2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

SSDcheck: Timely and Accurate Prediction of Irregular Behaviors in Black-Box SSDs

Joonsung Kim*, Pyeongsu Park*, Jaehyung Ahn†, Jihun Kim†, Jong Kim†, and Jangwoo Kim*
*Department of Electrical and Computer Engineering, Seoul National University
†Department of Computer Science and Engineering, POSTECH

***Abstract*—Modern servers are actively deploying Solid-State Drives (SSDs). However, rather than just a fast storage device, SSDs are complex devices designed for device-specific goals (e.g., latency, throughput, endurance, cost) with their internal mechanisms undisclosed to users as the proprietary asset, which leads to unpredictable, irregular inter/intra-SSD access latencies. This unpredictable irregular access latency has been a fundamental challenge to server architects aiming to satisfy critical quality-of-service requirements and/or achieve the full performance potential of commodity SSDs.**

In this paper, we propose *SSDcheck*, a novel SSD performance model to accurately predict the latency of next access to commodity black-box SSDs. First, after analyzing a wide spectrum of real-world SSDs, we identify key performance-critical features (e.g., garbage collection, write buffering) required to construct a general SSD performance model. Next, SSDcheck runs diagnosis code snippets to extract static feature parameters (e.g., size, threshold) from the target SSD, and constructs its performance model. Finally, during runtime, SSDcheck dynamically manages the performance model to predict the latency of the next access.

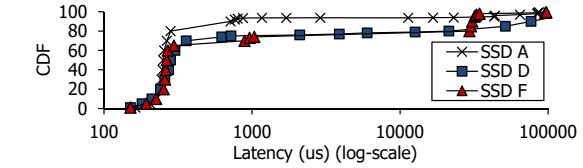
Our evaluations show that SSDcheck achieves up to 98.96% and 79.96% on-average prediction accuracy for normal-latency and high-latency predictions, respectively. Next, we show the effectiveness of SSDcheck by implementing a new volume manager improving the throughput by up to 4.29× with the tail latency reduction down to 6.53%, and a new I/O request handler improving the throughput by up to 44.0% with the tail latency reduction down to 26.9%. We then show how to further improve the results of scheduling with the help of an emerging Non-Volatile Memory (e.g., PCM). SSDcheck does not require any hardware modifications, which can be harmlessly disabled for any SSDs uncovered by the performance model.

***Index Terms*—Performance Modeling, SSD, Storage System**

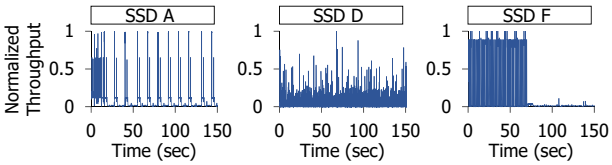
I. Introduction

Solid-State Drives (SSDs) are becoming primary storage devices for not only personal computers but also server-scale storage systems thanks to their higher performance, reliability, and energy efficiency over Hard Disk Drives (HDDs) [1]. For example, all-SSD based storage systems (e.g., Pure Storage [2], [3], NetApp SolidFire [4], EMC XtremIO [5]) are now adopted as effective solutions to meet the industry's increasing storage performance demand. However, aside from the improved performance and energy efficiency, modern SSDs also come with their own disadvantage: *performance inconsistency due to irregular behaviors*.

A modern SSD device, rather than just a fast storage device, is a highly complex computer system which consists of an embedded processor and various hardware units (e.g., flash cells, channels). The embedded processor executes a complex program to enable programmability on the non-volatile cells (e.g., read, write, erase operations) and also improves the device's overall performance and endurance (e.g., garbage collection, buffering, load balancing). As the performance, endurance, and cost of each SSD device highly depend on its actual implementation, SSD device vendors intentionally design and optimize their SSD devices to achieve some predetermined design goals.



(a) Latency distribution on different SSDs



(b) Throughput variation and degradation on different SSDs

Fig. 1: Irregular performance behaviors in commodity SSDs. (a) shows existence of long tail latencies in all SSDs. (b) shows high throughput variations in each SSD and among SSDs.

However, such device-specific optimizations force modern SSD devices to incur highly variable intra/inter-SSD access latencies (i.e., up to 100x variations [6]). As a result, modern SSDs often fail to satisfy the applications' key quality-of-service (QoS) requirements and to achieve the target SSD's full performance potential. Fig. 1 shows the existence of irregular behaviors both within a single SSD device and among different SSD devices. For this experiment, we run a synthetic benchmark which generates a series of random writes and reads on three commodity SSDs. [1] Fig. 1a shows the cumulative distribution function (CDF) of the access latency for each SSD, clearly indicating the existence of extremely long tail latencies in each SSD. This intra-SSD irregular access latency prevents architects from satisfying critical quality-

[1]We anonymized the product names to avoid giving a false impression of comparing the performance and consistency of different SSDs. Note that vendors optimize their products for device-specific design points. Please refer to Table I to check SSDs we used for this work.

978-1-5386-6240-3/18/$31.00 ©2018 IEEE
DOI 10.1109/MICRO.2018.00044

of-service (QoS) requirements [6]–[9]. Fig. 1b shows large time-dependent throughput fluctuations in each SSD and large overall performance differences among different SSDs. This intra/inter-SSD irregular performance variations prevent architects from achieving high performance [10]–[12].

To overcome the irregular behaviors in modern SSDs, architects want *a methodology to accurately predict the latency of future accesses* for their SSDs. With such predictions available, a request scheduler can schedule normal requests over delay-predicted requests, which would significantly improve the latency QoS and the overall throughput.

