Journal of Systems Architecture 98 (2019) 434–442

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Delay-based I/O request scheduling in SSDs

Renhai Chen[a], Qiming Guan[a], Chenlin Ma[b,*], Zhiyong Feng[a]

[a] College of Intelligence and Computing, Shenzhen Research Institute of Tianjin University, Tianjin University, China
[b] Department of Computing, Hong Kong Polytechnic University, Hong Kong

A R T I C L E I N F O

Keywords:
Request scheduling
NAND flash memory
Performance prediction

A B S T R A C T

To pave the way for fast data access, multiple parallel components (e.g., multiple dies) in SSDs are crafted. Therefore, how to fully utilize the parallel resources has become a challenging issue. To address this issue, we first design an SSD model, which not only considers the structure of the parallel components in SSDs but also investigates the utilization of these components. Then, a novel delay-based request scheduling strategy is devised based on the proposed model. The proposed scheduling strategy can predict the resource utilization and intelligently allocate requests to the parallel components (i.e., channels) to achieve the low storage access latency. We integrate the delay-based request scheduling with the proposed SSD model into a trace-driven simulator and evaluate the effectiveness of the proposed scheme on the Financial and WebSearch data sets. The experimental results show that the IOPS and request latency are improved by 16.5% and 14.2% respectively compared with the state-of-the-art scheduling strategy.

1. Introduction

Flash-based SSDs have been gaining ever-increasing prevalence in electronic devices due to its attractive features, such as low standby power, high density, and shock resistance [1–14]. Although modern SSDs use high-performance interfaces such as 16 GB/s PCIe interface [15,16], the speed of NAND flash [17–19] (the storage media inside SSDs) is limited, such as 40 MB/s[20]. To bridge this huge performance gap, multiple parallel components are designed to provide fast data access and how to well utilize these parallel components has become a challenging issue.

Several prior arts have investigated how to exploit the parallel resources in SSDs. Chen et al. studied the system-level parallelism, focusing on a multi-channel and multi-way architecture [21]. Parallel-request scheme [22] and Micron-style interleaving [23] were proposed to improve this system-level parallelism. In contrast, Hu et al. [24] studied advanced flash operations and the interplay of such operations in order to take advantage of flash-level parallelism. Channel interleaving [25], plane-pairs [26], and the multi-plane flash operation [24] were proposed to maximize this flash-level parallelism. However, these studies neglect the different access features of read/write requests and simply use the round-robin [27] approach to evenly distribute read/write requests to different parallel components, leading to the under-utilization of available bandwidth in SSDs. To address this issue, a weight-based scheduling [28] was proposed to assign different weights to read/write requests. However, this work uses a fixed weight for each read/write request without considering the status of the accessed parallel components.

In this paper, we build an SSD model, which can predict the utilization of parallel components in SSDs. To build this model, the SSD's characteristics that affect its service time are abstracted into a set of parameters. The parameters include the number of channels, the number of flash chips in each channel, the data access time in the bus and etc. A statistical linear model is exploited to depict the organization and resource utilization in SSDs. After obtaining this model, a delay-based scheduling policy is elaborated to predict the resource utilization and reschedule requests to fully utilize the parallel resources in SSDs. The delay denotes the waiting time for a request to be served. Specifically, the delay is the time period from the request arrival to the request execution. In detail, the delay-based scheduling strategy dynamically uses the proposed model to calculate the loads on each channel and schedules the requests to the channel with the minimal burden. Therefore, compared with the weight-based scheduling, the delay-based scheduling takes the device status as an important factor to perform request scheduling, thus generating the more balanced request dispatching among the channels.

We implement the proposed model and delay-based request scheduling policy in a trace-driven simulator. In the experiment, we use 16 channels and 8 flash chips per channel (128 flash chips in total) to emulate an SSD device. Micron 16 Gb multi-level cell (MLC) NAND flash [29] is used as the simulated flash device. We evaluate the effectiveness of the proposed scheme on the Financial and WebSearch data sets. The experimental results show that the IOPS (Input/Output Operations Per

* Corresponding author.
E-mail addresses: guanqiming@tju.edu.cn (Q. Guan), cscma@comp.polyu.edu.hk (C. Ma), zyfeng@tju.edu.cn (Z. Feng).

https://doi.org/10.1016/j.sysarc.2019.05.001
Received 20 November 2018; Received in revised form 12 April 2019; Accepted 1 May 2019
Available online 1 May 2019

Table 1
Definition of notations used in Sections 3 and 4.

Notation	Definition
T_{bus}	The time of transmitting one byte on the data bus
S_{page}	The physical page size in SSDs
$T_{commands}$	The time of transmitting commands on the data bus
$N_{request\_batch}$	The number of handling read/write requests at a time
T_{data}	The time of transmitting a physical page on the data bus
$T_{channel\_w}$	The time at which write data is transmitted on the channel
$T_{channel\_r\_commands}$	The time at which the read request commands are transmitted on the channel
$T_{channel\_r\_data}$	The time of transmitting multiple physical pages from flash to the SSD controller over the channel
T_R	The time of reading the physical page to the cache register
T_{PROG}	The page program (write) time from register
T_{flash_w}	The time of handling write requests in the flash
T_{flash_r}	The time of handling read requests in the flash
$T_{request\_arrive}$	The time at which the request arrived the SSD controller
$N_{commands}$	The number of commands issued to the flash channel for one request
$T_{predict\_request\_end}$	The time point of finishing handling the batch requests.
$T_{request\_time}$	The request handling time
$delay_i$	The delay of the ith batch requests
$queueTotalDelay_n$	The total delay of the nth channel
$T^i_{requests}$	The execute time of the ith batch requests
$Latency^i_n$	The total latency of the ith batch requests in the nth channel
$T_{channel-contention}$	The time at which spent on resolving the channel-level contention
$T_{channel-idle}$	The time at which spent on resolving the channel-level idle
$T^m_{datamovement}$	The time at which the mth request data is transmitted on the channel
$Latency^m_{way}$	The latency of the flash memory for the mth request
$Latency^m_{memory}$	The time of the flash memory handling read/write for the mth request
$Latency^m_{memory-busy}$	The time of the device busy with the request address

Second) and request latency of the SSD equipped with the proposed delay-based request scheduling strategy are improved by 16.5% and 14.2% respectively compared with the state-of-the-art scheduling strategy.

The main contributions are summarized as follows:

- We devise a delay-based request scheduling strategy to well utilize the parallel resources in SSDs.
- In order to better use the internal die interleaving and plane sharing of flash memory, we propose the request batching strategy.
- We build a simulator, which can support both the channel-level and flash-level parallelism, and implement the delay-based request scheduling with the proposed SSD model into this simulator.

