Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives

Feng Chen[1], David A. Koufaty[2], and Xiaodong Zhang[1]

[1]Dept. of Computer Science & Engineering
The Ohio State University
Columbus, OH 43210
{fchen, zhang}@cse.ohio-state.edu

[2]System Technology Lab
Intel Corporation
Hillsboro, OR 97124
david.a.koufaty@intel.com

ABSTRACT

Flash Memory based Solid State Drive (SSD) has been called a "pivotal technology" that could revolutionize data storage systems. Since SSD shares a common interface with the traditional hard disk drive (HDD), both physically and logically, an effective integration of SSD into the storage hierarchy is very important. However, details of SSD hardware implementations tend to be hidden behind such narrow interfaces. In fact, since sophisticated algorithms are usually, of necessity, adopted in SSD controller firmware, more complex performance dynamics are to be expected in SSD than in HDD systems. Most existing literature or product specifications on SSD just provide high-level descriptions and standard performance data, such as bandwidth and latency.

In order to gain insight into the unique performance characteristics of SSD, we have conducted intensive experiments and measurements on different types of state-of-the-art SSDs, from low-end to high-end products. We have observed several unexpected performance issues and uncertain behavior of SSDs, which have not been reported in the literature. For example, we found that fragmentation could seriously impact performance – by a factor of over 14 times on a recently announced SSD. Moreover, contrary to the common belief that accesses to SSD are uncorrelated with access patterns, we found a strong correlation between performance and the randomness of data accesses, for both reads and writes. In the worst case, average latency could increase by a factor of 89 and bandwidth could drop to only 0.025MB/sec. Our study reveals several unanticipated aspects in the performance dynamics of SSD technology that must be addressed by system designers and data-intensive application users in order to effectively place it in the storage hierarchy.

Categories and Subject Descriptors

B.3.2 [**Design Styles**]: Mass storage; D.4.2 [**Storage Management**]: Secondary Storage

General Terms

Experimentation, Measurement, Performance

Keywords

Flash Memory, Hard Disk Drive, Solid State Drive

SIGMETRICS/Performance'09, June 15–19, 2009, Seattle, WA, USA.

1. INTRODUCTION

For the last two decades, researchers have made continuous efforts to address several open issues of Hard Disk Drive (HDD), such as long latencies of handling random accesses (e.g. [15, 21, 32]), excessively high power consumption (e.g. [10, 18, 20]), and uncertain reliability (e.g. [14, 42]). These issues are essentially rooted to the hard disk's mechanic nature, and thus difficult to be solved physically by disks themselves.

Recently, flash memory based Solid State Drive (SSD) has become an emerging technology and received strong interest in both academia and industry [3, 5, 25, 28, 29, 35]. Unlike traditional rotating media, SSD is based on semiconductor chips, which provides many strong technical merits, such as low power consumption, compact size, shock resistance, and most importantly, extraordinarily high performance for random data accesses. Thus flash memory based SSD has been called "a pivotal technology" to revolutionize computer storage systems [39]. In fact, two leading on-line search engine service providers, *google.com* and *baidu.com*, both announced their plans to migrate existing hard disk based storage system to a platform built on SSDs [13].

An important reason behind such an optimistic prediction and such actions is that SSD shares a common interface with traditional hard disks physically and logically. Most SSDs on the market support the same host interfaces, such as Serial Advanced Technology Attachment (SATA), used in hard disks. SSD manufacturers also carefully follow the same standard [1] to provide an array of Logical Block Addresses (LBA) to the host. On the one hand, such a common interface guarantees backward compatibility, which avoids tremendous overhead for migrating existing HDD based systems to SSD based platforms. On the other hand, such a thin interface can hide complex internals from the upper layers, such as operating systems. In fact, many sophisticated algorithms, such as cleaning and mapping policies, have been proposed, and various hardware optimizations are adopted in different SSD hardware implementations [3, 16]. The complicated internal design and divergent implementations behind the 'narrow' interface would inevitably lead to many performance dynamics and uncertainties. In other words, despite the same interface, SSD is not just another 'faster' disk. Thus we need to conduct a thorough investigation, particularly for understanding its intrinsic limits and unexpected performance behavior.

Most existing research literature and technical papers, however, only provide limited information beyond high level description and standard performance data (e.g. bandwidth and latency), similar to documented specifications for HDDs. Some specification data provided by SSD manufacturers may

even overrate performance or not provide critical data for certain workloads (e.g. performance data for random writes is often not presented in SSD specifications). Some previous work has reported SSD performance data based on simulation [3]. So far, little research work has been presented to report the first-hand data and analysis on the unique performance features of different types of SSDs.

In this paper, we carefully select three representative, state-of-the-art SSDs built on two types of flash memory chips. Each SSD is designed to aim at different applications and markets, from low-end PCs, middle-level performance desktops, to high-end enterprise servers. By using the Intel Open Storage Toolkit [36] to generate various types of I/O traffic, we conducted intensive experiments on these SSDs and analyzed collected performance data. Our purpose is not to compare the performance of these competing SSDs on the market. Instead, we attempt to get insightful understanding on performance issues and unique behavior of SSDs through micro-benchmarks. We hope to inspire the research community, especially OS designers, to carefully consider many existing optimizations, which were originally designed for hard disks, for the emerging SSD based platform. We will answer the following questions and reveal some untold facts about SSDs through experiments and analysis.

1. A commonly reported feature of SSD is its uniform read access latency, which is independent of access patterns of workloads. Can we confirm this or observe some unexpected results with intensive experiments?

2. Random writes have been considered as the Achilles´ heel of SSDs. Meanwhile, high-end SSDs employ various solutions to address this problem. Has the random write problem been solved for the recent generation of SSDs? Is random write still a research issue?

3. Caching plays a critical role for optimizing performance in HDDs to mitigate expensive disk head seeks. Can we make a case or not for a cache in SSDs, which do not have the problem of long seek latency?

4. Writes are much more expensive than reads in flash memory. What are the interactive effects between these two types of operations in SSDs? Would low-latency reads be negatively impacted by high-latency writes? What's the reason behind such interference?

5. Many elaborate algorithms are adopted in SSD firmware to optimize performance. This leads to many internal operations running in the background. Can we observe a performance impact on foreground jobs by such internal operations? How seriously would such background operations affect overall performance?

6. Adopting a log-structure internally, SSD can map logically continuous pages to non-continuous physical pages in flash memory. Since a flash memory block may contain both valid and invalid pages, internal fragmentation could be a problem. How would internal fragmentation affect performance?

7. Having well documented our findings on SSDs, what are the system implications? How can we use them to guide system designers and practitioners for effectively adding SSDs into the storage hierarchy?

The rest of this paper is organized as follows. Section 2 introduces background about flash memory and SSD design. Section 3 presents the experimental setup environment. Then we present our experimental results in Section 4 and 5. In Section 6 we summarize the key observations from our experiments and discuss the implications to users and OS designers. Related work is presented in Section 7 and the final section concludes this paper.

2. BACKGROUND

2.1 Flash Memory

There are two types of flash memories, NOR and NAND [33]. NOR flash memory supports random accesses in bytes and it is mainly used for storing code. NAND flash memory is designed for data storage with denser capacity and only allows access in units of sectors. Most SSDs available on the market are based on NAND flash memories. In this paper, flash memory refers to NAND flash memory specifically.

NAND flash memory can be classified into two categories, Single-Level Cell (SLC) and Multi-Level Cell (MLC) NAND. A SLC flash memory cell stores only one bit, while a MLC flash memory cell can store two bits or even more. Compared to MLC, SLC NAND usually has a 10 times longer lifetime and lower access latency (see Table 1). However, considering cost and capacity, most low-end and middle-level SSDs tend to use high-density MLC NAND to reduce production cost. In this paper, we examined two MLC-based SSDs and one SLC-based SSD.

For both SLC and MLC NAND, a flash memory package is composed of one or more *dies* (chips). Each die is segmented into multiple *planes*. A typical plane contains thousands (e.g. 2048) of *blocks* and one or two registers of the page size as an I/O buffer. A block usually contains 64 to 128 pages. Each page has a 2KB or 4KB data part and a metadata area (e.g. 128 bytes) for storing Error Correcting Code (ECC) and other information. Exact specification data varies across different flash memory packages.

