Revisiting Widely Held SSD Expectations and Rethinking System-Level Implications

Myoungsoo Jung and Mahmut Kandemir
Department of Computer Science and Engineering
The Pennsylvania State University, University Park, PA 16802, USA
mj@cse.psu.edu and kandemir@cse.psu.edu

ABSTRACT

Storage applications leveraging Solid State Disk (SSD) technology are being widely deployed in diverse computing systems. These applications accelerate system performance by exploiting several SSD-specific characteristics. However, modern SSDs have undergone a dramatic technology and architecture shift in the past few years, which makes widely held assumptions and expectations regarding them highly questionable. The main goal of this paper is to question popular assumptions and expectations regarding SSDs through an extensive experimental analysis using 6 state-of-the-art SSDs from different vendors. Our analysis leads to several conclusions which are either not reported in prior SSD literature, or contradict to current conceptions. For example, we found that SSDs are not biased toward read-intensive workloads in terms of performance and reliability. Specifically, random read performance of SSDs is worse than their sequential and random write performance by 40% and 39% on average, and more importantly, the performance of sequential reads gets significantly worse over time. Further, we found that reads can shorten SSD lifetime more than writes, which is very unfortunate, given the fact that many existing systems/platforms already employ SSDs as read caches or in applications that are highly read intensive. We also performed a comprehensive study to understand the worst-case performance characteristics of our SSDs, and investigated the viability of recently proposed enhancements that are geared towards alleviating the worst-case performance challenges, such as TRIM commands and background-tasks. Lastly, we uncover the overheads brought by these enhancements and their limits, and discuss system-level implications.

Categories and Subject Descriptors

B.3.1 [**Hardware**]: Memory Structures—*Semiconductor Memories*; D.4.2 [**Software**]: Operating Systems—*Storage Management*; C.4 [**Computer System Organization**]: Performance of Systems

SIGMETRICS'13, June 17–21, 2013, Pittsburgh, PA, USA.

Keywords

Solid State Disk, NAND Flash, TRIM Command, Garbage Collection, Performance, Parallelism, Reliability.

1. INTRODUCTION

NAND Flash-based Solid State Disks (SSDs) have recently become immensely popular and been employed in different types of environments ranging from embedded systems to personal computers to high performance computing (HPC) systems. Moreover, various memory and storage systems have been proposed to take advantage of the performance benefits of SSDs over conventional block devices. For example, to reap the benefits of high bandwidth on writes, prior HPC studies consider SSDs as a burst buffer [32], which can absorb heavy write traffic caused by check-pointing [37]. There also exist many applications developed under the expectation that NAND flash is biased toward reads in terms of performance and reliability. Enterprise servers, for example, consider employing SSDs for applications that exhibit many random reads [33, 39, 40] or use them as read caches [28, 3, 30, 36], sitting between main memory and hard disk drive (HDD). Similarly, SSDs are also introduced as a main memory replacement, memory extension, and a part of existing virtual memory systems [13, 12, 14, 39].

While many of these SSD applications and usage scenarios are proposed and developed based on common expectations from SSDs, modern SSDs and NAND flash systems have undergone severe technology shift and architectural changes in the last couple of years. Specifically, NAND flash cells have shrunk from 5x nm to 2x nm in the past four years, and now fewer electrons are stored per floating gate. These cell-level characteristics make flash devices less reliable and introduce extra operations (e.g., multi-step I/O, verification, error correction processes) to successfully complete I/O requests, which in turn imposes longer latencies. State-of-the-art NAND flash packaging technologies employ an increased number of planes and dies within a single flash chip, a command queue, ECC engines, and faster data movement interfaces [9, 35]. These technological changes led in turn to modulations in SSD behavior and performance characteristics. In parallel, SSD internal architecture has dramatically changed; modern SSDs now employ multiple internal resources such as flash chips, I/O buses, controllers and cores in an attempt to achieve high internal parallelism. In addition, to reduce performance variations and garbage collection overheads, flash firmware employs advanced strategies such as finer-granular address mappings, DRAM buffer and background tasks. Finally, thin storage interfaces of mod-

ern SSDs define command feature sets, which provide a way to efficiently expose underlying SSD characteristics to operating systems (OS). Consequently, OS can manage SSD internal resources more efficiently by utilizing system level information.

Unfortunately, most prior works study SSD behavior and performance characteristics based on limited information, or evaluate them based on select I/O access patterns to understand SSD-level parallelism and performance implications. In our opinion, these studies do not help OS and system designers in understanding critical SSD features, and integrating SSDs into existing storage stacks and efficiently optimizing them. Further, a more problematic issue is that, even though SSD NAND flash technology has changed dramatically over the last couple of years, many research groups still employ SSDs based on assumptions that do no hold anymore.

In this paper, we conduct an extensive experimental evaluation with six state-of-the-art SSDs carefully selected by considering different types of flash fabric technologies, manufacturers, cores, chips, and over-provisioning strategies. Based on our empirical evaluations, we next perform a comprehensive data analysis and uncover critical SSD/flash characteristics, which are not reported, to the best of our knowledge, in the literature so far, and are opposite to the widely held expectations on SSDs. Our main goal is to correct common misconceptions on SSDs using new data, which greatly effect performance as well as reliability of modern SSDs, but have not been studied well in the past. We hope to motivate both academia and industry to rethink SSD system design, management and optimization based on our evaluations and data analysis. In this paper, we answer, either directly or indirectly, several questions described in the following subsections, and reveal some critical data regarding state-of-the-art SSDs, which should be, in our opinion, taken into account by both OS and SSD designers. The questions we want to address can be categorized into five groups.

1.1 Rethinking Read Performance

A well-known intrinsic characteristic of NAND flash is that their read performance is tens to hundreds times better than their write performance [34]. In addition, since SSDs have no moving parts, they are expected to provide fast random read accesses [39]. Motivated by these, many platform designers consider SSDs for the applications that contain mostly random reads.

1. Are SSDs biased toward reads at a system level? Why do random reads constitute a performance bottleneck in modern SSDs?
2. For the sequential read accesses, could SSDs support sustained performance? Is there any performance degradation on reads? How could a system achieve a sustained read performance?
3. What is the relationship between read performance and internal SSD parallelism? Can users characterize read performance by examining different I/O access patterns?

1.2 Examining Reliability on Reads

Unlike writes, reads require no erase operation or content-update on NAND flash. Consequently, many computing domains exploit SSDs as a read cache or an intermediate layer when targeting read-intensive workloads to extend SSD lifetime and avoid heavy write penalties.

1. Do program/erase (PE) cycles of SSDs increase during read-only access periods? If it is, why do reads need a block erasure?
2. Are there performance impacts caused by internal I/O operations on reads?
3. What parameters do system-level designers need to control in order to extend SSD lifetime?

1.3 Reconsidering Write Performance

Many schemes have been proposed by prior SSD research to reduce garbage collection (GC) overheads such as over-provisioning [16], DRAM buffer [29, 20, 22], and finer-granular address mappings [19, 15, 34]. Based on this, it is expected that long GC operations can be reordered and deferred, and therefore, they do not cause severe throughput degradation at a system level.

1. How much impact do GC latencies have on system performance in practice?
2. Is there any relationship between the worst-case latency of GCs and system throughput? Could we quantify this relationship?
3. Could DRAM buffer help firmware in reducing the GC overheads? If not, why?

1.4 OS Support

TRIM commands enable OS to invalidate deleted system-level data contents, which can in turn reduce GC overheads significantly. Motivated by this, many emerging SSD platforms (e.g., flash virtual memory, file system, database) are expected to send TRIM commands to underlying SSDs as much as they can.

1. In theory, OS and users can eliminate unnecessary GC operations through TRIMs. How much of GC overheads can be eliminated using TRIM commands?
2. Does TRIM command request pattern matter?
3. Do TRIM commands themselves impose any overheads?

1.5 Characterizing Background Tasks

Since state-of-the-art SSDs employ many computational resources, they could perform SSD-internal tasks in the background. The background tasks are expected to allow SSDs to expedite foreground tasks, which in turn is expected to achieve stable and sustainable performance.

1. What types of background tasks affecting system performance exist in modern SSDs?
2. Do the background tasks of current SSDs guarantee stable and sustainable performance? If not, what is the main difficulty?

2. PRELIMINARIES

State-of-the-art SSDs are composed of multiple cores, memory modules, data buses, and storage media. In the following, we provide a quick overview of SSDs and NAND flash, basic flash firmware features, storage interfaces, and reliability issues.

2.1 SSD and NAND Flash Internals

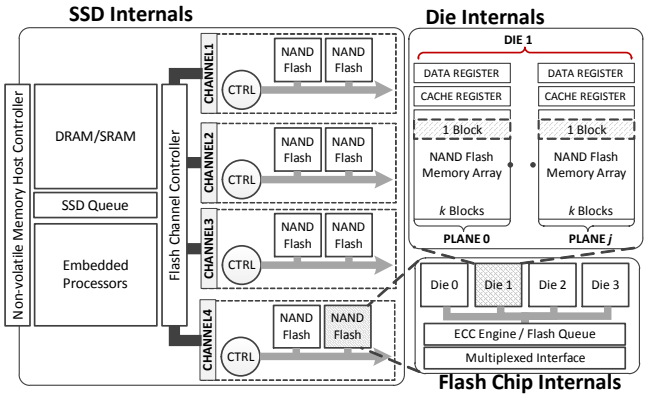


Figure 1: Modern SSD internal architecture. Note that an I/O request can be simultaneously served by many internal resources, which is one of the important characteristics of SSDs.