The rest of this paper is organized as follows. In Section 2, we discuss the background of the internal architecture of SSDs and the motivation of this paper. In Section 3, we present the design of the proposed SSD model and delay-based request scheduling strategy. The performance analysis of the delay-based request scheduling strategy is presented in Section 4. Then, the experimental results are presented in Sections 5 and 6 concludes this work. Finally, for the sake of easy presentation and better comprehension, we summarize the definition of main notations used in the whole paper in Table 1.

2. Background and motivation

2.1. SSD internals

A typical SSD includes four major components as shown in Fig. 1. A host interface connects to the host through an interface connection (e.g. SATA or IDE bus). An SSD controller manages flash memory space, translates incoming requests, and issues commands to flash memory packages via a flash memory controller. Some SSDs have a dedicated DRAM buffer to hold metadata or data, and some SSDs only use a small integrated SRAM buffer to lower production cost [30–33]. In most SSDs, multiple channels are designed to connect the controller with flash memory packages. Each channel may be shared by more than one package. Actual implementations may vary across different models, and previous work [1,22,34,35] gives detailed descriptions about the architecture of SSDs.

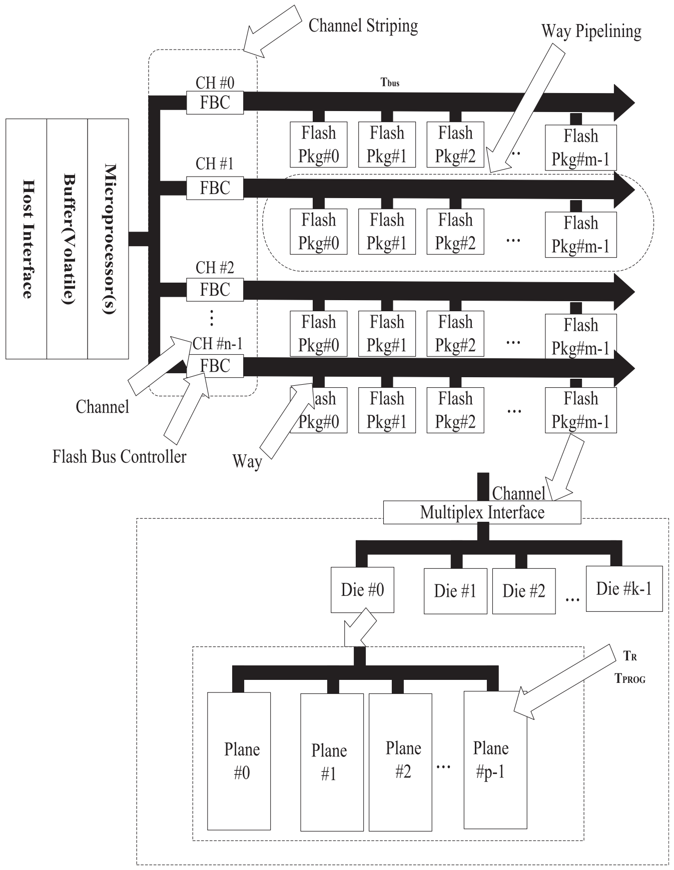


Fig. 1. The internal structure of a typical SSD.

By examining the internal architectures of SSDs, we can find that parallelism is available at different levels, and operations at each level can be parallelized or interleaved.

- Channel-level parallelism: In an SSD, multiple flash packages are connected to the flash controller through multiple channels. The flash controller allocates multiple requests to each channel, monopolizing the I/O bus before the data is transferred to flash in each channel, channel bus affects the performance of system-level data concurrency [36,37]. Some SSDs adopt multiple Error Correction Code (ECC) engines and flash controllers, each for a channel, for performance purposes [38].
- Flash-level parallelism: In an SSDs system-level structure, channels and flash packages can determine the unique data transmission path. At a time, only one flash in a channel is active. Multiple flash packages share the same channel, so the I/O bus clock frequency of the channel determines the speed of parallelism. Otherwise, due to operations on flash memory packages attached to the same channel can be interleaved, so the bus utilization can be optimized [1].
- Die-level parallelism: A flash package usually contains two or more dies, dies used to persist data. As Fig. 1 shows, this is called die interleaving. As in the pipelining method, the data movements and flash command controls in this die interleaving need to be serialized because of the single data path between the flash interface and multiple flash dies. However, in an ideal case, where the data movements are overlapped with the underlying memory operations, the performance increases by about k times, where k is the number of dies.
- Plane-level parallelism: Since a plurality of planes composes a die, parallel access to data can be effectively performed when multiple requests arrive at the die. Plane share concurrently activates flash operations on multiple planes, which can improve the performance by about p times, where p is the number of planes. Finally, these two parallel data access methods can be combined when incoming I/O requests span all of the flash internal components. This method is referred to as die interleaving with multiplane, and it can improve the performance by about k x p times. However, higher parallelism in these methods is available only via the advance flash commands. Therefore, the access pattern to be built by the advance flash commands is important for achieving high die-level and plane-level parallelism.

Such a highly parallelized structure provides rich opportunities for parallelism. In this paper, we fully explored the characteristics of parallel structures and parameterized these characteristics to design a better performance scheduling algorithm to improve the performance of SSD systems. Finally, it should be noted that, while system-level components(i.e., channels) have few restrictions that prevent them from being enabled in parallel, data accesses should satisfy specific flash control commands and data movement sequences to activate all of the flash-level components(i.e., dies). For example, to enable all of the planes in a flash die, the target memory addresses of an I/O request should not only indicate different plane numbers but also have all of the same page offsets because of the shared wordline architecture. These low-level constraints and flash management protocols are specified in the advanced flash commands, which are defined by the flash maker.

2.2. Native Command Queuing (NCQ)

NCQ is an extension of the Serial ATA protocol that allows hard disk drives to internally optimize the order of commands, and it is still widely employed in SSDs [39]. With NCQ support, the device can accept multiple incoming commands from the host and schedule the jobs internally. NCQ is especially important to SSDs, because the highly parallelized internal structure of an SSD can be effectively utilized only when the SSD is able to accept multiple concurrent I/O jobs from the host (operating system) [40,41]. Early generations of SSDs do not support NCQ and thus cannot benefit from parallel I/O [42]. A good scheduling constraint for NCQ can improve the performance in terms of the average response time

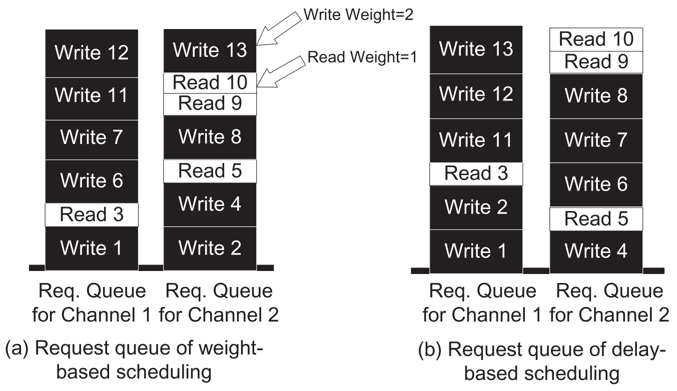


Fig. 2. Illustration of request dispatching by adopting the weight-based and delay-based scheduling policies, respectively.

