float	D	D	D	D	M_{bb}	D	D	D
Set 1	-	D	-	-	-	D	-	-
Set 2	-	D	-	-	-	D	-	-
Set 3	-	D	-	-	-	D	-	-
Set 4	-	-	D	-	-	-	D	-
Set 5	-	-	D	-	-	-	D	-
Set 6	-	-	-	D	-	-	-	D
Set 7	-	-	-	M_{bs}	-	-	-	M_{bs}

that of our framework, our framework still enhances performance by 6.1 percent, on average, compared to the PD-DTM; our framework still provides higher big core frequencies for the applications by exploiting the whole available resources in heterogeneous multi-core processors.

To further analyze the performance improvement of our proposed framework, we categorize the results into the three groups (Group M_{bb}, Group M_{bs}, and Group D), as shown in Table 5. For Group M_{bb}, our framework only employs $\text{Migration}_{\text{big}\leftrightarrow\text{big}}$, since thermal emergency is solely resolved with $\text{Migration}_{\text{big}\leftrightarrow\text{big}}$. For the other groups, thermal emergency is not solely resolved with $\text{Migration}_{\text{big}\leftrightarrow\text{big}}$ due to heavy computations of running applications. In this case, our framework employs either $\text{Migration}_{\text{big}\leftrightarrow\text{small}}$ (Group M_{bs}) or DVFS (Group D) after $\text{Migration}_{\text{big}\leftrightarrow\text{big}}$.

For Group M_{bb}, our proposed framework does not need to reduce the frequency of the big cores to resolve thermal emergency, whereas IPA and PD-DTM reduce the big core frequency. In addition, different from M-DTM, our framework does not need to migrate applications to the small cores. For these reasons, it provides more time for the applications to run at higher frequencies of the big cores, compared to the IPA, M-DTM, and PD-DTM. In addition, as explained in Section 3.1, the performance overhead of migration among the big cores is negligible. As a result, our framework improves performance of this group (shown in Fig. 8) by 5.5, 9.5, and 7.3 percent, on average, compared to IPA, M-DTM, and PD-DTM, respectively.

In case of Group M_{bs}, our proposed framework adopts $\text{Migration}_{\text{big}\leftrightarrow\text{small}}$ after $\text{Migration}_{\text{big}\leftrightarrow\text{big}}$. In this group, the

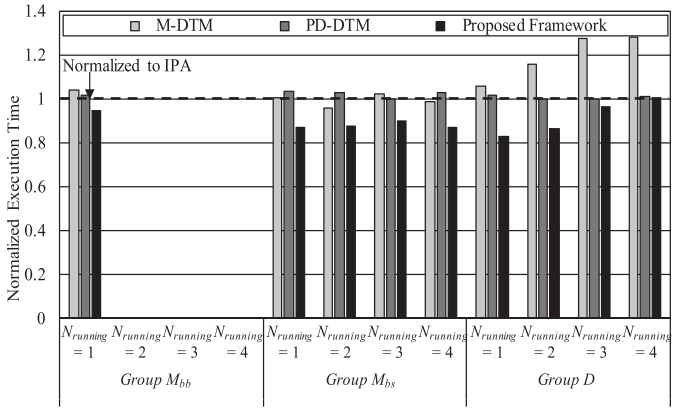
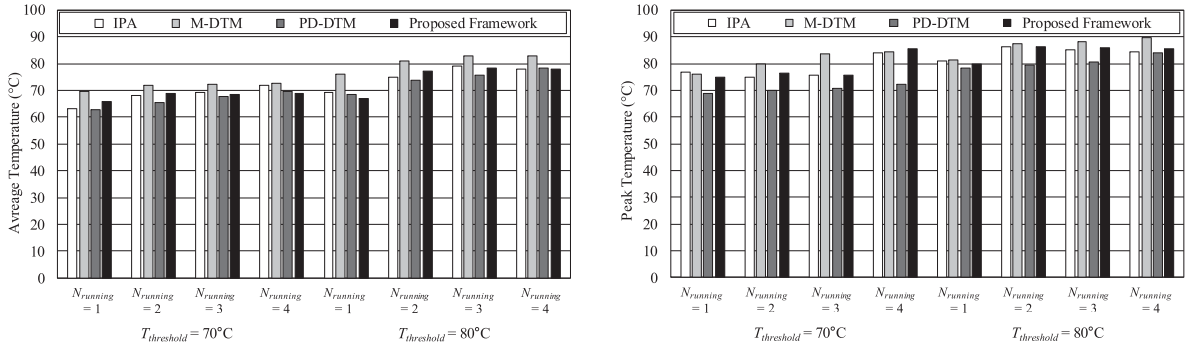


Fig. 8. Normalized execution time of M-DTM, PD-DTM, and proposed framework for each group (Note execution time is normalized to that of IPA).

applications (e.g., routelookup) commonly have a relatively large number of branch mispredictions or small working set sizes. Since the small cores have shorter branch misprediction penalty, the applications in this group exhibit less slowdown when our proposed framework migrates them to the small cores, compared to the applications in the other groups. In addition, due to the small working set sizes, the migration overhead between the big and small cores is not significant. In this case, the benefit of $\text{Migration}_{\text{big}\leftrightarrow\text{small}}$ ($\text{Migration}_{\text{big}\leftrightarrow\text{small}}$ provides more time for the applications to run at higher frequencies of the big cores by rapidly cooling down the big cores) outweighs its adverse effects (slowdown on small cores and migration overhead). Moreover, since our framework employs $\text{Migration}_{\text{big}\leftrightarrow\text{big}}$ before $\text{Migration}_{\text{big}\leftrightarrow\text{small}}$, it provides even more time for the applications to run at the highest frequency of the big cores, compared to IPA, M-DTM, and PD-DTM. As a result, our framework enhances average performance of this group by 12.7 and 12.6 percent when $N_{running} = 1$ and $N_{running} = 2$, respectively, compared to IPA. In addition, even when $N_{running} = 3$ and $N_{running} = 4$, our proposed framework enhances average performance by 10.2 and 12.9 percent, respectively, compared to IPA. Our framework also enhances performance by 11.5 and 14.1 percent, on average, compared to M-DTM and PD-DTM, respectively.

In case of Group D, the applications (e.g., bezier01float) commonly have a small number of branch mispredictions and large working set sizes, so that they tend to exhibit large performance degradation when migrating to the small cores. Due to the reason, for this group, our proposed framework adopts DVFS instead of $\text{Migration}_{\text{big}\leftrightarrow\text{small}}$, after $\text{Migration}_{\text{big}\leftrightarrow\text{big}}$. Compared to the IPA and PD-DTM, the performance improvement of our framework mostly comes from $\text{Migration}_{\text{big}\leftrightarrow\text{big}}$; employing $\text{Migration}_{\text{big}\leftrightarrow\text{big}}$ before DVFS allows more time for the applications to run at the highest big core frequency. As a result, for this group, our proposed framework improves performance by 8.5 and 9.3 percent, on average, compared to IPA and PD-DTM, respectively. In addition, our framework enhances performance by 23.7 percent, on average, compared to M-DTM, since DVFS performs better for the applications whose performance is severely degraded on the small cores or migration overhead is huge.



(a) Average temperature (b) Peak temperature

Fig. 9. Average and peak temperature of the IPA, M-DTM, PD-DTM, and proposed framework (The result is the average of 20 applications from EEMBC).

5.2.2 Temperature

Fig. 9 shows average and peak on-chip temperature of IPA, M-DTM, PD-DTM, and our proposed framework. As shown in Fig. 9a, the average on-chip temperatures of IPA, M-DTM, PD-DTM, and our framework are near $T_{threshold}$ (70 and $80\,^{\circ}\mathrm{C}$). In addition, our proposed framework shows peak temperature similar to that of IPA and M-DTM, as shown in Fig. 9b. PD-DTM shows lower peak temperature, compared to the other three DTM techniques, as its predictive DVFS tries to maintain the temperature strictly below the $T_{threshold}$.

In Fig. 9b, peak temperatures of the techniques are often higher than $T_{threshold}$ (70 and $80\,^{\circ}\mathrm{C}$) since the techniques sample on-chip temperature every 100 ms. In other words, the techniques do not take any action until the next sampling interval begins (100 ms in the worst case), even when the temperature exceeds $T_{threshold}$. However, as soon as the DTM operations are triggered, the on-chip temperature decreases below $T_{threshold}$. For this reason, the applications do not run above the peak temperature for a long time. Note, in our experiments, the peak temperature of our proposed framework is always lower than the critical temperature ($90\,^{\circ}\mathrm{C}$ in our experiments). As a result, our proposed framework has never been overridden by the Linux kernel module (or the hardware-level DTM) which is triggered at the higher thermal threshold (e.g., 90 or $95\,^{\circ}\mathrm{C}$).

On the other hand, as shown in Fig. 10, our proposed framework spends considerably shorter time in thermal emergency (the period of time when the temperatures of one or more big cores stay above $T_{threshold}$) compared to the IPA and M-DTM. Especially, in case that only one application runs ($N_{running} = 1$) with $T_{threshold}$ of $80\,^{\circ}\mathrm{C}$, our proposed framework spends only 1.5 percent of the execution time in thermal emergency whereas the IPA and M-DTM spends 79.5 and 69.8 percent of the execution time in thermal emergency (on average), respectively. The reason is that our proposed framework more rapidly reduces the on-chip temperature of the big cores, compared to IPA and M-DTM, by utilizing the cold big cores through migration. Since the time spent in thermal emergency affects IC lifetime reliability [20], our proposed framework might have a positive impact on lifetime reliability of microprocessors, compared to IPA and M-DTM. In the case of the PD-DTM, the predictive DVFS of the PD-DTM tries to suppress temperature strictly below the $T_{threshold}$. On the contrary, our framework tries to maintain temperature as close to $T_{threshold}$ as possible (though it may slightly exceed $T_{threshold}$). Due to this reason, our framework spends a longer time in thermal emergency compared to the PD-DTM. Nonetheless, the average temperature of our framework is close to that of the PD-DTM, as shown in Fig. 9a. Hence, an actual adverse reliability impact of our framework might not be much different from that of the PD-DTM.

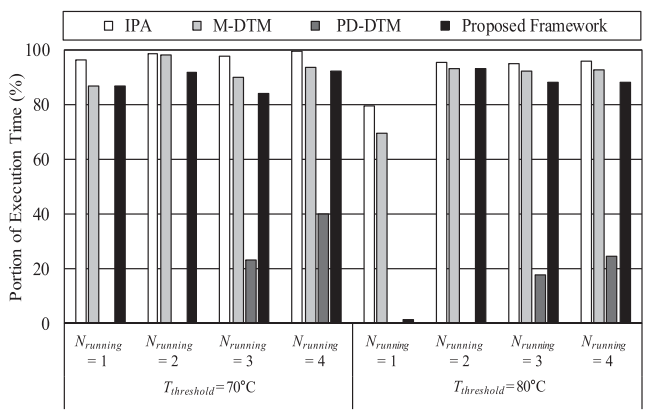


Fig. 10. Average portion of time spent in thermal emergency (The result is the average of 20 applications from EEMBC).

5.2.3 Energy Delay Product

Fig. 11 shows system-wide EDP (Energy Delay Product) of our proposed framework; note system-wide EDP of our framework, M-DTM, and PD-DTM at each $T_{thresold}$ and $N_{running}$ is normalized to that of IPA. Our proposed framework reduces system-wide EDP by 8.6, 36.6, and 8.8 percent, on average, compared to IPA, M-DTM, and PD-DTM, respectively. Since our proposed framework provides higher big core frequencies for the applications, it tends to dissipate a little more power, compared to IPA and PD-DTM. Due to the increased power, our proposed framework consumes 4.6 and 2.9 percent more energy, compared to IPA and PD-DTM, respectively. Nevertheless, since our proposed framework significantly improves performance of applications, as explained in Section 5.2.1, it eventually reduces system-wide EDP of applications. This result is meaningful since improving performance in case of thermal emergency has been more crucial in recent embedded systems; the performance constraint violations in case of thermal emergency often cause critical problems in recent embedded systems [41], [44].

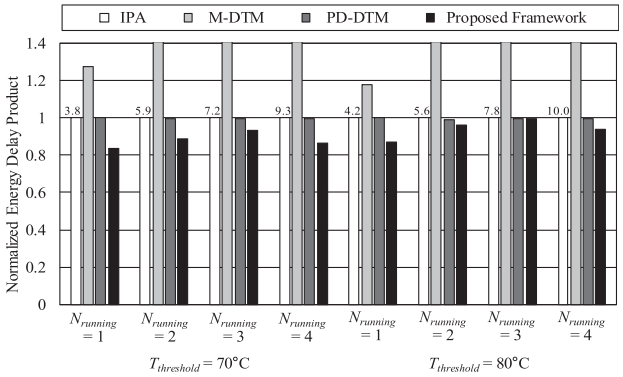


Fig. 11. Normalized system-wide EDP (energy delay product) of IPA, M-DTM, PD-DTM, and proposed framework (The result is the average of 20 applications from EEMBC. Note the numbers above the white bar stand for the absolute EDP in mWh^2 of the IPA).

6 Conclusion

In heterogeneous multi-core processors (such as ARM's big. LITTLE), the conventional DTM techniques primarily rely on DVFS. However, since the techniques do not utilize available resources, such as the cold big cores and the small cores, they provide only a limited time for applications to run at the higher frequencies of the big cores.

In this paper, we propose an adaptive DTM framework for heterogeneous multi-core processors. When the temperature of a big core exceeds a pre-defined threshold, our proposed framework checks the number of cold big cores. When there exist cold big cores, our framework employs $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{big}}$ to cool down the hot big cores without reducing the big core frequency. On the other hand, when there is no available cold big core, our proposed framework employs one between $\mathrm{Migration}_{\mathrm{big}\leftrightarrow\mathrm{small}}$ and a DVFS-based DTM technique, which is expected to perform better. In our experiments on an embedded development board, our proposed framework improves performance by 8.9 and 10.4 percent (on average), compared to ARM's DVFS-based IPA [46] and a state-of-the-art predictive DVFS-based DTM technique [43] respectively, satisfying thermal constraints. We believe our proposed framework can be adopted in a wide variety of embedded systems, such as vision/image processing systems, where high-performance real-time processing is critical.

Acknowledgments

This work was supported by Next-Generation Information Computing Development Program through National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (2017M3C4A7080243), the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2018R1D1A3B07045908), Samsung Electronics, and College of Information, Korea University. The authors would also like to thank the editor and anonymous reviewers for their helpful feedback.

References

[1] R. Ayoub *et al.*, "OS-level power minimization under tight performance constraints in general purpose systems," in *Proc. Int. Symp. Low Power Electronics Des.*, 2011, pp. 321–326.

[2] G. Bhat, S. Gumussoy, and U. Y. Ogras, "Power-temperature stability and safety analysis for mutli-processor systems," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5, pp. 1–19, 2017, Art. no. 145.

[3] G. Bhat, G. Singla, A. K Unver, and U. Y. Ogras, "Algorithmic optimization of thermal and power management for heterogeneous mobile platforms," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 26, no. 3, pp. 544–557, Mar. 2018.

[4] E. Cai and D. Marculescu, "TEI-turbo: temperature effect inversion-aware turbo boost for FinFET-based multi-core systems," in *Proc. Int. Conf. Comput.-Aided Des.*, 2015, pp. 500–507.

[5] J. Choi, C.-Y. Cher, H. Franke, H. Hamann, A. Weger, and P. Bose, "Thermal-aware task scheduling at the system software level," in *Proc. Int. Symp. Low Power Electronics Des.*, 2007, pp. 213–218.

[6] A. K. Coskun, J. L. Ayala, D. Atienza, T. S. Rosing, and Y. Leblebici, "Dynamic thermal management in 3D multicore architectures," in *Proc. Des. Automat. Test Europe Conf.*, 2009, pp. 1–6.

[7] K. V. Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in *Proc. Int. Symp. Comput. Archit.*, 2012, pp. 213–224.

[8] A. Das, M. J. Walker, A. Hansson, B. Al-Hashimi, and G. Merrett, "Hardware-software interaction for run-time power optimization: A case study of embedded linux on multicore smartphones," in *Proc. Int. Symp. Low Power Electron. Des.*, 2015, pp. 165–170.

[9] J. S. Gaggatur, I. Raja, and G. Banerjee, "On-chip non-intrusive temperature detection and compensation of a fully integrated CMOS RF power amplifier," in *Proc. Int. Conf. VLSI Des. Int. Conf. Embedded Syst.*, 2017, pp. 21–26.

[10] Y. Ge, P. Malani, and Q. Qiu, "Distributed task migration for thermal management in many-core systems," in *Proc. Des. Automat. Conf.*, 2010, pp. 579–584.

[11] V. Hanumaiah and S. Vrudhula, "Temperature-aware DVFS for hard real-time applications on multicore processors," *IEEE Trans. Comput.*, vol. 61, no. 10, pp. 1484–1494, Oct. 2012.

[12] P.-C. Hsiu, P.-H. Tseng, W.-M. Chen, C.-C. Pan, and T.-W. Kuo, "User-centric scheduling and governing on mobile devices with big.LITTLE processors," *ACM Trans. Embedded Comput. Syst.*, vol. 15, no. 1, pp. 1–23, 2016, Art. no. 17.

[13] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *Proc. Int. Symp. Microarchit.*, 2006, pp. 347–358.

[14] H. B. Jang *et al.*, "Exploiting application/system-dependent ambient temperature for accurate microarchitectural simulation," *IEEE Trans. Comput.*, vol. 62, no. 4, pp. 705–715, Apr. 2013.

[15] H. B. Jang, I. Yoon, C. H. Kim, S. Shin, and S. W. Chung, "The impact of liquid cooling on 3D multi-core processors," in *Proc. Int. Conf. Comput. Des.*, 2009, pp. 472–478.

[16] H. Khdr *et al.*, "Power densit-aware resource management for heterogeneous tiled multicores," *IEEE Trans. Comput.*, vol. 66, no. 3, pp. 488–501, Mar. 2017

[17] J. M. Kim, Y. G. Kim, and S. W. Chung, "Stabilizing CPU frequency and voltage for temperature-aware DVFS in mobile devices," *IEEE Trans. Comput.*, vol. 64, no. 1, pp. 286–292, Jan. 2015.

[18] Y. G. Kim, M. Kim, J. M. Kim, and S. W. Chung, "M-DTM: Migration-based dynamic thermal management for heterogeneous mobile multi-core processors," in *Proc. Des. Automat. Test Europe Conf.*, 2015, pp. 1533–1538.

[19] Y. G. Kim, J. Kong, and S. W. Chung, "A survey on recent OS-level energy management techniques for mobile processing units," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 10, pp. 2388–2401, Oct. 2018.

[20] J. Kong, S. W. Chung, and K. Skadron, "Recent thermal management techniques for microprocessors," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 1–42, 2012, Art. no. 13.

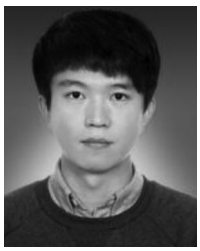
[21] J. S. Lee, K. Skadron, and S. W. Chung, "Predictive temperature-aware DVFS," *IEEE Trans. Comput.*, vol. 59, no. 1, pp. 127–133, Jan. 2010.

[22] C.-A. Lefebvre, J. L. Montero, and L. Rubio, "Implementation of a fast relative digital temperature sensor to achieve thermal protection in zynq SoC technology," *Microelectronics Rel.*, vol. 79, pp. 433–439, 2017.

[23] J. Long, D. Li, and S.-O. Memik, "Theory and analysis for optimization of on-chip thermoelectric cooling systems," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 32, no. 10, pp. 1628–1632, Oct. 2013.

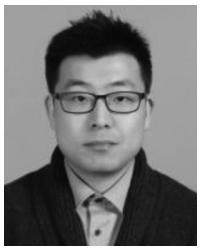
[24] P. Mercati, A. Bartolini, F. Patena, L. Benini, and T. S. Rosing, "Workload and user experience-aware dynamic reliability management in multicore processors," in *Proc. Des. Automat. Conf.*, 2013, pp. 1–6.

