Parameter-Aware I/O Management for Solid State Disks (SSDs)

Jaehong Kim, Sangwon Seo, Dawoon Jung, Jin-Soo Kim, *Member*, *IEEE*, and Jaehyuk Huh, *Member*, *IEEE*

Abstract—Solid state disks (SSDs) have many advantages over hard disk drives, including better reliability, performance, durability, and power efficiency. However, the characteristics of SSDs are completely different from those of hard disk drives with rotating disks. To achieve the full potential performance improvement with SSDs, operating systems or applications must understand the critical performance parameters of SSDs to fine-tune their accesses. However, the internal hardware and software organizations vary significantly among SSDs and, thus, each SSD exhibits different parameters which influence the overall performance. In this paper, we propose a methodology which can extract several essential parameters affecting the performance of SSDs, and apply the extracted parameters to SSD systems for performance improvement. The target parameters of SSDs considered in this paper are 1) the size of read/write unit, 2) the size of erase unit, 3) the size of read buffer, and 4) the size of write buffer. We modify two operating system components to optimize their operations with the SSD parameters. The experimental results show that such parameter-aware management leads to significant performance improvements for large file accesses by performing SSD-specific optimizations.

Index Terms—Solid state disk(SSD), measurement, storage management, and operating systems.

1 Introduction

A solid state disk (SSD) is a data storage device that uses solid state memory to store persistent data. In particular, we use the term SSDs to denote SSDs consisting of NAND flash memory, as this type of SSDs is being widely used in laptop, desktop, and enterprise server markets. Compared with conventional hard disk drives (HDDs), SSDs offer several favorable features. Most notably, the read/write bandwidth of SSDs is higher than that of HDDs, and SSDs have no seek time since they have no moving parts such as arms and spinning platters. The absence of mechanical components also provide higher durability against shock, vibration, and operating temperatures. In addition, SSDs consume less power than HDDs [1].

During the past few decades, the storage subsystem has been one of the main targets for performance optimization in computing systems. To improve the performance of the storage system, numerous studies have been conducted which use the knowledge of internal performance parameters of hard disks such as sector size, seek time, rotational delay, and geometry information. In particular, many researchers have suggested advanced optimization techniques using various disk parameters such as track boundaries, zone information, and the position of the disk head [2], [3], [4]. Understanding these parameters also helps to model and analyze disk performance more accurately [5].

However, SSDs have different performance parameters compared with HDDs due to the difference in the characteristics of underlying storage media [6]. For example, the unit size of read/write operations in SSDs, which we call *the clustered page size*, is usually greater than the traditional sector size used in HDDs. Therefore, if the size of write requests is smaller than the clustered page size, the rest of the data should be read from the original data, incurring the additional overhead of a read operation [7]. Issuing read/write requests in a multiple of the cluster page size can avoid this overhead. However, the actual value of such a parameter varies depending on the type of NAND flash memory employed and the internal architecture of SSDs. SSD manufacturers have been reluctant to reveal such performance parameters of SSDs.

In this paper, we propose a methodology which can extract several essential parameters affecting the performance of SSDs and apply them to SSD systems for performance improvement. The parameters considered in this paper include the size of read/write unit, the size of erase unit, the size of read buffer, and the size of write buffer. To extract these parameters, we have developed a set of microbenchmarks which issue a sequence of read or write requests and measure the access latencies. By varying the request size and the access pattern, the important performance parameters of a commercial SSD can be successfully estimated.

The extracted performance parameters of SSDs can be used for various purposes. The parameters can fine-tune the model of a given real SSD. For example, when simulating an

- *J. Kim, S. Seo, and J. Huh are with the Department of Computer Science, Korea Advanced Institute of Science and Technology, 335 Gwahak-ro (373-1 Guseong-dong), Yuseong-gu, Daejeon 305-701, Republic of Korea. E-mail: {jaehong, swseo}@camars.kaist.ac.kr, jhuh@cs.kaist.ac.kr.*
- *D. Jung is with Samsung Electronics, Samsung Semiconductor R&D Center, San #16 Banwol-Dong Hwasung Gyeonggi-Do, Republic of Korea. E-mail: dw0904.jung@samsung.com.*
- *J.-S. Kim is with the School of Information and Communication Engineering, Sung Kyun Kwan University, 300 Cheoncheon-dong Jangan-gu, Suwon 440-746, Republic of Korea. E-mail: jinsookim@skku.edu.*

Manuscript received 14 June 2010; revised 29 Nov. 2010; accepted 6 Mar. 2011; published online 22 Mar. 2011.
Recommended for acceptance by E. Miller.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2010-06-0344.
Digital Object Identifier no. 10.1109/TC.2011.76.

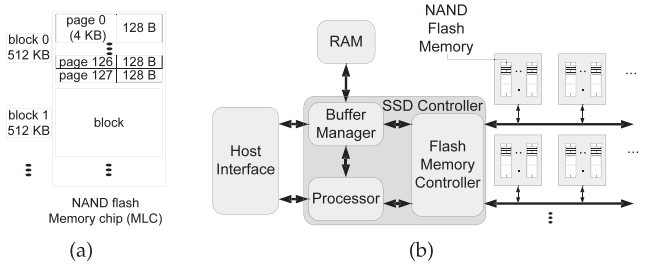


Fig. 1. NAND flash memory internals (a) and the block diagram of an SSD (b).

SSD exploiting multichip parallelism, performance parameters such as the size of a clustered page can be used to model the channels with a striping technique [8]. In addition, file systems can be optimized to model and exploit the characteristics of SSDs, with the critical performance parameters such as the clustered block size [9].

In this paper, to demonstrate the application of the extracted SSD parameters, we redesign two I/O components of a Linux operating system. We modify the generic block layer and I/O scheduler to parameter-aware components. Although the benefits of the optimizations vary by SSD designs and the characteristics of workloads, the I/O optimizations with the SSD performance parameters result in up to 24 percent bandwidth improvement with the postmark benchmark, and up to 321 percent improvement with the filebench benchmark.

The rest of the paper is organized as follows: Section 2 overviews the characteristics of NAND flash memory and SSDs, and Section 3 presents related work. Section 4 describes the detailed methodology for extracting performance parameters in SSDs and presents the results. Section 5 describes the designs of parameter-aware I/O components for a commercial operating system. Section 6 shows the performance evaluation of the proposed parameter-aware system, and Section 7 concludes the paper.

2 Background

2.1 NAND Flash Memory

NAND flash memory is a non-volatile semiconductor device. A NAND flash memory chip consists of a number of erase units, called *blocks*, and a block is usually comprised of 64 or 128 pages. A *page* is a unit of read and write operations. Each page in turn consists of data area and spare area. The data area accommodates user or application contents, while the spare area contains management information such as ECCs (error correction codes) and bad block indicators. The data area size is usually 2 KB or 4 KB, and the spare size is 64 B (for 2 KB data) or 128 B (for 4 KB data). Fig. 1 illustrates the organization of NAND flash where a block contains 128 4-KB-pages.

NAND flash memory is different from DRAMs and HDDs in a number of aspects. First, the latency of read and write operations is asymmetric. Second, NAND flash memory does not allow in-place update; once a page is filled with data, the block containing the page should be erased before new data are written to the page. Moreover, the lifetime of NAND flash memory is limited to 10,000-100,000 program/erase cycles [10].

2.2 Solid State Disks (SSDs)

A typical SSD is composed of a host interface control logic, an array of NAND flash memory, a RAM, and an SSD controller, as shown in Fig. 1b. The host interface control logic transfers command and data from/to the host via the USB, PATA, or SATA protocol. The main role of the SSD controller is to translate read/write requests into flash memory operations. During handling read/write requests, the controller exploits RAM to temporarily buffer write requests or accessed data. The entire operations are governed by a firmware, usually called a flash translation layer (FTL) [11], [12], run by the SSD controller.

To increase the read/write bandwidth of SSDs, many SSDs use an interleaving technique in the hardware logic and the firmware. For example, a write (or program) operation is accomplished by the following two steps: 1) loading data to the internal page register of a NAND chip, and 2) programming the loaded data into the appropriate NAND flash cells. Because the data programming time is longer than the data loading time, data can be loaded to another NAND chip during the data programming time. To increase the bandwidth, the interleaving technique exploits the parallelism of accessing multiple NAND chips simultaneously. If there are multiple independent channels, the read/write bandwidth of SSDs can be accelerated further by exploiting interchannel and intrachannel parallelism [13], [14].

2.3 Flash Translation Layer (FTL)

FTL is the main control software in SSDs that gives an illusion of general hard disks, hiding the unique characteristics of NAND flash memory from the host. One primary technique of FTL to achieve this is to map Logical Block Addresses (LBA) from the host to physical addresses in flash memory. When a write request arrives, FTL writes the arrived data to a page in an erased state and updates the mapping information to point to the location of the up-to-date physical page. The old page that has the original copy of data becomes unreachable and obsolete. A read request is served by reading the page indicated by the mapping information.

Another important function of FTL is *garbage collection*. Garbage collection is a process that erases *dirty* blocks which have obsolete pages and recycles these pages. If a block selected to be erased has valid pages, those pages are migrated to other blocks before erasing the block.

According to the granularity of mapping information, FTLs are classified into page-mapping FTLs [15] and block-mapping FTLs. In page-mapping FTLs, the granularity of mapping information is a page, while that of block-mapping FTLs is a block. As the size of a block is much larger than that of a page, block-mapping FTL usually requires less memory space than page-mapping FTL to keep the mapping information in memory. Recently, several hybrid-mapping FTLs have been proposed. These hybrid-mapping FTLs aim to improve the performance by offering more flexible mapping, while keeping the amount of mapping information low [16], [17].