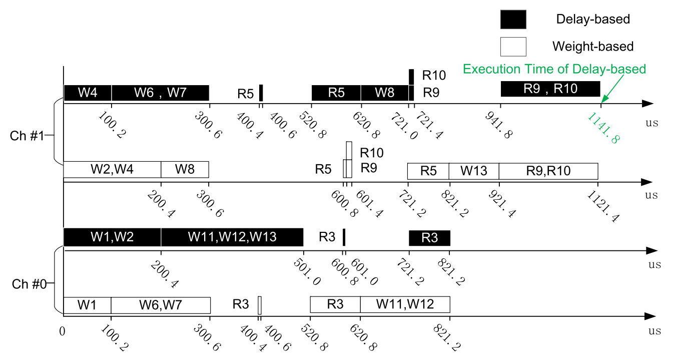
without any loss of IOPS (Input/Output Operations Per Second) and I/O bandwidth [43].

2.3. Motivating example

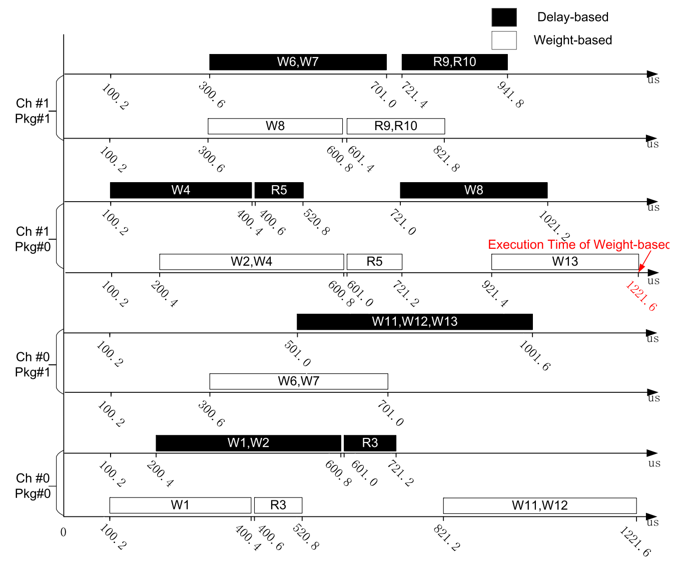
Fig. 2 illustrates the reordered requests in the request queue by adopting the weight-based and delay-based scheduling policies, respectively. The weight-based scheduling maintains a table of weights in each queue to predict the latency of processing all requests in these queues. Since there are different types of requests (e.g., read and write) with various latencies, the weight-based scheduling assigns different weights to the different types of requests. As shown in Fig. 2(a), the weights of read and write requests are assigned to be 1 and 2, respectively. When 13 requests are accumulated in the SSD controller, the requests are first classified into different groups based on the request type (e.g., write request). For write request, the weight-based scheduling strategy selects a channel with the least value of weight, and inserts the write request to this channel. In addition, for a read request, it is inserted into the corresponding channel according to the path of the read request. However, the delay caused for each read or write request heavily depends on the underlying storage status and simply using the fixed weights to predict the performance of each channel is inaccurate. As a result, the weight-based scheduling may introduce the unbalanced channel response and seriously affect the overall performance of SSDs, which will be unveiled in the following section.

Fig. 2(b) illustrates the request dispatching using the proposed delay-based scheduling. Similarly, when 13 requests are received in the SSD controller, the delay-based scheduling strategy dynamically calculates the loads on each channel and schedules the requests to the channel with the minimal burden. Therefore, compared with the weight-based scheduling, the delay-based scheduling takes the device status as an important factor to perform request scheduling, thus generating the more balanced request dispatching among different channels.

Fig. 3 illustrates the execution of requests in an SSD by using weight-based and delay-based scheduling polices. Micron 16GB MLC NAND flash [29] is used as the simulated flash device. The page read and write time are configure to be 20 us and 200 us, respectively. The transfer rate of the buses is configured to be 38 MB/s. As shown in Fig. 3, different from the weight-based scheduling, requests 1 and 2 are assigned to Ch#0Pkg#0 for delay-based scheduling. This is because the global waiting time of all the requests can be minimized by adopting this assignment. The request 3 is a read request, so it can only be assigned to Ch#0Pkg#0 to access the required data. Then, for write request 4, the delay-based scheduling policy selects Ch#1Pkg#0 to achieve the global optimization. However, since the read request 5 can only be assigned to Ch#1Pkg#0, the scheduling policy predicts that the channel can only transfer 2 write requests. Therefore, the write requests 6, 7 are assigned to Ch#1Pkg#1. Next, we will conduct a detailed analysis



(a) The execution of requests on the channel



(b) The execution of requests in the flash memory

Fig. 3. The execution of requests in an SSD.

of the weight-based request scheduling strategy. In order to make full use of die interleaving and plane sharing, the write requests 2 and 4 are assigned to Ch#1Pkg#0. However, since requests 3 and 5 are read requests, they are assigned to Ch#0Pkg#0 and Ch#1Pkg#0 respectively according to their request path. In order to fully use the channel bus and the parallelism inside the flash memory, the write requests 6 and 7 are assigned to Ch#0Pkg#1 as a batch request. Since the read requests R9 and R10 satisfy the creation condition of the read batch requests, and the idle time of the channel bus is sufficient to respond to the two requests, the read operations R9 and R10 are assigned to Ch#1Pkg#1 as the read batch requests. Finally, this example shows that the request latency of the delay-based request scheduling strategy is improved by 6.5% compared with the weight-based request scheduling strategy.

3. The proposed model and scheduling policy

3.1. Design of SSD model

In an SSD-based multiple parallel systems, in order to fully utilize the parallel resources, it is critical to construct a model so as to effectively and efficiently predict the resource utilization of SSD devices. In the process, the SSD's characteristics that affect its service time are abstracted into a set of parameters. The parameters include the number of channels, the number of flash chips in each channel, the data access time in the bus and etc. Note that the proposed model can cover a wide range of modern SSDs and is integrated into the SSD controller, which can easily obtain the internal information inside SSDs. Next, we present the proposed model in detail.

The parallel resources inside SSDs include channel-level parallelism and flash-level parallelism. We first present the time consumption used in each flash channel according to Eqs. (1) and (4). Then, the data access time used in multiple dies/planes can be represented by using Eqs. (6) and (7).