Flash memory supports three major operations, *read*, *write*, and *erase*. Read is performed in units of pages. Each read operation may take 25µs (SLC) to 60µs (MLC). Writes are normally performed in page granularity, but some NAND flash (e.g. Samsung K9LBG08U0M) supports sub-page operations [41]. Pages in one block must be written sequentially, from the least significant to the most significant page addresses. Each write operation takes 250µs (SLC) to 900µs (MLC). A unique requirement of flash memory is that a block must be erased before being programmed (written). An erase operation can take as long as 3.5ms and must be conducted in block granularity. Thus, a block is also called an *erase block*.

Flash memory blocks have limited erase cycles. A typical MLC flash memory has around 10,000 erase cycles, while a SLC flash memory has around 100,000 erase cycles. After wearing out, a flash memory cell can no longer store data. Thus, flash memory chip manufacturers usually ship with extra flash memory blocks to replace bad blocks.

2.2 SSD Internals

Since an individual flash memory package only provides limited bandwidth (around 40MB/sec [3]), flash memory based SSDs are normally built on an array of flash memory packages. As logical pages can be striped over flash memory chips, similar to a typical RAID-0 storage, high bandwidth can be achieved through parallel access. A serial I/O bus

connects the flash memory package to a controller. The controller receives and processes requests from the host through connection interface, such as SATA, and issues commands and transfers data from/to the flash memory array. When reading a page, the data is first read from flash memory into the register of the plane, then shifted via the serial bus to the controller. A write is performed in the reverse direction. Some SSDs are also equipped with an external RAM buffer to cache data or metadata [9, 19].

A critical component, called the *Flash Translation Layer* (FTL), is implemented in the SSD controller to emulate a hard disk and exposes an array of logical blocks to the upper-level components. The FTL plays a key role in SSD and many sophisticated mechanisms are adopted to optimize SSD performance. We summarize its major roles as follows.

Logical block mapping – As writes in flash cannot be performed in place as in disks, each write of a logical page is actually conducted on a different physical page. Thus some form of mapping mechanism must be employed to map a logical block address (LBA) to a physical block address (PBA). The mapping granularity could be as large as a block or as small as a page [19]. Although a page-level mapping [23] is efficient and flexible, it requires a large amount of RAM space (e.g. 512MB in [9]) to store the mapping table. On the contrary, a block-level mapping [2], although space-wise efficient, requires an expensive read-modify-write operation when writing only part of a block. Many FTLs [12,22,26,30] adopt a hybrid approach by using a block-level mapping to manage most blocks as *data blocks* and using a page-level mapping to manage a small set of *log blocks*, which works as a buffer to accept incoming write requests efficiently. A mapping table is maintained in persistent flash memory and rebuilt in volatile RAM buffer at startup time.

Garbage Collection – Since a block must be erased before it is reused, an SSD is usually over-provisioned with a certain amount of clean blocks as an allocation pool. Each write of a page just needs to invalidate the previously occupied physical page by updating the metadata, and the new data can be quickly appended into a clean block allocated from the pool, like a *log*, without incurring a costly erase operation synchronously. When running out of clean blocks, a *garbage collector* scans flash memory blocks and recycles invalidated pages. If a page-level mapping is used, the valid pages in the scanned block are copied out and condensed into a new block; otherwise, the valid pages need to be merged together with the updated pages in the same block. Such a cleaning process is similar to the Log-Structured File System [40] and can be conducted in the background.

Wear Leveling – Due to the locality in most workloads, writes are often performed over a subset of blocks (e.g. file system metadata blocks). Thus some flash memory blocks may be frequently overwritten and tend to wear out earlier than other blocks. FTLs usually employ some wear-leveling mechanism to 'shuffle' cold blocks with hot blocks to even out writes over flash memory blocks.

A previous work [3] has an extensive description about the broad design space of flash memory based SSD. Their work indicates that many design trade-offs can be made in SSD, which lead to very different SSD performance. Our experimental results confirmed that diverse performance is resident on various types of SSDs. Since details of SSD hardware design, especially their FTL algorithms, are regarded as the intellectual property of SSD manufacturers and remain highly confidential, we can only speculate about detailed SSD internals and we may not be able to explain all performance behavior observed on SSD hardware. On the other hand, our intention is not to reverse engineer the internal design of SSDs. Instead, we attempt to reveal performance issues and dynamics observed on SSD hardware and give a reasonable explanation to help explore the implications to system designers and practitioners.

3. MEASUREMENT ENVIRONMENT

3.1 Solid State Drives

Currently, many SSDs are available on the market. Their performance and price is very diverse, depending on many factors, such as flash memory chips (MLC/SLC), RAM buffer size, and complexity of hardware controller. In this work, we selected three representative, state-of-the-art SSDs fabricated by two major SSD manufacturers. Each SSD targets at a different market with various performance guarantees, from low-end, middle-class, to high-end. Among them, two SSDs are based on MLC flash memory, and the high-end one is based on SLC flash memory. Since our intention is not to compare the performance of these competing SSDs, we refer to the three SSDs using *SSD-L*, *SSD-M*, and *SSD-H*, from low-end to high-end. Table 1 shows more details about the SSDs. Listed price is as of October 2008.

	SSD-L	SSD-M	SSD-H
Capacity	32GB	80GB	32GB
Price ($/GB)	$5	$10	$25
Flash memory	MLC	MLC	SLC
Page Size (KB)	4	4	4
Block Size (KB)	512	512	256
Read Latency (µs)	60	50	25
Write Latency (µs)	800	900	250
Erase Latency(µs)	1500	3500	700

Table 1: Specification data of the SSDs.

3.2 Experiment System

Our experiment system is a Dell™ PowerEdge™ 1900 server. It is equipped with two Intel® Core™ 2Quad Xeon X5355 2.66GHz processors and 16GB FB-DIMM memory. Five 146GB 15,000 RPM SCSI hard drives are attached on a SCSI interface card to hold the operating system and home directories. We use RedHat Enterprise Linux Sever 5 with Linux Kernel 2.6.26 and Ext3 file system. The SSDs are connected through the on-board SATA 3.0Gb/s connectors. There is no partition or file system created on the SSDs. All requests are performed to directly access SSDs as raw block devices to avoid interference from the OS kernel, such as the buffer cache and the file system.

In our experiment system, the hard disks use the *CFQ* (Completely Fair Queuing) scheduler, the default I/O scheduler used in the Linux kernel, to optimize the disk performance. For the SSDs, which have sophisticated firmware internally, we use the *noop* (No-Op) scheduler to leave the I/O performance optimization directly handled by the block devices, so that we can expose internal behavior of the SSDs. In our experiments, we also found *noop* usually outperforms the other I/O schedulers on the SSDs.

3.3 Intel® Open Storage Toolkit

We attempt to gain insightful understanding of the behavior of SSDs for handling workloads with clear patterns. Thus, our experiments are mainly conducted through well-controlled micro-benchmarks, whose access patterns are known in advance. Macro-benchmarks, such as TPC-H database

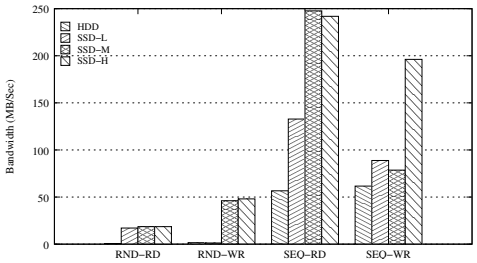


Figure 1: SSD Bandwidths. Four workloads, *Random Read*, *Sequential Read*, *Random Write*, and *Sequential Write*, are denoted as *RND-RD*, *SEQ-RD*, *RND-WR*, and *SEQ-WR* in the figure.

workloads, usually have complicated and variable patterns and thus would not be appropriate for our characteristic analysis of the SSDs. The Intel® Open Storage Toolkit [36] is designed and used for storage research at Intel. It can generate various types of I/O workloads to directly access block devices with different configurations, such as read/write ratio, random/sequential ratio, request size, and think time, etc. It reports bandwidth, IOPS, and latency.

In order to analyze I/O traffic to storage devices in detail, we use *blktrace* [7], which comes with the Linux 2.6 kernel, to trace the IO activities at the block device level. The *blktrace* tool captures various I/O activities, including queuing, dispatching, completion, etc. The most interesting event to us is the *completion* event, which reports the latency for processing each individual request. In our experiments, the trace data is collected in memory during test and then copied to the hard disks that are connected through an RAID interface card to minimize the interference from tracing. The collected data is further processed using *blkparse* and our post-processing scripts and tools off line.

