Disk Schedulers for Solid State Drives

Jaeho Kim
School of Computer Science
University of Seoul
Seoul, Korea
kjhnet07@uos.ac.kr

Yongseok Oh
School of Computer Science
University of Seoul
Seoul, Korea
ysoh@uos.ac.kr

Eunsam Kim
School of Computer & Inform. Eng.
Hongik University
Seoul, Korea
eskim@hongik.ac.kr

Jongmoo Choi
Division of Information and CS
Dankook University
Seoul, Korea
choijm@dankook.ac.kr

Donghee Lee*
School of Computer Science
University of Seoul
Seoul, Korea
dhl_express@uos.ac.kr

Sam H. Noh
School of Computer & Inform. Eng.
Hongik University
Seoul, Korea
samhnoh@hongik.ac.kr

ABSTRACT

In embedded systems and laptops, flash memory storage such as SSDs (Solid State Drive) have been gaining popularity due to its low energy consumption and durability. As SSDs are flash memory based devices, their performance behavior differs from those of magnetic disks. However, little attention has been paid on how to exploit SSDs from the disk scheduling algorithm view point. In this paper, we first describe behaviors of SSDs that inspires us to design a new disk scheduler for the Linux operating system. Specifically, read service time is almost constant in an SSD while write service time is not. Moreover, appropriate grouping of write requests eliminates any ordering-related restrictions and also maximizes write performance. From these observations, we propose two disk schedulers: *IRBW-FIFO* and *IRBW-FIFO-RP*. Both schedulers arrange write requests into bundles of an appropriate size while read requests are independently scheduled. Then, the IRBW-FIFO scheduler provides complete FIFO ordering to each bundle of write requests and each individual read requests while the IRBW-FIFO-RP scheduler gives higher priority to read requests than the bundles of write requests. We implement these schedulers in Linux 2.6.23, and results of executing our set of benchmark programs shows that performance improvements of up to 17% compared to existing Linux disk schedulers are achieved.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management – Secondary storage; D.4.8 [**Operating Systems**]: Performance – Measurements

General Terms

Performance, Design, Experimentation.

* Corresponding Author

Keywords

Solid State Drive, Disk scheduler, Linux, Implementation Study.

1. Introduction

For decades, the magnetic disk has been the most popular storage device in computer systems, and its prosperity has demanded that software be optimized for its inherent characteristics. For example, as the magnetic disk has moving heads and rotating platters popular disk schedulers are based on the elevator algorithm to minimize head movement [1]. Another example is the Fast File System (FFS) that divides the file system space into a number of cylinder-groups and tries to allocate relevant data within a cylinder group to minimize head movement [2]. Also, exploiting locality of read/write requests, which has been a long lasting research area for disk cache management [3-7], is an effort to hide the inherent delay incurred by magnetic disks. Even applications such as multimedia or database systems have been tuned to comply with the inherent properties of magnetic disks.

Recently, a new type of storage, specifically the SSD (Solid State Drive), has been introduced and is gaining popularity in embedded systems and laptops. Even enterprise servers are showing interest in this new type of storage to boost performance and save energy. As SSDs use NAND flash memory chips as their storage medium, their behavior differs from that of magnetic disks. Obviously SSDs do not incur seek times and rotational delays anymore as moving parts no longer exist in an SSD. However, there are other subtle differences which we discuss later.

The main goal of this paper is to investigate the inherent nature and performance behavior of flash memory storage from the disk scheduler perspective. In particular, an SSD inherits some distinct properties from the raw NAND flash memory chip that it is composed of. Investigation of properties of flash memory storage have revealed that read service time is generally constant while write service time is not. On the other hand, maximizing write throughput requires elaborate grouping of write requests in SSDs, and we will present some experimental data regarding this idea in this paper.

To exploit the inherent properties of flash memory, we devise two disk schedulers for SSD; *IRBW-FIFO (Individual Read Bundled Write FIFO)* and *IRBW-FIFO-RP (Individual Read Bundled Write FIFO with Read Preference)*. They both arrange write requests into bundles of appropriate sizes. The IRBW-FIFO scheduler applies FIFO ordering to the bundles of write requests and individual read requests. The IRBW-FIFO-RP scheduler maintains separate FIFO ordering among read requests and among the bundles of write requests, and then gives higher priority to read requests than to the bundles of write requests. This read preference is based on the well known fact that synchronous read requests must be served before asynchronous write requests. Benchmark results of these schedulers implemented in Linux show that IRBW-FIFO and IRBW-FIFO-RP improve performance up to 4% and 17%, respectively, over existing I/O schedulers.

The rest of the paper is organized as follows. In the next section, we provide background knowledge of flash memory storage, Linux schedulers, and related works. In Section 3, we discuss behaviors observed due to Linux disk schedulers on SSDs. We discuss some experiments and their results that form the basis for developing the new disk schedulers as well as the new disk schedulers themselves in Section 4. In Section 5, we present experimental results of the proposed schedulers, and then finally, we conclude this paper in Section 6.

2. Flash Memory Storage and Linux Disk I/O Scheduler

In the last decade, more and more embedded systems have moved to flash memory as their permanent data store, and recently, even enterprise database systems have started to consider SSDs as a means to improve performance and also to save energy. Lee et al. showed that adopting SSDs for transaction log, rollback segments, and temporary tables can enhance performance of database applications by up to an order of magnitude [8]. Flash memory storage is now regarded to have the potential to change the landscape of the memory hierarchy in computer systems [9]. The benefits of flash memory storage can be maximized by understanding and exploiting properties of flash memory. In this section, we briefly describe the properties, drawbacks, and performance behaviors of flash memory storage.

2.1 NAND Flash Memory Storage

NAND flash memory is a kind of non-volatile memory that has evolved from EEPROM (Electrically Erasable Programmable Read-Only Memory). As it inherits its essential structure from EEPROM, it has the following distinctive properties. A NAND flash memory chip has a number of blocks that can be erased independently. Each block has a fixed number of pages where data can be written/read to/from. Before data can be written to an already used page, the block containing the page must be erased as overwriting is not allowed. A typical block size and page size is 16~256 KB and 0.5~4KB, respectively. Block erase takes around 1.5~2 milliseconds, while reading and writing a page takes tens of microsecond and hundreds of microsecond, respectively [10, 11].

