DLV: Exploiting Device Level Latency Variations for Performance Improvement on Flash Memory Storage Systems

Jinhua Cui, Youtao Zhang, *Member*, *IEEE*, Weiguo Wu, *Member*, *IEEE*, Jun Yang, *Member*, *IEEE*, Yinfeng Wang, and Jianhang Huang

***Abstract*—NAND flash has been widely adopted in storage systems due to its better read and write performance and lower power consumption over traditional mechanical hard drives. To meet the increasing performance demand of modern applications, recent studies speed up flash accesses by exploiting access latency variations at the device level. Unfortunately, existing flash access schedulers are still oblivious to such variations, leading to sub-optimal I/O performance improvements.**

In this paper, we propose DLV, a novel flash access scheduler for exploring scheduling opportunities due to device level access latency variations. DLV improves flash access speeds based on process variations and data retention time difference across flash blocks. More importantly, DLV integrates access speed optimization with access scheduling such that the average access response time can be effectively reduced on flash memory storage systems. Our experimental results show that DLV achieves an average of 41.5% performance improvement over the state-of-the-art.

***Index Terms*—Flash memories, Process variation, Retention age, LDPC, RBER, Out-of-order scheduler.**

I. Introduction

NAND Flash memory based solid-state drives (SSDs), due to their performance and energy consumption advantages over traditional hard disk drives (HDDs), are widely adopted in modern computer systems, ranging from mobile devices to servers in data centers [1]. Over the past decade, the capacity of flash memory based SSDs has increased dramatically, as a result of technology scaling from 65nm to the latest 10nm technology and the bit density improvement from 1 bit per cell to the latest 6 bits per cell [2], [3]. Unfortunately, flash reliability degrades as flash density increases, that is, there are more retention, read disturbance, cell-to-cell program interference and program/erase (P/E) cycling noises. These noises have led to the raw bit error rate (RBER) increase and access performance degradation accordingly. Adopting reliable yet slow write strategy and strong yet expensive error correction code (ECC) schemes helps to improve access reliability but degrade read and write performance significantly. It has become a major challenge to develop high performance flash memory storage systems to meet the increasing performance demand of modern applications [4].

The I/O performance improvement is often a tradeoff of many factors, such as RBER, read speed, and write speed. Flash read speed and RBER are highly correlated. The higher the RBER is, the stronger capability the ECC requires, the higher complexity the ECC scheme has, and the slower the read requests become. Similarly, there is a close correlation between RBER and the write speed. Studies [5]–[7] showed that a smaller program step size ΔV_p of the incremental step pulse programming (ISPP) scheme, could decrease RBER at the cost of write speed degradation.

To achieve further performance improvement, it is beneficial to differentiate the access latency difference at device level. There are two typical sources. One comes from the process variation (PV) in flash memory [5], [8]–[10], i.e., memory blocks exhibit different RBER under the same P/E cycling. Instead of adopting PV-oblivious programming strategies that assume the worst-case block behavior, Shi *et al.* proposed to use large ΔV_p for strong pages that have low RBER and allocate hot data to these pages [5]. The other device level latency variation comes from retention age variation, i.e., a flash page tends to have high RBER when it was programmed a long time ago. Cai *et al.* showed that lower read-out thresholds can be applied as the actual age of the data increase [4]. Shi *et al.* proposed to adopt higher programming voltages for pages with longer retention time. Liu *et al.* proposed to use finer ΔV_p for performance improvement [11]. While exploiting the latency variations at device level helps to improve flash access speeds, a limitation of these schemes is that latency variations are not exposed to the I/O scheduler. Given that the response time of flash accesses includes cell access time and queuing time, existing schemes focus mainly on the former, i.e., reducing the per access service time, which leads to sub-optimal I/O performance improvements.

In this paper, we propose DLV, a novel flash access scheduler for exploiting device level access latency variations. The following summarizes our contributions.

- DLV employs latency variation-aware I/O scheduling to

J. Cui, W. Wu and J. Huang are with the school of Electronic and Information Engineering, Xi'an Jiaotong University, Shaanxi, 710049, China (E-mails: {cjhnicole, huangjhsx}@gmail.com; wgwu@xjtu.edu.cn).

Y. Zhang is with the Computer Science Department, University of Pittsburgh, Pittsburgh, PA 15260, USA (E-mail: zhangyt@cs.pitt.edu).

J. Yang is with the Electrical and Computer Engineering Department, University of Pittsburgh, Pittsburgh, PA 15261, USA (E-mail: juy9@pitt.edu).

Y. Wang is with the Department of Software Engineering, ShenZhen Institute of Information Technology, Guangdong 518172, China (E-mail: wangyf@mailst.xjtu.edu.cn).

This work is supported in part by the National Key Research and Development Program of China under grant No. 2016YFB1000303 and No. 2016YFB0201800, and in part by National Science Foundation under grants CCF-1718080, and in part by the National Natural Science Foundation of China under grant No. 91630206, and No. 61672423.

Manuscript received Jun. 27, 2017; revised Sep. 05, 2017.

reduce the average access response time. In particular, when prioritizing short flash accesses, DLV evaluates not only the amount of data to access, but also the latency variation in reading and writing each flash page. By exploiting device level latency variation in flash access scheduling, DLV achieves accurate evaluation of the access response time, which enables better scheduling decisions.
- DLV exploits device level block characteristics for maximized scheduling benefits. Given that strong flash blocks in an SSD are often limited, DLV maps hot writes to strong pages only if doing so helps to reduce the access response time. DLV also tracks the PV and retention time of flash blocks to effectively speed up read accesses.
- We evaluate the proposed DLV design and compare it to the state-of-the-art designs. The results show that, on average, DLV achieves an average of 41.5% performance improvement over the state-of-the-art.

In the rest of the paper, Section II presents the background and the motivation. Section III elaborates the details of DLV. The experimental results are analyzed in Section IV. We present more related work in Section V and conclude the paper in Section VI.

II. Background and Motivation

In this section, we briefly discuss the SSD organization and the design tradeoff among RBER, read speed and write speed in NAND flash memory based SSDs. We then present the two sources that lead to device level access latency variations, and motivate our design with preliminary studies.

A. Flash-based SSD Organization

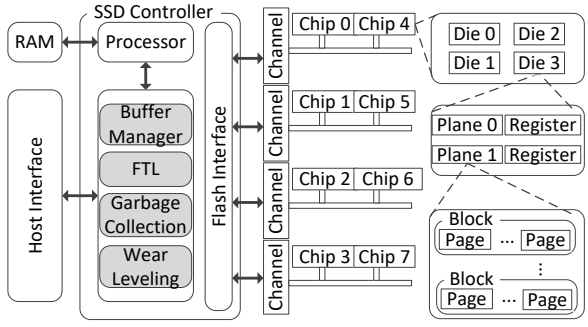


Fig. 1. The SSD organization.

A typical NAND flash memory based SSD contains the SSD controller, RAM, NAND flash memory, host interface and flash interface, as shown in Figure 1. One SSD often contains several channels while each channel connects multiple chips, each chip consists of one or more dies, and each die consists of multiple planes. A plane is the smallest unit that can be accessed independently and concurrently. Each plane is composed of a number of erase units, called blocks, and a block is usually comprised of multiple pages, which are the smallest unit to read/write. There are four main levels of parallelism which can be exploited to accelerate the read/write bandwidth of SSDs, namely channel-level, chip-level, die-level, and plane-level parallelism.

The SSD controller is responsible for converting the read and write requests from the host to the I/O operations of the flash memory. It consists of three main components — flash translation layer (FTL), wear leveling (WL) and garbage collection (GC). The FTL residing in the SSD controller provides logical sector updates by maintaining a mapping table (MPT) of the logical address (LPN) from upper file system to a physical address (PPN) on the flash. The LPN to PPN mapping schemes can be classified into two categories: static and dynamic. A static mapping scheme predetermines channel, chip, die, and plane locations before allocating a logical page. While adopting a static mapping, the FTL still needs to determine the block number within the corresponding plane and the page number within the block at runtime, e.g., mapping the LPN to the next available PPN in the corresponding plane. A dynamic mapping scheme assigns a logical page to any free physical page at any location across the flash memory array, which achieves flexible mapping with additional cost.

Commercial SSD products often adopt the static mapping because the static mapping achieves consistently better performance in serving read requests than that of dynamic ones, and dynamic mapping demands extra space [12], [13]. In this paper, we adopt the static page mapping with the striping order being Channel-first, Chip-second, Die-third and Plane-fourth (CWDP). The CWDP order was proven to be the best for a wide range of workloads [14].

The GC component reclaims used block(s) when the number of the prepared free blocks is below the preset threshold value. If a block containing valid pages is selected as a victim block, GC reallocates those valid pages to other blocks, and updates the mapping table accordingly. To extend the overall lifetime of NAND flash memory, WL techniques are often employed to distribute P/E cycles as evenly as possible among flash blocks.

The RAM in SSD is typically used to temporarily buffer the write requests or accessed data and the mapping table. The host interface connects the SSD and the host system to transfer command and data via USB, PCIe, or SATA interface. The flash interface connects the SSD controller and the NAND chips to transfer data between the controller and the page register [15].

Schemes have been proposed to exploit parallelism for performance improvement. Roh *et al.* proposed *psync I/O* for B+-trees to harvest the internal parallelism in SSDs to enhance B+-tree performance [16]. Hu *et al.* showed that SSD performance can be significantly improved by matching data allocation and priority order to CWDP mapping [12].

B. The Design Tradeoff among RBER, Write Speed, and Read Speed

An NAND flash memory based SSD often seeks to achieve the best tradeoff among RBER, write speed, and read speed. To address RBER, each flash page is associated with an ECC code, e.g., BCH, LDPC (Low-Density Parity-Check Code).

