Appears in the 17^{th} International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems

A Methodology for Extracting Performance Parameters in Solid State Disks (SSDs)

Jae-Hong Kim, Dawoon Jung
Computer Science Department
Korea Advanced Institute of Science and Technology (KAIST), South Korea
{jaehong, dwjung}@camars.kaist.ac.kr

Jin-Soo Kim
School of Information and Communication Engineering
Sungkyunkwan University, South Korea
jinsookim@skku.edu

Jaehyuk Huh
Computer Science Department
Korea Advanced Institute of Science and Technology (KAIST), South Korea
jhuh@cs.kaist.ac.kr

***Abstract*—Solid state disks (SSDs) consisting of NAND flash memory are being widely used in laptops, desktops, and even enterprise servers. SSDs have many advantages over hard disk drives (HDDs) in terms of reliability, performance, durability, and power efficiency. Typically, the internal hardware and software organization varies significantly from SSD to SSD 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. 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 type of NAND flash memory used, (4) the size of read buffer, and (5) the size of write buffer. Obtaining these parameters will allow us to understand the internal architecture of the target SSD better and to get the most performance out of SSD by performing SSD-specific optimizations.[1]

I. 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 [24].

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 disk head [17], [26], [28]. Understanding these parameters also helps to model and analyze disk performance more accurately [15].

[1]This work was supported by the Korea Science and Engineering Foundation(KOSEF) grant funded by the Korea government(MEST) (No. R01-2007-000-11832-0).

However, SSDs have different performance parameters compared with HDDs due to the difference in the characteristics of underlying storage media. 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 additional overhead [1]. To avoid this overhead, it is helpful to issue read/write requests in a multiple of the clustered page size. The problem is that the actual value of such a parameter varies depending on the type of NAND flash memory employed and the internal architecture of SSDs.

In this paper, we propose a methodology which can extract several essential parameters affecting the performance of SSDs. The parameters considered in this paper include the size of read/write unit, the size of erase unit, the type of NAND flash memory used, 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 latency. By varying the request size and the access pattern, important performance parameters of a commercial SSD can be successfully estimated.

The rest of the paper is organized as follows. Section II overviews the characteristics of NAND flash memory, and SSDs, and describes some related work. In Section III, the detailed methodology for extracting several performance parameters of SSDs is described. We present experimental results in Section IV to show the effectiveness of our approach. In Section V, we discuss future work and conclude the paper.

II. Background

A. 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). Figure 1 illustrates the organization of NAND flash where a block contains 128 4 KB-pages.

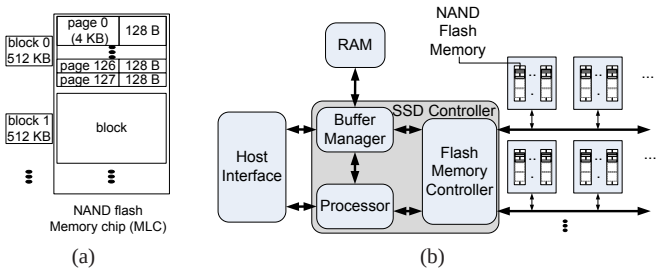


Fig. 1. NAND flash memory internals (a) and the block diagram of an SSD (b)

NAND flash memory is different from DRAMs and HDDs in a number of aspects. First, the latency of read and write operations is asymmetric as shown in Table I. 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 is written to the page. Moreover, the lifetime of NAND flash memory is limited by 10,000-100,000 program/erase cycles [22].

According to the manufacturing technology, NAND flash memory can be classified into two types, SLC (Single-Level Cell) and MLC (Multi-Level Cell). In SLC NAND flash memory, a memory cell only represents one-bit data like conventional memory devices. On the contrary, the voltage level of a cell in MLC NAND flash memory is minutely divided into four or more levels, and this allows a cell to express two or more bits. As a result, MLC NAND flash memory provides higher density and larger capacity than SLC NAND. Currently, two-bit MLC NAND flash memory where a cell represents two bits is commercialized.

Although MLC NAND flash memory significantly reduces cost per bit, its operational characteristics are worse than those of SLC NAND. First of all, the programming latency is increased by three or four times, and the read performance is slightly degraded. In addition, the programming latency of MLC NAND fluctuates in a relatively wide range. For example, two-bit MLC NAND flash memory typically exhibits two notable ranges of programming latency. This is due to the device characteristics of MLC NAND where two different pages (called *pair pages*) within a block are internally linked together; programming the first pages can be done quickly, but programming the second pages require more time to finish [10]. Another disadvantage is that the bit error rate of MLC NAND flash memory is higher than that of SLC NAND [14]. This enforces the use of stronger ECCs. Finally, the lifetime of MLC NAND is reduced to 5,000-15,000 program/erase cycles. Table I compares the characteristics of contemporary SLC and MLC NAND flash memory [20], [21][2].