Modern SSDs and NAND flash chips employ several components to scale their performance under a given technology. As shown in Figure 1, an SSD employs multiple internal resources described below.

Controllers. For physical layer (PHY) management, SSDs have two different controllers: 1) non-volatile memory host controller (NVMHC) and 2) flash channel controller (FCC). NVMHC manages the front-end PHY layer to communicate with outside through a conventional thin interface/bus. FCC, in contrast, handles the back-end PHY layer to control underlying flash packages and corresponding interfaces.

Multicore. While the controllers are responsible for handling the PHY layer, the embedded processor is dedicated to running the flash firmware, which is composed of multiple software layers. Since each layer of the firmware has a different goal and some of their functionalities can be parallelized, modern SSDs employ multiple cores (or processors) in an attempt to minimize computation overheads [23, 25].

Multiple Channels. The flash package interconnection design is very important from a parallelism angle. In general, considering hardware complexity and signal integrity, several flash packages are connected to a single data bus, referred to as *channel*, and multiple channel are used in SSDs.

Flash Package. A flash die is composed of multiple planes, and multiple dies are stacked in a single flash package, which helps one improve storage capacity and flash-level parallelism. Multiple requests can be interleaved through a limited number of CE (chip enable) and I/O pins. In addition, modern flash packages employ a queue and an ECC engine to offload system level overheads imposed by flash commands management and error code checking, respectively.

2.2 Flash Firmware

Depending on the specifics of the underlying hardware configuration, the role of flash firmware can be quite diverse. We now explain the common tasks of the firmware, which have an impact on system performance.

Parallelism. Flash firmware can strip an I/O request over multiple channels, flash packages, and dies therein, in order to improve system performance in terms of both latency and throughput [23, 25, 34, 17]. Since internal parallelism is key to boosting SSD performance, efficient parallelization of data accesses is one of the crucial tasks of the firmware.

Address Mapping. Since NAND flash allows no in-place updates in a block, when a write arrives, flash firmware stores data in a temporal block (which is prepared in advance), and remaps the original address of the request (*virtual address*) and the actual location (*physical address*) of the corresponding data for future reads. In some cases, flash firmware can also remap the addresses in order to improve internal parallelism on writes [25, 10].

Garbage Collection. When the prepared blocks run out, flash firmware needs to reclaim physical block(s) so that it can serve an incoming update request. Since this block reclaiming task, called *garbage collection* (GC), is basically a series of extra internal operations, which include reading/writing live data from the target blocks to new block, erasing the old blocks and updating the mapping information, it can introduce long latencies and degrade performance. To reduce GC overheads as much as possible, modern SSDs employ more elaborate address mapping schemes, including some proposals that perform GC in the background.

Endurance. Flash blocks have limits in terms of the number of program (write) and erase cycles, referred to as *PE cycle*. Typically, a block that experiences higher PE cycles also experiences more errors and worse memory characteristics. Further, once a block reaches its PE cycle limit, it is not available anymore for storage. Since guaranteed PE cycles get smaller as technology shrinks, flash firmware needs to consider endurance related issues.

Wear Leveling. Since not all the information stored within the same location changes with the same frequency, it is important to keep the aging of each block as minimum and as uniform as possible. Flash firmware is also responsible for ensuring that all physical blocks are evenly used (to the maximum extent possible) and keeping the aging under a reasonable value. These tasks are collectively referred to as *wear leveling*.

Disturbance. Since a flash block is composed of multiple *NAND strings* to which the memory cells are connected in series (in groups of 64, 128, or 256), a memory operation on a specific flash cell may influence the charge contents on a different cell. This is referred to as *disturbance*, which can occur on any flash operation and lead to errors in undesignated memory cells.

2.3 Reliability Challenges on Reads

A read operation may fail because of 1) *read disturbance*, 2) *retention error* (leakage problem), and 3) *noise* (e.g., at the power rails). We now explain what SSDs do to address read failures.

ECC Recovery. To avoid failure on reads, *error-correcting codes* (ECC) are widely employed [7]. ECC can correct certain bit errors but typically introduces extra computation cycles on both reads and writes. More specifically, while the encoding takes a few cycles (on writes), the decoding requires lots of cycles. This cycle disparity between encoding and decoding imposes extra overheads on reads, which in turn degrades system performance. Since wider ECCs are required as flash technology shrinks, ECC overheads become more pronounced in modern SSDs.

Read Disturbance Management. Read disturbance can occur when reading the same target cell multiple times without any erase operation. When reading data from a specific

	Basic Test			Specific Test		
Postfix Name →	-L	-C	-Z	-A	-X	-P
Storage (GB)	120	256	256	256	240	240
DRAM (MB)	128	256	0	256	0	0
Numbers of Chips	16	16	8	8	16	16
Technology (nm)	34	25	25-32	32	25	25
Numbers of Cores	2	3	1	3	1	1
Over-provision (%)	15	7.3	14.5	9.5	14.4	14.4

Table 1: Device characteristics of SSDs used in our study.

cell, V_{read} (0 V) is applied to that cell, and all other cells are biased at V_{pass} (4∼5 V), which makes them behave as pass-transistors. As a result, the cells on successive read operations can gain charge, which has similar impact on unintended writes. Since read disturbance can be corrected if the corresponding block is physically erased, it is necessary to erase the block associated with target page address causing the read disturbance. In general, to preserve data consistency, flash firmware reads all live data pages, erases the block, and writes down live pages to the erased block. Some flash firmware migrates live data to new block and remaps the address information between the old and new blocks [2]. This process, called *read block reclaiming*, introduces long latencies and degrades performance.

Runtime Bad Block Management. Even though ECC can correct certain bit errors and flash firmware keeps aging under control, endurance characteristics of flash storage get worse over time. In particular, raw bit error rate increases exponentially with PE cycles [9, 2], which leads to *uncorrectable ECC* (UECC) errors. To avoid UECC errors, flash firmware marks the blocks whose raw error rates have reached the error recovery coverage limit as "bad blocks". It then replaces each bad block with a new block by remapping addresses, in an attempt to avoid future UECC errors. Similar to read block reclaiming and GCs, this *bad block management* also degrades system performance.

2.4 Storage Interfaces, TRIM and SMART

Conventional storage interfaces hinder the scalability of modern SSDs and make efficient SSD management difficult. To help with this, high-speed interfaces such as SATA 6.0Gbps and PCI Express are employed. Further, the most recent version of SATA provides SSD-specific command feature sets, which enable underlying SSDs to expose their internal characteristics to the OS. Specifically, *TRIM*, one of these command feature sets, allows the OS to invalidate data blocks that are no longer considered in use and delete the obsolete data at a system level. TRIM commands are expected to significantly reduce GC overheads and alleviate potential write degradation in many SSD applications. *SMART* is another command feature set, which enables self-monitoring, analysis, and state-reporting. Using it, OS designers can effectively manage SSDs by retrieving internal SSD information such as PE cycles and the number of channels and physical blocks.

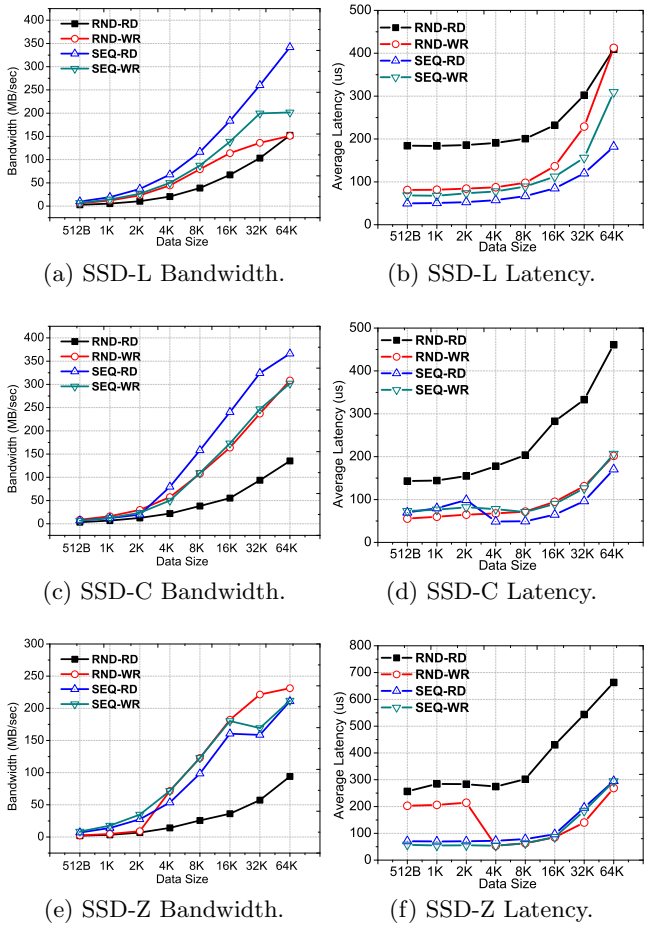
3. EVALUATION SETUP

Solid State Disks. Today, there exist many different SSDs on the market, with quite different performance characteristics based on the vendor and system configurations, in terms of the DRAM buffer size, the number of cores, and the number of flash chips. For our experiments, we chose six representative products shipped by five different SSD-makers. All these SSDs are manufactured between 2011 and 2012, and their firmware are updated with the latest available version for our evaluations. Since our goal is not to perform reverse engineering or make performance comparison across these commercial products, we refer to each of them using a different postfix character, instead of giving its full name. Our SSDs and their important characteristics are listed in Table 1. Since the runtime information provided by different vendors varies a lot, in each of our evaluations, we select an appropriate subset of our SSDs and use them, and also mention the reason behind our selection. In general, SSD-L, -C, and -Z are evaluated for all basic tests, and SSD-A, -X, -P are used for more specific evaluations.

