This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2693281, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Exploiting Parallelism for Access Conflict Minimization in Flash-based Solid State Drives

Congming Gao, Liang Shi*, Cheng Ji, Yejia Di, Kaijie Wu, Chun Jason Xue, and Edwin H.-M. Sha

***Abstract*—Solid state drives (SSDs) have been widely deployed in personal computers, data centers, and cloud storages. In order to improve performance, SSDs are usually constructed with a number of channels with each channel connecting to a number of NAND flash chips, each flash chip consisting of multiple dies and each die containing multiple planes. Based on this parallel architecture, I/O requests are potentially able to access parallel units simultaneously. Despite the rich parallelism offered by the parallel architecture, recent studies show that the utilization of flash parallel units is seriously low. Our study shows that the low parallel unit utilization is highly caused by the *access conflict* among I/O requests. In this work, we propose Parallel Issue Queueing (PIQ), a novel I/O scheduler at the host systems. PIQ groups I/O requests without conflicts into the same batch and I/O requests with conflicts into different batches. Hence the multiple I/O requests in one batch can be fulfilled simultaneously by exploiting the rich parallelism of SSDs. Extensive experimental results show that PIQ delivers significant performance improvement especially for the applications which have heavy access conflicts.**

***Index Terms*—Solid State Drives, Parallelism, Access Conflict, I/O Scheduler.**

I. Introduction

Solid state drives (SSDs) are widely deployed in personal computers, data centers, and cloud storages given its well-identified advantages, such as shock resistent, high random access performance, low power consumption, and light-weight form factors [1] [2] [6] [23] [26]. In order to improve performance, SSDs are usually constructed with a number of channels with each channel connecting to a number of NAND flash chips, each flash chip consisting of multiple dies, and each die containing multiple planes [1] [8] [20]. In this parallel architecture, chips are the widely supported smallest parallel unit without taking dies and planes into consideration [1]. However, die and plane can also be able to process requests in parallel when the SSD is enhanced with advanced commands [13]. In this case, plane instead of chip will be the smallest parallel unit. Currently, there are several works proposed to exploit the rich parallelism architecture for performance improvement [1] [8] [4] [32] [11] [29] [12] [13] [15] [35]. However, recent work shows that the rich parallelism is hard to be exploited for many reasons [11] [18] [35]. For example, once there are successive I/O requests targeting the same parallel unit (e.g., channels, chips, dies, and planes), the latter requests may be blocked and wait until the occupation of the target unit is released [18]. This kind of I/O access characteristics are defined as *access conflicts*, which will induce a low parallel unit utilization. Figure 1 shows the preliminary results of parallel unit utilizations for chips and planes in an well configured SSD. The detailed experiment settings are presented in Section V. As shown in Figure 1, the parallel unit utilizations for the various benchmarks are mostly under 20% and 10% for chips and planes, respectively. The reason for the much lower plane utilization stems from the additional limitation for the plane level parallelism. In order to understand the parallel unit utilization space, the ideal parallel unit utilizations at chip and plane levels are evaluated and collected. When the scheduling queue length is infinite and a lot of burst requests arrive continually, the ideal utilizations can be obtained through issuing more requests without access conflicts at SSDs. To get the ideal utilizations, the scheduling queue length is set to 512 and the arriving time of requests is compressed by 10 times to simulate the burst requests in this work. Figure 1 presents the results, where there is a large improvement room for traditional chip and die utilization. In this work, approaches are proposed to reduce access conflicts within SSDs and then improve the parallel unit utilizations at chip and plane levels, respectively. Chip and plane level parallelism are the smallest parallel unit parallelism when advanced commands are supported or not. Once these two level parallelism are fully exploited, the rich channel level parallelism and die level parallelism can be fully obtained as well. Therefore, only the chip and plane utilizations are presented.

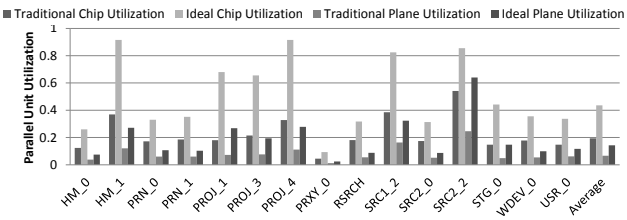


Fig. 1: Traditional and ideal chip/plane utilizations on a typical SSD.

Recently, many works have been proposed to improve the rich parallelism from different perspectives. First, for the organization of SSDs, Agrawal et al. [1] and Dirik et al. [8]

C. Gao, L. Shi, Y. Di, K. Wu and E. Sha are with the College of Computer Science, Chongqing University, Chongqing, P.R. China.

C. Xue and C. Ji are with the Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong.
E-mail: shi.liang.hk@gmail.com.

This work are partially supported by the Fundamental Research Funds for the Graduate Scientific Research and Innovation Foundation of Chongqing, China (Grant No.CYB14043 and CYS16019), NSFC 61402059, the Fundamental Research Funds for the Central Universities (106112016CDJZR185512), Huawei Innovation Research Program(HIRP), Chongqing Research Program cstc2014yykfB40007, NSFC (61472052 and 61572411), and National 863 Programs 2015AA015304.

evaluated different types of SSD organizations to investigate their impacts on performance cast by the number of channels and the number of chips per channel. Jung et al. [15] and Hu et al. [12] [13] evaluated different data allocation schemes with further considering the other two level parallel units, die and plane. These works presented that with a better SSD organization and/or data allocation scheme, the parallel unit utilization can be improved significantly. Another important factor that prevents better parallel unit utilization is the access conflicts. The access conflicts among successive I/O requests naturally prevent the multiple units in an SSD to be fully utilized in parallel. On recognizing the problem, Jung et al. [18] and Chen et al. [7] proposed approaches to reduce access conflicts. For example Jung et al. [18] proposed to expose the physical addresses of SSDs to the I/O requests. Then, requests are issued based on the physical addresses to explicitly exploit the parallelism of SSDs. Chen et al. [7] proposed to reduce the access conflicts among read requests by storing more conflicted requests in the buffer cache. Hence the performance improvement contributed from the degradation of read access conflicts is significant. He et al. [11] proposed to reduce the conflicts between I/O requests and internal activities (e.g., garbage collection, wear leveling) of SSDs. They proposed to maintain multiple page copies at different parallel units. When I/O requests are issued, the requests can be processed in the parallel unit with no internal activities. However, these works are commonly implemented in the SSD controller with hardware supports and additional storage overhead, which cannot exploit the host side information for improvement.

In this work, we propose a novel host side I/O scheduler, Parallel Issue Queueing (PIQ), to explore the rich parallelism of SSDs. Three types of access conflicts, i.e., read and write conflicts, read and read conflicts, and write and write conflicts, which affect the exploration of the parallelism, are fully taken into account and exploited. The key challenge in designing PIQ is to detect all these kinds of access conflicts among successive I/O requests at the host side. In order to solve this problem, a conflict detection approach is first proposed to obtain the conflict information at host side. With the knowledge of the detected conflicts, the proposed PIQ first separates the read and write requests into two groups to avoid the read and write interference [19]. Then, within each separated group, PIQ separates I/O requests according to the detected conflicts among them: the I/O requests without conflicts will be put into one batch, while the I/O requests with conflicts will be put into different batches. Finally, a batch level scheduling scheme is proposed for the request batches.

There are two assumptions for the proposed approaches: First, static data allocation scheme should be applied within SSDs; Second, internal activities do not migrate data among parallel units. From our investigation and study, we believe that these two assumptions are reasonable for state-of-the-art SSDs, which are also widely studied. For example, static allocation scheme has been widely used and studied in previous works [13] [12] [15]; data migration from GC and WL only happens in parallel units is also the default settings for most of recent works [1] [13] [18]. In addition to the above two assumption, there are some special designed SSD, such as open channel SSDs [34] [36], which open several internal information to the host systems. The proposed approaches can be easily integrated. Compared with the previous work [9], the extended version of this work has following additional contributions:

- Extended the basic PIQ to general parallel units, which then can exploit the internal parallelism for performance improvement;
- Proposed a batch-level scheduling scheme, which is designed to prioritize the write batches with long waiting time to avoid starvation;
- Efficient implementation of PIQ is presented, which is overhead negligible.

The rest of the paper is organized as follows. Section II presents the background and related works. Section III presents the problem in the parallelism exploration of SSDs. In Section IV, the details of PIQ are presented. Experiments are presented in Section V. Finally, Section VI summarizes the paper.

II. Background and Related Work

In this section, NAND flash memory based SSDs are first presented with their organizations and features. Then, the related works are presented, including parallelism exploration and the I/O scheduling schemes on the SSDs based storage systems.

A. Solid State Drives

In this section, the organization of SSDs is first presented. Then, the controller for the parallel flash array is introduced.

1) Parallel Organization: Figure 2 shows an example on the organization of SSDs with 4 channels, 2 chips per channel, 2 dies per chip, and 2 planes per die. This four level parallel architecture of SSDs has been widely studied and exploited for performance improvement. For the relationships among these four level parallelism, they have been well studied in previous works [12] [13] [15]. The channels and chips are the naturally supported parallel units, which can process I/O requests in parallel. The dies and planes, which are called internal parallel units, are also able to process I/O requests in parallel if the SSDs are enable with the support of advanced commands [12] [13]. In the following, we give details for these four level parallel units.