4. GENERAL TESTS

Before we proceeded to the detailed analysis, we first made some general performance measurements on the SSDs. We used the toolkit to generate four distinct workloads, *Random Read*, *Random Write*, *Sequential Read*, and *Sequential Write*. The random workloads used a request size of 4KB to randomly access data in the first 1024MB storage space. The sequential workloads used a request size of 256KB. All the workloads created 32 parallel jobs to maximize the bandwidth usage, and we used direct I/O to bypass the buffer cache and access the raw block devices synchronously. Each test ran for 30 seconds. In order to compare with hard disks, a Western Digital WD1600JS Caviar 7200RPM hard disk was used as a reference HDD.

Figure 1 shows the bandwidths of the four workloads running on the devices. As expected, when handling *random read*, the SSDs exhibit clear performance advantages compared to the HDD. Specifically, all the three SSDs significantly outperform the HDD (0.54MB/sec) by over 31 times higher bandwidths. For *random write*, SSD-M and SSD-H achieve bandwidths of 46MB/sec and 48MB/sec, respectively. SSD-L, the low-end device, has a much lower bandwidth (1.14MB/ sec), which is only comparable to the HDD (1.49MB/sec). When handling *sequential read*, the bandwidth of SSD-L reaches 133MB/sec, and both SSD-M and SSD-H achieve an adorable bandwidth of over 240MB/sec, which is nearly 2/3 of the full bandwidth supported by the SATA bus. In contrast, the HDD has a bandwidth of only 56.5MB/sec. For *sequential write*, SSD-L and SSD-M have bandwidths of 88MB/sec and 78MB/sec, respectively. SSD-H, the enterprise SSD, achieves a much higher bandwidth, 196MB/sec. Considering the long-held belief that SSD has poor write performance, this number is surprisingly high.

As we see in the figure, the performance of the SSDs is better than or comparable to the HDD for all workloads. Compared to SSD-L, the two recently announced higher-end SSDs do show better performance, especially for *random write*.[1] In the next few sections, we will further examine other performance details about the SSDs.

5. EXPERIMENTAL EXAMINATIONS

5.1 Micro-benchmark Workloads

In general, an I/O workload can be characterized by its access patterns (random or sequential), read/write ratio, request size, and concurrency. In order to examine how an SSD handles various types of workloads, we use the toolkit to generate I/O traffic using various combinations of these factors. Three access patterns are used in our experiments.

1. **Sequential** - Sequential data accesses using specified request size, starting from sector 0.

2. **Random** - Random data accesses using specified request size. Blocks are randomly selected from the first 1024MB data blocks of the storage space.

3. **Stride** - Strided data accesses using specified request size. The distance between the end and the beginning of two consecutive accesses is the request size.

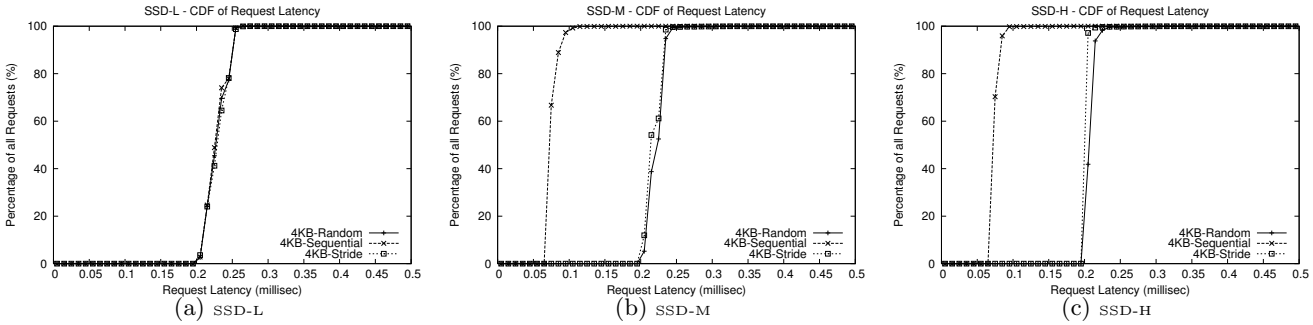
In default, each micro-benchmark runs for 10 seconds to limit trace size while collecting enough data. In order to simplify analysis, we set only one job for all experiments but vary the other three factors, access pattern, read/write ratio, and request size, according to various experimental needs. All workloads directly access the raw block devices. Requests are issued to devices synchronously.

Before our experiments, we first filled the whole storage space using sequential writes with request size of 256KB. After such an initial overwrite, the SSDs in our experiments are regarded as 'full', and the status would remain stable.[2] Note that such a full status remains unchanged thereafter, even if users delete files or format the device, since the liveliness of blocks is unknown at the device level.

Writes into SSDs may change the page mapping dynamically. In order to guarantee that the status of the SSDs remains largely constant across experiments, before each experiment we reinitialize the SSD status by sequentially overwriting the SSD with request size of 256KB. In this way, we make sure that the logical page mapping would be reorganized into an expected continuous manner before each run.

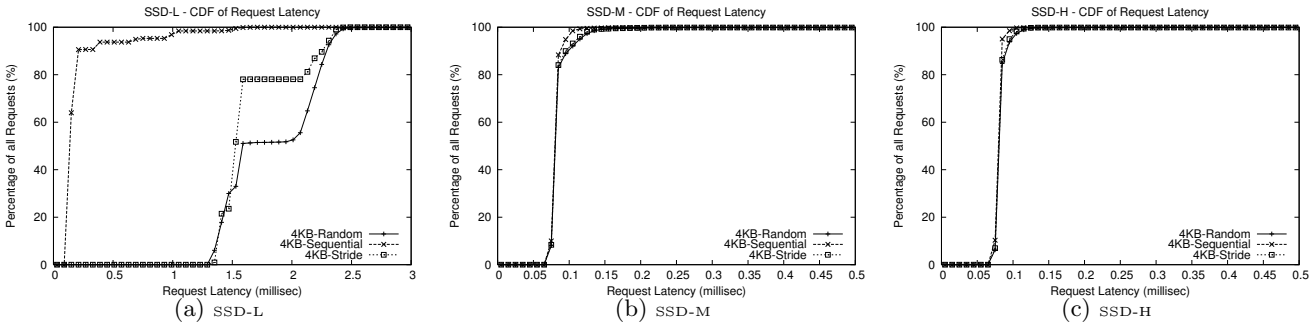
5.2 Do reads on SSD have a uniform latency?

Due to its mechanical nature, the hard disk has a *non-uniform* access latency, and its performance is strongly correlated with workload access patterns. Sequential accesses on hard disks are much more efficient than random accesses. An SSD, without expensive disk head seeks, is normally believed to have a *uniform* distribution of access latencies, which is independent of access patterns. In our experiments, we found this is not always true, for both reads and writes. In this section, we will first examine read operations.



(a) SSD-L (b) SSD-M (c) SSD-H

Figure 2: CDF of request latencies for Read operations on the SSDs.



(a) SSD-L (b) SSD-M (c) SSD-H

Figure 3: CDF of request latencies for Write operations on the SSDs.

In order to examine the distribution of read latencies on SSDs, we run *sequential*, *random*, and *stride* workloads using 4KB read requests on the three SSDs. Figure 2 shows the Cumulative Distribution Function (CDF) of request latencies for the three workloads on the SSDs.

SSD-L does not distinguish random and sequential reads, which is an expected *uniform* distribution. Nearly all reads of the three workloads have latencies of 200-250µs on SSD-L. However, SSD-M and SSD-H, the two higher-end SSDs, do show a *non-uniform* distribution of latencies. Specifically, *sequential* has latencies of only 75-90µs, which is nearly 65% less than *stride* and *random* (200-230µs).

Several reasons may contribute to the unexpected non-uniform distribution of latencies in SSD-M and SSD-H. First, a sequential read in flash memory is three orders of magnitude faster than a random read. For example, in the Samsung K9LBG08U0M flash memory, a random read needs 60µs, while a sequential read needs only 25ns [41]. Second, similar to hard disks, a readahead mechanism can be adopted in an SSD controller to prefetch data for sequential reads with low cost. Most flash memory packages support two-plane operations to read multiple pages from two planes in parallel. Also, operations across dies can be interleaved further [41]. Since logical pages are normally striped over the flash memory array, reading multiple logically continuous pages in parallel for readahead can be performed efficiently.