Measurement and Characterization Tools. In order to uncover hidden performance characteristics and examine widely held expectations on our SSDs, we need well-defined I/O access patterns, which can be controlled and reproduced irrespective of the underlying test platform. Consequently, we use Intel open source based storage tool, *Iometer* [18], as our default measurement and characterization tool. Iometer can generate various I/O workloads parameterized in terms of read/write ratio, sequential access/random access ratio, request sizes and the number of queue entries. However, Iometer reports performance results in terms of only average/min/max values at the end of the entire evaluation process. Therefore, for some of our evaluations that require a more microscopic view with finer resolution than what Iometer provides, we use a *modified Iometer*, which captures the latency per individual I/O requests and performance characteristics on a second-basis without any underlying software intervention. Lastly, to evaluate the PHY level latencies, especially for the TRIM command overhead characterization, we use the commercial LeCroy SATA protocol analyzer (*Sierra M6-1*) [31] and double check the protocol status with this analyzer.

Protocol Controls. To accurately evaluate different technologies employed by modern SSDs, we also need a clean evaluation chamber under our control. For example, even though an application tool can mimic system idleness to evaluate background tasks by injecting artificial idle periods, the advanced host controller interface (AHCI) driver/controller can periodically send commands like SMART to examine the underlying system, which can make SSDs continuously busy. To the best of our knowledge, there exist no public tool, which can generate a specific ATA command, check its PHY level latency, and directly handle the AHCI. This is why, for some of our investigations that require the management of ATA commands (e.g., TRIM, SMART) and the control of the AHCI, we needed an in-house driver. Therefore, we also developed an *AHCI miniport driver*, as a part of WDM (windows driver model, which can generate TRIM commands by filling target addresses for the deleted contents with different access patterns (random/sequential), handle SMART commands to check the PE cycles of the SSDs, and manually control specific power modes to examine background tasks.

Experimental System. Our experimental system is equipped with an Intel Quad Core i7 Sandy Bridge 2600 3.4 GHz processor and 4GB DDR3-1333Mhz memory. Intel Z64 chipset is employed as the I/O controller hub in southbridge, and all SSDs we tested are connected to Z64 through the SATA 6.0Gbps interface. We execute all our scenarios in Microsoft



(a) SSD-L Bandwidth. (b) SSD-L Latency.

(c) SSD-C Bandwidth. (d) SSD-C Latency.

(e) SSD-Z Bandwidth. (f) SSD-Z Latency.

Figure 2: Read/write performance comparison under varying data transfer sizes and access patterns. In these comparisons, *RND* and *SEQ* denote the random access pattern and sequential access pattern respectively, and *RD* and *WR* stand for read and write.

NTFS, store logs and output results into separate block devices in a full asynchronous fashion; and neither a system partition nor a file system is created on our SSD test-beds. This configuration allows each SSD test-bed to be completely separated from the evaluation scenarios and tools.

4. TESTING EXPECTATIONS ON READS

The most remarkable performance characteristic shifts on modern SSDs are observed in reads, since they are vulnerable to changes in internal SSD architecture. In this section, we first examine overall performance, comparing reads with other types of operations, and then analyze the challenges on reads in terms of performance sustainability and performance dependency on internal parallelism. Lastly, we uncover reliability problems on read-intensive workloads, which is one of the most critical issues, in our opinion, for both the OS and SSD designers.

4.1 Are SSDs good for applications that exhibit mostly random reads?

To compare read performance with the performance of other types of operations, we executed different workloads composed of *sequential accesses* and *random accesses* with varying data transfer sizes (ranging from 1 sector to 128 sectors) on SSD-L, -C and -Z; we observed that SSD-A exhibits similar performance characteristics to SSD-C, and SSD-P and -X achieve similar performance results to SSD-Z in this test. Performance comparison across our three SSDs is plotted in Figure 2. Specifically, Figures 2a, 2c and 2e show variance in overall bandwidth, and Figures 2b, 2d, and 2f plot variance in latency for SSD-L, -C and -Z, respectively.

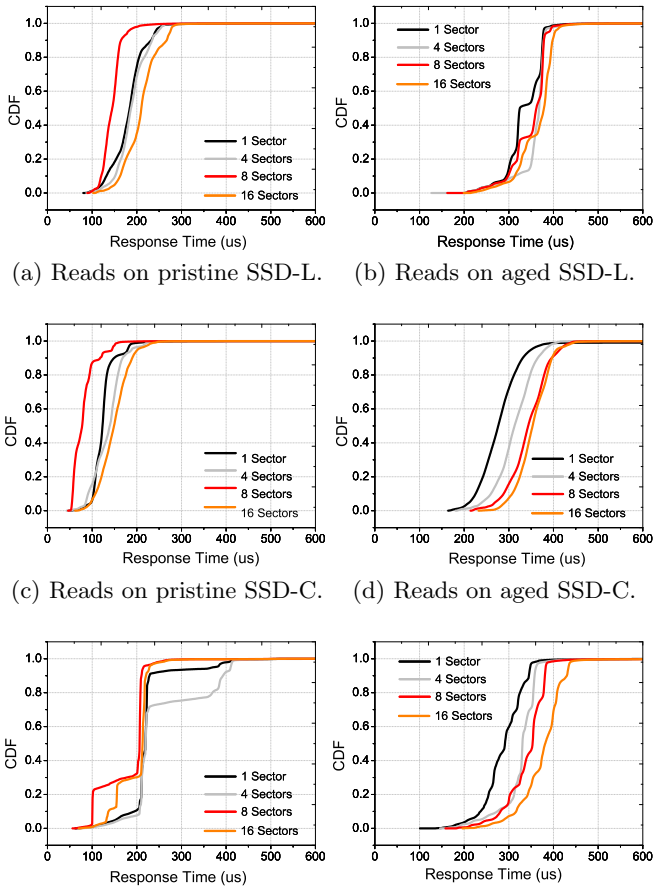
Our first observation is that the *performance values with random read accesses (denoted using RND-RD) are worse than other types of access patterns and operations, including even random write accesses*, which is in *direct contrast* with the widely held expectation on read performance of SSDs in the literature. Specifically, the bandwidth values with random read accesses of SSD-L, -C, and -Z account for 59.7%, 39.4% and 23.7% of the corresponding values with random writes, respectively. Read latency characteristics are not much different from bandwidth; the latency values observed with sequential writes, random writes, and sequential reads only account for 41.3%, 35.2%, and 35.9% of the latencies observed with random reads, respectively (on average).

We believe that the main reason why SSDs can experience opposite performance characteristics at a flash level (reads are much faster than writes at a memory cell level) is the lack of internal parallelism on random reads. Note that, sequential accesses can be striped over multiple channels in a round-robin fashion, and the striped sub-requests can be interleaved across multiple flash dies in each channel. In contrast, random read accesses can potentially create a scenario where multiple requests end up contending for the same internal resources (e.g., channel, package, die, plane), referred to as *resource conflicts*. A request experiencing resource conflict has to wait for the completion of the other request(s) heading to the same resources. Therefore, the resource conflict on random reads causes low parallelism and thus degrades both bandwidth and latency. Unlike reads, flash firmware can easily forward the incoming write requests to a target sitting in idle by remapping addresses, which leads to low resource conflicts and high levels of parallelism.

One potential concern on this read-write comparison would be the impact of the internal DRAM buffer. Since writes can be buffered in DRAM, if the internal DRAM does not flush the in-memory data to the flash medium, write performance would be much better than reads. However, as shown in Figures 2e and 2f, we observed that DRAM-less SSDs, namely SSD-P, -X, and -Z, exhibit very similar performance characteristics to 128MB and 256MB DRAM-equipped SSDs. Further, the amount of data written into those SSDs is over 200GB, which cannot be buffered by a small size DRAM; in fact, the DRAM capacity accounts for under 1% of the total amount of data we wrote in these experiments.

4.2 Can we achieve sustained read performance with sequential accesses?

As demonstrated in the previous section, all the SSDs tested generate their best performance on sequential read accesses. In this section, we further examine whether the sequential read performance can be sustained over time or not, which might have a significant impact on read-intensive SSD applications. For this set of experiments, we executed sequential read accesses with transfer sizes ranging from 1



(a) Reads on pristine SSD-L. (b) Reads on aged SSD-L.

(c) Reads on pristine SSD-C. (d) Reads on aged SSD-C.

(e) Reads on pristine SSD-Z. (f) Reads on aged SSD-Z.

Figure 3: Cumulative distribution function (CDF) of latency variance in sequential reads for "pristine" SSDs and "aged" SSDs (i.e., writing data with a random access pattern to the entire storage space of SSDs). Note that all the curves presented in the CDFs are shifted from left to right as SSDs get older.

to 16 sectors on two different SSD sets; "pristine" SSDs and "aged" SSDs, and measured their latencies per request with the modified Iometer. The results are presented in Figure 3. Specifically, Figures 3a, 3c and 3e plot cumulative distributed function (CDF) of latency for pristine SSD-L, -C and -Z, and Figures 3b, 3d and 3f plot the same for aged SSD-L, -C and -Z, respectively.