Due to fast technology scaling, the NAND flash reliability deteriorates rapidly, which demands strong ECC protection such as LDPC. LDPC achieves better error correction capability than BCH through soft-decision design, and is replacing BCH in modern SSD products. For example, Macronix 6 bit/cell CT flash chips support LDPC-Based ECC [2], while Samsung 21nm MLC flash chips support at least seven quantization levels [17]. In this paper, we adopt LDPC as the default ECC scheme and exploits its characteristics to improve read performance.

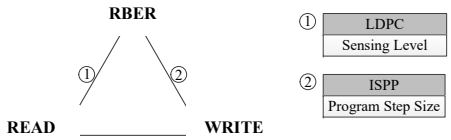


Fig. 2. The tradeoff between RBER, read speed, and write speed.

As shown in Figure 2, there is a tradeoff between read speed and ECC complexity, i.e., the error correction capability. This is due to soft-decision memory sensing, which uses more than one quantization levels between two adjacent storage states [7]. On the one hand, as the number of quantization levels used between two adjacent storage states increases, the read operations which aim to sense and digitally quantize the threshold voltage of each memory cell are delayed. On the other hand, the number of sensing levels also affects the error correction strength of LDPC code decoding. More sensing levels mean a preciser memory sensing in the context of NAND flash memory, leading to more accurate input probability information of each bit for LDPC code decoding, which improves its error correction capability. Therefore, the relationship between error correction capability and read speed can be explored, leading to strong correlation between RBER and read speed.

Another tradeoff is between RBER and the program speed, in particular, the program step size ΔV_p. Flash programming widely adopts ISPP (incremental step pulse programming) scheme, that is, it uses Fowler-Nordheim (FN) tunneling to increase the threshold voltage V_{th} of flash memory cells by a certain step size i.e. ΔV_p, where ΔV_p directly affects write speed and RBER. On the one hand, larger ΔV_p means fewer steps to the desired level and thus shorter write latency. On the other hand, the margin for tolerating retention errors is reduced as ΔV_p gets larger, leading to higher RBER.

Given that the expected RBER has to be within the error correction capability of the deployed LDPC code, an SSD has to make the tradeoff among RBER, read speed, and write speed before being released. Table I shows an example of three cases based on the NAND flash memory device model described in [18]. The parameters used in the model are trained by the public datasets in [19]. Accordingly, Monte Carlo simulations are carried out to obtain the cell threshold voltage distribution with various ΔV_p values. We then determine the tradeoff among RBER and the read speeds for the corresponding LDPC [7]. From the table, for one page, we may either improve the write speed (i.e., the case A in the table, when the case B is the normal case) or the read speed (i.e., the case C in the table).

TABLE I
AN EXAMPLE OF THE DESIGN TRADEOFFS

	write(μs)	RBER	read(μs)
case A	867	.0104	150
case B	1300	.0076	75
case C	5000	.0044	38

In this work, we focus on read and write latency variation for SSD performance improvement. To the best of our knowledge, this is the first study for the tradeoff among RBER, write speed, and read speed. We believe that it will broaden the design space for optimizing flash I/O performance and endurance.

C. Sources of Device Level Latency Variations

In this paper, we exploit two types of device level variations for performance improvement.

1) Process variation in NAND flash memory: The first type is hardware process variations (PVs), i.e., fabricating flash chips at nano-scale exhibits non-negligible oxide thickness and gate width/length variations. Given flash pages from different memory blocks, (1) they may have different RBER at programming time even if they have the same P/E cycling; (2) they may have different P/E cycling endurance when they are protected with an ECC to correct a fixed number of errors; (3) they may have different charge leaking rates. As a result, two pages, even if they have the same P/E cycling and are programmed with the same RBER, may still have different RBER at read-out time after the same duration time.

Pan *et al.* showed that the RBER of flash blocks follows a log Gaussian distribution [8]. Due to PVs, we classify the flash blocks into strong and normal ones, which accumulate bit errors at slow and normal speeds, respectively.

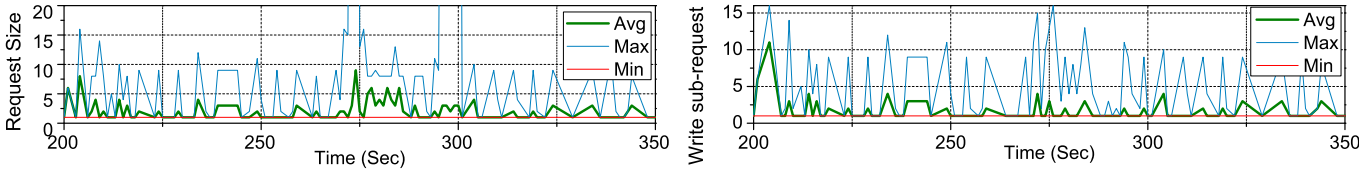
2) Retention age variation in NAND flash memory: The second type is retention age (RA) variation. Retention age is the length of time since a flash cell was programmed [4]. The intervals between page programming time and reading time differ significantly for different read operations. Since a flash page keeps leaking charge after being programmed, the longer the retention age is, the higher the RBER is, and the slower the read speed is. Data retention error is one of the dominant errors in SSDs.

For flash memory based SSDs, incoming data are often sequentially programmed in the physical pages of the active blocks. Given that SSDs usually keep only a small number active blocks, i.e., one to four active blocks [20], [21], the pages in one block are of close retention age. We are to exploit this property in latency aware scheduling.

D. Motivation

We next study the characteristics of workloads and motivate the design of variation-aware I/O scheduling schemes.

An I/O request sent to the SSD control is a quadruple (*ArrivalTime, LPN, Size, Read/Write*), where *ArrivalTime* is the arrival time of the request, *LPN* is the starting logical page



(a) The maximum, average and minimum request sizes

(b) The maximum, average and minimum write sub-request sizes

Fig. 3. The maximum, average and minimum request sizes over 150 seconds for *hm0*. The requests are measured in the number of pages with each page being 4KB.

number of the user data, and *Size* is the length of the data, i.e., the number of flash pages. The SSD controller dispatches the data into the read/write request queue (RRQ/WRQ) for processing. Writes are preferably assigned to idle or less busy flash planes while reads have to fetch from the target pages. The requests may span multiple planes to exploit parallelism for improved performance. In this case, the access in each plane is referred to as a sub-request.

We first compare the request lengths at runtime. Figure 3 reports the average, maximum, and minimum number of flash pages for all the requests and the particular write requests for the workload *hm0* running on a 128GB SSD with 4 KB flash page size. The setting details for the simulated SSD can be found in Section IV. From the figure, there is a significant variation of request length at runtime. Therefore, we adopt the scheduling strategy that prioritizes short job, which reduces the average queuing time. We next study how to exploit device level latency variations to further reduce the request response time.

Observation 1*: The response time of a write request can be reduced if its sub-request with longest pending time can be programmed faster.*

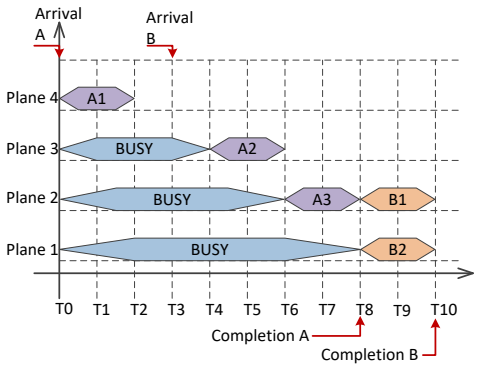
If a write request spans across multiple chips, its response time is determined by the response time of the slowest sub-request, even if other sub-requests can finish early. The response time of a request $T_{WR}^{response}$ consists of the queuing time (i.e., waiting to be serviced), the data transfer time (through the data bus to transfer data from SSD controller to the targeting data register of corresponding plane, e.g., around 5 μs when adopting ONFI/Toggle Interface), and the program time of the sub-request (programming the data cells of the flash, e.g., 600 μs/4KB page).

$$\begin{aligned} T_{WR}^{response} &= \quad Max\left(T_i^{response}\right), i \in [1, M] \\ and \quad T_i^{response} &= \quad T_i^{queuing} + T_i^{transfer} + T_i^{access} \end{aligned} \tag{1}$$

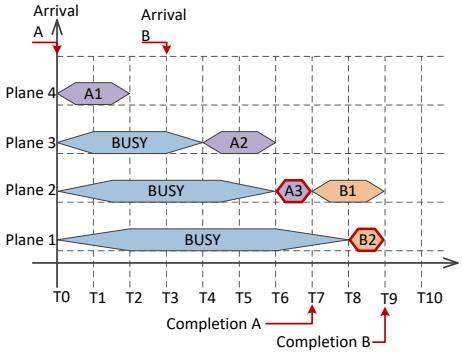
where M is the total number of sub-requests; $T_i^{response}$ is the response time of the *i*-th sub-request, $T_i^{pending}$, $T_i^{transfer}$, and T_i^{access} are the queuing time, transfer time and program time of the *i*-th sub-request, respectively.

Figure 4 illustrates how device level latency variation can assist the I/O scheduling. The two write requests A and B arrive at time T0 and T3 and consist of 3 and 2 sub-requests, respectively. According to CWDP scheduling [14], sub-requests A1-A3 and B1-B2 are mapped to planes 4, 3, 2, and 2, 1, respectively. The CWDP scheduling cannot predict the exact finish time of each sub-request.