[25] P. Mercati, A. Bartolini, F. Patena, T. S. Rosing, and L. Benini, "A linux-governor based dynamic reliability management for android mobile devices," in *Proc. Des. Automat. Test Europe Conf.*, 2014, pp. 1–6.
[26] P. Mercati, F. Paterna, A. Bartolini, L. Benini, and T. S. Rosing, "WARM: Workload-aware reliability management in linux/android," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 36, no. 9, pp. 1557–1570, Sep. 2017.
[27] A. Merkel, F. Bellosa, and A. Weissel, "Event-driven thermal management in SMP systems," in *Proc. Int. Workshop Temp.-Aware Comput. Syst.*, 2005, pp. 1–10.
[28] R. Miftakhutdinov, E. Ebrahimi, and Y. N. Patt, "Predicting performance impact of DVFS for realistic memory systems," in *Proc. Int. Symp. Microarchit.*, 2012, pp. 155–165.
[29] A. Munir, S. Ranka, and A. Gordon-Ross, "High-performance energy-efficient multicore embedded computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 4, pp. 684–700, Apr. 2012.
[30] T. S. Muthkaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin, "Hierarchical power management for asymmetric multicor in dark silicon era," in *Proc. Des. Autom. Conf.*, 2013, pp. 1–6.
[31] T. Needham, "A visual explanation of Jensen's inequality," *Amer. Math. Monthly*, vol. 100, no. 8, pp. 768–771, 1993.
[32] S. Paek, W. Shin, J. Lee, H.-E. Kim, J.-S. Park, and L.-S. Kim, "Hybrid temperature sensor network for area-efficient on-chip thermal map sensing," *IEEE J. Solid-State Circuits*, vol. 50, no. 2, pp. 610–618, Feb. 2015.
[33] S. Park *et al.*, "Accurate modeling of the delay and energy overhead of dynamic voltage and frequency scaling in modern microprocessors," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 32, no. 5, pp. 695–708, May. 2013.
[34] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder, "Using SimPoint for accurate and efficient simulation," in *Proc. Int. Conf. Meas. Model. Comput. Syst.*, 2003, pp. 318–319.
[35] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin, "Power-performance modeling on asymmetric multi-cores," in *Proc. Int. Conf. Compilers Architectures Synthesis Embedded Syst.*, 2013, pp. 1–10.
[36] B. Salami, M. Baharani, and H. Noori, "Proactive migration with a self-adjusting migration threshold for dynamic thermal management of multi-core processors," *J. Supercomputing*, vol. 68, no. 3, pp. 1068–1087, 2014.
[37] W. Seo, D. Im, J. Choi, and J. Huh, "Big or little: A study of mobile interactive applications on an asymmetric multi-core platform," in *Proc. IEEE Int. Symp. Workload Characterization*, 2015, pp. 1–11.
[38] S. Sharifi, R. Ayoub, and T. S. Rosing, "TempoMP: Integrated prediction and management of temperature in heterogeneous MPSoCs," in *Proc. Des. Automat. Test Europe Conf.*, 2012, pp. 593–598.
[39] S. Sharifi, A. K. Coskun, and T. S. Rosing, "Hybrid dynamic energy and thermal management in heterogeneous embedded multiprocessor SoCs," in *Proc. Asia South Pacific Des. Automat. Conf.*, 2010, pp. 873–878.
[40] P.-H. Tseng, P.-C. Hsiu, C.-C. Pan, and T.-W. Kuo, "User-centric energy-efficient scheduling on multi-core mobile devices," in *Proc. Des. Automat. Conf.*, 2014, pp. 1–6.
[41] H. Usui, L. Subramanian, K. K.-W. Chang, and O. Mutlu, "DASH: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, 2016, Art. no. 65.
[42] Z. Wang, C. Bash, N. Tolia, M. Marwah, and X. Zhu, "Optimal fan speed control for thermal management of servers," in *Proc. ASME/Pacific Rim Tech. Conf. Exhib. Packag. Integr. Electron. Photon. Syst.*, 2009, pp. 709–719.
[43] E. W. Wachter, C. D. Bellefroid, K. R. Basireddy, A. K. Singh, B. M. Al-Hashimi, and G. Merrett, "Predictive thermal management for energy-efficient execution of concurrent applications on heterogeneous multicores," *IEEE Trans. Very Large-Scale Integr. Syst.*, vol. 27, no. 6, pp. 1404–1415, Jun. 2019.
[44] G. Xie, G. Zeng, Z. Li, R. Li, and K. Li, "Adaptive dynamic scheduling on multifunctional mixed-criticality automotive cyber-physical systems," *IEEE Trans. Veh. Technol.*, vol. 66, no. 8, pp. 6676–6692, Aug. 2017.
[45] Q. Xie, J. Kim, Y. Wang, D. Shin, N. Chang, and M. Pedram, "Dynamic thermal management in mobile devices considering the thermal coupling between battery and application processor," in *Proc. Int. Conf. Comput.-Aided Des.*, 2013, pp. 242–247.
[46] ARM, ARM Intelligent Power Allocation, 2015. [Online]. Available: https://developer.arm.com/tools-and-software/open-source-software/linux-kernel/intelligent-power-allocation.
[47] ARM, big.LITTLE Technology Moves Towards Fully Heterogeneous Global Task Scheduling, 2013. [Online]. Available: https://arm.com/why-arm/technologies/big-little
[48] EEMBC, The Embedded Microprocessor Benchmark, 1997. [Online]. Available: http://www.eembc.org
[49] Hardkernel, Odroid-XU3, 2014. [Online]. Available: https://www.hardkernel.com/ko/shop/odroid-xu3/
[50] Linux, A Simplified Thermal Framework for ARM Platforms, 2012. [Online]. Available: http://elinux.org/images/2/2b/A_New_Simplified_Thermal_Framework_For_ARM_Platforms.pdf
[51] Linux, Energy-Aware Scheduling (EAS) Project, 2014. [Online]. Available: https://www.Linaro.org/blog/energy-aware-scheduling-eas-project/
[52] Linux, Overview of the Current Approaches to Enhance the Linux Scheduler, 2013. [Online]. Available: https://www.youtube.com/watch?v=QV9_SGFTnKw&list=ULTiLPL-zMndM&index=1310
[53] Monsoon Solution, Monsoon Power Monitor, 2017. [Online]. Available: http://www.msoon.com/LabEquipment/PowerMonitor/
[54] Samsung Electronics, Exynos 5422, 2014. [Online]. Available: http://www.samsung.com/global/business/semiconductor/product/application/detail?productId=7978&iaId=2341



Young Geun Kim received the BS and PhD degrees in computer science from Korea University, in 2014 and 2018, respectively. He is currently a postdoctoral research associate with the Schools of Computing, Informatics, and Decision Systems Engineering, Arizona State University. His research interests include OS-level power/thermal management for embedded systems.

Minyong Kim received the BS, MS, and PhD degrees in computer science from Korea University, in 2010, 2012, and 2016 respectively. He is currently working toward the MS degree in the Graduate School of Design, Harvard University. His research interests include low-power design, user-aware design, and system-on-chip design.



Joonho Kong received the BS, MS, and PhD degrees in computer science from Korea University, in 2007, 2009, and 2011, respectively. He is currently an assistant professor with the School of Electronics Engineering at Kyungpook National University. His research interests include computer architecture design, processor cache design, and hardware security.

Sung Woo Chung (Senior Member, IEEE) received the BS, MS, and PhD degrees in electrical engineering and computer science from Seoul National University, in 1996, 1998, and 2003, respectively. He is currently a professor with the Department of Computer Science, Korea University. His research interests include low-power design, temperature-aware design, and user-aware design. He was an associate editor of *IEEE Transactions on Computers* from 2010 to 2015. He was the Technical Program co-chair of the *IEEE International Conference on Computer Design*, in 2015. He serves (and served) on the technical program committees in many conferences, including Design Automation Conference (2015–2018), International Symposium on Low Power Electronics and Design (2016–2019), International Parallel and Distributed Processing Symposium (2016–2020), and International Symposium on Memory Systems (2019).

Algorithms for Inversion Mod p^k

Çetin Kaya Koç, *Fellow, IEEE*

Abstract—This article describes and analyzes all existing algorithms for computing $x = a^{-1} \pmod{p^k}$ for a prime p, and also introduces a new algorithm based on the exact solution of linear equations using p-adic expansions. The algorithm starts with the initial value $c = a^{-1} \pmod{p}$ and iteratively computes the digits of the inverse $x = a^{-1} \pmod{p^k}$ in base p. The mod 2 version of the algorithm is more efficient than all existing algorithms for small values of k. Moreover, it stands out as being the only one that works for any p, any k, and digit-by-digit. While the new algorithm is asymptotically worse off, it requires the minimal number of arithmetic operations (just a single addition) per step, as compared to all existing algorithms.

Index Terms—Number-theoretic algorithms, computer arithmetic, multiplicative inverse

1 Introduction

Hardware and software realizations of public-key cryptographic algorithms require implementations the multiplicative inverse mod p (prime) or n (composite). When the modulus is prime, we can compute the multiplicative inverse using Fermat's method as $a^{-1} = a^{p-2} \pmod{p}$. When it is composite, we can use Euler's method to compute the multiplicative inverse as $a^{-1} = a^{\phi(n)-1} \pmod{n}$, provided that we know or can compute $\phi(n)$.

On the other hand, the extended euclidean algorithm (EEA) works for both prime and composite modulus, and does not require the knowledge of ϕ. The classical EEA requires division operations at each step, which is costly. On the other hand, variations of the binary extended euclidean algorithms use shift, addition and subtraction operations [9], [14], [15]. We must note however that most inversion algorithms are variants of the classical euclidean algorithm for computing the greatest common divisor of two integers $g = \gcd(a, n)$.

2 Inversion Mod 2^k

The Montgomery multiplication algorithm is introduced by Peter Montgomery [13] in 1985. It computes the product $c = a \cdot b \cdot r^{-1} \pmod{n}$ for an arbitrary modulus n, without actually performing any mod n reductions. Interestingly, the algorithm does not directly need $r^{-1} \pmod{n}$, but it requires another quantity n' which is one of the numbers produced by the extended euclidean algorithm with inputs 2^k and n:

$$\begin{aligned} (u, n') &\leftarrow \text{EEA}(2^k, n) \\ u \cdot 2^k - n' \cdot n &= 1 \\ n' &= -n^{-1} \pmod{2^k}. \end{aligned}$$

In other words, the Montgomery multiplication algorithm requires the computation of $n^{-1} \pmod{2^k}$ rather than $r^{-1} \pmod{n}$. We may expect that inversion with respect to a special modulus such as 2^k might be easier than inversion with respect to an arbitrary modulus. Indeed this is the case. Several algorithms for computing multiplicative inverse mod 2^k appeared in the literature some of which are significantly simpler than the classical EEA algorithm.

3 Suffix Property of Inverse Mod 2^k and p^k

Given $x = a^{-1} \pmod{2^k}$, we can compute $y = a^{-1} \pmod{2^j}$ for $1 \le j < k$ by reduction: $y = x \pmod{2^j}$. We can easily prove that y is the inverse of a mod 2^j for some $j \in [1, k)$, by noting that $a \cdot x = 1 \pmod{2^k}$ implies $a \cdot x = 1 + N \cdot 2^k$ for some integer N; when we reduce both sides mod 2^j, we obtain:

$$\begin{aligned} a \cdot \sum_{i=0}^{k-1} X_i \cdot 2^i &= 1 + N \cdot 2^k \pmod{2^j} \\ a \cdot \sum_{j=0}^{j-1} X_i \cdot 2^i &= 1 \pmod{2^j}. \end{aligned}$$

Therefore, we conclude that $y = a^{-1} \pmod{2^j}$. Moreover if $x = a^{-1} \pmod{2^k}$ is expressed as a k-bit binary number $x = (X_{k-1} \cdots X_1 X_0)$, then the suffixes (the least significant bits) of x are actually the inverses mod 2^j for $j = 1, 2, \ldots, k-2$. That is, (X_0) is the inverse of a mod 2, and $(X_1 X_0)$ is the inverse of a mod 2^2, and so on, up to $k-1$.

For the case of p^k, we note that $a \cdot x = 1 \pmod{p^k}$ implies $a \cdot x = 1 + N \cdot p^k$ for some integer N, and therefore, when we reduce both sides mod p^j, we obtain:

$$\begin{aligned} a \cdot \sum_{i=0}^{k-1} X_i \cdot p^i &= 1 + N \cdot p^k \pmod{p^j} \\ a \cdot \sum_{i=0}^{i-1} X_i \cdot p^i &= 1 \pmod{p^j}. \end{aligned}$$

If the inverse x is expressed in base p, we have $X_i \in [0, p-1]$ and $x = (X_{k-1} \cdots X_1 X_0)$, and thus, the inverse mod p^j is equal to $(X_{j-1} \cdots X_1 X_0)$. In other words, the suffix property also holds for the inverse mod p^k, provided that the inverse x mod p^k is expressed in base p.

To summarize: if $x = a^{-1} \pmod{2^k}$ is available, we can reduce it mod 2^j to obtain $a^{-1} \pmod{2^j}$ for any $j \in [1, k-1]$. If x is expressed in binary as $x = (X_{k-1} \cdots X_1 X_0)$, then the inverse mod 2^j is simply the j-bit suffix of x as $(X_{j-1} \cdots X_1 X_0)$. Similarly, if $x = a^{-1} \pmod{p^k}$ is available, we can reduce it mod p^j to obtain $a^{-1} \pmod{p^j}$ for any $j \in [1, k-1]$. If x is expressed in base p as $x = (X_{k-1} \cdots X_1 X_0)$, then the inverse mod p^j is simply the j-digit suffix of x as $(X_{j-1} \cdots X_1 X_0)$.

4 Existing Inversion Algorithms

There are several algorithms in the literature. Dussé and Kaliski [5] gave an efficient algorithm for computing the inverse $x = a^{-1} \pmod{2^k}$ for an odd a, therefore, $\gcd(a, 2^k) = 1$. Arazi and Qi [1] review 3 known algorithms (as of 2008), and introduce a new algorithm (Algorithm 4) for computing $a^{-1} \pmod{2^k}$, where $k = 2^s$. Furthermore, Dumas proved [3], [4] that Algorithm 4 in [1] is a specific case of Hensel lifting [12], and introduced an iterative formula for computing $x = a^{-1} \pmod{p^k}$, where $k = 2^s$. In this section, we describe these algorithms.

4.1 Dussé and Kaliski Algorithm

Dussé and Kaliski algorithm [5] is based on a specialized version of the extended euclidean algorithm for computing the inverse. The pseudocode is given below [5], [10].

• *The author is with İstinye University, 34010 İstanbul, Turkey, and the Nanjing University of Aeronautics and Astronautics, Nanjing, Jiangsu 210016, China, and also with the University of California Santa Barbara, Santa Barbara, CA 93106. E-mail: cetinkoc@ucsb.edu.*

Manuscript received 28 June 2017; revised 20 Apr. 2018; accepted 24 Apr. 2018. Date of publication 30 Jan. 2020; date of current version 8 May 2020.
(Corresponding author: Çetin Kaya Koç.)
Recommended for acceptance by W. Liu.
Digital Object Identifier no. 10.1109/TC.2020.2970411

TABLE 1
Dussé and Kaliski Algorithm for Computing $23^{-1} \pmod{2^6}$

i	2^{i-1}	2^i	x	$a \cdot x \pmod{2^i}$	$2^{i-1} \stackrel{?}{<} a \cdot x$	x
2	2	4	1	$(23 \cdot 1 \bmod 4) \to 3$	$2 < 3$	$1 + 2 = 3$
3	4	8	3	$(23 \cdot 3 \bmod 8) \to 5$	$4 < 5$	$3 + 4 = 7$
4	8	16	7	$(23 \cdot 7 \bmod 16) \to 1$	$8 \not< 1$	7
5	16	32	7	$(23 \cdot 7 \bmod 32) \to 1$	$16 \not< 1$	7
6	32	64	7	$(23 \cdot 7 \bmod 64) \to 33$	$32 < 33$	$7 + 32 = 39$

function DusseKaliski$(a, 2^k)$
input: a, k where a is odd and $a < 2^k$
output: $x = a^{-1} \bmod 2^k$
1: $x \leftarrow 1$
2: **for** $i = 2$ **to** k
2a: **if** $2^{i-1} < a \cdot x \pmod{2^i}$
2aa: $x \leftarrow x + 2^{i-1}$
3: **return** x

As an example, consider the computation of $23^{-1} \pmod{2^6}$ illustrated in Table 1. Here, we have $a = 23$ and $k = 6$, and we start with $x = 1$.

At the end of the algorithm we find $x = 39$, implying $23^{-1} = 39 \pmod{2^6}$; this is indeed correct since $23 \cdot 39 = 1 \pmod{2^6}$. On the other hand, the inverses mod 2^j for $j = 1, 2, 3, 4, 5$ can be obtained by reduction $39 \pmod{2^j}$. We can also compute them using the suffix property, by expressing 39 in binary as $(100111)_2$, and taking its suffixes. However, we notice that the Dussé and Kaliski algorithm already computes consecutive inverses $23^{-1} \pmod{2^i}$ for $i = 1, 2, 3, 4, 5, 6$ as $(1)_2 = 1$, $(11)_2 = 3$, $(111)_3 = 7$, $(0111)_3 = 7$, $(00111)_3 = 7$, and $(100111)_3 = 39$.

These consecutive inverses are computed *in whole* at each step (rather than bit-by-bit, as we will see some other algorithms do). The j-bit inverse $23^{-1} \pmod{2^j}$ is computed at the jth step. This property affects the performance, since the entire j-bit number is computed (rather than a single bit).

4.2 Algorithm 2 in Arazi and Qi Paper

Arazi and Qi review three existing algorithms, and introduce a new algorithm in their paper [1]. All 4 algorithms in [1] compute $x = a^{-1} \pmod{2^k}$. First of all, Algorithm 1 is Dussé and Kaliski algorithm which we have already covered.

Algorithm 2 is described in the narrative of the article [1] without explicitly giving its steps. We find it useful to describe this algorithm and give its pseudocode. Assume a and x are k-bit binary numbers. Since a and x are both odd, i.e., $A_0 = X_0 = 1$, they can be written as

$$a = (A_{k-1}A_{k-2} \cdots A_1A_0) = (A_{k-1}A_{k-2} \cdots A_11)$$
$$x = (X_{k-1}X_{k-2} \cdots X_1X_0) = (X_{k-1}X_{k-2} \cdots X_11)$$

The main idea of Algorithm 2 is that the equality

$$a \cdot x = 1 = (00 \cdots 01)_2 \pmod{2^k},$$

implies that the least significant k bits of $y = a \cdot x$ is equal to $(00 \cdots 01)_2$, and y can be written as

$$y = a \cdot x = (\overbrace{Z_{k-1} \cdots Z_1Z_0}^{k \text{ bits}}\ \overbrace{00 \cdots 01}^{k \text{ bits}})_2. \quad (1)$$

Our aim is to compute the remaining bits of x, i.e., X_i for $i = 1, 2, \ldots, k-1$, making sure that as y is iteratively computed, its least significant k bits become equal to $(00 \cdots 01)_2$ according to Equation (1).

Notice that the LSB of a is 1, and thus, the ith bit of $2^i \cdot a$ is equal to 1 for any $i \in [1, k-1]$. Iterative computation of y is accomplished by starting with $y = a$, adding $2^i \cdot a$ to y if $Y_i = 1$, since this would make the resulting Y_i zero. By proceeding to the left, we make all $Y_i = 0$ for $i = 1, 2, \ldots, k-1$, except $Y_0 = 1$. The steps of Algorithm2 are given below. It computes the bits of the inverse x from the least significant to the most significant bit, at the ith step either adding $2^i \cdot a$ to y or not, and determining X_i as 1 or zero.