The first level parallelism is exploited via channels, where the requests issued to different channels can be transferred in parallel. The second level of parallelism is exploited via chips. Similar to the channels, the chips resided in the same channel are independent with each other, hence requests issued to different chips can be processed in parallel as well. Different from the above two levels' parallelism, the die and plane level parallelism are called internal parallelism of SSDs, which is supported by advanced commands [13]). These advanced commands are enabled with additional restrictions, which limit the exploration of them for performance improvement. That is, even when there are several requests issued to multiple planes without access conflicts, these requests are not allowed to be processed in parallel if the restrictions are not met.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2693281, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

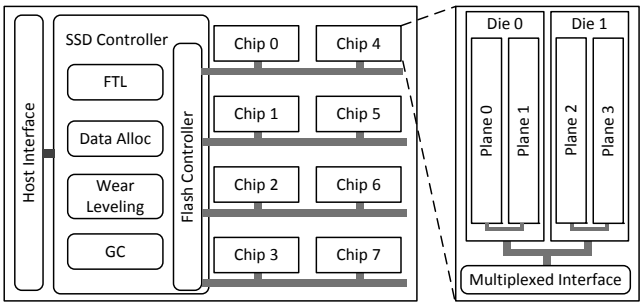


Fig. 2: The Architecture of SSD with four levels: Channels, Chips, Dies and Planes

2) Controller Design: The controller of SSDs is a key component in the design of SSDs, which takes charge of all issues of flash based SSDs. As shown in Figure 2, these components implemented in the SSD controller at least include flash translation layer (FTL), data allocation (DA), garbage collection (GC), and wear leveling (WL). With the equipment of these components, all issues of SSDs are well dealt.

For these mentioned components, they are designed to embrace different duties respectively. Since flash memory is not able to conduct in-place-update, FTL is designed to maintain a mapping between logical addresses and physical addresses. In this work, page mapping [10] is employed in the controller. DA is designed to manage the flash array of SSDs and decides the distribution of the data, which is highly correlated with the access conflict of requests [15] [38]. There are two types of data allocation schemes, static and dynamic data allocation schemes. These two data allocation schemes are the two major data allocation schemes for SSDs and have their advantages and disadvantages: For static allocation, it is easy to implement. For dynamic allocation scheme, it is good for performance and wear leveling. In this work, we select the static allocation scheme is because static allocation scheme has been widely studied in previous work for its simple design and effectiveness. In the following, we present the related works on studies of these two types of data allocation schemes. GC is used to collect invalid pages during data updates since the updated data are written to new places. The last component is wear leveling (WL), which is designed to extend the lifetime of SSDs.

B. Related Work

1) Data Allocation Schemes: In previous works, there exist several works aiming at discussing the difference between static allocation scheme and dynamic allocation scheme. For static data allocation scheme, Jung et al. [14] proposed to reorder requests for improving the write performance at host interface. Jung et al. [18] proposed another scheduler in controller to improve the performance via exposing the physical addresses of requests in advance. Jung et al. [15] did a series of experiments, where all types of static allocation schemes are measured and used to indicate the different types of static allocation schemes induced performance impact on SSDs. Jung et al. [17] proposed approach for improving the poor flash resource utilization, where the physical addresses of requests are determined by static allocation scheme, after that, these transactions accessing different flash resources can be issued together. In general, we can find that static allocation scheme has been widely studied and applied in recent works. For dynamic data allocation scheme, Shin et al. [32] found that the significant performance improvement of small and random requests benefit from the applying of dynamic allocation scheme inside SSDs. Hu et al. [12] [13] also verified and presented that the performance can be improved significantly for write dominant workloads with applying of dynamic allocation scheme. Chen et al. [4] [3] showed that the dynamic allocation scheme is possible to reduce the parallelism for read requests when a lot of data are distributed randomly.

In this work, we use the static data allocation scheme for its simple design and effectiveness.

2) Exploration of Parallelism: For the rich parallelism in the organization of SSDs, several previous works have been proposed to exploit them for performance improvement [4] [7] [11] [12] [13] [15] [16] [18] [28] [30] [32] [35]. Generally, these works can be divided into two types: channel/chip level parallelism exploration and internal level parallelism exploration.

For the widely supported parallelism, channel and chip level parallelism, Seol et al. [30] and Park et al. [28] proposed to exploit the multi-channels of SSDs from the view of write buffers to improve the write performance. Chen et al. [7] proposed a buffer cache management approach for SSDs to solve the read conflict problem by exploiting the read parallelism of SSDs. He et al. [11] proposed to exploit the multi-level parallelism for the internal conflict minimization through replicating the pages which may cause conflict.

For internal parallelism, Jung et al. [15] and Hu et al. [12] [13] showed that data allocation schemes of SSDs were very important in the exploration of parallelism. Wang et al. [35] proposed a PCFTL (Plane-Centric FTL) to exploit the internal parallelism of SSDs, which applying intra-plane `copy-back` operation to exploit the parallelism at plane level with the support of advanced commands. Jung et al. [15], Hu et al. [12] [13], and Shin et al. [32] proposed to exploit different levels of parallels to improve the write performance. Jung et al. [16] showed that random read performance of SSDs was the worst one compared with other patterns.

However, none of above works propose to reduce the access conflicts among successive I/O requests. Jung et al. [18] proposed a resource contention aware approach, physical address queueing (PAQ) under FTL, to issue more I/O requests without resource contention to the SSDs at one time. However, PAQ was implemented in the SSD controller, which required hardware modification.

In this work, a normal I/O scheduling scheme at host system is proposed to exploit the rich parallelism of SSDs to reduce both conflicts of read and write requests before they are issued to SSDs.

3) I/O Scheduling for Solid State Drives: Traditional I/O scheduling schemes for hard disk drives (HDDs) include no operation (NOOP), deadline, anticipation, and completely fair queueing (CFQ) [33], which were implemented in the host system. However, none of them can work efficiently with SSDs [19] because of the difference characteristics between HDDs

and SSDs. Recently, several new I/O scheduling schemes have been proposed for SSDs based systems [7] [18] [19] [22] [27] [31] [35]. These I/O schedulers are designed in flash controller or host system.

Jung et al. [18] proposed an I/O scheduler to reduce the access conflict in the flash controller. In this way, physical addresses (PAs) are exposed to I/O scheduler for scheduling requests without conflicts. Chen et al. [7] proposed a cache replacement policy in the flash controller for resolving the read conflict problem. However, this work had similar issues and only worked for one type of conflict.

At the host side, several scheduling schemes [19] [22] [27] [31] were proposed. Kim et al. [19] first proposed to bundle write requests and schedule read requests in advance to reduce the impact of slow write operations on read performance. Dunn et al. [22] exploited the block locality to improve the write performance. Park et al. [27] and Shen et al. [31] proposed two schedulers to achieve fairness among multi-tasks on SSDs. FIOS [27] scheduled requests with the awareness of read and write interference of SSDs, and FlashFQ [31] scheduled requests with the awareness of the internal parallelism for different applications. However, none of these works are proposed to solve the access conflict issue at the host side.

In this work, we propose a host-side I/O scheduler called PIQ to improve both read and write performance by minimizing all types of access conflicts. In addition, it can be used towards all existing SSDs as it does not require any changes in SSDs, which is a significant advantage over existing controller-level techniques.

III. Problem Statement

The parallel unit utilization of SSDs is seriously low [16] [29], which has been presented in Figure 1. One of the key issues for the low parallel unit utilization is the access conflicts among successive I/O requests. When several I/O requests in the scheduling queue are preparing to be issued in a short time interval, access conflicts may happen since the accessed parallel units are shared by multiple requests. Once there exit access conflicts among successive I/O requests, the former request will occupy the parallel units, in return, the latter one accessing the same parallel units will be blocked and delayed. As a result, the utilization of parallel units are going to be reduced.

In order to reveal the impact of access conflicts at the host side, we evaluate the percentages of conflict requests and performance impact induced by access conflicts for different applications. The detailed experiment settings can be found in Section V. Figure 3(a) shows the percentages of conflict read and write requests at the chip level. The y-axis represents the percentages of conflict requests to the total read or write requests. It can be observed from Figure 3(a) that the access conflicts among requests commonly exist in applications. Secondly, we also observe that the percentages of conflict write requests vary significantly among applications, while the percentages of conflict read requests are roughly at the same range. On average, 26.2% of read requests, and 23.3% of write requests are conflict requests. The conflict is more serious at the plane level since there are additional constraints.

Conflict requests will directly induce performance degradation. The performance degradation is presented in Figure 3(b), which has a matching pattern with Figure 3(a). In order to show the performance degradation, the experiment is redesigned and result are presented as follows. First, the ratio of total I/O latency of traditional case over the I/O latency in ideal case is applied as the metric. We defined that the access conflict happens at the case when there are idle parallel units and several requests are queued in the IO queue at the same time. In order to simulate the case without access conflict, we build an ideal case where all incoming requests can be distributed to the rest idle parallel units. In this case, the measured read and write latencies are optimal when the access conflicts are removed. The first observation is that read and write response time are both increased, and on average by 7.1 times and 4.8 times, respectively. The second observation is that the latency increase varies significantly among applications. The reason behind the various results is mainly due to different amount of access conflicts among the I/O requests of these applications. Since the conflict requests have to wait until the access parallel units are released by the former requests, and hence the operation latency will increase significantly.