3 Related Work

Extracting performance-critical parameters for HDDs has been widely studied for designing sophisticated disk

scheduling algorithms [18], [19], [20] and characterizing the performance of HDDs to build detailed disk simulators [21], [22], [23], [24]. However, as SSDs have completely a different architecture compared to HDDs, the methodology for extracting parameters in HDDs cannot be used for SSDs. Our work introduces a methodology for extracting the performance parameters of SSDs and show the effects of parameter-aware system design. To the best of our knowledge, our work is among the first to examine the performance parameters obtained from commercial SSDs and apply them to SSD systems.

Agrawal et al. [7] provide a good overview of the SSD architecture and present various tradeoffs in designing SSDs. Using a simulator, they explore the benefits and potential drawbacks of various design techniques by varying performance parameters such as the page size, the degree of overprovisioning, the amount of ganging, the range of striping, etc. Their study indicates that such parameters affect the performance of SSDs significantly.

Adrian et al. have developed Gordon, a flash memory-based cluster architecture for large-scale data-intensive applications [25]. The architecture of Gordon is similar to that of SSDs in that it uses NAND flash memory chips and an FTL-like flash memory controller. To achieve the high I/O performance of data-intensive work, they tune performance parameters such as the clustered page size, showing the performance sensitivity of flash-based storage to several critical parameters.

Our methodology for extracting performance parameters is a kind of the gray-box approach [26], [27]. The gray-box approach is a methodology that acquires information regarding a target system, with minimum partial knowledge on the system. This approach is different from the white-box approach or the black-box approach, which has the full knowledge or no knowledge on the target system, respectively. Instead, the gray-box approach assumes some knowledge of the algorithms or architectures adopted in the system.

Yotov et al. have applied the gray-box approach to the memory hierarchy [28]. They introduce a methodology which extracts several memory hierarchy parameters in order to optimize the system performance under a given platform. Timothy et al. have also characterized RAID storage array using the gray-box approach [29]. They employ several algorithms to determine the critical parameters of a RAID system, including the number of disks, chunk size, the level of redundancy, and layout scheme. For disk characteristics, Talagala et al. have proposed three disk microbenchmarks that obtain a subset of disk geometry and performance parameters in an efficient and accurate manner [30]. For storage clusters, Gunawi et al. have inferred the structure and policy of software systems for large-scale storage clusters by using standard tools for tracing both the disk and network traffics [31]. To find the buffer-cache replacement of operating systems, Burnett et al. have introduced a simple fingerprinting tool which identifies popular replacement algorithms [32]. Sivathanu et al. [33] have proposed the concept of a semantically-smart disk system that has detailed knowledge of how the file system uses the disk system. To obtain this knowledge automatically, they used a tool that discovers the structures of certain file systems [33]. Similarly, based on the existing knowledge on common SSDs, we devise a methodology for extracting the essential performance parameters of SSDs.

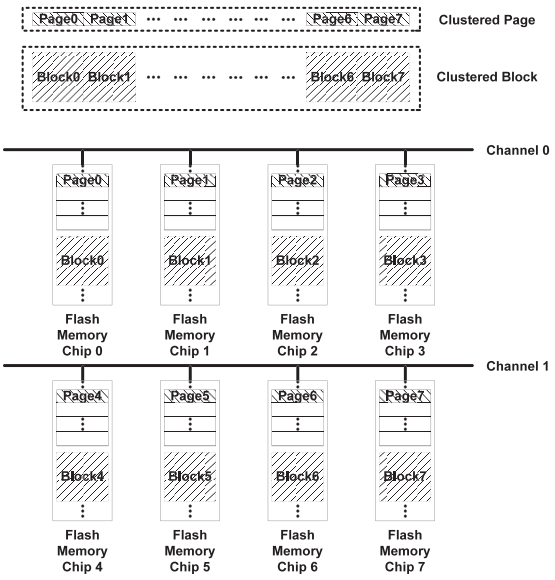


Fig. 2. Example of a clustered page (block), which is interleaved in eight flash memory chips on two channels.

4 Extracting Performance Parameters in SSDs

4.1 Parameters in SSDs

The performance parameters of SSDs are different from those of HDDs as described in Section 1. In this section, we describe the important performance parameters of SSDs which will be identified with our metholodogy.

4.1.1 Clustered Page

We define a *clustered page* as an internal unit of read or write operation used in SSDs. As discussed in Section 2.2, SSD manufacturers typically employ the interleaving technique to exploit inherent parallelism among read or write operations. One way to achieve this is to enlarge the unit size of read or write operations by combining several physical pages, each of which comes from a different NAND flash chip. Fig. 2 shows an example configuration where a clustered page is interleaved in eight flash memory chips on two channels. Note that, depending on FTLs used in SSDs, it is also possible to form a clustered page with just four physical pages on the same channel in Fig. 2, allowing two channels to operate independently.

The clustered page size is the same or a multiple of the physical page size of NAND flash memory. The clustered page size is a critical parameter for application-level I/O performance as shown in Gordon [25]. Adjusting the size of data transfer to the clustered page size, can enhance the I/O performance, since the FTL does not need to read or write more data than requested. In addition to enhancing the performance, the use of the clustered page can reduce the memory footprint required to maintain the mapping information inside SSDs.

4.1.2 Clustered Block

We define a *clustered block* as an internal unit of erase operation used in SSDs. Similar to the clustered page, SSDs often combine several blocks coming from different NAND flash chips into a single clustered block. Fig. 2 shows an example of a clustered block which consists of eight physical blocks. The use of the clustered block improves

TABLE 1
Characteristics of SSDs Used in This Paper

	SSD-A	SSD-B	SSD-C	SSD-D
Model	MCCOE64G5MPP	FTM60GK25H	SSDSA2MH080G1GN	TS64GSSD25S-M
Manufacturer	Samsung	Super Talent	Intel	Transcend
Form Factor	2.5 in.	2.5 in.	2.5 in.	2.5 in.
Capacity	64 GB	60 GB	80 GB	64 GB
Interface	Serial ATA	Serial ATA	Serial ATA	Serial ATA
Max Sequential Read Throughput(MB/s)	110	117	254	142
Max Sequential Write Throughput(MB/s)	85	81	78	91
Random Read Throughput - 4 KB(MB/s)	10.73	5.68	23.59	9.22
Random Write Throughput - 4 KB(MB/s)	0.28	0.01	11.25	0.01

the garbage collection performance by performing several erase operations in parallel. Using the clustered block is also effective in reducing the amount of mapping information, especially in block-mapping FTLs, since a clustered block, instead of an individual physical NAND block, now takes up one mapping entry.

4.1.3 Read/Write Buffer

Many SSD controllers use part of DRAM as a read buffer or write buffer to improve the access performance by temporarily storing the requested data into the DRAM buffer. Although users can obtain the DRAM buffer size via ATA IDENTIFY DRIVE command, it just displays the total DRAM size, not the size of read/write buffer. Thus, we present methodologies that can estimate the accurate sizes of these buffers in Section 4.2.5 and Section 4.2.6.

The read buffer size or the write buffer size can be a valuable hint to the buffer cache or I/O scheduler in the host operating system. For example, if we know the maximum size of write buffer, the I/O scheduler in the host system can merge incoming write requests in such a way that the request size does not go beyond the write buffer size. Similarly, the read buffer size can be used to determine the amount of data to be prefetched from SSDs.

4.2 Methodology for Extracting Performance Parameters in SSDs

4.2.1 Experiment Environment

We ran the microbenchmarks on a Linux-based system (kernel version 2.6.24.6). Our experimental system is equipped with a 2.4 GHz Intel(R) Core(TM)2 Quad and 4 GB of RAM. The system uses two disk drives, one hard disk drive (HDD) and one SSD, both of which are connected to the host system via SATA-II (Serial ATA-II) interface. The HDD is the system disk where the operating system is installed. In our experiments, we have evaluated four different SSDs commercially available from the market. The full details of each SSD used in this paper are summarized in Table 1.

Because we measure performance parameters empirically, the results sometimes vary from one execution to the next. Thus, we obtained all results in several trial runs to improve the accuracy. While we ran our microbenchmarks, we turned off SATA NCQ (Native Command Queuing) as SSD-C is the only SSD which supports this feature.

4.2.2 Assumptions on SSD Models

There are many possible architectural organizations and trade-offs in SSDs as discussed in [7]. As a result, the specifics of internal hardware architecture and software algorithm used in SSDs differ greatly from vendor to vendor. However, most of the commercial SSDs share a common basic organization described in Fig. 1b. With this hardware architecture, many commercial SSDs commonly employ a variant of block-mapping or page-mapping FTLs and the address maps of SSDs are stored in the DRAM [7], [34], [13], [8]. Furthermore, most of the SSDs use part of DRAM as a read buffer and a write buffer to improve the access performance by temporarily storing the requested data in the DRAM buffers.

For typical SSDs, as described in Section 2.2, our methodology can successfully extract the aforementioned parameters. Our methodology does not require any detailed knowledge on the target SSDs such as the number of channels, the number of NAND flash memory chips, garbage collection policies, etc. The methodology is based on the common characteristics found in most of the commercial SSDs, which are independent from the specific model details.

However, if SSDs have completely different hardware architectures and use software algorithm beyond our expected models, the extracting methodology should change. For example, some high-end SSDs, such as Fusion IO, support PCI Express and use the system memory for storing their mapping. It is also possible that SSDs may selectively cache page-level address mapping on the DRAM, while the rest of the page map is stored in flash memory [15]. In such case, it is hard to extract the parameters as the access latencies are also affected by hits or misses in the cached page maps in the DRAM.

4.2.3 Measuring the Clustered Page Size

As described in the previous section, the clustered page is treated as the unit of read and write operations inside SSDs in order to enhance the performance using channel-level and chip-level interleaving. This suggests that when only a part of a clustered page is updated, the SSD controller should first read the rest of the original clustered page that is not being updated, and combine it with the updated data, and write the new clustered page into flash memory. This read-modify-write operation [7] incurs extra flash read operations, increasing the write latency.