TABLE 2
Algorithm 2 for Computing $23^{-1} \pmod{2^6}$

i	y	Y_i	$y = y + 2^i \cdot a$	X_i
0	$23 = (000000\ 010111)$	1	$y = 23$	1
1	$23 = (000000\ 010111)$	1	$y = 23 + 2 \cdot 23 \to 69$	1
2	$69 = (000001\ 000101)$	1	$y = 69 + 2^2 \cdot 23 \to 161$	1
3	$161 = (000010\ 100001)$	0	$y = 161$	0
4	$161 = (000010\ 100001)$	0	$y = 161$	0
5	$161 = (000010\ 100001)$	1	$y = 161 + 2^5 \cdot 23 \to 897$	1
	$897 = (001110\ 000001)$			

function Algorithm2$(a, 2^k)$
input: a, k where a is odd and $a < 2^k$
output: $x = a^{-1} \bmod 2^k$
1: $y \leftarrow a$
2: $X_0 \leftarrow 1$
3: **for** $i = 1$ **to** $k - 1$
3a: **if** $Y_i = 1$
3aa: $y \leftarrow y + 2^i \cdot a$
3ab: $X_i \leftarrow 1$
3b: **else**
3ba: $X_i \leftarrow 0$
4: **return** $x = (X_{k-1} \cdots X_1X_0)_2$

The computation of $23^{-1} \pmod{2^6}$ using Algorithm 2 is illustrated in Table 2. The initial value of y is $a = 23$, and at each step Y_i is checked; if $Y_i = 1$, then $2^i \cdot a$ is added to y. As the progress of the algorithm shows, the lower $k = 6$ bits of y eventually becomes (000001). The inverse is computed as $x = (100111)_2 = 39$. This is indeed correct since $23 \cdot 39 = 1 \pmod{2^6}$.

Algorithm 2 computes the inverse $x = a^{-1} \pmod{2^k}$ *bit by bit*. At the jth step, the jth bit of x is computed. Hence, the inverse mod 2^j becomes available at the jth step: $(X_{j-1} \cdots X_1X_0)$ is the inverse mod 2^j.

4.3 Algorithm 3 in Arazi and Qi Paper

Arazi and Qi describe Algorithm 3 in detail [1], and give pseudocode. This algorithm has two stages: in the first stage which is called Algorithm 3a, the quantity $-v = (2^k)^{-1} \pmod{a}$ is computed. In the second stage (Algorithm 3b), the quantity $-v$ is used to compute $x = a^{-1} \pmod{2^k}$. This algorithm is essentially the extended euclidean algorithm. Given $\gcd(a, 2^k) = 1$, the EEA computes

$$(x, v) \leftarrow \text{EEA}(a, 2^k)$$
$$x \cdot a - v \cdot 2^k = 1$$
$$a^{-1} = x \pmod{2^k}$$
$$(2^k)^{-1} = -v \pmod{a}$$

After $-v$ is available, we can compute x using the identity

$$x = \frac{1 + v \cdot 2^k}{a},$$

which requires a shift (the computation of $v \cdot 2^k$), an increment operation, and a division by a operation (which is very expensive). Algorithm 3 is the least efficient of all 4 algorithms in [1], since it requires a full division with k-bit integers in the second stage of the algorithm.

TABLE 3
Steps 1 and 2 of Algorithm 3 for Computing $23^{-1} \pmod{2^6}$

i	v	V_0	$v = v + a$	$v = v/2$
0	$1 = (000001)$	1	$v = 1 + 23 \rightarrow 24$	$v = 24/2 \rightarrow 12$
1	$12 = (001100)$	0	$v = 12$	$v = 12/2 \rightarrow 6$
2	$6 = (000110)$	0	$v = 6$	$v = 6/2 \rightarrow 3$
3	$3 = (000011)$	1	$v = 3 + 23 \rightarrow 26$	$v = 26/2 \rightarrow 13$
4	$13 = (001101)$	1	$v = 13 + 23 \rightarrow 36$	$v = 36/2 \rightarrow 18$
5	$18 = (010010)$	0	$v = 18$	$v = 18/2 \rightarrow 9$

The computation of $-v = (2^k)^{-1} \pmod{a}$ for an odd a is quite easy, due to the Montgomery reduction algorithm called the Coarsely Integrated Operand Scanning (CIOS) whose details are found in [11].

Writing it as $-v = 2^{-k} \pmod{a}$, we first compute this quantity $v = (V_{k-1} \cdots V_1 V_0)$ using the CIOS algorithm at the end of Step 2; we then compute the inverse x in Step 3.

function Algorithm3$(a, 2^k)$
input: a, k where a is odd and $a < 2^k$
output: $x = a^{-1} \bmod 2^k$
1: $v \leftarrow 1$
2: **for** $i = 0$ **to** $k - 1$
2a: **if** $V_0 = 1$
2aa: $v \leftarrow v + a$
2b: $v \leftarrow v/2$
3: $x \leftarrow (1 + v \cdot 2^k)/a$
4: **return** x

The correctness of Algorithm 3 depends on the fact that the quantity $(1 + v \cdot 2^k)$ is divisible by a. This is easily proved by noting that $-v = 2^{-k} \pmod{a}$ implies $-v \cdot 2^k = 1 \pmod{a}$, and thus, $-v \cdot 2^k = 1 + N \cdot a$ for some integer N. Therefore, $1 + v \cdot 2^k = -N \cdot a$.

Steps 1 and 2 of Algorithm 3 for computing $23^{-1} \pmod{2^6}$ is illustrated in Table 3. The initial value is $v = 1$, and at each step V_0 is checked; if $V_0 = 1$, then a is added to v, and v is shifted to left (i.e., divided by 2).

At the end of Step 2 for $i = 5$, we obtain $-v = 9$. In Step 3, we use the formula $(1 + v \cdot 2^k)/a$ and the value of $-v = 9$, to compute the inverse as $x = (1 + (-9) \cdot 2^6)/23 = -25$, which is equal to $39 \pmod{2^6}$. This inverse is computed *in whole* in a single step, using a shift, an addition and a division operation involving k-bit numbers. On the other hand, the inverses mod 2^i for $i \in [1, k-1]$ can be computed only after Step 3 is completed, by reducing $x \bmod 2^i$.

4.4 Algorithm 4 in Arazi and Qi Paper

Algorithm 4 is the last one described in [1], and it is presented as the authors' contribution. It is based on the idea that, given $a = (a_H a_L) = a_H \cdot 2^i + a_L$ where a_H and a_L are the upper and lower i bits of the $2i$-bit binary number a, the inverse $x = a^{-1} \pmod{2^{2i}}$ can be computed from the inverse of $a_L \bmod 2^i$. Algorithm 4 computes the inverse of $a \bmod 2^k$ where k is a power of 2, that is, it computes $x = a^{-1} \pmod{2^{2^s}}$, and it accomplishes this computation in $s = \log_2(k)$ steps. In other words, the number of steps of Algorithm 4 is logarithmic in k.

Given $a = (a_H a_L) = a_H \cdot 2^i + a_L$ and $x = (x_H x_L) = x_H \cdot 2^i + x_L$, we assume $x_L = a_L^{-1} \pmod{2^i}$ is already computed and available. Note that a_H, a_L, x_H, x_L are all i-bit integers. Algorithm 4 computes the upper part x_H of the inverse $x = a^{-1} \pmod{2^{2i}}$ in 3 steps:

1) Compute the product $a_L \cdot x_L = (b_H b_L) = b_H \cdot 2^i + b_L = b_H \cdot 2^i + 1$.
2) Compute the product $a_H \cdot x_L = (c_H c_L) = c_H \cdot 2^i + c_L$.
3) Compute the expression $x_H = -(b_H + c_L) \cdot x_L \pmod{2^i}$.
4) The inverse is given as $x = (x_H x_L) = x_H \cdot 2^i + x_L$.

An algebraic proof is given in [1]. Here we illustrate this method for the 32-bit number $a = 2583209455 = (99f8a5ef)_{16}$. This gives $a_H = 39416 = (99f8)_{16}$ and $a_L = 42479 = (a5ef)_{16}$. Furthermore, we assume the inverse of the lower part a_L mod 2^{16} is already computed and available: $x_L = a_L^{-1} \pmod{2^{16}}$ as $x_L = 10511 = (290f)_{16}$. We then compute x_H using

1) $a_L \cdot x_L = 42479 \cdot 10511 = 446496769 = (1a9d0001)_{16} = (b_H b_L)$. This gives $b_H = 6813 = (1a9d)_{16}$ and $b_L = 1$.
2) $a_H \cdot x_L = 39416 \cdot 10511 = 414301576 = (18b1bd88)_{16} = (c_H c_L)$. This gives $c_H = (18b1)_{16} = 6321$ and $c_L = (bd88)_{16} = 48520$.
3) $x_H = -(6813 + 48520) \cdot 10511 \pmod{2^{16}}$. This gives $x_H = 26837 = (68d5)_{16}$.
4) The inverse: $x = (x_H x_L) = (68d5290f)_{16} = 1758800143$. This is indeed correct $2583209455 \cdot 1758800143 = 1 \pmod{2^{32}}$.

Algorithm 4 is a essentially a recursive algorithm. The inverse of $a \bmod 2^{32}$ invokes the computation of the inverse $a \bmod 2^{16}$, which the computation of the inverse $a \bmod 2^8$, and so on. However, it can also be made iterative by first computing the inverse mod 2^1, using this inverse to compute the inverse mod 2^2, and then mod 2^4, and so on. The authors describe Algorithm 4 in the narrative of the article [1], however they do not provide a pseudocode. Below we give the pseudocode for computing the inverse mod 2^k for $k = 2^s$. The binary expansion of a is expressed as $a = (A_{k-1} \cdots A_1 A_0)$ and $k = 2^s$ for some integer s.

function Algorithm4$(a, 2^k)$
input: a, k where a is odd, $a < 2^k$, and $k = 2^s$
output: $x = a^{-1} \bmod 2^k$
1: $a_L \leftarrow A_0$
2: $a_H \leftarrow A_1$
3: $x_L \leftarrow 1$
4: **for** $i = 1$ **to** s
4a: $(b_H b_L) \leftarrow a_L \cdot x_L$
4b: $(c_H c_L) \leftarrow a_H \cdot x_L$
4c: $x_H \leftarrow -(b_H + c_L) \cdot x_L \pmod{2^{2^{i-1}}}$
4d: $a_L \leftarrow (A_{2^i-1} \cdots A_0)_2$
4e: $a_H \leftarrow (A_{2^{i+1}-1} \cdots A_{2^i})_2$
4f: $x_L \leftarrow (x_H x_L)$
4: **return** $x = (x_H x_L)$

Table 4 illustrates the inverse computation $x = a^{-1} \pmod{2^{32}}$ for $a = (99f8a5ef)_{16}$, where $s = 5$. The algorithm computes the inverse $x = a^{-1} \pmod{2^{32}}$, by successively computing the inverse mod 2^i for $i = 1, 2, 4, 8, 16, 32$.

TABLE 4
Algorithm 4 for Computing $(99f8a5ef)_{16}^{-1} \pmod{2^{32}}$

s	$(a_H\ a_L)$	x_L	$(b_H\ b_L) \leftarrow a_L \cdot x_L$	$(c_H\ c_L) \leftarrow a_H \cdot x_L$	x_H	$(x_H\ x_L)$
1	$(1\ 1)_2$	$(1)_2$	$(0\ 1)_2$	$(0\ 1)_2$	$(1)_2$	$(1\ 1)_2$
2	$(11\ 11)_2$	$(11)_2$	$(10\ 01)_2$	$(10\ 01)_2$	$(11)_2$	$(11\ 11)_2$
3	$(e\ f)_{16}$	$(f)_{16}$	$(e\ 1)_{16}$	$(d\ 2)_{16}$	$(0)_{16}$	$(0\ f)_{16}$
4	$(a5\ ef)_{16}$	$(0f)_{16}$	$(0e\ 01)_{16}$	$(09\ ab)_{16}$	$(29)_{16}$	$(29\ 0f)_{16}$
5	$(99f8\ a5ef)_{16}$	$(290f)_{16}$	$(1a9d\ 0001)_{16}$	$(18b1\ bd88)_{16}$	$(68d5)_{16}$	$(68d5\ 290f)_{16}$

TABLE 5
Dumas Iteration for Computing $12^{-1} \pmod{5^{16}}$

i	x_{i-1}	p^{2^i}	$x_i = x_{i-1} \cdot (2 - a \cdot x_{i-1}) \bmod p^{2^i}$
1	$x_0 = 3$	5^2	$x_1 = 3 \cdot (2 - 12 \cdot 3) \rightarrow 23$
2	$x_1 = 23$	5^4	$x_2 = 23 \cdot (2 - 12 \cdot 23) \rightarrow 573$
3	$x_2 = 573$	5^8	$x_3 = 573 \cdot (2 - 12 \cdot 573) \rightarrow 358073$
4	$x_3 = 358073$	5^{16}	$x_4 = 358073 \cdot (2 - 12 \cdot 358073) \rightarrow 139872233073$

TABLE 6
Dumas Iteration for Computing $23^{-1} \pmod{2^{32}}$

i	x_{i-1}	2^{2^i}	$x_i = x_{i-1} \cdot (2 - a \cdot x_{i-1}) \bmod 2^{2^i}$
1	$x_0 = 1$	2^2	$x_1 = 1 \cdot (2 - 23 \cdot 1) \rightarrow 3$
2	$x_1 = 3$	2^4	$x_2 = 3 \cdot (2 - 23 \cdot 3) \rightarrow 7$
3	$x_2 = 7$	2^8	$x_3 = 7 \cdot (2 - 23 \cdot 7) \rightarrow 167$
4	$x_3 = 167$	2^{16}	$x_4 = 167 \cdot (2 - 23 \cdot 167) \rightarrow 14247$
5	$x_4 = 14247$	2^{32}	$x_5 = 14247 \cdot (2 - 23 \cdot 14247) \rightarrow 3921491879$

The result is indeed correct since $(99f8a5ef)_{16} \cdot (68d5290f)_{16} = 1 \pmod{2^{32}}$. Algorithm 4 also computes $a^{-1} \bmod 2^{2^i}$ for $i = 0, 1, 2, 3, 4, 5$ at every step:

$$
\begin{aligned}
(99f8a5ef)_{16}^{-1} &= (1)_2 \pmod 2 \\
(99f8a5ef)_{16}^{-1} &= (11)_2 \pmod{2^2} \\
(99f8a5ef)_{16}^{-1} &= (f)_{16} \pmod{2^4} \\
(99f8a5ef)_{16}^{-1} &= (0f)_{16} \pmod{2^8} \\
(99f8a5ef)_{16}^{-1} &= (290f)_{16} \pmod{2^{16}} \\
(99f8a5ef)_{16}^{-1} &= (68d5290f)_{16} \pmod{2^{32}}.
\end{aligned}
$$

It is not clear if Algorithm 4 *as formulated* can be generalized for an arbitrary k; it seems that it cannot be. There are s steps in the algorithm, and at step i the inverse mod 2^{2^i} computed for $i = 1, 2, \ldots, s$. The authors describe a method (without detail) in Section 2.2 of [1] for dealing with a composite k, but they do not give a method for computing the inverse for an arbitrary k. The inverse mod 2^k for an arbitrary (not a power of 2) is not directly computed by this algorithm. However, the inverse mod 2^k for an arbitrary k can be obtained by first computing the inverse mod 2^{2^s} for the nearest $2^s > k$, and then reducing the result mod 2^k. For example, if we need $a^{-1} \bmod 2^{29}$, then we will have to compute the inverse mod 2^{2^5} first, since $2^5 > 29$.

4.5 Newton-Raphson Iteration by Dumas

Dumas in [3], [4] shows that Algorithm 4 given by Arazi and Qi [1] is actually a specific case of Hensel lifting [12], and provides a proof of the derivation of it. Dumas also gives Hensel's lemma mod p^k and its proof from Newton-Raphson iteration. This results in several formulas for computing $a^{-1} \pmod{2^k}$ for $k = 2^s$, one of which is Algorithm 4. Dumas studies different implementation variants of this iteration and shows that the explicit formula works well for small exponent values but it is slower for large exponent, for example, more than 700 bits. An important contribution of Dumas is an iterative formula which computes $x_s = a^{-1} \pmod{p^{2^s}}$ for a prime p, by iterating over $i = 1, 2, \ldots, s$ as

$$
\begin{aligned}
x_0 &= a^{-1} \pmod p \\
x_i &= x_{i-1} \cdot (2 - a \cdot x_{i-1}) \bmod p^{2^i}.
\end{aligned}
$$

By selecting $p = 2$, the formula also specializes to the binary case. The number of steps of the iteration is $s = \log_2(k)$. Below we illustrate the computation of $x_s = a^{-1} \pmod{p^{2^s}}$ for $a = 12$, $p = 5$, and $s = 4$. The iteration starts with $x_0 = 12^{-1} \pmod 5$, which is found as $x_0 = 3$, and proceeds over $i = 1, 2, 3, 4$, as shown in Table 5.

The result $x_4 = 139872233073$ is indeed correct since $12 \cdot 139872233073 = 1 \pmod{5^{16}}$. We note that during its iteration the Dumas algorithm actually computes consecutive inverses $12^{-1} \pmod{5^{2^i}}$ for $i = 0, 1, 2, 3, 4$:

$$
\begin{aligned}
12^{-1} &= 3 \pmod 5 \\
12^{-1} &= 23 \pmod{5^2} \\
12^{-1} &= 573 \pmod{5^4} \\
12^{-1} &= 358073 \pmod{5^8} \\
12^{-1} &= 139872233073 \pmod{5^{16}}.
\end{aligned}
$$

However, inverses modulo other powers of 5 are not computed. While the algorithm takes $s = \log_2(k)$ steps, it also computes $s = \log_2(k)$ inverses. However, the inverse mod p^k for an arbitrary k can be obtained by first computing the inverse mod p^{2^s} for the nearest $2^s > k$, and then reducing the result mod 2^k. For example, if we need $a^{-1} \bmod p^{29}$, then we will have to compute the inverse mod p^{2^5} first, since $2^5 > 29$.

The binary version of the Dumas algorithm is similar, but it is more compact than Algorithm 4. It uses the same formula as for p, but taking $p = 2$ and assuming that a is odd. The starting value $x_0 = 1$ since $p = 2$ and a is odd. Below we illustrate the computation of $x_s = a^{-1} \pmod{p^{2^s}}$ for $a = 23$, $p = 2$, and $s = 5$. The iteration starts with $x_0 = 23^{-1} \pmod 2$, which is found as $x_0 = 1$, and proceeds over $i = 1, 2, 3, 4, 5$ by computing $x_i = x_{i-1} \cdot (2 - a \cdot x_{i-1}) \bmod 2^{2^i}$.

The result $x_5 = 3921491879$ is indeed correct since $23 \cdot 3921491879 = 1 \pmod{2^{16}}$. We note that during its iteration the Dumas algorithm actually computes $13^{-1} \pmod{2^{2^i}}$ for $i = 0, 1, 2, 3, 4, 5$, as shown in Table 6:

$$
\begin{aligned}
23^{-1} &= 1 \pmod 2 \\
23^{-1} &= 3 \pmod{2^2} \\
23^{-1} &= 7 \pmod{2^4} \\
23^{-1} &= 167 \pmod{2^8} \\
23^{-1} &= 14247 \pmod{2^{16}} \\
23^{-1} &= 3921491879 \pmod{2^{32}},
\end{aligned}
$$

However, inverses modulo other powers of 2 are not computed. Similarly, the inverse mod 2^k for an arbitrary k can be obtained by first computing the inverse mod 2^{2^s} for the nearest $2^s > k$, and then reducing the result mod 2^k.

