Exploiting Parallelism in I/O Scheduling for Access Conflict Minimization in Flash-based Solid State Drives

Congming Gao*, Liang Shi*, Mengying Zhao†, Chun Jason Xue†, Kaijie Wu*, and Edwin H.-M. Sha*
*College of Computer Science, Chongqing University, Chongqing, China
† Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong
Email: {albertgaocm, shi.liang.hk}@gmail.com, mengyzhao2-c@my.cityu.edu.hk,
jasonxue@cityu.edu.hk, {kaijie, edwinsha}@gmail.com

***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. Despite the rich parallelism offered by multiple channels and multiple chips per channel, recent studies show that the utilization of flash chips (i.e. the number of flash chips being accessed simultaneously) is seriously low. Our study shows that the low chip utilization is caused by the *access conflict* among I/O requests. In this work, we propose Parallel Issue Queuing (PIQ), a novel I/O scheduler at the host system, to minimize the access conflicts between I/O requests. The proposed PIQ schedules 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 SSD. And because PIQ is implemented at the host side, it can take advantage of rich resource at host system such as main memory and CPU, which makes the overhead negligible. Extensive experimental results show that PIQ delivers significant performance improvement to the applications that have heavy access conflicts..**

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 [**?**][**?**]. 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 [**?**][**?**]. Both the number of channels and the number of chips per channel are increasing in the last decade thanks to the advance in integration technologies. The rich parallelism offered by multiple channels and multiple chips per channel, however, has not been fully exploited. Figure 1 shows the evaluation result of chip utilization using a typical SSD with 8 channels and 8 chips per channel organization. 15 carefully selected MSR Cambridge traces from servers [**?**] are used in the experiments since they are always intensive applications. The detailed experiment settings are presented in Section V. As shown in Figure 1, the chip utilizations of the various benchmarks are mostly below 20%.

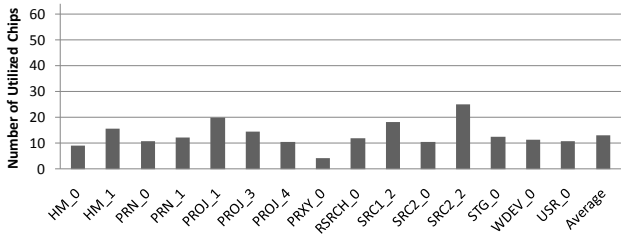
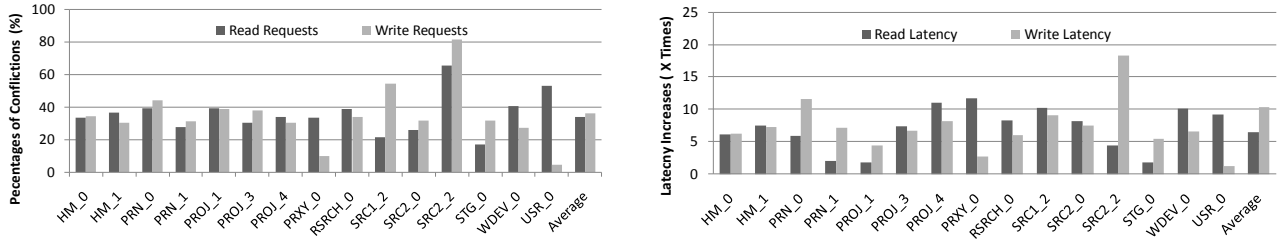


Figure 1. Chip utilization on a typical SSD with 8 channels and 8 chips per channel organization.

Recently many works have been proposed to improve the utilization of the rich parallelism from different perspectives. Jung et al. [**?**] and Hu et al. [**?**][**?**] show that data allocation schemes in the SSD controller are very important in exploiting the parallelism. Agrawal et al. [**?**] and Dirik et al. [**?**] evaluate different types of SSD organizations to investigate the impact on performance cast by the number of channels and the number of chips per channel. These works show that with a better SSD organization and/or controller design, the chip utilization can be improved significantly. Another important factor that prevents better utilization is the I/O access conflicts. The access conflicts among successive I/O requests naturally prevent the multiple chips in an SSD to be fully accessed in parallel. On recognizing the problem, Jung et al. [**?**] and Chen et al. [**?**] propose works to reduce read access conflicts. These works are implemented in the SSD controller, i.e., by moving the queue in the hardware interface and buffer cache in the controller to the beneath of FTL. However, these works only aim at improving read performance by considering only read and read conflicts, i.e., multiple read requests issued to a single chip in a short interval, while there are also read and write conflicts, and write and write conflicts.

In this work, we will propose a novel host side I/O scheduler, Parallel Issue Queuing (PIQ), to improve the exploitation of the rich parallelism in SSD. PIQ take into account all type of conflicts, i.e., read and write conflicts, read and read conflicts, and write and write conflicts. The

978-1-4799-5671-514$31.00



(a) Access Conflicts

(b) Performance Degradation

Figure 2. Access Conflict and Performance Impact: (a) The percentages of conflict read and write requests to the total I/O requests; (b) The performance degradation induced from access conflicts.

key challenge in designing PIQ is the detection of all these kinds of access conflicts among successive I/O requests from the host side. SSDs are different from traditional storage media, as there is a translation layer in the SSD controller that hides the data location information of I/O requests. In this work, a logical block number (LBN) based conflict detection approach is proposed to expose the conflict information to the host side. LBN represents the logical address of data used by the host systems, which is used in the determining of the location of data. With the knowledge of conflicts, the proposed PIQ separates I/O requests according to the conflicts between 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. The requests in the same batch can be fulfilled simultaneously by exploiting the rich parallelism of SSD. Also because PIQ is implemented at the host side, it can take advantage of rich resource available at the host system, such as the main memory and CPU of the system. This effectively makes the implementation overhead negligible.