Concerning our measurement results, we believe readahead is the main reason for the observed non-uniform distribution of latencies. On SSD-M and SSD-H, each sequential read accounts for only 75-90µs, which is even smaller than the time (100µs) to shift a 4KB page out over the serial bus, not to mention the time to transfer data across the SATA interface. Thus, the data is very likely to be fed directly from an internal buffer rather than from flash memory. If data accesses become non-sequential (*stride* or *random*), readahead would be ineffective and the latency increases to around 200-230µs. We also found that the first four requests in *sequential* have latencies of around 220µs, which appears to be an initial period for detecting a sequential pattern. In contrast, as a low-end product, SSD-L does not have a large buffer for readahead, thus each read has to load data directly from flash memory with a similarly long latency for different access patterns, which leads to a uniform distribution of latencies. Another interesting finding is that when handling random reads, the three SSDs have comparable latencies, though SSD-H uses expensive SLC flash.

5.3 Would random writes be the worst case?

A random write on flash memory is normally believed less efficient than a sequential write. Considering an erase block of N pages, a random write may incur up to $N-1$ page reads from the old block, N page writes to the new block, and one block erase. In contrast, a sequential write only incurs one page write and 1/N erase operations on the average. In practice, a log-structured approach [3,5,19] is adopted to avoid the high-cost *copy-erase-write* for random writes. Akin to the Log-Structured File System [40], each write only appends data to a clean block, and garbage collection is conducted in the background. Using such log-structured FTLs, the write is expected to be largely insensitive to access patterns. In this section, we will examine the relationship between writes on SSDs and workload access patterns.

Similar to the previous section, we run three workloads, *sequential*, *random*, and *stride* on the SSDs, except that all requests are write-only. Figure 3 plots the CDF of request latencies of the three workloads.

SSD-L shows a *non-uniform* distribution of request latencies for workloads with different access patterns. Sequential accesses are the most efficient. As shown in Figure 3(a), 88% of the writes in *sequential* have latencies of only 140-160µs. In contrast, *stride* and *random* are much worse. Over 90% of the requests in both workloads have latencies higher than 1.4ms. This seems to confirm the common perception that writes on SSD are highly correlated with access patterns.

In order to investigate why such a huge difference exists

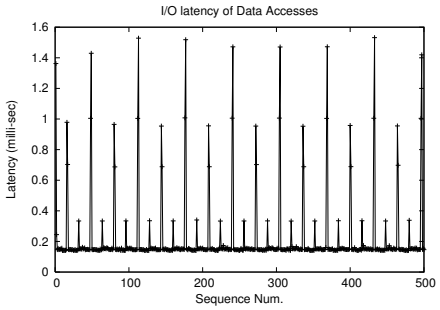
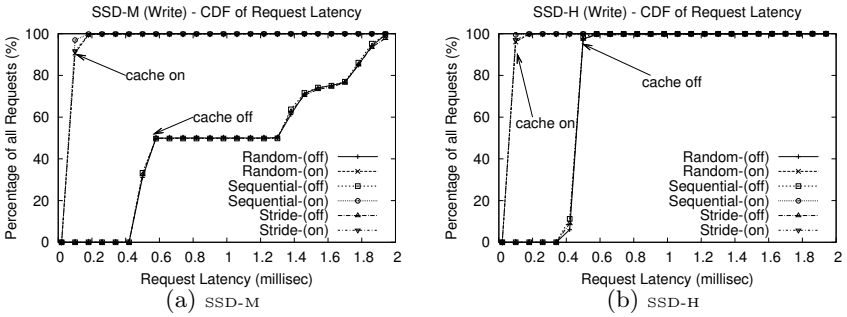


Figure 4: The first 500 requests of *sequential write* on SSD-L.



(a) SSD-M (b) SSD-H

Figure 5: The effect of a disk cache to writes on SSDs. Disk cache state is denoted as *off* and *on*.

between sequential and non-sequential writes on SSD-L, we plot the first 500 write requests of *sequential* in Figure 4 and we see an interesting pattern. As shown in the figure, most requests have a latency of around 140-160µs. A spike of around 1.4-1.5ms appears every 64 requests, and three spikes of around 1ms, 700µs, and 300µs periodically appear in between. The average latency is 209µs.

Our explanation is as follows. From the figure, SSD-L seems to use a small buffer (e.g. 32 pages) to hold data of incoming write requests, because most writes have a latency much lower than the programming time (800µs). If write requests are coming sequentially, data can be temporarily held in the buffer. Since the logically continuous pages can be striped over multiple flash memory chips, the buffered pages can be written into flash memory in parallel later. If incoming requests are non-sequential, such optimization cannot be done, thus data has to be directly written into flash memory, which leads to a much higher latency for each individual write. However, since SSD-L does not employ a sufficiently large buffer (only 16KB) to hold 32 pages, we suspect that SSD-L actually takes advantage of the 4KB registers of the 32 planes, which are organized in 16 flash memory packages, as a temporary buffer. As the data has to be transferred to the register through the serial bus, each write would cause a latency of around 140-160µs, which is slightly lower than a read (200-230µs) that needs extra time to read data from flash memory. When the registers are full, a page program confirm command is issued to initiate the programming process and data can be written into flash memory in parallel, which results in the observed high spikes.

Writes on SSD-M and SSD-H, in contrast, are nearly *independent* of workload access patterns. On both SSDs, over 90% of the requests of all the three workloads have latencies of 75-90µs. Since the observed latency is much lower than the latency for writing a page to flash memory, writes should fall into an on-drive cache instead of being actually written to flash memory medium. Actually, the observed write latency is nearly equal to the latency of sequential reads, which transfer data across SATA bus in the reverse direction. This also confirms our previous hypothesis about readahead on SSDs. As write requests can return to the host as soon as data reaches on-drive cache, distinct access patterns would not make any difference on the write latencies.

5.4 Is a disk cache effective for SSDs?

Modern hard disks are usually equipped with an on-drive RAM cache for two purposes. First, when the disk platter rotates, blocks under the disk head can be prefetched into the cache so that future requests to these blocks can be satisfied quickly. Second, write requests can immediately return as soon as data is transferred into the disk cache, which helps reduce write latency. Similar to disks, SSDs can also benefit from a large RAM cache, especially for relatively high-cost writes. For brevity, we still use the term, *disk cache*, to refer to the RAM buffer on SSDs.

In practice, many low-end SSDs omit the disk cache to lower production cost. For example, SSD-L has no external RAM buffer, instead, it only has a 16KB buffer in the controller, which cannot be disabled. In order to assess the value of the disk cache, we run *random*, *sequential*, and *stride* workloads with requests of 4KB on SSD-M and SSD-H. We use *hdparm* tool to enable and disable the disk cache and compare the performance data to examine the impact of the disk cache. In our experiments we found that disabling the disk cache would not affect read-only workloads. Thus we only present experimental results for write operations.

A disk cache has a significant impact on write performance. Figure 5(a) shows that with disk cache disabled, both SSD-M and SSD-H experienced a significant increase of latencies. On SSD-M, more than 50% of the requests suffer latencies of over 1.3ms, and all requests have latencies of over 400µs. Even with a disabled disk cache, the three workloads share the same distribution of latencies on both SSDs. This indicates that they adopt a log-structured approach and thus are insensitive to workload access patterns.

As a high-end SSD, SSD-H performs substantially better than SSD-M, when the disk cache is disabled. Without the disk cache, the performance strength of SLC over MLC is outstanding. Nearly all requests have latencies of 400-500µs. However, we should note that these latencies are still over 5 times higher than the latencies (75-90µs) observed when the disk cache is enabled. This means that the disk cache is critical to both MLC and SLC based SSDs.

5.5 Do reads and writes interfere with each other?

Read and write in SSDs can interfere with each other. Writes on SSDs often bring many high-cost internal operations running in the background, such as cleaning and asynchronous write-back of dirty data from the disk cache. These internal operations may negatively affect foreground read operations. On the other hand, reads may have negative impact on writes too. For example, reads may compete for buffer space with writes. Moreover, mingled read/write requests can break detected sequential patterns and affect pattern-based optimization, such as readahead.