TABLE 7
ModInverse Algorithm for Computing $12^{-1} \pmod{5^5}$

i	b_i	$X_i = c \cdot b_i \bmod p$	$b_{i+1} = (b_i - a \cdot X_i)/p$
0	$b_0 = 1$	$X_0 = (3 \cdot 1 \bmod 5) \rightarrow 3$	$b_1 = (1 - 12 \cdot 3)/5 \rightarrow -7$
1	$b_1 = -7$	$X_1 = (3 \cdot (-7) \bmod 5) \rightarrow 4$	$b_2 = (-7 - 12 \cdot 4)/5 \rightarrow -11$
2	$b_2 = -11$	$X_2 = (3 \cdot (-11) \bmod 5) \rightarrow 2$	$b_3 = (-11 - 12 \cdot 2)/5 \rightarrow -7$
3	$b_3 = -7$	$X_3 = (3 \cdot (-7) \bmod 5) \rightarrow 4$	$b_4 = (-7 - 12 \cdot 4)/5 \rightarrow -11$
4	$b_4 = -11$	$X_4 = (3 \cdot (-11) \bmod 5) \rightarrow 2$	...

However, it turns we do not need to compute the inverses mod up to p^{2^s} for $2^s > k$ in order to obtain the inverse mod p^k for an arbitrary $k < 2^s$. As suggested by one of the Reviewers, we can compute the inverses up to mod $p^{2^{s-1}}$ for the nearest $2^{s-1} < k$ (rather than $2^s > k$), and then apply one additional iteration for $i = s$

$$x_s = x_{s-1} \cdot (2 - a \cdot x_{s-1}) \pmod{p^k},$$

which is computed mod p^k (rather than mod p^{2^s}).

5 A New Algorithm for Inversion Mod p^k

We introduce a new algorithm for computing $x = a^{-1} \pmod{p^k}$ for a prime p and arbitrary positive integer k. Our algorithm relies on Dixon's algorithm [2] for exact solution linear equations using p-adix expansions, whose general idea is credited to German mathematician Kurt Wilhelm Sebastian Hensel. Dixon's algorithm aims to exactly solve a linear system of equations with integer coefficients, such as $\mathsf{A} \cdot \mathsf{x} = \mathsf{b}$ in the sense that the solutions are obtained as rational numbers rather than approximate values using floating-point arithmetic.

Similar to Dixon's approach, we formulate the inversion problem as the exact solution of the linear equation

$$a \cdot x = 1 \pmod{p^k},$$

for a prime p, an arbitrary positive integer $k > 1$ and $\gcd(a, p) = 1$ or $1 < a < p$. By solving this equation, we compute the inverse $x = a^{-1} \pmod{p^k}$. The algorithm starts with the computation of

$$c = a^{-1} \pmod{p},$$

using the extended euclidean algorithm. It is more often the case that the prime p is small, thus, this computation may not constitute a bottleneck. In fact, the computation of c for the case of $p = 2$ is trivial, since $c = 1$ for any odd a. The algorithm then iteratively finds the digits of x expressed in *base* p such that $x = a^{-1} \pmod{p^k}$. In other words, the algorithm computes the vector $(X_{k-1} \cdots X_1 X_0)_p$ with $X_i \in [0, p-1]$ such that

$$x = \sum_{i=0}^{k-1} X_i \cdot p^i = X_0 + X_1 \cdot p + X_2 \cdot p^2 + \cdots + X_{k-1} \cdot p^{k-1},$$

function ModInverse(a, p^k)
input: a, p, k where $\gcd(a, p) = 1$ and $a < p^k$
output: $x = a^{-1} \bmod p^k$
1: $c \leftarrow a^{-1} \pmod{p}$
2: $b_0 \leftarrow 1$
3: **for** $i = 0$ **to** $k - 1$
3a: $X_i \leftarrow c \cdot b_i \pmod{p}$
3b: $b_{i+1} \leftarrow (b_i - a \cdot X_i)/p$
4: **return** $x = (X_{k-1} \cdots X_1 X_0)_p$

Consider the computation of $12^{-1} \pmod{5^5}$. We have $a = 12$, $p = 5$, and $k = 5$. First we compute $c = a^{-1} \pmod{p}$, which is found as $c = 12^{-1} = 2^{-1} = 3 \pmod{5}$. Starting with the initial value $b_0 = 1$, the algorithm proceeds for $i = 0, 1, 2, 3, 4$ as illustrated in Table 7. The algorithm computes x expressed in base 5 as $x = (X_4 X_3 X_2 X_1 X_0)_5 = (24243)_5$. In decimal, this is equal to $2 \cdot 5^4 + 4 \cdot 5^3 + 2 \cdot 5^2 + 4 \cdot 5 + 3 = 1823$. Indeed $12^{-1} = 1823 \pmod{5^5}$ since $12 \cdot 1823 = 1 \pmod{5^5}$.

Our algorithm actually computes $12^{-1} \pmod{5^j}$ for $j = 1, 2, 3, 4, 5$ at each step, since it generates the base 5 digits of the inverse as $x = (X_4 X_3 X_2 X_1 X_0)_5 = (24243)_5$. The inverses for 5^j are the suffixes of the inverse $x = (24243)_5$, given as

$$\begin{aligned}
12^{-1} &= (3)_5 \;=\; 3 \pmod{5} \\
12^{-1} &= (43)_5 \;=\; 23 \pmod{5^2} \\
12^{-1} &= (243)_5 \;=\; 73 \pmod{5^3} \\
12^{-1} &= (4243)_5 \;=\; 573 \pmod{5^4} \\
12^{-1} &= (24243)_5 \;=\; 1823 \pmod{5^5}.
\end{aligned}$$

6 Correctness of ModInverse

First of all, the term $(b_i - a \cdot X_i)$ in Step 3b is divisible by p for every i since

$$b_i - a \cdot X_i = b_i - a \cdot c \cdot b_i = b_i - b_i = 0 \pmod{p},$$

due to the fact that $a \cdot c = 1 \pmod{p}$. Therefore, b_i is integer for every $i \in [0, k-1]$. It also follows that when $i = 0$, the term $(b_0 - a \cdot X_0) = (1 - a \cdot c)$ is divisible by p. Furthermore, the terms b_i and x_i are found as

$$\begin{aligned}
b_i &= (1 - a \cdot c)^i / p^i \\
b_i \cdot p^i &= (1 - a \cdot c)^i \\
X_i &= c \cdot b_i \pmod{p},
\end{aligned}$$

for $i = 0, 1, \ldots, k-1$. The identity for b_i can be proven by induction on i.

The Basis Step: For $i = 0$, we have

$$\begin{aligned}
b_0 &= 1 \\
X_0 = c \cdot b_0 \;&=\; c \pmod{p}.
\end{aligned}$$

These follow from Step 2 and Step 3a of the algorithm for $i = 0$.

The Inductive Step: Assume the formulas for b_i and X_i are correct for i. Due to Step 3b, we can write $b_{i+1} \cdot p = b_i - a \cdot X_i$, and thus

$$\begin{aligned}
b_{i+1} \cdot p &= b_i - a \cdot X_i \\
&= (1 - a \cdot c)^i / p^i - a \cdot c \cdot (1 - a \cdot c)^i / p^i \\
&= (1 - a \cdot c)^i \cdot (1 - a \cdot c) / p^i \\
&= (1 - a \cdot c)^{i+1} / p^i \\
b_{i+1} \cdot p^{i+1} &= (1 - a \cdot c)^{i+1}.
\end{aligned}$$

Once b_{i+1} is available, we can write from Step 3a as $x_{i+1} = c \cdot b_{i+1} \pmod{p}$. This concludes the induction.

TABLE 8
ModInverse Algorithm for Computing $23^{-1} \pmod{2^6}$

i	b_i	$X_i = b_i \pmod 2$	$b_{i+1} = (b_i - a \cdot X_i)/2$
0	$b_0 = 1$	$X_0 = 1 \pmod 2 \rightarrow 1$	$b_1 = (1 - 23 \cdot 1)/2 \rightarrow -11$
1	$b_1 = -11$	$X_1 = -11 \pmod 2 \rightarrow 1$	$b_2 = (-11 - 23 \cdot 1)/2 \rightarrow -17$
2	$b_2 = -17$	$X_2 = -17 \pmod 2 \rightarrow 1$	$b_3 = (-17 - 23 \cdot 1)/2 \rightarrow -20$
3	$b_3 = -20$	$x_3 = -20 \pmod 2 \rightarrow 0$	$b_4 = (-20 - 23 \cdot 0)/2 \rightarrow -10$
4	$b_4 = -10$	$X_4 = -10 \pmod 2 \rightarrow 0$	$b_5 = (-10 - 23 \cdot 0)/2 \rightarrow -5$
5	$b_5 = -5$	$X_5 = -5 \pmod 2 \rightarrow 1$	

TABLE 9
Complexity Analysis of the Modular Inversion Algorithms

Algorithm	Steps	Number of Operations	Operand Sizes	$a^{-1} \bmod p^j$	p	k	Output
DK [5]	k	$1M + 2A$	$1, .., k$	$j = 1, .., k$	2	any	whole
AQ [1] Alg 2	k	$1M + 1A$	$1, .., k$	only $j = k$	2	any	bits
AQ [1] Alg 3	k	$1M + 1A$	k	only $j = k$	2	any	whole
	1	$1D$	k				
AQ [1] Alg 4	s	$3M + 2A$	$2^1, .., 2^s$	$j = 2^0, .., 2^s$	2	2^s	bits
Dumas [3], [4] p^k	s	$2M + 1A$	$2^1, .., 2^s$	$j = 2^0, .., 2^s$	any	2^s	digits
Dumas [3], [4] 2^k	s	$2M + 1A$	$2^1, .., 2^s$	$j = 2^0, .., 2^s$	2	2^s	bits
ModInv p^k	k	$1M$	1	$j = 1, .., k$	any	any	digits
	k	$1M + 1A$	k				
ModInv 2^k	k	$1A$	k	$j = 1, .., k$	2	any	bits

To prove that the algorithm indeed computes $x = a^{-1} \pmod{p^k}$, we note that $a \cdot x$ can be written as

$$\begin{aligned} a \cdot \sum_{i=0}^{k-1} X_i \cdot p^i &= a \cdot \sum_{i=0}^{k-1} c \cdot b_i \cdot p^i \\ &= a \cdot \sum_{i=0}^{k-1} c \cdot (1 - a \cdot c)^i \\ &= a \cdot c \cdot \frac{(1 - a \cdot c)^k - 1}{1 - a \cdot c - 1} \\ &= 1 - (1 - a \cdot c)^k. \end{aligned}$$

Thus, we find $a \cdot x = 1 - (1 - a \cdot c)^k$. We have already determined that $(1 - a \cdot c)$ is a multiple of p, thus, $(1 - a \cdot c)^k$ is a multiple of p^k. This gives $a \cdot x = 1 \pmod{p^k}$.

7 Inversion Mod 2^k

The proposed algorithm significantly simplifies when $p = 2$, and it constitutes an efficient alternative to the existing algorithms. First of all, for $x = a^{-1} \pmod{2^k}$ to exist, $\gcd(a, 2^k)$ must be 1, which implies that a is odd. Given an odd a, the value of $c = a^{-1} \pmod 2$ is trivially found: $c = 1$. The modified algorithm is given below.

function ModInverse$(a, 2^k)$
input: a, k where a is odd and $a < 2^k$
output: $x = a^{-1} \bmod 2^k$
1: $b_0 \leftarrow 1$
2: **for** $i = 0$ **to** $k - 1$
2a: $X_i \leftarrow b_i \pmod 2$
2b: $b_{i+1} \leftarrow (b_i - a \cdot X_i)/2$
3: **return** $x = (X_{k-1} \cdots X_1 X_0)_2$

The mod 2 operation in Step 2a is computed by checking the LSB. Obviously we have $X_i \in \{0, 1\}$, and the inverse x is produced in base 2, that is $x = (X_{k-1} \cdots X_1 X_0)_2$. On the other hand, the division by 2 in Step 2b is performed by right shift. Below, we illustrate the computation of $a = 23$ and $k = 6$, in order to compare to the presented algorithms.

The algorithm produces the binary result $x = (100111)_2 = 39$. This is indeed correct, since $23^{-1} = 39 \pmod{2^6}$. Moreover, our algorithm computes $23^{-1} \pmod{2^j}$ for $k = 1, \ldots, 6$, which are given in base 2 as: $(1)_2 = 1$, $(11)_2 = 3$, $(111)_3 = 7$, $(0111)_3 = 7$, $(00111)_3 = 7$, and $(100111)_3 = 39$, as shown in Table 8.

8 Complexity Analysis

For each algorithm presented in this paper, we analyze the number steps (within the for-loop), the number of arithmetic operations in each step, and the types and sizes of the operands involved, and what the algorithm actually computes. These algorithms differ from another in terms of the number of steps, the types of outputs (for example, the whole number at once or digit-by-digit) and whether or not the consecutive inverses are computed.

A realistic complexity analysis of the algorithms would require that we count of number of bit operations. However, operations requiring $O(1)$ bit operations per step can safely be ignored. These include *check the LSB* and *right or left shift of the operands*. Two important parameters are k (the size of a) and $s = \log_2(k)$. The symbols D, M, and A stand for the processing times for division, multiplication, and addition or subtraction operations. Table 9 summarizes our analysis.

There are four aspects of these modular inversion algorithms, and the interpretation of their complexity results should take them into account.

First of all, these algorithms can be divided into two categories in terms of their asymptotic complexity: linear versus logarithmic, i.e., those requiring k steps versus those requiring $s = \log_2(k)$ steps. There are 3 algorithms requiring logarithmic time which are Arazi and Qi Algorithm 3, and Dumas Algorithms for modulus p^k and 2^k. The remaining 5 algorithms require $O(k)$ steps. It is not automatically concluded that the logarithmic time algorithms are superior. First of all, this will depend on the size of k. As we have discussed in Section 2, the most common use of the modular inversion algorithm is for the implementation of the Montgomery multiplication algorithm. In

regard to this application, we note. The classical Montgomery algorithm [13] requires k to be as large as the size of the RSA modulus n, thus, 512 to 2048. Here, the linear versus logarithmic complexity would be hugely different. However, the classical algorithm is hardly used in practice. The most deployed implementations use the CIOS algorithm [11] which chooses k to be the word size of the processor. If $k = 32$, then $s = \log_2(32) = 5$, and thus, the difference between linear versus logarithmic is not that great. For example, comparing Algorithm 4 to ModInverse algorithm, we see that the former requires $5 \cdot (3M + 2A)$ operations while the latter requires $32 \cdot A$ operations.

Algorithmically, a multiplication operation has at least logarithmic depth in gate delays compared to addition (in both cases of carry save and carry propagate adders) which is 4 or 5 if the operand size is 16 or 32 bits. Taking Pentium as a modern architecture example, we see that integer multiplication (in Pentium 2/3 and 4) Takes 5, 7 clock cycles of latency compared to integer addition which take 1 clock cycle, as seen in Table 5.2 of [7]. On the other hand, the latency is 1 cycle for an integer addition and 3 cycles for an integer multiplication in Intel Core Duo 2. One can find the latencies and throughput in Appendix C, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/TC.2020.2970411. of the "Intel 64 and IA-32 Architectures Optimization Reference Manual", which is located in [8].

We conclude that for $k = 32$, Algorithm 4 (Arazi and Qi) requires $5 \cdot (3M + 2A)$ operations, Dumas Algorithm requires $5 \cdot (2M + 1A)$ operations, while ModInverse requires only $32A$ operations. Assuming $M = 4A$, Algorithm 4 requires $70A$, Dumas Algorithm requires $45A$ and ModInverse requires only $32A$ operations. For $M = 3A$, the number of additions becomes $55A$, $35A$ and $32A$ for the Algorithm 4, Dumas Algorithm, and ModInverse algorithm.

The second point about comparing these 8 algorithms is that they can be divided into 2 categories: algorithms computing the inverse mod p^k (or 2^k) for any value of k versus algorithms that work only for specific values of k, here namely, for those k that is a power of 2. The modular inversion algorithms that work for any k are the Dussé and Kaliski Algorithm, Arazi and Qi Algorithms 2 and 3, and ModInverse Algorithms for p^k and 2^k. The remaining 3 algorithms compute the inverse mod p^k where $k = 2^s$. These algorithms will require an additional reduction to compute the inverse for an arbitrary k; for example, to compute the inverse mod p^{29}, we will first have the compute the inverse for mod p^{2^s} for an s such that $2^s > k$, and then obtain the inverse mod p^{29} by an additional reduction operation.

The third point about comparing these 8 algorithms is that they can be divided into 2 categories: algorithms that compute and output the consecutive inverses (for example, for $j = 1, 2, \ldots, k$ or $j = 1, 2, \ldots, s$) versus algorithms that compute the output for a single k only (however, consecutive inverses can still be obtained by reductions). Only Arazi and Qi Algorithm 2 and 3 compute the inverse for a single modulus; while the Dussé and Kaliski Algorithm and ModInverse Algorithms for p^k and 2^k compute and output the consecutive inverses for mod for $i = 1, 2, \ldots, k$. On the other hand, Arazi and Qi Algorithm 4 and Dumas Algorithms for p^k and 2^k) compute the inverse for consecutive s moduli, specifically for p^{2^j} for $j = 1, 2, \ldots, s$.

The fourth point about comparing these 8 algorithms is that they can be divided into 2 categories: algorithms that work only for $p = 2$ and algorithms that work for any prime p. The first category contains 6 algorithms; while only two algorithms, namely ModInverse and Dumas algorithms work for any p.

Finally, we note that the ModInverse algorithm is the only algorithm that produce the digits (base p or base 2) of the inverse directly, starting from the least significant digits proceeding to the most significant. These digit-by-digit arithmetic algorithms are also named as *on-line arithmetic*. Such algorithms introduce parallelism between sequential operations by overlapping these operations in a digit-pipelined fashion [6].

Furthermore, the ModInverse algorithm for mod 2^k requires the minimal number of arithmetic operation (just a single addition) among all 8 algorithms.

9 Conclusion

We have introduced a new algorithm for computing the inverse $a^{-1} \pmod{p^k}$ given a prime p and $a \in [1, p-1]$. The algorithm is based on the exact solution of linear equations using p-adic expansions, due to Dixon [2]. The new algorithm starts with the initial value $c = a^{-1} \pmod{p}$ and iteratively computes the inverse $x = a^{-1} \pmod{p^k}$. The binary version of the proposed algorithm (that is, when $p = 2$) is significantly more efficient than the existing algorithms for computing $a^{-1} \pmod{2^k}$ when k is small, which is the case for the CIOS Montgomery multiplication algorithm. Moreover, the proposed algorithm computes all inverses mod p^i or 2^i for $i = 1, 2, \ldots, k$ and work for an arbitrary k. We have also described and analyzed 6 existing algorithms, and provided an extensive comparison and interpretation o the proposed algorithm.

Our proposed algorithm stands out as being the only one that works for any p, any k, and digit-by-digit. Moreover it requires the minimal number of arithmetic operations (just a single addition) per step.

Acknowledgments

The author thanks to Francois Grieu for comments in [6], Watson Ladd for comments on Dixon's algorithm being actually due to Hensel, Markku-Juhani Olavi Saarinen for comments on Newton-Raphson algorithm, and Michael Scott for reminding the references [1], [3] and commenting on the first version of this article. The author also thanks reviewers for pointing out that the article [3] was also published in IEEE Transactions on Computers [4] and for greatly improving and correcting the article with their rigorous reviews.