Consider a case (1) in Fig. 3a, where all the write requests are aligned to the clustered page boundary. In this case, no extra operations are necessary other than normal write operations. However, cases (2) and (3) illustrated in Fig. 3a necessitate read-modify-write operations as the first (in case (2)) or the second (in case (3)) page is partially updated.

To measure the clustered page size, we have developed a microbenchmark which exploits the difference in write latency depending on whether the write request is aligned to the clustered page boundary or not. The microbenchmark repeatedly writes data sequentially setting the request size as an integer multiple of physical NAND page size (e.g., 2 KB). Owing to the extra overhead associated with

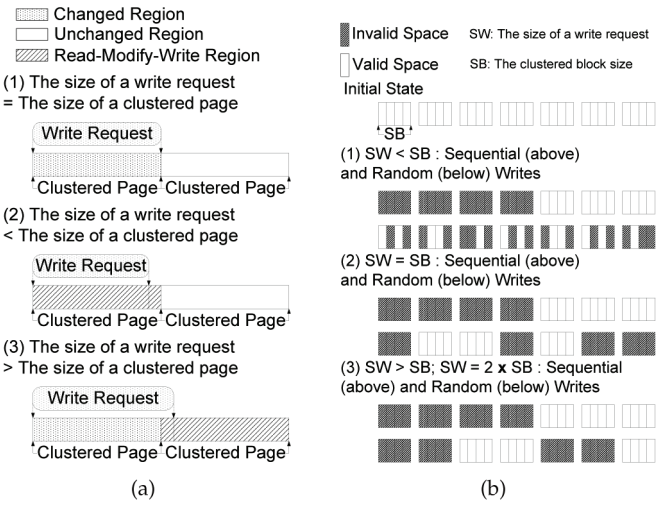


Fig. 3. (a): A write request that is aligned (1) and unaligned (2, 3) to the

unaligned write requests, we expect to observe a sharp drop in the average write latency whenever the request size becomes a multiple of the clustered page size. Procedure 1 describes the pseudocode of our microbenchmark.

Procedure 1. ProbeClusteredPage
Input: F, /* file descriptor for the raw disk device opened with O_DIRECT */
TSW, /* the total size to write (in KB, e.g., 1,024 KB) */
ISW, /* the increment in size (in KB, e.g., 2 KB) */
NI /* the number of iteration (e.g., 64) */
1: $SW \Leftarrow 0$ /* the size of write request (in KB) */
2: write_init(F) /* initialize the target SSD by sequentially updating all the available sectors to minimize the effect of garbage collection */
3: **while** $SW \leq TSW$ **do**
4: $SW \Leftarrow SW + ISW$
5: lseek(F, 0, $SEEK\_SET$) /* set the file pointer to the offset 0 */
6: $Start \Leftarrow gettimeofday()$
7: **for** $i = 1$ to NI **do**
8: write_file(F, SW) /* write SW KB of data to F */
9: ATA_FLUSH_CACHE() /* flush the write buffer */
10: **end for**
11: $End \Leftarrow gettimeofday()$
12: **print** the elapsed time by using $Start$ and End
13: **end while**

There are some implementation details worth mentioning in Procedure 1. First, we open the raw disk device with O_DIRECT flag to avoid any influence from buffer cache in the host operating system. Second, before the actual measurement, we initialize the target SSD by sequentially updating all the available sectors to minimize the effect of garbage collection during the experiment [12]. Third, we make the first write request during each iteration always begin at the offset 0 using lseek(). Finally, all experiments are performed with the write buffer in SSDs enabled. To reduce the effect of the write buffer, we immediately flush data to NAND flash memory by issuing ATA FLUSH CACHE command, after writing data to the target SSD. Most of these implementation strategies are also applied to other microbenchmarks presented in the following sections.

To estimate the clustered page size, we have measured the latency of each write request varying the request size up to 1,024 KB. Fig. 4 plots the results obtained by running Procedure 1 on the tested SSDs. All the experiments for SSD-A, SSD-B, and SSD-D are performed with the write buffer enabled. Enabling the write buffer in SSD-C makes it difficult to measure the latency accurately as the cost of the internal flush operation highly fluctuates. Thus, the microbenchmark was run with the write buffer disabled in SSD-C so that the measurement is not affected by the activity of flush operation.

In Fig. 4, the general trend is that the latency increases in proportion to the request size. However, we can observe that there are periodic drops in the latency. For example, in Fig. 4a, the latency drops sharply whenever the request size is a multiple of 16 KB. Therefore, we can conclude that the clustered page size of SSD-A is 16 KB. For the same reason, we believe that the clustered page size of SSD-B, SSD-C, and SSD-D is 128 KB, 4 KB, and 128 KB, respectively.

Unlike other SSDs, the result of SSD-C shows no notable drop in the write latency. Upon further investigation, it turns out that SSD-C internally allows the update of only one sector (512 B); thus, the additional overhead for read-modify-write is eliminated. An intriguing observation in Fig. 4 is that there are several spikes in the write latency, most notably in Fig. 4b, 4c, and 4d. We suspect this is due to garbage collection which should be occasionally invoked to make free blocks.

4.2.4 Measuring the Clustered Block Size

The clustered block is the unit of an erase operation in SSDs to improve the write performance associated with garbage collection. This indicates that if only a part of a clustered block is updated when garbage collection is triggered, live

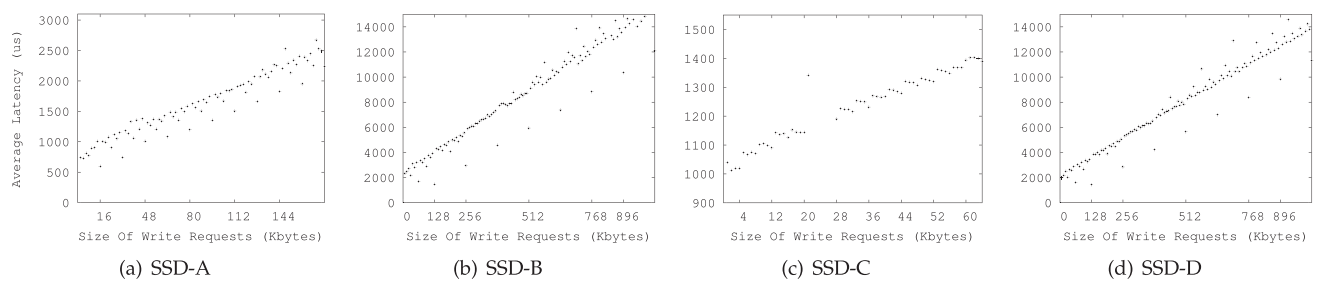
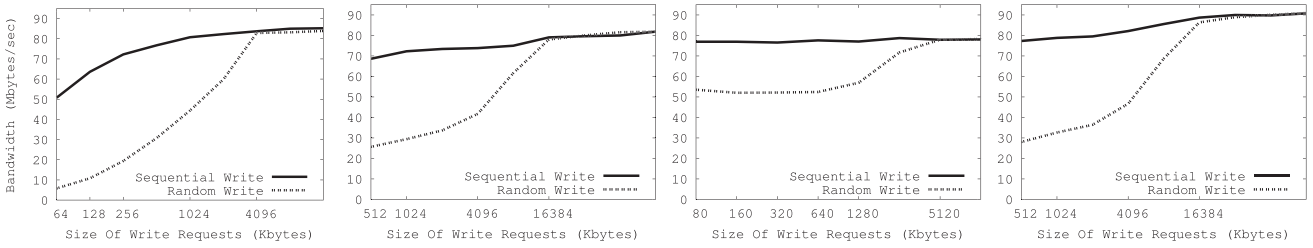


Fig. 4. The average write latency with varying the size of write requests.



(a) SSD-A (b) SSD-B (c) SSD-C (d) SSD-D

Fig. 5. Sequential versus random write bandwidth according to the size of write requests.

pages in the original clustered block should be copied into another free space in SSDs. This valid copy overhead affects the write performance of SSDs, decreasing the write bandwidth noticeably.

Consider a case (1) illustrated in Fig. 3b, where the size of write requests is smaller than that of the clustered block. Assume that the leftmost clustered block has been selected as a victim by the garbage collection process. When a series of blocks are updated sequentially, there is no overhead other than erasing the victim block during garbage collection. However, if there are many random writes whose sizes are smaller than the clustered block size, the write bandwidth will suffer from the overheads of copying valid pages. As shown in cases (2) and (3) of Fig. 3b, the additional overhead disappears only when the size of random write requests becomes a multiple of the clustered block size.

To retrieve the clustered block size, our microbenchmark exploits the difference in write bandwidth between sequential and random writes. Initially, the size of write request is set to the clustered page size. And then, for the given request size, we issue a number of sequential and random writes which are aligned to the clustered page boundary, and measure the bandwidth. We repeat the same experiment, but each time the request size is doubled. As the request size approaches to the clustered block size, the gap between the bandwidth of sequential writes and that of random writes will become smaller. Eventually, they will show the similar bandwidth once the request size is equal to or larger than the clustered block size. Procedure 2 briefly shows how our microbenchmark works to probe the clustered block size.