Due to the constraint that page overwrite is forbidden, many flash memory storage systems have used complicated software techniques integrated into what is referred to as the FTL (Flash Translation Layer) [12-14]. The FTL reserves some redundant blocks aside from the data blocks where valid sector data are stored. Then the FTL maintains a pool of writable pages by pre-erasing the redundant blocks. If a sector write is requested, the FTL writes the sector data to an available page in the pool and updates a map that translates a logical sector number to the physical location on the flash memory chip.

Essentially, there are two approaches to FTL design: page mapping and block mapping. Specifically, page mapping FTL has been used in some low capacity flash memory cards while block mapping FTL or a page/block hybrid mapping approach has been used in most high capacity NAND flash memory storage devices such as SSDs and USBs. In page mapping FTL, the map translates a sector number to a combination of page number and block number where the sector data exists. Therefore, each modified sector can be written to any available page of any redundant block. After writing the modified sector, the map is updated to reflect the new location of the sector and the old location of the sector will be reclaimed by a garbage collection mechanism that is similar to the *cleaning* mechanism used for LFS (Log-structured File System) [15].

In block mapping FTL, the map is used to translate a logical block number to a physical block number. To describe the operations of block mapping FTL, let us assume that N_s sectors can be stored in a block. To read sector data, the logical block number where the sector resides is calculated by dividing the sector number by N_s. Then, the map is used to translate the logical block number to a physical block number. If sectors are stored in ordered manner in the block, then the target sector can be found easily in the block; otherwise a second-level map is required to locate the sector within the block.

To modify data, the block mapping FTL reserves some redundant blocks called *log blocks*. To modify sector data in data block *b*, the FTL allocates a log block for the data block *b*. Then modified sectors belonging to data block *b* are written to the log block. At some given time, the log block and original data block *b* need to be merged. The merge operation consists of 1) obtaining a new empty block (usually via an erasure of a block), 2) copying valid sectors from the old data block *b* and/or the log block, and 3) updating the map to designate that sectors are now in the new block and that the old data block *b* and the log block are now invalid empty blocks.

Though page mapping FTL seems to match well with the natural properties of flash memory, it suffers from high mapping overhead and inconsistent performance caused by the variance of garbage collection overhead. Therefore, many SSDs have used block mapping FTLs or hybrid schemes. Lee et al. proposed a hybrid scheme that uses sector mapping (similar to page mapping) for log blocks while using block mapping for data blocks [16]. Yoon et al. proposed another hybrid scheme that applies block mapping to some log blocks and page mapping to other log blocks [17].

Further optimizations for block mapping FTLs have been proposed. The Page Padding method proposed by Kim et al. pre-copies the missing sector *z* when non-adjacent sectors *x* and *z* are written to a log block [18]. Through pre-copying, more of the merges becomes switch merges, which is much more efficient than the general merge described above. (For more information regarding the advantage of switch merge compared to the general merge, see [14].) Also, Kim and Ahn proposed a block level LRU policy called BPLRU as a block replacement policy in SSDs [19].

Though block mapping FTL has been used in most SSDs to date, some SSD designers have considered page mapping FTL for their SSDs. The greatest virtue of page mapping FTL is excellent random write performance because it writes randomly requested sectors to any of the available pages of the pre-erased blocks. Later, however, garbage collection is required to reclaim dead sectors, with random writes more likely to increase garbage collection overhead than sequential writes. Hence, in the end, the inherent property of block erasure may have a considerable effect on the long-term performance of page mapping FTL. However, aggressive background garbage collection may possibly hide the disadvantages of page mapping FTL. This is why SSD designers are now considering page mapping FTL in high-end SSDs that now come equipped with powerful processing power and abundant RAM space, which are essential for efficient background garbage collection [20]. Indeed, SSDs employing page mapping FTL are available for purchase, but they come with a relatively hefty price tag. Hence, for the immediate future, consumers of these costly storage devices are expected to be those who are less cost-sensitive such as early adaptors and makers of high-end enterprise database systems. Our study is limited to SSDs that use block or hybrid mapping FTLs, which would be more popular with the more cost conscientious customers.

2.2 Solid State Drive Architecture

SSDs have the same form factor as the conventional magnetic disks except that it uses flash memory to store data. Usually, a typical SSD comprises a number of flash memory chips, SRAM buffer, SDRAM buffer, and microprocessor that are all connected by an AMBA bus as depicted in Figure 1. The microprocessor executes an optimized FTL that controls and schedules buffering, data transmission, and all flash memory operations within the SSD. An important role of the FTL is to utilize all resources concurrently, and concurrent utilization of buses and chips is achieved with bus-level and chip-level interleaving. Through these interleaving operations, performance of an SSD exceeds that of a single flash memory chip.

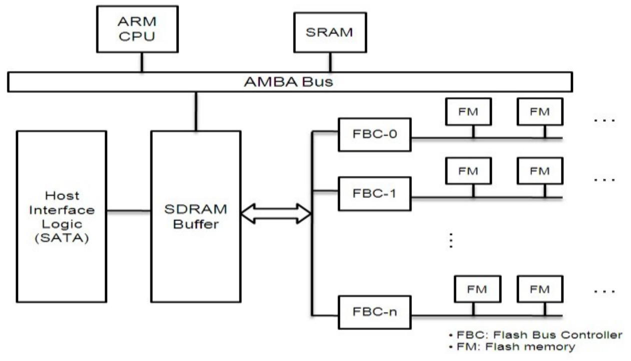


Figure 1. MTRON SSD Block Diagram [21]

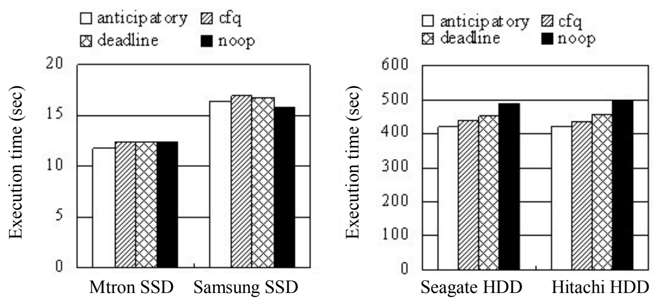
In the SSD, multiple flash memory chips form a single logical flash memory chip. Also, all physical blocks with the same block number of those chips form a single logical block and all physical pages with the same page number in those blocks form a single logical page. As data can be read/written from/to the physical pages of a logical page in parallel, this logical union of physical chips enhances the performance of an SSD without modification of the basic operations of the FTLs that may drive a single flash memory chip, though real implementations must consider many details for interleaving operations.