Assume it takes two time units to service one write sub-request without exploiting page access latency variations, or



(a) Conventional latency variation-oblivious scheduling



(b) Latency variation-aware scheduling

Fig. 4. Integrating device level latency variation in I/O scheduling helps to reduce the request response time: (a) when adopting latency variation-oblivious scheduling; (b) when adopting latency variation-aware scheduling.

one time unit when the write data are mapped to a strong page. When adopting a conventional device level latency-oblivious scheduling, each write sub-request takes two time units. The two requests complete at T8 and T10, respectively, as shown in Figure 4(a). When adopting device level latency-aware scheduling and having all the sub-requests mapped to use strong pages, as shown in Figure 4(b), the two requests complete at T7 and T9, respectively. However, servicing all sub-requests with strong pages is unnecessary as servicing only A3 and B2 with strong pages would result in the same response time reduction. Given that each plane contains a limited number of strong pages, the latter scheduling is a better choice as it preserves strong pages for servicing other requests.

Observation 2*: Considering retention age variation increases scheduling opportunities for read requests.*

Previous studies [4], [22]–[25] have shown that retention-induced charge leakage is the dominant source of flash memory errors, which lead to significant RBER variation for blocks

with different retention ages. Figure 5 compares the retention age distribution at read-out time from different workloads within one week operations. The read sub-requests include reads from host and from GC execution. The figure shows that different workloads have significant different access patterns. For example, most accessed data in *src21* are within four-day retention age.

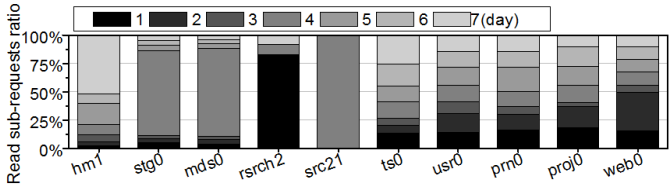


Fig. 5. The retention age distribution for one week operations.

From our previous discussion, flash pages with short retention age tend to have low RBER and fast read speed. Schemes have been proposed to exploit this tradeoff to speed up the read sub-requests. Cai *et al.* showed that lower read-out thresholds can be applied as the actual age of the data increases [4]. Shi *et al.* proposed to adopt higher programming voltages for pages with long retention age [26]. Liu *et al.* proposed to use finer ΔV_p for performance improvement [11].

While preceding discussion shows that exploiting the latency variations at device level helps to improve flash read/write speeds, a limitation of existing schemes is that latency variations are not exposed to the access scheduler. To reduce the average request response time, it is better to track the retention age and prioritize requests that can finish early. That is, exploiting both the SSD PV and the workload characteristics may expose more scheduling opportunities for better I/O performance.

III. The Details of DLV

In this section, we first outline the system architecture of the proposed device level access latency aware I/O scheduling algorithm (DLV) and then elaborate its major components. At last, we analyze the overhead for DLV scheduling.

A. Overview

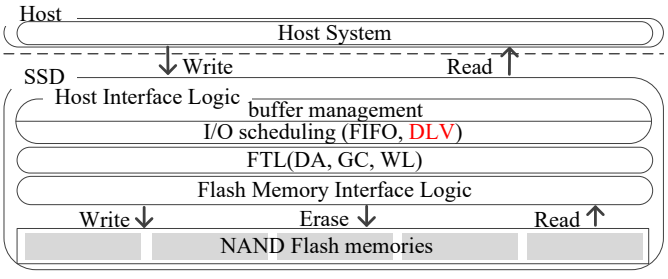


Fig. 6. The DLV-enhanced SSD organization. The baseline adopts FIFO-style I/O scheduler in HIL.

Figure 6 presents an overview of the DLV-enhanced SSD organization, where DLV is integrated in the host interface logic (HIL) at the SSD side. The HIL is responsible for receiving the I/O requests from the host side kernel I/O scheduler, buffering and scheduling them before sending them to the FTL [27]. Comparing to the kernel I/O scheduler, the HIL scheduler exploits the parallelism in the SSD for optimized I/O performance. Conventionally, it adopts FIFO scheduling in the HIL. In this paper, we replace the FIFO scheduler with our proposed DLV scheduler, as shown in Figure 6. DLV tracks not only the device level latency variations, i.e., if there are strong blocks left in a flash plane, but also the request characteristics, i.e., how many data pages a host request needs to access.

DLV is composed of two components, a hotness-aware write scheduling (HWS) and a retention-age- and hotness-aware read scheduling (RRS). HWS identifies request hotness based on their request sizes and serves hot requests with PV-induced fast flash blocks. RRS reduces page read latency by exploiting block retention age and PV in corresponding blocks. Both HWS and RRS schedule fast requests preferentially to minimize the access pending latency.

B. Hotness-aware Write Scheduling

The design goal of hotness-aware write scheduling (HWS) is to integrate PV-introduced write speed variation in I/O scheduling such that the average request response time can be effectively reduced. Intuitively, HWS allocates the data of hot writes to strong flash pages and schedules hot writes with priority.

In this paper, we categorize the hot/cold requests according to their read or write data sizes, i.e., small-sized requests are treated as hot ones. The size-based strategy was widely adopted to classify hot and cold data in recent studies [**?**], [5], [28], [29]. Prioritizing small-sized requests helps to reduce the request waiting time. In addition, many such requests process metadata and thus are critical to system performance. The use of the hot/cold data classification scheme is orthogonal to the design of HWS. While recent designs confirmed the effectiveness of this metric, data hotness can be defined differently [30]–[35], which can also be used to classify hot/cold data in our design.

The HWS scheduling works as follows. To assist I/O scheduling, HWS keeps a plane mapping table (PMT) for each plane in the SSD, which identifies not only its strong pages but also those that are available. It keeps two flags ($PCnt$,$IdleT$) for each plane, where $PCnt$ indicates the number of available strong pages. $IdleT$ indicates the (estimated) time that the plane becomes idle. HWS scheduling consists of three steps.

- First, HWS sets a deadline ($ArrivalTime$+T_d) to each incoming request, where $ArrivalTime$ is the arrival time of the request, and T_d is a fixed time duration. The requests may be scheduled out of order if the current time is before the deadline; otherwise, the requests from host are scheduled using the default FIFO scheduling approach. Given the requests from the host are placed in the queue in order, if the head of the queue has not reached its deadline, the following requests should not either.
  In a serial ATA (SATA) interface, the deadline information is configured by placing '01b' into the Priority (PRIO) field of NCQ commands (READ FPDMA QUEUED and WRITE FPDMA QUEUED) to mark them as isochronous, and then

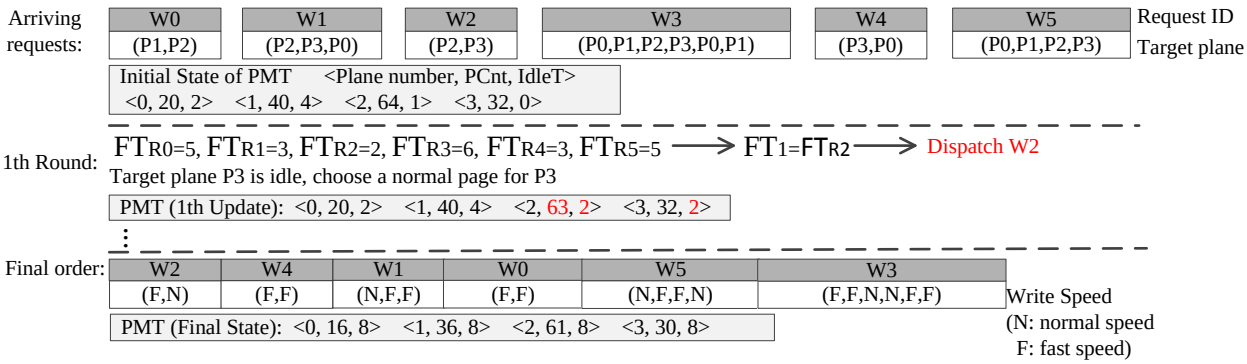


Fig. 7. An example of HWS scheduling.

filling the corresponding deadline values in the Isochronous Command Completion (ICC) field [36]. Note that ICC Bit 7 is cleared to zero so that the time interval is fine-grained. By setting the deadline for each incoming request, we prevent the out-of-order scheduling adopted in HWS from starving large requests that write many pages.

- Second, HWS adopts greedy estimation to determine the request finish time. Since we adopt static CWDP mapping, the target planes are determined by their LPNs. HWS uses the following equation to evaluate the *best* finish time for each sub-request by taking the PMT status of each plane, i.e., the availability of the strong pages and the next idle time, into consideration.
  For each request R_i (1≤i≤N, and N is the total number of write requests to schedule), its finish time FT_{R_i} is estimated by assuming this request is scheduled the next and all its data are mapped to strong pages (if available).

$$FT_{R_i} = MAX_{\forall j}(IdleT(p(R_{ij}))+XferT(R_{ij})+WrT(R_{ij})) \quad (2)$$

  where 1≤j≤M and M is the number of sub-requests that R_i has; R_{ij} is the j-th sub-request of R_i. p(R_{ij}) maps the sub-request to the corresponding plane index. $XferT()$ returns the transfer time; and $WrT()$ returns the programming time of the sub-request. In computing the programming time, HWS estimates the best finish time by assuming all available strong pages (from the plane) can be allocated to service the sub-request. As discussed, $IdleT()$ indicates the next available time that the hardware can start programming. Clearly, the finish time of the request is determined by its slowest sub-request.
  HWS then determines the request whose finish time is the earliest. That is, finding k (1≤i,k≤N) in the queue such that

$$FT_k = MIN_{\forall i}(FT_{R_i}) \quad (3)$$

- Third, HWS schedules the selected request R_k and updates the $IdleT$ for all the planes that it needs to access. In scheduling a sub-request to its corresponding plane, HWS maps the data to strong pages only if otherwise the finish time of that plane is later than FT_k. After scheduling each request, HWS updates ($PCnt$,$IdleT$) accordingly. Note, CWDP statically maps a logic page to the corresponding plane while HWS determines the page type (i.e., strong or normal) within the plane. It is the FTL that finally decides the physical page location within the plane.
  To reduce the runtime overhead, both $XferT()$ and $WrT()$ are computed only once, i.e., at the time when it is inserted in the request queue. The estimation may be optimistic as the number of strong pages reduces. Our experiments show that the impact is negligible.
- Fourth, there are two exceptions. (1) If a plane is being garbage collected, the write requests that demand this plane are blocked. HWS skipped blocked requests, even if their deadlines have passed. (2) If a request does not conflict with any other scheduled requests, i.e., all its requested planes are idle, HWS immediately schedules the request and map its data to normal pages.