To the best of our knowledge, this work is the first in proposing I/O scheduler at the host side to reduce access conflicts. Extensive experimental results show that PIQ is able to improve the access performance by 19% and 47% for read requests and write requests, respectively, compared to state-of-the-art techniques. This work achieves the following contributions:

- Verified that the various kinds of access conflicts at the host systems, including read and read, read and write, and write and write conflicts, are the key reason that prevents the rich parallelism from being fully exploited;
- Proposed an LBN based conflict detection approach to identify the access conflicts among successive I/O requests at the host side;
- Proposed a host-level conflicts-reduction I/O scheduler to improve both read and write performance;
- An efficient implementation of PIQ is presented with negligible overhead. Since PIQ is implemented at host system, it does not introduce any change to current SSD design. Experimental results show that PIQ is effective in reducing access conflicts and exploiting parallelism of SSDs for performance improvement.

The rest of the paper is organized as follows. Section II presents the problem in the parallelism exploration of SSDs. Section III presents the background and related works. In Section IV, access conflict detection approach and the access parallelism exploration I/O scheduler are proposed. Experiments are presented in Section V. Finally, Section VI summarizes the paper.

II. Problem Statement

Recent studies show that the rich parallelism of SSDs cannot be easily exploited, and the utilization of parallelism in SSDs is seriously low [**?**], [**?**]. One of the key issues for low chip utilization is the access conflicts among successive I/O requests. Access conflicts happen when several I/O requests access a same flash chip in a short time interval. These requests cannot be fulfilled simultaneously, which results in a low chip utilization. The types of conflicts include read and read conflicts, read and write conflicts, and write and write conflicts.

In order to reveal the impact of access conflict 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 2(a) shows the percentages of conflict read and write requests. It can be observed from Figure 2(a) that the access conflicts among requests commonly exist in applications. For example, for USR_0, 53.0% of read requests are conflicted, and for SRC2_2, 81.4% of write requests are conflicted. Secondly, we also observe that the percentages of conflicted write requests vary significantly among applications, while the percentages of conflicted read requests are roughly in the same range. On average, 35.2% of read requests, and 36.2% of write requests are conflict requests.

Conflict requests directly induce performance degradation. The performance degradation is presented in Figure 2(b), which has a matching pattern with Figure 2(a). The ratio of increased response time over the un-impacted response time is used as the metric. Similarly, the first observation shows that read and write response time are both increased,

and on average by 6.5 times and 10.1 times, respectively. The second observation shows that the latency increase, especially for write, varies significantly among applications. For example, there are 11.6 times increase in PRXY_0's read response time, and 18.3 times increase in SRC2_2's write response time, as well as only 1-3 times increase in PRN_1, PROJ_1, and STG_0's read response time, and PRXY_0, and USR_0's write response time. The reason behind the varying results is mainly due to different amount of access conflicts among the I/O requests of these applications. We can then conclude that access conflicts commonly exist in various applications running on SSDs, and the applications with stronger access conflicts will see more significant performance degradation.

III. Background and Related Work

In this section, NAND flash memory based SSD is presented with its organization and features. Then, the related works are presented, including parallelism exploration and the I/O scheduler for SSDs.

A. Solid State Drives

SSDs are constructed with NAND flash memory chips organized in channels, where an SSD controller is used to manage the storage array. In this section, the organization of SSDs is first presented. Then, the controller of SSDs is introduced.

1) Parallel Organization: An SSD is composed of multiple channels, and each channel is connected to multiple NAND flash memory chips. Each flash chip is further composed of multiple dies, each die is composed of multiple planes, and each plane contains multiple flash blocks and two register caches. Figure 3 shows an example of the organization of SSDs with 4 channels, 2 chips per channel, 2 dies per chip, and 2 planes per die. This is the four level parallel architecture of SSDs, which can be exploited for performance improvement. The relationship among these four levels have been well studied in previous works [**?**][**?**][**?**]. The die and plane level parallelism is called internal parallelism of SSDs, which is directly exploited by the SSD controller, and is hidden from up level system. In this work, the die and plane level parallelism of SSDs is not taken into consideration for two reasons: First, the die and plane level parallels require advance command support, which is not widely supported by most of SSDs. Second, the data location of SSDs is constantly changing between dies and planes in a flash chip by GC and WL [**?**][**?**][**?**].

2) Controller Design: The controller of SSDs is a key component in the design of SSDs, which needs to cover all issues of flash based SSDs. First, flash memory has many distinct characteristics, such as inability of in-place-update, limited lifetime, and slow write operations. All these issues are handled by the SSD controller. In addition, the management of the multiple chips organized in SSDs

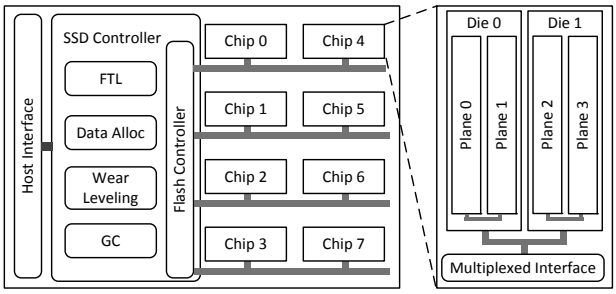


Figure 3. Architecture of SSD with four levels: Channels, Chips, Dies and Planes

is also implemented in the SSD controller. As shown in Figure 3, there are several components implemented in the SSD controller, including flash translation layer (FTL), data allocation (DA), garbage collection (GC), and wear leveling (WL).

Since flash memory is not able to conduct in-place-update, FTL is designed to maintain a mapping between logical address and physical address. When data are updated, the updated mapping is recorded for future accesses. 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. Previous works have evaluated different types of data allocation schemes, and found that the four levels of parallelism have different effects in the exploration of parallelism [**?**][**?**][**?**][**?**]. GC is used to collect invalid data during data updates since the updated data are written to a new place. The last component is wear leveling (WL), which is used to extend the lifetime of SSDs.