B. 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 Figure 1-(b). The host interface control logic transfers command and data from/to the host via 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 the write requests or accessed data. The entire operations are governed by a firmware, usually called a flash translation layer (FTL) [9], [11], run by the SSD controller.

[2]Note that the actual page size or the block size may vary depending on the flash memory chip model and the manufacturer. For example, some SLC NAND flash memory has 4KB page size [23].

TABLE I
CHARACTERISTICS OF SLC [21] AND MLC [20] NAND FLASH MEMORY

	SLC NAND	MLC NAND
page size	(2048+64) B	(4096+128) B
block size	(128+4) KB	(512+16) KB
# pages/block	64	128
read latency	77.8μs (2 KB)	165.6μs (4 KB)
write latency	252.8μs (2 KB)	905.8μs (4 KB)
erase latency	1500μs (128 KB)	1500μs (512 KB)

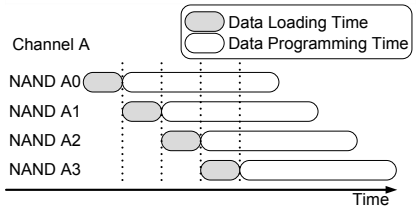


Fig. 2. 4-way interleaving on the same bus

Recently, developing a high-performance SSD has been a key design goal. To increase the read/write bandwidth of SSDs, many SSDs make use of the 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. Figure 2 illustrates a situation where 4-way interleaving is performed on the channel (or *bus*) to hide the latency of page programming in NAND flash memory. If there are multiple independent channels, the read/write bandwidth of SSDs can be accelerated further by exploiting inter-channel and intra-channel parallelism [18], [12].

C. 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 [6] 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 [8], [16].

D. Related Work

Extracting performance-critical parameters for HDDs has been widely studied for designing sophisticated disk scheduling algorithms [30], [27], [29] and characterizing the performance of HDDs to build the detailed disk simulator [5], [13], [19], [25]. However, as SSDs have completely different architecture compared to HDDs, the methodology for extracting performance parameters in HDDs is different from that of SSDs. Our work introduces a methodology for extracting performance parameters of SSDs and show the results of four commercial SSDs. To the best of our knowledge, our work is among the first to examine the performance parameters obtained from commercial SSDs.

Agrawal et al. provide a good overview of the SSD architecture and present various tradeoffs in designing SSDs [1]. In order to obtain several tradeoffs for SSDs, they developed a modified version of the DiskSim simulator. Using this 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 play important roles in the performance of SSDs.

Recently, Adrian et al. have developed Gordon, a flash memory-based cluster architecture for large-scale data-intensive applications [3]. The architecture of Gordon is similar to that of SSDs in that it uses NAND flash memory and flash memory controller. The controller supports the FTL functionality and multi-channel structure that can exploit parallelism. To acquire the high I/O performance of data-intensive work, they tune some performance parameters such as the clustered page size that will be introduced in Section III-A. They also found that the performance of NAND flash-based storage is affected by a number of parameters.

Our methodology for extracting performance parameters is similar to the gray-box approach [2], [7]. The gray-box approach is a methodology that acquires information which is unavailable or cumbersome to maintain. 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 used in the system. Yotov et al. have applied the gray-box approach to the memory hierarchy [31]. 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 [4]. They employ several algorithms to determine the critical parameters of a RAID system, including the number of disks, chunk size, level of redundancy, and layout scheme. Similarly, based on the existing knowledge on SSDs, we can devise a methodology for extracting essential performance parameters of SSDs.

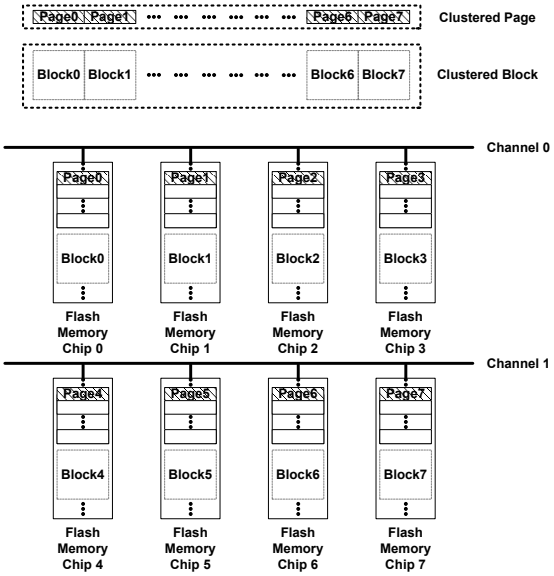


Fig. 3. An example of a clustered page (block), which is interleaved in eight flash memory chips on two channels

III. METHODOLOGY

A. Parameters in SSDs

The performance parameters of SSDs are different from those of HDDs as described in Section I. We now describe some of important performance parameters of SSDs before we present our methodology in detail.