References

[1] O. Arazi and H. Qi, "On calculating multiplicative inverses modulo 2^m," *IEEE Trans. Comput.*, vol. 57, no. 10, pp. 1435–1438, Oct. 2008.
[2] J. D. Dixon,"Exact solution of linear equations using p-adic expansions," *Numerische Mathematik*, vol. 40, no. 1, pp. 137–141, 1982.
[3] J. Dumas, "On Newton-Raphson iteration for multiplicative inverses modulo prime powers," *arXiv:1209.6626v3*, 2012. [Online]. Available: https://arxiv.org/abs/1209.6626v3
[4] J. Dumas, "On Newton-Raphson iteration for multiplicative inverses modulo prime powers," *IEEE Trans. Comput.*, vol. 63, no. 8, pp. 2106–2109, Aug. 2014.
[5] S. R. Dussé and B. S. Kaliski Jr, "A cryptographic library for the Motorola DSP56000," in *Proc. Workshop Theory Appl. Cryptographic Techn.*, 1990, pp. 230–244.
[6] F. Grieu, "Answer to 'How to determine the multiplicative inverse modulo 64 (or other power of two)?'," StackExchange Cryptography, 2017. [Online]. Available: https://crypto.stackexchange.com/questions/47493
[7] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*, New York, NY, USA: Springer, 2004.
[8] Intel, "Intel 64 and IA-32 architectures software developer manuals," Jan. 18, 2018. [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm
[9] B. S. Kaliski Jr., "The Montgomery inverse and its applications," *IEEE Trans. Comput.*, vol. 44, no. 8, pp. 1064–1065, Aug. 1995.
[10] Ç. K. Koç, "High-speed RSA implementation," RSA Lab., Hebron, CT, Tech. Rep. TR 201, Nov. 1994.
[11] Ç. K. Koç, T. Acar, and B. S. Kaliski Jr., "Analyzing and comparing Montgomery multiplication algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26–33, Jun. 1996.
[12] E. V. Krishnamurthy and V. K. Murty, "Fast iterative division of p-adic numbers," *IEEE Trans. Comput.*, vol. 32, no. 4, pp. 396–398, Apr. 1983.
[13] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, no. 170, pp. 519–521, Apr. 1985.
[14] E. Savaş and Ç. K. Koç, "The montgomery modular inverse - revisited," *IEEE Trans. Comput.*, vol. 49, no. 7, pp. 763–766, Jul. 2000.
[15] E. Savaş and Ç. K. Koç, "Montgomery inversion," *J. Cryptographic Eng.*, vol. 8, pp. 201–210, 2017.

A Fast Filtering Mechanism to Improve Efficiency of Large-Scale Video Analytics

Chen Zhang, Qiang Cao, *Senior Member, IEEE*, Hong Jiang, Wenhui Zhang, *Student Member, IEEE*, Jingjun Li, and Jie Yao, *Member, IEEE*

Abstract—Surveillance cameras are ubiquitous around us. Emerging full-feature object-detection models can analyze surveillance videos with high accuracy but consume much computation. Directly applying these models for practical scenarios with large-scale cameras is prohibitively expensive. This, however, is wasteful and unnecessary considering that user-defined anomalies occur rarely among these videos. Therefore, we propose FFS-VA, a multi-stage Fast Filtering Mechanism for Video Analytics, to make video analytics much cost-effective. FFS-VA filters out the frames without the user-defined events by two stream-specialized filters and a cheap full-function model, to reduce the number of frames reaching the full-feature model. FFS-VA presents a global feedback-queue approach to balance the processing speeds of different filters in intra-stream and inter-stream processes. FFS-VA designs a dynamic batch technique to achieve a trade-off between throughput and latency. FFS-VA can also efficiently scale to multiple GPUs. We evaluate FFS-VA against the state-of-the-art YOLOv3 under the same hardware and video workloads. The experimental results show that under a 12.88 percent target-object occurrence rate on two GPUs, FFS-VA can support up to 30 concurrent video streams (15× more than YOLOv3) in the online case, and obtain 10× speedup when offline analyzing a stream, with an accuracy loss of less than 2 percent.

Index Terms—Filters, frames, video analytics

1 Introduction

An increasing number of surveillance cameras with low cost and high quality have been deployed in key public areas all over a city (e.g., street corners, shopping malls, and office buildings), to monitor potential accidents as well as record critical clues. In traditional practice, the video surveillance collects live streams to an operational center for further manual observations, which is labor-intensive, error-prone and cost expensive. Automatic video analysis based on object detection has been introduced recently to mitigate human intervention while significantly improving the performance and accuracy. The cases of automatic analysis for surveillance videos can be categorized into two main types: (1) real-time analysis to detect anomalies; and (2) post-facto analysis to look for a certain event retroactively.

Early automatic analysis methods [1] employ support vector machine, but its accuracy and function are limited [2]. Benefitting from the recent development in complex model structures based on neural network (NN), the accuracy and speed of object detection have been significantly improved. In 2014, R-CNN [3] first achieves 53.7 percent mean average precision (mAP) on PASCAL VOC 2010 [4]. Afterwards, SSD [5] (74.3 percent mAP, 59 frame per second (FPS)), R-FCN [6] (83.6 percent mAP, 5 FPS), YOLOv2 [7] (76.8 percent mAP, 67 FPS), and YOLOv3 [8] (30 FPS) have been proposed to continuously improve both accuracy and detection speed, making real-time online analysis and high-speed offline analysis for video streams feasible in practical scenarios.

However, these existing full-feature object-detection models are extremely computationally hungry. A powerful GTX Titan X GPU only supports one video stream at 30 FPS with YOLOv3 model. Considering that a typical video surveillance generally deploys hundreds of cameras, these full-feature models are ill-equipped to perform real-time analysis for large-scale video streams directly, due to their unacceptably high hardware cost.

Fortunately, in large-scale video analytics, a typical anomalous event occurs rarely and when it does occur it appears in a tiny fraction of all frames. For example, even if a serious traffic jam takes place on the main route of a big city, the average blocked time in a day is just less than 5 percent [9]. Therefore, passing all frames over these accurate but weighty models actually wastes considerable computational capability, which is totally unnecessary. The key idea is to fast filter out most frames that are not related to user-defined events while leaving the remaining frames to be accurately analyzed by the final full-feature NN model.

To efficiently employ these highly accurate object detection models on limited computing devices to achieve large-scale high-resolution video analytics, we propose FFS-VA, a fast filtering mechanism, to dramatically reduce the number

- *C. Zhang, Q. Cao, W. Zhang, and J. Li are with the Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System of Ministry of Education, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {hust_zchen, caoqiang, jingjunli}@hust.edu.cn, singularity_x@outlook.com.*
- *H. Jiang is with the University of Texas at Arlington, Arlington, TX 76019. E-mail: hong.jiang@uta.edu.*
- *J. Yao is with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: jackyao@hust.edu.cn.*

Manuscript received 2 Aug. 2019; revised 1 Dec. 2019; accepted 26 Jan. 2020. Date of publication 30 Jan. 2020; date of current version 8 May 2020. (Corresponding author: Qiang Cao.) Recommended for acceptance by Y. Han. Digital Object Identifier no. 10.1109/TC.2020.2970413

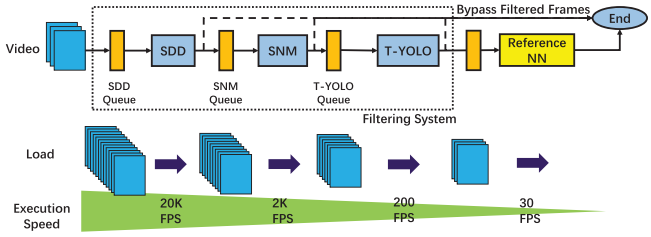


Fig. 1. FFS-VA adds three filters before the Reference NN to filter out frames that are not related to the user-defined events.

of frames actually reaching the full-feature reference model by filtering out vast amounts of frames that do not satisfy the conditions of the user-defined events (e.g., frames without target objects or with a number less than a predefined threshold of target objects) to support both fast offline analysis and large-scale online analysis.

FFS-VA is a pipelined multi-stage filtering system and each stage is equipped with a specific model to filter out frames with a certain feature. Considering that most surveillance cameras are of a fixed viewpoint, we design and train a stream-specialized difference detector (SDD) to remove frames only containing background and a stream-specialized network model (SNM) to identify target-object frames. The remaining frames are further screened by a cheap full-function Tiny-YOLO-Voc model (T-YOLO), that is shared by multiple streams, to filter out frames whose target objects are fewer than a predefined threshold. Finally, the surviving frames are fed into an ultimate full-feature reference model (Reference NN) for high-accuracy analysis. FFS-VA is designed to run on the mainstream heterogeneous servers with several GPUs and CPUs. Fig. 1 shows the filtering structure of FFS-VA over one video stream.

There are four key challenges in FFS-VA that need to be addressed. (1) For a mainstream server with two GPUs (at least), the workloads should be evenly distributed on CPUs and GPUs of the heterogeneous server to achieve a high performance. (2) Theoretically, the number of frames processing in the several stages within a stream is gradually decreasing due to filtering. Accordingly, the filter at a later stage is also slower than one at an earlier stage in processing speed. Unfortunately, due to the unpredictable video contents the number of frames entering each stage varies significantly over time. How to dynamically balance the video loads among filters within a stream and among multiple streams is a key problem. (3) For the network model executed on the GPU (e.g., SNM), in order to process a frame, the corresponding network model, image data, and the predicted result must be loaded between the CPU memory and the GPU memory. To reduce the overhead of frequent data exchange, a large and static batch size is intuitively better for obtaining high computational efficiency but at the cost of lengthened processing latency. In a multi-stage stream processing system with multiple CNNs, the impact of batch size is considerable. So it is necessary to dynamically trade off at runtime between latency and throughput based on the video contents. (4) If there are multiple GPUs available in the system, all of them can be used to improve the performance of offline analysis and online analysis. To efficiently parallel the GPUs without affecting the order of frames in a stream, it also demands to appropriately assign all tasks to these GPUs at runtime.

FFS-VA uses the next several techniques to solve the above challenges. (1) If just two GPUs are available on a server, all SDDs are executed on the CPU, both SNMs and T-YOLO are executed on a single GPU. The Reference NN runs on another GPU alone. To achieve a high computational efficiency, adding a queue between any two consecutive stages unlocks all stages from synchronous lock steps, enabling them to be executed concurrently. (2) FFS-VA builds a global feedback approach to orchestrate the processing speeds of all stages based on their respective queue controls. (3) FFS-VA adopts a dynamic batch technique to dynamically determine batch size according to current video contents in a short period of time. (4) If multiple GPUs are configured on a server, a dynamic GPU scheduling method is utilized to allocate stages to GPUs automatically at runtime. And then FFS-VA presents four GPU parallel schemes to make multiple GPUs run efficiently and ensure the frame order during processing a video stream.

Leveraging these system-level optimizations, FFS-VA is able to efficiently perform both online and offline video analytics on large-scale video streams. Experimental results show that, compared with the state-of-the-art YOLOv3 model with the same hardware environment consisting of two CPUs and two GPUs, FFS-VA supports up to 15× more concurrent video streams in online video analysis, and obtains 10× speedup in offline video analysis, with a negligible accuracy loss of less than 2 percent. Besides, as the number of GPUs increases, FFS-VA can support real-time detection for more video streams. In addition, we also analyze the performance differences between YOLOv2 model and YOLOv3 model in terms of throughput and accuracy, and show the impact of the improvement of Reference NN model on system performance.

In summary, the key contributions of our work are:

1) We propose a pipelined multi-stage fast filtering mechanism for large-scale video analytics (FFS-VA) in both online and offline scenarios.
2) FFS-VA introduces a global feedback-queue approach to control the processing speeds of all filters in both intra-stream and inter-stream processes. FFS-VA designs a dynamic batch technique with a video-content-based batch-size adjustment to automatically trade off between latency and throughput.
3) FFS-VA adopts a scheduling method to dynamically allocate stages on GPUs. To efficiently perform a stage on multiple GPUs, FFS-VA further designs four GPU parallel schemes under different scenarios.
4) We implement a FFS-VA prototype system to compare it with the state-of-the-art YOLOv3, which indicates that FFS-VA improves the overall throughput by up to 15× under the same hardware environment.

The rest of this paper is organized as follows. Section 2 introduces the background of video analysis and key observations that motivate this research. The design and implementation of FFS-VA are presented in Sections 3 and 4 respectively. Section 5 evaluates the performance, sensitivity of key design parameters, batch technique, scalability and limitations of FFS-VA. Prior studies most relevant to FFS-VA are reviewed in Section 6. Section 7 concludes the paper.

2 Background and Motivation

2.1 Convolutional Neural Network Models

Optical Flow [10], Support Vector Machine (SVM)[1], and Convolutional Neural Network (CNN) [11] can all be used for the recognition of specific targets. The insight of Optical Flow is matching pixels between images using temporal and gradient information. SVM builds linear decision hyperplanes in the feature space to perform classification. The main idea of CNN is to learn the features of targets by dealing with numerous images and updating the weights of artificial neurons. Since CNN has great image recognition capabilities and can recognize the characteristics of the target objects easily and accurately, it has been widely developed in recent years. In what follows, we briefly introduce the typical CNN models for video analysis.

CNN consists of a series of connected layers, including convolutional layer (CONV), fully connected layer (FC), pooling layer (POOL), and so on. The CONV layer is responsible for extracting local features from high-resolution feature maps. The POOL layer is in charge of organizing the local features from the CONV layer and abstracting them into a low-resolution feature map. The FC layer is used to output the actual prediction based on the outcome of preceding layers. By combining several such layers in a certain order and configuring all layers with appropriate weights, a CNN model is formed.

Once the CNN model is determined, it can be used for further inferencing on a video stream by passing all frames in the video stream one by one or in batches. For each video frame, the CNN gives a predicted probability of whether target objects occur in this frame.

2.2 Object Detection

Object detection technology has been widely used to automatically analyze video contents, such as detecting accidents [12] and traffic congestion [13], searching a certain object [14], understanding the flow of vehicles and pedestrians to provide users with the most reasonable traffic planning [15], etc., which greatly reduce the cost of manpower.

In terms of the use of spatial-temporal information on video frames, Nicolas Ballas *et al.* [16] and Lai Jiang *et al.* [17] correct the detection results by using relevant timing and context information, which greatly improve the accuracy of object detection. For static image detection, the accuracy of R-CNN (53.7 percent mAP) [3], fast R-CNN (65.7 percent mAP) [18], faster R-CNN (70.4 percent mAP) [19], and R-FCN (83.6 percent mAP) [6] has been continually improved, but the execution speed of these full-feature object detection techniques remains relatively low and inadequate for real-time detection. Recently, advanced methods such as YOLOv2 (67 FPS) [7], SSD (59 FPS) [5], and YOLOv3 (30 FPS) [8] have been developed to achieve real-time detection, but they are also computationally expensive as a powerful GTX Titan X GPU merely supports the analysis of no more than two concurrent video streams in real time. So in a limited hardware environment, using these advanced models to perform real-time detection for large-scale video streams is a huge challenge.

In fact, there are two methods that are usually used to speed up the inference process of CNN models: compression and specialization. The commonly used compression methods include removing some CONV layers and FC layers [20], reducing the feature size of models [7], and matrix pruning [21]. The general compression methods usually achieve a huge increase in execution speed at the expense of accuracy unless the compression is performed efficiently and effectively. On the other hand, specialization refers to the CNN models are trained by using the dataset in specific conditions or scenes. So the generality requirements for these models are sacrificed, and if they are used in other conditions or scenes the accuracy may drop dramatically. Due to only one specific context is considered, these specialized models can be both accurate and fast. Certainly, both of these two methods are used in this paper.

2.3 Motivation

As a fundamental requirement for large-scale surveillance video analysis, users expect to know whether their concerned anomalous events occur in a timely manner. The applicable scenarios for video analysis can be roughly classified into two categories: offline and online. In the offline case, all stored videos need to be processed as fast as possible to capture interesting scenes. In the online case, an analytic system is expected to support more live streams while timely determining any anomaly and providing warning for the upcoming risks.

Indeed, the frames without target objects generally are not worth further analyzing and need to be filtered out. The computationally expensive full-feature models should only process the frames with the target object(s). Therefore, we define the target object ratio (TOR) as

$$TOR = num_{target-object-frames}/num_{all-frames}, \tag{1}$$

where $num_{all-frames}$ is the total number of all frames in a video stream during a given period of time, and $num_{target-object-frames}$ is the number of frames containing the target objects. TOR can help characterize the frequency of target-object frames that appear in a video slice. TOR is primarily determined by video contents, objectively reflecting the actual utilization of a video analytic system. For large-scale surveillance videos, TOR is generally low in long-period videos, which means that the anomalous events are actually infrequent for all monitored videos. According to the analysis results of numerous webcams [22], the target-object occurrence rate in a day is only 8 percent.

Therefore, under the actual low TOR, passing all frames over a full-feature model such as YOLOv3 is a huge waste of computational resources. This insight motivates us to design an effective and efficient fast filtering system to identify the frames with anomalous events from massive video frames. And then only those identified frames are worth performing over the subsequent full-feature model to further extract interesting information. Therefore, we design three types of preceding filters and a pipelined multi-filter architecture to achieve the aforementioned goal.

Additionally, the fast filtering system should evenly distribute all tasks on CPUs and GPUs, exploiting the maximal potential of the underlying hardware to boost the overall performance under the promised latency and negligible accuracy loss. Besides, If multiple GPU devices are available

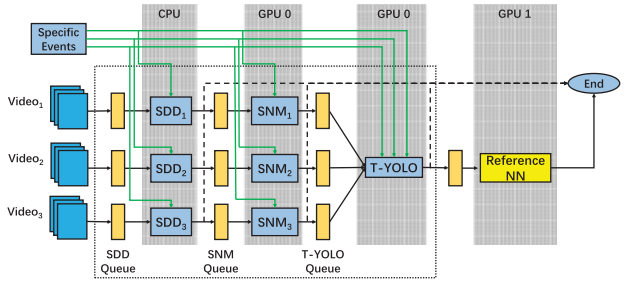


Fig. 2. Architecture of FFS-VA on two GPUs.

in the server, they can be used to improve the performance of online analysis and offline analysis. We will elaborate on these issues in detail next.

3 Design

3.1 Overview of FFS-VA

Fig. 2 illustrates the architectural overview of FFS-VA on a server with two GPU devices, which consists of three types of filters: (1) specialized difference detectors (SDDs), (2) specialized network models (SNMs), and (3) a globally shared object detection model Tiny-YOLO-Voc (T-YOLO).

3.1.1 Main Functions

First, target events, such as the occurrences of cars, persons, etc., as well as their counts, need to be predefined by users. For each given video stream, both SDD and SNM are specialized for it and its predefined event. The two filters are then followed by a T-YOLO model that is globally shared by all streams. As a result, FFS-VA supports various target events are detected in multiple concurrent video streams.

In the FFS-VA, SDD is dedicated to filtering out background frames. SNM is used to identify the target-object frames and filter out the non-target-object frames. Afterwards, T-YOLO is employed to filter out the frames containing fewer than a threshold number of target objects. Finally, the surviving frames are input to the full-feature reference model for final high-accuracy identification and analysis. The details of filters are illustrated in Section 3.3. For clarity and meaningful evaluation, we choose the state-of-the-art full-feature model, YOLOv3, as the Reference NN model for FFS-VA in this paper.

3.1.2 Pipeline Design

The three preceding filters and final Reference NN model form a four-stage pipelined architecture. Each filter is connected by its corresponding input and output queues for reading and forwarding frames. For each video stream, a prefetching thread is specifically created to capture video frames from cameras or disks. In fact, all frames of a video stream should pass its dedicated SDD, the very first filter along the pipeline. And then, the number of frames processed by each subsequent filter decreases gradually in proportion to the filtering rate of the preceding filter. For better performance, the processing speed of each filter in the pipeline also exhibits gradual decrease accordingly. Note that the input frame rate of each stage is varied with the fluctuation of video contents over time. In order to dynamically balance loads across varied filters, we propose a global feedback-queue approach, as detailed in Section 4.3.1, to coordinate the processing speeds of various filters. In addition, a dynamic batch technique also be introduced to help trade off between throughput and latency automatically at runtime, which is illustrated in Section 4.3.2.

Considering the varied computational complexity of filters and Reference NN, in order to fully exploit the potential of the hardware, SDDs are executed on the CPUs, and SNMs and T-YOLO are executed on a single GPU. The Reference NN model uses another GPU alone. Besides, the runtime scheduling in the system (e.g., feedback queue and dynamic batch) is also controlled by the CPU. In FFS-VA, each filter is associated with an independent thread. Through the parallel and pipelined structure, FFS-VA achieves a high analyzing throughput. Although only two GPUs are used in this part, FFS-VA can conveniently scale the processing capacity to a server with more GPUs or a server cluster, which will be discussed in Section 4.4.