However, unlike HDDs whose latency model can be easily constructed [13]–[17], accurately predicting the latency of each SSD access is extremely difficult as follows. First, modern SSDs are basically *black-box devices* whose internal structure and device-specific optimizations are considered as the company's proprietary asset, and thus strictly undisclosed to users. Second, an access in modern SSDs is a very complex operation involving many intra-SSD units (e.g., NAND flash, controller), software layers (e.g., flash translation layer), and complex interactions between them (e.g., request buffering). Therefore, the lack of publically available information along with the complex SSD behaviors makes it extremely difficult to construct an accurate performance model for modern SSDs.

In this paper, we propose SSDcheck, a novel SSD performance model to accurately predict the latency of future SSD accesses on commodity SSDs. First, we identify key features critical to latency in modern SSDs by constructing a reconfigurable SSD device [18], implementing various performance-sensitive features [19]–[23] with performance measurement units, and conducting extensive profiling and architectural analysis. Our in-depth analysis indicates that write buffer (WB) and garbage collection (GC) are the two most latency-critical features in modern SSDs. SSDcheck then constructs a general performance model based on the two features.

Next, before launching an application, SSDcheck runs carefully designed diagnosis code snippets to extract device-specific, static feature parameters (e.g., the existence of internal volumes, the size of write buffer, buffer flush algorithm), and uses the information to configure the general SSD performance model for the target device.

During runtime, SSDcheck monitors the incoming I/O requests to dynamically manage the performance model. The model effectively predicts the latency of requests based on the predicted occurrence of irregular behaviors. SSDcheck can also calibrate its model parameters to adapt the model to cope with unmodeled features. If the prediction accuracy is below the target, SSDcheck harmlessly turns off the prediction to avoid mispredicting for SSDs outside the model coverage.

For evaluation, we first validate the prediction accuracy of SSDcheck using seven commodity SSDs selected from four major vendors. As the good performance model must achieve high accuracy for both normal-latency and high-latency predictions, we measure both prediction accuracies for each workload. As a result, SSDcheck achieves up to 98.96% and 79.96% on-average prediction accuracy, for normal-latency and high-latency predictions, respectively.

To show the effectiveness of SSDcheck, we also evaluate two example use cases: a new volume manager and a new I/O scheduler exploiting the latency prediction. The new volume manager, Volume-Aware Logical Volume Manager (VA-LVM), improves the overall throughput by up to 4.29× (or 2.38× on average), while reducing the tail latency at 99.5^{th} percentile down to 6.53% (or 20.3% on average). The new I/O scheduler, Prediction-Aware Scheduler (PAS), improves the overall throughput by up to 44.0% (or 30.0% on average), while reducing the tail latency down to 26.9% (or 30.5% on average). We also show that the new I/O scheduler can further reduce the impact of irregular behaviors by exploiting a fast non-volatile memory (e.g., NVM).

The key contributions of SSDcheck are as follows:

- **Simple, but effective method.** It is a software-based, per-request latency prediction which works for real SSDs.
- **Accurate prediction.** It achieves high accuracy for both normal and high-latency request predictions.
- **Timely prediction.** It immediately predicts the latency of near-future requests during runtime.
- **Strong results.** It significantly improves both QoS and throughput of real-world SSDs.
- **Software release.** We release our framework and use cases to the community.

The rest of the paper is organized as follows. §II explains the general micro-architecture of SSDs and motivates our work with its key design goals. §III describes how SSDcheck extracts key micro-architectural features and constructs the accurate performance model. §IV shows effective use cases which use SSDcheck. We provide detail evaluation results for SSDcheck on §V. Finally, §VI, §VII, §VIII provide discussion, related work, and conclusion, respectively.

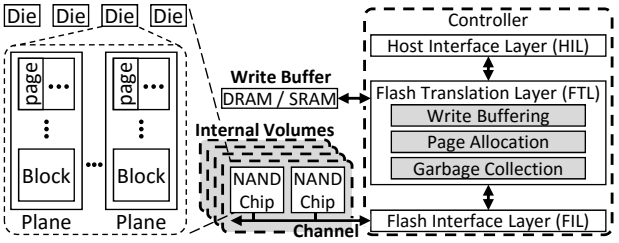
II. Background and Motivation

We first explain the key characteristics of NAND flash memory and a conventional SSD architecture (See §II-A). Next, we show the source of irregular behaviors in SSDs and the benefits of latency prediction (See §II-B, §II-C). Then we discuss the limitations of previous work (See §II-D). Finally, we present the design goals of SSDcheck (See §II-E).

A. Background

A modern SSD itself is a highly complex computer system consisting of various hardware units which are controlled by a complex software stack. Fig. 2a illustrates a typical architecture of a NAND flash chip, the principal storage medium of an SSD. Each flash chip has multiple *dies*, and each die consists of multiple *planes*. Each plane has many *blocks*, and each block consists of a set of *pages*.

A flash chip performs I/O requests in parallel at plane granularity, and each request is implemented with three basic operations: *read*, *write* (also called *program*), and *erase*. Both read and write operations are performed at page granularity, while erase operations are performed at block granularity. The latency values of read, write, and erase operations are around



(a) NAND flash chip (b) Modern SSD architecture

Fig. 2: The general architecture of a modern SSD device.

60 us, 1,000 us, and 3,500 us, repectively [24]. The NAND flash chips have 'erase-before-write' constraints; that is, it is impossible to write to the same page multiple times without erasing its block.