Clustered Page: We define a *clustered page* as an internal unit of read or write operation used in SSDs. As discussed in Section II-B, 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. Figure 3 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 Figure 3, 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 very critical parameter for application-level I/O performance as shown in Gordon [3]. If we adjust the size of data transfer to the clustered page size, we can enhance the I/O performance more effectively, since 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.

Clustered Block: We define a *clustered block* as an internal unit of erase operation used in SSDs. Similar to the clustered

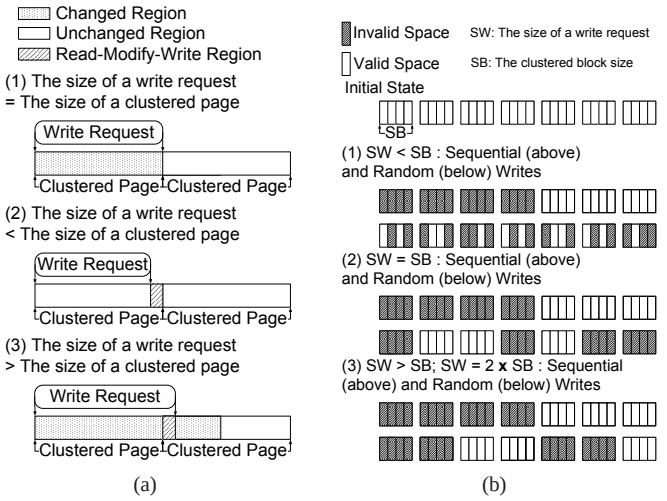


Fig. 4. (a): A write request that is aligned (1) and unaligned (2, 3) to the clustered page boundary (b): Sequential vs. random writes when the request size is smaller than (1), equal to (2), or larger than (3) the clustered block size

page, SSDs often combine several blocks coming from different NAND flash chips into a single clustered block. Figure 3 shows an example of a clustered block which consists of eight physical blocks. The use of the clustered block improves 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.

Type of NAND Flash: Because of the contrasting characteristics of SLC NAND and MLC NAND, SLC SSDs and MLC SSDs are currently aiming at different market segments. SLC SSDs are typically used for servers and high-performance storage due to higher reliability and higher performance offered by SLC NAND. On the other hand, MLC SSDs are preferred for laptops and desktop PCs because of their lower price.

For end-users, it is sometimes useful to find out which type of NAND flash memory has been employed in their SSDs. In this paper, we suggest a methodology to identify the type of NAND flash without cracking in to the hardware.

Read/Write Buffer: Many SSD controllers use a part of DRAM as 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 III-E and Section III-F.

The read buffer size or the write buffer size can be a valuable hint to 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.

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., 1024 KB) */
ISW, /* the increment in size (in KB, e.g., 2 KB) */
NI /* the number of iteration (e.g., 64) */

```
1:  SW ⇐ 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 ≤ TSW do
4:      SW ⇐ SW + ISW
5:      lseek(F, 0, SEEK_SET) /* set the file pointer to the offset 0 */
6:      Start ⇐ 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 ⇐ gettimeofday()
12:     print the elapsed time by using Start and End
13: end while
```

There are many possible architectural organizations and tradeoffs in SSDs as discussed in [1]. As a result, the specifics of internal hardware architecture and software algorithm used in SSDs differ greatly from vendor to vendor. In spite of this, our methodology is successful in extracting the aforementioned parameters from four commercial SSDs. This is because 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, the address mapping and garbage collection policies, etc. We develop a generic methodology based on the common overheads and characteristics found in all target SSDs, which are independent of the specific model details. The proposed methodology is applicable to many SSDs as long as they use the interleaving technique to enhance the I/O bandwidth and employ a variant of block-mapping or page-mapping FTLs.

B. Measuring the Clustered Page Size

As described in the previous subsection, 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 [1] incurs extra flash read operations, increasing the write latency.

Consider a case (1) in Figure 4(a), 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 Figure 4(a) 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 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 2 ProbeClusteredBlock

Input: F, /* file descriptor for the raw disk device opened with O_DIRECT */
SP, /* the clustered page size obtained in Section III-B (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., ($8 \times 1024 \times 1024$ KB)) */
NP /* the initial number of clustered pages (e.g., 2). $NP \times SP$ is the actual size of write requests */

```
1:  NI ⇐ 0 /* the number of iteration */
2:  while NP ≤ TNP do
3:      NP ⇐ NP × 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 ⇐ gettimeofday()
6:      lseek(F, 0, SEEK_SET) /* set the file pointer to the offset 0 */
7:      NI ⇐ TSW/(NP × SP)
8:      for i = 1 to NI do
9:          write_file(F, NP × SP) /* write (NP × SP) KB of data to F */
10:         ATA_FLUSH_CACHE() /* flush the write buffer */
11:     end for
12:     End ⇐ gettimeofday()
13:     print the elapsed time of sequential writes by using Start and End
14:     write_init(F)
15:     Start ⇐ gettimeofday()
16:     for i = 1 to NI do
17:         R ⇐ rand()%NI /* choose R randomly */
18:         R ⇐ R × (NP × SP) × 1024
19:         lseek(F, R, SEEK_SET)
20:         write_file(F, NP × SP)
21:         ATA_FLUSH_CACHE()
22:     end for
23:     End ⇐ gettimeofday()
24:     print the elapsed time of random writes by using Start and End
25: 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 [11]. 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 subsections.

C. 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 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 Figure 4(b) 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 Figure 4(b), 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 3 ProbeNANDType

Input: F, /* file descriptor for the raw disk device opened with O_DIRECT */
SP, /* the clustered page size obtained in Section III-B (in KB, e.g., 16 KB) */
SB, /* the clustered block size obtained in Section III-C (in KB, e.g., 4×1024 KB)*/
NB /* the number of clustered blocks (e.g., 16) */