One can observe from these results that most of the read requests on all pristine SSDs are served within 300 $\sim$ 400 μsec. However, when SSDs get order, CDF curves shift from left to right exhibiting worse performance characteristics. The aged SSDs take over 600 μsec for serving all the I/O requests, which is two to three times worse than our pristine SSDs. One can conclude from this analysis that, *sequential read performance characteristics get worse with aging and as I/O requests are being processed*, which are unfortunately captured neither by NAND flash data sheets or nor by SSD specifications. We believe that this performance degradation on reads is mainly caused by fragmented physical data layout and reliability management overheads on reads. This read performance degradation also implies that the read behavior of an SSD cannot be easily characterized by the OS by examining only the current I/O request patterns despite recent works [6, 11] claiming that. We will provide a deeper evaluation and more evidence on this read performance characteristic in the following sections.

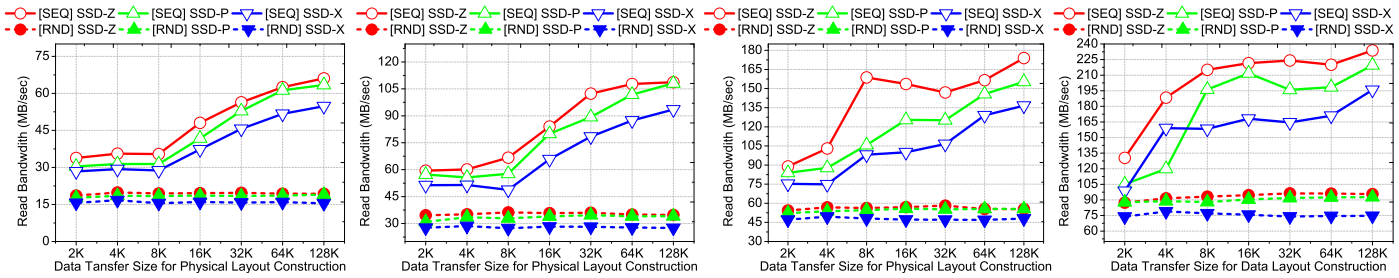
4.3 What is the relationship between read performance and previous writes accesses?

In this section, we analyze read performance of SSDs by building 14 different physical layouts, in an attempt to reveal the relationship between read performance and previous writes accesses as well as internal SSD parallelism. The insight behind this evaluation is that the order of random writes can be transformed into a kind of sequential pattern by the underlying flash firmware, which may have performance impact on current writes as well as future reads. Specifically, the address remapping process allows the flash firmware to easily strip the write requests over multiple internal resources irrespective of the access pattern, which leads to high levels of parallelism as well as improved performance on random writes. However, this physical data layout construction on writes may also introduce different performance behavior for future reads. Unlike writes, for a read to occur, the data to be read should exist and it must reside in a particular location. As a result, the degree of parallelism and performance on reads depends highly on the underlying data layout, *which is constructed during the previous writes.*

To quantify this impact, we chose SSDs that have no DRAM, to minimize any potential side effects of buffering on both read and writes. We then randomly wrote data into the entire address space of the DRAM-less SSDs, SSD-P, -X and Z, with seven different data transfer sizes ranging from 4 sectors to 256 sectors. As a control group, we also wrote data into the same type of SSDs but different devices, using sequential access patterns composed of the same data transfer sizes used in the random writes. We then read the entire space of those SSDs with varying data sizes ranging from 4KB to 32KB, and measured the bandwidth and latency. The results are plotted in Figures 4 and 5.

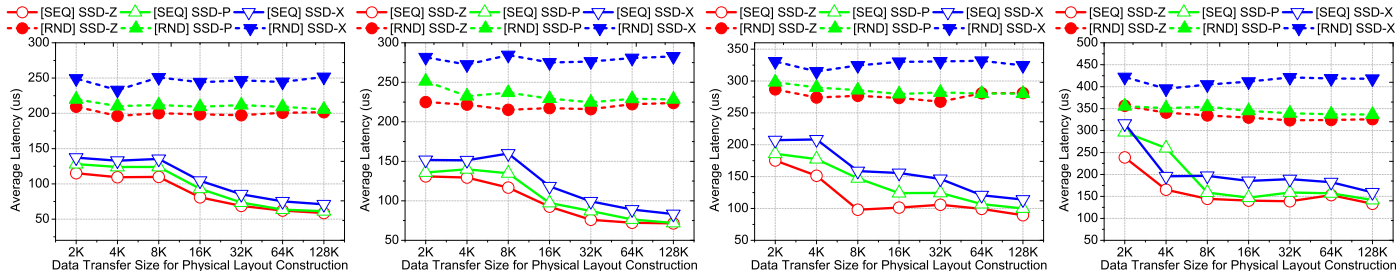
To make our discussion easier to follow, let *RND-PDT* denote the physical data layout resulting from random writes, and *SEQ-PDT* denote the physical data layout resulting from sequential writes. In Figures 4 and 5, the dashed-lines and solid-lines indicate the read performance on RND-PDT and SEQ-PDT, respectively. One can observe that, *read performance significantly varies based on the physical data layout organization even though current I/O request access patterns are exactly the same.* More specifically, bandwidth values for all the evaluations on RND-PDT (Figure 4) are under 80 MB/s, whereas bandwidth values on SEQ-PDT reach up to 220 MB/s. As the data transfer size used during the physical data layout construction increases, the performance gains are more pronounced since this allows the flash firmware to more easily build a physical data layout by sequentially writing data back-to-back. This performance impact is also observed in our latency characterization plotted in Figure 5. While the minimum latency with RND-PDT is 210 μsec, the latency with SEQ-PDT is around 80 μsec.

Based on these evaluations, one can conclude that, since the virtual address space that the flash firmware provides on RND-PDT has been constructed during previous writes, the order of sequential read accesses on the virtual address space



(a) 4KB Sequential Reads. (b) 8KB Sequential Reads. (c) 16KB Sequential Reads. (d) 32KB Sequential Reads.

Figure 4: Bandwidths with different physical data layouts. *SEQ* and *RND* denote sequential writes and random writes, used for the physical data (*PDT*) layout construction. Observe that throughput significantly varies based on the physical data layout, constructed by previous writes, even under same read request patterns.



(a) 4KB Sequential Reads. (b) 8KB Sequential Reads. (c) 16KB Sequential Reads. (d) 32KB Sequential Reads.

Figure 5: Latencies with different physical data layouts. These latency comparison explains how the physical data layout is related to internal parallelism in two aspects. First, the read latency performed on RND-PDT is 2.3 times higher than that of SEQ-PDT that induces lower resource conflicts. Second, as the data movement size of reads increases, the magnitude of the latency improvement with SEQ-PDT is shorter than the improvement with RND-PDT that has many resource conflicts potential.

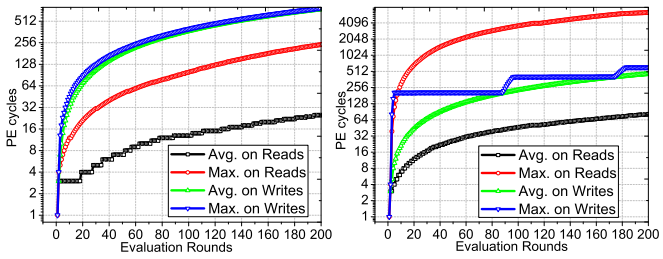
are jumbled. Consequently, the read accesses can suffer from multiple resource conflicts at a specific channel, chip and die, even though the access pattern itself is sequential. This in turn can degrade read performance due to low internal SSD parallelism. In contrast, sequential accesses on SEQ-PDT can be simultaneously served from multiple internal resources in different locations without any major resource conflict since there are no changes in the order of virtual addresses.

4.4 Do program/erase (PE) cycles of SSDs increase during read only operations?

To study the PE cycle characteristics on reads, we executed Iometer with two different *read-only* workloads, composed of sequential and random access patterns, about 200 rounds, each with a running time of 1 hour (total 200 hours). In each round, we sent a SMART command using our in-house AHCI minport driver and measured PE cycles by decoding return codes based on the SMART attribute table [41]. To compare the PE cycles between reads and writes, we also measured the PE cycles on write-only workloads with the same access patterns and measurement method used in the read-only workload evaluations. We observed that unfortunately all the SSDs tested provide insufficient information to understand SSD internal characteristics; in particular, all the data are normalized or provided as percentage based on their lifespan expectations, and some SSDs do not even report their PE cycles on reads. Therefore, we present the reliability evaluation results of a specific version of SSD-A, which is used in Apple MacBook Air; unlike other SSDs we tested, SSD-A provides absolute maximum/average PE cycles on both reads and writes.

Figures 6a and 6b give the variance in PE cycles on two different SSD-A instances under sequential and random access patterns, respectively. One can see from these plots that *PE cycles increase in every evaluation round, in a direct contrast to what the current literature on systems exploiting SSDs would lead one to believe.* In sequential reads, the maximum PE cycles reach the half of PE cycles on writes, as shown in Figure 6a. Ironically, the maximum PE cycles with the random read-only workload are higher than that of writes by about 12x (Figure 6b). We believe that the reason behind this PE cycle increase on read-only workloads is the read disturbance and runtime bad block management. Since these activities require erasing block(s) and live-data migration to the target block(s), *read requests can shorten the SSD lifespan and significantly degrade overall performance.* Further, the disparity between the maximum and average PE cycles tell us another story; wear leveling strategies employed by current flash firmware mainly focus on writes, *not* on reads. While SSD-A firmware keeps reducing the gap between the maximum and average PE cycles on writes, the maximum PE cycles on reads is 247 times higher than the average PE cycles in each round, which makes certain blocks wear out faster and worsen SSD endurance characteristics.

4.5 Is there any performance impact of the reliability management on reads?



(a) Sequential Patterns. (b) Random Patterns.