B. Related Work

1) Exploration of Parallelism: Since there are rich parallelism in the organization of SSDs, several proposals have been proposed to exploit the parallelism of SSDs [**?**][**?**][**?**][**?**][**?**][**?**]. Chen et al. [**?**] first evaluated and showed that the parallelism of SSDs is very important for performance improvement. They stated that with the introduction of parallelism, the performance of write operations had no relationship with their patterns (random/sequential), and even better than read operations. Based on this idea, Chen et al.[**?**] proposed to use SSDs as write buffers of hard disk drives. Jung et al.[**?**] and Hu et al. [**?**][**?**] showed that allocation schemes of SSDs were very important in the exploration of parallelism. They studied the differences among the four level parallelism and found that these four levels have different priorities in the exploration of access latency and system throughput. Seol et al. [**?**] and Park et al. [**?**] proposed to exploit the multi-channels of SSDs from the view of write buffers to improve the write performance. Both of them did not exploit the internal parallelism of SSDs. Jung et al.[**?**], Hu et al. [**?**][**?**], and Shin et al. [**?**] proposed to exploit different levels of parallels to improve the write

performance. Jung et al. [**?**] showed that random read performance of SSDs was the worst compared with other patterns. They inferred that bad random read performance is induced when a large number of read requests are conflict requests. However, none of these works consider access conflict reduction.

Jung et al. [**?**] proposed a read resource contention aware approach, physical address queuing (PAQ) under FTL, to issue more I/O requests to the SSDs. PAQ was implemented in the SSD controller, which required hardware modification. PAQ does not work for the conflicts at the host systems. Chen et al. [**?**] first proposed a buffer cache management approach for SSDs to solve the read conflict problem by exploiting the read parallelism of SSDs. They proposed to store more data for the most conflict chips in the buffer cache. However, these works are only able to solve the conflict problem when the conflicts have already taken place, which induce limited performance improvement. In addition, none of these approaches target conflict write requests. In this work, an I/O scheduler at host systems is proposed to exploit the rich parallelism of SSDs to reduce both conflict read and write requests before they are issued to SSDs.

2) Scheduler for Solid State Drives: Traditional I/O schedulers for HDD include NOOP, Deadline, Anticipate, and Completely Fair Queuing (CFQ) [**?**]. However, none of them work efficiently on SSDs [**?**]. Kim at el. [**?**] first proposed an I/O scheduler for SSDs with the awareness of read/write interferences. They proposed to bundle write requests and schedule read requests first to reduce the impact of slow write operations on read performance. Dunn et al. [**?**] proposed another write scheduler, which exploits the block locality to improve the write performance. This approach is able to improve the garbage collection efficiency since requests with data in the same blocks are issued together. Park et al. [**?**] and Shen et al. [**?**] proposed two schedulers to achieve fairness among multi-tasks on SSDs. FIOS [**?**] scheduled requests with the awareness of read and write interference of SSDs, and FlashFQ [**?**] scheduled requests with the awareness of the internal parallelism for different applications. However, none of these works are proposed to solve the access conflict problem.

Jung et al. [**?**] proposed the first work in read parallelism optimization. They have evaluated that random reads are the worst access pattern of SSDs. With the understanding that read parallelism can be exploited by the physical address of SSDs, a physical address queue based scheduler, PAQ, is proposed. However, PAQ requires hardware modification and only works in the SSD controller. In addition, they did not solve the write conflict issue. 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, i.e., read and read, read and write, and write and write conflicts. Since the proposed PIQ is implemented at the host side, it can take advantage of the rich resource available at the host system, such as main memory and CPU, thereby minimizing its implementation overhead. In addition, it can be used towards all existing SSDs as it does not require any change to be made in SSD design, which is a significant advantage over existing controller-level techniques.

IV. Parallel Issue Queuing for Parallelism Exploration

Motivated by low chip utilization of SSDs and high access conflicts, Parallel Issue Queuing (PIQ) in the host systems is proposed to reduce access conflicts and improve the performance of SSDs. Figure 4 shows the SSD based storage systems with PIQ implemented in the host systems, where FTL, DA, GC and WL are four most important components of an SSD controller, and multiple flash chips are equipped in multiple channels as the storage device. Host systems communicate with SSD devices through the host interface. Based on this organization, the key challenge in the design of PIQ is the detection of access conflicts. As shown in Figure 3, the internal architecture of SSDs is hidden by the SSD controller. The access conflict information is not exposed to the I/O scheduler. To solve this problem, a host side access conflict detection approach is first proposed, which is based on the understanding of the data allocation scheme of SSD controller and the logical address number of I/O requests. Then, based on the detected access conflicts, PIQ reduces the conflicts by smartly scheduling I/O requests. The basic idea of PIQ is to separate I/O requests with conflicts into different scheduling batches based on the detected access conflicts. Then, the requests in each batch can be processed in parallel. Finally, an efficient implementation of PIQ is presented in this section.

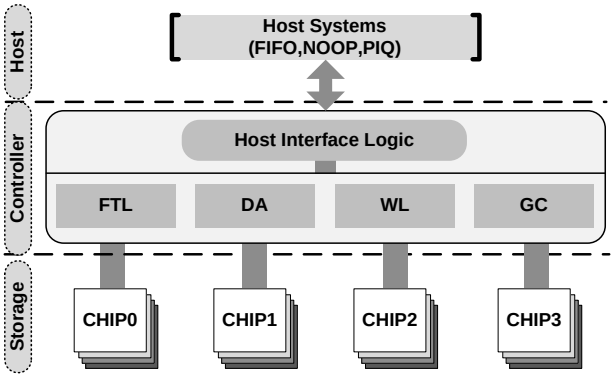


Figure 4. Organization of SSD based storage systems, where PIQ is implemented in host systems.

A. Access Conflict Detection

Access conflict is highly correlated with the issue time of I/O requests and location of data. If the issue time interval between requests is small enough, and the accessed data exists in the same flash chip, these accesses will conflict. In

order to detect access conflict, the location of data should be first determined. The basic idea in the determining of data location is to construct the relationship between I/O request's logical block number (LBN) and data locations in the SSDs. Note that the location of the data indicates the flash chip number, where the accessed data is stored. Even though WL and GC will constantly move data among SSDs, most of the state-of-art WL and GC approaches only move data within a chip [**?**][**?**][**?**]. In this case, WL or GC does not impact the location of data at the chip level. Based on this observation, we propose an access conflict detection approach.