Eventually, we can draw the conclusion that access conflicts commonly exist in various applications running on SSDs, and the applications with stronger access conflicts will see more significant performance degradation. If we can detect the conflicts at host system side, there will exist a flexible space to schedule the I/O requests for achieving rich parallelism.

IV. Parallel Issue Queueing for Parallelism Exploration

In this section, a host side access conflict detection approach is first proposed. Then, based on the detected access conflicts, PIQ proposes to separate I/O requests with conflicts into different scheduling batches. Finally, a batch scheduling approach is presented in this section. Figure 4 shows the flash memory based SSDs with PIQ implemented in the host systems.

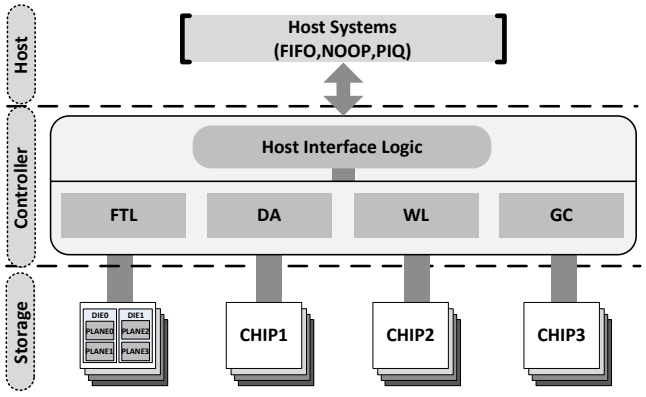
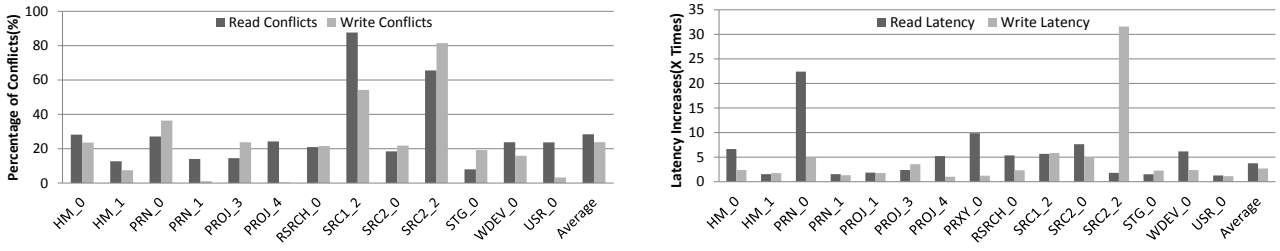


Fig. 4: Organization of SSD based storage systems, where PIQ is implemented in host systems.

A. Access Conflict Detection

Access conflict is highly correlated with the issue time of I/O requests and physical locations of data. If the issue time

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2693281, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



(a) Access Conflicts (b) Performance Degradation

Fig. 3: Access Conflict and Performance Impact at Chip Level: (a) The percentages of conflict read and write requests to the total I/O requests; (b) The performance degradation induced from access conflicts.

interval among these successive I/O requests is small enough, and there exist parallel units shared by these requests, these I/O requests will conflict with each other.

1) Issue Time Interval: Since the proposed approach focuses on the design of an I/O scheduler, all the issue time intervals among the outstanding I/O requests and waiting requests in the I/O queue have the potential in access conflict. Based on this observation, outstanding I/O requests and waiting requests in the I/O queue are taken into account during access conflict detection.

2) Data Locations: Location of data is highly correlated with two factors, data allocation scheme and I/O request's LSN. Recently, data allocation scheme has been widely studied [13] [12] [15] [32], where the location of data is determined based on LSN. Once LSN is given, the location of data can be computed based on the data allocation scheme. In this work, we use *location vector* to represent the data location of requests, where the number of bits in the vector equals to the number of parallel units in the SSDs. Initially, location vector is set with all zeros. When each request is added to the I/O queue, the corresponding bits of location vector of request should be set based on the access parallel units. We use the following settings to generate the location vector:

- The number of parallel units in the SSDs is N_{pu} and the size of flash page is $Page\_Size$ in bytes;
- I/O requests are defined with $(\mathrm{R}_i, LSN,\ Sector\_Number,\ T)$, where R_i is the request type, such as read or write requests, LSN is the logical sector number that the request is to access, $Sector\_Number$ is the accessing data size in term of the number of sectors, $Sector\_Size$ is the sector size in bytes, and T is the time when the I/O request is added to the I/O queue.
- The data allocation scheme is Channel-Chip-Die-Plane [1] as example, which stripes the requests across channels, chips, dies and planes in order. Note that other types of data allocation scheme also works for PIQ.

Based on these settings, the set bits of the location vector of a request in the SSDs can be determined, which are from pu_first to pu_last, as follows.

$$pu\_first = (\lfloor \frac{LSN \times Sector\_Size}{Page\_Size} \rfloor)\%N_{pu};$$
$$pu\_last = (\lceil \frac{(LSN + Sector\_Number) \times Sector\_Size}{Page\_Size} \rceil - 1)\%N_{pu}; \quad (1)$$

Then, based on pu_first, pu_last, and N_{pu}, the location vector towards different level parallelism is generated.

As we mentioned above, the smallest parallel unit can be changed from chip to plane when the advanced commands are supported. Two examples at different levels of parallelism are presented to illustrate the processes on generating the location vectors. First, we assume that $Page\_Size$=4KB, and a read request is $(\mathrm{R}_i, LSN, Sector\_Number, T) = (r, 14896, 16, 0.02332)$, and $Sector\_Size$=512B. If the advanced commands are not enable, then the smallest parallel unit is chip, and N_{pu}=8. We assume that there are two channels, each channel contains four chips. We can derive that, pu_first = 6 and pu_last = 7, which means that access parallel chips of the read request are chip 6 and chip 7. Then the location vector of this request, $V(\mathrm{R}_i)$, is (00000011) = 3.

When the advanced commands are enabled, the smallest parallel unit is plane. We assume that each chip contains one die, and each die is composed of two planes, thereby, N_{pu} = 16. We can derive that, pu_first = 6 and pu_last = 7. Then the location vector of this request, $V(\mathrm{R}_i)$, is (0000001100000000) = 768. Compared with access conflict detection at chip level, access conflict detection at plane level is designed with considering the rich internal parallelism. The inherent internal parallelism is potential in further improving the performance of SSDs. However, the restrictions of advanced commands cannot be satisfied in host system side due to the opaque feature of SSDs.

The conflict detection is realized by the location vectors, which can be obtained without considering the variation of data allocation schemes. The proposed conflict detection approach can be applied to any existing data allocation schemes. However, the above conflict detection scheme has two limitations when GC or WL is activated: The first one is that data migration for GC and WL should be within parallel units. Otherwise, it can be wrong. In this work, we assume that the GC and WL do not migration data among parallel units; the second one is that the full parallelism might not be always achieved at the host system side due to GC and WL activates. This is because we cannot detection these activities at the host systems. In this case, if these activities happen, requests issued will be blocked. Since this is another new topic for this work, we will leave it as our future work.

3) Conflict Detection: Based on the location vectors of I/O requests, the access conflict can be detected among the waiting and outstanding requests in the I/O queue. If there exists a parallel unit shared by two requests in the I/O queue, these two requests are defined as conflict requests. For example, if the location vectors of requests R_i and R_j are $V(\mathrm{R}_i)$ and

$V(\mathrm{R}_j)$, the conflict of these two requests can be detected as follows:

$$Conflict = V(R_i)\&V(R_j); \tag{2}$$

If $Conflict$ == 0, these two requests do not conflict with each other. Otherwise, they conflict.

Figure 5 shows an example of conflict detection. As show in Figure 5, there exist 8 parallel units, which are able to be accessed simultaneously. In the I/O queue, there are five I/O requests. Based on the allocation scheme and I/O request's location information, the location vectors of each request are computed and recorded. Then, based on the conflict detection of Formula 2, the access conflicts among requests can be identified. For example, requests 1 and 3 conflict at parallel unit 5 (12 & 6=4), requests 2 and 4 conflict at parallel unit 1 (64 & 96=64), and requests 3 and 5 conflict at parallel unit 6 (6 & 3=2).

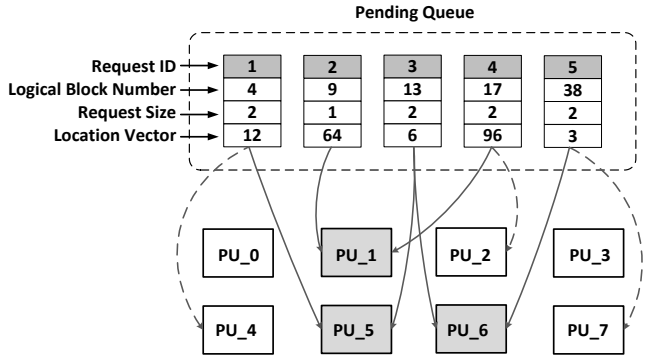


Fig. 5: Access Conflict of I/O Requests.