Figure 6: PE-cycle comparison between reads and writes. Note that PE cycles increase under read-only workloads. Further, with random accesses, the maximum PE cycles on reads are 12 times greater than that on writes.

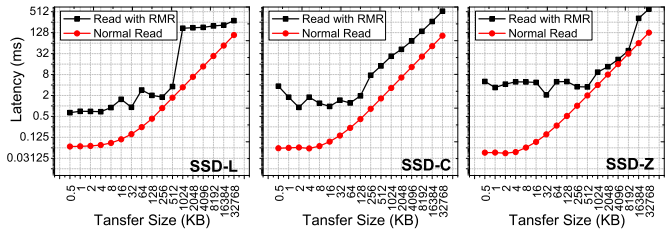


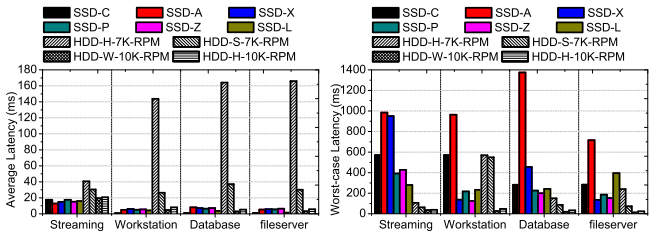
Figure 7: Latency comparison between reads with reliability management (RMR) and ordinary reads (i.e., reads without RMR). When RMR is employed, the latency is at least 5 times higher than the latency of reads without RMR.

All the activities for handling read disturbance management, runtime bad block management, and ECC, referred collectively to as *reliability management on reads* (RMR), require additional I/O operations and compute cycles. These overheads are not revealed to users, but can contribute to long latencies on normal operations. In this section, we examine the latency variation between reads *with RMR* and reads *without RMR*, which is veiled by most SSD manufactures. For our evaluation, we executed the random read-only workload, used in Section 4.4, on three devices, SSD-L, -C, and -Z, and the results are given in Figure 7.

As indicated by these plots, *the read latency with RMR is at least 5 times higher than the latency of reads without RMR*, which would be unacceptable for latency-sensitive SSD applications. Further, RMR overheads are more pronounced with small size random access patterns (1 sector $\sim$ 64 sectors), which is the dominant request size in many file systems. Considering 8 sector (4KB) requests as an example, while the read latencies without RMR on SSD-L, -C, and -Z are 75, 60, and 47 μsec, respectively, the increased read latencies with RMR are 685, 1787, and 4944 μsec, in the same order. Even though the latency disparity between ordinary reads and reads with RMR tends to decrease as the transfer size increases, high RMR-induced latencies with large data sizes (1MB $\sim$ 32MB) still seem to be problematic.

5. TESTING EXPECTATIONS ON WRITES

Modern SSD firmware and architecture are well optimized to improve write performance, but the worst-case performance on writes is still a problematic challenge. In this



(a) Average Latency. (b) Worst-case Latency.

Figure 8: SSD and HDD latency comparison. While SSDs overall outperform HDDs, the worst-case latencies of SSDs are much higher than the worst-case latencies of HDDs.

section, we examine the worst-case write latencies, analyze their correlation with system throughput, and investigate the impact of the internal DRAM buffer on latency. To evaluate the worst-case write latency, we prepared a set of *fully-utilized devices* by writing data (with sequential access pattern) that covers the entire storage space of SSDs. This makes SSDs reclaim block(s) for new incoming requests so that we can easily capture the worst-case latency and system throughput imposed by GCs.

5.1 How much impact does the worst-case latency have on modern SSDs?

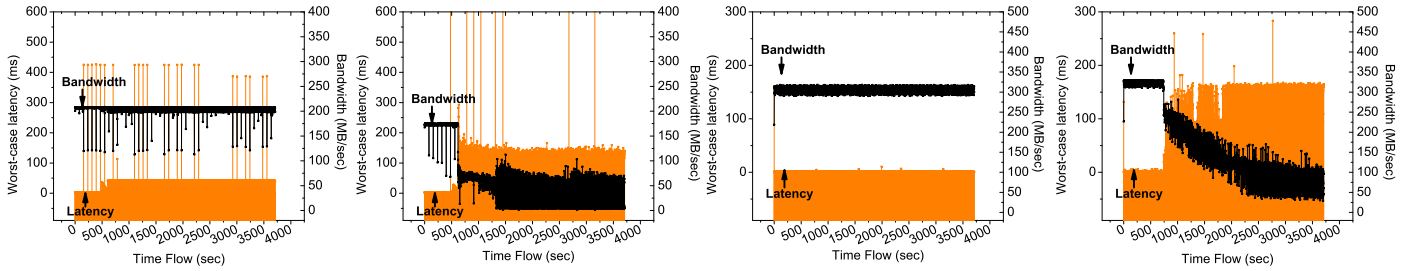
We compare all the SSDs tested and two types of enterprise-scale HDDs (7K RPM and 10K RPM) in terms of both the average latency and the worst-case latency. To quantify the average latency, we use pristine devices, and for the worst-case latency evaluation, we employ the fully-utilized devices for SSD and HDD. We run Iometer with its enterprise open-workloads including streaming, workstation, database and filesever applications, with the fraction of writes being 99%, 20%, 33% and 20% (of total I/Os), respectively.

Figures 8a and 8b plot the average latency and worst-case latency of our SSDs and HDDs. We see that, the average latencies of all the SSDs are better than HDDs, irrespective of the workload type used. Especially, compared to the 7K RPM HDDs, SSDs provide 2 $\sim$ 173 times shorter latency. However, *the worst-case latencies on fully-utilized SSDs are much worse than that of HDDs*, which is problematic for many write-intensive SSD applications. Specifically, the worst-case latencies of all the SSDs tested are 12 and 17 times worse than that of 10K RPM HDD-H and HDD-W, respectively, on an average. This is mainly because NAND flash in SSDs allows no in-place update with overwrites, which leads to GC invocations that contribute to overall latency.

5.2 What is the correlation between the worst-case latency and system throughput?

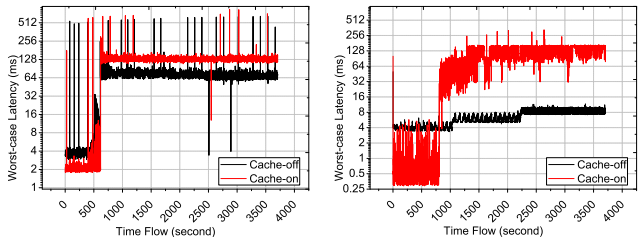
To study the correlation between GC and the worst-case latencies, we prepared two sets of fully-utilized devices, each consisting of SSD-L and -C. We then executed our modified Iometer with write-intensive workloads composed of 100% random accesses and sequential accesses for an hour on these SSDs, and measured the latency and throughput values.

Figures 9a (SSD-L) and 9c (SSD-C) plot the time series for both latency and system throughput along with GCs under the sequential access pattern. For both SSD-L and SSD-C,



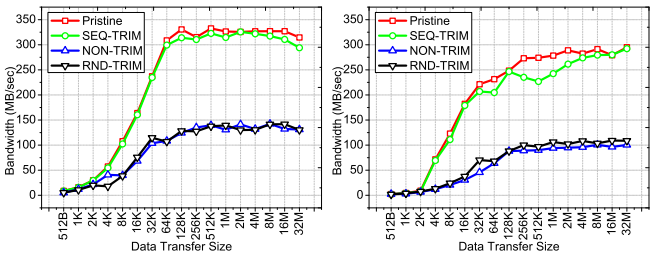
(a) SSD-L non write cliff. (b) SSD-L write cliff. (c) SSD-C non write cliff. (d) SSD-C write cliff.

Figure 9: Impact of write cliff. Initially, SSDs provide reasonable performance even though GCs are invoked. However, once write cliff begins, the performance significantly degrades and is not recovered later.



(a) SSD-L. (b) SSD-C.

Figure 10: Worst-case latency correlation between the DRAM buffer cache and GC. The DRAM buffer provides excellent latency, but after the write cliff, it makes latencies even worse.



(a) SSD-C. (b) SSD-Z.

Figure 11: Bandwidth impact of TRIM. While SEQ-TRIM (the order of target addresses is ascending) can effectively remove GCs, RND-TRIM (the order of targets is random) has no impact on GC overheads.

GCs are infrequently invoked, occasionally imposing long latencies but not impacting system throughput much. The execution in this case recovers performance immediately after the GC. In contrast, with the random write workloads, the worst-case latencies imposed by GCs significantly increase, which in turn dramatically drop the system throughput as shown in Figures 9b and 9d. This performance characteristic caused by GCs under random write workloads is referred to as *write cliff*. Specifically, *once the write cliff begins, SSD latencies (bandwidth) become 11x (3x) worse than the normal case*. Further, *more problematic challenge of modern SSDs is that the performance degradation on the write cliff is not recovered even after many GCs are executed*. We believe that this is because the range of random access addresses is not covered by the reclaimed block(s). Consequently, block reclaims performed by GC are required for each access, which in turn leads to successive GC invocations. Even though we focused on two SSDs in this section (due to space concerns), we observed write cliffs in all the SSDs tested.

5.3 Could DRAM buffer help the firmware to reduce garbage collection overheads?

Since writes can be buffered using internal the DRAM, modern SSDs are somewhat expected to hide GC overheads. In this section, to examine the DRAM impact on GCs, we setup two fully-utilized device sets and write data with a sequential access pattern. In these experiments, one of these two sets are evaluated under the disabled (DRAM) cache (cache-off), and the other set is evaluated under the cache (cache-on). To make device status cache-off, we submit cache-disabled command, which brings the force access unit (FAU) tag of SATA 3.0 [41] for every I/O requests.