```
1:  AW ⇐ 0 /* the amount of written data so far (in KB) */
2:  write_init(F) /* initialize the target SSD */
3:  lseek(F, 0, SEEK_SET) /* set the file pointer to the offset 0 */
4:  while AW ≤ NB × SB do
5:      AW ⇐ AW + SP
6:      Start ⇐ gettimeofday()
7:      write_file(F, SP) /* write SP KB of data to F */
8:      ATA_FLUSH_CACHE() /* flush the write buffer */
9:      End ⇐ gettimeofday()
10:     print the elapsed time by using Start and End
11: end while
```

D. Identifying the Type of NAND flash

As explained in Section II-A, NAND flash memory used in SSDs can be classified into two types, SLC and MLC, according to the manufacturing technology. Because of distinctive device characteristics between SLC and MLC, the distribution of write latencies in SLC NAND is different from that of write latencies in MLC NAND.

While SLC NAND shows a relatively consistent write latency, the write latency in MLC NAND fluctuates severely depending on the location of the page written (cf. Section II-A). Therefore, the latency of each write request falls into a narrow range in SLC NAND, but that is grouped into two or more clusters in MLC NAND.

To identify the type of NAND flash memory, our microbenchmark issues a number of write requests that can fill the entire clustered block, and measures the elapsed time of each request. All the write requests are aligned to the boundary of clustered page, and the request size is equal to the clustered page size. Procedure 3 shows the pseudocode of the benchmark.

E. 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.

Procedure 4 ProbeReadBuffer

Input: F, /* file descriptor for the raw disk device opened with O_DIRECT */
TSR, /* the total size to read (in KB, e.g., 1024 KB) */
ISR /* the increment in size (in KB, e.g., 1 KB) */

```
1:  SR ⇐ 0 /* the size of read request (in KB) */
2:  write_init(F) /* initialize the target SSD */
3:  while SR ≤ TSR do
4:      SR ⇐ SR + ISR
5:      R ⇐ rand()%1024 /* choose R randomly */
6:      lseek(F, 1024 × 1024 × 1024 + R × 16 × 1024 × 1024, SEEK_SET)
        /* set the file pointer randomly */
7:      read_file(F, 16 × 1024) /* read 16 MB of data from F */
8:      R ⇐ rand()%63
9:      lseek(F, R × 16 × 1024 × 1024, 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 × 16 × 1024 × 1024, SEEK_SET)
12:     Start ⇐ gettimeofday()
13:     read_file(F, SR)
14:     End ⇐ gettimeofday()
15:     print the elapsed time by using Start and End
16: end while
```

Procedure 5 ProbeNANDReadLatency

Input: F, /* file descriptor for the raw disk device opened with O_DIRECT */
TSR, /* the total size to read (in KB, e.g., 1024 KB) */
ISR /* the increment in size (in KB, e.g., 1 KB) */

```
1:  SR ⇐ 0 /* the size of read request (in KB) */
2:  write_init(F) /* initialize the target SSD */
3:  while SR ≤ TSR do
4:      SR ⇐ SR + ISR
5:      R ⇐ rand()%1024 /* choose R randomly */
6:      lseek(F, 1024 × 1024 × 1024 + R × 16 × 1024 × 1024, SEEK_SET)
        /* set the file pointer randomly */
7:      read_file(F, 16 × 1024) /* read 16 MB of data from F */
8:      R ⇐ rand()%63
9:      lseek(F, R × 16 × 1024 × 1024, SEEK_SET) /* set the file pointer
        randomly (We assume that the size of read buffer is smaller than 16 MB) */
10:     Start ⇐ gettimeofday()
11:     read_file(F, SR) /* read SR KB of data from F */
12:     End ⇐ gettimeofday()
13:     print the elapsed time by using Start and End
14: end while
```

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 4 and 5.

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[3]. 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.

[3]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.

Procedure 6 ProbeWriteBuffer

Input: F, /* file descriptor for the raw disk device opened with O_DIRECT */
TSW, /* the total size to write (in KB, e.g., 1024 KB) */
ISW, /* the increment in size (in KB, e.g., 1 KB) */
NI /* the number of iteration (e.g., 30) */