3.2 Formal Description

We provide a theoretical description based on the system with three filters and a Reference NN, which can help understand the key criteria of the filtering mechanism. This description can also be extended to the other system with more filters and various models.

3.2.1 Throughput

In Fig. 2, when one stream runs on the system and V frames are processed, the passing ratios of these frames in the three filters and the Reference NN are r_1, r_2, r_3, and r_4 respectively (their corresponding filtering ratios are $1-r_1$, $1-r_2$, $1-r_3$, and $1-r_4$). In this case, the number of frames outputted by the Reference NN can be calculated as

$$V_{output} = V \cdot r_1 \cdot r_2 \cdot r_3 \cdot r_4. \tag{2}$$

The overall filtering ratio of the system is defined as

$$R_F = (V - V_{output})/V = 1 - r_1 \cdot r_2 \cdot r_3 \cdot r_4. \tag{3}$$

Suppose the detection speeds of three filters and Reference NN are s_1, s_2, s_3, and s_4 respectively. Considering the four stages can be executed concurrently, the total execution time is presented as

$$T = max\left(\frac{V}{s_1}, \frac{V \cdot r_1}{s_2}, \frac{V \cdot r_1 \cdot r_2}{s_3}, \frac{V \cdot r_1 \cdot r_2 \cdot r_3}{s_4}\right). \tag{4}$$

Similarly, the Equations (5) and (6) show the overall filtering ratio and execution time of FFS-VA for the analysis of N video streams. Thus the actual throughput of the system can be considered as the ratio of the total number of input frames to the execution time (Equation (7))

$$R_F = 1 - \frac{\sum_{i=1}^{N} V^i_{output}}{\sum_{i=1}^{N} V^i} = 1 - \frac{\sum_{i=1}^{N} V^i \cdot r^i_1 \cdot r^i_2 \cdot r^i_3 \cdot r^i_4}{\sum_{i=1}^{N} V^i} \tag{5}$$

$$T = max\left(\frac{\sum_{i=1}^{N} V^i}{s_1}, \frac{\sum_{i=1}^{N} V^i \cdot r_1^i}{s_2}, \frac{\sum_{i=1}^{N} V^i \cdot r_1^i \cdot r_2^i}{s_3}, \frac{\sum_{i=1}^{N} V^i \cdot r_1^i \cdot r_2^i \cdot r_3^i}{s_4}\right) \tag{6}$$

$$Xput = \frac{\sum_{i=1}^{N} V^i}{T}. \tag{7}$$

Based on the above definitions, the filtering ratio of FFS-VA is mainly related to the contents of video streams and can be significantly affected by the actual passing ratio at each stage in practice. In addition, except for the filtering ratio, the detection speeds of stages are another factors that affects the actual throughput. Given the video loads show a downward trend due to filtering, the detection speeds of filters and Reference NN should also be gradually decreasing (i.e., $s_1 > s_2 > s_3 > s_4$).

3.2.2 Latency

The video frames outputted by the Reference NN satisfy all the conditions of the user-defined events, and their processing latency is also a key metric. We define the processing latency of a frame as the time interval between it incoming and leaving the system. In Equation (8), it is divided into three parts: $L_{service}$, L_{wait}, and L_{sync}

$$L = L_{service} + L_{wait} + L_{sync}. \tag{8}$$

$L_{service}$ means the sum of processing time on all stages, which is mainly related to hardware configurations, the complexity of models, and batch size. L_{wait} is the queue time of frames, which is affected by resource competition, queue depth, video contents, and so on. And L_{sync} is defined as the sum of synchronous time between multiple GPUs at each stage. L_{sync} is negligible unless one of the stages is assigned to more than one GPU.

In short, $L_{service}$ can be shortened with more powerful computing devices or smaller batch size. Besides, reasonable system configurations and runtime scheduling need to be utilized to reduce L_{wait}. And L_{sync} can be optimized by designing and using appropriate GPU parallel schemes.

3.2.3 Accuracy

In fact, it is possible for a frame to be recognized by the Reference NN but filtered out by its preceding filters, i.e., a false negative. Of course, if a frame unrelated to the user-defined event passes all stages, this is called as a false positive. To quantitatively analyze the accuracy of the system, we define the error rate as

$$ErrorRate = \frac{\sum_{i=1}^{N} FN_i + FP_i}{\sum_{i=1}^{N} V_i}, \tag{9}$$

where FN_i and FP_i refer to the number of false negatives and false positives respectively among V_i frames in a stream i. Then the accuracy is described as

$$Acc = 1 - ErrorRate. \tag{10}$$

Since the last stage of the system is the Reference NN, which serves as the accuracy baseline of the system, and all output frames have to go through this stage, so there are no false positives in output frames in practice.

(a) Background (b) Frame with a car (c) Subtracted frame

Fig. 3. An example of SDD. The subtracted frame highlights the car that entering the scene.

3.2.4 Scalability

If there are idle GPUs available in the server, they can be used to improve the performance of the system. More GPU devices can mitigate resource competition in intra-stream and inter-stream processes as well as can increase the detection speeds of filters and Reference NN (i.e., s_1, s_2, s_3, and s_4). So the system throughput can also be significantly improved. But multiple GPUs also introduce some extra scheduling overheads, such as stage allocations on GPUs, transferring data between CPU memory and GPU memory, and collecting the analysis results synchronously from the GPUs. These overheads lead to that the increase in the throughput is not proportional to the increase in the number of GPUs.

In addition, the scalable approach cannot reduce the latency L since the processing time of all stages $L_{service}$ is not shortened. On the contrary, in order to guarantee the frame order in a stream, a synchronous or reordering process is required when collecting analysis results from multiple GPUs, which introduces a little synchronous time L_{sync}.

More importantly, under different video loads, the optimal stage allocations are varied. For example, when an anomaly occurs in FFS-VA, idle GPUs are more suitable for speeding up the processing of the Reference NN because of its expensive computation. But for the scenes without anomalies, T-YOLO stage is more likely to become the bottleneck of the system and needs to be assigned more GPUs. The differences of stage allocations on GPUs inevitably affect the detection speed and the system throughput even with the same number of GPUs.

3.3 Detailed Design of Filters

3.3.1 Specialized Difference Detector (SDD)

As the first filter of the system, SDD is responsible for foreground segmentation [23] and determines whether a unlabeled frame is a background frame. Commonly used foreground segmentation methods include Averaging, Single Gaussian, Kalman filter, Gaussian Mixture Model, and so on. In FFS-VA, all SDDs use Averaging method because of its fast execution speed (Fig. 3).

In fact, SDD calculates the distance between the reference image (i.e., background image) and the unlabeled frame to determine whether these two images are identical. For simplicity, the reference image is usually computed as the average of dozens of background frames. The distance between two images can be characterized by Mean Square Error (MSE), Normalized Root Mean Square Error (NRMSE), or Sum of Absolute Differences (SAD). Take MSE as an

example, if MSE is larger than the threshold δ_{diff}, an obvious content change is construed to have occurred in the current unlabeled frame. Otherwise, the frame is considered as a background frame. Note that most surveillance cameras are deployed in a fixed viewpoint. Hence, the background frames can be safely discarded.

Naturally, the threshold δ_{diff} has a critical affect on the filtering ratio and accuracy. A low δ_{diff} may result in poor filtering ratio of SDD, while a high δ_{diff} can lead to a high error rate. Furthermore, the threshold δ_{diff} may vary greatly in different scenes. For example, a video with a mostly empty sidewalk (a static background) might have a small δ_{diff}. However, a background with changing light color and intensity in the same scene (a dynamic background) results in a larger δ_{diff}. In addition, weather, light intensity, etc. can all contribute to the value of MSE [23], so the filtering ratio of SDD varies greatly at different scenes.

SDD processes 100*100-pixel images at 100K FPS (0.9M multiplication operations and 1.8M addition operations per 30 frames). In the FFS-VA, the detection speed of SDD is 600× faster than the reference model YOLOv3 (7.1G multiplication operations and 34.8G addition operations per 30 frames), as demonstrated in Section 5.

3.3.2 Specialized Network Model (SNM)

Another key filter used in FFS-VA is SNM. SNM utilizes a specialized CNN to detect whether a video frame contains target objects. Generic models can classify and recognize thousands of object classes no matter what the scenes are, but the generality of these methods leads to huge computational overhead and long execution time. On the contrary, SNM can only identify a class of predefined target objects in the specific video stream, and thus trading off reducing the generality for boosting its speed (70× real-time). In addition, for a fixed-angle camera, the position and the moving trail of the target objects in the scene are relatively fixed. In this case, using SNM for rapid image recognition can also ensure the accuracy to be over 95 percent [24].

In fact, SNM is a three-layer CNN (CONV, CONV, and FC). The design and training process is shown in Section 2.1. When a video frame is inferred by SNM, SNM first outputs a predicted probability c of the target object appearing in the frame. If c is below the threshold c_{low}, no target object is considered to be in the frame. If c is higher than c_{high}, the frame is a target-object image. Otherwise, it is unsure whether the frame is target-object or non-target-object.

In our design, threshold t_{pre} is utilized (between c_{low} and c_{high}) at runtime to help SNM distinguish target-object frames and non-target-object frames, which is shown in Section 4.2. SNM puts the target-object frames ($c \geq t_{pre}$) into the T-YOLO queue, and the non-target-object frames ($c < t_{pre}$) are filtered out.

Experiments show that SNM processes 50*50-pixel images at 5K FPS using about 200 KB GPU memory (159.7M multiplication operations and 422.8M addition operations per 30 frames). It is 60× faster than the reference model YOLOv3 on the real hardware platform.

3.3.3 Tiny-YOLO-Voc (T-YOLO)

The third filter of FFS-VA is a cheap and compressed object detection network, T-YOLO [25], which is used to filter out the frames whose the number of target objects is less than a certain level. Other object detection networks, such as YOLO, SSD, and R-FCN, due to the slow execution speed caused by more layers and more classes, are not used in our filtering system preferentially. As a compressed model, T-YOLO just consists of 9 CONV layers and 6 POOL layers, and is trained by the VOC dataset with 20 classes. Benefitting from fewer classes and smaller model size, T-YOLO can perform at up to 220 FPS for 416*416 pixel images with just 1.2 GB GPU memory usage (1.3G multiplication operations and 5.1G addition operations per 30 frames).

(a) Car detection (b) Person detection

Fig. 4. Examples of object detections.

T-YOLO can recognize multiple target objects in one image (Fig. 4). First, T-YOLO divides the input image into 13*13 grid cells automatically. Each grid cell predicts 5 bounding boxes and confidence scores for these boxes. If the confidence score exceeds the threshold (e.g., 0.2), one target object is considered to appear in the image. By combining the individual grid cell detections, the total number of target objects appearing in the video frame can be obtained.

As the filter is shared among all video streams, T-YOLO needs to traverse each T-YOLO queue of all streams one by one and extracts at most num_{t-yolo} video frames from the queue for detection, skipping the stream if its queue is empty.

Here we employ a generic model to identify tens of classes for two main reasons: 1) For different video streams, sharing the same model can reduce the switch overhead of loading different models from CPU memory to GPU memory (e.g., 1.2 GB for T-YOLO). 2) We not only support to detect different target objects for different video streams, but also support the detection of multiple target objects within a video stream in the later stages, to facilitate the understanding of the scene.

3.4 About False Negatives

In video surveillance, users are particularly concerned about missing scenes rather than missing frames. Given a live stream with 30 FPS, target objects could continuously appear in a series of frames. Even if just a few legible frames are identified, the scene is in fact correctly identified. This means that the rest of the frames pertaining to the scene can be considered redundant or duplicate and filtered out without affecting the detection of the target scene.

Take SNM stage as an example, false negatives in this stage can be categorized into two cases. On one case where a merely partial appearance of target objects (Fig. 5a), i.e., incomplete target object (e.g., head of vehicle) appears, can be identified by the Reference NN but missed by the filters (e.g., SNM and T-YOLO). Nevertheless, its subsequent frames in the same scene can contain entire target objects that SNM and T-YOLO are able to correctly detect (Fig. 5b). In this case, the scene is considered correctly detected. On the

(a) Partial appearance of the object (b) Entire object

Fig. 5. A false negative case but without missing scene.

other hand, if dozens of continuous frames containing the complete target objects are filtered out incorrectly, then the scene is considered lost. The latter case should be avoided as much as possible. Similarly, in T-YOLO stage, not identifying all target objects can also cause false negatives.

Considering false negatives do not necessarily result in scene loss, so the FN in Equation (9) refers to the number of false negatives that actually cause the scene loss in practice.

In fact, if we slightly relax the filtering condition of a filter (e.g., set the filtering threshold slightly below the target threshold required by the models or anomalous events) and forward a little more frames to the follow-up filters (the latter filter can detect the results of the previous again), the false negatives could be reduced. Therefore, the cascaded structure and relaxed filtering conditions can prevent excessive filtering errors. And we elaborate on these issues in Section 5.3.

4 Implementation

4.1 Training SDD and SNM

We apply the model training method mentioned in NoScope [24] to train specialized models (i.e., SDDs and SNMs) for each video stream. For a video stream, we first extract a representative fraction of the video and label its video frames by using YOLOv3. These labeled data are divided into two subsets as a training dataset and a test dataset. The former is used to train a SDD and a SNM for each video stream and the latter is used to select a set of suitable thresholds for δ_{diff}, c_{low}, and c_{high} to meet the requirement for accuracy and execution speed by comparing the predicted results against the real truths.

Different from NoScope, which uses specialized models to query for target-object frames in a single off-line video and only extracts one frame from every 30 frames, FFS-VA designs the specialized filters to filter out the non-target-object video frames from large-scale video streams in both online and offline modes. For each online video stream, FFS-VA is designed to process at a rate of at least 30 FPS. In addition, the main metric of NoScope is throughput, but FFS-VA also pays attention to latency.

Before each filter is executed, the raw frames need to be resized to meet the filter's default feature size. The resizing times of SDD, SNM, and T-YOLO are about 40, 150, and 400 us respectively. Accordingly, the raw frames always exist in the system unless these frames are filtered by any one filter.

4.2 Filter Control

4.2.1 FilterDegree

Although the two thresholds in SNM, c_{low} and c_{high}, basically determine the prediction of a video frame, the values between the two thresholds may also be acceptable. Because different video streams have different c_{low} and c_{high} values, the choice for t_{pre} can vary. So we compute the t_{pre} value as follows:

$$t_{pre} = (c_{high} - c_{low}) \times FilterDegree + c_{low}. \tag{11}$$

FilterDegree is a parameter set by users, which reflects the aggressiveness of filtering in SNM stage. When $FilterDegree = 1$ ($t_{pre} = c_{high}$), the output frames have a high credibility, but it increases the probability of false negatives. When $FilterDegree = 0$ ($t_{pre} = c_{low}$), more frames pass to the T-YOLO filter in this case, which bring a heavier burden to the T-YOLO filter. Therefore, *FilterDegree* can directly affect filtering ratio of SNM and accuracy of FFS-VA. In our system, the cases $t_{pre} < c_{low}$ and $t_{pre} > c_{high}$ are not considered. This is because values within this range would result in a dramatically increase of error rate (e.g., 2.4 percent) or a remarkable decrease of throughput (e.g., 60 percent).

4.2.2 NumberofObjects

NumberofObjects refers to the filtering threshold in T-YOLO stage, which controls the filtering conditions of the T-YOLO model. Normally, *NumberofObjects* is required to be equal to the intensity of the target objects in the predefined events (i.e., target threshold). But *NumberofObjects* may be less than the target threshold in the case of relaxing filtering condition.

In the running process of the T-YOLO filter, if the number of target objects appearing in a frame is less than *NumberofObjects*, the target object is considered to have a low intensity, which is outside the scope of the user's interest. Otherwise, it is likely that some unexpected events have occurred.

In fact, T-YOLO cannot only monitor the intensity of the target object, achieving purposeful filtering based on user needs, but also catch and correct the false positives of SNM. For example, if a non-target-object frame is passed by the SNM accidentally, T-YOLO can still filter out the frame by counting the number of target objects, reducing the false positives of FFS-VA.

4.3 System Optimization

In this section we introduce two methods to optimize throughput and latency of FFS-VA.

4.3.1 Feedback Queue

Note that the processing speed of one type of filter is often different from that of another. SDD processes about 10× faster than SNM and 100× faster than T-YOLO. Because of the fluctuation of video contents over time, the number of video frames processed by each filter also varies over time. When frames arrive in bursts, their processing filter threads are able to compete for hardware resources with each other seriously and even block.

Sine resource competition between different filters exists in both intra-stream and inter-stream processes, it is necessary to balance the execution speeds of these filters at runtime. Therefore, we propose a global feedback-queue approach. First, FFS-VA controls the detecting speed of a filter at an earlier stage in the pipeline by detecting the queue

depth of the filter at a later stage. For example, when the T-YOLO queue depth exceeds a threshold, the SNM thread automatically slows down or even gets blocked, and stops pushing frames to the T-YOLO queue until the T-YOLO queue depth is blow the threshold. As long as the system can keep at least 30 FPS for each video stream, the stream is being analyzed in real time. In addition, for some video streams, the number of target-object frames varies greatly over time. To balance loads among video streams, T-YOLO filter extracts a different number of frames from different queues in one cycle. For the video streams with a long T-YOLO queue depth, FFS-VA extracts up to num_{t-yolo} frames from the queue per cycle. Otherwise, a fewer number of frames are processed by the T-YOLO filter in this cycle.

The setting of the queue depth thresholds is important. Too small an threshold may reduce the detection speeds of filters (e.g., s_2), even the system throughput $Xput$, due to the filters at later stages cannot get enough frames (e.g., a batch) from their queues at a time. On the contrary, too large an threshold will increase feasible overloads and queue time L_{wait} since too many frames are cached in the queue. Unless otherwise stated, we initially and empirically determine 2, 10, 10 and 10 as the queue depth thresholds of SDD queues, SNM queues, T-YOLO queues, and the Reference NN queue respectively.

4.3.2 Dynamic Batch

Since each video stream has its own SNM, it causes the GPU to load these models frequently when processing frames coming from different video streams. The data exchange overhead significantly lowers the overall computational efficiency. Intuitively, feeding a batch of frames to the GPU and processing them at a time can greatly reduce the amount of model and image data loading. For example, when the batch size is 30, the loading frequency is reduced by 30×.

Feedback-queue indeed achieves a higher throughput by static batch. However, a triggering strategy based on a fixed number of input frames per batch can lead to unnecessary delay. First, if the number of frames in the queue is less than the batch size, in order to compose a batch of data, waiting for the remaining frames can introduce some delay (i.e., L_{wait}). In addition, even if enough frames exist in the queue, the batch processing itself can also introduce additional latency (i.e., $L_{service}$), especially for a large batch size. All of these will result in a long latency for the system.

In order to solve these problems, we propose a dynamic batch technique based on the feedback-queue approach. If there are enough video frames in the SNM queue, to get a high throughput, SNM pops out a batch of (*BatchSize*, e.g., 30) images from the queue for SNM prediction. Otherwise, the frames are popped from the SNM queue until the queue is empty to guarantee a low system latency. Note that although batch processing is used in SNM, when we pop (or push) frames from (or to) a queue, the chronological order of video frames needs to be strictly guaranteed. We elaborate the reasons in Section 4.4. In fact, dynamic batch can also be applicable to other CNN models for a better trade-off between throughput and latency. Experimental results show that compared with using the feedback-queue approach alone, the dynamic batch technique reduces the average latency by 27 percent while lowering the throughput by only 6 percent, ensuring better real-time performance.