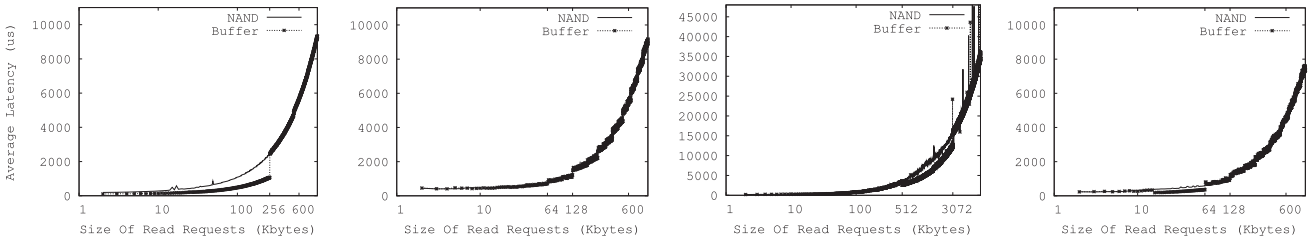
Procedure 2. ProbeClusteredBlock

Input: F, /* file descriptor for the raw disk device opened with O_DIRECT */
SP, /* the clustered page size obtained in Section 4.2.3 (in KB, e.g., 16 KB) */
TNP, /* the total number of cluster pages (e.g., 1024) */
TSW, /* the total size to write (in KB, e.g., $(8 \times 1{,}024 \times 1{,}024$ KB)) */
NP /* the initial number of clustered pages (e.g., 2). $NP \times SP$ is the actual size of write requests */

1: $NI \Leftarrow 0$ /* the number of iteration */
2: **while** $NP \leq TNP$ **do**
3: $NP \Leftarrow NP \times 2$ /* We assume the clustered block size is a power of 2 multiple of the clustered page size */
4: write_init(F) /* initialize the target SSD */
5: $Start \Leftarrow gettimeofday()$
6: lseek(F, 0, $SEEK\_SET$) /* set the file pointer to the offset 0 */
7: $NI \Leftarrow TSW/(NP \times SP)$
8: **for** $i = 1$ to NI **do**
9: write_file(F, $NP \times SP$) /* write ($NP \times SP$) KB of data to F */
10: ATA_FLUSH_CACHE() /* flush the write buffer */
11: **end for**
12: $End \Leftarrow gettimeofday()$
13: **print** the elapsed time of sequential writes by using $Start$ and End
14: write_init(F)
15: $Start \Leftarrow gettimeofday()$
16: **for** $i = 1$ to NI **do**
17: $R \Leftarrow rand()\%NI$ /* choose R randomly */
18: $R \Leftarrow R \times (NP \times SP) \times 1{,}024$
19: lseek(F, R, $SEEK\_SET$)
20: write_file(F, $NP \times SP$)
21: ATA_FLUSH_CACHE()
22: **end for**
23: $End \Leftarrow gettimeofday()$
24: **print** the elapsed time of random writes by using $Start$ and End
25: **end while**

To determine the clustered block size, the microbenchmark measures the bandwidth of sequential and random writes, increasing the request size up to 128 MB. Fig. 5 compares the results for four tested SSDs. The value of NP, which represents the initial number of clustered pages to test, is set to two for SSD-A, SSD-B, and SSD-D. For SSD-C, we configure $NP = 10$ as there was no difference in the bandwidth between sequential and random writes with $NP = 2$.

From Fig. 5a, we find that the bandwidth of sequential writes is higher than that of random writes when the size of write request is smaller than 4,096 KB. If the request size is increased beyond 4,096 KB, there is virtually no difference in the bandwidth. As mentioned above, the bandwidth of random writes converges to that of sequential writes as the request size approaches to the clustered block size. This suggests that the clustered block size of SSD-A is 4,096 KB. Similarly, we can infer that the clustered block size of SSD-B, SSD-C, and SSD-D is 16,384 KB, 5,120 KB, and 16,384 KB, respectively.



(a) SSD-A (b) SSD-B (c) SSD-C (d) SSD-D

Fig. 6. The latency of read requests with increasing the size of read requests.

4.2.5 Measuring the Read Buffer Capacity

The read buffer in SSDs is used to improve the read performance by temporarily storing the requested and/or prefetched data. If the requested data cannot be found in the read buffer, or if the size of the read request is larger than the size of the read buffer, then the data has to be read directly from NAND flash memory, which results in larger read latencies.

To differentiate the read request served from the read buffer from that served from NAND flash memory, we have developed two microbenchmarks, ProbeReadBuffer() and ProbeNANDReadLatency(), as shown in Procedure 3 and Procedure 4.

Procedure 3. ProbeReadBuffer

Input: F, /* file descriptor for the raw disk device opened with O_DIRECT */
TSR, /* the total size to read (in KB, e.g., 1,024 KB) */
ISR /* the increment in size (in KB, e.g., 1 KB) */

1: $SR \Leftarrow 0$ /* the size of read request (in KB) */
2: write_init(F) /* initialize the target SSD */
3: **while** $SR \leq TSR$ **do**
4: $SR \Leftarrow SR + ISR$
5: $R \Leftarrow rand()\%1{,}024$ /* choose R randomly */
6: lseek(F, $1{,}024 \times 1{,}024 \times 1{,}024 + R \times 16 \times 1{,}024 \times 1{,}024$, $SEEK\_SET$) /* set the file pointer randomly */
7: read_file(F, $16 \times 1{,}024$) /* read 16 MB of data from F */
8: $R \Leftarrow rand()\%63$
9: lseek(F, $R \times 16 \times 1{,}024 \times 1{,}024$, $SEEK\_SET$) /* set the file pointer randomly (We assume the size of read buffer is smaller than 16 MB) */
10: read_file(F, SR) /* read SR KB of data from F */
11: lseek(F, $R \times 16 \times 1{,}024 \times 1{,}024$, $SEEK\_SET$)
12: $Start \Leftarrow gettimeofday()$
13: read_file(F, SR)
14: $End \Leftarrow gettimeofday()$
15: **print** the elapsed time by using $Start$ and End
16: **end while**

Procedure 4. ProbeNANDReadLatency

Input: F, /* file descriptor for the raw disk device opened with O_DIRECT */
TSR, /* the total size to read (in KB, e.g., 1,024 KB) */
ISR /* the increment in size (in KB, e.g., 1 KB) */

1: $SR \Leftarrow 0$ /* the size of read request (in KB) */
2: write_init(F) /* initialize the target SSD */
3: **while** $SR \leq TSR$ **do**
4: $SR \Leftarrow SR + ISR$
5: $R \Leftarrow rand()\%1{,}024$ /* choose R randomly */
6: lseek(F, $1{,}024 \times 1{,}024 \times 1{,}024 + R \times 16 \times 1{,}024 \times 1{,}024$, $SEEK\_SET$) /* set the file pointer randomly */
7: read_file(F, $16 \times 1{,}024$) /* read 16 MB of data from F */
8: $R \Leftarrow rand()\%63$
9: lseek(F, $R \times 16 \times 1{,}024 \times 1{,}024$, $SEEK\_SET$) /* set the file pointer randomly (we assume that the size of read buffer is smaller than 16 MB) */
10: $Start \Leftarrow gettimeofday()$
11: read_file(F, SR) /* read SR KB of data from F */
12: $End \Leftarrow gettimeofday()$
13: **print** the elapsed time by using $Start$ and End
14: **end while**

The microbenchmark ProbeReadBuffer() is used to measure the latency of read requests served from the read buffer, if any. The microbenchmark repeatedly issues two read requests, each of which reads data from the same location O.[1] It measures the latency of the second request, hoping that a read hit occurs in the read buffer for the request. Before reading any data from O, the benchmark fills the read buffer with the garbage by reading large data from the random location far from O. In each iteration, the size of read request is increased by 1 KB, by default. If the size of read request becomes larger than the read buffer size, the whole data cannot be served from the read buffer and the request will force flash read operations to occur. Thus, we expect to observe a sharp increase in the average read latency whenever the request size is increased beyond the read buffer size.

On the other hand, ProbeNANDReadLatency() is designed to obtain the latency of read requests which are served from NAND flash memory directly. The benchmark is similar to ProbeReadBuffer() except that the first read request (lines 7-8) in ProbeReadBuffer() has been eliminated to generate read misses all the times.

To estimate the capacity of the read buffer, we compare the latency measured by ProbeReadBuffer() with that obtained by ProbeNANDReadLatency(), varying the size of each read request. Fig. 6 contrasts the results with respect to the read request size from 1 KB to 1,024 KB (4,096 KB for SSD-C). In Fig. 6, the labels "NAND" and "Buffer" denote

1. In each iteration, this location is set randomly based on the R value, which eliminates the read-ahead effect, if any, in target SSDs. In the tested SSDs, however, we could not observe any read-ahead mechanism.

the latency obtained from ProbeNANDReadLatency() and from ProbeReadBuffer(), respectively. As mentioned above, ProbeNANDReadLatency() always measures the time taken to retrieve data from NAND flash memory, while ProbeReadBuffer() approximates the time to get data from the read buffer as long as the size of read requests is smaller than the read buffer size.

In Fig. 6a, when the size of read requests is smaller than 256 KB, "Buffer" results in much shorter latency compared to "NAND." This is because requests generated by ProbeReadBuffer() are fully served from the read buffer. On the other hand, if the request size exceeds 256 KB, both "Buffer" and "NAND" exhibit almost the same latency. Since "NAND" represents the time to read data from NAND flash memory, this result means that read requests whose sizes are bigger than 256 KB cannot be handled in the read buffer. Therefore, we can conclude that the read buffer size of SSD-A is 256 KB. For SSD-C and SSD-D, the similar behavior is also observed for the request sizes from 512 KB to 3,072 KB (SSD-C), or from 16 KB to 64 KB (SSD-D). Therefore, the read buffer size of SSD-C and SSD-D is 3,072 KB and 64 KB, respectively. However, in case of SSD-B, the results of both "NAND" and "Buffer" show exactly the same behavior, which implies that SSD-B does not use any read buffer.

4.2.6 Measuring the Write Buffer Capacity

As discussed in Section 4.1, the main role of the write buffer in SSDs is to enhance the write performance by temporarily storing the updated data into the DRAM buffer. This implies that when the size of write requests exceeds the write buffer size, some of data should be flushed into NAND flash memory. This additional flush operation results in extra flash write operations, impairing the write latency.