```
1:  SW ⇐ 0 /* the size of write request (in KB) */
2:  write_init(F) /* initialize the target SSD */
3:  while SW ≤ TSW do
4:      SW ⇐ 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 ⇐ gettimeofday()
9:          write_file(F, SW) /* write SW KB of data to F */
10:         End ⇐ gettimeofday()
11:         print the elapsed time by using Start and End
12:     end for
13: end while
```

Procedure 7 ProbeNANDWriteLatency

Input: F, /* file descriptor for the raw disk device opened with O_DIRECT */
TSW, /* the total size to write (in KB, e.g., 1024 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 ⇐ 0 /* the size of write request (in KB) */
2:  write_init(F) /* initialize the target SSD */
3:  while SW ≤ TSW do
4:      SW ⇐ SW + ISW
5:      for i = 1 to NI do
6:          ATA_FLUSH_CACHE() /* flush the write buffer */
7:          lseek(F, 16 × 1024 × 1024, SEEK_SET)
            /* We assume that the size of write buffer is smaller than 16 MB */
8:          write_file(F, 16 × 1024) /* write 16 MB of data to F */
9:          lseek(F, 0, SEEK_SET) /* set the file pointer to the offset 0 */
10:         Start ⇐ gettimeofday()
11:         write_file(F, SW) /* write SW KB of data to F */
12:         End ⇐ gettimeofday()
13:         print the elapsed time by using Start and End
14:     end for
15: end while
```

F. Measuring the Write Buffer Capacity

As discussed in Section III-A, 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.

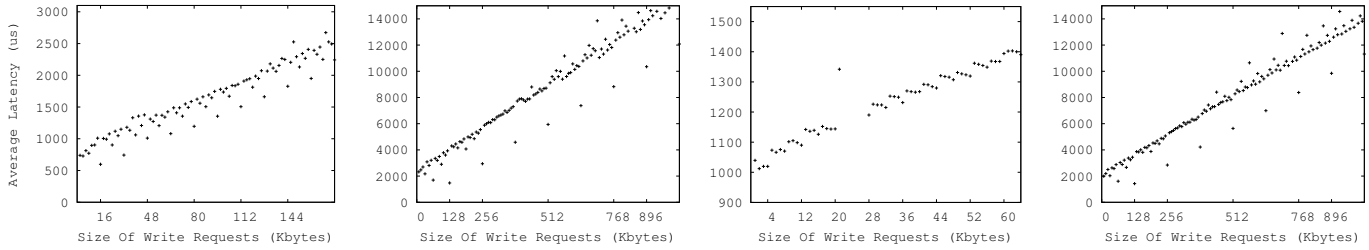
To determine whether the write request is handled by the write buffer or NAND flash memory, we have developed two microbenchmarks, ProbeWriteBuffer() and ProbeNANDWriteLatency(), as shown in Procedure 6 and 7. The former measures the time taken to write data into the write buffer, if any, while the latter is intended to measure the time to write the requested data to NAND flash memory.

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

TABLE II
THE 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



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

Fig. 5. The average write latency with varying the size of write requests

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.

IV. EXPERIMENTAL RESULTS

A. Experiment Environment

We ran the microbenchmarks described in Section III on a Linux-based system (kernel version 2.6.25.10). Our experimental system is equipped with a 2.0 GHz AMD Athlon 64 processor 3200+ and 2 GB of RAM. We attached 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. HDD is the system disk where the operating system is installed. In our experiments, we have evaluated four different SSDs that are commercially available from the market. The full details of each SSD used in this paper are summarized in Table II.

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 Queueing) as SSD-C is the only SSD which supports this feature.

B. The clustered Page Size

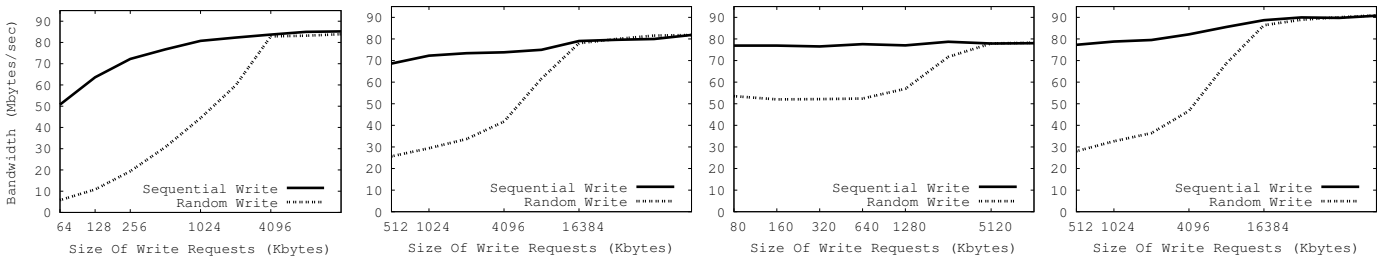
To estimate the clustered page size, we have measured the latency of each write request varying the request size up to 1024 KB (cf. Section III-B). Figure 5 plots the results obtained by running Procedure 1 on the tested SSDs. All the experiments for SSD-A, SSD-B, and SSD-D are performed with the write 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 Figure 5, 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 Figure 5(a), the latency drops sharply whenever the request size is a multiple of 16 KB. As described in Section III-B, this is because the data to be written are aligned to the clustered page boundary at these points, eliminating the need for read-modify-write operation. 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 Figure 5 is that there are several spikes in the write latency, most notably in Figure 5(b), (c), and (d). We suspect this is due to garbage collection which should be occasionally invoked to make free blocks.