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 scheduling queue have the potential in access conflict. Based on this observation, outstanding I/O requests and waiting requests in the I/O scheduling 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 LBN. Recently, data allocation scheme has been widely studied [**?**][**?**][**?**][**?**], where location of the data is determined based on LBN. If LBN is given, the location of data can be computed based on the 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 the number of flash chips in the SSDs. Initially, location vector is set with all zeros. When data are stored in one flash chip, the corresponding bits of a location vector of a request are set. We use the following settings to generate the location vector:

- The number of chips in the SSDs is N_C and the size of flash page is $Page\_Size$ in bytes;
- I/O requests are defined with $(\mathrm{R}_i, LBN, Sector\_Number, T)$, where R_i is the request type, such as read or write requests, LBN is the logical block number that the request is to access, $Sector\_Number$ is the accessed data size in term of the number of sectors, SS is the sector size in bytes, and T is the time when the I/O request is added to the I/O scheduler queue.

Based on these settings, the set bits of the location vector of a request in the SSDs can be determined, which are from $chip\_first$ to $chip\_last$, as follows.

$$chip\_first = (\lfloor \frac{LBN \times SS}{Page\_Size} \rfloor) \% N_C;$$
$$chip\_last = (\lceil \frac{(LBN + Sector\_Number) \times SS}{Page\_Size} \rceil - 1) \% N_C; \quad (1)$$

Then, based on $chip\_first$, $chip\_last$, and N_C, the location vector is generated. Let's illustrate with an example. Assuming N_C=8, SS=4KB, a read request is $(\mathrm{R}_i, LBN, Sector\_Number, T) = (r, 14854, 2, 0.02332)$, and BS=4KB, we can derive that, $chip\_first$ = 6 and $chip\_last$ = 7, which means that data of the read request are stored in two chips, Chip 6 and Chip 7. Then the location vector of this request, $V(\mathrm{R}_i)$, is (00000011) = 3. Note that after GC, WL, and data updates, the location vector remains the same because these operations only take place within a chip.

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 the data of two requests in the outstanding and waiting queue are located in the same flash chip, these two requests are determined 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:

$$Conflict = V(R_i) \& V(R_j); \quad (2)$$

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

Figure 5 shows an example of conflict detection. As shown in Figure 5, an SSD is constructed with four channels, where each channel is equipped with 2 chips. The data allocation scheme is channel first and chip second [**?**]. In the pending queue, there are five I/O requests. Based on the allocation scheme and I/O request's LBN, the location vectors of each request are computed and recorded. Then, based on the conflict detection of Formula 2, we can identify the conflict requests. For example, requests 1 and 3 conflict at Chip 5 (12 & 6=4), requests 2 and 4 conflict at Chip 1 (64 & 96=64), and requests 3 and 5 conflict at Chip 6 (6 & 3=2).

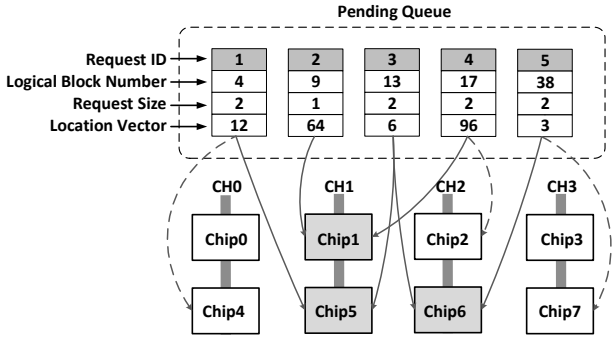


Figure 5. Access Conflict of I/O Requests.

B. Parallel Issue Queuing (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 there is no conflict in each batch.

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, 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. *Batch location vectors* is used to record the data locations of all the requests in each batch. Initially, batch location vector is set to 0. When one request is added to the pending queue, the conflict is detected by checking the batch vector with the entering I/O request. 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. Since PIQ takes outstanding requests and waiting requests in the scheduling queue into account, when new requests are added to the scheduling 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. Based on the proposed approach, request fairness may be a problem since a request can be delayed for a long time. The worst case happens when one request is over-passed by the other requests which do not conflict with the outstanding requests. However, since only the requests in the I/O queue are processed, and the requests are processed based on the creation order of batches, the fairness problem has little impact to the performance. Note that dependencies between I/O requests are avoided when they are added to the scheduling queue using traditional approaches. All requests processed in the I/O queue have no dependency with each other.

Figures 6 and 7 show examples of the comparison of NOOP and PIQ. In this example, we assume there are six I/O requests in the pending queue (three read requests (R0, R1, R2) and three write requests (W0, W1, W2)), with each accessing 2 flash pages. These six I/O requests are added to the pending queue in the First In First Out (FIFO) order, from left to right. The SSD is organized with four flash chips with one chip per channel. The data location of the six I/O requests are marked in the four chips 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. We find that all six I/O requests conflict. 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 pending 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. In this case, read and write requests are separated 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, and the batch vector B_1 is initialized with 6. 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 conflict. Then, R_2 is added to B_0, and location vector of B_0 is updated based on Formula 4 to 15. Finally, PIQ issues the I/O requests in batches, where all the batches are processed in the creation order. We find that with PIQ, only two read and two write cycles are required to service these requests. In order to avoid the starvation of write requests, once the read batches in the current queue are serviced, the write batches are serviced.

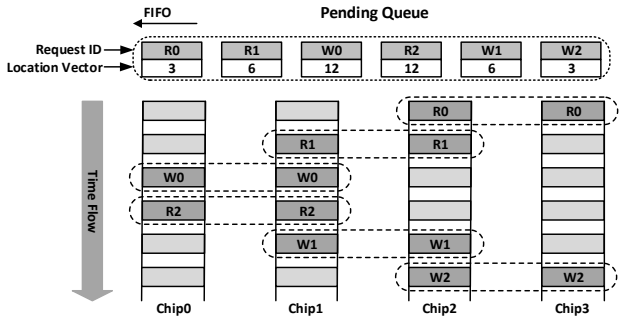


Figure 6. NOOP: requests are issued in FIFO order.