Fig. 2b shows a typical architecture of modern SSD consisting of controller, write buffer, and NAND flash chips [25]. The controller runs firmware consisting of host interface layer (HIL), flash translation layer (FTL), and flash interface layer (FIL). HIL receives I/O requests, and delivers them to FTL which translates them to a series of NAND operations and sends them to FIL. FIL sends the operations to independently working chips through channels. By managing multiple channels, an SSD can achieve inter/intra-channel parallelism [26].

In this structure, the most important component determining the SSD's design point is FTL, who has three key functions: *write buffering*, *page allocation*, and *garbage collection*.

Write buffering. To hide the long write latency of NAND flash, FTL can temporarily store the write data in *write buffer* (WB) and have them drain to the flash chips in background. However, if the buffer becomes full (or reach a certain condition), FTL must flush the buffered data immediately to the flash chips. During this long buffer-flush period, performance-critical read operations can be significantly delayed.

Page allocation. A page allocation unit maps a logical block address (LBA) to a physical block address (PBA) in the flash chips. To hide the long erase latency, it allocates a new PBA (and even flushed data from the write buffer) in a new page selected from a free (erased) page pool. The previous PBAs created by this new allocation are marked as invalid.

To reduce page allocation overhead, some SSDs adopt multiple user-unexposed *allocation volumes* which independently control per-volume page allocation unit, flash chips in a set of channels, and write buffer. FTL determines a target volume by examining the specific address bits of a requested LBA, and each LBA is mapped to only the set of flash chips in the volume.

Garbage collection. When the free page pool becomes short of free pages, FTL calls *garbage collection* (GC) to reclaim more free pages. In general, GC is a very expensive function involving all operations of NAND flash. It first selects victim blocks based on a victim selection algorithm (e.g., greedy). Next, it reads valid pages in the victim blocks, and merge them to free pages (called *merge operation*). Finally, it erases the victim blocks (called *erase operation*) which returns the erased block's free pages to the free page pool. Due to the performance reasons (e.g., inter-chip interference), valid pages in *merge operation* can be moved to the predefined chips specified by *GC volume* only.

FTL can also perform other functions to improve performance (e.g., power management, read-ahead, SLC caching) and reliability (e.g., wear-leveling, ECC), but we will show that above three functions are the most performance-critical features in §III-A and §III-B1.

B. Source of Irregular Behaviors

Modern SSDs are subject to non-trivial performance irregularities in each SSD (*intra-SSD irregularity*) and among SSDs (*inter-SSD irregularity*). The intra-SSD irregularity is mostly the side effect of the intra-SSD components and their interactions. For example, even though a write buffer is effective in hiding the write latency, its irregular buffer flush can significantly delay read requests [27]. In addition, even though allocating multiple volumes is effective in performance, some LBA access patterns can incur irregular volume activations. The irregular GC invocations with their highly variable overheads also significantly contribute to the intra-SSD irregularity.

The inter-SSD irregularity comes mostly from the device-specific design points and optimizations. For example, a vendor might want to make a fast SSD by implementing a large write buffer and many allocation volumes, while a different vendor might want to improve the lifetime of SSD by applying a wear-aware, but slower GC algorithm. As a result, such device-specific designs and optimizations lead to highly irregular behaviors among different SSDs.

C. Benefits of Predicting Irregular Behaviors

Accurately predicting the next request's latency can play a critical role in reducing the performance irregularity in modern SSDs. For example, accurately predicting the slower-than-normal (or high-latency (*HL*)) requests can improve both throughput and QoS by scheduling normal-latency (*NL*) requests ahead of *HL* requests. Mispredicting the *HL* requests as the *NL* requests loses such opportunities. In contrast, mispredicting *NL* requests as *HL* requests can incur significant performance loss by delaying performance-critical *NL* requests. Accurately predicting the *NL* requests affect neither throughput nor QoS.

Therefore, in order to maximize the performance and QoS benefits, while minimizing delayed requests, architects need a sophisticated mechanism which enables accurate *HL*-request and *NL*-request predictions, respectively.

D. Limitations of Previous Work

Some studies [20], [28]–[32] model the GC overhead to find a better GC algorithm. Other researchers [33], [34] extract structure-related features of SSDs (e.g., chunk size, erase unit size) with diagnosis code snippets to better exploit the physical

structure. Another group [35], [36] develops static performance models which just show the existence of workload-dependent irregular behaviors. Finally, many researchers propose to modify the structure of an SSD for reducing the irregular behaviors [10], [23], [37]–[48]. All these studies are orthogonal to our work because they cannot predict the latency of irregular behaviors or require device-specific modifications. More detailed discussions about the related work are provided in §VII.

E. Design Goals

Based on the discussion so far, we set our key design goals as follows.

- **Accurate per-access latency prediction.** It should accurately predict the irregularity of the next requests.
- **Fast, low-overhead prediction.** It should enable fast, low-cost predictions which can be applied during runtime.
- **Software-only solution.** It should not require any hardware modifications.

III. SSDcheck

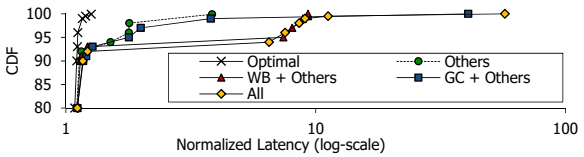
This section first describes SSDcheck's key modeling components (See §III-A). Then we provide multiple diagnosis snippets to extract device-specific internal features (i.e., allocation volume, GC volume, write buffer) (See §III-B). Lastly, we present the performance model of black-box SSDs with extracted features to predict the future slowdown occurrences (See §III-C).