To determine whether the write request is handled by the write buffer or NAND flash memory, we have developed two microbenchmarks, ProbeWriteBuffer() and ProbeNANDWriteLatency(), as shown in Procedure 5 and Procedure 6. The former measures the time taken to write data into the write buffer, if any, while the latter is intended to measure the time to write the requested data to NAND flash memory.

Procedure 5. ProbeWriteBuffer

Input: F, /* file descriptor for the raw disk device opened with O_DIRECT */
TSW, /* the total size to write (in KB, e.g., 1,024 KB) */
ISW, /* the increment in size (in KB, e.g., 1 KB) */
NI /* the number of iteration (e.g., 30) */

1: $SW \Leftarrow 0$ /* the size of write request (in KB) */
2: write_init(F) /* initialize the target SSD */
3: **while** $SW \leq TSW$ **do**
4: $SW \Leftarrow SW + ISW$
5: **for** $i = 1$ to NI **do**
6: ATA_FLUSH_CACHE() /* flush the write buffer */
7: lseek(F, 0, $SEEK\_SET$) /* set the file pointer to the offset 0 */
8: $Start \Leftarrow gettimeofday()$
9: write_file(F, SW) /* write SW KB of data to F */
10: $End \Leftarrow gettimeofday()$
11: **print** the elapsed time by using $Start$ and End
12: **end for**
13: **end while**

Procedure 6. ProbeNANDWriteLatency

Input: F, /* file descriptor for the raw disk device opened with O_DIRECT */
TSW, /* the total size to write (in KB, e.g., 1,024 KB) */
ISW, /* the increment in size (in KB, e.g., 1 KB) */
NI /* the number of iteration for outer loop (e.g., 30) */

1: $SW \Leftarrow 0$ /* the size of write request (in KB) */
2: write_init(F) /* initialize the target SSD */
3: **while** $SW \leq TSW$ **do**
4: $SW \Leftarrow SW + ISW$
5: **for** $i = 1$ to NI **do**
6: ATA_FLUSH_CACHE() /* flush the write buffer */
7: lseek(F, $16 \times 1{,}024 \times 1{,}024$, $SEEK\_SET$) /* We assume that the size of write buffer is smaller than 16 MB */
8: write_file(F, $16 \times 1{,}024$) /* write 16 MB of data to F */
9: lseek(F, 0, $SEEK\_SET$) /* set the file pointer to the offset 0 */
10: $Start \Leftarrow gettimeofday()$
11: write_file(F, SW) /* write SW KB of data to F */
12: $End \Leftarrow gettimeofday()$
13: **print** the elapsed time by using $Start$ and End
14: **end for**
15: **end while**

ProbeWriteBuffer() repeatedly measures the write latency, increasing the request size by 1 KB. Before the actual measurement, the benchmark makes the write buffer empty by issuing the flush operation supported by the ATA command. After flushing the write buffer, we expect that the subsequent write request is handled in the write buffer, if any, as long as the request size is smaller than the write buffer size. When the request size is too large to fit into the write buffer, the request will cause flash write operations, prolonging the average write latency severely.

ProbeNANDWriteLatency() is analogous to ProbeWriteBuffer() except that lines 7-8 are added to fill the entire write buffer with garbage intentionally. Since the write buffer is already full, some part of data is flushed to NAND flash memory upon the arrival of the next write request.

Note that, in ProbeWriteBuffer() and ProbeNANDWriteLatency(), we repeatedly measure the write latency NI times for the given request size. This is because it is not easy to accurately measure the time needed to write data in the presence of asynchronous flush operations. Especially, when the write buffer has some valid data, the actual timing the flush operation is performed and the amount of data flushed from the write buffer to NAND flash memory can vary from experiment to experiment. To minimize the effect of these variable factors, we obtain enough samples by repeating the same experiment multiple times.

As mentioned above, ProbeWriteBuffer() measures the latency required to store data to the write buffer, while ProbeNANDWriteLatency() estimates the write latency needed to flush data to NAND flash memory. Fig. 7 plots the measured latencies for four commercial SSDs with various request sizes ranging from 1 KB to 1,024 KB. In

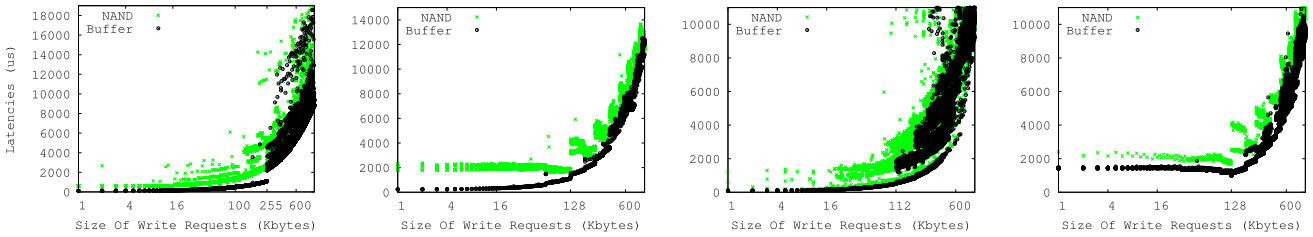


Fig. 7. The latency of write requests with increasing the size of write requests.

Fig. 7, "NAND" and "Buffer" indicate the latencies obtained from ProbeNANDWriteLatency() and ProbeWriteBuffer(), respectively.

When the size of write requests is less than or equal to 255 KB, "Buffer" shows much shorter latencies than "NAND" in Fig. 7a. This indicates that such write requests are fully handled in the write buffer. On the other hand, if the size of write requests becomes larger than 255 KB, "Buffer" shows a sharp increase in the write latency probably because the write buffer cannot accommodate the requested data and causes flash write operations. In particular, the lowest latency of "Buffer" is similar to that of "NAND" when the request size is 255 KB. This confirms that the size of write buffer in SSD-A is 255 KB. Any attempt to write data larger than 255 KB incurs extra flush overhead, although the write buffer is empty. For SSD-C, the similar behavior is also observed when the request size is 112 KB. Thus, we believe that write buffer size of SSD-C is 112 KB.

In cases of SSD-B and SSD-D, slightly different behaviors have been noted. For SSD-B, we can see that "Buffer" exhibits the faster latency compared to "NAND" when the request size is between 1 KB and 128 KB. For the same reason with SSD-A and SSD-C, the size of the write buffer for SSD-B is estimated to 128 KB. For SSD-D, it appears that SSD-D does not make use of any write buffer, however, we could not draw any conclusion using our methodology since the behavior of SSD-D is so different from other SSDs.

4.2.7 Elapsed Times to Extract Parameters

Table 2 is added to show the elapsed times for measurement. Extraction times range from 18 sec to 327 min, and we believe they are short enough to be used for practical purposes. Note that the parameter extraction is necessary only once for each SSD. Using only a quarter of a day in the worst case, our methodology can successfully extract the performance parameters in all tested SSDs.

4.3 More Parameters in SSDs

In addition to the four performance parameters we discussed in the previous section, we also identify two policies of using the read and write buffers to enhance performance. The first identified policy is whether an SSD uses a prefetching or read-ahead mechanism to reduce read latencies. To determine whether prefetching is used, we compare two cases. The first case is to set the location R on line 8 in Procedure 4 (ProbeNANDReadLatency()) randomly, which eliminates the read-ahead effect. The second case is to set the location R statically, which triggers the read-ahead mechanism, if any prefetch policy was used in SSDs. In the tested SSDs, however, we identified that no read prefetching is used, since there is no performance advantage in the second case compared with the first one.

The second policy we identify is whether the available DRAM for write traffics is organized to a write cache or a write buffer. To reduce writes to the flash memory, a write cache keeps frequently updated pages in the DRAM, with a possibly better replacement policy and more DRAM capacity than a write buffer. However, a write buffer flushes updated pages to the flash memory in FIFO order, as soon as the buffer is full. Our test issues two writes to the same logical address, and the second one may hit in the DRAM, if the first write is still in the DRAM. For the three SSDs except for SSD-C, the results show that the second write hits in the DRAM only when the second one is issued immediately after the first one without any other writes between them, showing the DRAM is used as a simple write buffer, not a write cache.

However, SSD-C does not present a clear pattern as shown in Fig. 7c. This is because the flush operations show asynchronous characteristics in the SSD. Furthermore, the actual timing the flush operation is performed and the amount of data flushed from the write buffer to NAND flash memory vary from iteration to iteration. Table 3 summarizes the additional parameters obtained from all tested SSDs.

4.4 Limitations

Since our extracting methodology exploits the common architectures and characteristics found in many commercial SSDs, it has some limitations as well. First, for a clustered page, if an SSD supports a mechanism to hide read-modify-write overheads, the size of a clustered page is difficult to extract. For example, SSD-C shows no notable drop in the

TABLE 2
Elapsed Time to Extract Parameters

	SSD-A	SSD-B	SSD-C	SSD-D
clustered page	0m18s	1m3s	1m4s	1m13s
clustered block	30m46s	92m14s	327m46s	90m50s
read buffer	10m54s	14m13s	46m9s	12m15s
write buffer	1m18s	9m5s	1m48s	8m21s

TABLE 3
More Parameters in SSDs

	SSD-A	SSD-B	SSD-C	SSD-D
prefetching	no	no	no	no
write buffer or cache	buffer	buffer	buffer	buffer

write latency at every 4 Kbytes. That means the manufacturer of SSD-C internally exploits the performance optimizations to eliminate the overhead for read-modify-write for writes with small data sizes. For a clustered block size, if an SSD reduces the garbage collection overheads between sequential/random writes, obtaining the clustered block becomes difficult. For a read/write buffer, if an SSD uses a read-ahead technique and manages the DRAM not as a buffer but as an associative write cache, we should redesign our methodology to extract a read/write buffer size.