Figure 7 presents an example illustrating how HWS works. We assume there are six write requests arrived at time 0, from left to right. Each request consists of one or more sub-requests. For example, request W0 needs to write two pages in plane 1 and 2, respectively. The planes are determined by CWDP scheduling based on the LPN of the request. Assume the SSD has 4 channels, and each channel has one plane.

Assume it takes one time unit to write a page at fast speed and two time units at normal speed. The transfer time is small and thus is neglected in the estimation. By estimating the finish time using Equation 2, i.e. $FT_{R_0} = 5$, $FT_{R_2} = 2$, HWS preferentially dispatches the W2 request. As a comparison, the baseline FIFO scheduler would first dispatch W0, which prolongs the queuing time for subsequent requests. According to Equation 3, $FT_2 = 2$, and mapping the write to a normal page in plane 3 would keep the same finishing time. Therefore, HWS schedules the two sub-requests to a strong page in plane 2 and a normal page in plane 3, respectively. HWS schedules the following requests similarly, i.e., in the order of W2, W4, W1, W0, W5, W3. In this example, the average request response time is 4.67 time units when adopting HWS, and 8.33 time units when adopting the baseline greedy algorithm. In summary, exploiting write latency variation in write request scheduling helps to reduce the average request response time.

C. Retention Age-aware Read Scheduling

The design goal of retention age-aware and hotness-aware read scheduling (RRS) is to integrate RA- and PV- introduced read speed variation in I/O scheduling for improved performance. Since read requests are on the critical path, they

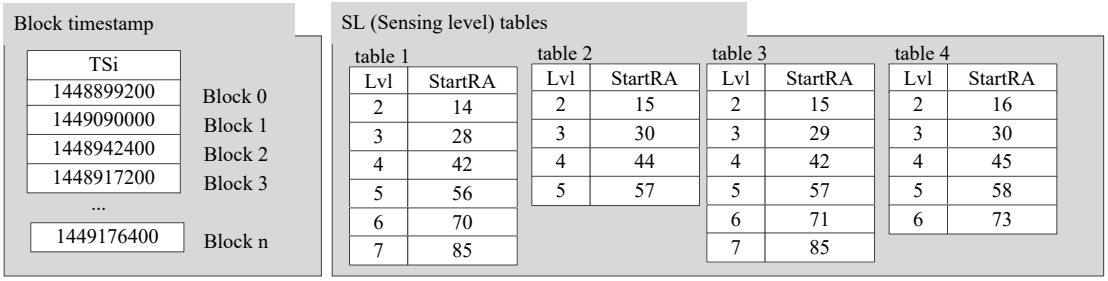


Fig. 8. Block timestamp and SL (sensing level) tables used in RRS scheduling.

are often scheduled before write requests for reducing the average read response time. Intuitively, RRS identifies the read operations that can be further sped up due to retention age difference and schedule them with priorities.

Studies have shown retention errors are dominant errors for read operations in flash-based SSDs [4]. To mitigate such errors, modern SSDs widely adopt strong ECC protection, e.g., LDPC [7]. An LDPC-based read scheme may need to read the page multiple times with different sensing levels. The longer the retention age is, the higher the RBER is, the more times the read has to try, and thus the slower the read speed is.

We next present how RRS exploits the RA- and PV-introduced read speed variation. RRS scheduling is built on top of HWS scheduling, with the difference elaborated as follows.

- RRS scheduling is similar to HWS scheduling. A deadline is added to each incoming request. Requests can be scheduled out of order if their deadlines have not passed, and using FIFO otherwise. Read operations are prioritized such that reads are scheduled before writes if they compete for the same plane.
- For most SSDs, the pages within one flash block are programmed in sequential order. Since only few blocks are kept active in each plane, the pages from one block, e.g., block i, share the similar retention age. RRS approximates the page RA by tracking the one with the longest RA in the block. That is, RRS attaches a timestamp TS_i to block i, where TS_i is the time when the first page of the block was written after its last erase.

  To differentiate the retention time difference among different block types, i.e., strong blocks tend to leak charge at a slow speed, RRS keeps an SL (sensing level) table for each type of blocks. For example, Figure 8 uses four SL tables that correspond to different types of blocks. There are at most 6 tuples (Lvl, $StartRA$) in each SL table — Lvl lists all sensing levels from two to seven when seven is the maximal number of sensing levels in LDPC; $StartRA$ indicates that for $StartRA$ days or longer RA, the read operation should try start with more sensing levels, instead of from only one level. For example, in Figure 8, the 2nd entry of the table 2 is (3, 30). It means that for a strong page belonged to table 2, whose RA is 30 days or longer, we should start a read try with three sensing level. Starting at a large level count reduces the failed tries for these pages as their read only can succeed with three or more sensing levels.
- RRS updates the SL tables heuristically at runtime. That is, given a flash page, RRS determines its block type and RA and find the sensing levels from the SL table. RRS assists the LDPC decoding process by providing appropriate sensing level information. If the read fails, LDPC tries again with more sensing levels till success or fail after reaching a threshold. If the page can be readout with more sensing levels, the SL table is updated so that similar pages (of the same PV and the same RA) do not have to try with fewer levels.

  For the example in Figure 8, when reading strong pages with RA fewer than 30 days, by exploiting the SL table, LDPC would start the decoding with two levels. If a page with 29 days RA fails with two sensing levels but succeeds with three levels, RRS updates the second entry to (29, 3) indicating that, from now on, pages with 29 days or longer RA start with three sensing levels. For conventional LDPC implementations, LDPC decoding always starts with one sensing level, and try more levels after failing the decoding with fewer levels. The conventional LDPC tends to have longer read latency than that of SL table-assisted LDPC.
- To prevent an outlier page from setting the sensing level to a large count, RRS periodically decrements the count in the table. As an example, an outlier 27-day-RA page may succeed after using six sensing levels, which updates the table and demands all 27-day or longer RA pages to read with six sensing levels. This could be pessimistic as most 27-day-RA pages can succeed using two or three sensing levels. RRS addresses this issue by periodically decrementing the $StartRA$ value, e.g., after 100 successful reads. This helps to reset the try count to the appropriate number for most pages in each setting.
- RRS evaluates the request finish time and selects the one that can finish the earliest as the next one to schedule. Comparing to the baseline that prioritizes short jobs, RRS utilizes page read latency difference, which depends on the block type and its corresponding SL in the table.

  A benefit of RRS is that it adapts naturally to read performance degradation with P/E cycling. If a good wear leveling is adopted, all strong pages shall have similar RBER such that we keep one SL table for strong blocks. Otherwise, we can create two or more SL tables for strong blocks with different P/E cycling. For the latter, we use not only the block type and the RA, but also the P/E cycling estimation to determine the appropriate starting sensing levels.

Recently, Du *et al.* proposed LaLDPC [37], a read optimization that is close to our design. LaLDPC attaches a try count to each page to record the number of sensing levels that the

last read operation succeeded. While both LaLDPC and RRS exploit the RA variation for read performance improvement, there exist two differences. One is that LaLDPC does not change request order as RRS does. The other difference is that LaLDPC maintains the count at page granularity. While it achieves finer control, it tends to suffer from archive workloads, i.e., when updating a large number of pages after a long RA (after its last write and without reads in between), it tends to introduce more retries.

D. Overhead

Storage overhead. The proposed DLV approach needs extra storage to save the metadata to enable latency variation aware scheduling. As we show next, the storage overhead introduced in DLV is negligible.

- DLV keeps a bitmap that identifies the strong blocks in the SSD. For the 128GB SSD simulated in our experiment, there are 128K blocks, resulting in 16KB bitmap if using two types of blocks (i.e., strong and normal blocks), or 32KB bitmap if using four types (i.e., three types of strong blocks with different RBER characteristics and one normal type).
- Each plane keeps one 3B counter for each strong page type to track the number of strong pages of that type; and one 3B timestamp to track when the plane becomes idle. Given that there are 128 planes, we need less than 4KB when differentiating four block types.
- Each block type keeps an SL table to track the appropriate sensing levels when reading pages with different RAs. Since an LDPC read scheme uses seven sensing levels in the worst-case [37], an SL table contains six entries. For the SL table, we use 9 bits to record the $StartRA$ field. Recent studies showed that the RA of data of enterprise applications is typically within three months [19], it is sufficient to use 9 bits to record the RA range up to 512 days. The Lvl field is omitted as it is the same as the entry index. Thus, the storage overhead is m $\times$ 6 $\times$ (9+3) bits where m is number of SL tables. We have m being 2 if we just different strong and normal flash blocks, and being up to 10 in our experiments for finer block grouping. The storage overhead is negligible.
- We add a 20-bit timestamp to each flash block in the SSD. It records the time when it was first written after its last erase.