The transmission time of the batch write requests in the flash channel is calculated as follows:

$$T_{channel\_w} = N_{request\_batch} * (T_{commands} + T_{data}) \tag{1}$$

$T_{commands}$ and T_{data} can be calculated via Eqs. (2) and (3) respectively as follows:

$$T_{commands} = N_{commands} * T_{bus} \tag{2}$$

$$T_{data} = S_{page} * T_{bus} \tag{3}$$

The calculation of $T_{commands}$ in Eq. (4) is divided into three parts. First, a T_{bus} time is used to transmit a start command (00h) over the bus. After that, it takes 5 T_{bus} times to transfer the address of the read/write operation over the bus, followed by a T_{bus} time transfer end command (30h). The T_{data} is the time used to transfer a physical page on the bus. This model is built based on the real NAND flash memory [44]. For NAND flash memory, the bus only transfers two types of information (i.e., commands and data) with a fixed transfer speed. So, the time consumed in the bus can be modeled by Eqs. (2) and (3). The transmission time of the batch read requests in the flash channel is calculated as follows:

$$T_{channel\_r\_commands} = N_{request\_batch} * T_{commands} \tag{4}$$

$$T_{channel\_r\_data} = N_{request\_batch} * T_{data} \tag{5}$$

Eq. (4) represents the time used to transfer the commands and Eq. (5) represents the time used to transfer the batch data requests.

In the flash chip, the execution time of read and write requests can be represented by the following equations:

$$T_{flash\_w} = N_{request\_batch} * (T_{commands} + T_{data}) + T_{PROG} \tag{6}$$

$$T_{flash\_r} = N_{request\_batch} * (T_{commands} + T_{data}) + T_{R} \tag{7}$$

This flash program and read model are established according to the real NAND flash memory [44]. The flash program operation contains two stages, including the data/command bus transfer and flash program. Therefore, flash program time for batch requests (T_{flash_w}) can be modeled by Eq. (6). Similarly, the flash read time for batch requests can be modeled by T_{flash_w}.

T_R and T_{PROG} are the read and write time in the plane, respectively. For write requests, the die interleaving command sends multiple requests to the die. When the write requests are issued to the die, multiple write requests are sent to the cache register. Then, the flash memory begins to program data. The time at which the flash executes the write requests is shown in Eq. (6). However, for read requests, the die interleaving command sends multiple read requests from different plane addresses to each die. When the read requests are issued to the die, multiple read request addresses are sent to the plane. Then, the read data is cached in the cache register. The time consumed to handle the read requests is shown in Eq. (7).

Eq. (8) is used to calculate the total delay of the nth channel, which is described as follows:

$$queueTotalDelay_n = \sum_{i=1}^{N} delay_i \tag{8}$$

N represents the number of waiting requests in a queue.

The Eq. (9) for calculating $delay_i$ is presented as follows:

$$delay_i = T_{predict\_request\_end} - T_{request\_arrive} - T_{request\_time} \tag{9}$$

The $T_{request\_arrive}$ indicates the time when the request arrived, $T_{request\_time}$ indicates the time when the request was executed.

We implement a trace-driven simulator that emulates the internal structure of SSDs. The internal structure inside SSDs can be divided into multiple parallel components, including multiple channels and planes/dies. When multiple requests are issued to the SSD, the SSD controller first assigns these requests across multiple channels. In Eq. (8), we use n to represent the number of channels. To model the time consumption in the flash channels, T_bus in Eq. (2) is used to represent the time consumed in each channel. In flash memory, flash-level activities should be serialized, since the multiplexed interfaces of each flash package are shared within one channel. However, the data movements inside flash memory can be pipelined between multiple dies/planes. Thus, the SSD performance can also be influenced by utilizing these pipelining resources. In Eqs. (1), (6) and (7), we use k and p to represent the number of dies and planes in the flash memory, respectively. Based on this simulator, we design a delay-based request scheduling strategy. The delay-based strategy fully utilizes the parallel resources by online recording and calculating the requests issued in the SSD and the parallel resource utilization. The time consumed by the issued requests can be calculated according to Eqs. (1), (4) and (5). The time consumed by the read/write requests inside the flash memory is calculated according to Eqs. (6) and (7). Based on these equations, we can obtain the channel and die/plane status (i.e., idle or busy). Finally, by reordering the pending requests with the goal modeled by Eq. (8), we can obtain the desired scheduling results.

3.2. Design of delay-based request scheduling strategy

We perform intelligent request scheduling, called delay-based scheduling, according to the proposed models described in Section 3.1. The delay-based scheduling strategy dynamically uses the proposed model to calculate the loads on each channel and schedules the requests to the channel with the minimal burden. In addition, in order to make full use of the die interleaving and plane sharing inside the flash memory, the requests are batched into different groups. The requirements for request batching are presented as follows:

- For write request group, the number of write requests inside the write request group should be less than or equal k X p. Meanwhile, the write request within a request group should fully use the idle channel buses.
- For read request group, the read requests of different planes in the same flash memory are merged into the same batch request. Meanwhile, the read request within a request group should fully use the idle channel buses.

Algorithm 1 describes the proposed delay-based scheduling policy. When the requests are accumulated in the SSD controller, Algorithm 1 will all the received requests are assigned to the channel queue according to some special rules. Then, the request in the channel queue is waiting to be executed. Since the access location of the read requests are fixed, we only need to allocate the flash space for the write requests. The procedures of request scheduling are presented as follows. First, the delay-based scheduling determines the type of the request. In the algorithm, we use 1 to represent the write type (line 2). In line 5, we loop through all the channels to find the nth channel with the smallest total delay. Then, we find the earliest free flash m in the nth channel, which is described in line 6. We create a temporary batch requests (line 9), which creation rules are shown in the Section 3.2. The delay-based request scheduling strategy utilizes the idle time of the channel bus, the free flash and temporary batch requests linearly predict the actual number of batch requests and the requested delay time(line 17). The $batch\_requests$ is a request set which contains the concurrently executed

Algorithm 1: Delay-based Request Scheduling

```
Input :
        Requests issued from the applications
Output:
        Reordered requests
1  /*type = 1 : write; type = 0 : read; type = other : error*/
2  if requests(0).type == 1 then
3      /*Obtain the channel with the least delay
4      according to variable queueTotalDelay*/
5      c ← select_channel (channels);
6      f ← select_flash (c.flashs);
7      /*According to the creation rule of the write
8      request group*/
9      tmp_batch_requests ← build_batch_requests (requests);
10 else if requests(0).type == 0 then
11     c, f ← obtain_channel_flash(r);
12     /*According to the creation rule of the read request
13     group*/tmp_batch_requests ←
       build_batch_requests (requests, requestsCount);
14 else error ("no such type of operation") ;
15 /*A delay-based request scheduling strategy is used to linearly
   predict the delay of this batch request and the execution end time.
16 */
17 delay_i, T_predict_request_end, batch_requests ←
   linear_predict_parameters (c, f, tmp_batch_requests);
18 /*Update the channel information, for
19 example, update total delay of the channel
20 according to Equation (8) and insert batch_requests
21 into channel queue*/
22 update_channel_Information (c, f, batch_requests,
23                    delay_i, requests, T_predict_request_end);
```

requests at a time. The $T_{predict\_request\_end}$ denotes the time point of finishing handling the batch requests. We can use Eqs. (1) and (6) to predict the end time of batch write requests. In addition, we use Eqs. (4), (5), and and (7) to predict the end time of batch read requests. Finally, the delay-based request scheduling policy inserts linear prediction information into the nth channel queue and updates the total delay time of this channel (line 22). Linear forecasting information mainly includes batch requests, request delay time, and the batch requests start and end times.