Another important policy for SSDs is the garbage collection mechanisms and policies. Although they affect the performance and longevity of SSDs, it is extremely difficult to infer them without any knowledge of internal organization, mapping policy, and FTL designs. Furthermore, the irregularity of garbage collection triggers worsens the difficulty of inferring the garbage collection policy. For the above reason, inferring garbage collection policies without any prior knowledge of the FTL is open for further investigation, which is beyond the limited scope of this paper.

For SSD-A, we have confirmed from the manufacturer of the SSD that all the parameter values we found are correct. Unfortunately, however, the parameters of the other SSDs could not be verified by the manufacturers, as they do not make the parameters publicly available. However, the extracted parameters are meaningful, so long as they can exhibit expected performance improvements by optimizing SW components with the parameters. To verify the parameters indirectly, in the next section, we show the performance improvement of the system by using the parameters extracted from each SSD.

5 Parameter-Aware I/O Component Design

In this section, to demonstrate the benefits of using the extracted parameters for system optimizations, we propose new parameter-aware designs of I/O components in a commercial operating system. We modify two kernel layers in a Linux operating system, generic block layer and I/O scheduler. Using the extracted performance parameters from different SSDs, the modified two components adjust the size and alignment of block requests from the file system, to send optimized requests to SSDs.

5.1 Linux I/O Components Overview

Linux kernels have several layers of I/O components to manage disk I/O operations. The topmost file system layer provides the file system abstraction. The file system manages the disks at block granularity, and the request sizes from the file systems are always multiples of the block size. The generic block layer under the file system, receives block requests and process them before passing them to the I/O scheduler layer. The generic block layer merges block requests from the file system layer to optimize the request size. If requests from the file system access consecutive disk addresses, the block layer merge them into a single request. Such merging amortizes the cost of sending requests, reducing the overhead per byte transferred.

The merged requests from the block layer are transferred to the I/O scheduler layer. The I/O scheduler layer schedules I/O requests to reduce the seek times of hard disks. The I/O scheduler reorders the requests to minimize disk arm movements, while still providing the fairness of disk accesses. For hard disks, one of the most commonly used I/O schedulers is CFQ (completely fair queuing), which provides equal sharing of disk bandwidth. However, SSDs have completely different characteristics from hard disks. SSDs do not require seek times to move mechanical arms, so complex scheduling is not necessary in the I/O scheduler. It has been shown that in SSDs, doing nothing in the I/O scheduler (NOOP scheduler) performs better than a traditional I/O scheduler for disks, such as CFQ [34], [35]. In this paper, we modify the NOOP scheduler to a parameter-aware I/O scheduler for SSDs.

5.2 Optimizing Linux Generic Block Layer for SSDs

As discussed in Section 5.1, the generic block layer merges consecutive requests from the file system to reduce the number of requests delivered to the disks. However, such merging operations incur trade-offs between the total disk throughput and the response time for each request. Since requests for large data take longer response times than requests for small data, increasing the request size hurts the response times for individual requests. Furthermore, large requests consume more memory buffer cache resources. Therefore, the block layer must use an optimal maximum request size, which can balance between the throughput and the response time. To find the optimal request size for traditional hard disks, Schindler et al. used disk track boundary [3].

In this paper, we propose two designs for the block layer using the SSD performance parameters. The first design, *parameter aware splitting (PAS)*, sets the maximum request size for merging in the block layer to the read or write buffer size of SSDs. The second design, *parameter aware splitting and aligning (PASA)*, not only sets the maximum request to the read and write buffer sizes, but also aligns each request to the clustered page or block size.

Parameter-aware splitting (PAS) uses the size of a read/write buffer obtained in Section 4 to set the maximum request size in the block layer. As shown in Fig. 8a, when the contiguous requests are submitted from the file system layer to the generic block layer, PAS merges the requests and splits them into pieces whose size is the read or write buffer size of the SSD. PAS can achieve a short response time, since the entire request can be sent to the fast DRAM buffer in SSDs.

Second, parameter-aware splitting and aligning (PASA) further optimizes PAS by aligning requests to the clustered page or block size. Unlike PAS, PASA first checks whether a request is aligned to the clustered page or block boundary. If the request is aligned to the clustered page/block boundary, PASA splits the requests by the size of the read or write buffer. Otherwise, PASA first aligns the requests to the clustered page/block boundary and splits the requests by the read or write buffer size, as shown in a right figure in Fig. 8a.

5.3 Linux I/O Scheduler Optimization

The block layer does not reorder requests, but can only merge and split consecutive requests from the file system. The I/O scheduler can reorder requests and look for further opportunities to merge and split requests by the read or write buffer sizes. We redesign the I/O scheduler to make requests aligned and well-split for SSDs.

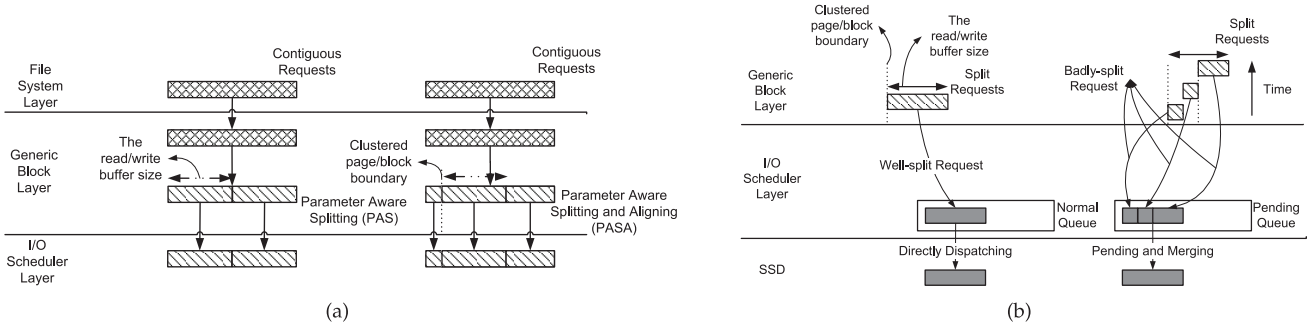
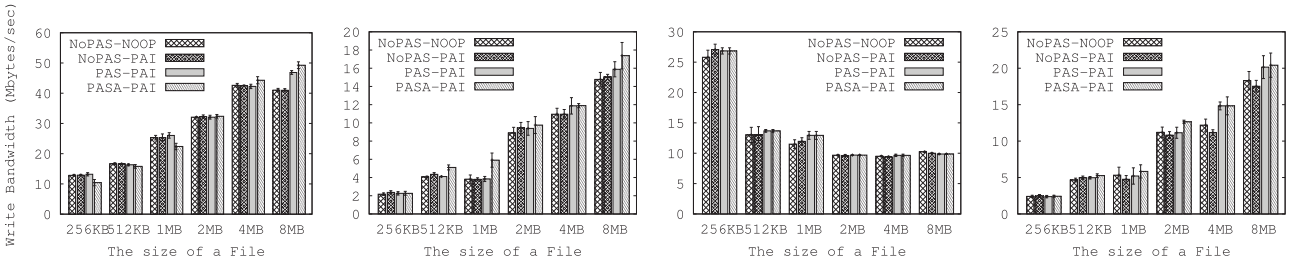


Fig. 8. Parameter aware splitting (PAS) and aligning (PASA) (a), and parameter aware I/O scheduler (PAI) (b).



(a) SSD-A (Request alignment to the clustered page size (16 KB)) (b) SSD-B (Request alignment to the clustered page size (128 KB)) (c) SSD-C (Request alignment to the clustered page size (4 KB)) (d) SSD-D (Request alignment to the clustered page size (128 KB))

Fig. 9. Postmark : Write bandwidth with postmark (the read bandwidth results show similar trends to the write bandwidth results).

Parameter-aware I/O scheduler (PAI) maintains two queues, a normal queue and a pending queue, to find more mergeable requests, as shown in Fig. 8b. The normal queue manages well-split requests, which are aligned to the clustered page or block size and split by the read or write buffer size. The pending queue manages the badly-split requests, which are not aligned to the clustered page or block boundary or split by the read or write buffer size. The badly-split requests stay in the pending queue for a while, to be merged with other requests. A similar technique is used in schedulers for traditional hard disks [36].

Since the requests in the normal queue are optimized to the SSD, PAI dispatches the requests before the requests in the pending queue. To prevent increasing the response times of the pending requests and avoid starvation, PAI sets the time bound for dispatching the requests from the pending queue.

6 Performance Evaluation

6.1 Experimental Setup

We have implemented PAS and PASA in the original block layer of the Linux operating system, and PAI as a Linux kernel module. We use Linux kernel 2.6.24.6. The system configurations used in this experiment are described in Section 4.2.1. Four different SSDs used in this evaluation are summarized in Table 1. All experiments are performed with the ext3 file system, and the block size of the file system is set to 4 KB.

In Section 6.2 and Section 6.3, we evaluate the performance of SSDs with four configurations. We use the NoPAS-NOOP (NoPAS: No-Parameter-Aware-Splitting) as the baseline configuration and compare NoPAS-PAI, PAS-PAI and PASA-PAI to the baseline. We use the NOOP scheduler as the baseline I/O scheduler, since the NOOP scheduler provides the best performance among the available disk-based I/O schedulers with SSDs and has a reasonable fairness guarantee for SSDs that has no seek time [35], [34].

We use two benchmarks to evaluate the modified block layer and the I/O scheduler: postmark [37], filebench [38]. All experiments were repeated 10 times, and we mark an error range for each result. To improve the experimental accuracy, we flush the page cache in Linux and the write caches of SSDs before each experiment is performed.