Computation overhead. The computation overhead in DLV comes from out-of-order I/O scheduling, i.e., finding the next request to schedule. For each request, its page access time, i.e., reading or writing the device pages on the corresponding planes, is evaluated only once, DLV then adopts Equations (2) and (3) to find the next request. While the time complexity is O(N), where N is the number of queued I/O requests, the computation is simple and thus is very fast. In our experiments, we observe less than 1% percentage slowdown due to scheduling overhead.

IV. Experiment and Analysis

In this section, we present the experimental methodology and the setting details, and then analyze the results with comparison to the state-of-the-art schemes.

A. Experimental Methodology

We evaluated the proposed scheme using SSDsim, an event-driven simulator, that was widely adopted in the community. The accuracy of SSDSim has been validated via hardware prototyping [38]. We simulated a 128GB SSD with four channels, each of which is connected to four NAND flash memory chips. There are total 256 pages in one block, where each page size is 4KB. Table II lists the setting details.

Page-level FTL is implemented as the default FTL mapping scheme, where the priority order of SSD parallelism levels, Channel-first, Chip-second, Die-third and Plane-fourth (CWD-P) is used for page allocation [12], [14]. And greedy garbage collection and dynamic wear leveling are also implemented to assist mapping management. GC is triggered when the number of free blocks goes below 10% of the total number of blocks. The GC operations are executed in the background in order to minimize the influence on the foreground requests. The percentage of over-provisioning area is set to 7% of the SSD, which is consistent with most SSDs on the market [39]. For the 2bit/cell flash based SSD, the program latency is 600 μs when ΔV_p is 0.3V; the sensing and data transfer latencies are 90 μs and 5 μs, respectively, when adopting LDPC with seven reference voltages [6].

To model the process variation of Flash memory and its impact on RBER distribution, we followed the approach in [8], i.e., we modeled using bounded Gaussian distribution with the mean μ and the standard deviation σ being 3.7×10^{-4} and 9×10^{-5}, respectively.

TABLE II
Main Characteristics of Baseline SSD Configuration

Parameter	Value	Parameter	Value
capacity	128 GB	OP ratio	7%
channel	4	GC	Greedy GC
chip per channel	4	WL	Dynamic WL
die per chip	4	DA scheme	CWDP
plane per die	2	I/O queue depth	64
block per plane	1024	read sensing time	90 μs
page per block	256	program time	600 μs
page size	4 KB	erase time	3000 μs

Workloads. We evaluated the proposed scheme using 20 traces from enterprise applications. The benchmark programs include online transaction processing (OLTP) application traces [40], and enterprise servers traces at Microsoft Research Cambridge [41], [42]. These workloads are widely used in previous studies [12], [38], [43], [44]. Table III summarizes characteristics of our disk traces in terms of write ratio, read ratio, I/O intensity, and duration.

The traces are classified into three groups: write intensive group, read intensive group, and balanced group. A trace is write intensive if its write ratio is greater than 0.75; a trace is read intensive if the read ratio is greater than 0.75; other traces are considered as balanced ones. The RA of most traces is about one week, with the exception that *fin1* and *fin2* are less than one day duration.

Schemes for comparison. To evaluate the overall performance improvement, we implemented and compared the following schemes.

TABLE III
CHARACTERISTICS OF THE EVALUATED I/O WORKLOADS

Workload	Write ratio	Read ratio	Classify	Duration (Days)
fin1	0.77	0.23	write intensive	0.51
mds0	0.88	0.12	write intensive	7
prn0	0.89	0.11	write intensive	7
proj0	0.88	0.12	write intensive	7
rsrch0	0.91	0.09	write intensive	7
src12	0.75	0.25	write intensive	7
src20	0.89	0.11	write intensive	8.09
stg0	0.85	0.15	write intensive	7
ts0	0.82	0.18	write intensive	8.09
fin2	0.18	0.82	read intensive	0.47
hm1	0.05	0.95	read intensive	6.89
mds1	0.07	0.93	read intensive	6.95
proj3	0.05	0.95	read intensive	7
src21	0.02	0.98	read intensive	7.95
web2	0.01	0.99	read intensive	7
hm0	0.64	0.36	balanced	7
rsrch2	0.34	0.66	balanced	6.78
stg1	0.36	0.64	balanced	7
usr0	0.60	0.40	balanced	7
web0	0.70	0.30	balanced	7

Baseline. This is the default setting in SSDsim. It uses FIFO I/O scheduling, CWDP static mapping, and does not exploit device level latency variations.

WOO. The WOO (Write only optimization) scheme implements the hotness-aware write scheduling (HWS) for wirte requests. The read requests are handled the same as the baseline. The deadline of write requests is set to 5s after arrival.

ROO. The ROO (Read only optimization) scheme implements retention age-aware read scheduling (RRS) for read request. The write requests are handled the same as the baseline. The deadline of read requests is set to 500ms after arrival.

DLV. This is the scheme proposed in this paper, which integrates both WOO and ROO optimizations.

Evaluation metrics. We evaluated DLV by measuring the storage bandwidth, the I/O operations per second (IOPS), and the average response time improvement ratio over the baseline. We also evaluated the sensitivity on different hardware settings by varying queue depth and the number of flash chip, and the effectiveness of DLV under different buffering policies.

B. Overall Performance Improvement

Bandwidth comparison. We first compared the storage bandwidth under different scheduling algorithms and summarized the results in Figure 9. On average, DLV achieves 81% bandwidth improvement over $Basline$. The improvements come from the reduction of sensing time for read operation and cell programming time for write operations, and the reduction of the average request queuing time. Similarly, WOO benefits from reduction in sensing time and queuing while ROO benefits from reduction in cell programming and queuing. The improvements for WOO and ROO closely correlate to the read and write ratios in each benchmark. For example, WOO achieves better improvement than ROO for $proj0$ because $proj0$ is a write intensive workload.

IOPS comparison. We compared the average number of finished I/O operations per second (IOPS) under different schemes and summarized the results in Figure 10. On average, DLV achieves 81% IOPS improvement over $Basline$. Similarly as those in Figure 9, WOO performs better for write intensive workloads while ROO performs better for read intensive workloads.

DLV achieves the largest bandwidth and IOPS improvements, i.e., 255.3%, for $proj3$, and the smallest, i.e., 21.4%, for $hm1$.

Average response time comparison. Figure 11 presents the average response time improvements under different schemes. The results are normalized to the baseline. Compared with the traditional baseline, DLV achieves an average of 41.5% improvement. By exploiting strong blocks to reduce programming latencies and recently programmed blocks to reduce sensing latencies, DLV prioritizes jobs that can finish early, which effectively reduces the average response time in servicing I/O accesses. In summary, DLV outperforms WOO and ROO by 22.01%, 17.77% on average, respectively.

To study the efficiency of DLV, Figure 12 compares the cumulative distribution (CDF) of different response time in servicing write and read requests, respectively, under a typical workload *hm0*. From the figure, more requests are serviced quicker in DLV than they are in other schemes. For example, we observed that about 74.3%, 60.1%, 68.9%, and 61.8% of write requests are serviced with quicker than two milliseconds when using DLV, $baseline$, WOO, and ROO, respectively. Since about 36% of total requests in *hm0* are read, as shown in Table III, the CDF of read latency in DLV is similar to that of ROO.

C. Write Performance Improvement

We next analyzed the write improvement in DLV and compared it to the state-of-the-art.

- DLV/WOO — this is the one that disables RRS scheduling in DLV, i.e., it is the WOO scheme.
- PV+$reorder$ — this is the scheme that further disables the processing of cold sub-requests in WOO, i.e., all sub-requests of prioritized requests are allocated to strong pages, if applicable.
- PV+$Allocation$ — this is the scheme in [5]. It does not reorder I/O requests. However, it allocates strong blocks to short jobs to reduce the average queuing time.

Figure 13 summarizes the normalized write performance comparison results. Compared with the traditional baseline scheme, WOO, PV+$reorder$, and PV+$Allocation$ achieve 34.3%, 33.6%, and 29.3% write latency improvements, respectively. The improvement of PV+$Allocation$ comes from placing hot data in strong blocks, which shows 17.6% to 79.7% improvements over the baseline. PV+$reorder$ gains additional improvement by prioritizing hot write requests. The more the requests are prioritized, the larger improvement it achieves. We reported the percentages of prioritized write requests (hot) from different workloads in Figure 14(a). From the two figures, we observed larger the write latency improvements when more write are prioritized. For example, more than 25% requests

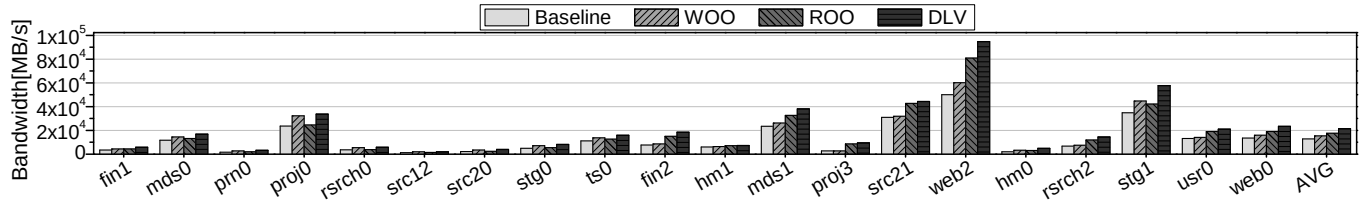


Fig. 9. Comparing the storage bandwidth using different schemes.