C. Implementation

In this section, we present the implementation of PIQ in I/O scheduler. Two mechanisms are implemented, conflict detection and batch creation. To do so, data location vector,

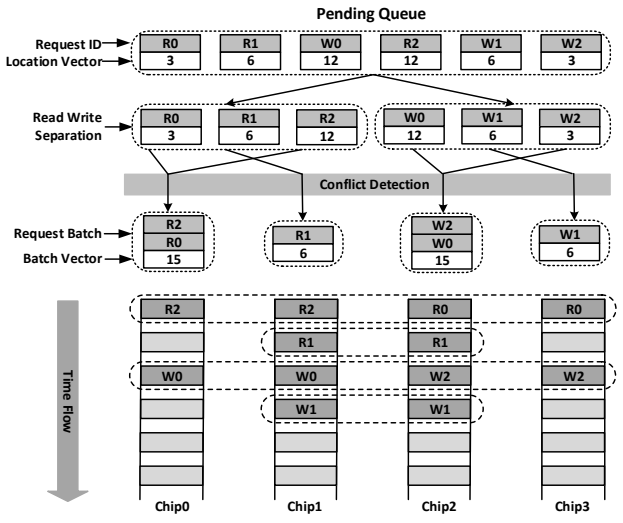


Figure 7. PIQ: RWS, RRS, and WWS are applied to the queue to separate the I/O requests into different batches based on the detected conflicts.

batch location vector and batch list are maintained. Data location vector is created based on the location of the data as shown in Section IV-A. Batch location vector is used to record the data location of all requests in a batch, which is created based on the separating schemes as shown in Section IV-B. For the batch list, it is used to maintain the batches, where all requests in a batch can be serviced in parallel, and batches are issued in the order of creation time.

Algorithm 1 shows the implementation of PIQ. Conflict detection is implemented by applying the location vector, which indicates the data location of the request. As shown in Algorithm 1, once an I/O request is added to the scheduling queue, $V(Req)$ is produced based on Formula 1 (Line 1). Then, based on the types of the requests, they are added to different batch lists, read batch list or write batch list (Lines 2-6). This step is implemented to realize the read and write request separations. Then, batches are created based on conflict detection. Initially, the batch list is empty. In this case, a new batch is created, and the batch vector is initialized (Lines 18-22). When the batch list is not empty, conflict is checked between the batch vector and location vector of the entering I/O request (Line 9). If there is no conflict, the I/O request is added to the batch, and the batch vector is updated (Lines 10-14). Otherwise, the next batch is checked (Line 15). If the I/O request conflicts with all the batches in the batch list, a new batch is created, and its location vector is initialized (Line 18-22).

Overhead analysis: The implementation of PIQ needs to maintain batch location vectors and a batch list in the I/O queue. The batch location vector size is determined by the number of flash chips in the SSDs. In this case, the size of each batch location vector is N_C bits. We assume that the queue length of I/O scheduler is N_{piq}, then the maximal size of storage required by the batch location vectors is $N_{piq} \times N_C$ bits. This storage overhead is negligible for an I/O queue. In addition, there are two batch lists maintained in the I/O queue, which has negligible cost. For the computation overhead of PIQ, there are only `MOD`, `OR` and `AND` operations, which also have negligible cost.

Algorithm 1 PIQ: Parallel Issue Queuing

Input:
Req – I/O Request;

1: $V(Req)$ is produced by Formula 1;
2: **if** Req == $READ$ **then**
3: $Batch\_List$ = $Read\_Batch\_List$;
4: **else**
5: $Batch\_List$ = $Write\_Batch\_List$;
6: **end if**
7: $Batch$ = $Batch\_List.Head$;
8: **while** $Batch$ **do**
9: $Conflict$ = $V(Batch)$ & $V(Req)$;
10: **if** $Conflict$ == 0 **then**
11: Add_to_Batch($Batch$, Req);
12: $V(Batch) = V(Batch) \mid V(Req)$;
13: Break;
14: **else**
15: $Batch$ = $Batch.next$;
16: **end if**
17: **end while**
18: **if** $Batch$ == `NULL` **then**
19: Create_New_Batch($Batch$, Req);
20: $V(Batch) = V(Req)$;
21: Add_to_Batch_List($Batch\_List$, $Batch$);
22: **end if**
23: Issue $Batch\_List.Head$;

V. Experiment and Analysis

In this section, we first present the experimental methodology. Then, the experimental results of the proposed approach are presented with the analysis of performance, access conflict relaxation, and the improved chip utilization of SSDs.

For validation, we implemented five related schemes to show the effectiveness of the proposed approaches.

- The first scheme is "NOOP", which represents the traditional I/O scheduler implemented in operating systems. NOOP is proposed to service requests in FIFO order, where dependencies between requests have already resolved;
- The second scheme is "RWS", which is implemented based on NOOP, where RWC are detected and applied in the separation of read and write requests. Read requests are processed with a higher priority. Note that in order to avoid the starvation of write requests, only pending read requests have higher priorities than write

requests. This scheme is similar to the approaches in [**?**][**?**];

- The third scheme is "PIQ_R", which is implemented over "RWS", where RRC are detected and applied with the separation of read requests into batches. This scheme is implemented to show the effect of the proposed approach on the read performance;
- The fourth scheme is "PIQ_W", which is implemented over "RWS", where WWC are detected and applied with the separation of write requests into batches. This scheme is implemented to show the effect of the proposed approach on the write performance;
- The fifth scheme is "PIQ", which is implemented with "PIQ_R" and "PIQ_W", where RWC, RRC, and WWC are detected and applied with the separation of I/O requests into batches.

Among these five schemes, "NOOP" and "RWS" are the most related works with PIQ implemented in the I/O scheduling queue of host systems. Note that FlashFQ and FIOS are not compared with the proposed work since they are proposed to achieve fairness between applications, which will be our further work.