2.3 Linux Disk I/O Scheduler

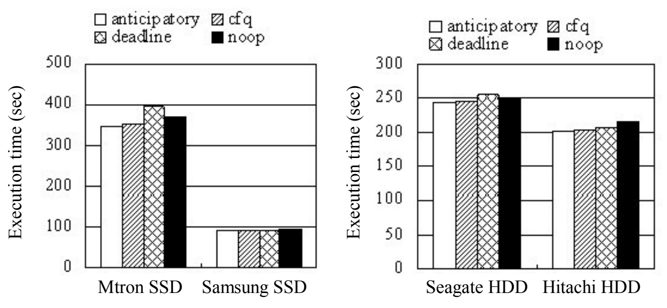
Though numerous disk scheduling algorithms such as SCAN (and its variations) and Shortest Seek Time First (SSTF) can be found in the literature, in this paper, we will focus on disk schedulers found in the Linux operating system. The most recent Linux kernel 2.6.x provides four disk schedulers that one can employ. In this section, we briefly discuss these disk schedulers. One of them, namely, *Noop*, is based on the FCFS (First Come First Served) algorithm and the others, namely, *Deadline*, *Anticipatory*, *CFQ* (Complete Fair Queueing), are based on the elevator algorithm.

- *Noop* serves read/write requests based on their arrival time. It also merges adjacent requests to a larger one, when possible.
- *Deadline* serves requests in ascending (or descending) order of their sector number. Also, read requests are scheduled to be serviced within its deadline. However, if the waiting time of a request exceeds its deadline, the request is served immediately.
- *Anticipatory* is similar to the *Deadline* scheduler except that the Anticipatory scheduler waits for new in-coming read requests for a predetermined period. If the new request arrives within the period and is adjacent to a previous request, then the new request is served before the requests of other processes in the waiting queue. Though the disk may sit idle for the waiting period, it tries to exploit temporal and spatial localities of bursty read activities of processes so as to improve I/O performance [22].
- *CFQ (Complete Fair Queueing)* provides a separate queue for each process and serves requests of queues in round-robin order. In this manner, the CFQ scheduler guarantees completely fair I/O service to all processes.

Though the Noop merges write requests, it does nothing to reduce seek time. On the other hand, the Deadline and the Anticipatory schedulers try to minimize head movement of magnetic disks because they are based on the elevator algorithm. The CFQ algorithm focuses on fairness but also considers characteristics of magnetic disks by sorting requests in the queue. Like this the disk schedulers of Linux can be said to be optimized for magnetic disks.



(a) SSDs (b) HDDs
Figure 2. Read execution time for IOmeter benchmark



(a) SSDs (b) HDDs
Figure 3. Write execution time for IOmeter benchmark

3. Behavior of Existing Schedulers on SSD

To understand the behavior of existing Linux disk schedulers on SSDs and HDDs, we measured the performance of the two benchmark programs shown in Table 1 on four disks listed in Table 2. The experimental results are quite interesting and gave us insights for the design of our new disk schedulers. We discuss the results in this section.

Due to properties of flash memory, we expected fast and constant read time for both sequential and random read requests on SSDs and, indeed, the random and sequential read requests finished in almost the same time on both SSDs. (Refer to Figure 6 for quantitative results.) On the contrary, sequential reads were almost 16 times faster than random reads on magnetic disks. (Refer to Figure 7 for quantitative results.) The disk schedulers in Linux did not make a notable difference under read-oriented workloads on SSDs while they did have some influence on magnetic disks as can be seen in Figure 2. From Figure 2(b), we see that Anticipatory scheduler, which aggressively exploits locality, shows the best performance among the four disk schedulers for the magnetic disks. While read performance of SSDs using different scheduling algorithms differs by 0.8 and 1.1 seconds for the Mtron and Samsung SSDs, for write performance, the difference is 48 and 4.8 seconds, respectively, in Figure 3. For write performance of HDDs, the difference is 12 and 13 seconds for the Seagate and Hitachi HDDs, respectively.

Table 1. Workload

Benchmark	Type of workload
IOmeter [23]	File-server access pattern with read/write requests
FIO [24]	5 concurrently running processes that make random read requests

Table 2. Disks

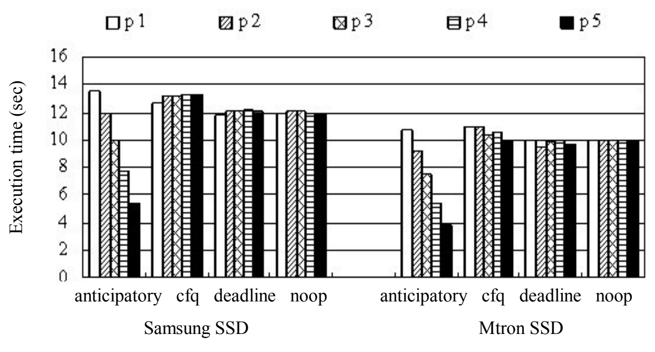
Type	Manufacturer	Model
SSD	Samsung	MCCOE64G5MPP, 2.5" 64GB SATA-2
	Mtron	MSD-SATA3025, 2.5" 32GB SATA-1
HDD	Hitachi	HDS721616PLA380, 7k-rpm 160GB SATA-2
	Seagate	ST380021A, 7k-rpm 80GB PATA

These experimental results reaffirms the fact that optimization of the disk scheduler for the properties that magnetic disks show benefits performance and that particular Linux schedulers seem to service its purpose well. However, it should be noted that, due to near constant read time of SSDs, read performance is not affected by optimization for SSDs but by other scheduling decisions such as preference to read requests.