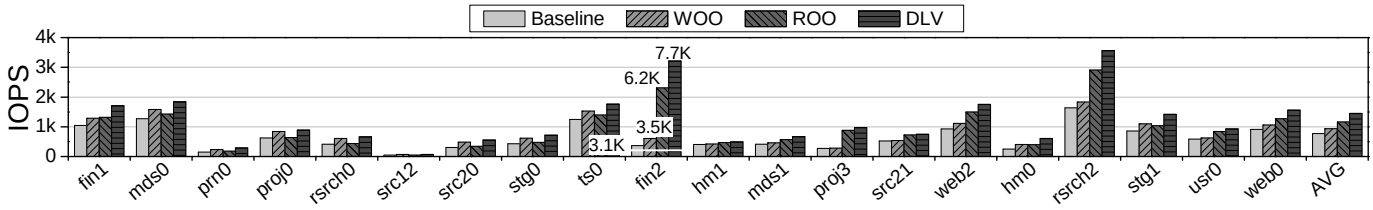


Fig. 10. Comparing the average I/O operations per second (IOPS) using different schemes.

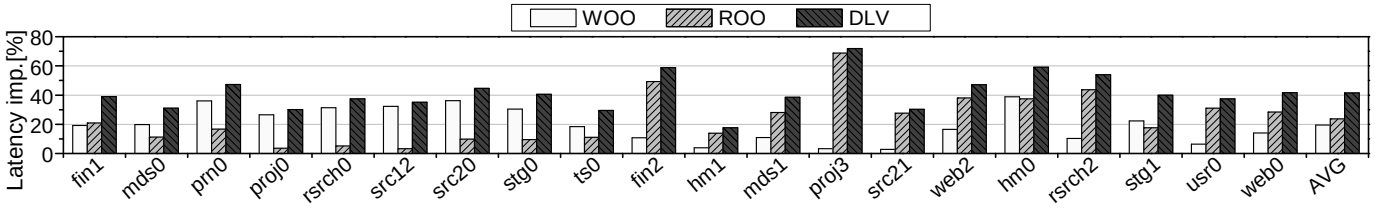


Fig. 11. Comparing the average response time improvement.

Fig. 12. The cumulative distribution function (CDF) of write and read request latency in *hm0*.

are prioritized in *rsrch0* while only a small percentage of writes are prioritized in *mds0*. PV+$reorder$ achieves larger improvement over PV+$Allocation$ for *rsrch0* and comparable improvement for *mds0*.

Comparing to PV+$reorder$, DLV/WOO maps some sub-requests to normal pages when doing so shall not prolong the request finish time. By reserving more strong pages for future requests, DLV/WOO achieves further improvement over PV+$reorder$ for some workloads, e.g., 2.8% more improvement for *proj0*. However, as shown in Figure 13, the impact is insignificant for most workloads. This is because the available strong pages are often abundant for one workload. We expect larger improvements for the system in the long run.

D. Read Performance Improvement

We then analyzed the read improvement in DLV and compared it to the state-of-the-art.

- DLV/ROO — this is the scheme that disables HWS scheduling in DLV, i.e., it is the ROO scheme.
- LV-$LDPC$ — this is the scheme that further disables the reorder of incoming read requests in ROO.
- $LaLDPC$ — this is the scheme in [37], which records the number of sensing levels used by the last read.

Figure 15 compares the normalized read latency improvements using above schemes. On average, DLV/ROO, LV-$LDPC$, $LaLDPC$ improve read performance by 51.8%, 45.5%, and 43.8%, respectively. $LaLDPC$ reduces read sensing time when reading data that has lower retention age. LV-$LDPC$ shares the similar target but reduces the number of retries by using the knowledge when reading similar pages. When reading flash pages that have not been accessed for a long interval, $LaLDPC$ needs more retries to find the appropriate number of sensing levels. DLV/ROO reorders read requests to prioritize hot read requests, which is effective for a number of workloads, e.g., $hm0$.

To fully understand the read performance improvement in DLV, Figure 14 (b) reports the percentages of prioritized read requests. By comparing the results in Figure 15 and Figure 14 (b), we observed that the improvement is larger when the percentage of prioritized reads is higher. This is because workloads with intensive I/Os tend to have larger queuing latency, which was reduced by DLV by scheduling

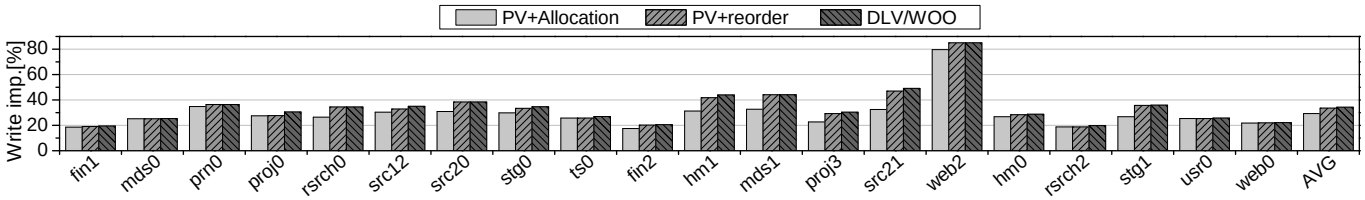
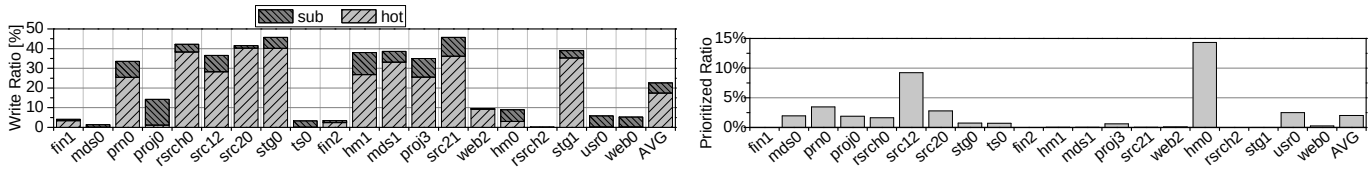


Fig. 13. Analyzing the write latency improvement.



(a) The percentage of prioritized write requests in DLV/WOO

(b) The percentage of prioritized read requests in DLV/ROO

Fig. 14. The percentage of prioritized requests in DLV

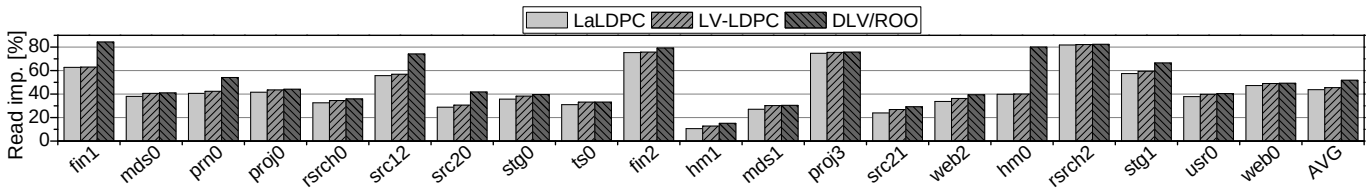


Fig. 15. Analyzing the read latency improvement.

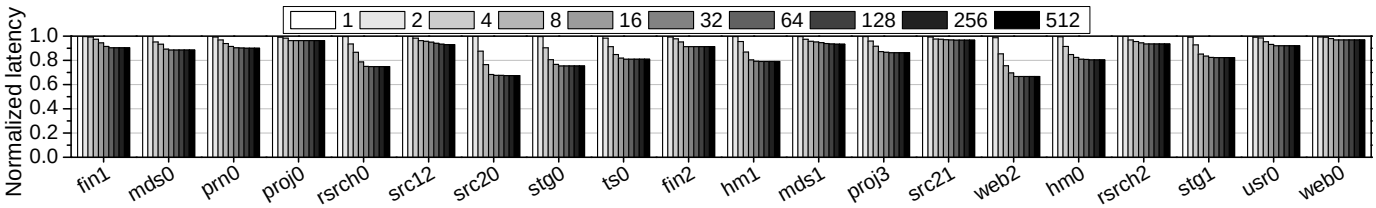


Fig. 16. Studying the impact of different queue lengths, ranging from 1 to 512 (normalized to length=1 setting).

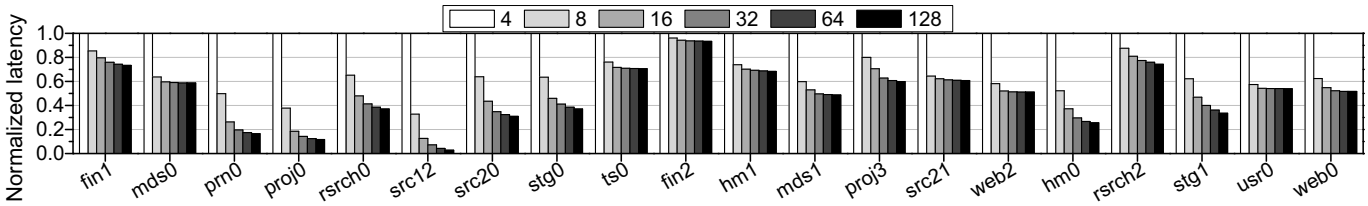


Fig. 17. Studying the impact of different numbers of flash chips, ranging from 4 to 128 (normalized to 4-chip setting).

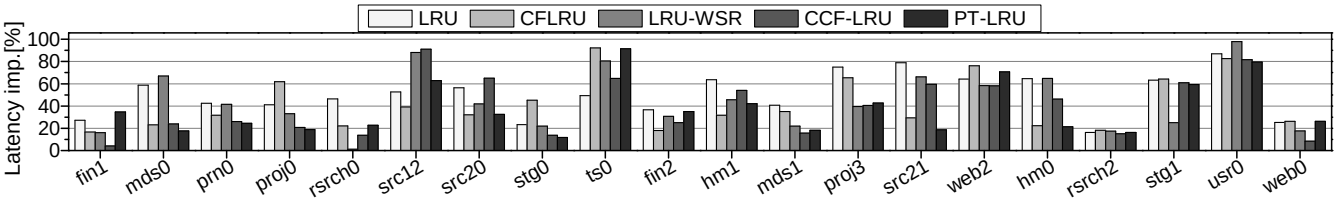


Fig. 18. Studying the effectiveness of DLV under different buffering policies.

short read requests. Overall, the results demonstrate that DLV is effective in reducing the read response time.