B. Parallel Issue Queueing (PIQ)

In this section, PIQ is proposed to exploit the parallelism of SSDs for access conflict minimization. The basic idea of PIQ is to separate I/O requests into batches, where the resided requests in each batch have no conflict with each other. In addition, in order to further improve the performance, scheduling scheme on the batch level is also proposed to avoid the potential write request starvation issue.

1) Request Separation: Based on the types of requests, access conflicts can be classified into three types, read and write conflicts (RWC), read and read conflicts (RRC), and write and write conflicts (WWC). Based on this classification, separating schemes are proposed to solve these three types of conflicts. There are three separating schemes as follows:

- Read and Write Separation (RWS): Read and write requests are separated into different batches to avoid RWC. Write operations are much slower than read operations in flash memory [21] [25], prioritizing the scheduling of read operations has benefit in reducing RWC between read and write requests;
- Read and Read Separation (RRS): read requests are separated into different batches to avoid RRC between read requests;
- Write and Write Separation (WWS): write requests are separated into different batches to avoid WWC between write requests.

2) Request Batches: Once the request separation schemes are decided, request batches are proposed to group requests without conflicts into batches. Similar to the request location vector, we use the *Batch location vectors* to record the access parallel units of all the requests in each batch. Initially, batch location vector is set to 0. When one request is added to the I/O queue, the conflict is detected by checking the location vector of entering I/O request with the batch location vectors. Similar to the request conflict detection, the location vector of the request is checked with the batch location vector as follows.

$$Conflict = V(B_i)\&V(R_i); \tag{3}$$

where B_i is the checked batch and R_i is entering request. When $Conflict$ is 0, R_i is added to the batch, and the batch vector is updated as follows:

$$V(B_i) = V(B_i)|V(R_i); \tag{4}$$

Otherwise, a new batch is created, and the entering request is added to the new batch. Since PIQ takes outstanding requests and waiting requests in the I/O queue into account, when new requests are added to the I/O queue, they are checked and added to the corresponding batch. With this approach, all the conflicts in one batch are avoided before the requests being issued to the SSDs. During scheduling, PIQ prefers to schedule read batch to avoid the RWC. Note that dependencies between I/O requests are avoided when they are added to the I/O queue using traditional approaches.

3) Batch Level Scheduling: Similar to previous work which proposed to prioritize read over write requests, PIQ scheme is also designed to prioritize the read batches over write batches. In addition, a newly created read/write batch is added to the tail of the read/write batch list. With these approaches, the read and write batches can be issued without conflict.

The above scheme still have two issues, which may induce performance degradation. First, write batches may suffer starvation especially when a sudden burst of read requests is issued and the write batch has been waiting for a long time. Second, write requests are prone to be delayed not only from the read over write scheme, they are also prone to be delayed for the generation of the write batches. In order to solve these issues, we propose to use a waiting time threshold based scheduling scheme for the write batches. If the write batch at the head, which is the earliest created batch, have waited over the threshold, it should be scheduled as soon as possible. However, the set of the threshold is hard. A small threshold would induce many write batches scheduled ahead, which will decrease the read performance. A long threshold would be not able to solve the starvation issue. In the design, we adopted a fixed threshold for several reasons. First, a fixed threshold is simple with little CPU consumption. Second, based on the evaluation of several workloads, we find that fixed threshold is suitable enough in preventing the starvation of write requests, and improving the write performance with little penalty of read performance. Note that only one write batch at one time can be prioritized to avoid the penalty of read performance.

4) Comparison Between NOOP and PIQ: Figures 6 and 7 show examples on the comparison of NOOP and PIQ. In this example, we assume there exist six I/O requests in the I/O queue (three read requests (R0, R1, R2) and three write requests (W0, W1, W2)), with each request accessing 2 parallel units. These six I/O requests are added to the pending queue in the First In First Out (FIFO) order, from left to right. We assume that SSD is composed of 4 parallel units. The data location of the six I/O requests are marked in the four parallel units and the location vectors of these requests are produced.

In Figure 6, we assume that NOOP is used as the I/O scheduler. Then, I/O requests are scheduled in the order of FIFO order. We find that all six I/O requests conflict with each other. In this case, these six I/O requests are processed sequentially, which requires three read and three write cycles to service these requests.

In Figure 7, RWS, RRS, and WWS are applied to the I/O queue to separate the I/O requests into different batches. In the first step, RWS is applied, where read and write requests are separated to avoid RWC. Second, RRS and WWS are applied to separate the read and write requests into different batches. Take R_0, R_1 and R_2 as an example, where $V(R_0) = 3$, $V(R_1) = 6$, and $V(R_2) = 12$. Initially, there are no batches in the I/O queue. When R_0 is added, a read batch is created, and the batch vector B_0 is initialized with 3. When R_1 is added to the queue, R_1 is checked with B_0, where $B_0 \& R_1 = 2$. These two requests have conflict (RRC). In this case, a new read batch is created, then, the batch vector B_1 is initialized and calculated, respectively. When R_2 is added to the queue, R_2 is checked with B_0, where $B_0 \& R_2 = 0$. These two requests are not conflicted with each other. Then, R_2 is added to B_0, and location vector of B_0 is updated based on Formula 4 to 15. When all batches are created, they are ordered based on their creation time in read and write batch list, respectively. One special case is when one write batch waiting for a too long time, it will be scheduled in priority. In Step II of Figure 7, we use a tag at the head write batch for the prioritized scheduling. If its waiting time is longer than the threshold, it is set and vice versa. Finally, PIQ issues the I/O requests in batches, where all the batches are processed one by one. We find that with PIQ, only two read and two write cycles are required to service these requests.

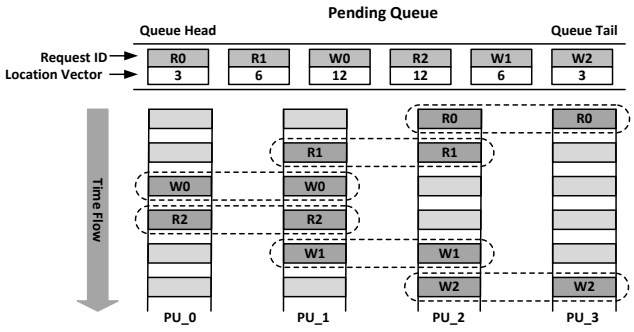


Fig. 6: NOOP: requests are issued in FIFO order.

Overhead analysis: The implementation of PIQ needs to maintain batch location vectors, batch lists and waiting time of each write batch in the batch list. The size of batch

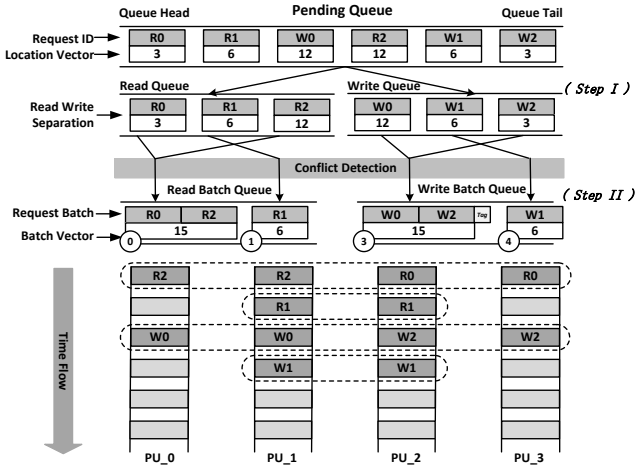


Fig. 7: PIQ: RWS, RRS, and WWS are applied to the queue to separate the I/O requests into different batches based on the detected conflicts.

location vector and request location vector are determined by the number of parallel units in the SSDs. In this case, the size of each batch and request location vector are N_{pu} bits. We assume that the length of I/O queue is N_{piq}, then the maximal size of space required by the batch location vectors is $N_{piq} \times N_{pu}$ bits. In addition, each write batch needs one timestamp to record its creation time. By default, the length of the I/O queue is 64. This storage overhead is negligible for an I/O queue at the host side. For the computation overhead of PIQ, there are only `MOD`, `OR` and `AND` operations for detecting access conflicts. In general, the computation overhead of PIQ is negligible.

V. Experiment and Analysis

In this section, six related schemes are implemented for evaluating the effectiveness in the performance improvement of SSDs.

- The first scheme is "NOOP", which represents the traditional I/O scheduler implemented at host side.
- The second scheme is "RWS", which is implemented based on NOOP, where RWCs are detected and released.
- The third scheme is "PIQ_R", which is implemented over "RWS", where RRCs among read requests are detected and released;
- The fourth scheme is "PIQ_W", which is implemented over "RWS", where WWCs among write requests are detected and released;
- The fifth scheme is "PIQ", which is implemented with integrating "PIQ_R" and "PIQ_W". This scheme is implemented to show the effect of proposed approach on the overall performance;
- The six scheme is "PIQ+", which is an enhanced version of the basic "PIQ" scheme, where the batch level scheduling scheme is applied for avoiding the write starvation issue.