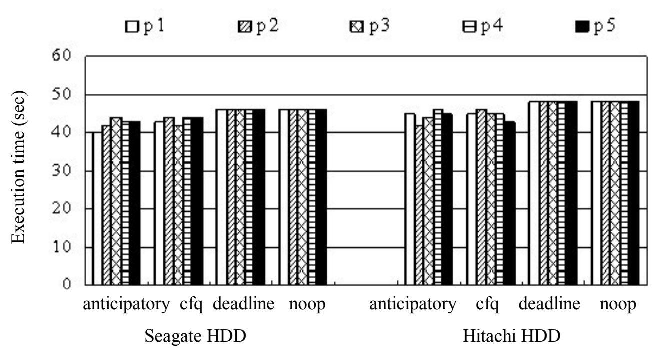
Unlike read performance, write performance of both the SSD and magnetic disk are affected by optimization for underlying storage devices. As the mechanical structure of magnetic disks is well-known, we will not mention why disk schedulers affect performance on magnetic disks. Now, the question is why the disk scheduler influences write performance on SSDs, and we will discuss this later in the next section.

In the experiments on SSDs, we also found an interesting side effect of the Anticipatory scheduler. That is, it has a carefully tuned timing parameter adequate for magnetic disks. Specifically, the Anticipatory scheduler voluntarily waits for some time period for the next read request hoping that it may be adjacent to the previous one. This aggressive anticipation, though, has the potential of unfairly affecting competing processes if the timing parameter is not properly set for the underlying storage device.

This side effect is reflected on experiments that we performed with the FIO benchmark, which concurrently runs five processes that generate I/O requests. The results of these experiments on magnetic disks are depicted in Figure 4(b). Here we see that regardless of the disk scheduler the execution times of the five processes are roughly even. In detail, Anticipatory scheduler has the best performance but shows some variance in execution time. These results are quite understandable because the Anticipatory scheduler is willing to condone unfairness to improve I/O performance. Fortunately, careful tuning of the timing parameter for magnetic disks seems to successfully minimize the sacrifice of fairness as seen in Figure 4(b).

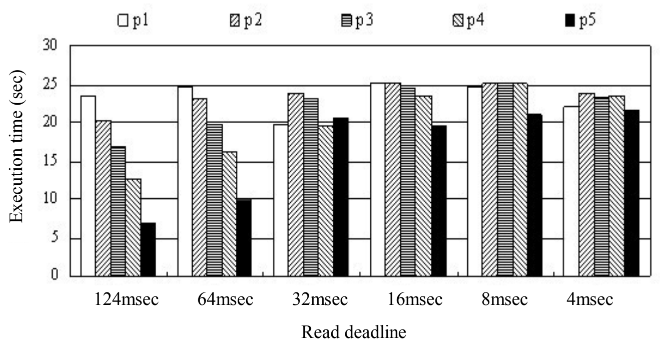


(a) SSDs

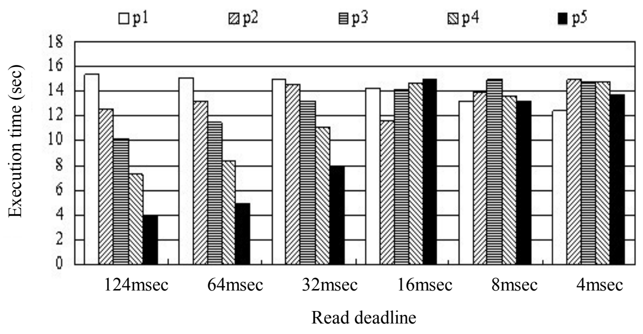


(b) HDDs

Figure 4. Read execution time of five processes of FIO benchmark on SSDs and HDDs



(a) Samsung SSD



(b) Mtron SSD

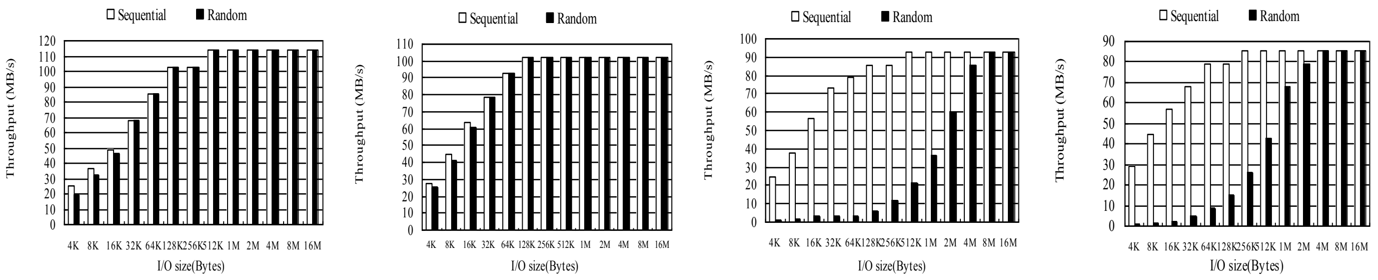
Figure 5. Execution time of anticipatory I/O scheduler with modified read deadline

Behavior of the Anticipatory scheduler on SSDs is quite different from that on magnetic disks. In Figure 4(a), the Anticipatory scheduler has the least sum of execution times of the five processes and thus, shows the best overall performance among the schedulers. However, the completion times of each of the processes vary wildly. For example, the execution time of process 5 is less than half of that of process 1.

I/O performance may sometimes be improved by exclusively serving a particular process with bursty requests. However, unbearable unfairness caused by partiality should be avoided. To find the balance between performance and fairness, the timing settings of the Anticipatory scheduler should be tuned for SSDs. To tune the parameter, we need to understand the timing behavior of the Anticipatory scheduler. When the Anticipatory scheduler receives a request from a process, it anticipates a request from the same process. The predefined period that it waits for the next request is called the anticipation time. If the same process generates a read request within the anticipation time and the request is adjacent to the previous one, then the scheduler serves it immediately. However, for this to happen all pending requests must be within their deadline. If any pending request has past its deadline or the anticipation time has passed, then the Anticipatory scheduler serves the pending requests in the I/O queue.