E. Sensitivity Analysis

At last, we studied the sensitivity of DLV by varying the I/O queue length and the parallelism granularity in the SSD. We also studied the effectiveness of DLV under different buffering policies. The buffer size is set to 50 MB in the experiments.

Figure 16 compares the average response time when varying the I/O queue length from 1 to 512 in DLV. The results are normalized to the setting with length=1. From the figure, the response time decreases as the length increases. For example, when the I/O queue length grows from 1 to 16, the I/O performance in *rsrch0* shows 21.3% improvement. This is because there are more scheduling opportunities with more requests in the I/O queue. Also from the figure, the performance improvement saturates when the length reaches a threshold, i.e., 16 in the experiment.

Figure 17 compares the average response time when the number of flash chips varies from 4 to 128. The results are normalized to the setting with 4 chips. From the figure, the average response time decreases when the number of chips increases. This is because more I/O requests can be serviced in parallel when there are more chips, which reduces the queuing time in servicing I/O requests. However, the performance improvements show large variations. By comparing the results in Figure 10, the response time reduction is larger when the IOPS is lower. For example, for *src12*, the average response time reduces by 67.31% from 8 chips to 4 chips. It has few IOPS operations.

Figure 18 compares the average response time improvement of the proposed DLV scheme, when adopting different buffering schemes, i.e., LRU, CFLRU [45], LRU-WSR [46], CCF-LRU [47] and PT-LRU [48], respectively. The results are normalized to the baseline. In summary, compared with the baseline case, on average, the DLV scheme achieves 50.67%, 41.67%, 43.81%, 39.41% and 37.34% I/O performance improvement under LRU, CFLRU, LRU-WSR, CCF-LRU and PT-LRU schemes, respectively. Thus, the proposed DLV scheduler remains effective under different buffer policies.

V. Related Work

In this section, we will discuss additional related work in exploiting the tradeoff between RBER, read speed and write speed.

PV-introduced latency variations. Recent studies exploited PV in SSDs for better wear-leveling, i.e., use the strong blocks within SSDs to maximize lifetime. Pan et al. [8] extended flash memory lifetime by using RBER statistics as the measurement of memory block wear-out pace for the wear-leveling algorithm. Woo et al. [9] introduced a new measure that predicts the remaining lifetime of a flash block more accurately than the erase count based on the findings that all the flash blocks could survive much longer than the guaranteed numbers and the number of P/E cycles vary significantly among blocks. Shi et al. [5] exploited PV for better tradeoff between RBER and write speed. They used coarser ΔV_p for strong pages that do not accumulate errors as fast as normal pages and allocated strong blocks to hotter data. In this work, our hotness-aware write scheduling algorithm also takes advantage of strong blocks based on the PV-aware data allocation.

RA-introduced latency variations. The impact of data retention skew on storage system performance has been exploited to minimize refresh cost. For example, Luo et al. [32] introduced a write-hotness aware retention management policy for NAND flash memory to relax the flash retention time for SSD data that are frequently written. Di et al. [10] proposed a refresh minimization method by writing the data of long retention time requirement into high endurance blocks. Recent studies adjusted cell programming/read-out parameters for improved performance. For example, Cai et al. [4] presented a retention optimized reading (ROR) method that periodically learns a tight upper bound and applies the optimal read reference voltage for each flash memory block online. Shi et al. [26] proposed a retention trimming approach for wearing reduction by decreasing programming voltages when the estimated retention time is lower. Liu et al. [11] achieved write response time speedup based on the estimated retention time, by adapting both the programming step size ΔV_p and ECC strength. Du *et al.* proposed LaLDPC [37] to reduce read sensing time by optimizing sensing quantization levels, avoiding unnecessary read retries. In this work, our retention-aware read scheduling algorithm takes advantage of data with low retention age based on the retention-aware ECC adaptation.

When using PV-based fast write and retention age-based fast read, the requests are accelerated in varying degrees, which inevitably lead to the significant read and write latency variations. I/O scheduler may exploit the speed variations to improve read/write performance. While most flash-based I/O schedulers focused on how to reduce the access conflict and improve chip utilization by exploiting the internal parallelism of SSDs [6], [27], [49], [50], we focus on the reduction of access latency by taking advantage of latency variations.

VI. Conclusion

In this paper, we proposed a device level latency variation aware I/O scheduling algorithm DLV for NAND flash-based SSDs. In addition to exploiting the latency variation among blocks to speed up read and write accesses, DLV reschedules I/O requests and prioritizes the requests that can finish early, which effectively reduces the requests pending time and the I/O requests response time. Our experimental results show that the proposed technique achieves 41.5% I/O performance improvement on average.

Acknowledgment

The authors would like to thank the anonymous reviewers for their detailed and thoughtful feedback which improved the quality of this paper significantly. We would also like to thank our shepherd for the conference version of this paper, Peter Desnoyers, as well as the anonymous MSST reviewers. The work of Jinhua Cui was done when she was a visiting student at the University of Pittsburgh, sponsored by the Chinese Scholarship Council (CSC).

REFERENCES

[1] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. Sfs: random write considered harmful in solid state drives. In *FAST*, page 12, 2012.

[2] Kin-Chu Ho, Po-Chao Fang, Hsiang-Pang Li, Cheng-Yuan Michael Wang, and Hsie-Chia Chang. A 45nm 6b/cell charge-trapping flash memory using ldpc-based ecc and drift-immune soft-sensing engine. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2013 IEEE International*, pages 222–223. IEEE, 2013.

[3] Scott Zuloaga, Rui Liu, Pai-Yu Chen, and Shimeng Yu. Scaling 2-layer rram cross-point array towards 10 nm node: A device-circuit co-design. In *Circuits and Systems (ISCAS), 2015 IEEE International Symposium on*, pages 193–196. IEEE, 2015.

[4] Yu Cai, Yixin Luo, Erich F Haratsch, Ken Mai, and Onur Mutlu. Data retention in mlc nand flash memory: Characterization, optimization, and recovery. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 551–563. IEEE, 2015.

[5] Liang Shi, Yejia Di, Mengying Zhao, Chun Jason Xue, Kaijie Wu, and Edwin H-M Sha. Exploiting process variation for write performance improvement on nand flash memory storage systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(1):334–337, 2016.

[6] Qiao Li, Liang Shi, Congming Gao, Kaijie Wu, Chun Jason Xue, Qingfeng Zhuge, and Edwin H-M Sha. Maximizing io performance via conflict reduction for flash memory storage systems. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 904–907. EDA Consortium, 2015.

[7] Kai Zhao, Wenzhe Zhao, Hongbin Sun, Tong Zhang, Xiaodong Zhang, and Nanning Zheng. Ldpc-in-ssd: making advanced error correction codes work effectively in solid state drives. In *FAST*, volume 13, pages 244–256, 2013.

[8] Yangyang Pan, Guiqiang Dong, and Tong Zhang. Error rate-based wear-leveling for nand flash memory at highly scaled technology nodes. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(7):1350–1354, 2013.

[9] Yeong-Jae Woo and Jin-Soo Kim. Diversifying wear index for mlc nand flash memory to extend the lifetime of ssds. In *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, pages 1–10. IEEE, 2013.

[10] Yejia Di, Liang Shi, Kaijie Wu, and Chun Jason Xue. Exploiting process variation for retention induced refresh minimization on flash memory. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pages 391–396. IEEE, 2016.

[11] Ren-Shuo Liu, Chia-Lin Yang, and Wei Wu. Optimizing nand flash-based ssds via retention relaxation. *Target*, 11(10):00, 2012.

[12] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Chao Ren. Exploring and exploiting the multilevel parallelism inside ssds for improved performance and endurance. *IEEE Transactions on Computers*, 62(6):1141–1155, 2013.

[13] Nima Elyasi, Mohammad Arjomand, Anand Sivasubramaniam, Mahmut T Kandemir, Chita R Das, and Myoungsoo Jung. Exploiting intra-request slack to improve ssd performance. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 375–388. ACM, 2017.

[14] Ji-Yong Shin, Zeng-Lin Xia, Ning-Yi Xu, Rui Gao, Xiong-Fei Cai, Seungryoul Maeng, and Feng-Hsiung Hsu. Ftl design exploration in reconfigurable high-performance ssd for server applications. In *Proceedings of the 23rd international conference on Supercomputing*, pages 338–349. ACM, 2009.

[15] Seokhei Cho, Changhyun Park, Youjip Won, Sooyong Kang, Jaehyuk Cha, Sungroh Yoon, and Jongmoo Choi. Design tradeoffs of ssds: From energy consumptions perspective. *ACM Transactions on Storage (TOS)*, 11(2):8, 2015.

[16] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. *Proceedings of the VLDB Endowment*, 5(4):286–297, 2011.

[17] Chulbum Kim, Jinho Ryu, Taesung Lee, Hyunggon Kim, Jaewoo Lim, Jaeyong Jeong, Seonghwan Seo, Hongsoo Jeon, Bokeun Kim, Inyoul Lee, et al. A 21 nm high performance 64 gb mlc nand flash memory with 400 mb/s asynchronous toggle ddr interface. *IEEE Journal of Solid-State Circuits*, 47(4):981–989, 2012.