A. Experiment Setup

In this paper, we use a trace drive simulator, SSDsim [**?**][**?**], to verify the proposed framework. SSDsim has been widely applied in the exploration of parallelism of SSDs. In this work, the proposed approach is implemented in the I/O queue of the host systems to schedule I/O requests.

In this study, we modeled a 128GB SSD, which is configured with 8 channels with each channel equipped with 8 chips. Each flash block consists of 64 pages with a page size of 2KB. Page mapping scheme is implemented as the default FTL mapping scheme. 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 [**?**][**?**][**?**]. For the data allocation scheme, the most widely used channel first, chip second scheme is applied since they have the best performance in term of the parallelism of SSD [**?**]. We use 64 as the default queue length for I/O scheduling queue. The experiment settings represent a typical modern SSD. In the experiment results, we vary the length of I/O queue to gain further insight into how the proposed approach behaves under various interfaces and SSDs.

The workloads studied in this work include a set of carefully selected MSR Cambridge traces from servers [**?**] based on the identified conflicts. These traces are widely used in previous works in the studies of SSD performance [**?**][**?**][**?**]. Table I shows the characteristics of traces evaluated in the experiments, where the conflicts and utilized chips are collected with NOOP as the I/O scheduler. Read I/Os and Write I/Os represent the number of read and write requests in the traces. Totally, the number of read and write I/Os are 500000, which is a common number used in previous work. In addition, there are other 500000 I/Os for warming up the device. Read Conflicts and Write Conflicts represent the number of read and write requests enrolled in conflict. The last collum represents the average number of utilized chips for different applications. Initially, the SSD is warmed up with all the data used by the traces and the other space is filled up with random data.

B. Experimental Results

1) Performance Evaluation of Basic PIQ: In this section, we present the performance comparison among NOOP, RWS, PIQ_R, PIQ_W, and PIQ at chip level. And all the schemes are processed with the basic design without batch scheduler. In Figure 8, Y-axis represents the average execution latency of read and write requests normalized to NOOP.

RWS: As shown in Figure 8, RWS, which prioritizes the process of read requests, has little performance improvement for most applications. In order to understand the reason for performance improvement, the percentages of prioritized read requests are presented in Figure 9. As shown in Figure 9, the percentages of prioritized read requests for most applications are within 5%. However, for HM_0, PRXY_0, and SRC1_2, there are more than 5% of read requests prioritized, which induces improved read performance. The prioritized read requests would induce slight write performance degradation as shown in Figure 8(b).

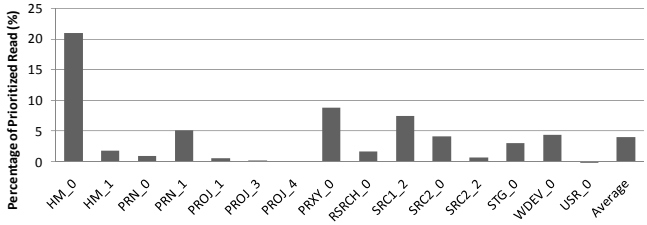
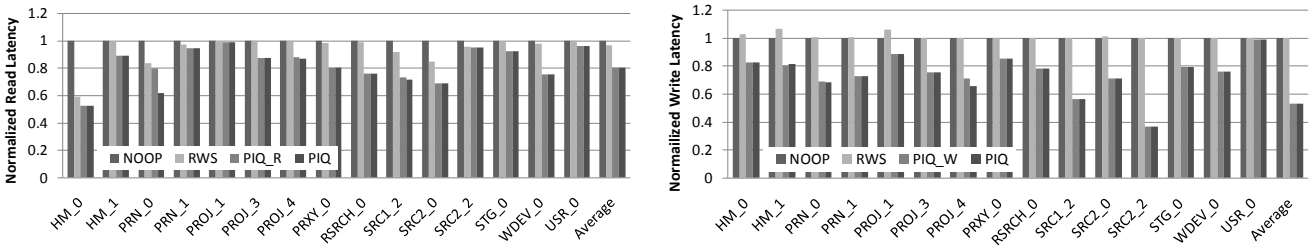


Figure 9. Percentages of prioritized read requests comparison between NOOP and RWS.

PIQ_R: PIQ_R is proposed to detect the conflict read requests and exploit the read parallelism for performance improvement. As shown in Figure 8(a), the read latency is reduced significantly for most applications (13 out of 15). For example, for SRC2_0, the read latency is reduced by 31.2%. In order to reveal the reason for the improvement of read performance, normalized average request waiting time for read requests induced by access conflicts is collected and shown in Figure 10(a). Read request waiting time is produced when a read request conflicts with the requests ahead of it. As shown Figure 10(a), the average request waiting time induced by access conflicts of SRC2_0 are reduced by 37.1%. However, we also find that PROJ_1 has little read performance improvement. As shown in Figure 10(a), we find that the average read request waiting time of

Table I
TRACE CHARACTERISTICS.

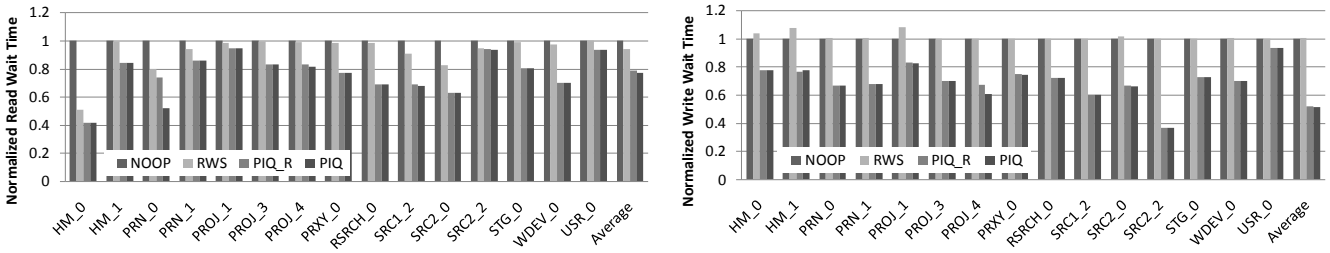
Applications	Read I/Os	Write I/Os	Read Conflicts	Write Conflicts	Chips Utilizations
HM_0	129066	370933	43512	128117	9.076
HM_1	484454	15445	178104	4690	15.592
PRN_0	75305	424694	29616	188458	10.698
PRN_1	346001	153995	95643	47896	12.221
PROJ_1	486780	13219	191133	5138	19.892
PROJ_3	419058	80930	127710	30810	14.349
PROJ_4	465194	34806	157480	10554	10.543
PRXY_0	15385	484614	5140	48992	4.179
RSRCH_0	49400	450596	19112	153581	11.76
SRC1_2	71500	428498	15334	232312	18.269
SRC2_0	97924	402069	25343	128229	10.38
SRC2_2	19469	480530	12765	391383	24.965
STG_0	197403	302594	33406	197403	12.342
WDEV_0	99000	400992	40044	108946	11.222
USR_0	148822	351165	78940	15450	10.733