Similarity, since the request path for the read request is fixed, there is no need to dynamically select the channel and flash (line 11).

4. Performance analysis

In this section, we will analyze the time complexity and space complexity of the delay-based request scheduling strategy.

4.1. Timing analysis

The time complexity of the delay-based request scheduling strategy is mainly related to the requested batch, and the more batches requested, the worse the time complexity. The execution process of the ith batch request is described as follows. The ith batch requests latency, $T^i_{requests}$, can be represented by $Latency^i_n + T_{channel-contention} + T_{channel-idle}$. The $Latency^i_n$ is the total execution time in nth channel, whereas $T_{channel-contention}$ and $T_{channel-idle}$ are the time spent to resolve the channel-level contention and on idle, respectively. The $Latency^i_n$ can be calculated by $\sum_{m=1}^{k} Latency^m_{way} + T^m_{datamovement}$. The k is the maximum number of I/O requests spread across the free flash package in data bus, and $T^m_{datamovement}$ shows the latency of bring out the request data from the memory or deliver request data from the channel. Lastly, mth request latency, $Latency^m_{way}$, can be represented by $T^m_{memory} + T^m_{memory-busy}$, where

T^m_{memory} and $T^m_{memory-busy}$ indicate the operation time for a read or write, and the device busy duration of the request address, respectively. The delay time of the ith batch requests is calculated linearly, and the main factors are related to $T_{channel-contention}$ and $T_{channel-idle}$. Through the process analysis of the above request execution time, it can be known that when the number of requests accepted by the SSD controller is constant, the more the request batch, the worse the performance. Because the more batches are requested, the more time it takes to resolve channel-level contention and idle time. So its time complexity is O(n), where n is the number of request batches. For example, in Section 2.3, when there are 13 requests arriving at the SSDs controller, the delay-based request scheduling algorithm divides the 13 requests into 8 batches, so the time complexity is O(8). However, the weight-based request scheduling strategy divides the 13 requests into 9 batches with the time complexity of O(9). Therefore, the delay-based request scheduling strategy has better time performance.

4.2. Space overhead analysis

The space complexity of the delay-based request scheduling strategy is closely related to the number of requests and channels. The more the number of requests and channels, the worse the space complexity. The specific analysis is as follows, the delay-based request scheduling policy selects a channel with the lowest total delay for a request. So each channel has to cache a $queueTotalDelay_n$, its space complexity is O(n), and n is the number of SSDs channels. Furthermore, since the delay-based request scheduling policy linearly predicts the delay time of the request, its valid information is cached for each batch request (i.e., $delay_i$). So the space complexity for this part is O(k), where k is the number of requested batches. Finally, the delay-based request scheduling policy inserts the requests into the channel queue and waiting to be executed, the time complexity is O(s), and s is the number of requests. In summary, the total time complexity of the delay-based request scheduling strategy is O(n + k + s). Since the number k of batch requests is calculated linearly by the number n of requests. Therefore, the time complexity of the delay-based request scheduling strategy is O(n + s). In addition to this, the weight-based request scheduling policy stores the total weight in each channel and inserts the corresponding request into the channel queue. Therefore, the weight-based request scheduling strategy has a space complexity of O(n + s). For example, in Section 2.3, when the number of channels is 2 and the number of requests is 13, the space complexity of the delay-based and weight-based request scheduling strategies is O(15).

5. Evaluation

In this section, we first describe the experimental setup for evaluating the effectiveness of the proposed delay-based scheduling strategy. Then, we comprehensively evaluate and analyze the experimental results of the proposed scheme.

5.1. Experimental setup

We develop a trace-driven SSD simulator and implement three schemes: RR [27], weight-based [28], and delay-based. Note that the RR used in this section integrates prior wisdoms with the consideration of the parallel resources in SSDs. In the SSDs simulator, we use 16 channels and 8 flash chips per channel (128 flash chips in total) to emulate an SSD device. Micron 16 Gb multi-level cell (MLC) NAND flash [29] is used to simulate the flash device. We evaluate the effectiveness of the proposed scheme on the Financial and WebSearch data sets [45]. The transfer speed of buses inside SSDs is configured to be 38 MB/s or 50 MB/s. The page read latency of MLC NAND is configured to be 20 us or 30 us, and the page write latency is configured to be 200 us or 400 us. The page size that we used in this experiment is 4 KB. The block consists of 128 pages and each plane contains 2048 blocks. The MLC flash uses dual-die and two-plane architecture.

Table 2
Characteristics of the benchmarks.

	Data size (MB)	Write fraction (%)	Avg. write size (KB)	Avg. read size (KB)
Financial1	16,944	82.3	1.6	1.4
Financial2	9284	22.4	1.4	1.3
WebSearch1	62,108	0.01	1.7	1.8
WebSearch3	62,996	0.02	1.6	1.8

Table 2 presents the characteristics of the benchmarks used in our experiment, including request size and read/write ratio. We choose real enterprise workloads [45], which contain the read-intensive web search (e.g., WebSearch1) and write-intensive financial transactions (e.g., Financial1).

5.2. Experimental results

To evaluate the impact of RR, delay-based, and weight-based request scheduling strategies, we use IOPS (Input/Output Operations Per Second) and latency as our metrics. In addition, we explore the impact of the page read and write latency and bus transfer rate.

5.2.1. Performance comparison of delay-based and weight-based request scheduling strategy

For evaluating RR, delay-based, and weight-based request scheduling strategies, we first configure the bus transfer rate to be 38 MB/s, the page read and write latency to be 20.0 us and 200 us, respectively. The baseline dispatching policy is the simple round-robin (RR) dispatching, which uniformly distributes write requests to all channels in SSDs. As shown in Figs. 4 and 5, for write-intensive Financial1, the IOPS and request latency of the weight-based request scheduling strategy are improved by 9.6% and 8.8% respectively compared with the RR-based request scheduling strategy. However, for read-intensive WebSearch1, the IOPS and request latency of the weight-based request scheduling strategy are improved by 77.8% and 43.8% respectively compared with the RR-based request scheduling strategy. The experimental results show that the weight-based request scheduling strategy is better than that of the RR request scheduling strategy for read-intensive applications. In addition, we compare the impact of delay-based and weight-based request

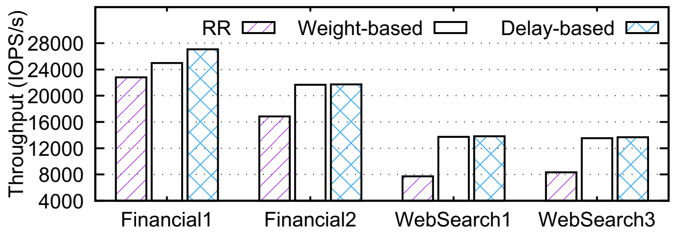


Fig. 4. The throughput of SSDs.