6.2 Postmark

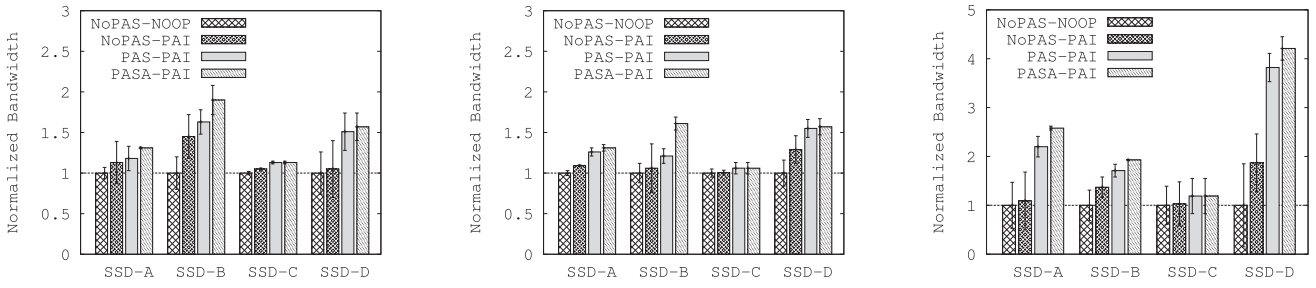
The first benchmark we use for evaluation is the Postmark (version 1.51) benchmark [37] with the file sizes varied from 256 KB to 8 MB. We use 200 simultaneous files in 200 subdirectories. The benchmark runs 3,000 transactions for SSD-A and SSD-C, and 300 transactions for SSD-B and SSD-D. The seed for random number generator is set to 712. Fig. 9 shows the evaluation results for four configurations as mentioned in Section 6.1. Figs. 9a, 9b, 9c, and 9d show the results on SSD-A, SSD-B, SSD-C, and SSD-D, respectively.

For postmark, using PAI with neither PAS nor PASA (NoPAS-PAI) shows mixed results. Compared to the baseline (NoPAS-NOOP), the performance with NoPAS-PAI can either improve or drop, depending on the file size and the SSDs. The mixed results are due to the trade-off of using PAI. PAI may slightly delay issuing requests to the SSD to increase the chance to merge requests in the pending queue. If merging does not occur, the baseline (NoPAS-NOOP), which sends requests without delay, performs better than NoPAS-PAI. If merging occurs frequently, the benefits of PAI outweigh the delaying overhead.

However, combining PAS or PASA with PAI provides noticeable performance improvements over the baseline for large file sizes of 4 and 8 MB. With smaller input file sizes, the performance gains are generally minor. The benefits of

TABLE 4
Composition of Requests Dispatched from PAI to the Block Device Driver (Postmark with SSD-A)

Configuration	All requests (%)	Aligned requests (%)	Unaligned requests (%)	$\leq$ 256KB requests	$>$ 256KB requests
NoPAS 8M	34788 (100 %)	13310 (38.26 %)	21478 (61.74 %)	8121 (23.34 %)	26667 (76.66 %)
PAS 8M	59348 (100 %)	20906 (35.22 %)	38442 (64.78 %)	59348 (100 %)	0 (0 %)
PASA 8M	64834 (100 %)	57371 (88.48 %)	7463 (11.52 %)	64834 (100 %)	0 (0 %)
NoPAS 256K	3197 (100 %)	858 (26.84 %)	2339 (73.16 %)	3148 (98.46 %)	49 (1.54 %)
PAS 256K	2972 (100 %)	1288 (43.33 %)	1684 (56.66 %)	2972 (100 %)	0 (0 %)
PASA 256K	4587 (100 %)	2541 (55.39 %)	2046 (44.60 %)	4587 (100 %)	0 (0 %)



(a) webserver profile (b) fileserver profile (c) OLTP profile

Fig. 10. Normalized bandwidth with Filebench (webserver, fileserver, and OLTP profiles): y-axis is normalized to NoPAS-NOOP for each SSD.

size adjustment and alignment are much higher in large file inputs than small file inputs. With the small file sizes, PAS and PASA have few chances of adjustment, as requests are mostly smaller than the write buffer size.

To show the effects of the request size adjustment and alignment, table 4 presents the number of total requests, the number of aligned requests, and the number of small (smaller than 256 KB[2]) and large (larger than 256 KB) requests. In general, the total number of requests increases with PAS and PASA. For the 8MB file configuration, the total numbers of requests submitted to an SSD with PAS and PASA, increase by 171 percent and 186 percent, respectively, compared to NoPAS. The ratio of adjusted requests ($\leq$256 KB requests) in PAS (100 percent) and PASA (100 percent) is higher than that in NoPAS (23 percent). Furthermore, the ratio of aligned requests in PASA (88 percent) is higher than that in PAS (35 percent) and NoPAS (38 percent). Thus, in 8 MB configuration with SSD-A, PASA outperforms PAS and NoPAS by five percent and 20 percent with PAI.

However, in 256 KB configuration, most of the requests (98 percent) are small ones with less than 256 KB. Therefore, without PAS or PASA, the original requests fit in the write buffer. Furthermore, there are relatively small increases in aligned requests compared to 8 MB configuration. The performance gains with 256 KB are smaller that those with 8 MB, as the performance benefits from the adjustment and alignment are lower than the performance costs from the increased number of requests.

The patterns of performance changes in SSD-B and SSD-D are similar to that of SSD-A. However, SSD-C exhibits a different behavior, with little improvement for large files by our optimization. Instead, for SSD-C, there are modest improvements for 256 KB, 512 KB, and 1 MB file sizes.

2. The size of the read/write buffer in SSD-A is 256 KB.

6.3 Filebench

The second benchmark to evaluate parameter-aware optimizations is the filebench (version 1.64 and 1.48) benchmark [38]. Among several workloads, we use three profiles (webserver, fileserver and OLTP(online transaction processing)) of the filebench. The webserver profile reads the file set of random sizes and appends a log file using writes. We use 1,000 files and 16 MB average file size with 16 threads in SSD-A and SSD-C and use 100 files in SSD-B and SSD-D. The fileserver profile creates, writes, appends and deletes the set of files randomly. We use 10,000 files and 1 MB average file size with 50 threads. The OLTP profile specifies datafiles, logfiles and database write threads. We use 10 MB average file size and 200 shadow processes to handle the OLTP transactions. For all the profiles, we use configurations with relatively large request sizes.

Fig. 10 shows the bandwidth results from the three profiles in the filebench for four configurations as described in Section 6.1. In Fig. 10, NoPAS-PAI outperforms NoPAS-NOOP in all SSDs. The bandwidth is further improved with PAS or PASA, except for SSD-C. For SSD-A, SSD-B, and SSD-D, combining PASA and PAI results in significant bandwidth improvements, up to 321 percent with PASA-PAI in SSD-D on OLTP profile. PAI alone can provide relatively small improvements, but the synergy between the optimizations in the block layer and I/O scheduler improves the bandwidth significantly when PAI is combined with PAS or PASA. The performance enhancement of the OLTP profile is higher than those of the other profiles (webserver and fileserver). Since the OLTP profile produces intensive writes, using write buffers efficiently with PAI-PAS/PASA improves the overall performance significantly. SSD-C shows little improvement with the optimizations, which are consistent with the observation from the large file configuration in the Postmark benchmark in the previous section.

6.4 Discussion

The results with the two benchmarks showed that the parametrization of the block layer and I/O scheduler

provides significant performance improvements for SSD-A, SSD-B, and SSD-D, with large input file sizes. In general, the optimizations become more effective as the average request sizes increase, since the block layer and I/O scheduler can have more chances to adjust request sizes and alignment with large request sizes. Combining PASA in the block layer and PAI in the I/O scheduler provides the best improvements for the three SSDs, as it can benefit from the synergy of the two optimizations.

SSD-C exhibits a different behavior from the other three SSDs. The cluster page size was extracted less clearly in SSD-C than in the other SSDs. As SSD-C seemingly uses some internal optimizations such as eliminating the overheads for read-modify-write, the performance gain by the parametrization of the block layer and I/O scheduler is modest for SSD-C.

However, for all four SSDs, there are few cases where the parametrization causes consistent performance drops. A minor modification of the two components with the extracted SSD parameters provides significant performance improvements for large requests, with little negative impact for the other cases. We have attempted to apply the extracted SSD parameters only to the two components, since our goal is to demonstrate the usefulness of the parameters. Further optimizations of applications or operating systems are open for future work, and we expect the knowledge on the critical SSD parameters to help to improve the performance of I/O-intensive applications.

7 Conclusion

In this paper, we proposed a new methodology that can extract several important parameters affecting the performance of SSDs and apply them to two components of a Linux operating system to improve the bandwidth of SSDs. The parameters discussed in this paper include clustered page size, clustered block size, and the size of read/write buffer. By optimizing the OS components with the extracted parameters, the write bandwidths for postmark and filebench improved by up to 24 percent and 321 percent, respectively.

Although the methodology requires minimal assumptions on the common architecture of SSDs, we believe the methodology is generic enough to be applied for many currently available SSD designs. Such understanding of the internal parameters of SSDs not only help to improve the system performance by parameter-aware optimizations, but also help to model SSD systems more accurately for system studies. Also, the performance improvement we achieved suggests the potential benefits of making the parameters of SSDs publicly available by the manufacturers. Despite of the reluctance of SSD manufacturers to open the internal architectures of SSDs, this study shows that even a limited information of SSD internals can improve file system performance significantly for certain cases.

References

[1] Samsung Elec., "Samsung SSD," http://www.samsung.com/global/business/semiconductor/products/flash/ssd/2008/home/home.html, 2009.

[2] R.V. Meter, "Observing the Effects of Multi-Zone Disks," *Proc. USENIX Ann. Technical Conf. (ATC '97),* p. 2, 1997.

[3] J. Schindler, J.L. Griffin, C.R. Lumb, and G.R. Ganger, "Track-Aligned Extents: Matching Access Patterns to Disk Drive Characteristics," *Proc. USENIX Conf. File and Storage Technologies (FAST '02),* pp. 259-274, 2002.