A. Key Modeling Components

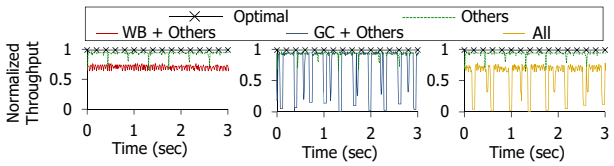
Among many features that make SSDs unpredictable, we identify that the most latency-relevant components are *write buffer* (WB) and *garbage collection* (GC). To verify it, we build a custom-made SSD on top of a Zynq-7000 FPGA with Hynix MLC NAND flash chips. The custom-made SSD has four channels, four chips per channel, and two planes per chip (total 32 planes). We build FTL on top of representative algorithms such as page-level address mapping [49], [50], greedy-based GC [21], [29], and threshold-based wear-leveling [21], [51]. It distributes the buffered write requests to all chips in channels in parallel [26], [34]. For reliability (e.g., read disturbance [52]), each flash controller has an ECC engine [53]–[55].

We implement five versions of the prototyped SSD spanning from an optimal SSD ($SSD_{Optimal}$) to a real SSD (SSD_{All}) to identify the performance impact of WB and GC. $SSD_{Optimal}$ represents the optimal model which immediately returns an acknowledgement for every I/O request without any internal operations. SSD_{Others} excludes WB flush and GC from SSD_{All}, while $SSD_{WB+Others}$ and $SSD_{GC+Others}$ includes WB flush and GC overheads on top of SSD_{Others}, repectively. In this experiment, we run a 4KB random write workload on these five versions of the SSD.

Fig. 3 shows the performance impact of WB and GC on SSDs. Fig. 3a illustrates the latency distribution of five different versions of the SSDs. Here, we compare 99.5^{th} percentile



(a) Latency distribution on different versions of prototyped SSDs.



(b) Throughput comparision: All figures show $SSD_{Optimal}$ and SSD_{Others} on top. Each figure represents a corresponding second-line entity in the legend.

Type of operations	Frequency of occurrence (%)
Others	93.37
WB	6.39
GC	0.24

(c) Portion of each operation.

(d) Latency overhead breakdown. (*HL* means high-latency)

Fig. 3: Performance profiling results from prototyped SSDs. It shows the performance impact of WB and GC.

latency. With WB flush, SSD_{WB} has 8.24× longer tail latency over $SSD_{Optimal}$. SSD_{GC} shows that GC overheads further exacerbate tail latency problems, which leads to 46.67× longer tail latency. SSD_{All}, putting all overheads together, shows the tail latency is 47.12× worse than $SSD_{Optimal}$.

Fig. 3b shows how WB and GC affect throughput on SSDs. Along with the throughput of $SSD_{Optimal}$ and SSD_{Others}, each subfigure additionally shows the throughput of $SSD_{WB+Others}$, $SSD_{GC+Others}$, and SSD_{All}, respectively. $SSD_{WB+Others}$ shows that including buffer flush overheads causes throughput degradation to 70.3%. $SSD_{GC+Others}$ indicates that GC further leads huge throughput variations. With WB flush and GC overheads, SSD_{All} shows severe throughput fluctuation and degradation.

We quantify frequencies of WB and GC operations and present the latency overhead breakdown. Fig. 3c shows the overall distribution of each type of operations: WB, GC, and Others. WB and GC account for 6.39% and 0.24% of the total number of I/O requests respectively, which seems to be a modest portion compared to Others (93.37%). However, the latency overheads of WB and GC take a large portion in the overall latency overheads. Fig. 3d shows the breakdown of latency overheads for all requests (left) and high-latency requests (right). The result shows that 44.3% of latency overheads come from WB and GC. Furthermore, the overheads of WB and GC take 92.3% of all high-latency requests; WB and GC account for 43.4% and 48.9%, respectively. In short, WB and GC are the major factors of the latency of an SSD.

Now, we know two major features causing long tail latency and throughput problems. To distinguish the WB and GC effects from others, we classify requests into normal-latency requests (*NL* requests) and high-latency requests (*HL* requests)

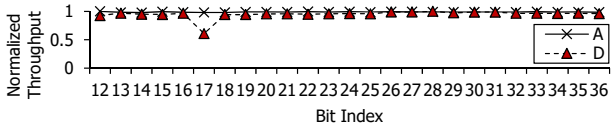


Fig. 4: Throughput comparison over all bit indices on SSD A and D. SSD A has a single allocation volume. SSD D consists of two allocation volumes and its volume index is 17.

based on the existence of interference. If a request experiences internal interference (e.g., buffer flush, GC), then it is *HL*; otherwise *NL*. For example, *NL* read requests fetch data from flash memory without any interference, and *NL* write requests put the data in the write buffer and returns immediately. Note that most requests are *NL* requests, but the *HL* requests severly degrades the performance. Therefore, we should accurately predict the *HL* requests to reduce the tail latency and improve performance.

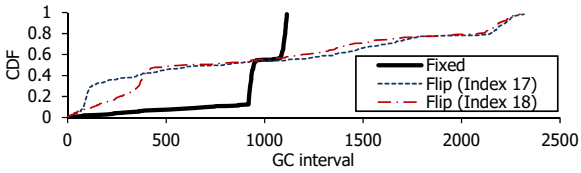
B. Feature Extraction

We extract three key features (i.e., *allocation volumes*, *GC volumes*, *write buffer*) which differ between SSDs to correctly predict the occurrences of future *HL* requests.