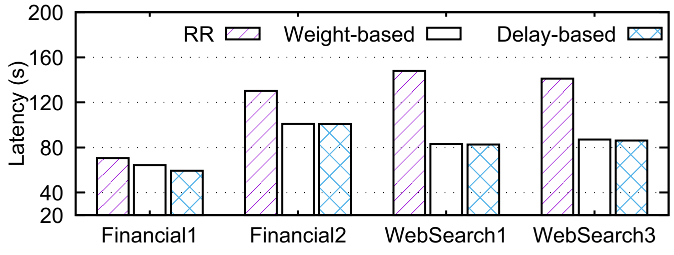


Fig. 5. The latency of benchmarks.

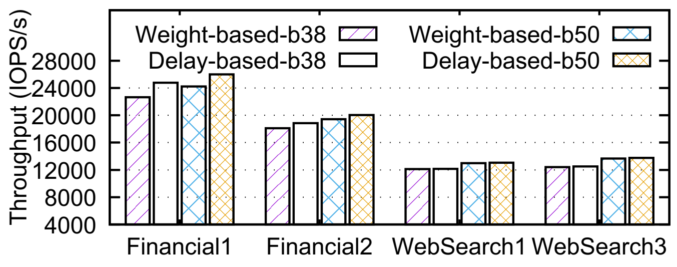


Fig. 6. The throughput with the consideration of the bus speed.

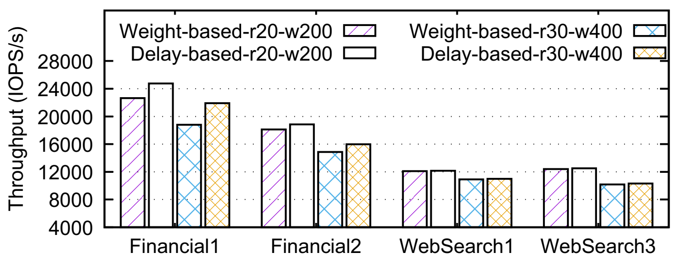


Fig. 8. The throughput under the different page read and write time.

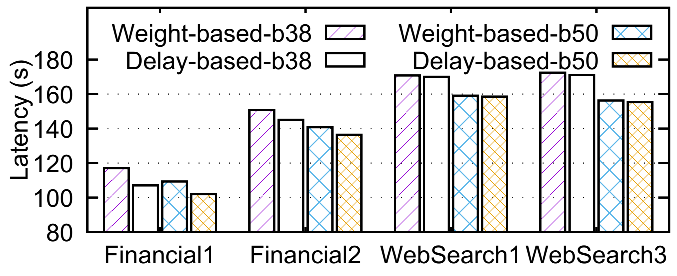


Fig. 7. The latency with the consideration of the bus speed.

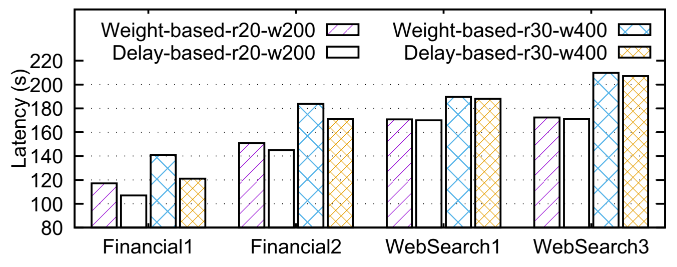


Fig. 9. The latency under the different page read and write time.

scheduling strategies on IOPS and request latency. For write-intensive application Financial1, the IOPS and request latency of the delay-based request scheduling strategy is improved by 8.4% and 7.8% respectively compared with the weight-based request scheduling strategy. At the time, for the read-intensive application WebSearch1, the IOPS and request latency of the two policies are roughly the same. In summary, the delay-based request scheduling performs better than the weight-based request scheduling strategy for write-intensive applications. However, for read-intensive applications, the proposed design achieves the similar performance as the weighted-based request scheduling. For the RR baseline scheme, the delay-based request scheduling strategy performs better for both read-intensive and write-intensive applications.

5.2.2. *The effect of bus transfer rate on two request scheduling strategies*

The transmission speed of the bus is one of the factors that affects the channel-level parallelism. Therefore, we compare the impact of the request allocation strategies on SSDs performance at different bus transmission speed. The page read and write latency are configured to be 20 us and 200 us, respectively. The transfer rate of the buses is configured to be 38 MB/s or 50 MB/s. As shown in Figs. 6 and 7, for write-intensive application Financial1, when the bus transfer rate is 38 MB/s, the IOPS and request latency of the delay-based scheduling strategy are improved by 9.3% and 8.5% respectively compared with the weight-based request scheduling strategy. In addition, when the transfer rate of the bus is 50 MB/s, the IOPS and request latency of the delay-based request scheduling strategy is improved by 7.2% and 6.7% respectively compared with the weight-based request scheduling strategy.

5.2.3. *Impact of page read and write latency on the scheduling strategies*

The page read and write latency are one of the factors that affects the latency of the flash response. In this section, we compare the impact of request scheduling strategies on SSDs performance at different page read and write latency. The bus transfer rate is configured to be 38 MB/s, and the read and write latency are set to be 20 us, 200 us, and 30 us, 400 us respectively. As shown in Figs. 8 and 9, for write-intensive application Financial1, when the read and write latency are set to be 20 us, 200 us, the IOPS and request latency of the delay-based request scheduling strategy is improved by 9.3% and 8.5% respectively compared with the weight-based request scheduling strategy. In addition, when the read and write latency are configured to be 30 us and 400 us, respectively, for the write-intensive Financial data sets, the IOPS and request latency of the delay-based request scheduling strategy are improved by 16.5% and 14.2% respectively compared with the weight-based request scheduling strategy. On the other hand, for the read-intensive WebSearch1, the proposed design achieves limited performance improvement (i.e., 0.899% for IOPS and 0.896% for latency). Different from read requests with the fixed access address, write requests can be dynamically allocated to different physical locations with the help of the mapping table integrated in FTLs. Therefore, the proposed delay-based request scheduling strategy is favorable to write-intensive applications.