[4] R.Y. Wang, T.E. Anderson, and D.A. Patterson, "Virtual Log Based File Systems for a Programmable Disk," *Proc. Third Symp. Operating Systems Design and Implementation (OSD '99),* pp. 29-43, 1999.

[5] E.K. Lee and R.H. Katz, "An Analytic Performance Model of Disk Arrays," *Proc. ACM SIGMETRICS Conf.,* pp. 98-109, 1993.

[6] J.-H. Kim, D. Jung, J.-S. Kim, and J. Huh, "A Methodology for Extracting Performance Parameters in Solid State Disks (ssds)," *Proc. IEEE/ACM Int'l Symp. Modeling, Analysis, and Simulation of Computer and Telecomm. Systems (MASCOTS '09),* pp. 133-143, 2009.

[7] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," *Proc. USENIX Ann. Technical Conf. (ATC '08),* pp. 57-70, 2008.

[8] J. Seol, H. Shim, J. Kim, and S. Maeng, "A Buffer Replacement Algorithm Exploiting Multi-Chip Parallelism in Solid State Disks," *Proc. Int'l Conf. Compilers, Architecture, and Synthesis for Embedded Systems (CASE '09),* pp. 137-146, 2009.

[9] C. Hyun, J. Choi, Y. Oh, D. Lee, E. Kim, and S.H. Noh, "A Performance Model and File System Space Allocation Scheme for SSDs," *Proc. Int'l Symp. Massive Storage Systems and Technologies (MSST '10),* pp. 1-6, 2010.

[10] Samsung Elec., "NAND Flash Memory," http://www.samsung.com/global/business/semiconductor/products/flash/Products_NANDFlash.html, 2009.

[11] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System," *Proc. USENIX Technical Conf.,* pp. 13-13, 1995.

[12] J. Kim, J.M. Kim, S. Noh, S.L. Min, and Y. Cho, "A Space-Efficient Flash Translation Layer for CompactFlash Systems," *IEEE Trans. Consumer Electronics,* pp. 366-375, 2002.

[13] C. Park, P. Talawar, D. Won, M. Jung, J. Im, S. Kim, and Y. Choi, "A High Performance Controller for NAND Flash-Based Solid State Disk (NSSD)," *Proc. Non-Volatile Semiconductor Memory Workshop (NVSMW '06),* pp. 17-20, 2006.

[14] J.H. Kim, S.H. Jung, and Y.H. Song, "Cost and Performance Analysis of NAND Mapping Algorithms in a Shared-bus Multi-chip Configuration," *Proc. Int'l Workshop Software Support for Portable Storage (IWSSPS '08),* pp. 33-39, 2008.

[15] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings," *Proc. ACM Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '09),* pp. 229-240, 2009.

[16] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "A Superblock-Based Flash Translation Layer for NAND Flash Memory," *Proc. Int'l Conf. Embedded Software (EMSOFT '06),* pp. 161-170, 2006.

[17] Y.-G. Lee, D. Jung, D. Kang, and J.-S. Kim, "μ-FTL:: A Memory-Efficient Flash Translation Layer Supporting Multiple Mapping Granularities," *Proc. Int'l Conf. Embedded Software (EMSOFT '08),* pp. 21-30, 2008.

[18] B.L. Worthington, G.R. Ganger, Y.N. Patt, and J. Wilkes, "On-Line Extraction of SCSI Disk Drive Parameters," *Proc. ACM SIGMETRICS Conf.,* pp. 146-156, 1995.

[19] P.J. Shenoy and H.M. Vin, "Cello: A Disk Scheduling Framework for Next Generation Operating Systems," *Proc. ACM SIGMETRICS Conf.,* pp. 44-55, 1998.

[20] B.L. Worthington, G.R. Ganger, and Y.N. Patt, "Scheduling Algorithms for Modern Disk Drives," *Proc. ACM SIGMETRICS Conf.,* pp. 241-251, 1994.

[21] G.R. Ganger, B.L. Worthington, and Y.N. Patt, "The DiskSim Simulation Environment," technical report, 1998.

[22] D. Kotz, S.B. Toh, and S. Radhakishnan, "A Detailed Simulation Model of the HP 97560 Disk Drive," technical report, Dartmouth College, 1994.

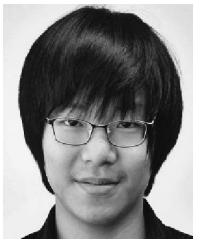
[23] C. Ruemmler and J. Wilkes, "An Introduction to Disk Drive Modeling," *Computer,* vol. 27, no. 3, 17-28, Mar. 2000.

[24] J. Schindler and G.R. Ganger, "Automated Disk Drive Characterization, CMU," Technical Report, CMU-CS-99-176, Carnegie Mellon University, Pittsburgh, PA, Dec. 1999.

[25] A.M. Caulfield, L.M. Grupp, and S. Swanson, "Gordon: Using Flash Memory to Build Fast, Power-Efficient Clusters for Data-Intensive Applications," *Proc. ACM Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '09),* pp. 217-228, 2009.

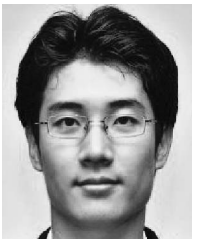
[26] A.C. Arpaci-Dusseau and R.H. Arpaci-Dusseau, "Information and Control in Gray-Box Systems," *Proc. ACM Symp. Operating Systems Principles (SOSP '01),* pp. 43-56, 2001.

[27] N. Joukov, A. Traeger, R. Iyer, C.P. Wright, and E. Zadok, "Operating System Profiling Via Latency Analysis," *Proc. Symp. Operating Systems Design and Implementation (OSDI '06)*, pp. 89-102, 2006.
[28] K. Yotov, K. Pingali, and P. Stodghill, "Automatic Measurement of Memory Hierarchy Parameters," *Proc. ACM SIGMETRICS Conf.*, pp. 181-192, 2005.
[29] T.E. Denehy, J. Bent, F.I. Popovici, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau, "Deconstructing Storage Arrays," *Proc. ACM Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '04)*, pp. 59-71, 2004.
[30] N. Talagala, R. Arpaci-Dusseau, and D. Patterson, "Micro-Benchmark Based Extraction of Local and Global Disk," Technical Report, CSD-99-1063, Univ. of California at Berkeley, CA, 2000.
[31] H.S. Gunawi, N. Agrawal, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, and J. Schindler, "Deconstructing Commodity Storage Clusters," *Proc. Int'l Symp. Computer Architecture (ISCA '05)*, pp. 60-71, 2005.
[32] N.C. Burnett, J. Bent, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau, "Exploiting Gray-Box Knowledge of Buffer-Cache Management," *Proc. USENIX Ann. Technical Conf. (ATC '02)*, pp. 29-44, 2002.
[33] M. Sivathanu, V. Prabhakaran, F.I. Popovici, T.E. Denehy, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau, "Semantically-Smart Disk Systems," *Proc. USENIX Conf. File and Storage Technologies (FAST '03)*, pp. 73-88, 2003.
[34] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S.H. Noh, "Disk Schedulers for Solid State Drivers," *Proc. Int'l Conf. Embedded Software (EMSOFT '09)*, pp. 295-304, 2009.
[35] D.P. Bovet and M. Cesati, "Understanding the Linux Kernel," O'Reilly Media Inc., 2005.
[36] S. Iyer and P. Druschel, "Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O," *Proc. ACM Symp. Operating Systems Principles (SOSP '01)*, pp. 117-130, 2001.
[37] J. Katcher, "PostMark: A New File System Benchmark," http://www.netapp.com/technology/level3/3022.html, TR3022, 1997.
[38] "FileBench," http://www.solarisinternals.com/wiki/index.php/FileBench, 2011.

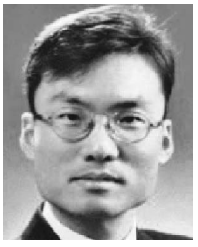


Jaehong Kim received the BS degree in computer engineering from Sung Kyun Kwan University, Suwon, South Korea, in 2008, and the MS degree in computer science, in 2010, from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea. He is currently a PhD candidate at KAIST. His research interests include flash memory, cloud computing, and virtualization.

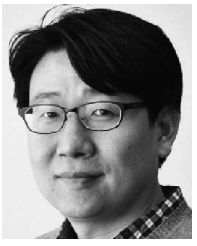
Sangwon Seo received the BS degree in computer engineering from Kyung-Hee University, Seoul, South Korea, in 2008, and the MS degrees in computer science, in 2010, from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, and Technische Universitat (TU), Berlin, Germany, respectively. He is currently a PhD candidate at KAIST. His research interests include distributed system, cloud computing, and virtualization.



Dawoon Jung received the BS, MS, and PhD degrees in computer science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, in 2002, 2004, and 2009, respectively. He is currently a senior software engineer of the Flash Software Development Team, Memory business Samsung Electronics. His research interests include operating systems, embedded systems, and flash-based storage systems.



Jin-Soo Kim received the BS, MS, and PhD degrees in computer engineering from Seoul National University, Korea, in 1991, 1993, and 1999, respectively. He is currently an associate professor in Sung Kyun Kwan University, Suwon, South Korea. Before joining Sung Kyun Kwan University, he was an associate professor at Korea Advanced Institute of Science and Technology (KAIST) from 2002 to 2008. He was also with the Electronics and Telecommunications Research Institute (ETRI) from 1999 to 2002, as a senior member of research staff, and with the IBM T.J. Watson Research Center as an academic visitor from 1998 to 1999. His research interests include embedded systems, storage systems, and operating systems. He is a member of the IEEE and the IEEE Computer Society.



Jaehyuk Huh received the BS degree in computer science from Seoul National University, South Korea, and the MS and PhD degrees in computer science from the University of Texas at Austin. He is an assistant professor of computer science at Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea. His research interests are in computer architecture, parallel computing, virtualization, and system security. He is a member of the IEEE.