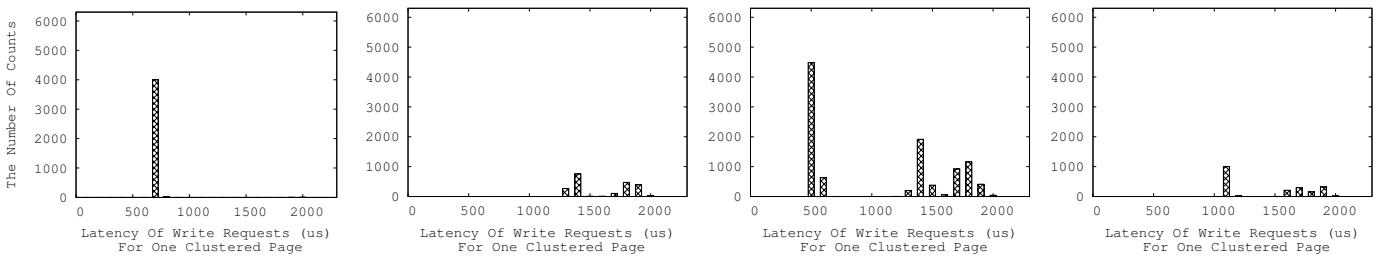
C. The clustered Block Size

To determine the clustered block size, the microbenchmark introduced in Section III-C measures the bandwidth of sequen-



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

Fig. 6. Sequential vs. random write bandwidth according to the size of write requests



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

Fig. 7. The distribution of write latencies

tial and random writes, increasing the request size up to 128 MB. Figure 6 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 Figure 6(a), we find that the bandwidth of sequential writes is higher than that of random writes when the size of write request is smaller than 4096 KB. If the request size is increased beyond 4096 KB, there is virtually no difference in the bandwidth. As mentioned in Section III-C, the bandwidth of random writes converges to that of sequential writes as the request size approaches to the clustered block size. This is mainly due to that the number of valid pages in a clustered block is getting smaller, reducing the overhead of garbage collection gradually. This suggests that the clustered block size of SSD-A is 4096 KB. Similarly, we can infer that the clustered block size of SSD-B, SSD-C, and SSD-D is 16384 KB, 5120 KB, and 16384 KB, respectively.

D. The Type of NAND Flash Memory

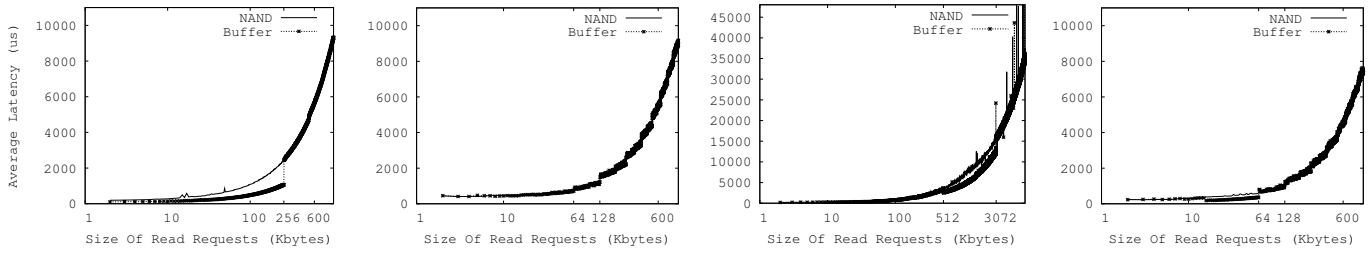
Depending on the type of NAND flash memory used in SSDs, the distribution of each write latency exhibits two representative patterns; a single cluster for SLC NAND and multiple clusters for MLC NAND. Figure 7 displays the distribution of a number of write latencies obtained from ProbeNANDType() in Section III-D.

In Figure 7, we can see that the distribution of the latency for writing one clustered page is grouped into a narrow range in SSD-A, but this is not the case for SSD-B, SSD-C, and SSD-D. Hence, we can easily identify that the type of NAND flash memory employed in SSD-A is SLC, while the rest of SSDs are composed of MLC NAND flash memory.