6. Conclusion and future work

The parallel components inside SSDs have become the key role in determining the performance of SSDs. We observe that the state-of-the-art request scheduling strategies such as weight-based scheduling strategy may lead to the under-utilization of available bandwidth in SSDs. To address this issue, we propose an SSD model and a delay-based scheduling strategy to well utilize the parallel resources in SSDs. The SSD model not only considers the organization of the parallel components in SSDs but also explores the utilization of these components. By well utilizing the SSD model, the delay-based request scheduling strategy can accurately predict the resource utilization and intelligently allocate requests to the parallel components (i.e., channels) to achieve the high I/O throughput and low device access latency.

We present detailed evaluation and analysis to quantify how well our techniques can enhance the I/O performances of SSDs via efficient device modeling and request scheduling. Our experimental results show that the proposed techniques yield up to 16.5% performance improvement over the state-of-the-art techniques. In addition, we observe that the performance improvement is sensitive to the differences between the bus transfer rate and page read time. The experimental results show that when the performance gap between the time consumed in the internal flash bus and page preparation becomes larger, the proposed scheme can achieve more compared with the baseline schemes. In the future, how to combine the proposed techniques with modern computation devices (e.g., GPUs) [46–48] would be an interesting topic to be explored.

Acknowledgments

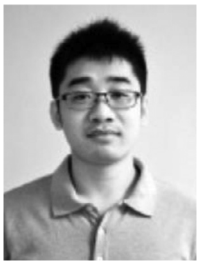
The work described in this paper is partially supported by the grants from the National Natural Science Foundation of China (61702357, 61672377 and 61502309), Shenzhen Science and Technology Foundation (JCYJ20170816093943197 and JCYJ20170817100300603), Natural Science Foundation of Tianjin (18JCQNJC00300), Guangdong Natural Science Foundation (2016A030313045), and Peiyang Scholar Foundation of Tianjin University (2019XRG-0004).