In the following, these six schemes are evaluated at chip and plane level respectively.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2693281, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

A. Experiment Setup

In this paper, we use a trace driven simulator, SSDsim [12] [13], to verify the proposed framework. SSDsim has been widely applied in the exploration of parallelism of SSDs. In this study, we modeled an 128GB SSD, which is configured with 8 channels with each channel equipped with 8 chips. Each chip is consisted of 2 dies, and each die is constructed with 2 planes. Each plane contains 2048 flash blocks. Each flash block consists of 64 pages with a page size of 4KB. Page mapping scheme is implemented as the default FTL mapping scheme [10]. Greedy garbage collection scheme and dynamic wear leveling scheme is implemented. The over-provisioning ratio is set to 10% of the SSD, which complies with the setting in several recent works [1] [12] [13]. For the data allocation scheme, the most widely used Channel-Chip-Die-Plane scheme is applied since it has the best performance in term of the parallelism of SSD [1]. The default length for I/O queue is set to 64. The experiment settings represent a typical modern SSD.

The workloads used in this work include a set of MSR Cambridge traces from servers [24]. In this work, original traces without compression are obtained and used in proposed approach and baseline. For achieving stable and convincing results, each trace contains 500,000 records for running and additional 500,000 records for warming up the devices.

B. Experimental Results

1) PIQ Evaluation at Chip Level: In this section, we present the performance comparison among NOOP, RWS, PIQ_R, PIQ_W, PIQ, and PIQ+. All these schemes are implemented at chip level. Figure 8 shows the results for the raw execution latencies of read and write requests. The top parts of Figure 8 indicate the process time of requests and the bottom parts of Figure 8 represent the waiting time of requests. In order to analyze the performance improvement induced from the proposed approaches, RWS, PIQ_R, PIQ_W, PIQ, and PIQ+ are explained one by one.

RWS: As shown in Figure 8, RWS, which prioritizes the process of read requests, has little performance improvement for several applications. In order to understand the reasons for the different performance improvement of RWS, Figure 10 presents the results in terms of the percentages of prioritized read requests. As shown in Figure 10, the percentages of prioritized read requests for most applications are within 10%. In this case, the read latency can be finitely improved. For other workloads, such as HM_0, PRXY_0, SRC1_2, SRC2_0 and SRC2_2, they have much more prioritized read requests. In this case, their read latencies are improved significantly, as shown in Figure 8(a). Taking HM_0 as example, the read performance is improved by 71%, with 37% of read requests prioritized. However, for these workloads, the write requests can be delayed due to the prioritized read requests in turn. As shown in Figure 8(b), the write latencies can be slightly increased. For instance, PROJ_4 is a read intensive one, which has a slightly increased write latency.

PIQ_R: PIQ_R, which integrates RWS with RRS, is proposed to further detect and relax the read conflicts among read

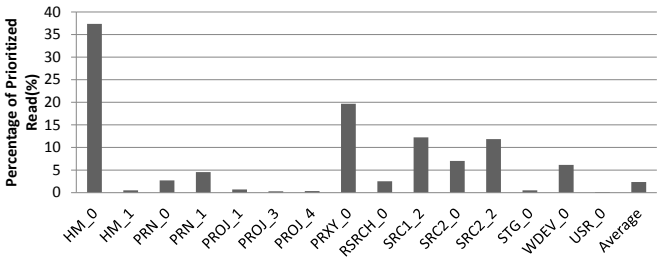


Fig. 10: Percentages of prioritized read requests comparison between NOOP and RWS.

requests by exploiting the parallelism of SSDs. Figure 8(a) shows that the read latency of PIQ_R is reduced significantly for most applications (13 out of 15). The most noteworthy is the performance improvement of PROJ_4, which is completely generated from RRS. In order to understand the reason for the significant performance improvement resulted from RRS, the raw request waiting time for read requests and chip utilization are evaluated, which are presented in Figure 9(a) and Figure 11. The request waiting time for read requests is produced while read request conflicts occur. As shown in Figure 9(a), the raw request waiting time has a matching pattern with the performance results presented in Figure 8(a). For instance, PROJ_4, which achieves significant performance improvement, is able to obtain a significant waiting time degradation as well. However, we also find that PROJ_1 has little read performance improvement. As shown in Figure 9(a), we find that the raw read request waiting time of PROJ_1 is reduced slightly. This is because there are little read conflicts for PROJ_1 and these access conflicts induced little read waiting time increases, as shown in Figure 3.

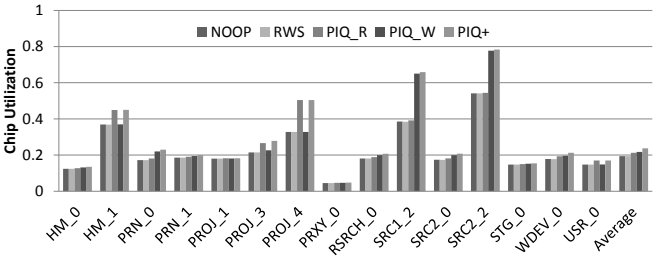


Fig. 11: Chip utilization of NOOP, RWS, PIQ_R, PIQ_W, and PIQ.

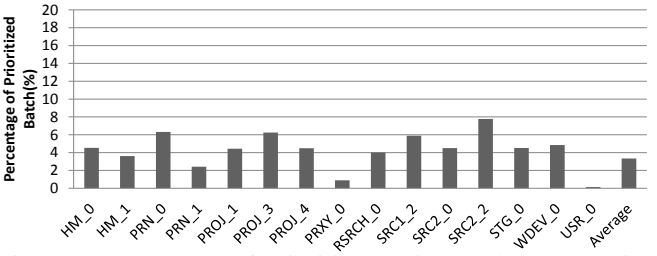
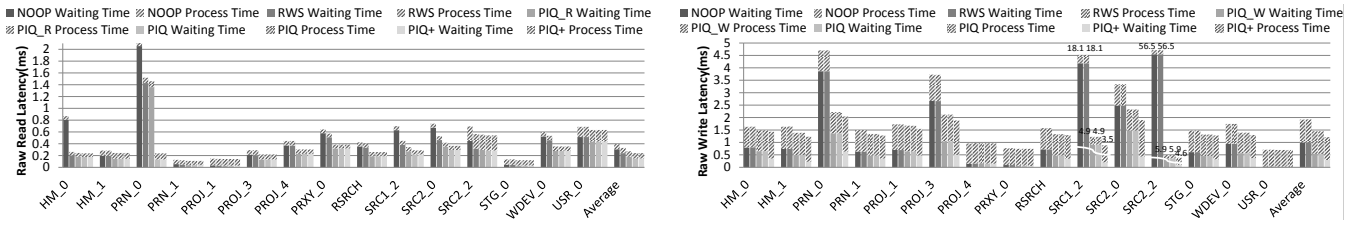


Fig. 12: Percentages of prioritized write batches comparison between PIQ and PIQ+.

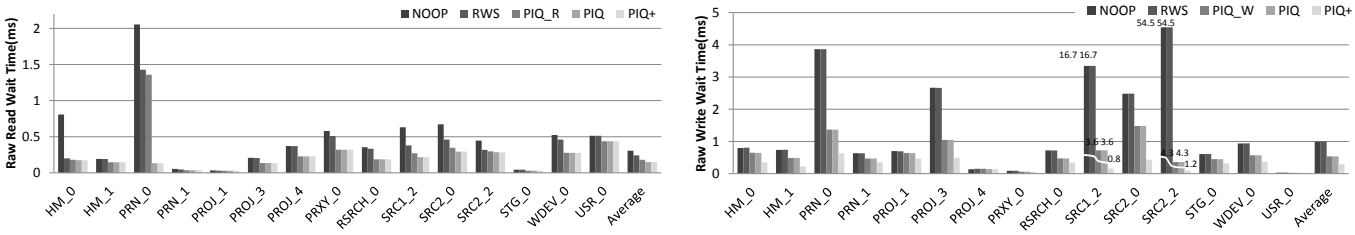
As a result, the read latency, the average raw request waiting time and chip utilization are improved by 41.7%, 51.3%, and 9.2% on average, respectively. Based on these results, we draw the conclusion that PIQ_R works for read intensive workloads,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2693281, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



(a) Read Latency Comparison (b) Write Latency Comparison

Fig. 8: Performance comparison among NOOP, RWS, PIQ_R, PIQ_W, and PIQ.



(a) Raw Read Request Waiting Time (b) Raw Write Request Waiting Time

Fig. 9: Raw request waiting time comparison among NOOP, RWS, PIQ_R, PIQ_W, and PIQ.

where the conflicted read requests can be processed in parallel.