(a) Read Latency Comparison

(b) Write Latency Comparison

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



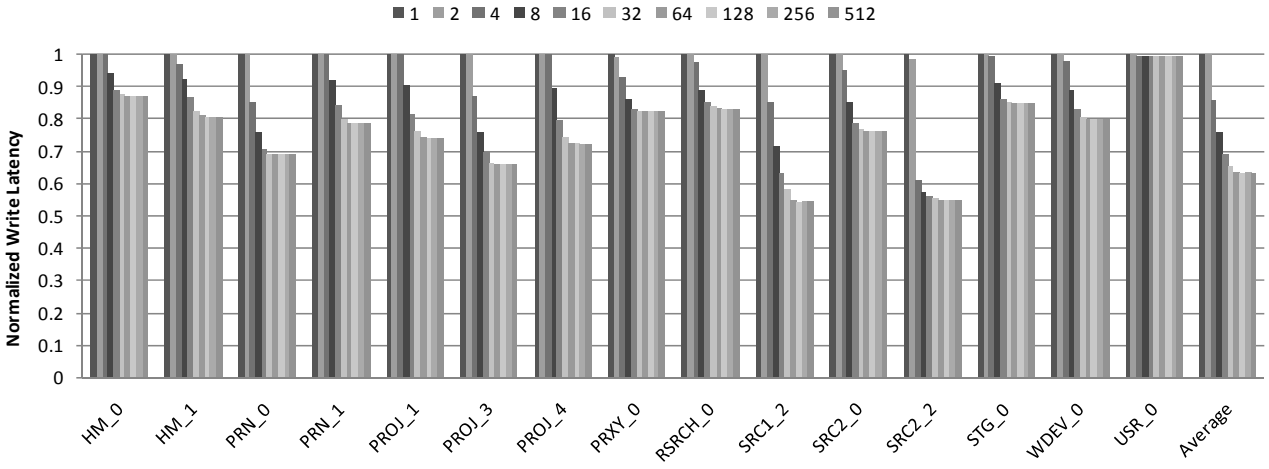
(a) Normalized Average Read Request Waiting Time

(b) Normalized Average Write Request Waiting Time

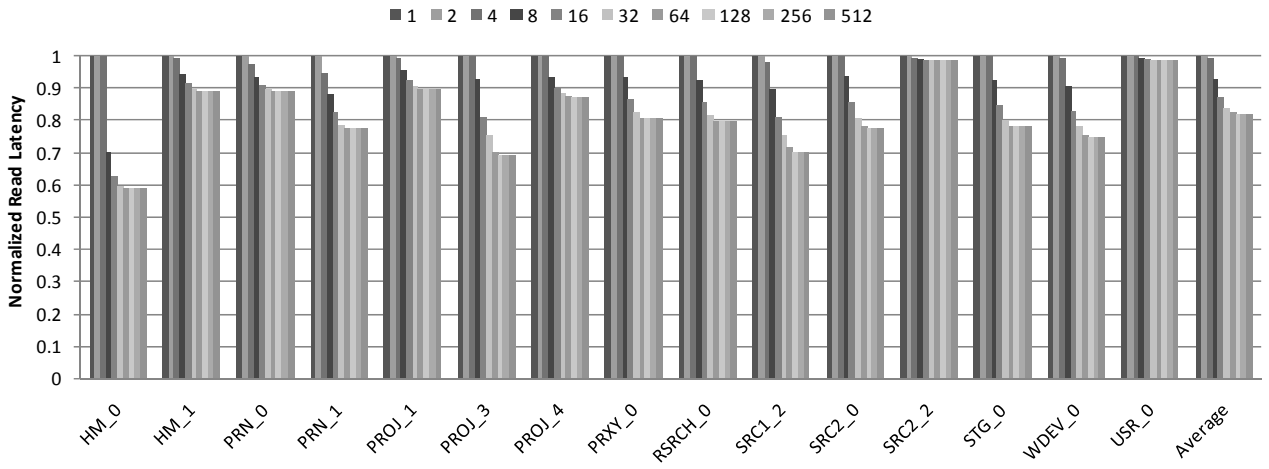
Figure 10. Average request waiting time comparison among NOOP, RWS, PIQ_R, PIQ_W, and PIQ.

PROJ_1 is reduced slightly. In Figure 12, the chip utilization of SSDs is presented to show the reason of read performance improvement. We can find that PROJ_1 achieves little chip utilization improvement with PIQ_R. The reason is that the conflict read requests of PROJ_1 cannot be processed in parallel. On average, the read latency, the average request waiting time and chip utilization are improved by 19.6%, 21.0%, and 14.7%, respectively. Based on these results, we conclude that PIQ_R works for read intensive applications with a large number of conflict read requests, and the conflict read requests can be processed in parallel.