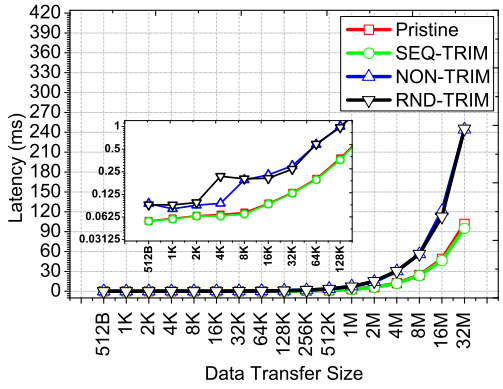
Figures 10a and 10b illustrate the time series comparison between the cache-on and cache-off status devices, SSD-L and -C, respectively. The worst-case latencies of SSD-L are hidden by the DRAM buffer before the write cliff begins. However, once GCs start to be invoked in a series, the latency of the cache-on SSD-L becomes two times worse than that of the cache-off SSD-L. In the case of SSD-C, the performance disparity between the cache-on and cache-off status are more pronounced. *While the DRAM buffer provides four times shorter latency compared to cache-off SSD-C before the write cliff begins, it introduces four times worse latencies when the write cliff kicks in.* Even though a system can react before the data is written into the actual flash device, the DRAM buffer needs to flush the in-memory data to the flash medium periodically. Since target addresses of the buffered data are fully random, this flushing of data introduces a large number of random accesses, which can in turn accelerate GC invocations of SSDs and introduce write cliffs.

6. TESTING EXPECTATIONS ON ADVANCED SCHEMES

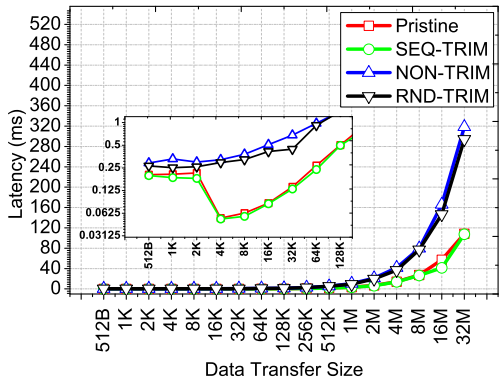
In this section, we evaluate two advanced SSD schemes, TRIM OS support and background tasks, which recently received a lot of attention.

6.1 Can TRIM command reduce GC overheads?

To quantify the performance impact of TRIM commands, we first wrote data over the entire storage space of SSD-C and -Z using sequential and random access patterns, respectively. Then, at a system level, we deleted all the data written using TRIM commands, which consists of two command-



(a) SSD-C.

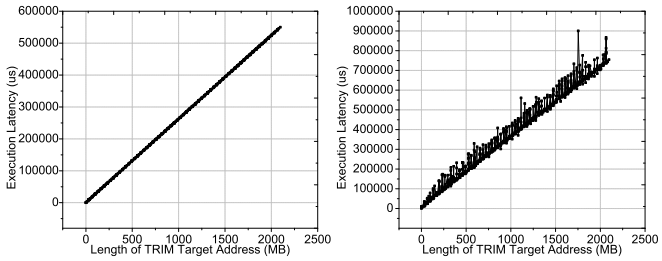


(b) SSD-Z.

Figure 12: Latency impact with TRIM. Similar to bandwidth impact, there is no latency gain with RND-TRIMs.

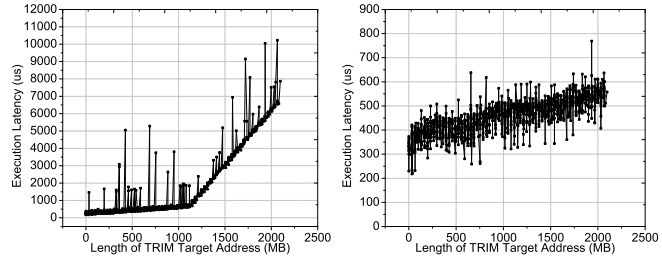
composition steps. The first step is to setup the TRIM field of the DATA-SET-MANAGEMENT command and send it to the SSD to let it know that the host wants to delete data, which is specified target addresses by following the TRIM request data (TRD) frames. Next, we need to configure the logical block address (LBA) range entries in the TRD frames and submit them to the SSD. Using multiple TRIM commands and TRD frames that cover the entire SSD address space, we wiped out all the written data in the previous step. If the order of the target LBAs in the TRD frames is ascending, we refer to the corresponding TRIM command pattern as *SEQ-TRIM*. On the other hand, if the order of the target addresses in the TRD frames form a random access pattern, which has been used for writing data in the previous step, we denote this TRIM pattern as *RND-TRIM*.

We measured performance by writing data of different sizes trimmed using SEQ-TRIM and RND-TRIM. In addition, we also evaluated the performance of our pristine-state SSDs, denoted by *Pristine*, and SSDs that have no TRIM command management, called *NON-TRIM*. The main insight from this evaluation is that, if SSDs tested handle the TRIM commands appropriately, all the written data should be successfully deleted, irrespective of which TRIM pattern is employed. As a result, the trimmed SSDs are expected to exhibit the same performance as our pristine state SSDs. As shown in Figure 11, SEQ-TRIM works very well for deleting



(a) SSD-L. (b) SSD-Z.

Figure 13: E-TRIM overheads. Since E-TRIM performs block erasure on demand and do not return control to the storage system, the host can be disabled until the TRIM process finishes.



(a) SSD-A. (b) SSD-C.

Figure 14: I-TRIM overheads. I-TRIM is more efficient in controlling the TRIM commands, but the latency overheads are still about 6 $\sim$ 153 times worse than a 4KB write-latency.

data in both SSD-C and -Z. As expected, the bandwidth of the SSDs trimmed by SEQ-TRIM is similar to that of Pristine. In contrast, RND-TRIM shows no success in alleviating the GC overheads in both SSD-C and -Z. Our latency evaluation results also show similar performance characteristics. To better understand the execution-time impact of TRIM, we also studied performance at a finer-level, focusing on small data transfer sizes, ranging from 1 sectors to 256 sectors, in Figures 12a and 12b. As can be observed, the latencies of the SSDs trimmed by RND-TRIM are longer than those of the SSDs trimmed by SEQ-TRIM by about 3x on average. One can conclude from this analysis that, *SSDs do not trim all the data, and their behavior is strongly related to the TRIM command submission patterns.* In our evaluation, only SEQ-TRIM could successfully delete data, providing a similar latency to pristine SSDs.

6.2 Does a TRIM command incur any overheads?

One concern that OS designers might have is the potential overheads that can be experienced by an SSD in processing the TRIM command itself. In this section, we measure the latency incurred when processing a TRIM command, using our in-house AHCI miniport driver and the LeCroy protocol analyzer, Sierra M6-1. To do this, we wrote data varying from 512B to 2GB, which has the same address-range coverage as a TRD frame, and sent a TRIM command by filling the corresponding LBAs into the TRD frame. We then captured the time duration from the DATA-SET-MANAGEMENT command submission to the end of the response of the following TRD frames as *TRIM-latency*. Since Sierra M6-1 [31] provides a detailed protocol-level timing

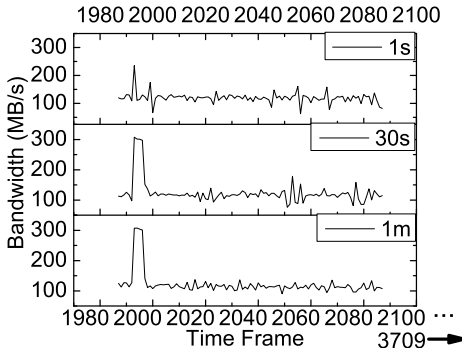
model for the command issue and completion, we are able to capture TRIM-latency for each TRIM process. As shown in the previous section, since SSDs do not get any benefit from RND-TRIM, in this test, we focus on measuring TRIM-latency on SEQ-TRIM.

Based on our observations and experience, we can classify existing TRIM command management strategies into two types: 1) *block erasure in real-time*, called E-TRIM, and 2) *data invalidation based on address and prompt response*, referred to as I-TRIM. As shown in Figure 13, E-TRIM requires long processing times to erase physical blocks based on the target addresses specified by TRIM. For example, to process a single TRIM command covering a storage space of 2GB, SSD-L and SSD-Z take 754 *m*sec and 550 *m*sec, respectively. Note that the TRIM-latency increases linearly with the amount data that LBAs in the TRD frames aim to delete due to physical block erasures.