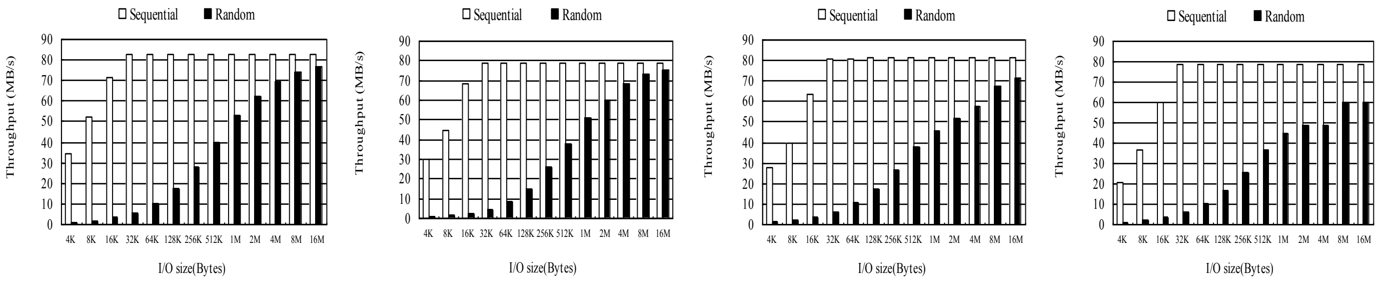
In its original setting, the read request deadline is set to 124 milliseconds, and this seems to work well for most magnetic disks. For example, in Figure 4(b), the Anticipatory scheduler performs the best without notable unfairness though its variation of execution times is slightly larger than other schedulers. This deadline value, however, is too long for SSDs. As read service time of an SSD is consistent and far smaller than that of a magnetic disk, the read deadline seldom expires and consequently, the process being serviced holds the exclusive I/O right given by the Anticipatory scheduler for an inadequately prolonged duration.

To see how the read deadline affects SSDs, we modified the read deadline to various values ranging from 124 milliseconds to 4 milliseconds, which is the minimum value allowed in our system. The results are shown in Figure 5. As can be seen from the results, as the read deadline value decreases, the variation in the execution time tends to lessen. However, there does seem to be some sacrifice in performance as the performance of the fastest completing process tends to converge towards the slower completing process. From these results, we can conjecture that it is harder to achieve performance and fairness at the same time in SSDs than in HDDs. It is not yet clear why this is happening, and we are in the processing of investigating this phenomenon.



(a) Samsung SSD read (b) Mtron SSD read (c) Samsung SSD write (d) Mtron SSD write

Figure 6. Sequential and random read/write throughputs of SSDs for various request sizes



(a) Seagate HDD read (b) Hitachi HDD read (c) Seagate HDD write (d) Hitachi HDD write

Figure 7. Sequential and random read/write throughputs of HDDs for various request sizes

4. SSD Performance Characteristics and New I/O Schedulers for Linux

4.1 Logical block

In the previous section, we note that the read performance of an SSD is almost consistent and independent of the ordering and geometrical distance between read requests, while its write performance shares a common behavior with magnetic disks where sequential writes are much faster than random writes. We also showed that the behavior of SSDs is based on the inherent nature of flash memory much as the behavior of magnetic disks is based on the structure of the moving head and the rotating platter. The question now is how to devise a disk scheduler so that it may exploit the characteristics of SSDs. Is reordering of requests necessary? Can merging read/write requests increase performance? What is the best request size for read/write?

The nature of flash memory implies that read service time is almost constant, and our experimental results confirmed this. A recently study by Feng et al. show that a long sequential read request provides better performance than sequence of short distinct read requests in some SSDs [25]. This may be true for purely long sequential read and short distinct reads, but in real workloads where this is rarely the case, the difference is minimal. Also, gathering and merging read requests to make a long sequential request is not desirable for disk schedulers because it delays service of read requests and eventually increases process execution time. Rather, we believe that minimizing service time of each read request is more beneficial than delaying and merging read requests even in SSDs. Therefore, it seems that we do not need to pay much attention to scheduling read requests. Also, location independence gives us the freedom to focus on scheduling write requests and also for interleaving read requests between write requests. Aggressive policies for tight deadline or fairness may be feasible on an SSD.

We now concentrate on the write behavior of SSDs. As mentioned before, an SSD using block mapping FTL regards multiple blocks as a logical block. This implies that sequential writes to a logical block is much more efficient than random writes going to various logical blocks because the sequential writes would involve only one logical block in a merge operation while the random writes would generally involve numerous logical blocks for a merge operation. From this observation, we can devise a way to maximize write performance in an SSD. Let us assume that we can arrange write requests into bundles the size of a logical block so that write requests falling in a logical block belong to the same bundle. Also assume that we can sort all requests within a bundle and then write them sequentially at once. Then, interestingly, the writing order of the bundles themselves will not be important if all requests within the bundle are written sequentially at a time because the structure of flash memory does not impose any penalty for jumping from one logical block to another logical block for writing. Hence, the bundle, that is, the logical block size becomes the key issue. Henceforth, we will refer to the bundle, aligned along the logical block boundary as the *LBA-bundle (Logical Block Aligned-bundle)* and the process of making an LBA-bundle as *LBA-bundling*.

To prove this theory and to identify the logical block size of an SSD, we conducted some experiments on SSDs. If the theory is true, then write throughput would be the maximum when writing LBA-bundles and also, bundles larger than the logical block size should not improve the performance any further. Moreover, write ordering of bundles should no longer matter when the bundle size exactly matches the logical block size and, therefore, random write throughput of LBA-bundles should reach the maximum throughput of an SSD. (Like magnetic disks, the maximum write throughput is obtained through huge sequential writes in most SSDs.) Inversely, we can identify the logical block size by comparing the throughput of random writes of a specific request size with the maximum write throughput of an SSD and, if they match, then that particular write request size is the logical block size of the SSD.

In our experiments, we compared the sequential write throughput with the random write throughput of specific bundle sizes. If they did not match, then we increased the bundle size and repeated the experiments. The results of these experiments are reported in Figure 6, and we see here that the random write throughput matches exactly to that of the sequential one when the bundle size is 4MB on the MTRON SSD and 8MB on the Samsung SSD, and that the performance does not improve any further with larger bundle sizes. This set of experimental results validates our theory deduced from the characteristics of flash memory and the architecture of SSDs. In particular, the exact match of maximum and random write throughputs shows that the geometrical distance and write ordering of LBA-bundles do not affect performance on an SSD. This is unique to SSDs because on magnetic disks, random write throughput of cylinder group size bundles cannot match the maximum (totally sequential) write throughput as shown in Figure 7.