PIQ_W: PIQ_W is proposed to detect the conflict write requests and exploit the write parallelism for performance improvement. As shown in Figure 8(b), the write latency is reduced significantly for most applications (13 out of 15). For example, for SRC1_2, the write latency is reduced by 43.5%. In order to reveal the reason for the improvement of write performance, normalized average request waiting time for write requests is collected and shown in Figure 10(b). Write request waiting time is induced when a write request conflicts with other requests ahead of it. As shown Figure 10(b), the average request waiting time induced by write conflicts of SRC1_2 is reduced by 39.6%. However, we also find that USR_0 has little write performance improvement. The reason for USR_0 can be found from Figure 10(b), where the average write request waiting time is reduced slightly. In Figure 12, the chip utilization of SSDs is presented to show the reason of write performance



(a) Read Latency



(b) Write Latency

Figure 11. Sensitive Studies with Varying Queue Length from 1 to 512.

improvement. On average, the write latency, the average request waiting time and chip utilization are improved by 47.0%, 48.4%, and 11.9%, respectively. Based on these results, we conclude that PIQ_W works for write intensive applications with a large number of write conflicts, and the conflict 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 achieves read and write performance improvement similar to PIQ_R and PIQ_W, respectively. The read latency is reduced by 19.7%, the write latency is reduced by 47.0%, the average read request waiting time are reduced by 22.5%, the average write request waiting time are reduced by 48.4%, and the chip utilization is increased by 24.2%, compared with RWS, on average. Based on these results, we conclude that PIQ works effectively for I/O intensive applications and shows significant performance improvement.

2) Sensitivty Studies: In this section, we vary the I/O queue length to show the effect of basic PIQ. I/O scheduling queue length is varied from 1 to 512. The number of chips and channels are configured with 8 chips per channel and 8 channels connected to the SSD controller.

Figure 11 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 scheduling 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

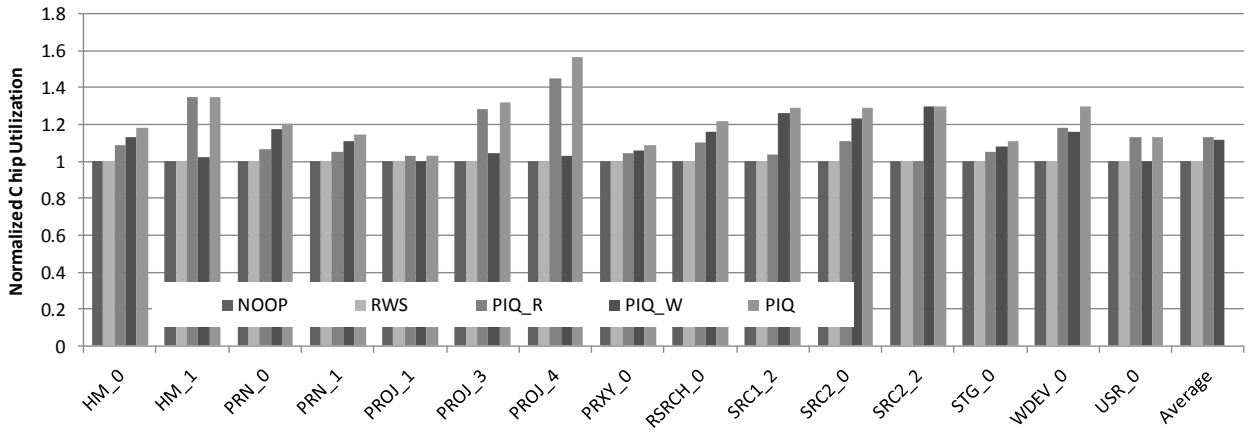


Figure 12. Normalized chip utilization of NOOP, RWS, PIQ_R, PIQ_W, and PIQ.

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 11, 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.

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. Unlike previous works that focus on the studies of the parallelism of SSDs or separation of read and write requests to reduce the read and write interferences, PIQ is implemented in the host systems. It takes all three types of access conflicts into consideration and exploits the parallelism of SSDs to reduce the access conflicts. This is accomplished through the access conflict detection and requests separation. An efficient implementation of PIQ with negligible overhead is proposed. Experimental results show that PIQ achieves performance improvement by 15.6% and 28.6% for read and write latency, on average.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *ATC'08*, 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 and D. Koufaty. Hystor: Making the best use of solid state drives in high performance storage systems. In *ICS'11*, 2011.

[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'11*, pages 266–277, 2011.

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

[6] 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'09, pages 279–289, 2009.

[7] 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.

[8] 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'11*, pages 96–107, 2011.

[9] M. Jung and M. Kandemir. An evaluation of different page allocation strategies on high-speed ssds. In *FAST'12*, pages 9–9, 2012.

[10] M. Jung and M. Kandemir. Revisiting widely held ssd expectations and rethinking system-level implications. In *SIGMETRICS'13*, pages 203–216, 2013.

[11] M. Jung, E. H. Wilson, III, and M. Kandemir. Physically addressed queueing (paq): improving parallelism in solid state disks. In *ISCA'12*, pages 404–415, 2012.

[12] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh. Disk schedulers for solid state drivers. In *EMSOFT'09*, pages 295–304, 2009.

[13] D. Marcus and R. A. L. Narasimha. A new i/o scheduler for solid state devices. In *Technical Report TAMU-ECE-2009- 02, Dept. of Electrical and Computer Engineering*, Texas A&M Univ., Apr. 2009.

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

[15] C. 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.

[16] S. Park and K. Shen. Fios: A fair, efficient flash i/o scheduler. In *FAST'12*, pages 13–13, 2012.

[17] S. K. Park, Y. Park, G. Shim, and K. H. Park. Cave: Channel-aware buffer management scheme for solid state disk. In *SAC'11*, pages 346–353, 2011.

[18] J. Seol, H. Shim, J. Kim, and S. Maeng. A buffer replacement algorithm exploiting multi-chip parallelism in solid state disks. In *CASES'09*, pages 137–146, 2009.

[19] K. Shen and S. Park. Flashfq: A fair queueing i/o scheduler for flash-based ssds. In *ATC'13*, pages 67–78, 2013.

[20] 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'09*, pages 338–349, 2009.

[21] A. S. Tanenbaum and A. Tannenbaum. *Modern operating systems*, volume 2. Prentice hall Englewood Cliffs, 1992.