Since requests submitted in parallel apparently would interfere with each other, we only examine requests submitted in sequence. We especially designed a tool to generate four

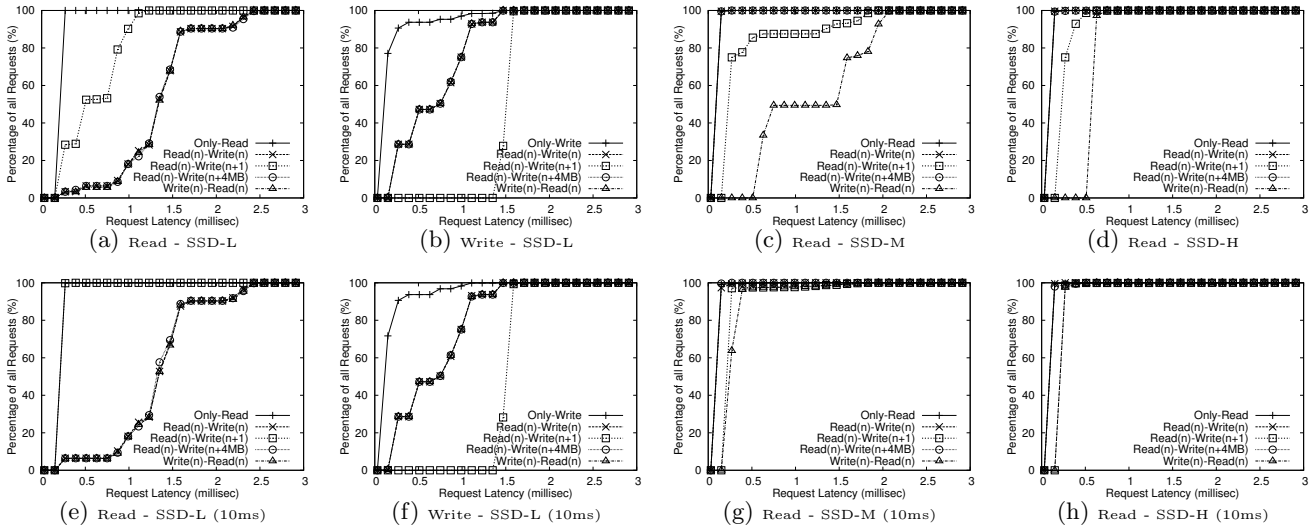


Figure 6: Interference between reads and writes. The second row shows workloads with 10ms interval.

workloads to see how read and write running in sequence interfere with each other. Similar to the Intel® Open Storage Toolkit, this tool directly accesses storage devices as raw block devices, and the other system configurations remain the same. Since sequential workloads are the most efficient operations on SSDs, as shown in previous sections, we use *sequential read* and *sequential write* as two baseline cases, denoted as *only-read* and *only-write*, respectively. We create four types of read/write sequences as follows. Each workload is composed of 1,000 requests for 4KB data.

1. **Read(n)+Write(n)** - sequentially read a page and then write the same page.

2. **Write(n)+Read(n)** - sequentially write a page and then read the same page.

3. **Read(n)+Write(n+1)** - sequentially read a page n and write page $n+1$, then read page $n+2$ and write paper $n+3$, and so on.

4. **Read(n)+Write(n+4MB)** - sequentially read page 0,1,2..N. Simultaneously, we sequentially write pages 4MB apart. Thus, there are two sequential streams.

Figure 6 shows the CDF of read and write latencies of four workloads running on the SSDs. Since writes on SSD-M and SSD-H in this experiment show the same distribution as in Section 5.3, we only show CDF of read latencies here.

Compared to *only-read* and *only-write*, both reads and writes in the four workloads running on SSD-L experienced a substantial degradation (Figure 6(a) and 6(b)). Even when sequentially reading and writing the same page (*read(n) + write(n)* and *write(n) + read(n)*), performance is seriously affected. This indicates that SSD-L does not use a shared buffer for read and write, since a read that immediately follows a write to the same page still encounters a high latency.

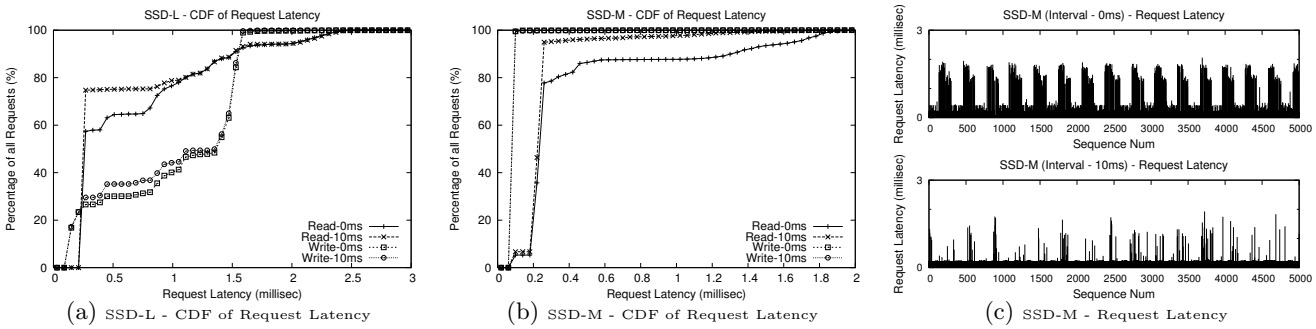
As mentioned in Section 5.3, SSD-L optimizes performance for sequential writes. It seems to detect a sequential write pattern by recording the previous write request's LBN, instead of tracking the sequence of mingled read and write requests. Among the four workloads, *read(n) + write(n+1)* is the only one whose write requests are non-continuous. As expected, it shows the worst write performance, and nearly all requests have a latency of over 1.3ms. Its read operations are also interfered with mingled writes, but it outperforms the other three workloads. We found that inserting an interval of 10ms between two consecutive requests (Figure 6(e)) improves its read performance close to that when running without interleaved with writes. This suggests that the increase of read latencies is a result of asynchronous operations triggered by the previous write.

The other three workloads on SSD-L show nearly identical curves (overlapped in the figures). We can see that writes are apparently affected by mingled reads, but much less severely. Reads, however, are significantly affected by writes, and inserting a delay would not improve performance. We believe that this is because writes just transfer data to the flash memory plane registers, and reads trigger the programing process and cause a synchronous delay (800µs). This also explains the fact that over 80% of the reads in these workloads have a latency of higher than 1ms.

SSD-M and SSD-H both have a disk cache, thus only reads are sensitive to the interleaved read and write accesses. For *read(n)+write(n)* and *read(n)+write(n+4MB)*, read latencies (around 75µs) do not change, even when being interleaved with writes, just like in *only-read*. This indicates that readahead is effective for both cases. However, *read(n) + write(n+1)* and *write(n) + read(n)* are negatively impacted by mingled writes. In the former workload, since each incoming read request is not continuous with the previous read request (interrupted by a write), all of the reads have a latency of over 220µs (i.e. random read latency). In the latter case, it is interesting to see in Figure 6(c) that the curve is similar to the curve of writes when the disk cache is disabled (see Figure 5(a)). This is because when the previous write returns, its dirty data held in the disk cache is flushed to flash memory asynchronously. When a read request arrives, it has to wait for the asynchronous write-back to complete, which leads to a delay. To confirm this, we again inserted an interval of 10ms after each write, and this high read latency disappears, as expected (Figure 6(g)). We also found that even with a delay, both cases only can achieve the performance of *random read*, which shows that readahead is not in effect. We have similar findings for SSD-H.

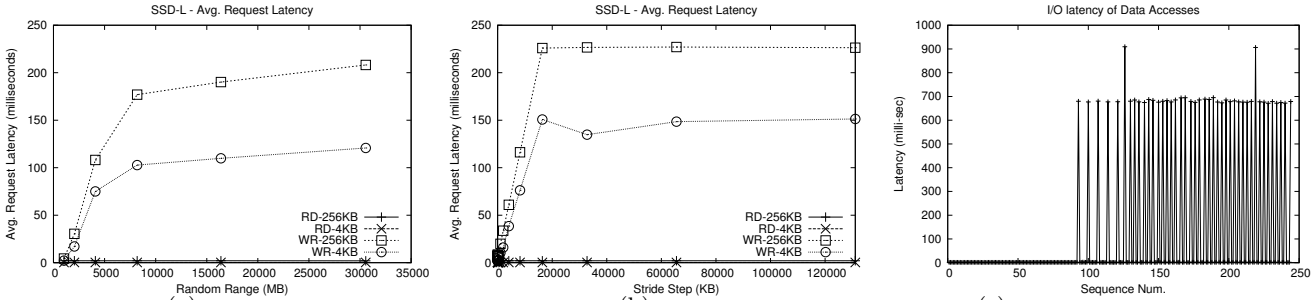
5.6 Do background operations affect performance?

SSDs have many internal operations running in the back-



(a) SSD-L - CDF of Request Latency (b) SSD-M - CDF of Request Latency (c) SSD-M - Request Latency

Figure 7: Background operations on SSD-M. All workloads use 4KB requests.