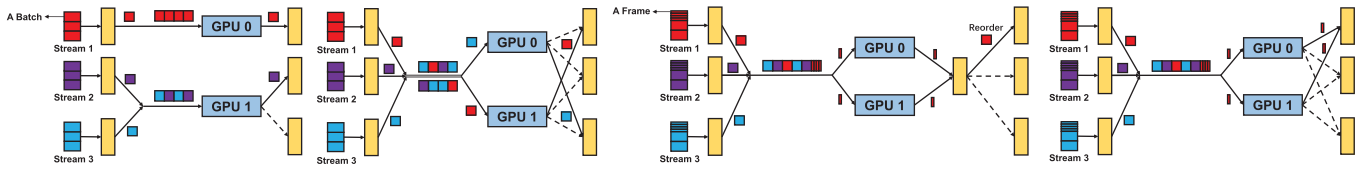
4.4 Scalability

We previously described the design of FFS-VA based on the typical system with two GPUs. Indeed, FFS-VA also can scale to the server with more GPUs, to provide larger processing capacity for both offline and online analysis. It introduces a new challenge how to reasonably and dynamically allocate streams and their relevant stages across multiple GPUs with workload variation, to ensure the high efficiency of hardware.

As aforementioned, the models in the latter two stages are highly consumptive in computation (e.g., T-YOLO at 200 FPS and Reference NN at 30 FPS per GPU). When $TORs$ of the streams are in a low level, only a few of frames can arrive at T-YOLO and then Reference NN. In this case, two working GPUs are enough. Nevertheless, when target events occur in one or more streams immediately, the loads can overload these two GPUs. The extra idle GPUs are gradually activated to share the heavier-load stages.

Specifically, FFS-VA adopts a utilization-based scheduling method. It measures the GPU utilizations of the T-YOLO stage and Reference NN stage every three seconds. When the average utilization of a GPU (or GPUs) performing T-YOLO stage in five minutes exceeds a threshold (e.g., 75 percent), FFS-VA activates an idle GPU to join this stage. When the average utilization is lower than 20 percent, one of the working GPUs will be deactivated. It is similar for the stage of Reference NN. To ensure normal operations of the system, FFS-VA guarantees that at least one GPU runs on each of these two stages. We design two interfaces *ActivateGPUDevice()* and *InactivateGPUDevice()* to be responsible for the allocation and release of the two models on GPUs at runtime. The specific operations are implemented by traversing each layer of the NN model as well as calling CUDA interfaces *cudaMalloc()*, *cudaMemcpy*, and *cudaFree()*.

When multiple GPUs share the same stage, FFS-VA needs to further determine how to assign frames of streams to the GPUs. Notice that the chronological order of frames in a stream should be strictly kept after these frames passing a stage, thereby guaranteeing the functional correctness of FFS-VA. To achieve this goal, we present four parallel schemes according to the scheduling granularity (stream, batch in a stream, or frame in a batch) and the parallel mode of GPUs (asynchronous or synchronous) as SMa, BMa, FMa, and FMs (Fig. 6). *SMa* means that each stream (stream granularity) is fixedly allocated to a GPU. In this case, each GPU traverses its own involving streams and extracts a batch of frames for processing. *BMa* means that each GPU can serve all streams but merely alternatively and exclusively extracts a batch of frames (batch granularity) in a stream at a time. *FMa* means that all GPUs work for a batch of frames (frame granularity) in a stream simultaneously in a frame-level asynchronous manner. The specific number of processing the frames depends on the computing power of the GPUs. However, the processed frames need to be reordered in a buffer to maintain their chronological order. *FMs* means that all GPUs work for a batch of frames (frame granularity) in a stream simultaneously but in a frame-level synchronous way. These frames are orderly allocated to all GPUs in turn, and each GPU gets one video frame at a time.



(a) SMa (b) BMa (c) FMa (d) FMs

Fig. 6. Four parallel schemes are presented according to the scheduling granularity and the parallel mode.

These four schemes have their own advantageous scenarios. FMa and FMs are advantageous in the cases with small number of streams since all GPUs can serve one stream simultaneously. SMa has a better latency due to a batch of frames is serially processed without inter-frame synchronization and reordering process. BMa does well in heterogeneous environments. FFS-VA can well support these four parallel schemes. The experimental performance analysis for these four schemes will be further explained in Section 5.5.

5 Evaluation

5.1 Experimental Setup

We use two real-world public videos, *Jackson* and *Coral*, as our evaluation workloads. *Jackson* describes the scenes of various vehicles (e.g., car, bus, truck, etc.) traveling at a crossroad. *Coral* describes the scenes of people watching colorful fish in an aquarium. Their relevant information is summarized in Table 1. Each video contains about one million video frames in the time span of one day. We extract typical non-overlapping video clips from each video file to simulate multiple video streams. Even for the same video, the number of target objects varies greatly with time. Hence, we can analyze the impact of scenes with different TOR values on the filtering performance. For fairness, the feature sizes of YOLOv3 used in both FFS-VA and the baseline are similar to be set as 416*416.

Hardware Platform. We perform our experiments on a platform with four NVIDIA Geforce GTX 1080 GPUs, two GTX Titan X GPUs, dual Intel Xeon E5-2683 v3 CPU, and 128 GB DRAM. The multi-core CPUs provide good support for multi-threaded evaluation workload.

First, we explore the system performance of FFS-VA in online and offline situation. Second, we analyze the sensitivity of filtering thresholds in FFS-VA to the overall system accuracy and filtering ratio. And then, we study the impact of batch techniques on the throughput and latency. Finally, the performance of FFS-VA deployed into a multi-GPU environment is demonstrated. For the sake of simplicity, we select 5,000 consecutive frames from each video stream to perform the inference tasks.

TABLE 1
Information of Evaluation Videos

Video Name	Resolution	Object	FPS	TOR
Coral	1280*720	Person	30 FPS	50%
Jackson	600*400	Car	30 FPS	8%

5.2 System Performance

Fig. 7 shows the average throughput and latency as a function of the number of video streams with 0.129 TOR on two GTX 1080 GPUs.

Offline Analysis. With a single video stream, which represents the offline analysis performance, the maximum throughput that FFS-VA can support is 592 FPS, which is 10× and 6× that supported by YOLOv3 and YOLOv2 respectively. Compared with YOLOv3 the total execution time of FFS-VA is reduced by 90.9 percent. In addition, for a 55 GB video file, the entire system uses less than 8 GB CPU memory, which implies greatly increased support capacity for long-time high-resolution video files.

Online Analysis. Experiments show that our system can support up to 31 video streams for real-time detection, which is 15 × more than what YOLOv3 can support. Although the throughput of YOLOv2 is about 2× than YOLOv3 in the offline situation, both of them can only support real-time analysis of no more than 3 concurrent video streams. Besides, dynamic batch technique has a 27 percent lower latency than using feedback-queue approach alone in most cases, but at the cost of 6 percent reduction in throughput caused by the small batch size. Moreover, it is necessary to note that although FFS-VA has a latency of several seconds, these delays are insignificant and tolerable in some applications, such as intelligent video surveillance [26].

For video streams with 1.000 TOR as an extreme case, Fig. 8 shows that FFS-VA can only support 5 video streams in real time. This is because SDDs and SNMs filter out fewer video frames and most of the frames are still fed to the T-YOLO for filtering, limiting the amount of increase in the overall throughput. In addition, the offline detection throughput has also dropped noticeably in our experimental platform, and the overall execution time is only 67 percent shorter than the YOLOv3. This is because, on the same hardware platform, only T-YOLO performs efficient filtering on one GPU and another GPU is used to perform YOLOv3, while the baseline YOLOv3 can perform on both two GPUs.

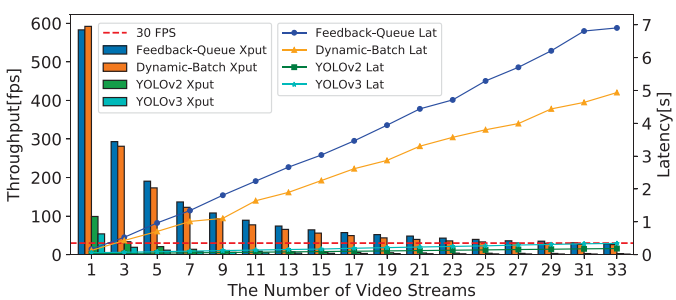


Fig. 7. The average throughput and latency as a function of the number of video streams with a TOR value of 0.129.

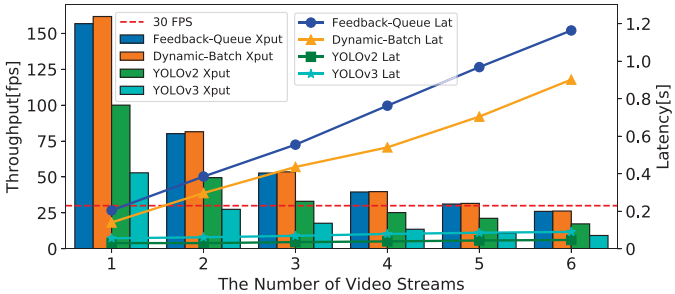


Fig. 8. The average throughput and latency as a function of the number of video streams with a TOR value of 1.000.

Fig. 9 presents the ratio of frames executed in each filter with different TOR during the day. SDD filters out few frames due to frequent movement and video contents change in the daytime. The filtering ratio of SNM is largely related to the TOR. And T-YOLO can all work well in any case. It is worth noting that different time periods, weather, occlusion, video contents, illumination, etc., may all affect the performance of the three filters, and we only show a small part of them here.

5.3 Sensitivity of Key Thresholds

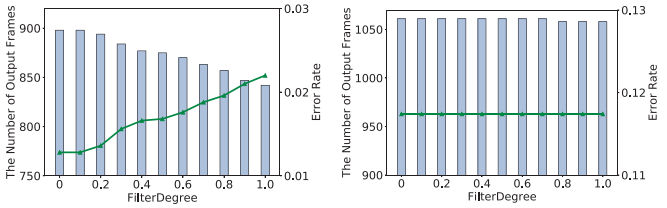
Next, we use offline video streams to examine how two key thresholds, *FilterDegree* and *NumberofObjects*, impact the filtering ratio and accuracy of FFS-VA. To verify the accuracy, all the filtered frames by FFS-VA are completely detected by the reference model YOLOv3.

5.3.1 FilterDegree

Fig. 10a illustrates the effect of the *FilterDegree* threshold on the number of output frames and the filtering error rate for the car detection (TOR=0.186). As the threshold increases, more frames whose prediction probability c is between c_{low} and c_{high} are filtered out, so the error rate also rises accordingly. Fig. 10b shows the results of person detection (TOR=1.000). The adjustment of the *FilterDegree* value has little effect on the filtering ratio and error rate in this case. This is because during the entire observed time period, the aquarium is in the tourist peak and all frames contain many persons, which prevents SNM from filtering out any video frame.

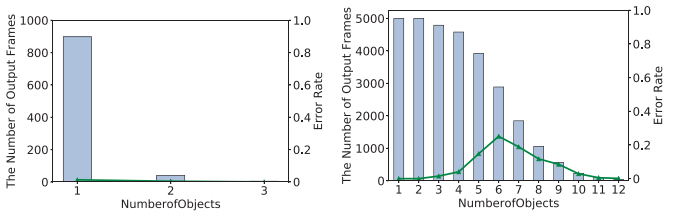
5.3.2 NumberofObjects

In Fig. 11a, for car detection, as *NumberofObjects* increases, the number of output frames decreases significantly (about 80 percent). This is because, in this video, the size of a car is relatively large in a scene that can only contain no more than three target objects. Fig. 11b illustrates the case of person detection. The number of output frames gradually decreases with the increase of *NumberofObjects*.



(a) Car detection (TOR=0.186) (b) Person detection (TOR=1.000)

Fig. 10. The throughput and error rate as a function of *FilterDegree*.



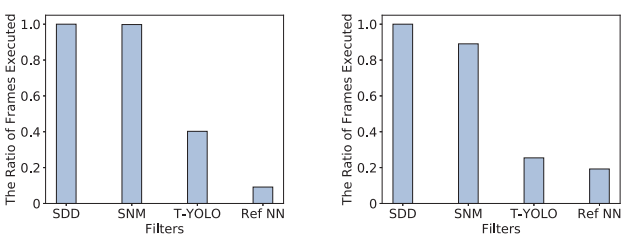
(a) Car detection ($TOR = 0.186$) (b) Person detection ($TOR = 1.000$)

Fig. 11. Number of output frames and error rate as a function of *NumberofObjects*.

5.3.3 Accuracy Analysis

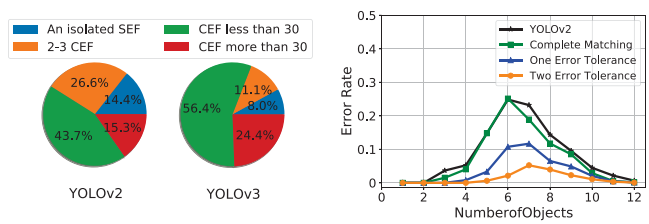
To better understand the error rate, we first analyzed the false-negative frames for car detection with a TOR value of 0.186. Fig. 12a illustrates the statistics of these error frames. The cases of one isolated single-error frame (SEF) and 2-3 continuously-error frames (CEF) do not affect the correct identification of the scene. In addition, a series of consecutive error frames whose size is less than a certain level (e.g., 30 frames) is usually caused by a distinguish criterion for partial-appearance of target object between T-YOLO and YOLOv3. In this case where there are many consecutive error frames, it is because a single partially appeared vehicle is waiting for traffic lights. By analyzing these images, we observe that about 50 frames out of a total of 5,000 frames are those with actual scene losses ($< 2\%$). In addition, by comparing the statistics of error frames based on YOLOv2 and YOLOv3, we find that the proportion of one isolated SEF and 2-3 CEF in YOLOv3 is smaller, and YOLOv3 has a stronger and more stable recognition ability than YOLOv2.

In addition, as shown in Fig. 11b, the error rate is relatively high. This is due to for the detection of small and dense targets, such as persons in the crowd, T-YOLO generally



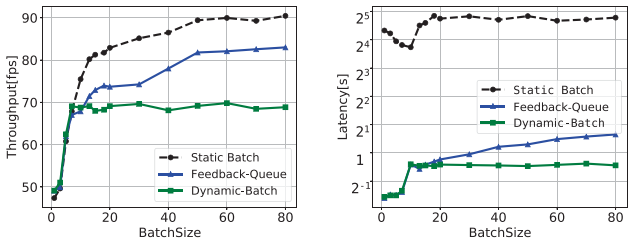
(a) Car detection (TOR=0.435) (b) Person detection (TOR=0.259)

Fig. 9. The ratio of frames executed in each filter. The processing speeds of the four stages at runtime are about 20K FPS, 2K FPS, 200 FPS, and 30 FPS, respectively.



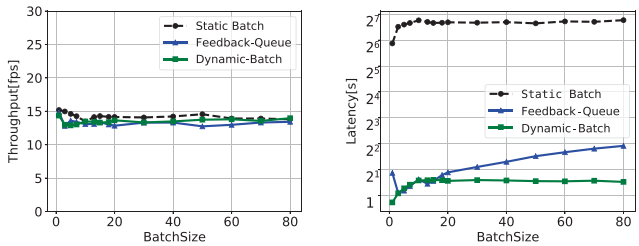
(a) YOLOv2 vs. YOLOv3 (b) Relaxing Filtering Conditions

Fig. 12. (a) Statistics of error frames in 5000 consecutive video frames. (b) The error rate as a function of NumberofObjects by relaxing filtering conditions.



(a) Throughput (b) Latency

Fig. 13. Throughput and latency under different batch techniques with TOR 0.203.



(a) Throughput (b) Latency

Fig. 14. Throughput and latency under different batch techniques with TOR 0.980.

identifies fewer target objects than YOLOv3, resulting in a high error rate. In this case, Fig. 12b shows that if one or two object misjudgment can be tolerated by relaxing the filtering threshold, the error rate will be greatly reduced (57.2 and 91.5 percent respectively). Even if relaxing filtering conditions may have a little impact on the filtering ratio of the system (about 12.6 and 22.2 percent respectively), it is worthwhile to ensure the overall accuracy of the system ($<$ 2%). Therefore, it exists a trade-off between accuracy and filtering ratio. Moreover, in the process of accuracy analysis for filtered frames, the error rate of using YOLOv2 as Reference NN is slightly higher than that of YOLOv3 in terms of complete matching. This is because YOLOv2 occasionally misses a few target objects during the detection process so that more frames are filtered incorrectly.

In short, the experiments show that thanks to the relaxed filtering conditions and the cascaded structure, the actual cases of missing scenes are less than 2 percent in our system, which is arguably negligible.

5.4 Batch Technique

We analyze the impacts of the static batch without feedback-queue, feedback-queue, and dynamic batch techniques on average throughput and latency over 10 video streams.

5.4.1 Throughput of FFS-VA

Fig. 13a shows the average throughput of the static batch, feedback-queue, and the dynamic batch techniques respectively with 0.203 TOR. When *BatchSize* is low, these three methods can process a full batch of video frames at a time from the SNM queue. While SNM processes the current batch of video frames, SDD quickly pushes enough video frames to the SNM queue to form the next batch of video frames, thus having little impact on the throughput. When *BatchSize* is high, for the static batch technique, the throughput can still continue to increase with sufficient data provided by SNM queues. For the feedback-queue approach, the wait for a batch of frames increases the execution time, resulting in a slight decrease in throughput (about 8 percent). For the dynamic batch technique, the smaller batch size further leads to a decrease in computation efficiency.

Fig. 14a shows the experimental results with 0.980 TOR. In this case, most of the frames are eventually executed by T-YOLO model no matter what the *BatchSize* value is. Therefore, *BatchSize* has little effect on the throughput.

5.4.2 Average Latency

Fig. 13a shows that when *BatchSize* is small, the difference in average latency between the feedback-queue and the dynamic batch techniques is very small. As *BatchSize* increases, more video frames need to wait a period of time in the feedback-queue because of the fixed batch size. For the dynamic batch technique, since the batch size can be adjusted automatically according to video contents, the average latency is basically unchanged. Due to the same queue management method, for video streams with 0.980 TOR, the average latency has a similar trend in Fig. 14b.

In summary, for videos with a low TOR value, the feedback-queue approach has a greater throughput, and the dynamic batch technique has a smaller average latency. For videos with a high TOR value, there is not much difference in throughput between the feedback-queue and dynamic batch techniques, but the dynamic batch technique has a lower average latency and should be considered first.

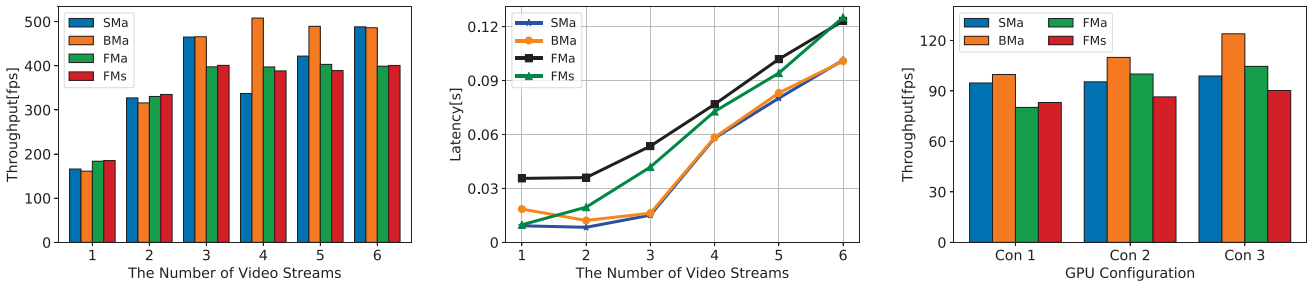
5.5 Scalability

We take T-YOLO stage as an example to evaluate the throughput and latency of four GPU parallel schemes on a server with three homogenous GPU devices. Then, for heterogeneous GPU devices, the performance of these four parallel schemes on the Reference NN stage is demonstrated. Finally, we show the scale performance of system as a function of TOR in online and offline cases.