PIQ_W: PIQ_W, which is implemented by taking WWS into consideration, is proposed to detect and relax the write request conflicts and exploit the parallelism for performance improvement. As shown in Figure 8(b), the write latency is reduced significantly for most applications. Similarly, the performance improvement of SRC1_2 is almost resulted from WWS. In order to reveal the reason for the improvement of write performance, the raw request waiting time for write requests and chip utilization are collected, as shown in Figure 9(b) and Figure 11. Write request waiting time is induced when write request conflicts occur among write requests. As shown in Figure 9(b), the average request waiting time induced by write request conflicts of SRC1_2 is reduced by 78.4% while the performance improvement is significant as well. However, we also find that USR_0 has little write performance improvement. The reason is because, as described in Figure 9(b), the raw write request waiting time is reduced slightly. For bringing the deeply reason into light, Figure 11 is presented with measuring the chip utilization of each workload. Compared with SRC1_2, the chip utilization of USR_0 is significantly lower, which produces the variation in the performance improvement.

On average, the write latency, the average raw request waiting time and chip utilization are improved by 33.7%, 46.1%, and 11.9%, respectively. Based on these results, we conclude that PIQ_W works for write intensive workloads, where the conflicted write requests can be processed in parallel.

PIQ: PIQ is proposed to take RRS and WWS into consideration for performance improvement. As shown in Figure 8, PIQ is able to further improve the read performance compared with PIQ_R. The reason is that the reduced write latency achieved by PIQ_W is still more time-consuming than read latency. Hence, PIQ can achieves better read performance, when the incoming read requests following the batched write requests suffer from less waiting time. Due to the same reason, the write performance improvement achieved by PIQ is small compared with PIQ_W.

On average, the read latency is reduced by 41.7%, the write latency is reduced by 33.8%, the read request waiting time is reduced by 51.3%, the write request waiting time is reduced by 46.3%, and the chip utilization is increased by 22.0%, compared with NOOP. Based on these results, we conclude that PIQ works effectively for I/O intensive workloads and shows significant performance improvement.

PIQ+: PIQ+, which is implemented over PIQ, takes the batch level scheduling into consideration. Currently, PIQ+ adopts a fixed waiting time threshold which is set to 1ms. With the evaluation of many workloads, we find that 1ms is enough in preventing the starvation of write requests with little penalty of read performance. Figure 8 presented the PIQ+ induced performance improvement. We can find that the read performance is almost the same as the PIQ. However, write performance, in contrast, is improved significantly. As shown in Figure 8(b), the average write performance is improved by 5.9% compared with PIQ. In comparing with NOOP, the average write performance is improved by 39.7%.

In order to understand the efficiency of PIQ+, the percentages of prioritized write batches over total write batches are collected, which is presented in Figure 12. As shown in Figure 12, several workloads, such as SRC2_0, which assign many write batches with higher priority, achieve better performance improvement from PIQ to PIQ+.

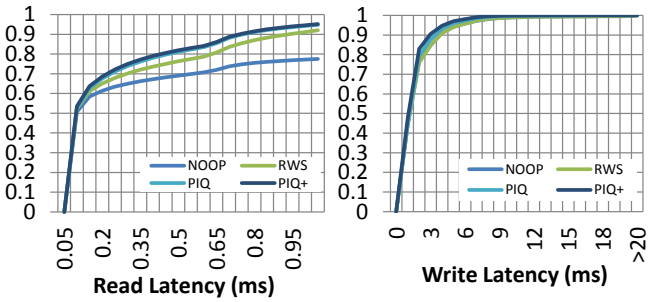
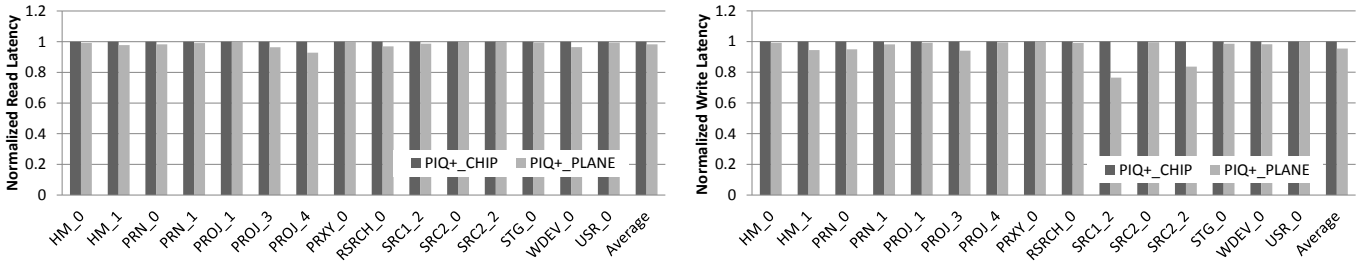


Fig. 13: Cumulative Distribution Function (CDF) of Read and Write Requests for HM_0.

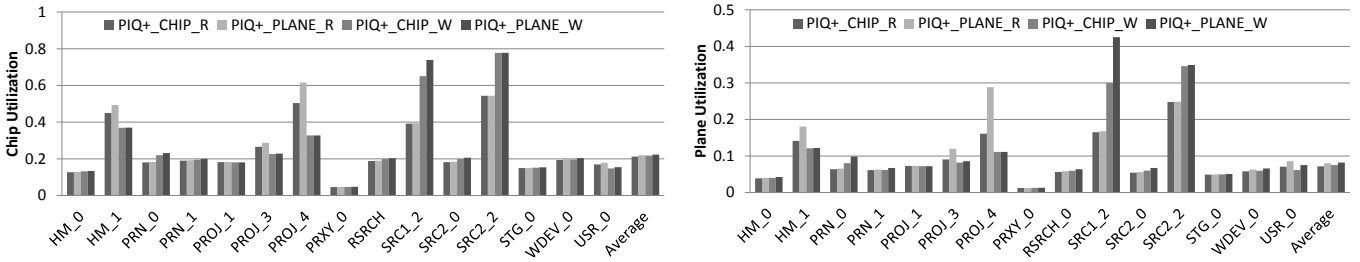
In addition, for the tail latencies of requests, the results

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2693281, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



(a) Read Latency Comparison (b) Write Latency Comparison

Fig. 14: Performance Comparison among NOOP, PIQ_CHIP, and PIQ_PLANE.



(a) Chip Utilizations (b) Plane Utilizations

Fig. 15: Chip and Plane Utilizations of PIQ_CHIP_R, PIQ_PLANE_R, PIQ_CHIP_W, and PIQ_PLANE_W.

are measured and presented in Figure 13, where a typical workload, HM_0, is selected. Compared with the tail latencies of read and write requests of NOOP, PIQ+ is able to reduce the tail latencies of read requests and write requests significantly. This is because that the proposed PIQ+ not only improves the average read and write requests, but also further reduces the tail latencies through avoiding the starvation of write requests.

As a result, PIQ+ achieves better performance compared with PIQ. Compared with NOOP, the write latency, the average write request waiting time, and chip utilization are improved by 39.7%, 70.5%, and 22.0%, respectively, while the read latency is almost the same as PIQ. The tail latencies of read and write requests of PIQ+ are also further reduced compared with PIQ.

In order to further show the performance improvement brought by PIQ+, the read and write performance comparison among NOOP with static allocation scheme, NOOP with dynamic allocation scheme and PIQ+ is measured and presented. The implemented basic static-NOOP and dynamic-NOOP schemes are the typical schemes in representing the scheme in current SSDs. The data allocation schemes applied within SSD are the most typical data allocations schemes [12] [13] and the host I/O scheduler is the traditional NOOP scheduler, which outperforms other host I/O scheduler [22]. Figure 16 shows the performance comparison among these three cases. In these two figures, we can find that there exist half of all traces (8 out of 15) showing that static data allocation scheme is more effective than dynamic data allocation scheme at read performance of SSDs. This is because that static data allocation scheme is able to strip data across parallel units evenly so that read requests can be processed in parallel [13] [12]. However, for some extreme case, static allocation scheme does not show good read performance. This is because these traces have serious access conflict, which would induce bad read performance. For example, PRN_0 and SRC2_2, which has the worst conflict, do not show good performance with static data allocation. In addition, we can find that PIQ+ achieves best read performance compared with Static-NOOP and Dynamic-NOOP. For write performance, PIQ+ is able to improve the poor write performance of Static-NOOP though Dynamic-NOOP achieves the best write performance. That is, PIQ+ not only can further improve read performance, but also can achieve better write performance while static allocation is applied in read dominant workloads.

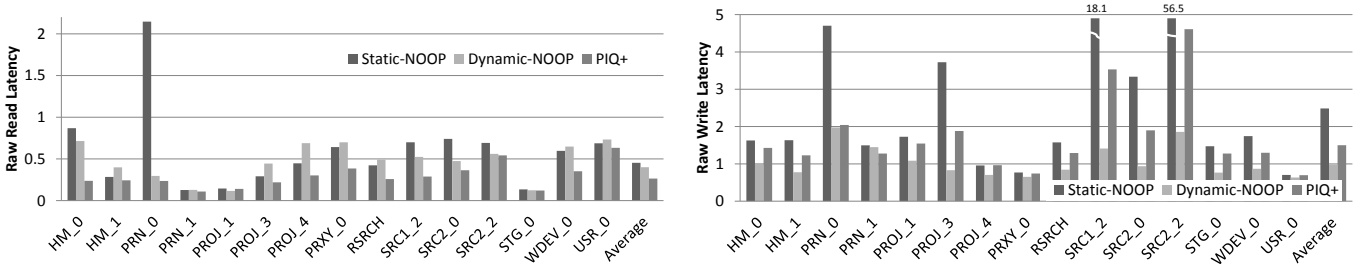
2) PIQ Evaluation at Plane Level: In this section, the design of PIQ at plane level is evaluated, which is proposed to exploit the internal parallelism of SSDs. The latency of read and write requests is measured as the metric to evidence the efficiency of proposed PIQ+ at plane level. The results are presented in Figure 14, where the PIQ+ at plane level is normalized to PIQ+ at chip level.