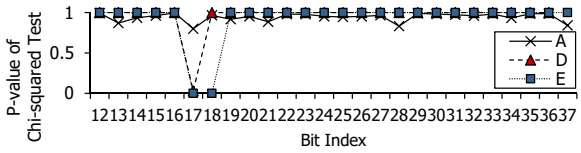
1) Allocation Volumes: An SSD consists of multiple logical allocation volumes including NAND chips, write buffer, and an address mapping table, and the allocation volume is an independent unit to process incoming I/O requests. The page allocation logic in FTL translates a logical address into a physical address determining an actual physical location in NAND chips. Based on the page allocation, the FTL simply forwards each request into an allocation volume based on bit values of specific bit indices (*allocation volume indices*) in its LBA. We need to determine the allocation volume indices involved in the *allocation volume selection* process to eliminate interference among allocation volumes for accurate performance analysis.

SSDcheck uses diagnosis code snippets to identify the allocation bit indices. The snippets randomly access all address ranges while fixing some bits in the addresses. We execute these snippets with different fixed bits and compare the throughputs of each case. When the fixed bits intersect with the allocation volume indices, only corresponding allocation volumes become active. Compared with the cases that activate all volumes, the overall throughput decreases as the number of active allocation volumes is reduced.

We modify flexible I/O tester (fio) [56] to generate a manipulated access pattern which only fixes a value (0 or 1) for a given bit index. We execute the manipulated workload for each bit index in an available address range, and measure the throughputs of each case. Fig. 4 shows the throughput results on all bit indices. The throughput is halved in a bit index 17 on SSD D. Here, we can ascertain that the SSD has two allocation volumes, and the 17^{th} bit determines a corresponding allocation volume. The FTL forwards incoming requests by using the 17^{th} bit value. In contrast, the throughput is constant for all bit indices on SSD A. Therefore, we can



(a) CDF of GC intervals on SSD E. Two bit indices determine GC volumes.



(b) P-value of chi-squared test of GC interval over all bit indices.

Fig. 5: GC volume identification. (a) shows GC interval distribution on SSD E. Only cases of bit indices 17 and 18 show different distributions. (b) shows the result (p-value) of chi-squared test. SSD D and E have multiple GC volumes while SSD A has a single GC volume.

conclude that there are no multiple allocation volumes in SSD A.

2) GC Volumes: Similar to allocation volumes, an SSD also has multiple GC volumes in which GC is triggered independently. The FTL handles GC operations (e.g., victim selection, merge, erase) in each GC volume separately. A GC volume is determined by bit values of some bit indices (*GC volume indices*) in the LBA. Therefore, we need to extract the GC volume indices to distinguish GC overheads in one GC volume from other GC volumes. This GC volume identification makes SSDcheck achieve more accurate performance analysis.

To identify GC volume indices, we carefully design diagnosis code snippets: *Fixed* and $Flip_x$. *Fixed* performs multiple writes to the same address, and this leads to self-invalidation (i.e., all pages in a block are invalidated before GC [57]). Since all blocks are already invalidated, SSD hardly triggers the merge operations during GC, and only issues erase operations to reclaim free blocks. Without the merge operations, each GC invocation reclaims the similar number of free pages. Therefore, the number of write requests between back-to-back GC invocations [2] is similar. We call the number of write requests between two consecutive GC invocations as *GC interval*, and Fig. 5a shows the distribution of GC intervals on SSD E. Most GC intervals of *Fixed* are similar.

$Flip_x$ alternately performs writes to two addresses of which the only x^{th} bit value is different. If all requests go into the same GC volume, GC interval distribution would be similar to the distribution of *Fixed*. If two requests are forwarded into two different GC volumes respectively, the distribution would be different with the distribution of *Fixed*. In Fig. 5a, only $Flip_{17}$ and $Flip_{18}$ show the different distribution compared to

[2]We can easily identify GC requests by measuring latency. GC consists of an erase and multiple reads and writes; thus, GC shows significantly longer latency than the others, which makes GC requests easily be distinguished.

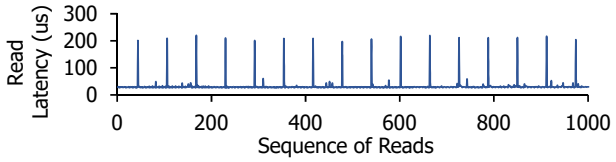


Fig. 6: WB profiling results (SSD A). High latency requests are periodically observed as following the size of write buffer.

Fixed. Therefore, SSD E has four GC volumes, and its GC volume indices are 17 and 18.

We use a chi-squared test between *Fixed* and $Flip_x$ over all bit indices to determine which bit index get involved with the GC volume selection. Fig. 5b shows p-value, a result from the chi-squared test, over all bit indices on SSD A, D, and E. High p-value means that the GC interval distributions of two cases are similar and vice versa. As shown in the figure, SSD A shows a high p-value (nearly one) over all bit indices, so SSD A has a single GC volume. Otherwise, SSD D and E shows zero p-value on some bit indices (17 and 17, 18 respectively); therefore, SSD D and E consist of multiple GC volumes, two and four volumes respectively. Note that, allocation volume indices and GC volume indices are the exactly same on each SSD.