E. The Read Buffer Capacity

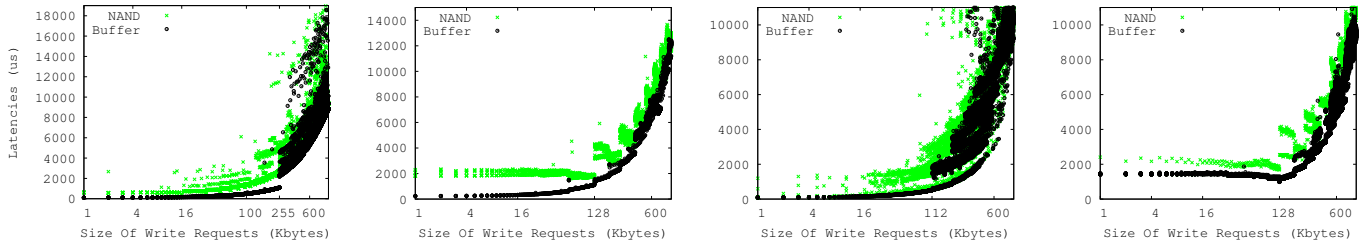
To estimate the capacity of the read buffer, we compare the latency measured by ProbeReadBuffer() with that obtained by ProbeNANDReadLatency(), varying the size of each read request. Figure 8 contrasts the results with respect to the read request size from 1 KB to 1024 KB (4096 KB for SSD-C). In Figure 8, the labels “NAND” and “Buffer” denote the latency obtained from ProbeNANDReadLatency() and from ProbeReadBuffer(), respectively. As described in Section III-E, 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 Figure 8(a), 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 3072 KB (SSD-C), or from 16 KB to 64 KB (SSD-D). Therefore, the read buffer size of SSD-C and SSD-D is 3072 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.



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

Fig. 8. The latency of read requests with increasing the size of read requests



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

Fig. 9. The latency of write requests with increasing the size of write requests

F. The Write Buffer Capacity

We have introduced two procedures, ProbeWriteBuffer() and ProbeNANDWriteLatency() in Section III-F. 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. Figure 9 plots the measured latencies for four commercial SSDs with various request sizes ranging from 1 KB to 1024 KB. In Figure 9, "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 Figure 9(a). 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. An interesting point is that the result of "NAND" is getting improved as the request size is increased from 1 KB to 128 KB. This phenomenon is related to the fact that the clustered page size of SSD-B is 128 KB (cf. Section IV-B); when the request size is less than the clustered page size, the latency suffers from the read-modify-write overhead. This overhead becomes smaller as the request size approaches to the clustered page size, and the amount of data read from the original clustered page gets smaller.

In SSD-D, the overall trend looks similar to SSD-B. However, when we compare the write latency of SSD-D with the read latency shown in Figure 8(d), we can notice that the former (approximately 1500 μsec) is much slower than the latter (approximately 200 μsec) with 1 KB request size. If the data were written to the buffer, there is no reason for the write request to take a significantly longer time compared to the read request. The only possible explanation is that, for the write requests less than or equal to 128 KB, SSD-D bypasses the write buffer and stores the data directly to NAND flash memory. Although it appears that SSD-D does not make use of any write buffer, we could not draw any conclusion using our methodology since the behavior of SSD-D is so different from other SSDs.

V. Conclusion

In this paper, we have proposed a new methodology that can extract several important parameters affecting the performance of SSDs. The parameters discussed in this paper include the clustered page size, the clustered block size, the type of NAND flash memory, and the size of read/write buffer.

The clustered page size is found by using the overhead associated with the read-modify-write operation, which arises when the write request is not aligned to the clustered page boundary. A clustered page in SSDs is the basic unit of alignment for read/write operations. Thus, the clustered page size obtained by our methodology can be used when designing and optimizing higher-level software to improve the I/O performance on SSDs.

A clustered block is the erase unit in SSDs. The erase operation is occurred when SSDs trigger the garbage collection process to make free blocks by cleaning obsolete pages. This garbage collection is one of main reasons that cause significant performance degradation in SSDs. Therefore, the clustered block size can be a useful hint when we update a large amount of data; if we issue write requests so that they are aligned to the clustered block boundary, the overhead of garbage collection will be minimized.

We have identified the type of NAND flash memory used in SSDs by investigating the distribution of write latencies. We have estimated the size of read/write buffer by comparing the time to handle the request from the buffer with the time to handle the same request from NAND flash memory. For this, we have designed our microbenchmarks carefully so that they can generate both buffer hit or miss scenarios.

For SSD-A, we have confirmed that all the parameter values we found are correct. From the specification of each SSD model, we have verified that the type of NAND flash memory identified by our methodology is also correct for all the tested SSDs. Unfortunately, however, other parameter values could not be validated due to the absence of enough information.