PIQ+_PLANE: PIQ+_PLANE represents the implementation of the PIQ+ approach at plane level. The PIQ+_CHIP presented in this section is the same as the above mentioned PIQ+ approach, which is implemented at chip level. As shown in Figure 14, PIQ_PLANE achieves slight performance improvement compared with PIQ_CHIP. For the read latency, the performance improvement from PIQ+_CHIP to PIQ+_PLANE is not significant, which is 1.8% on average. Figure 14(b) shows the write request evaluation results. The average write performance improvement is 4.6% from chip level to plane level. The biggest write performance improvement among these workloads exceeds 24.5%. In general, two observations can be concluded from these results:

- PIQ+_PLANE achieves better performance compared with PIQ+_CHIP on average;
- The performance improvement achieved by PIQ+_PLANE is small.

Firstly, in order to reveal the reasons of the improvement from the chip level to the plane level, normalized chip uti-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2693281, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



(a) Read Latency (b) Write Latency

Fig. 16: Performance Comparison among Static-NOOP, Dynamic-NOOP and PIQ+.

lization and plane utilization from chip level to plane level are collected, which are presented in Figure 15. There are 4 evaluation schemes.

- The first scheme is "PIQ+_CHIP_R", which represents the chip utilizations of PIQ+_R;
- The second scheme is "PIQ+_PLANE_R", which represents the plane utilizations of PIQ+_R;
- The third scheme is "PIQ+_CHIP_W", which represents the chip utilizations of PIQ+_W;
- The fourth scheme is "PIQ+_PLANE_W", which represents the plane utilizations of PIQ+_W.

In Figure 15, the parallel unit utilizations of "PIQ+_CHIP_R" and "PIQ+_CHIP_W" are used as the metric for "PIQ+_PLANE_R" and "PIQ+_PLANE_W", respectively. Then, we evaluate the impact of PIQ+_R and PIQ+_W on parallel unit utilization improvement from chip level to plane level. The evaluation results of Figure 15 have a matching pattern with Figure 14. In Figure 15(a), the average chip utilizations are improved slightly for most workloads. On average, the chip utilization improvement achieved by PIQ_R and PIQ_W are increased by 3.5% and 4.1%, respectively. In Figure 15(b), the average plane utilizations have little improvement for most workloads. However, there exist several workloads the utilizations of which are increased significantly. On average, the plane utilization improvement achieved by PIQ+_R and PIQ+_W are increased by 11.2% and 12.1%, respectively.

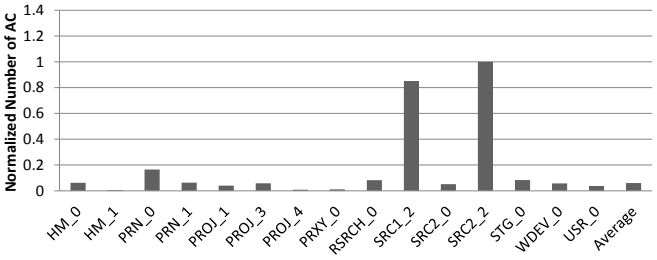


Fig. 17: Normalized Number of Requests Processed via Advanced Commands.

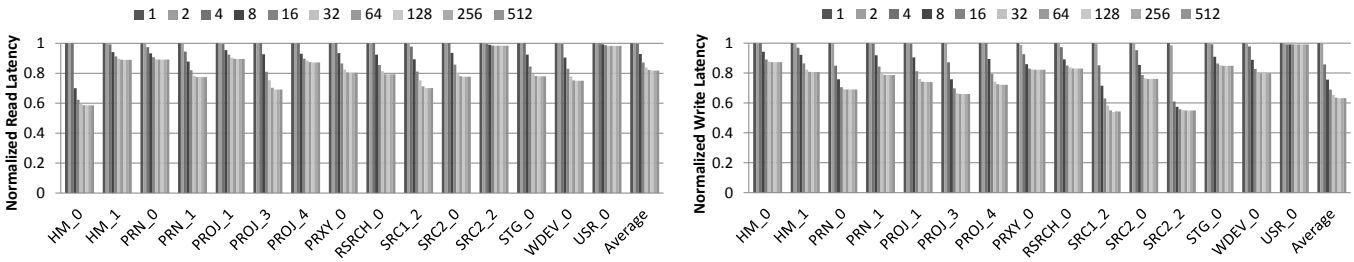
Secondly, the reason for the slight performance improvement of PIQ+_PLANE can be explained from two aspects. On the one hand, PIQ_PLANE is proposed to exploit the internal parallelism of SSDs. However, the internal parallelism is supported by advanced commands, which have special requirements for requests. In addition, the proposed PIQ+, which is implemented at host side, is hard to obtain the related information regarding advanced commands. Therefore, the batched requests based on the PIQ+_PLANE also can not fully exploit the internal parallelism and only can achieve limited performance improvement from chip level to plane level. In order to evidence this reason, Figure 17 measures the number of requests which are processed via applying advanced commands. Most workloads presented in Figure 17 can not make a full use of advanced commands even PIQ+ is implemented at plane level.

On the other hand, the internal parallelism of SSDs is further exploited by PIQ_PLANE, where more write requests are batched at plane level. That is, when more write requests are grouped into one batch due to the fine-grained access conflict detection, more write requests may be delayed and then suffer from longer waiting time. Therefore, it is possible that some workloads may achieve limited performance improvement compared with PIQ_CHIP.

3) Sensitive Study: In this section, we first vary the I/O queue length to show the effect of PIQ+. I/O queue length is varied from 1 to 512. In addition, the sensitive study at chip level is implemented as the example. Next, in order to achieve better performance of SSDs, the number of parallel units should be speculated based on the capacity of SSD. In this step, we vary the number of parallel units to show the efficiency of PIQ+.

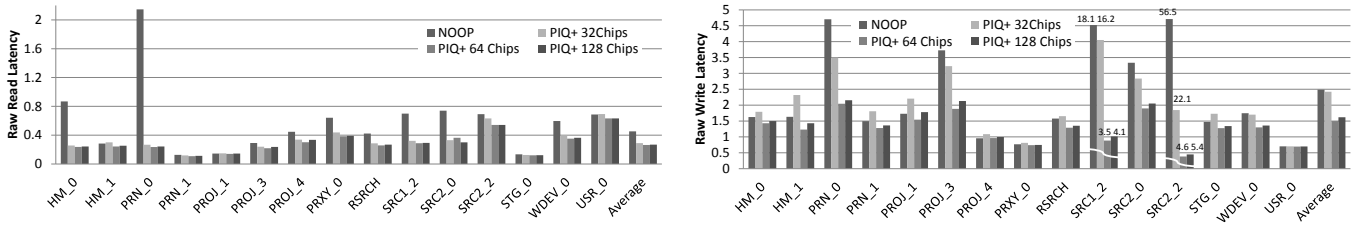
Figure 18 shows the experiment results on the read and write latency improvement with the varying of queue length. Several observations can be concluded from these experiment results. First, the increases of the length of I/O queue have significant improvement on the performance. For example, from 1 to 512, the read and write latency is improved by 19% and 37% on average, respectively. The reason is that with the increases of I/O queue length, more I/O requests can be processed in parallel. Second, when the depth of I/O queue is increased to a threshold, the performance improvement is diminished. For example, when queue length increases from 1 to 32, the read latency is reduced by 18%. However, when it is increased from 32 to 512, the read latency is only reduced by 1%. The write latency reduction has the similar characteristics. The reason is that when the queue length is large enough, the opportunities in the exploration of parallelism are diminished. Third, write latency has more benefit in the increases of queue length. As shown in Figure 18, the average read latency and write latency improvement is 19.7% and 47.0%, on average. The reason is that read requests are prone to access conflicts, especially when they conflict with write requests. Figure 19 shows the experiment results on the read and write latency with assuming 32 chips and 128 chips inside SSDs. In these

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2693281, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



(a) Read Latency (b) Write Latency

Fig. 18: Sensitive Studies with Varying Queue Length from 1 to 512.



(a) Read Latency (b) Write Latency

Fig. 19: Sensitive Studies with Varying Number of Flash Chips inside SSDs.

two figures, we can find that the performance improvements are also significant when access conflict detection computes the physical addresses of requests with inaccurate number of chips inside SSDs. In the meanwhile, we find that the speculated number of chips larger, the performance is better. This is because that larger speculated number of chips can group more parallel requests.