From the analysis and experiments, we can conclude that LBA-bundling of write requests provides two advantages: one is performance and the other is free ordering. If write requests are grouped in LBA-bundles, then write throughput can be maximized by writing a bundle at a time. Also, the writing order of LBA-bundles does not affect performance.

4.2 Design of New Disk Schedulers

In this section, we describe new I/O schedulers for SSDs. Specifically, the new schedulers utilize the order freedom of read requests and LBA-bundle writes to implement specific policies such as maximizing fairness or minimizing response time.

LBA-bundling, though, may raise a *request-reordering* problem. Let us assume that a disk scheduler focuses on minimizing read and write response times. For that purpose, if read and write service times of an SSD is constant, the easiest way to guarantee the minimum read/write response time is to apply the FIFO order to the read and write requests. Indeed, as read service time of an SSD is almost constant, the FIFO ordering for read requests can be a perfect approach. However, minimizing response time of write requests is much more complicated in real SSDs because the write service time varies significantly according to their ordering and bundling. As previously mentioned, appropriate bundling can maximize write throughput. However, grouping may also increase response time. For example, assume that four sector writes, *w0*, *w1*, *w2*, and *w3*, were requested in that order and *w0*, *w2*, and *w3* belong to LBA-bundle 0 and *w1* belongs to LBA-bundle 1. If a disk scheduler writes LBA-bundle 0 before LBA-bundle 1, then writes *w2* and *w3* in LBA-bundle 0 will respond earlier than *w1* in LBA-bundle 1, while in the reverse case, *w1* will respond earlier than *w0*. In this way, LBA-bundling has the potential to conflict with particular scheduling criteria such as minimizing the response time of write requests.

In most cases, this conflict cannot be resolved without sacrificing some other critera such as performance. For example, to guarantee true FIFO ordering for write requests, LBA-bundling should not be used. In this kind of situation, one has no choice but to decide based on one's needs. Acknowledging that LBA-bundling cannot guarantee true FIFO ordering of write requests, by LBA-bundling, we choose to deliver improved throughput performance, an important criteria in general purpose operating systems. Moreover, improving overall write throughput through LBA-bundling may also result in reduced write response time.

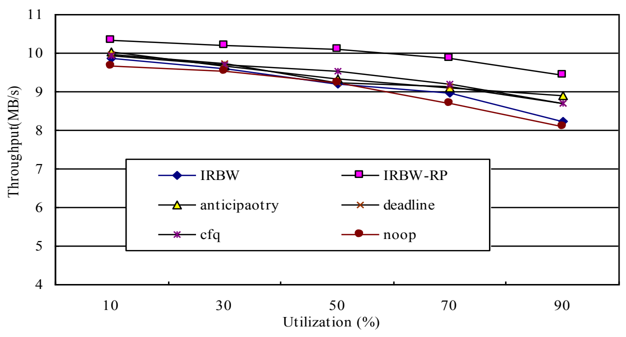
Focusing on throughput performance, we propose two scheduling policies: the *IRBW-FIFO (Individual Read Bundled Write FIFO) Scheduler* and the *IRBW-FIFO-RP (Individual Read Bundled Write FIFO with Read Preference) Scheduler*. Both scheduling policies bundle write requests into LBA-bundles. IRBW-FIFO then retains the ordering of individual read requests and the LBA-bundles. IRBW-FIFO does not provide favor to either read or write requests to guarantee request ordering, except for those caused by the LBA-bundling. The IRBW-FIFO-RP scheduler, on the order hand, maintains separate orderings of read requests and the LBA-bundles but gives favor to the read requests. Favoring reads, however, may result in write starvation. Hence, to avoid write starvation, in our implementation, we allow each LBA-bundle to yield to a read request only once. The design choice of the IRBW-FIFO-RP scheduler is based on the fact that synchronous reads must have higher priority than asynchronous writes. For the write order of the LBA-bundles, we choose the time of the oldest request to represent the request time of each LBA-bundle so as to minimize the worst case service time of the write requests.

5. Experimental Results

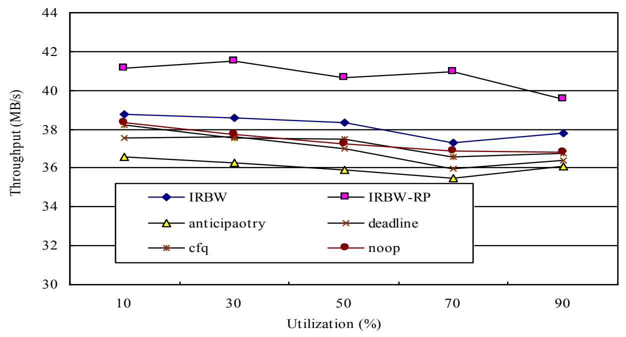
In this section, we present the results of the experiments performed on the environment shown in Table 3. For evaluation, we implemented the two proposed I/O schedulers in Linux 2.6.23. From the implementation viewpoint, IRBW-FIFO is a scheme similar to the Noop scheduler since it does not discriminate between the read and write requests. Also, IRBW-FIFO-RP is similar to the Deadline scheduler as it gives priority to read requests.

Table 3. Experimental environment

Type	Specifics	
CPU/RAM	Intel Core2 Duo E4500 2.2GHz / 2GB DRAM	
SSD	Samsung 64GB MCCOE64G5MPP SATA-2	
	Mtron 16GB MSD-SATA6025 SATA-1	
OS	Linux-kernel 2.6.23 / Ext3 File system	
Benchmark	Postmark	Type-A: 1KB~32KB file size
		Type-B: 1MB~16MB file size
	IOmeter	File server access pattern
Targets	IRBW-FIFO, IRBW-FIFO-RP, and existing Linux I/O schedulers	