3) Write Buffer: An SSD uses a write buffer to hide a long write latency of NAND flash and maximize internal parallelism by using multiple NAND chips concurrently [58]. An SSD stores incoming write requests into the write buffer and immediately returns an acknowledgment to the host. In background, the write buffer manager in FTL flushes data from write buffer to NAND chips in specific conditions (e.g., the write buffer is full). Note that write buffer exists in each internal volume, and each write buffer is independently flushed into NAND flash.

We need to find the buffer-related features for the accurate the latency and GC models. While flushing the write buffer into NAND chips, the chips cannot process other read requests, which results in *HL* read requests. Moreover, the GC invocations are closely related to the number of buffer flush events because the buffer flush consumes free pages that GC generates.

We provide a method to extract key characteristics of the write buffer (i.e., the buffer size, buffer type, flush algorithms). Algorithm 1 describes how SSDcheck finds *buffer size*, *buffer type*, and *flush algorithms* for each allocation volume. To figure out the buffer size, we carefully design a diagnosis code snippet (line 1: *background_read_test*). This test concurrently issues random writes with random reads whose address distribution differs with that of writes. While issuing reads in the background, the test generates 4KB random writes which stall for thinktime between I/Os. [3]

Since read requests get data from NAND flash chips, the requests are blocked while the NAND chips process write operations (i.e., buffer flush). By monitoring latencies of read requests, the test can detect timing and latency overheads of buffer flush. Fig. 6 shows results of *background_read_test* on SSD A. It shows periodic latency spikes on sequence of read requests, and we can calculate the buffer size from the number of write requests between back-to-back *HL* read requests. Here, the size of SSD A's write buffer is 248KB.

SSDcheck further identifies an additional flush algorithm (line 4: *read_trigger_flush_test*). Buffer flush is commonly triggered when the buffer is full; however, some SSDs trigger buffer flush on every read request even if the write buffer holds only a single page. SSDcheck issues the workload mixing read/write requests with random thinktime to check whether a read request explicitly triggers the buffer flush or not. If a read request shows the high latency regardless of the number of previous write requests and submission timing of each request, the test determines that the *read-trigger* flush algorithm exists.

For SSDs having the read-trigger flush algorithm, SSDcheck issues random write requests to a single allocation volume in each SSD and detects a periodic high latency (line 5: *write_only_test*). If the test observes the periodic high latency and its latency is similar to NAND write overheads, the test calculates and returns buffer size (i.e., the total amount of write requests between adjacent periodic *HL* requests). When the test detects buffer size, SSDcheck classifies its buffer type as $fore$ (line 6) because this SSD copes with buffer flush before returning an acknowledgment for write requests. When write buffer can store incoming writes in another buffer while flushing data from write buffer to NAND chips, SSDcheck classifies its buffer type as *back* (line 2).

Finally, SSDcheck returns key characteristics of write buffer (e.g., buffer size, buffer type, flush algorithms). Note that SSDcheck conducts the write buffer analysis subsequent to the internal volume analysis to eliminate an interference among internal volumes for accurate analysis.

Algorithm 1: Write Buffer Analysis.

```
   output : a tuple buffer size, buffer type, and a list of flush algorithms.
   /* Find the size of write buffer. If a test cannot find
      buffer size, return 0                                          */
1  if buf_size = background_read_test() > 0 then
2      buf_type = back
3      flush_algorithms = [full_trigger]
4  else if read_trigger_flush_test() then
       /* Find existence of read-trigger flush algorithm */
5      if buf_size = write_only_test() > 0 then
6          buf_type = fore
7      else
8          buf_type = Unknown
9      end
10     flush_algorithms = [full_trigger, read_trigger]
11 else
12     return (0, Unknown, Unknown)
13 end
14 return (buf_size, buf_type, flush_algorithms)
```

C. SSD Performance Model and Runtime Framework

SSDcheck extracts static feature parameters and constructs a performance model for a target device (See §III-C1). Then,

[3] We use multiple thinktime (e.g., 500us, 1000us, 5000us) for this test, and verify that results (*calculated buffer size*) are the same in all cases.

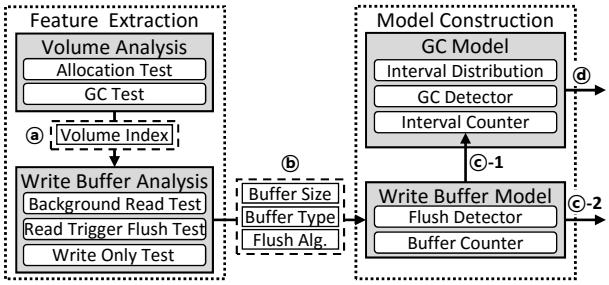


Fig. 7: How SSDcheck constructs a performance model for a target SSD.

during runtime, SSDcheck dynamically manages the performance model to predict the future latency (See §III-C2).

1) SSD Performance Model: Fig. 7 shows how SSDcheck constructs a performance model for the target SSD. SSDcheck first extracts the internal volume features (ⓐ), uses the information to extract the device-specific write-buffer parameters by performing various tests (Algorithm 1), and delivers the extracted features to the model construction part (ⓑ). The model construction dynamically builds two models: *write buffer model* and *GC model*.

Write buffer model. The write buffer model dynamically manages *buffer counter* and *flush detector* during runtime. When a write request arrives to the write buffer model, the model increases the buffer counter. When the buffer counter reaches the buffer size, the flush detector assumes an occurrence of buffer flush and resets the buffer counter. At the same time, the flush detector notifies this buffer flush event to the GC model (ⓒ-1), and also exposes the event to the outside of the performance model (ⓒ-2). If the write buffer uses the *read-trigger* flush algorithm, the flush detector assumes an occurrence of buffer flush on receiving a read request with non-empty write buffer.