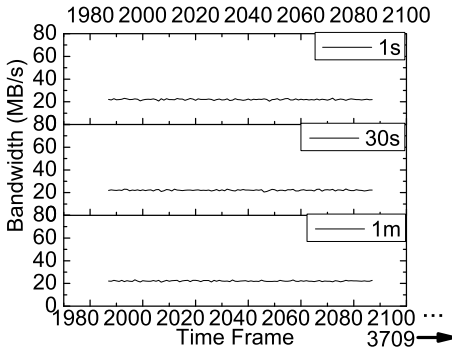
In contrast, SSDs with I-TRIM just mark flags into a mapping table indicating that the contents are not valid anymore and return response immediately. This TRIM strategy can alleviate the time consuming activities of GCs such as the live-data lookup and relocation, even though it does not erase actual blocks. Although I-TRIM has some downsides (e.g., extra DRAM requirement, the possibility of losing in-memory TRIM data in the case of power failure), it is expected to improve SSD reliability to some extent. Specifically, 2x *n*m technology flash chips are much less reliable than larger feature size (3x∼5x *n*m) flash chips, mainly because their memory array suffers considerably more from disturbance and has lower endurance characteristics. Further, the industry observed that such disturbances occur even when erasing a block and the endurance gets worse as PE cycles increase [9, 2]. Consequently, lower technology flash-equipped SSDs employ I-TRIM rather than E-TRIM in an attempt to reduce the number of block erasures as much as possible and improve reliability. From a performance perspective, the latency of I-TRIM is much shorter than the latency of E-TRIM since the former requires only a few cycles to update the underlying mapping table, as shown in Figure 14. However, I-TRIM still takes 400 μs ∼10000 μs to handle individual TRIM commands. This I-TRIM latency is longer than an 8 sector write latency by 6 times ∼ 153 times. We believe that this is because the flash firmware cannot maintain all the mapping information in the internal DRAM, which leads to extra I/Os on the flash medium to load/store the mapping table on demand. As a result, it takes some cycles to manage the mapping table and update the TRIM information in the appropriate entries of the table. One can conclude from this analysis that *modern SSDs require much longer latencies to trim data than normal I/Os would take*, which may put extra pressure on host systems.

6.3 What types of background tasks exist in modern SSDs?

To answer this question, we first wrote sequential data into the entire space of all our 6 SSDs to observe the background task activities. We artificially introduced idle times after writing the data right until GCs begin. In order to avoid *deceptive idle time*, which means that even though application makes the underlying storage system idle, a logical block adapter (e.g., AHCI, ATAPI) can periodically communicate with it by sending a system check command (e.g., SMART), which in turn keeps the storage system busy, we disabled



(a) Cache-on SSD.



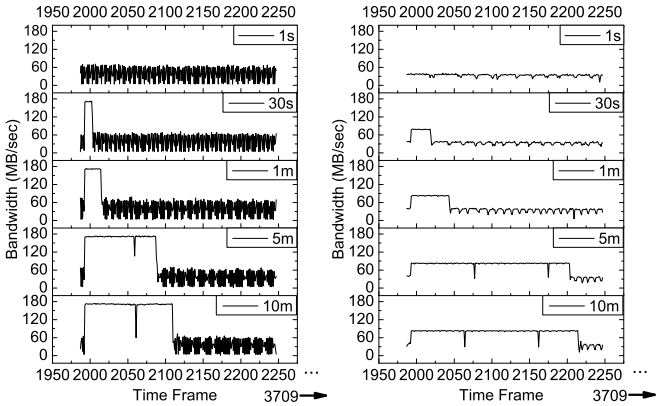
(b) Cache-off SSD.

Figure 15: Background cache flushing (*BFLUSH*). BFLUSH writes in-memory data to the flash medium utilizing relatively short periods (30 sec) compared to the times needed by other background tasks.

the block adapter by intercepting the device-generated check commands and discarding them.We found that, among the 6 SSDs tested, only two SSDs have background activities: SSD-C and SSD-L.

During idle periods, SSD-C flushes in-memory data into the flash medium, creating extra room, which can be used to buffer the new incoming I/O requests. This process is referred to as *background flush*, BFLUSH. *The BFLUSH writes down all in-memory structures and data within 30 seconds and improves SSD-C bandwidth by three-fold* over SSD-C without injecting any idle time, as shown in Figure 15a. To ensure that this performance benefit is coming from BFLUSH, not from any other background tasks, we also performed the same experiment on SSD-C with disabled cache. As shown in Figure 15b, we observed no performance improvement on the cache-off SSD-C, irrespective of the amount of idle time introduced.

In comparison, SSD-L performs GCs in the background, referred to as *background GC* (BGC). As shown in Figure 16a, when we introduced 30-second idle times as in the case of SSD-C, the bandwidth has been recovered to about 98% of that of the pristine state SSDs in 12 seconds. *This performance recovery property of BGC requires much longer time to provide stable performance, but it also exhibits more sustainable performance than BFLUSH.* In our evaluations, BGC needed 10 minutes to fully recover the performance loss on the write cliff. Note that, unlike BFLUSH, BGC of SSD-L can also be observed in the cache-off device. We further study the performance sustainability of these background tasks below.



(a) Cache-on SSD. (b) Cache-off SSD.

Figure 16: Background garbage collection (*BGC*). BGC requires much longer idle times (10 minutes) compared to BFLUSH, but its recovered performance is more sustainable compared to that of BFLUSH (40 times).

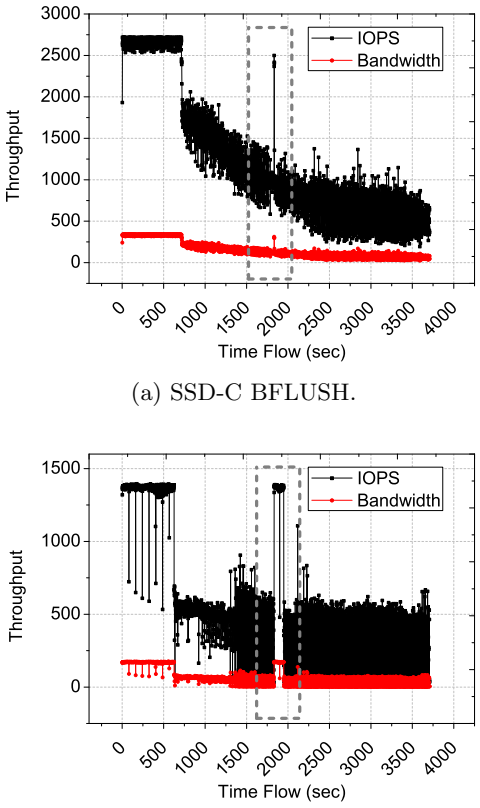
6.4 Can background tasks of current SSDs guarantee sustainable performance?

Even though BFLUSH and BGC can recover their SSD performance unlike SSD-P, -X, -Z and -A that do not have any background activity, we observed that *the performance is not sustained or stable when considering the total execution time of write cliff*. Figures 15 and 16 plot performance characteristics of BFLUSH and BGC, respectively, when we inject an idle period of 1 hour. In the case of BFLUSH (Figure 17a), the recovered performance is sustained for 3 seconds, which provides stable performance for only 0.1% of the total execution time on the write cliff (1,711 seconds). As shown in Figure 17b, the performance sustainability of BGC is not much different from that of BFLUSH. Even though BGC can keep the recovered performance much longer than BFLUSH, it still sustains the recovered performance only about 120 seconds, which accounts for only 7% of the total execution time on the write cliff. We describe the difficulties behind the background tasks in the next section.

6.5 Why can't BGC help with the foreground tasks?

In public domain, there exist several benchmark results published assuming BGC, and even SSD makers indicate that they alleviate GC overheads by utilizing idle times (the vendors of 4 of our SSDs claim that they execute background tasks!). However, excluding the BFLUSH impact, we observed that SSD-L is the only device that performs BGC.

We believe that *there are three main difficulties for modern SSDs preventing them from performing BGC. 1) endurance characteristics, 2) block erasure acceleration, and 3) power consumption problem on idle state.* First, as explained in Section 6.2, lower flash technologies promise much less reliability, and endurance characteristics tend to get worse as the PE cycles increase. Considering the fact that most modern SSDs are using 2x *n*m technology now, and some of them are considering adopting 19 *n*m flash technology, it would be preferred to reduce the number of GC invocations even if they are going to be executed in the background. Second, the flash firmware has to choose victim blocks to



(a) SSD-C BFLUSH.

(b) SSD-L BGC.

Figure 17: Performance sustainability of the background tasks. Even though BFLUSH and BGC almost recover the performance on write cliff, they sustain the recovered performance for very short time (just a few seconds).

reclaim free pages, which are used for serving new I/O requests. If the reclaimed pages are not enough, the firmware has to perform GCs until it secures enough pages. Since each of these processes has to erase the victim blocks, many flash firmware postpone GCs as much as they can, which leads to on-demand GC rather than BGC. Lastly, BGC needs to consume power to perform GC even under the device idle state, and it is difficult to preserve data consistency in the case of power outage. Most host vendors have their own power consumption specification, which regulates the maximum bound on power usage when the device is in the idle mode. Therefore, BGC is inherently limited in page reclaiming during the idle periods. Further, while performing BGC, the firmware has to preserve data consistency by journaling meta-data for the GC in an attempt to prepare for sudden power outages. This also introduces extra I/Os at idle times.

Note that, unlike the other devices tested, SSD-L employs more reliable flash chips with 35 *n*m technology and the highest degree of over-provisioning. These two factors make the write amplification factor of SSD-L lower and thus make the device reliable [16] even though BGC introduces more block erasures. Also, SSD-L needs more power to perform BGC more than SSD-C and SSD-Z, by about 270% and 78%, respectively, in idle periods.

7. RETHINKING SSD SYSTEMS

Based on our experimental results and observations, we

now provide a summary of our answers to the questions we raised at the beginning of this paper.

7.1 Reads

As against the common expectation, the random read performance of SSDs, in terms of both latency and bandwidth, is worse than the other types of operations even including writes. Further, sequential read performance is degraded over time because of two factors: 1) physical data layout changes on writes, which lead to modulations in internal parallelism, and 2) reliability management overheads on reads.

Read Request Reordering. A flash-aware I/O scheduler can transform the random order of addresses on reads to a sequential order, or schedule them by being aware of the internal parallelism, in order to avoid the poor random read performance. There are several studies in the literature that schedule write addresses [29, 20, 22]; in comparison, reordering read requests has received considerably less attention.