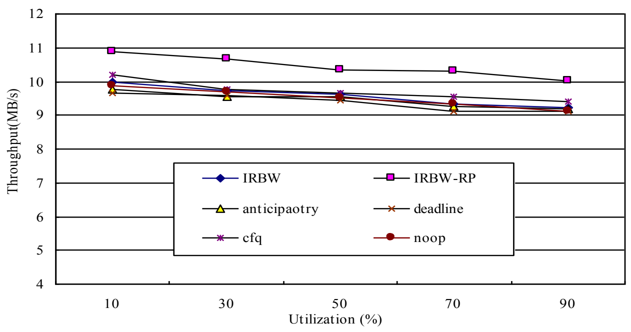


(a) Postmark Type-A

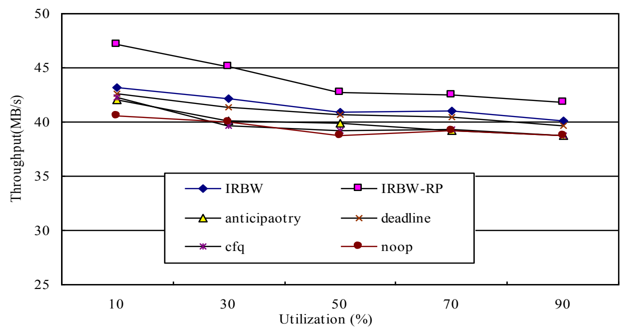


(b) Postmark Type-B

Figure 8. Throughput of Postmark benchmarks on Samsung SSD

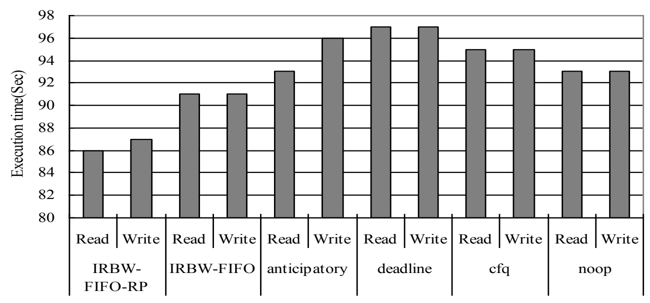


(a) Postmark Type-A

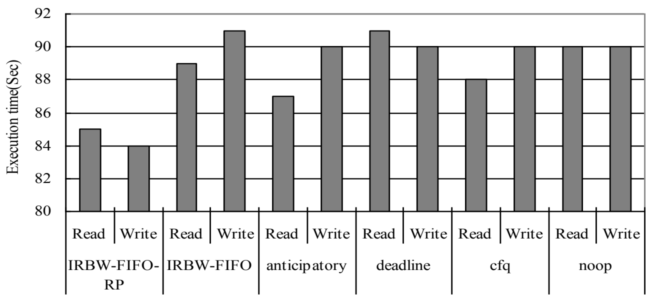


(b) Postmark Type-B

Figure 9. Throughput of Postmark benchmarks on Mtron SSD



(a) Samsung SSD



(b) Mtron SSD

Figure 10. Execution time of IOmeter benchmark

5.1 Throughput results

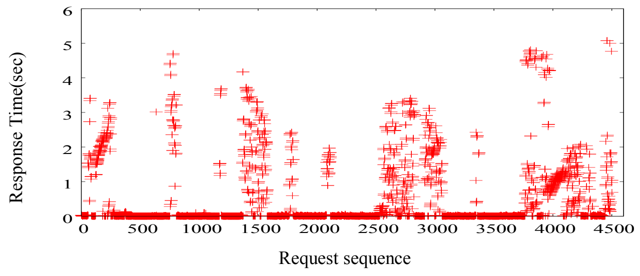
We used the Postmark benchmark version 1.5, which mimics Internet software such as electronic mail, net-news, and web-based commerce. We first randomly fill up the disk with files until the desired utilization is met, then perform 5,000 and 10,000 transactions, respectively, for the Type-A and Type-B benchmarks. A transaction in the Postmark benchmark consists of combinations of file read, write, create, and delete operations.

The Postmark benchmark results are presented in Figure 8 and 9 where the *y*-axis is the throughput in MB/sec, while the *x*-axis refers to the utilization of the SSD. In Figure 8(a), Postmark Type-A, IRBW-FIFO (which we denote IRBW) outperforms the Noop scheduler, which is a similar scheme, by 1~3% while performing worse than the other Linux schedulers by 2~7%. The reason behind the degradation is that synchronous reads are delayed since synchronous requests are given the same priority as asynchronous requests. On the other hand, IRBW-FIFO-RP (which we denote IRBW-RP) outperforms the Deadline scheduler and the other existing Linux schedulers by 15% and 2-4%, respectively. In Figure 8(b), we see that for the Postmark Type-B, IRBW-FIFO and IRBW-FIFO-RP improves performance by 1-3% and 10-17%, respectively, over existing Linux schedulers. Results for the Mtron SSD presented in Figure 9 show almost the same results as those of the Samsung SSD. In Figure 9, IRBW-FIFO-RP improves performance by up to 16% and 17% for the Type-A and Type-B benchmarks, respectively. The results confirm that giving favor to read requests, like what IRBW-FIFO-RP is doing, is better for throughput and that LBA-bundling plays an important role in improving performance.

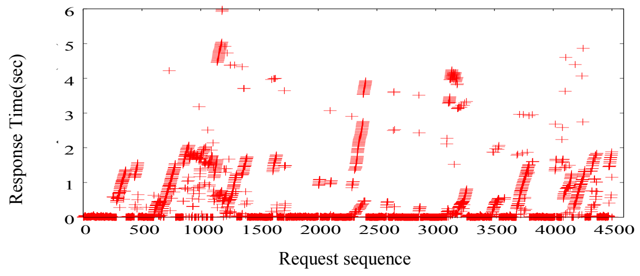
We next ran the IOmeter benchmark whose workload shows a file server access pattern. A file server is a computer that is attached to a network whose primary purpose is to provide locations for storing computer files such as documents, sound files, photographs, movies, databases, etc. File servers generally do not perform heavy calculations. The IOmeter benchmark result is presented in Figure 10, where the *y*-axis is the execution time in seconds, while the *x*-axis is each scheduler with separate read/write request results. Similarly to the Postmark benchmark results, IRBW-FIFO-RP shows the best performance compared to the other schedulers. IRBW-FIFO-RP reduces the execution time of reads and writes by up to 8% and 7% compared to the best cases of existing Linux schedulers (Anticipatory for read and Noop for write), respectively, for Samsung SSD (Figure 10(a)). Similarly, IRBW-FIFO-RP reduces read and write performance by 2% and 7% compared to the Anticipatory scheduler for the Mtron SSD as shown in Figure 10(b).