All of these parameters can be used to predict the performance of SSDs and to analyze their performance behaviors. In addition, those parameters will allow us to understand the internal architecture of the target SSD better and to achieve the best performance by performing SSD-specific optimizations. We will extend our methodology to cover other parameters, and refine it further through more case studies on commercial SSDs. We also plan to optimize I/O schedulers or file systems so they can take maximum advantage of the performance parameters introduced in this paper.

REFERENCES

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *ATC' 08: USENIX 2008 Annual Technical Conference*, pages 57–70, Berkeley, CA, USA, 2008. USENIX Association.

[2] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 43–56, New York, NY, USA, 2001. ACM.

[3] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 217–228, New York, NY, USA, 2009. ACM.

[4] T. E. Denehy, J. Bent, F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Deconstructing storage arrays. *SIGOPS Oper. Syst. Rev.*, 38(5):59–71, 2004.

[5] G. R. Ganger, B. L. Worthington, and Y. N. Patt. The disksim simulation environment. technical report cse-tr-358-98, dept. of electrical engineering and computer science, univ. of michigan. Technical report, 1998.

[6] A. Gupta, Y. Kim, and B. Urgaonkar. Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 229–240, New York, NY, USA, 2009. ACM.

[7] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok. Operating system profiling via latency analysis. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 89–102, Berkeley, CA, USA, 2006. USENIX Association.

[8] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee. A superblock-based flash translation layer for nand flash memory. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 161–170, New York, NY, USA, 2006. ACM.

[9] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *TCON'95: Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings*, pages 13–13, Berkeley, CA, USA, 1995. USENIX Association.

[10] D.-H. Kim and Y.-T. Lee. Flash memory device having multi-level cell and reading and programming method thereof. United States Patent, no. 7,035,144, February 2006.

[11] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, May 2002.

[12] 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. *IWSSPS '08: The 3rd International Workshop on Software Support for Portable Storage*, 3(3):33–39, Oct 2008.

[13] D. Kotz, S. B. Toh, and S. Radhakishnan. A detailed simulation model of the hp 97560 disk drive. technical report pcs-tr94-220, dept. of computer science, darthmouth college. Technical report, 1994.

[14] M. Lasser and K. Yair. Flash memory management method that is resistant to data corruption by power loss. United States Patent, no. 6,988,175, January 2006.

[15] E. K. Lee and R. H. Katz. An analytic performance model of disk arrays. In *SIGMETRICS '93: Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 98–109, New York, NY, USA, 1993. ACM.

[16] Y.-G. Lee, D. Jung, D. Kang, and J.-S. Kim. μ-ftl:: a memory-efficient flash translation layer supporting multiple mapping granularities. In *EMSOFT '08: Proceedings of the 8th ACM international conference on Embedded software*, pages 21–30, New York, NY, USA, 2008. ACM.

[17] R. V. Meter. Observing the effects of multi-zone disks. In *ATEC '97: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 1997. USENIX Association.

[18] 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). *Non-Volatile Semiconductor Memory Workshop, 2006. IEEE NVSMW 2006. 21st*, pages 17–20, 2006.

[19] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. pages 462–473, 2000.

[20] Samsung Elec. 2gx8 bit nand flash memory (k9gag08u0m). http://www.samsung.com/global/business/semiconductor/productInfo.do?fmly_id=672&partnum=K9GAG08U0M, 2009.

[21] Samsung Elec. 2gx8 bit nand flash memory (k9wag08u1a). http://www.samsung.com/global/business/semiconductor/productInfo.do?fmly_id=159&partnum=K9WAG08U1A, 2009.

[22] Samsung Elec. Nand flash memory. http://www.samsung.com/global/business/semiconductor/products/flash/Products_NANDFlash.html, 2009.

[23] Samsung Elec. Samsung product selection guide. http://www.samsung.com/global/business/semiconductor/support/brochures/downloads/memory/psg_memory_200804.pdf, 2009.

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

[25] J. Schindler and G. R. Ganger. Automated disk drive characterization, cmu scs technical report cmu-cs-99-176. Technical report, 1999.

[26] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned extents: Matching access patterns to disk drive characteristics. In *FAST '02: Proceedings of the Conference on File and Storage Technologies*, pages 259–274, Berkeley, CA, USA, 2002. USENIX Association.

[27] P. J. Shenoy and H. M. Vin. Cello: A disk scheduling framework for next generation operating systems. Technical report, Austin, TX, USA, 1998.

[28] R. Y. Wang, T. E. Anderson, and D. A. Patterson. Virtual log based file systems for a programmable disk. Technical report, Berkeley, CA, USA, 1999.

[29] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling algorithms for modern disk drives. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 241–251, New York, NY, USA, 1994. ACM.

[30] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes. On-line extraction of scsi disk drive parameters. *SIGMETRICS Perform. Eval. Rev.*, 23(1):146–156, 1995.

[31] K. Yotov, K. Pingali, and P. Stodghill. Automatic measurement of memory hierarchy parameters. In *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 181–192, New York, NY, USA, 2005. ACM.