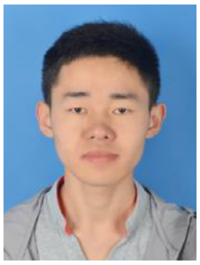
Conflict of interest

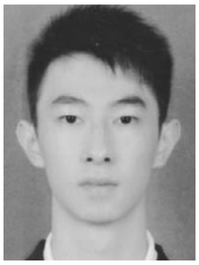
The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] C. Dirik, B. Jacob, The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization, in: International Symposium on Computer Architecture, 2009, pp. 279–289.
[2] Y. Luo, S. Ghose, Y. Cai, E.F. Haratsch, O. Mutlu, Heatwatch: improving 3D NAND flash memory device reliability by exploiting self-recovery and temperature awareness, in: IEEE International Symposium on High Performance Computer Architecture, 2018, pp. 504–517.
[3] Y. Luo, S. Ghose, Y. Cai, E.F. Haratsch, O. Mutlu, Improving 3D NAND flash memory lifetime by tolerating early retention loss and process variation, in: International Conference on Measurement and Modeling of Computer Systems, 2018. 106–106
[4] W.S. Ngueya, J.M. Portal, H. Aziza, J. Mellier, S. Ricard, An ultra-low power and high performance single ended sense amplifier for low voltage flash memories, J. Low Power Electron. 14 (1) (2018) 157–169.
[5] J. Boukhobza, S. Rubini, R. Chen, Z. Shao, Emerging NVM: a survey on architectural integration and research challenges, ACM Trans. Des. Autom.Electron. Syst. 23 (2) (2018) 14:1–14:32.
[6] R. Chen, Y. Wang, D. Liu, Z. Shao, S. Jiang, Heating dispersal for self-healing NAND flash memory, IEEE Trans. Comput. 66 (2) (2017) 361–367.
[7] X. Chen, E.H. Sha, Q. Zhuge, T. Wu, W. Jiang, X. Zeng, L. Wu, UMFS: an efficient user-space file system for non-volatile memory, J. Syst. Archit. 89 (2018) 18–29.
[8] J. Zhan, Y. Zhang, W. Jiang, J. Yang, L. Li, Y. Li, Energy-aware page replacement and consistency guarantee for hybrid NVM-DRAM memory systems, J. Syst. Archit. 89 (2018) 60–72.
[9] Y. Kang, X. Zhang, Z. Shao, R. Chen, Y. Wang, A reliability enhanced video storage architecture in hybrid SLC/MLC NAND flash memory, J. Syst. Archit. 88 (2018) 33–42.
[10] K. Qiu, Z. Gong, D. Zhou, W. Chen, Y. Xu, X. Shi, Y. Liu, Efficient energy management by exploiting retention state for self-powered nonvolatile processors, J. Syst. Archit. 87 (2018) 23–35.
[11] D. Liu, X. Luo, Y. Li, Z. Shao, Y. Guan, An energy-efficient encryption mechanism for NVM-based main memory in mobile systems, J. Syst. Archit. 76 (2017) 47–57.
[12] Y. Wang, T. Wang, D. Liu, Z. Shao, J. Xue, Fine grained, direct access file system support for storage class memory, J. Syst. Archit. 72 (2017) 80–92.
[13] Y. Tan, B. Wang, J. Wen, Z. Yan, H. Jiang, W. Srisa-an, Improving restore performance in deduplication-based backup systems via a fine-grained defragmentation approach, IEEE Trans. Parallel Distrib. Syst. 29 (10) (2018) 2254–2267.
[14] R. Chen, C. Zhang, Y. Wang, Z. Shen, D. Liu, Z. Shao, Y. Guan, DCR: Deterministic crash recovery for NAND flash storage systems, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. (2018). 1–1
[15] M. Bellato, G. Collazuol, I. D'Antone, P. Durante, D. Galli, B. Jost, I. Lax, G. Liu, U. Marconi, N. Neufeld, et al., A PCIe Gen3 based readout for the LHCb upgrade, J. Phys. 513 (2014).
[16] Y. Ou, N. Xiao, F. Liu, Z. Chen, W. Chen, L. Wu, Gemini: a novel hardware and software implementation of high-performance PCIe SSD, Int. J. Parallel Program. 45 (4) (2017) 923–945.
[17] Z. Qin, Y. Wang, D. Liu, Z. Shao, Y. Guan, MNFTL: an efficient flash translation layer for MLC NAND flash memory storage systems, in: Design Automation Conference, 2011, pp. 17–22.
[18] D. Liu, K. Zhong, T. Wang, Y. Wang, Z. Shao, E.H. Sha, J. Xue, Durable address translation in PCM-based flash storage systems, IEEE Trans. Parallel Distrib. Syst. 28 (2) (2017) 475–490.
[19] R. Chen, Z. Qin, Y. Wang, D. Liu, Z. Shao, Y. Guan, On-demand block-level address mapping in large-scale NAND flash storage systems, IEEE Trans. Comput. 64 (6) (2015) 1729–1741.
[20] M. Jung, Exploring parallel data access methods in emerging non-volatile memory systems, IEEE Trans. Parallel Distrib. Syst. (2017) 746–759.
[21] F. Chen, R. Lee, X. Zhang, Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing, in: International Conference on High-Performance Computer Architecture, 2011, pp. 266–277.
[22] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, R. Panigrahy, Design tradeoffs for SSD performance, in: USENIX Annual Technical Conference, 2008, pp. 57–70.
[23] B. Mao, S. Wu, L. Duan, Improving the SSD performance by exploiting request characteristics and internal parallelism, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 37 (2) (2018) 472–484.
[24] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, S. Zhang, Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity, in: International Conference on Supercomputing, 2011, pp. 96–107.
[25] E.H. Nam, B.S.J. Kim, H. Eom, S.L. Min, Ozone (O3): an out-of-order flash memory controller architecture, IEEE Trans. Comput. 60 (5) (2011) 653–666.
[26] A.M. Caulfield, A. De, J. Coburn, T.I. Mollow, R.K. Gupta, S. Swanson, Moneta: A high-performance storage array architecture for next-generation, non-volatile memories, in: IEEE/ACM International Symposium on Microarchitecture, 2010, pp. 385–395.
[27] M. Shreedhar, G. Varghese, Efficient fair queuing using deficit round-robin, IEEE/ACM Trans. Netw. 4 (3) (1996) 375–385.
[28] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, J. Cong, An efficient design and implementation of LSM-tree based key-value store on open-channel SSD, in: Ninth Eurosys Conference, 2014, pp. 16:1–16:14.
[29] Micron, 16Gb MLC NAND flash memory, https://www.micron.com/products/nand-flash/mlc-nand/16Gb (2018).
[30] L. Wang, Q. Wang, L. Chen, X. Hao, Fine-grained data management for DRAM/SSD hybrid main memory architecture, IEICE Trans. Inf. Syst. 99 (12) (2016) 3172–3176.
[31] H.J. Song, Y.H. Lee, I/o performance and power consumption analysis of HDD and DRAM-SSD, Commun. Comput. Inf. Sci. 150 (2011) 191–196.
[32] Z. Chen, Y. Lu, N. Xiao, F. Liu, A hybrid memory built by SSD and DRAM to support in-memory big data analytics, Knowl. Inf. Syst. 41 (2) (2014) 335–354.
[33] J. Wang, E. Lo, M.L. Yiu, J. Tong, G. Wang, X. Liu, Cache design of SSD-based search engine architectures: an experimental study, ACM Trans. Inf. Syst. 32 (4) (2014) 21:1–21:26.
[34] A.I. Alsalibi, S. Mittal, M.A. Al-Betar, P.B. Sumari, A survey of techniques for architecting SLC/MLC/TLC hybrid flash memorybased SSDs, Concurr. Comput. Pract. Exper. (6) (2018).
[35] J.H. Huang, R.S. Liu, DI-SSD: desymmetrized interconnection architecture and dynamic timing calibration for solid-state drives, in: Asia & South Pacific Design Automation Conference, 2018, pp. 34–39.
[36] J.W. Hsieh, H.Y. Lin, D.L. Yang, Multi-channel architecture-based FTL for reliable and high-performance SSD, IEEE Trans. Comput. 63 (12) (2014) 3079–3091.
[37] D. Kim, K.H. Park, C.H. Youn, Supa: a single unified read-write buffer and pattern-change-aware FTL for the high performance of multi-channel SSD, ACM Trans. Storage 13 (4) (2017) 1–30.
[38] C. Park, P. Talawar, D. Won, M.J. Jung, A high performance controller for NAND flash-based solid state disk (NSSD), in: Non-Volatile Semiconductor Memory Workshop, 2006, pp. 17–20.
[39] Y. Kim, An empirical study of redundant array of independent solid-state drives (RAIS), Cluster Comput. 18 (2) (2015) 1–15.
[40] Y.J. Yu, I.S. Dong, H. Eom, H.Y. Yeom, NCQ Vs. i/o scheduler: preventing unexpected misbehaviors, ACM Trans. Storage 6 (1) (2010) 658–673.
[41] L. Mamatas, V. Tsaoussidis, Differentiating services with noncongestive queuing (NCQ), IEEE Trans. Comput. 58 (5) (2009) 591–604.
[42] L. Bouganim, B. Jnsson, P. Bonnet, uFLIP: understanding flash IO patterns (2009).
[43] Y. Cho, T. Kim, An efficient scheduling algorithm for NCQ within SSDs, IEICE Electron. Express 12 (4) (2015). 20150066–20150066
[44] Micron, 2015. (https://www.micron.com/products/nand-flash/3d-nand).
[45] U.T. Repository, 2007. (http://traces.cs.umass.edu/index.php/Storage/Storage).
[46] J. Zhou, Q. Guo, H.V. Jagadish, L. Krcál, S. Liu, W. Luan, A.K.H. Tung, Y. Yang, Y. Zheng, A generic inverted index framework for similarity search on the GPU, in: IEEE International Conference on Data Engineering, 2018, pp. 893–904.
[47] J. Zhou, A.K.H. Tung, Smiler: A semi-lazy time series prediction system for sensors, in: International Conference on Management of Data, 2015, pp. 1871–1886.
[48] R. Chen, Y. Wang, J. Hu, D. Liu, Z. Shao, Y. Guan, Image-content-aware I/O optimization for mobile virtualization, ACM Trans. Embedded Comput. Syst. 16 (1) (2016) 12:1–12:24.


Renhai Chen received the B.E. degree and M.E.degree in the Department of Computer Science and Technology, Shandong University, China, in 2009 and 2012, respectively. He received the Ph.D. degree in the Department of Computing at the Hong Kong Polytechnic University in 2016. He is currently an Assistant Professor at Tianjin University. His research interests include embedded systems, mobile virtualization, and hardware/software codesign.


Qiming Guan received the B.E. degree from the School of Computer Science & Software Engineering, Tianjin Polytechnic University, Tianjin, China, in 2017. He is currently pursuing the M.E. degree at Tianjin University, Tianjin, China. His current research interests include embedded systems and hardware/software codesign.



Chenlin Ma received the BSc computing degree in the Department of Computing, Hong Kong Polytechnic University, in 2014. He is currently working toward the PhD degree in the Department of Computing, Hong Kong Polytechnic University. His research interests include system level design and implementation, and emerging memory techniques for embedded systems.

Zhiyong Feng(M'10) received the Ph.D. degree from Tianjin University, China. He is currently a Full Professor with the School of Computer Software, College of Intelligence and Computing, Tianjin University. He has authored one book, over 130 articles, and 39 patents. His research interests include knowledge engineering, service computing, and security software engineering. He is a member of the IEEE Computer Society and ACM.