5.2 Response time results

Response time is another important criterion for the scheduler. We compare the response times of the IRBW-FIFO-RP and Deadline schedulers as they are similar schemes. Figure 11 shows the response time for the Deadline and IRBW-FIFO-RP schedulers running the Postmark benchmark Type-A on the Samsung SSD. The value on the *y*-axis indicates the response time in seconds and the *x*-axis indicates the request sequence of read/write requests. Figure 11(a) and Figure 11(b), which show the response time distribution for the Deadline scheduler and the IRBW-FIFO-RP scheduler, are similar. The average response times of the two schedulers are 0.475 and 0.458 seconds, respectively. The IRBW-FIFO-RP scheduler is faster than the Deadline scheduler by 4%, and the results show LBA-bundling does not incur extra delays.



(a) Deadline I/O scheduler



(b) IRBW-FIFO-RP I/O scheduler

Figure 11. Response time of Postmark benchmark on Samsung SSD

6. Conclusion

In this paper, we analyzed the characteristics of the SSD from the view point of the disk scheduler of the Linux operating system. Based on the analysis we proposed two new disk schedulers optimized for SSDs. Specifically, SSDs have much faster read service times than the magnetic disks with the service times being almost constant, while write request service times are more complex. To improve write performance, we proposed to aggregate requests into logical block sized bundles, which we refer to as LBA-bundles. Using LBA-bundles as our scheduling unit, we devise two schedulers for SSD: IRBW-FIFO and IRBW-FIFO-RP. Both schedulers arrange write requests into LBA-bundles while reads are independently scheduled. IRBW-FIFO-RP gives favor to read requests while IRBW-FIFO does not. We implement these schedulers in Linux 2.6.23, and results of executing our set of benchmark programs show that performance improvements of up to 17% compared to existing Linux disk schedulers are achieved.

7. Acknowledgements

This work was partly supported by the Korea Science & Engineering Foundation (KOSEF) grant funded by the Korea government(MOST) (No. R0A-2007-000-20071-0) and supported in part by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MEST) (No. R01-2008-000-12028-0). We would also like to thank the anonymous referees for their constructive comments.

8. References

[1] R. D. Eager and A. M. Lister, *Fundamentals of Operating Systems*, New York: Springer-Verlag, 1995.

[2] M. K. McKusick, W. N. Joy, S. J. Leffler and R. S. Fabary, “A Fast File System for UNIX,” ACM Transactions on Computer Systems, vol. 2, no. 3, pp. 181-197, 1984.

[3] E. J. O'Neil, P. E. O'Neil and G. Weikum, "The LRU-K Page Replacement Algorithm For Database Disk Buffering," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pp. 297-306.

[4] T. Johnson, D. Shasha and I. Novell, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," in *Proceedings of the 20th International Conference on Very Large Data Bases*, pp. 439-450, 1994.

[5] N. Megiddo and D. S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pp. 115-130, 2003.

[6] S. Jiang and X. Zhang, "LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *Proceedings of the 2002 ACM SIGMETRICS*, pp. 31-42.

[7] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min and Y. Cho, “LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies,” IEEE Transactions on Computers, vol. 50, no. 12, pp. 1352-1361, 2001.

[8] S.-W. Lee, B. Moon, C. Park, J.-M. Kim and S.-W. Kim, "A Case for Flash Memory SSD in Enterprise Database Applications," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data,* pp. 1075-1086.

[9] A. Leventhal, "Flash Storage Memory," Communications of the ACM, vol. 51, no. 7, pp. 47-51, July, 2008.

[10] "512M x 8Bit / 256M x 16Bit NAND Flash Memory (K9K4Gxxx0M) Data Sheets," Samsung Electronics, Co., 2003.

[11] "1G x 8Bit / 2G x 8Bit NAND Flash memory (K9L8G08U0M) Data Sheets," Samsung Electronics, Co., 2005.

[12] "Understanding the Flash Translation Layer (FTL) Specification," Intel Corporation, 1998.

[13] *Flash-Memory translation layer for NAND flash (NFTL)*, M-Systems.

[14] J. Kim, J. M. Kim, S. H. Noh, S. L. Min and Y. Cho, "A Space-efficient Flash Translation Layer for CompactFlash Systems," IEEE Transactions on Consumer Electronics, vol. 28, no. 2, pp. 366-375, 2002.

[15] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," ACM Transactions on Computer System*s,* vol. 10, no. 1, pp. 26-52, 1992.

[16] S. W. Lee, D. J. Park, T. S. Chung, D. H. Lee, S. Park and H.-J Song, "A Log Buffer based Flash Translation Layer using Fully Associative Sector Translation," ACM Transactions on Embedded Computing Systems*,* vol. 6, no. 1, Feb., 2007.

[17] J. H. Yoon, E. H. Nam, Y. J. Seong, H. Kim, B. S. Kim, S. L. Min and Y. Cho, "Chameleon: A High Performance Flash/FRAM Hybrid Solid State Disk Architecture," IEEE Computer Architecture Letters*,* vol. 7, no. 1, pp. 17-20, 2008.

[18] H. Kim, J. H. Kim, S. Choi, H. Jung and J. Jung, "A Page Padding Method for Fragmented Flash Storage," Lecture Notes in Computer Science*,* vol. 4705, pp. 164-177, 2007.

[19] H. Kim and S. Ahn, "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," in *Proceeding of the 6th USENIX Conference on File and Storage Technologies*, pp. 239-252, 2008.

[20] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse and R. Panigrahy, "Design Tradeoffs for SSD Performance," in *Proceedings of the USENIX 2008 Annual Technical Conference*, pp. 57-70.

[21] "Solid State Drive MSP-SATA 7035 3.5-inch Product Specification", MTRON, Co., 2007

[22] S. Iyer and P. Druschel, "Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O," in *Proceedings of the Eighteenth ACM SOSP*, pp. 117-130, 2001.

[23] IOmeter project. http://www.iometer.org/.

[24] J. Axboe, FIO - Flexible I/O Tester. http://freshmeat.net/projects/fio/.

[25] F. Chen, D. A. Koufaty and X. Zhang, "Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives," in *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems, ACM SIGMETRICS '09*, pp. 181-192, 2009.