GC model. The GC model manages *interval counter* and *interval distribution* to enable our history-based prediction which correlates the history of buffer flushes to GC occurrences. On detecting a buffer flush, the GC model increases the interval counter (ⓒ-1) and also stores the counter in the interval distribution. By examining the interval distribution, the GC detector determines whether a subsequent buffer flush would trigger a GC process or not. If the GC detector believes that a current request will be affected by GC, it exposes this GC occurrence to the outside of the performance model (ⓓ).

This history-based approach shows good prediction accuracy because the interval distribution is not change in a short period. The GC interval is closely related to the number of valid (or written) pages in the merge step of GC, and the number depends on the GC victim selection algorithm and the valid page distribution. Here, the valid page distribution changes continuously and slowly; thus, we can predict the future GC occurrences based on the interval distribution.

2) SSDcheck's Runtime Framework: Fig. 8 shows our runtime modeling framework consisting of *volume selector*, *prediction engine*, *latency monitor*, and *calibrator*. In this framework, the host sends a query to the framework to check

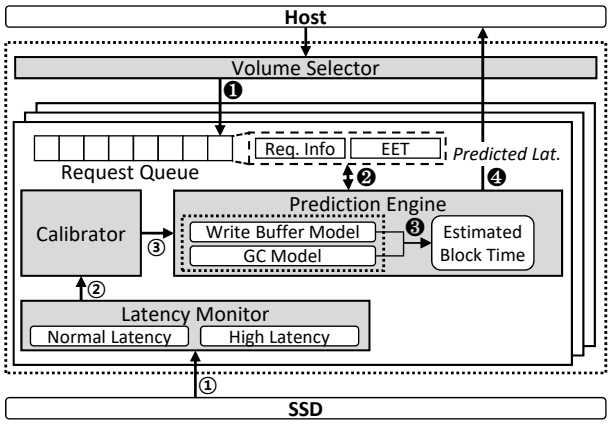


Fig. 8: The runtime framework of SSDcheck

an occurrence of an irregular event. The framework then returns the request's predicted latency.

Volume selector. The host issues a request to the framework for checking whether the request would be slow or not. Then the request passes through *volume selector* which forwards it into a corresponding internal volume identified by volume indices of its LBA (❶). The framework pushes the request into the request queue of the selected internal volume, and queries prediction to the prediction engine (❷).

Prediction engine. The prediction engine adopts *Estimated Block Time* (EBT) to predict the future HL requests. The prediction engine updates internal information (e.g., EBT, buffer counter, interval counter, interval distribution) in every I/O completion. When it expects the occurence of buffer flush or GC, EBT becomes the sum of the current time and the corresponding overheads (i.e., buffer flush or GC) extracted from the runtime monitoring (❸) (See *Latency Monitor*).

For an incoming I/O request, the prediction engine calculates a predicted latency, *Estimated End Time* (EET), by subtracting the query submission time from EBT. If the predicted latency (EET) is negative or lower than the latency threshold, the prediction engine assumes the incoming I/O request is NL request. If not, the engine assumes the request as HL request. The framework then returns the prediction results (e.g., EET, slow event) to the host (❹).

Latency monitor. SSDcheck monitors the latency of every request and classifies the requests into *NL* or *HL* based on *latency threshold* (①) as described in §III-A. Then the latency monitor delivers the latency logs and the categorized information to the calibrator (②).

To find the proper latency thresholds for write and read requests, we run two synthetic workloads. For write requests, we execute a sequential write workload, which shows small interference in SSDs [11], and use the spike latency from this test as the latency threshold. For read requests, we issue random reads (to make all reads go to the the NAND) and set the latency threshold so that it covers the NAND read latency.

Calibrator. SSDcheck's runtime framework dynamically adjusts the performance model to resolve any model discrepancy incurred by unconsidered features. The calibrator

receives latency logs from the latency monitor, and uses them to detect a buffer model discrepancy. For example, the latency monitor notifies an occurrence of HL requests to the calibrator while the buffer counter does not reach the buffer size yet. When the calibrator detects the buffer model discrepancy, it requests the prediction engine to reset its buffer status (③). In a similar manner, the calibrator can adjust the GC model. When the prediction accuracy gets lower, it also resets the interval distribution to remove the current, ineffective history (③).

In addition, the calibrator dynamically adjusts the overhead of buffer flush and GC based on the latency logs delivered from the latency monitor. It calculates average latency of request groups (e.g., without interference, with buffer flush, with GC). The prediction engine uses the average latency to add the overheads into EBT. The calibrator also detects an occurrence of buffer backpressure incurring periodic slowdowns, and feeds this information back to the prediction engine to increase EBT by the buffer flush overhead.

IV. Use Case

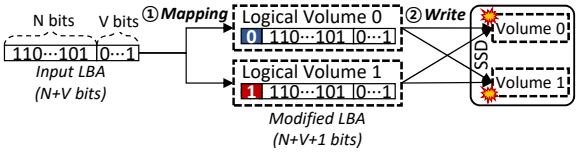
In this section, we describe two types of use cases: volume-aware logical volume manager and prediction-aware I/O scheduler. First, we provide a novel logical volume manager which divides an SSD into multiple logical volumes without any interference among the logical volumes (See §IV-A). We also propose a prediction-aware I/O scheduler using SSDcheck's prediction engine to improve the overall throughput and reduce tail latency for better QoS (See §IV-B).