Throughput. In Fig. 15a, since all GPUs can serve the same video stream at a time, FMa and FMs work well when there is a few video streams in FFS-VA. As the number of video streams increases, BMa achieves the best performance. This is because, in BMa, multiple GPUs only need to alternatively process a batch of frames of streams, without synchronizing and reordering process. For SMa, due to uneven distribution of streams on the GPUs, only when the number of video streams is an integer multiple of the number of GPUs, FFS-VA can achieve the load balance and the peak performance.

Latency. In Fig. 15b, due to without inter-frame synchronization (i.e., smaller L_{sync}), SMa and BMa exhibit a short latency. Switching video streams over multiple GPUs makes the latency of BMa slightly higher than the SMa. Besides, the extra reordering process also significantly increases the latency of FMa.

Heterogeneous GPU Environment. Fig. 15c shows the throughput of the four parallel schemes on the Reference NN stage over three GPU configurations: (1) three GTX 1080 GPUs (Con1); (2) two GTX 1080 GPUs and one GTX Titan X GPU (Con2); and (3) one GTX 1080 GPU and two GTX Titan X GPUs (Con3). Since GTX Titan X GPU is more powerful than GTX 1080 GPU, the throughput of Con3 is higher than Con2 and Con1 in all cases. However, due to



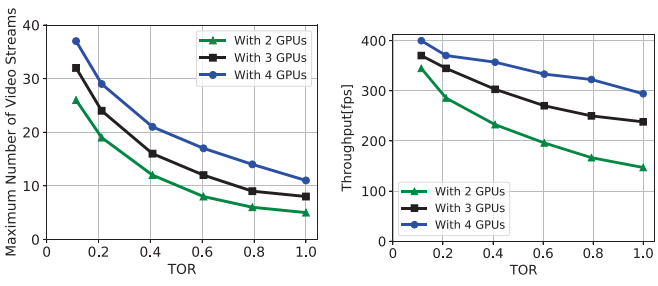
(a) Throughput (b) Latency (c) Heterogeneity

Fig. 15. The performance of four GPU parallel schemes.

frequent inter-frame synchronization and inter-stream synchronization, the slowest GPU (or GPUs) limits the overall throughput of the system. Therefore, improving GPU configurations have little impact on the performance of FMs and SMa. In contrast, the throughput of BMa is improved considerably as the GPU capacity improves.

Scale Performance. Based on the previous analysis, TOR has a pivotal impact on the maximum number of video streams supported by FFS-VA. Therefore, we discuss the impact of the number of GPUs on the overall performance under different $TORs$ with BMa. In fact, we assign two, three, and four GTX 1080 GPUs to FFS-VA respectively in the three experiments. Fig. 16a shows that the maximum number of video streams supported by FFS-VA increases as TOR decreases in the online case. Moreover, for a system configured with four GPUs, the overall performance has been improved by up to 122 and 45 percent than the system configured with two GPUs and three GPUs respectively. In addition, we also measure the impact of the number of GPUs on the latency. The results show that the latency remains essentially the same regardless of the number of GPUs. This is because, on the one hand, additional GPUs increase the detection speeds of filters but do not reduce processing time (i.e., $L_{service}$). On the other hand, the synchronous time L_{sync} of BMa is too small compared with the processing time $L_{service}$ and the queue time L_{wait}.

For offline scenarios, in Fig. 16b, as TOR increases, the throughput also shows a downward trend. When TOR is small in a video slice, few target-object frames exist, and the throughput is mainly determined by the detection speed of SNM. As a result, adding GPUs to T-YOLO stage and Reference NN stage has little effect on the throughput. With TOR increasing, growing frames arrive at the latter two stages. Compared with the server with two GPUs, deploying three and four GPUs can improve the throughput by 62 and 100 percent respectively. However, even with utilization-based scheduling method to allocate stages dynamically at runtime, using extra GPUs does not bring the same amount of performance gains. Due to the existence of scheduling overheads, the throughput improvement from adding the second GPU is about 20 percent lower than adding the first GPU. With more additional GPUs, new performance bottlenecks are gradually emerging.



(a) Online (b) Offline

Fig. 16. The scale performance as a function of TOR in online and offline cases.

5.6 Limitations and Remedies

While the FFS-VA filtering system is shown to be highly effective in real-time object detection for large-scale video streams, there remain some limitations.

Target Object Rate Sensitivity. In practice, a sudden $TORs$ increase in video streams can lead to poor filtering ratio, even if the probability of multiple videos having their $TORs$ increase simultaneously is extremely low. If necessary, we can temporarily store these video frames in a storage system, to be processed later. For some latency-sensitive scenes, it is necessary to use more GPUs or a server cluster to provision for peak-load periods.

Error Rate. The reason for the cases of relative high error rate is the performance difference between T-YOLO and the reference model YOLOv3. Deep compression [27] (e.g., pruning, sparsity constraint) can transform a larger but more accurate NN model to a tiny model without compromising the accuracy of the prediction, resulting in a 3× throughput improvement [28]. Therefore, we can replace T-YOLO with a high-accuracy mode that was deeply compressed to obtain a low error rate.

Scene Switch. We train SDD and SNM models for each fixed-angle camera and specific target object, so the changes of video scene may affect the detection accuracy. If the scene change in the video is periodic (e.g., alternating between day and night), the training data just needs to include representative frames under all conditions. However, when the scene changes dramatically or the function and position of the camera have changed, the previous specialized models will no longer work. If there are no saved models in the past that can match the current environment, a new network model needs to be trained according to the new scene, which takes about one hour.

Single Target Object. In this paper, we assume that there is only one user-interested target object for each video stream. If multiple target objects exist in a video stream, the structure of the specialized network model only needs to be changed to support the identification of all the target objects in the video.

6 Related Work

Prior studies relevant to FFS-VA can be classified into six main categories, model cascades, object detection, video monitoring, model specialization, stream processing, and video analysis.

Model Cascades. Cascade is defined as a sequence of classifiers to improve inference speed. Paul Viola *et al.* [29] proposed the first cascade, the Viola-Jones detector, which cascades traditional image processing features. The sub-windows which are not rejected by an initial classifier are processed by a sequence of classifiers. If any classifier rejects the sub-window, no further processing is performed. Recent work has concentrated on learning cascades. [30] achieves an optimal trade-off between accuracy and speed by learning a complexity aware cascade. [31] configured a CNN cascade for real-world face detection to accurately differentiate faces from the backgrounds. [32] proposed a cascaded regression approach for facial point detection to make more accurate predictions. In our filtering system, we use a cascade to filter out video frames that we do not care about, not specific to the features. Besides, FFS-VA focuses on improving the processing throughput of the system rather than the effectiveness of a single model.

Object Detection. SPPnet [33] and Fast R-CNN [18] have achieved a very high accuracy for the image object detection, by using region proposal object detection methods. OverFeat [34] and YOLO [35] have achieved a high detection speed by skipping the proposal step altogether, and predicting bounding boxes and confidences for multiple categories directly. Our goal is to increase the throughput of the overall system using these models and some other models for filtering in practice.

Video Monitoring. Video monitoring involves many tasks, including vehicle tracking [36], object detection [37] and so on. Each task has been tailored for a specific system (e.g., vehicle counting [38], license plate detection [27], cars or pedestrians tracking [39]). And the main target objects are car and pedestrian. Our filtering system focuses on video analysis, and several objects (e.g., dogs, cats) can be detected in FFS-VA to facilitate the understanding of the scene if needed. Besides, more event-related details (e.g., detail behavior analysis) can be fine-grainedly detected by the back-end network model instead of just trajectory analysis.

Model Specialization. Compared with generic models, specialized models can guarantee a high throughput and accuracy by sacrificing generality in the same hardware environment. Both NoScope [24] and Foucs [40] use this technique in cheap CNN to help query for target-object frames in a off-line video faster. In contrast, FFS-VA pays more attention to the analysis of more real-time video streams through model specialization. In addition, FFS-VA also analyzes the impact of video content fluctuation (e.g., *TOR*) on real-time system performance instead of just getting the target-object frames.

Stream Processing. The general stream processing challenges, such as distributed execution, fault tolerance, and real-time performance, have received widespread attention in various stream processing system [41]. In contrast, FFS-VA focuses on supporting more video streams on the same hardware device for high-accuracy fine-grained analysis by utilizing CNN models, which is orthogonal to these advanced works.

Video Analysis. In terms of information retrieval on videos, there are several techniques widely used to analyze video contents, such as semantic video search [42], spatio-temporal information-based video retrieval [43], and shot boundary detection [44]. The main goal of FFS-VA is to build a filtering system for video analysis. Benefit from low coupling structure and strict order guarantee for video frames, these advanced video analysis methods can be integrated into one stage of our system conveniently if necessary.

7 Conclusion

In real life, there are a lot of camera resources to be explored. And NN makes it easy to extract semantic information from these videos. However, its computational efficiency is low. Besides, the huge number of video frames in the large-scale video streams also poses a great challenge to neural network. In response, we propose a filtering system that equips a SDD model, a SNM model, and a global T-YOLO model for each video stream, filtering out the video frames that the users are not concerned about in the video stream, and reducing the number of video frames that need to be detected by the full-feature model. The experimental results show that our filtering system provides 5-15× scalability improvement over the state-of-the-art YOLOv3. In addition, the reference model in this paper is an object detection network, but FFS-VA can also be applied to other fields, such as face recognition, by configuring other appropriate reference models.

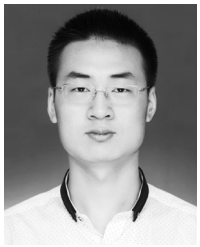
Acknowledgments

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions. This work was supported in part by National key research and development program of China (Grant No. 2018YFA0701804), Creative Research Group Project of NSFC No.61821003, NSFC No. 61872156, Fundamental Research Funds for the Central Universities No. 2018KFYXKJC037, the US National Science Foundation under Grant No.CCF-1704504 and No.CCF-1629625, and Alibaba Group through Alibaba Innovative Research (AIR) Program.

References

[1] Y. Lin *et al.*, "Large-scale image classification: Fast feature extraction and SVM training," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2011, pp. 1689–1696.
[2] Z. Wang, Z. Wang, H. Zhang, and X. Guo, "A novel fire detection approach based on CNN-SVM using tensorflow," in *Proc. Int. Conf. Intell. Comput.*, 2017, pp. 682–693.
[3] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2014, pp. 580–587.
[4] M. Everingham, L. J. V. Gool, C. K. I. Williams, J. M. Winn, and A. Zisserman,"The pascal visual object classes (VOC) challenge," *Int. J. Comput. Vis.*, vol. 88, no. 2, pp. 303–338, 2010.
[5] W. Liu *et al.*, "SSD: Single shot multibox detector," in *Proc. Eur. Conf. Comput. Vis.*, 2015, pp. 21–37.
[6] J. Dai, Y. Li, K. He, and J. Sun, "R-FCN: Object detection via region-based fully convolutional networks," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2016, pp. 379–387.
[7] J. Redmon and A. Farhadi, "YOLO9000: Better, faster, stronger," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 6517–6525.

[8] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," *CoRR*, vol. abs/1804.02767, 2018. [Online]. Available: http://arxiv.org/abs/1804.02767
[9] H. Wang, K. Rudy, J. Li, and D. Ni, "Calculation of traffic flow breakdown probability to optimize link throughput," *Appl. Math. Modelling*, vol. 34, no. 11, pp. 3376–3389, 2010.
[10] A. OttlikandH.-H. Nagel, "Initialization of model-based vehicle tracking in video sequences of inner-city intersections," *Int. J. Comput. Vis.*, vol. 80, no. 2, pp. 211–225, 2008.
[11] W. Yin, K. Kann, M. Yu, and H. Schütze, "Comparative study of CNN and RNN for natural language processing," *CoRR*, vol. abs/1702.01923, 2017. [Online]. Available: http://arxiv.org/abs/1702.01923
[12] L. Huang, Z. Li, andB. Wang, "Detection of abnormal traffic video images based on high-dimensional fuzzy geometry," *Autom. Control Comput. Sci.*, vol. 51, no. 3, pp. 149–158, 2017.
[13] L. Li, L. Chen, X. Huang, and J. Huang, "A traffic congestion estimation approach from video using time-spatial imagery," in *Proc. 1st Int. Conf. Intell. Netw. Intell. Syst.*, 2008, pp. 465–469.
[14] H. Wei, C. Yang, and Q. Yu, *Efficient Graph-Based Search for Object Detection*. Amsterdam, The Netherlands: Elsevier, 2017.
[15] G. Mo and S. Zhang, "Vehicles detection in traffic flow," in *Proc. Int. Conf. Netw. Comput.*, 2010, pp. 751–754.
[16] N. Ballas, L. Yao, C. Pal, and A. C. Courville, "Delving deeper into convolutional networks for learning video representations," in *Proc. 4th Int. Conf. Learn. Representations*, Y. Bengio and Y. LeCun, Eds. 2016. [Online]. Available: http://arxiv.org/abs/1511.06432
[17] L. Jiang, M. Xu, and Z. Wang, "Predicting video saliency with object-to-motion CNN and two-layer convolutional LSTM," *CoRR*, vol. abs/1709.06316, 2017. [Online]. Available: http://arxiv.org/abs/1709.06316
[18] R. B. Girshick, "Fast R-CNN," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2015, pp. 1440–1448.
[19] S. Ren, K. He, R. B. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2015, pp. 91–99.
[20] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. 3rd Int. Conf. Learn. Representations*, Y. Bengio and Y. LeCun, Eds. 2015. [Online]. Available: http://arxiv.org/abs/1409.1556
[21] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural network," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2015, pp. 1135–1143.
[22] Jackson, "Town-square-southwest," 2017. [Online]. Available: https://www.seejh.com/webcams/jacksonhole/jackson/town-square-southwest
[23] N. E. Buch, S. A. Velastin, and J. Orwell, "A review of computer vision techniques for the analysis of urban traffic," *IEEE Trans. Intell. Transp. Syst.*, vol. 12, no. 3, pp. 920–939, Sep. 2011.
[24] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia, "NoScope: Optimizing neural network queries over video at scale," *Proc. VLDB Endowment*, vol. 10, no. 11, pp. 1586–1597, 2017.
[25] A. Zainab, "Real-time object detection," 2017. [Online]. Available: https://blog.mindorks.com/detection-on-android-using-tensorflow-a3f6fe423349
[26] A. C. Nazare Jr, and W. R. Schwartz, "A scalable and flexible framework for smart video surveillance," *Comput. Vis. Image Understanding*, vol. 144, no. C, pp. 258–275, 2016.
[27] C. Anagnostopoulos, I. Anagnostopoulos, I. D. Psoroulas, V. Loumos, and E. Kayafas, "License plate recognition from still images and video sequences: A survey," *IEEE Trans. Intell. Transp. Syst.*, vol. 9, no. 3, pp. 377–391, Sep. 2008.
[28] S. Han *et al.*, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. Annu. Int. Symp. Comput. Archit.*, 2016, pp. 243–254.
[29] P. A. Viola and M. J. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2001, pp. 511–518.
[30] Z. Cai, M. J. Saberian, and N. Vasconcelos, "Learning complexity-aware cascades for deep pedestrian detection," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2015, pp. 3361–3369.
[31] H. Li, Z. Lin, X. Shen, J. Brandt, and G. Hua, "A convolutional neural network cascade for face detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 5325–5334.
[32] Y. Sun, X. Wang, and X. Tang, "Deep convolutional network cascade for facial point detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2013, pp. 3476–3483.
[33] K. He, X. Zhang, S. Ren, and J. Sun, "Spatial pyramid pooling in deep convolutional networks for visual recognition," in *Proc. Eur. Conf. Comput. Vis.*, 2014, pp. 346–361.
[34] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "OverFeat: Integrated recognition, localization and detection using convolutional networks," in *Proc. 2nd Int. Conf. Learn. Representations*, Y. Bengio and Y. LeCun, Eds. 2014. [Online]. Available: http://arxiv.org/abs/1312.6229
[35] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 779–788.
[36] B. Tian, Q. Yao, Y. Gu, K. Wang, and Y. Li, "Video processing techniques for traffic flow monitoring: A survey," in *Proc. Int. IEEE Conf. Intell. Transp. Syst.*, 2011, pp. 1103–1108.
[37] J. B. KimandH. J. Kim, "Efficient region-based motion segmentation for a video monitoring system," *Pattern Recognit. Lett.*, vol. 24, no. 1–3, pp. 113–128, 2003.
[38] H. Zhang, G. Ananthanarayanan, P. Bodík, M. Philipose, P. Bahl, and M. J. Freedman, "Live video analytics at scale with approximation and delay-tolerance," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2017, pp. 377–392.
[39] B. Babenko, M. Yang, and S. J. Belongie, "Robust object tracking with online multiple instance learning," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 33, no. 8, pp. 1619–1632, Aug. 2011.
[40] K. Hsieh *et al.*, "Focus: Querying large video datasets with low latency and low cost," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2018, pp. 269–286.
[41] H. Zhang, G. Ananthanarayanan, P. Bodík, M. Philipose, P. Bahl, and M. J. Freedman, "Live video analytics at scale with approximation and delay-tolerance," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2017, pp. 377–392.
[42] H. H. Kim and Y. H. Kim, "Semantic video search using tagsonomies," in *Proc. 73rd ASIS&T Annu. Meet. Navigating Streams Inf. Ecosyst.*, 2010, pp. 1–2.
[43] S. Verstockt, O. Janssens, S. V. Hoecke, and R. V. de Walle, "Spatio-temporal video retrieval by animated sketching," in *Proc. Int. Conf. Comput. Vis. Theory Appl.*, 2013, pp. 723–728.
[44] S. H. Abdulhussain, A. R. Ramli, M. I. Saripan, B. M. Mahmmod, S. A. R. Al-Haddad, and W. A. Jassim, "Methods and challenges in shot boundary detection: A review," *Entropy*, vol. 20, no. 4, 2018, Art. no. 214.



Chen Zhang received the BS degree in electromagnetic wave propagation and antenna from Xidian University, Xi'an, China, in 2016. He is currently working toward the PhD degree in the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China. His research interests include video analysis and neural network.

Qiang Cao (Senior Member, IEEE) received the BS degree in applied physics from Nanjing University, Nanjing, China, in 1997, and the MS degree in computer technology and the PhD degree in computer architecture from the Huazhong University of Science and Technology, Wuhan, China, in 2000 and 2003, respectively. He is currently a full professor of the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology. His research interests include computer architecture, large scale storage systems, and performance evaluation. He is a senior member of the China Computer Federation (CCF) and a member of the ACM.

Hong Jiang received the BSc degree in computer engineering from the Huazhong University of Science and Technology, Wuhan, China, in 1982, the MASc degree in computer engineering from the University of Toronto, Toronto, Canada, in 1987, and the PhD degree in computer science from the Texas A&M University, College Station, Texas, in 1991. He is currently a chair and Wendell H. Nedderman endowed professor with Department of Computer Science and Engineering, University of Texas at Arlington. His present research interests include computer architecture, computer storage systems and parallel I/O, high performance computing, big data computing, cloud computing, and performance evaluation.

Wenhui Zhang (Student Member, IEEE) received the BS degree in mathematics from Wuhan University, Wuhan, China, in 2011. He is currently working toward the PhD degree at the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China. His research interests include erasure codes, storage systems, and parallel algorithms. He is a student member of the ACM.

Jingjun Li received the BS degree in computer science and technology from the Huazhong University of Science and Technology, Wuhan, China, in 2016. He is currently working toward the master's degree at the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology. His main research interests include computer architecture, machine learning, and large scale key-value storage systems.

Jie Yao (Member, IEEE) received the BS degree in computer science and technology and the MS, and PhD degrees in computer architecture from the Huazhong University of Science and Technology, Wuhan, China, in 2001, 2004, and 2009, respectively. He is currently a lecturer with the Huazhong University of Science and Technology. His research interests include computer architecture, large scale storage systems, and performance evaluation. He is a member of the China Computer Federation (CCF) and ACM.