(a) Random (b) Stride (c) Random Write 4KB (30GB)

Figure 8: Randomness of workloads on SSD-L. Figure 8(c) shows random write of 4KB data in 30GB range.

ground asynchronously, such as delayed writes and cleaning. Though these internal operations create opportunities to leverage idle time, they may also compete for resources with incoming foreground jobs and cause increased latencies. Since write operations are most likely to trigger internal operations, we use the toolkit to run a *sequential* workload using request size of 4KB. The type (read/write) of each request is randomly determined and 50% of the requests are writes. In order to show the performance impact, we intentionally slow down the incoming requests by inserting a 10ms interval between two consecutive requests. Only those *non-stationary* latencies are likely to be caused by background operations. We use this case to *qualitatively* show the impact of background operations on foreground jobs. Due to the space constraints, we only present the data for SSD-L and SSD-M here. SSD-H has similar results.

As shown in Figure 7(a) and 7(b), we can clearly see the performance impact of background operations on foreground jobs, especially on reads. Specifically, after inserting a 10ms interval, the percentage of the reads that have latencies lower than 300µs increases from 57% to 74% on SSD-L, and from 78% to 95% on SSD-M. Writes, in contrast, are barely affected by internal operations, especially on SSD-M. Most writes still fall into the disk cache with low latency. We also examined *sequential* and *random* with write-only requests (not shown here), although we found that background operations can cause some spikes as high as 7-8ms, the overall performance impact on writes is still minimal. To visualize the effect of these background operations, we show the first 5,000 requests of the workloads running on SSD-M in Figure 7(c). We can see that after inserting a 10ms interval, most high spikes disappeared. It is apparent that the background operations are completed during the idle periods such that foreground jobs do not have to compete for resources with them. Though we cannot exactly distinguish various background operations causing these high spikes, we use this case to qualitatively show that such background operations exist and can affect foreground jobs.

5.7 Would increasing workload randomness degrade performance?

In previous sections, we have seen that both read and write on SSDs can be highly correlated to workload access patterns (i.e. sequential or random), similar to hard disks. In this section, we further examine how performance of random writes varies when we increase the randomness of workloads (seek range). We run *random write* and *stride write* on the three SSDs. For *random write*, we vary the random range from 1GB to 30GB. For *stride write*, we vary the stride step from 4KB to 128MB. Each workload uses a request size of 4KB. We did not see an increase of latency with random range and stride steps on SSD-M and SSD-H. Thus, we only report experimental results for SSD-L.

Increasing workload randomness has a significant impact on performance of SSD-L. Figure 8(a) shows that as the random range increases from 1GB to 30GB, the average request latency increases by a factor of 55 (from 2.17ms to 120.7ms), and the average bandwidth drops to as low as 0.03MB/sec. Such a low bandwidth is even 28 times lower than a 7200RPM Western Digit hard disk (0.85MB/sec), and the whole storage system is nearly unusable. The same situation also occurs in *stride write*. As we increase the stride step from 4KB to 128MB, the average latency increases from 1.69ms to 151.2ms (89 times increase) and the bandwidth drops to only 0.025MB/sec. Such a huge performance drop only applies to writes. Increasing request size to 256KB does not help to mitigate such a problem. After investigating workload traces, we found that as the request randomness increases, many requests emerge with a latency of as high as 680ms. The more random the workload is, the more frequently such extraordinarily high-cost requests appear. Figure 8(c) shows the case of randomly writing 4KB data in a 30GB range. The spike of 680ms appears in nearly every three requests in the figure.

Two reasons may cause such high spikes, metadata synchronization and log block merging. An SSD maintains a

mapping table to track the mapping between logical block addresses and physical block addresses. When writing data into SSD, the mapping table needs to be updated in a volatile buffer and synchronized to flash memory periodically. The frequency of metadata synchronization could be sensitive to access patterns. For example, some FTL algorithms employ a B^+ tree-like structure [6,43] to manage the mapping table. Using such a data structure, the more random writes are, the more indexing nodes have to be updated in an in-memory data structure (e.g. *journal tree* in JFFS3). With limited buffer space, SSD-L has to frequently lock the in-memory data structure and flush the metadata in volatile cache to persistent flash memory, which could cause a high spike to appear repeatedly. Some other FTL algorithms (e.g. [19]) can have similar problems with limited RAM space. Interestingly, we also observed similarly high spikes occasionally appearing on SSD-M and SSD-H, and the manufacturer confirmed that it is caused by metadata synchronization.

Another possible reason is log block merging. In order to maintain a small mapping table while achieving reasonable write performance, many FTLs [12, 22, 26, 30] adopt a large mapping granularity (e.g. a block) and use one or multiple *log blocks* to hold data for incoming writes. When running out of log blocks, each page in the log block has to be merged with the other valid pages in the same mapping unit, which causes a high-cost *copy-erase-write* operation. Obviously, if multiple pages in a log block belong to the same mapping unit, only one merge is needed. Thus, the more random the writes are, the more merge operations are needed. In the worst case, each individual page in a log block would belong to a different mapping unit and needs a merge operation correspondingly. This may explain why the performance would not get worse when the stride range exceeds 16MB (Figure 8(b)), because at that time, each page in a log block would belong to an individual mapping unit anyway. Although this speculation still needs confirmation from the SSD manufacturer, it reasonably explains the observed behavior.

5.8 How does fragmentation affect performance?

In order to achieve acceptable performance, many FTL algorithms are based on a log structured approach [3] to handle writes. With such a design, a logical page may be dynamically mapped to any physical flash memory page. When a logical page is overwritten, the data is appended to a clean block, like a *log*. The logical page is mapped to the new position, and the obsolete physical page is only invalidated by updating its metadata. Therefore, if the incoming writes are randomly distributed over the logical block address space, sooner or later all physical flash memory blocks may have an invalid page, which is called *internal fragmentation*.

Such an internal fragmentation may negatively impact performance. First of all, the *cleaning efficiency* drastically drops down. Suppose an erase block has N pages and each block has only one invalid page. In order to get a clean block, N flash memory blocks have to be scanned. During this process, $N \times (N-1)$ page reads, $N \times (N-1)$ page writes, and N block erases are needed. Second, since logically continuous pages are not physically continuous to each other, the readahead mechanism would not be effective any more, which impacts read performance.

We designed a workload, called *striker*, to create fragmentation in SSDs. This workload randomly writes 4GB data into SSD, each request writes only 4KB data, and each overwritten block is randomly selected from the whole SSD space. Under the impact of such extremely intensive random writes, an SSD would be significantly fragmented. In order to show the performance impact of fragmentation, we run *sequential* and *random* workloads using read and write requests of 4KB, and we compare the SSD performance before and after fragmentation. Since *striker* runs excessively slowly on SSD-L, we only report data for SSD-M and SSD-H.

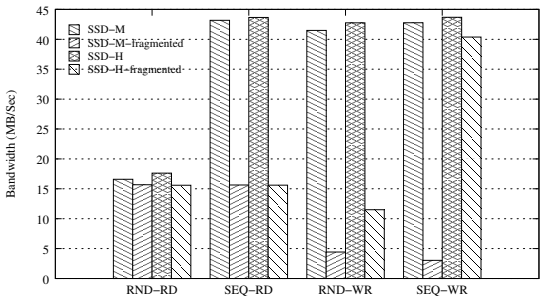


Figure 9: Bandwidth of fragmented SSDs. Four workloads, *Random Read*, *Sequential Read*, *Random Write*, and *Sequential Write*, are denoted as *RND-RD*, *SEQ-RD*, *RND-WR*, and *SEQ-WR*. All workloads use a request size of 4KB.

Figure 9 shows the bandwidths of four workloads running on SSD-M and SSD-H before and after being fragmented. Note that we use request size of 4KB and one job here, thus the sequential workloads do not reach the previously observed bandwidths (Figure 1). We can see that both SSD-M and SSD-H experienced significant performance degradation after being fragmented. Due to the internal fragmentation, logically continuous pages are actually physically non-continuous to each other, which makes readahead ineffective. This causes the bandwidth of *sequential read* to drop from 43MB/sec to only 15MB/sec, which is close to the bandwidth of *random read*. Such a 2.8 times bandwidth decrease is reflected by the corresponding increase of latency from 75µs to 235µs. However, *random read* is not affected much by fragmentation.

Fragmentation has even more significant impact on performance of writes. After fragmentation, the bandwidth of *sequential write* on SSD-M collapses from 42.7MB/sec to 3.02MB/sec (14 times lower). This bandwidth is even much lower than that on a regular laptop disk. Although the disk cache is enabled in this case, the average latency on the fragmented SSD-M increases to as high as 1.27ms. Similar performance degradation is present for *random write*. Compared to that on SSD-M, the write performance drop is less significant on SSD-H, an enterprise-level SSD. Its bandwidth still reaches 40.3MB/sec for *sequential write* and 11.4MB/sec for *random write*. This is attributed to its larger pool of over-provisioned erase blocks (25% of the SSD capacity) and faster erase and write in SLC flash memory.