[18] Yangyang Pan, Guiqiang Dong, and Tong Zhang. Exploiting memory device wear-out dynamics to improve nand flash memory system performance. In *FAST*, volume 11, pages 18–18, 2011.

[19] Hairong Sun, Pete Grayson, and Bob Wood. Quantifying reliability of solid-state storage from multiple aspects. *Proc. SNAPI*, 11, 2011.

[20] Laura M Grupp, John D Davis, and Steven Swanson. The harey tortoise: Managing heterogeneous write performance in ssds. In *USENIX Annual Technical Conference*, pages 79–90, 2013.

[21] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed solid-state drive. In *HotStorage*, 2014.

[22] Shuhei Tanakamaru, Chinglin Hung, Atsushi Esumi, Mitsuyoshi Ito, Kai Li, and Ken Takeuchi. 95%-lower-ber 43%-lower-power intelligent solid-state drive (ssd) with asymmetric coding and stripe pattern elimination algorithm. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 204–206. IEEE, 2011.

[23] Yu Cai, Erich F Haratsch, Onur Mutlu, and Ken Mai. Error patterns in mlc nand flash memory: Measurement, characterization, and analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 521–526. EDA Consortium, 2012.

[24] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F Haratsch, Adrian Cristal, Osman S Unsal, and Ken Mai. Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 94–101. IEEE, 2012.

[25] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F Haratsch, Adrian Crista, Osman S Unsal, and Ken Mai. Error analysis and retention-aware error management for nand flash memory. *Intel Technology Journal*, 17(1), 2013.

[26] Liang Shi, Kaijie Wu, Mengying Zhao, Chun Jason Xue, Duo Liu, and Edwin H-M Sha. Retention trimming for lifetime improvement of flash memory storage systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(1):58–71, 2016.

[27] Myoungsoo Jung, Wonil Choi, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut T Kandemir. Hios: A host interface i/o scheduler for solid state disks. *ACM SIGARCH Computer Architecture News*, 42(3):289–300, 2014.

[28] Gala Yadgar, Eitan Yaakobi, and Assaf Schuster. Write once, get 50% free: Saving ssd erase costs using wom codes. In *FAST*, pages 257–271, 2015.

[29] Li-Pin Chang. Hybrid solid-state disks: combining heterogeneous nand flash in large ssds. In *Design Automation Conference, 2008. ASPDAC 2008. Asia and South Pacific*, pages 428–433. IEEE, 2008.

[30] Xavier Jimenez, David Novo, and Paolo Ienne. Wear unleveling: improving nand flash lifetime by balancing page endurance. In *FAST*, volume 14, pages 47–59, 2014.

[31] Saher Odeh and Yuval Cassuto. Nand flash architectures reducing write amplification through multi-write codes. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*, pages 1–10. IEEE, 2014.

[32] Yixin Luo, Yu Cai, Saugata Ghose, Jongmoo Choi, and Onur Mutlu. Warm: Improving nand flash memory lifetime with write-hotness aware retention management. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pages 1–14. IEEE, 2015.

[33] Renhai Chen, Yi Wang, Duo Liu, Zili Shao, and Song Jiang. Heating dispersal for self-healing nand flash memory. *IEEE Transactions on Computers*, 66(2):361–367, 2017.

[34] Renhai Chen, Yi Wang, Jingtong Hu, Duo Liu, Zili Shao, and Yong Guan. Image-content-aware i/o optimization for mobile virtualization. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(1):12, 2016.

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

[36] Donald Joseph Molaro, Frank Rui-Feng Chu, Jorge Campello De Souza, Atsushi Kanamaru, Tadahisa Kawa, and Damien CD Le Moal. Data storage devices accepting queued commands having deadlines, September 17 2013. US Patent 8,539,176.

[37] Yajuan Du, Deqing Zou, Qiao Li, Liang Shi, Hai Jin, and Chun Jason Xue. Laldpc: Latency-aware ldpc for read performance improvement of solid state drives. MSST, 2017.

[38] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the international conference on Supercomputing*, pages 96–107. ACM, 2011.

[39] Kent Smith. Understanding ssd over-provisioning. *EDN Network*, 2013.

[40] OLTP Application. I/o. umass trace repository, 2007.

[41] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to ssds: analysis of

tradeoffs. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 145–158. ACM, 2009.

[42] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):10, 2008.

[43] Myoungsoo Jung and Mahmut T Kandemir. An evaluation of different page allocation strategies on high-speed ssds. In *HotStorage*, 2012.

[44] Jinhua Cui, Weiguo Wu, Xingjun Zhang, Jianhang Huang, and Yinfeng Wang. Exploiting latency variation for access conflict reduction of nand flash memory. In *Mass Storage Systems and Technologies (MSST), 2016 32nd Symposium on*, pages 1–7. IEEE, 2016.

[45] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. Cflru: a replacement algorithm for flash memory. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 234–241. ACM, 2006.

[46] Hoyoung Jung, Hyoki Shim, Sungmin Park, Sooyong Kang, and Jaehyuk Cha. Lru-wsr: integration of lru and writes sequence reordering for flash memory. *IEEE Transactions on Consumer Electronics*, 54(3), 2008.

[47] Zhi Li, Peiquan Jin, Xuan Su, Kai Cui, and Lihua Yue. Ccf-lru: a new buffer replacement algorithm for flash memory. *IEEE Transactions on Consumer Electronics*, 55(3), 2009.

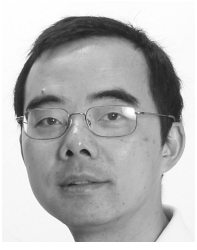
[48] Jinhua Cui, Weiguo Wu, Yinfeng Wang, and Zhangfeng Duan. Pt-lru: a probabilistic page replacement algorithm for nand flash-based consumer electronics. *IEEE Transactions on Consumer Electronics*, 60(4):614–622, 2014.

[49] Congming Gao, Liang Shi, Mengying Zhao, Chun Jason Xue, Kaijie Wu, and Edwin H-M Sha. Exploiting parallelism in i/o scheduling for access conflict minimization in flash-based solid state drives. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*, pages 1–11. IEEE, 2014.

[50] Bo Mao and Suzhen Wu. Exploiting request characteristics and internal parallelism to improve ssd performance. In *Computer Design (ICCD), 2015 33rd IEEE International Conference on*, pages 447–450. IEEE, 2015.

Jinhua Cui received the B.S. degree from Southwest University, Chongqing, China, in 2012. She has also been a visiting student in the Computer Science Department at University of Pittsburgh, Pittsburgh, PA, USA, during the period between Aug. 2016 and Aug. 2017, co-advised by Professor Youtao Zhang and Professor Jun Yang.

She is currently a Ph.D. candidate of Xi'an Jiaotong University, Xi'an, Shaanxi, China, under the guidance of Professor Weiguo Wu. Her current research interests include NAND flash memory, approximate storage, big data, cloud computing, HDFS, and database index.

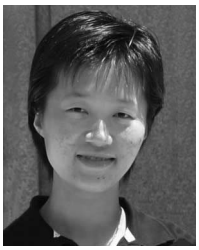


Youtao Zhang received the Ph.D. degree in computer science from the University of Arizona, Tucson, AZ, USA, in 2002, and the B.S. and M.E. degrees from Nanjing University, Nanjing, China, in 1993 and 1996, respectively.

He is currently an Associate Professor of Computer Science, University of Pittsburgh, Pittsburgh, PA, USA. His current research interests include computer architecture, program analysis, and optimization. Prof. Zhang was the recipient of the U.S. National Science Foundation Career Award in 2005. He is a member of ACM.

Weiguo Wu received his B.S., M.S. and Ph.D. degrees in computer science, all from Xi'an Jiaotong University, Shaanxi, China, in 1986, 1993 and 2006, respectively.

He is currently a Professor with school of Electronic and Information Engineering, Xi'an Jiaotong University, Shaanxi, China. He is also the deputy director of the Neo Computer Institute in Xi'an Jiaotong University, a senior member of Chinese Computer Federation, a standing committee member of High Performance Computing Clusters in Chinese Computer Federation, and a standing committee member of Microcomputer (Embedded System) in Chinese Computer Federation, the director of Shaanxi Computer Federation. His research interests include high performance computing, computer network, embedded system, VHDL, cloud computing, storage system, etc.



Jun Yang received the B.S. degree in computer science from Nanjing University, China, in 1995, the M.A. degree in mathematical sciences from Worcester Polytechnic Institute, Massachusetts, in 1997, the PhD degree in computer science from the University of Arizona in 2002.

She is a Professor with the Electrical and Computer Engineering Department, University of Pittsburgh, Pennsylvania. She is the recipient of US NSF Career Award in 2008. Her research interests include low power, and temperature-aware micro-architecture designs, new memory technologies, and networks-on-chip.

Yinfeng Wang received his Ph.D. degrees in computer science from Xi'an Jiaotong University, Shaanxi, China, in 2007. He was a Post-doctoral from Hong Kong baptist university, Hong Kong, China, a associate research fellow in the department of Electronic & Computer Engineering at the Hong Kong Polytechnic University, a postdoctoral research scholar at the University of Hong Kong, during the period between 2008 and 2011.

He is an Associate Professor with the Department of Software Engineering, Shenzhen Institute of Information Technology. His major research fields are in-memory database and cloud computing, etc.

Jianhang Huang received the B.S. and M.S. degrees from Xi'an Jiaotong University, Shaanxi, China, in 2012 and 2015, respectively. He was a software engineer of E-commerce Search, Sogou Technology Development Co.,Ltd, Beijing, China, during the period between 2015 and 2017.

He is currently a PHD candidate of Xi'an Jiaotong University, under the guidance of Prof. W. Wu. His major research interests include cluster rendering, distributed tracing and flash memory based storage system.