VI. Conclusions

In this paper, we have proposed an I/O scheduler, PIQ+, for NAND flash-based SSDs. PIQ+ is designed to reduce the access conflicts of SSDs through the exploration of the parallelism of SSDs. An efficient implementation of PIQ with negligible overhead is proposed. Experimental results show that PIQ+ achieves performance improvement by 41.1% and 39.7% for read and write latency, on average. In addition to the performance, lifetime is another key issue for SSD. In our future work, we will propose to combine PIQ with the recent lifetime enhancement schemes, such as the self-healing presented in the recent work [5] [37].

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *ATC*, pages 57–70, 2008.

[2] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. *SIGOPS Oper. Syst. Rev.*, pages 88–93, 2007.

[3] F. Chen, B. Hou, and R. Lee. Internal parallelism of flash memory-based solid-state drives. *ACM Transactions on Storage (TOS)*, 12(3):13, 2016.

[4] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *HPCA*, pages 266–277, 2011.

[5] R. Chen, Y. Wang, and Z. Shao. Dheating: Dispersed heating repair for self-healing nand flash memory. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, page 7. IEEE Press, 2013.

[6] Z. Chen, Y. Lu, N. Xiao, and F. Liu. A hybrid memory built by ssd and dram to support in-memory big data analytics. *Knowledge & Information Systems*, 41(2):335–354, 2014.

[7] Z. Chen, N. Xiao, and F. Liu. Sac: rethinking the cache replacement policy for ssd-based storage systems. In *SYSTOR*, pages 13:1–13:12, 2012.

[8] C. Dirik and B. Jacob. The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization. ISCA, pages 279–289, 2009.

[9] C. Gao, L. Shi, M. Zhao, C. Xue, K. Wu, and E.-M. Sha. Exploiting parallelism in i/o scheduling for access conflict minimization in flash-based solid state drives. In *MSST*, pages 1–11, 2014.

[10] A. Gupta, Y. Kim, and B. Urgaonkar. Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings. *Acm Sigplan Notices*, 44(3):229–240, 2009.

[11] B. He, J. X. Yu, and A. C. Zhou. Improving update-intensive workloads on flash disks through exploiting multi-chip parallelism. *Parallel and Distributed Systems, IEEE Transactions on*, 26(1):152–162, 2015.

[12] 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 Transactions on Computers*, 62(6):1141–1155, 2013.

[13] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang. Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity. In *ICS*, pages 96–107, 2011.

[14] M. Jung, W. Choi, S. Srikantaiah, J. Yoo, and M. T. Kandemir. Hios: A host interface i/o scheduler for solid state disks. *ACM SIGARCH Computer Architecture News*, 42(3):289–300, 2014.

[15] M. Jung and M. Kandemir. An evaluation of different page allocation strategies on high-speed ssds. In *FAST*, pages 9–9, 2012.

[16] M. Jung and M. Kandemir. Revisiting widely held ssd expectations and rethinking system-level implications. In *SIGMETRICS*, pages 203–216, 2013.

[17] M. Jung and M. T. Kandemir. Sprinkler: Maximizing resource utilization in many-chip solid state disks. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 524–535. IEEE, 2014.

[18] M. Jung, E. H. Wilson, III, and M. Kandemir. Physically addressed queueing (paq): improving parallelism in solid state disks. In *ISCA*, pages 404–415, 2012.

[19] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh. Disk schedulers for solid state drivers. In *EMSOFT*, pages 295–304, 2009.

[20] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar. Flashsim: A simulator for nand flash-based solid-state drives. In *Proceedings of the 2009 First International Conference on Advances in System Simulation*, pages 125–131, 2009.

[21] Y. Li, B. He, Q. Luo, and K. Yi. Tree indexing on flash disks. In *ICDE*, pages 1303–1306, 2009.

[22] D. Marcus and R. A. L. Narasimha. A new i/o scheduler for solid state devices. In *Technical Report TAMU-ECE-2009-02*, 2009.

[23] S. Mittal and J. S. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel & Distributed Systems*, pages 1–1, 2015.

[24] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating server storage to ssds: analysis of tradeoffs. In *EuroSys*, 2009.

[25] S. T. On, S. Gao, B. He, M. Wu, Q. Luo, and J. Xu. Fd-buffer: A cost-based adaptive buffer replacement algorithm for flashmemory devices. *IEEE Transactions on Computers*, 63(9):2288–2301, 2014.
[26] H. Park, S. Yoo, C. H. Hong, and C. Yoo. Storage sla guarantee with novel ssd i/o scheduler in virtualized data centers. *IEEE Transactions on Parallel & Distributed Systems*, pages 1–1, 2015.
[27] S. Park and K. Shen. Fios: A fair, efficient flash i/o scheduler. In *FAST*, pages 13–13, 2012.
[28] S. K. Park, Y. Park, G. Shim, and K. H. Park. Cave: Channel-aware buffer management scheme for solid state disk. In *SAC*, pages 346–353, 2011.
[29] S.-y. Park, E. Seo, J.-Y. Shin, S. Maeng, and J. Lee. Exploiting internal parallelism of flash-based ssds. *Computer Architecture Letters*, 9(1):9–12, 2010.
[30] J. Seol, H. Shim, J. Kim, and S. Maeng. A buffer replacement algorithm exploiting multi-chip parallelism in solid state disks. In *CASES*, pages 137–146, 2009.
[31] K. Shen and S. Park. Flashfq: A fair queueing i/o scheduler for flash-based ssds. In *ATC*, pages 67–78, 2013.
[32] 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 *ICS*, pages 338–349, 2009.
[33] A. S. Tanenbaum and A. Tannenbaum. *Modern operating systems*. Prentice hall Englewood Cliffs.
[34] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In *Proceedings of the Ninth European Conference on Computer Systems*, page 16. ACM, 2014.
[35] W. Wang and T. Xie. Pcftl: A plane-centric flash translation layer utilizing copy-back operations. *IEEE Transactions on Parallel & Distributed Systems*, pages 3420–3432, 2015.
[36] WIKIPEDIA. Open-channel SSD. https://en.wikipedia.org/wiki/Open-channel_SSD, 2015.
[37] Q. Wu, G. Dong, and T. Zhang. Exploiting heat-accelerated flash memory wear-out recovery to enable self-healing ssds. In *HotStorage*, 2011.
[38] T. Xie and Y. Sun. Dynamic data reallocation in hybrid disk arrays. *IEEE Transactions on Parallel & Distributed Systems*, 21(9):1330–1341, 2010.

Congming Gao received BS degree in Computer Science and Technology from Chongqing University in China in 2014. He is now a Ph.D candidate in the College of Computer Science, Chongqing University. His research interests include flash memory design, non-volatile memory and architecture optimizations.

Liang Shi received the B.S. degrees in Computer Science from Xi'an University of Post & Telecommunication, Xi'an, Shanxi, China, in July, 2008, Ph.D. degree from University of Science and Technology of China, Hefei, China, in June, 2013. He is now the associate professor in the College of Computer Science at the Chongqing University. His research interests include flash memory, embedded systems, and emerging non-volatile memory technology.

Cheng Ji received the BE degree from the School of Computer Science and Communication Engineering, Jiangsu University, China, in 2011, and ME degree from School of Computer Science and Technology, University of Science and Technology of China, China, in 2014, respectively. He is currently working toward the PhD degree in the Department of Computer Science at the City University of Hong Kong. His research interests include embedded systems, non-volatile memory, and hardware/software co-design.

Yejia Di received BS degree in Computer Science and Technology from Chongqing University in China in 2014. She is now a Ph.D candidate in the College of Computer Science, Chongqing University. Her research interests include embedded and real-time systems, flash memory and system optimizations.

Kaijie Wu received the BE degree from Xidian University, Xian, China, in 1996, the MS degree from the University of Science and Technology of China, Hefei, China, in 1999, and the Ph.D. degree in electrical engineering from Polytechnic University (Now Polytechnic Institute of New York University), Brooklyn , New York, in 2004. He then joined University of Illinois, Chicago, USA as an Assistant Professor. Since 2013, he becomes a professor at the College of Computer Science, Chongqing University, China. His research is on the big area of trustworthy computing with special interest on dependable computing and hardware security. He is the recipient of the 2004 EDAA Outstanding Dissertation Award for new directions in circuit and system test., and the Most Significant Paper award from the International Test Conference, 2014. He is a member of the IEEE.

Chun Jason Xue received BS degree in Computer Science and Engineering from University of Texas at Arlington in May 1997, and MS and PhD degree in computer Science from University of Texas at Dallas, in Dec 2002 and May 2007, respectively. He is now an Assistant Professor in the Department of Computer Science at the City University of Hong Kong. His research interests include memory and parallelism optimization for embedded systems, software/hardware co-design, real time systems and computer security.

Edwin H.-M. Sha (S'88-M'92-SM'04) received the Ph.D. degree from the Department of Computer Science, Princeton University, Princeton, NJ, in 1992. From August 1992 to August 2000, he was with the Department of Computer Science and Engineering at the University of Notre Dame, Notre Dame, IN. Since 2000, he has been a Tenured Full Professor at the Department of Computer Science at the University of Texas at Dallas, Richardson, TX. Since 2012, he has been serving as the Dean of the College of Computer Science at Chongqing University, Chongqing, China.