We believe that the huge performance drop in SSD-M is mainly due to internal fragmentation and exhausted clean blocks. When aggressively writing into SSD-M in a very random manner, the allocation pool of clean blocks is exhausted quickly. Meanwhile, considering the limited life-cycles of MLC flash memory, SSD-M may take a 'lazy' cleaning policy. This causes each write to clean and recycle invalidated pages synchronously. Meanwhile, since most flash memory blocks are significantly fragmented by random writes, each write becomes excessively expensive. The result is that nearly 38% of the writes have a latency higher than 1ms in *sequential write*, and many of them have a latency of 8-9ms. This in turn results in a bandwidth of as low as 3.02MB/sec. On the

other hand, we have to point out that such aggressive random writes only represent a very extreme case, which is unlikely to appear in real-life workloads. However, since most file systems normally generate writes smaller than 256KB, the fragmentation problem still needs to be paid attention, especially in an aged system.

We have attempted to restore SSD performance by inserting a long idle time or mixing read and write requests to create more chance for the firmware to conduct cleaning in the background. However, it seems that a long idle period would not automatically cure a fragmented SSD. Eventually we found a tricky way to restore the SSD performance. We first fill the SSDs with big sequential writes (256KB), then we repeatedly write the first 1024MB data using sequential writes of 4KB many times. The reason this method is effective may be that the cleaning process is triggered by continuous writes to generate clean blocks in the background. Meanwhile, since writes are limited in the first 1024MB, newly generated clean blocks would not be quickly consumed, which helps refill the allocation pool slowly. Other tricks may also be able to recover performance, such as reinitializing the mapping table in the firmware.

6. SYSTEM IMPLICATIONS

Having comprehensively evaluated the performance of three representative SSDs through extensive experiments and analysis, we are now in a position to present some important system implications. We hope that our new findings are insightful to SSD hardware architects, operating system designers, and application users, and hope that our suggestions and guidance are effective for them to further improve efficiencies of SSD hardware, software management of SSDs, and data-intensive applications on SSDs. This section also provides an executive summary of our answers to the questions we raised at the beginning of this paper.

Reads on SSD: Read access latencies on SSDs are *not* always as uniform as commonly believed. Reads on low-end SSDs with no readahead are generally insensitive to access patterns. On higher-end SSDs, however, sequential reads are much more efficient than random reads. This indicates that many existing OS optimizations, such as file-level readahead, will still be effective and beneficial even when migrating to an SSD-based storage. For SSD manufacturers, conducting readahead in SSD firmware is a cost efficient design, which can significantly boost read performance with low cost. More importantly, since most existing file systems have already taken a lot of effort to organize sequential read requests for performance optimization, hardware support on SSDs can be especially effective. For application designers and practitioners, they may believe that high performance can be naturally achieved on SSDs, even without considering data placement. This assumption has to be corrected. Optimizing data placement on SSD is still needed at all levels for improving read performance.

Writes on SSD: As expected, writes on low-end SSDs are strongly correlated to access patterns. Random writes in a large storage space can lead to excessively high latency and make the storage system nearly unusable. However, on higher-end SSDs equipped with a disk cache, write performance is exceptionally good and nearly independent of access patterns. The indication is two-fold. On the one hand, operating system designers should still be careful about random write performance on low-end SSDs. An OS kernel can intentionally organize large and sequential writes to improve write performance. For example, the file system can schedule write-back conservatively with a long interval for clustering writes. Also, flushing out dirty data in a large granularity would be an option. On the other hand, random writes may not continue to be a critical issue on high-end SSDs. Considering the technical trend that a disk cache would become a standard component on SSDs, future research work should not continue to assume that random writes are excessively slow. Instead, avoiding the worst case, such as fragmentation, is worth further research.

Disk Cache: A disk cache can significantly improve performance for both reads and writes in an SSD. With a large cache, high-latency writes can be effectively hidden, and an MLC-based SSD can achieve performance comparable to an SLC-based SSD in most cases. SSD manufacturers should consider a disk cache as a cost-effective configuration and integrate it as a standard component in SSDs, even on low-end ones. With a disk cache, higher-level components, such as OS kernels, can be relieved of optimizing writes specifically for SSDs, and system complexity can be reduced.

Interactions between reads and writes: Reads and writes on SSDs have a surprisingly high interference with each other, even when they are submitted in sequence rather than in parallel. This indicates that upper-level layers should carefully separate read and write streams by clustering different types of requests. Actually, some existing I/O schedulers in OS kernels already manage reads and writes separately, which would be effective on SSDs. Moreover, application designers should attempt to avoid generating interleaved read and write operations. For example, when processing many files in a directory, we should avoid reading and updating only one individual file each time.

Background operations on SSDs: We observed many background operations on SSDs, as well as their performance impact on foreground jobs, especially on reads. Since many operations (e.g. delayed write) must be done sooner or later, if the arrival rate of requests is very high, the performance impact is essentially inevitable. A sophisticated firmware may mitigate the problem. For example, we can estimate the idle period and avoid scheduling long-latency operations (e.g. erase) during a busy time. Moreover, a large write-back buffer can hide the interference more effectively. On the OS and application level, avoiding an interleaved traffic of reads and writes, as aforementioned, is a wise choice.

Internal Fragmentation: On the two higher-end SSDs, we observed significant performance degradation caused by internal fragmentation. In such a condition, the SLC-based SSD obviously outperforms the MLC-based SSD. Although the excessively intensive random writes, which are generated by *striker* in our experiments, rarely happen in practice, it warns that random writes still need to be carefully handled, even on high-end SSDs. Many optimizations can be done in the operating system. For example, some sort of throttling mechanism can be incorporated to intentionally slow down writes into SSD to create more opportunity to cluster consecutive small writes. In addition, spatial locality of writes becomes important. For example, when selecting dirty pages to evict from the buffer cache, intentionally selecting pages that can be grouped sequentially is a wise choice. In order to solve this problem, SSD manufacturers should design more efficient mapping algorithms, and over-provisioning a sufficient number of clean blocks can alleviate the problem. As a short-term solution, a firmware-level defragmentation tool, which can automatically reorganize the page layout inside SSD, is highly desirable. For practitioners, selecting high-end SLC-based SSDs, which show relatively better perfor-

mance under extreme conditions, is a wise choice to handle enterprise workloads with intensive small writes.

Comparing with HDD: If we compare an SSD with a hard disk, the SSD would not necessarily win in all categories. Besides higher price and smaller capacity, even the performance of an SSD is not always better than an HDD. When handling random reads, it is confirmed that SSDs do show much higher bandwidth than a typical 7200RPM hard disk. However, when handling other workloads, the performance gap is much smaller. Under certain workloads, such as random writes in a large area, a low-end SSD could be even substantially slower than a hard disk.

More importantly, we should note that the essential difference between SSD and HDD is not just the performance but the different internal mechanisms to manage data blocks. On HDDs, each logical block is statically mapped on the physical medium, which leads to a simplified management and relatively repeatable performance. On SSDs, the physical location of logical blocks depends greatly on the write order and could change over time (e.g. during cleaning or wear-leveling). This naturally leads to many performance uncertainties and dynamics. In general, we believe it is still too early to conclude that SSD will replace HDD very soon, especially when considering that the 'affordable' SSDs still need significant improvement. System practitioners should also carefully consider many issues exposed in this work before integrating SSDs into the storage hierarchy.

7. RELATED WORK

Flash memory based storage has been actively researched for many years, and a large body of research work is focused on addressing the 'no in-place write' problem of flash memory, either through an flash memory-based file system [4, 6, 17,34] or a flash translation layer (FTL) [12,16,22–24,26,30]. This pioneering research work serves as a concrete foundation in the SSD hardware design.