Read Frequency Control. Since block erasures are also involved in reads, reads may shorten the SSD lifespan. This means SSD applications need to be aware of the underlying read disturbances and runtime bad block management characteristics. Read frequency control can potentially improve SSD lifespan as well as the read performance. For example, it can remove the hot read spot regions, or reorganize file system blocks to ensure that the underlying physical blocks are accessed as evenly as possible.

De-indirection Interface. De-indirection interface [1, 42] is a promising approach to efficiently manage an SSD by removing the firmware level indirection (address remapping) [21]. Through the de-indirection interface (nameless write), a file system manages the returned physical address as a result of write, and keep updating the physical space changed by the weal-leveling scheme through a callback. In practice, a nameless write interface has to be aware of the underlying read reliability issues like read disturbances and UECC. This is because the address space can be reconstructed even on reads due to such reliability issues, which can in turn corrupt the data consistency of the file system using nameless-writes.

7.2 Writes

Modern SSDs are well optimized to hide GCs, but the throughput of writes significantly drops and the worse-case latency sharply increases when the write cliff is reached. Further, the worst-case latency of SSDs is much higher than HDDs, which implies that it needs to be paid much more attention especially in the context of latency-sensitive applications. Interestingly, the internal DRAM buffer would make the latency imposed by GCs on the write cliff much longer. This implies that the DRAM buffer management is in need of being aware of GCs in order to avoid making the worst-case latency even worse.

Background Task Scheduling. To alleviate the performance overheads on the write cliff, systems may utilize the background tasks by artificially injecting idle times. Since the recovered performance by background tasks is not sustained, the scheduler needs to periodically inject idle periods even under the I/O congestion periods [26]. In addition, considering the fact that the background tasks require long idle periods to fully recover the performance loss on the write cliff, the scheduler can inject idle times in an interleaving fashion and hide potential GC overheads over multiple SSD resources (e.g., flash array storage systems, and SSD RAID systems).

Exposing SSD Firmware API. A more promising approach to handle the background tasks would be exposing APIs that allow a host explicitly to handle flash firmware tasks. For example, similar to the TRIM mechanism, a host can explicitly call GCs or flush data through the DATA-SET MANAGEMENT command on idle times so that the host CPU-burst time can be overlapped with the SSD internal tasks. Based on our experiments, we believe that directly handling SSD internal tasks is much better approach to handle the write cliff than implicitly scheduling the background tasks.

7.3 Advanced Schemes

The magnitude of performance gains with the TRIM commands significantly varies depending on the TRIM request pattern. While SEQ-TRIM can effectively eliminate GC overheads, RND-TRIM has no positive impact on performance. In addition, SSDs require quite long execution times to process TRIMs, which can lead to unexpected performance degradation. To address this, a file system can consider new TRIM management strategies that are aware of the TRIM-process characteristics.

TRIM Buffer and Scheduler. From the beginning of the TRIM process, the host can send TRIM commands by composing target addresses in an increasing order. Similarly, a host module can buffer TRIMs and merge the delete information under the file system in order to transform RND-TRIM to SEQ-TRIM. In addition, it is better to utilize idle times to submit TRIM commands at a system level to avoid potentiol TRIM-latency overheads. A TRIM scheduler can blend legacy I/Os with TRIMs by utilizing system idle periods, thereby hiding the long TRIM execution times.

8. RELATED WORK

A lot of prior work focused on improving SSD performance and overcoming flash-intrinsic limitations such as the erase-before-write problem. Flash translation layers (FTL) have been developed to alleviate the write performance degradation by employing different granular address mappings [8, 19, 15, 34]. In addition to FTL, various buffer management schemes [29, 20, 22] have been investigated to improve write performance. Recently, GC schedulers utilizing idle periods have been proposed to avoid heavy performance penalties on write cliff [24, 27]. There also exist several efforts on SSD architecture. For example,[10, 34, 6, 11, 5, 4] revealed internal SSD architecture in detail. In addition, [25, 17] proposed different page allocation strategies to take advantage of the internal parallelism on writes. There exists a scheduler [23] that explicitly handles I/Os by avoiding resource conflicts, thereby improving the degree of parallelism. FlashVM [39] is a flash virtual memory to reap the benefits of random read performance superiority of SSDs, and Facebook flashcache [28] is a read cache leveraging read performance superiority of SSDs to improve MySQL. [3, 30, 36] also proposed SSD cache, filling the I/O gap between main memory and disks in data centers. In general, these SSD-oriented prior studies have been performed based on common expectations. In our experiments and data analyses, we observed many unexpected performance characteristics and reliability issues, which should be addressed by both academia and industry.

9. ACKNOWLEDGEMENTS

This research is supported in part by NSF grants 1017882, 0937949, and 0833126 and DOE grant DE-SC0002156.

10. CONCLUDING REMARKS

In this paper, we examined widely held expectations and conceptions on modern SSDs using six different commercial SSDs and a series of experiments. Our experimental results revealed many previously-unreported SSD characteristics from both performance and reliability angles. We also discussed what these characteristics mean to both SSD designers and system designers. Our ongoing work includes designing and implementing system support that can take into account our newly-discovered facts on SSDs, and evaluate this support using a diverse set of workloads drawn from embedded computing, enterprise computing and high-performance computing domains.

11. REFERENCES

[1] Arpaci-Dusseau, A. C., et al. Removing the costs of indirection in flash-based ssds with namelesswrites. In *HotStorage* (2010).

[2] Cai, Y., et al. Flash correct-and-refresh: Retention-aware error management. In *ICCD* (2012).

[3] Canim, M., et al. SSD bufferpool extensions for database systems. *VLDB* (2010).

[4] Caulfield, A. M., et al. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *ASPLOS* (2009).

[5] Caulfield, A. M., et al. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *MICRO* (2010).

[6] Chen, F., et al. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *HPCA* (2011).

[7] Choi, H., et al. Vlsi implementation of bch error correction for multilevel cell nand flash memory. In *VLSI* (2010).

[8] Choudhuri, S., and Givargis, T. Deterministic service guarantees for NAND flash using partial block cleaning. In *CODES+ISSS* (2008).

[9] Cooke, J. How ClearNAND flash simplifies and enhances system designs. In *Micron White Paper* (2011).

[10] Dirik, C., and Jacob, B. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. In *ISCA* (2009).

[11] Feng Chen and others. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS* (2009).

[12] Fusion-Io. ioCache. In *datasheet* (2012).

[13] Fusion-Io. ioMemory. In *datasheet* (2012).

[14] Fusion-Io. ioTurbine. In *datasheet* (2012).

[15] Gupta, A., et al. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS* (2009).

[16] Hu, X.-Y., et al. Write amplification analysis in flash-based solid state drives. In *SYSTOR* (2009).

[17] Hu, Y., et al. Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity. In *ISC* (2011).

[18] Intel. http://www.iometer.org/. In *Iometer User's Guide* (2003), Intel.

[19] J. Kang et al. A superblock-based flash translation layer for NAND flash memory. In *EMSOFT* (2006).

[20] Jo, H., et al. FAB: flash-aware buffer management policy for portable media players.

[21] Josephson, W. K., et al. Dfs: A file system for virtualized flash storage. In *FAST* (2010).

[22] Jung, M., et al. Memory system and data storing method thereof. *U.S. Patent 20090248987* (2009).

[23] Jung, M., et al. Physically addressed queueing (PAQ): Improving parallelism in solid state disks. In *ISCA* (2012).

[24] Jung, M., et al. Taking garbage collection overheads off the critical path in ssds. In *Middleware* (2012).

[25] Jung, M., and Kandemir, M. An evaluation of different page allocation strategies on high-speed SSDs. In *HotStorage* (2012).

[26] Jung, M., and Kandemir, M. Middleware - firmware cooperation for high-speed solid state drives. In *Middleware D&P* (2012).

[27] Jung, M., and Yoo, J. Scheduling garbage collection opportunistically to reduce worst-case I/O performance in solid state disks. In *IWSSPS* (2009).

[28] Kgil, T., Roberts, D., and Mudge, T. Improving NAND flash based disk caches. In *ISCA* (2008).

[29] Kim, H., and Ahn, S. BPLRU: A buffer management scheme for improving random writes in flash storage. In *FAST* (2008).

[30] Koltsidas, I., and Viglas, S. The case for flash-aware multi level caching.

[31] LeCroy. http://www.lecroy.com/.

[32] Liu, N., et al. On the role of burst buffers in leadership-class storage systems. In *MSST* (2012).

[33] Liu, Y., Huang, J., Xie, C., and Cao, Q. Raf: A random access first cache management to improve SSD-based disk cache. *NAS* (2010).

[34] N. Agrawal et al. Design tradeoffs for SSD performance. In *USENIX ATC* (2008).

[35] ONFI Working Group. Open nand flash interface 3.0. In *http://onfi.org/* (2012).

[36] Ou, Y., et al. Cfdc: a flash-aware replacement policy for database buffer management. In *DAMON* (2009).

[37] Ouyang, X., et al. Enhancing checkpoint performance with staging I/O and SSD. In *SNAPI* (2010).

[38] Park, S.-H., et al. Design and analysis of flash translation layers for multi-channel NAND flash-based storage devices. In *TCE* (2009).

[39] Saxena, M., et al. FlashVM: Virtual memory management on flash. In *USENIX ATC* (2010).

[40] Srinivasan, M., and Callaghan, M. Flashcache at facebook. In *Facebook White Paper* (2010).

[41] T13. *Serial ATA Specification 3.1*. 2012.

[42] Zhang, Y., et al. De-indirection for flash-based ssds with namelesswrites. In *FAST* (2012).