Recently, some research work has been conducted specifically on flash memory-based SSDs. For example, Agrawal et al. presented a detailed description about the hardware design of flash memory based SSD [3]. In addition to revealing the internal architecture of SSDs, they also reported performance data based on an augmented DiskSim simulator [8]. Similarly, Birrell et al. [5] also presented a hardware design of flash memory based SSD in detail. Concerning the RAM buffer in SSD, Kim and Ahn proposed a write buffer management scheme called BPLRU [25] in SSD hardware to mitigate the problem of high-cost random writes in flash memory. Gupta et al. [19] also proposed a selective caching method to cache a partial mapping table in limited RAM buffer to support efficient page-level mapping. Some papers also suggest to modify the existing interfaces to leverage the unique features of SSDs. For example, the JFTL [11] is designed to remove redundant writes in a journaling file system by directly transferring semantic information from the file system level to the SSD firmware. Transactional Flash [38] further suggests to integrate a transactional interface API to SSDs for supporting transactional operations in file systems and database systems.

Concerning the application of flash memory based SSDs, Sun® has designed a general architecture [31] to leverage the SSD as a secondary-level cache in the standard memory hierarchy. Aiming specifically at peta-scale storage systems, Caulfield et al. [9] proposed a flash memory based cluster to optimize performance and energy consumption for data-intensive applications. Recently, Narayanan et al. [37] presented an interesting model to analyze the potential of merging SSDs into enterprise storage systems from a cost-efficiency perspective. Their study suggests that SSD may not be as appropriate as expected for enterprise workloads.

SSDs are also received strong interest in the database community. Lee et al. [29] examined the performance potential for using SSDs to optimize operations in DBMS, such as hash join and temporary table creation. Lee and Moon proposed an in-page logging approach [28] to optimize write performance for flash-based DBMS. Koltsidas et al. [27] suggested to incorporate both flash and hard disks in a database system by placing read-intensive data on a flash disk and write-intensive data on a hard disk. In general, these SSD related solution papers are based on a common understanding about flash memory based SSD – random writes have poor performance. In our experiments, we confirmed that such assumption still holds on the low-end SSD. However, on the high-end SSDs, we did not see significant performance issues on random writes in normal conditions.

8. CONCLUSION

We have designed a set of comprehensive measurements on three representative, state-of-the-art SSDs fabricated by two major manufacturers. Our experimental results confirmed many well understood features of SSDs, such as the exceptional performance for handling random reads. At the same time, we also observed many unexpected performance issues in the dynamics, most of which are related to random writes. In general, our findings do show that significant advances have been made in SSD hardware design, providing high read access rates combined with reasonable write performance under many regular workloads. However, the substantial performance drop after a stress test showed that it is still too early to draw the firm conclusion that HDD will soon be replaced by SSD. For researchers on storage and operating systems, our work shows that SSD presents many challenges as well as opportunities.

Acknowledgments

We are grateful to our shepherd Prof. Pete Harrison and the anonymous reviewers for their constructive comments to improve the quality of this paper. We also thank our colleague Bill Bynum for reading this paper and his comments. We also thank Shuang Liang and Xiaoning Ding for many interesting and inspiring discussions during this work. This research was partially supported by the National Science Foundation under grants CCF-0620152 and CCF-072380.

9. REFERENCES

[1] Serial ATA revision 2.6. http://www.sata-io.org.

[2] SmartMedia specification. http://www.ssfdc.or.jp.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *Proc. of USENIX'08*, 2008.

[4] P. L. Barrett, S. D. Quinn, and R. A. Lipe. System for updating data stored on a flash-erasable, programmable, read-only memory (FEPROM) based upon predetermined bit value of indicating pointers. In *US Patent 5,392,427*, 1995.

[5] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. In *Microsoft Research Technical Report*, 2005.

[6] A. B. Bityutskiy. JFFS3 design issues. http://www.linux-mtd.infradead.org.

[7] Blktrace. http://linux.die.net/man/8/blktrace.

[8] J. Bucy, J. Schindler, S. Schlosser, and G. Ganger. DiskSim 4.0. http://www.pdl.cmu.edu/DiskSim.

[9] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Proc. of ASPLOS'09*, 2009.

[10] F. Chen, S. Jiang, and X. Zhang. SmartSaver: Turning flash drive into a disk energy saver for mobile computers. In *Proc. of ISLPED'06*, 2006.

[11] H. J. Choi, S. Lim, and K. H. Park. JFTL: a flash translation layer based on a journal remapping for flash memory. In *ACM Transactions on Storage*, volume 4, Jan 2009.

[12] T. Chung, D. Park, S. Park, D. Lee, S. Lee, and H. Song. System software for flash memory: a survey. In *Proc. of ICEUC'06*, 2006.

[13] T. Claburn. Google plans to use Intel SSD storage in servers. http://www.informationweek.com/news/ storage/systems/showArticle.jhtml?articleID=207602745.

[14] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Journal-guided resynchronization for software RAID. In *Proc. of FAST'05*, 2005.

[15] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. DiskSeen: Exploiting disk layout and access history to enhance I/O prefetch. In *Proc. of USENIX'07*, 2007.

[16] E. Gal and S. Toledo. Algorithms and data structures for flash memories. In *Computing Survey'05*, 2005.

[17] E. Gal and S. Toledo. A transactional flash file system for microcontrollers. In *Proc. of USENIX'05*, 2005.

[18] C. Gniady, Y. C. Hu, and Y. Lu. Program counter based techniques for dynamic power management. In *Proc. of HPCA'04*, 2004.

[19] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proc. of ASPLOS'09*, 2009.

[20] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DRPM: Dynamic speed control for power management in server class disks. In *Proc. of ISCA'03*, 2003.

[21] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. DULO: An effective buffer cache management scheme to exploit both temporal and spatial localities. In *Proc. of FAST'05*, 2005.

[22] J. Kang, H. Jo, J. Kim, and J. Lee. A superblock-based flash translation layer for NAND flash memory. In *Proc. of ICES'06*, 2006.

[23] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *Proc. of USENIX Winter*, 1995.

[24] B. Kim and G. Lee. Method of driving remapping in flash memory and flash memory architecture suitable therefore. In *US Patent No 6,381,176*, 2002.

[25] H. Kim and S. Ahn. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proc. of FAST'08*, 2008.

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

[27] I. Koltsidas and S. Viglas. Flashing up the storage layer. In *Proc. of VLDB'08*, 2008.

[28] S. Lee and B. Moon. Design of flash-based DBMS: An in-page logging approach. In *Proc. of SIGMOD'07*, 2007.

[29] S. Lee, B. Moon, C. Park, J. Kim, and S. Kim. A case for flash memory SSD in enterprise database applications. In *Proc. of SIGMOD'08*, 2008.

[30] S. Lee, D. Park, T. Chung, D. Lee, S. Park, and H. Song. A log buffer based flash translation layer using fully associative sector translation. In *IEEE Tran. on Embedded Computing Systems*, 2007.

[31] A. Leventhal. Flash storage memory. In *Communications of the ACM*, volume 51, July 2008.

[32] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou. C-Miner: Mining block correlations in storage systems. In *Proc. of FAST'04*, 2004.

[33] M-Systems. Two technologies compared: NOR vs NAND. In *White Paper*, 2003.

[34] C. Manning. YAFFS: Yet another flash file system. http://www.aleph1.co.uk/yaffs, 2004.

[35] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud. Intel Turbo Memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems. In *ACM Transactions on Storage*, volume 4, May 2008.

[36] M. Mesnier. Intel open storage toolkit. http://www.sourceforge.org/projects/intel-iscsi.

[37] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating enterprise storage to SSDs: analysis of tradeoffs. In *Proc. of EuroSys'09*, 2009.

[38] V. Prabhakaran, T. L. Rodeheffeer, and L. Zhou. Transactional flash. In *Proc. of OSDI'08*, 2008.

[39] D. Robb. Intel sees gold in solid state storage. http://www.enterprisestorageforum.com/technology/ article.php/3782826, 2008.

[40] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *ACM Transactions on Computer Systems*, volume 10(1):26-52, 1992.

[41] Samsung Elec. Datasheet (K9LBG08U0M). 2007.

[42] B. Schroeder and G. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean too you? In *Proc. of FAST'07*, 2007.

[43] C. Wu, L. Chang, and T. Kuo. An efficient B-Tree layer for flash memory storage systems. In *Proc. of Int. Conf. on Real-Time and Embeded Computing Systems and Applications 2003*, 2003.

Notes

[1]When connecting through a SATA controller card supporting parallel I/Os, SSD-M and SSD-H can achieve even higher bandwidths for random workloads and show more significant performance advantages over the low-end SSD. Since handling parallel I/O jobs is out of the scope of this paper, we still use the on-board SATA connectors in our experiments to avoid extra overhead that may be introduced by the controller card.

[2] When filling an SSD for the first time, the mapping table is being created and the number of clean blocks that are available for allocation decreases. During this process, the SSD may experience performance degradation.