A. Volume-Aware Logical Volume Manager

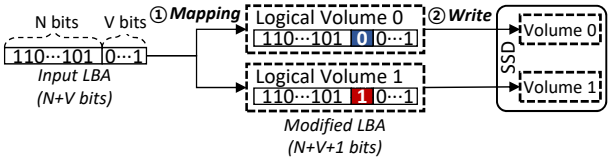
SSDs have become indispensable devices for cloud services because of their higher throughput, lower latency, and better cost competitiveness over HDDs. To leverage the high internal parallelism of SSDs, cloud storage systems colocate multiple workloads on the same SSD. Unfortunately, these multi-tenant settings further exacerbate the tail latency problem and performance degradation of SSDs [59]. For example, when read-intensive and write-intensive workloads run on the same SSD, excessive buffer flush and GC caused by the write-intensive workloads make the read-intensive workloads slow. Since most SSDs in the multi-tenant settings suffer from the performance degradation as well as the long tail latency, minimizing the interference among multiple tenants is important in the cloud storage systems.

Fortunately, we can achieve performance isolation among multiple tenants colocated in the same SSD by exploiting internal volumes in the SSD. SSDcheck identifies and provides the *volume indices* which determine the corresponding internal volumes for each I/O request (See §III-B1, §III-B2). Based on the information of internal volumes, we can split an SSD into multiple logical volumes and allot them to each tenant to minimize the interference.

Fig. 9 shows the examples of both a conventional partitioning scheme (Linear-LVM) and the proposed scheme. Since the conventional scheme is oblivious of the internal volumes



(a) Typical logical volume management (Linear-LVM)



(b) Proposed logical volume management (VA-LVM)

Fig. 9: Types of logical volume manager (LVM). (a) shows the typical example of LVM (Linear-LVM) which generates contention on each internal volume. (b) shows the proposed volume manager (VA-LVM). Each request is mapped into a corresponding internal volume without interference.

in SSDs, contention occurs in each internal volume, which significantly degrades the overall performance (Fig. 9a).

To minimize the contention, we propose volume-aware logical volume manager (VA-LVM), which splits an SSD into multiple logical volumes while minimizing the interference between the logical volumes (Fig. 9b). In this example, an SSD consists of two internal volumes and its volume index is V. Here, VA-LVM divides the SSD into two logical volumes and exposes them to the host system. When the host sends an I/O request to a corresponding logical volume, VA-LVM modifies an input LBA (N+V bits) based on the ID of each logical volume (e.g., 0 and 1). As shown in Fig. 9b, VA-LVM concatenates first N bits of the input LBA, one bit from the volume ID, and the last V bits of the input LBA. Finally, VA-LVM sends the I/O request to the SSD with the modified LBA (N+V+1 bits). Since the V^{th} bit value is fixed on each logical volume, I/O requests in different logical volumes do not interfere with each other.

We implement VA-LVM by using a device mapper which typically provides virtual volume management for underlying block devices in the linux kernel. We implement the *map* function to properly modify the input LBA following the logical volume ID.

B. Prediction-aware Scheduling

We propose prediction-aware scheduler (PAS), a novel I/O scheduler being aware of SSDcheck. PAS uses the latency prediction from SSDcheck to schedule I/O requests and has two different versions for traditional systems (*SSD-only PAS*) and emerging systems with a non-volatile memory (*Hybrid PAS*).

SSD-only PAS. SSD-only PAS reduces read tail latency and increases throughput for the systems using SSDs. Fig. 10 describes the key idea of SSD-only PAS where an SSD has two queus for writes (WQ) and reads (RQ). SSD-only PAS reduces the read tail latency due to write buffer flush by avoiding the overlap between reads and buffer flush. Suppose a scheduler

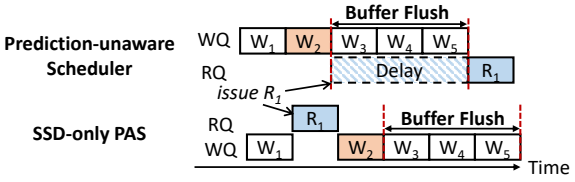


Fig. 10: How SSD-only PAS hides the buffer flush overheads. W_x and R_x represent access time of write and read requests, respectively.

has requests in order of W_1-W_2, R_1, and W_3-W_5, and W_2 is going to incur the flush. If the scheduler is unaware of the flush (i.e., *Prediction-unaware Scheduler*), it will schedule read request R_1 after write request W_2. Because the flush induced by W_2 occupies the NAND chips, R_1 should wait until the flush finishes (*Delay*) while another write buffer (i.e., *back*-type in III-B3) serves suceeding W_3-W_5. SSD-only PAS knows that R_1 will be slow due to the flush with the help of SSDcheck, so it schedules R_1 before W_2. In this way, SSD-only PAS can reduce the tail latency and increase the throughput by handling latency-critical read requests first in case of possible flush.

SSD-only PAS processes requests in the scheduler queue as follows. If all requests are read or write, it issues the oldest request. Otherwise, SSD-only PAS gets a latency prediction from SSDcheck for the oldest read based on the original order. Then, SSD-only PAS uses the specified threshold to classify the read request into *NL* or *HL*. If SSD-only PAS classifies read request into HL, then it is issued first regardless of the original order. In other cases, SSD-only PAS issues the older request. By prioritizing read requests over write requests, SSD-only PAS can hide the buffer flush overhead.

Hybrid PAS. With excessive writes, an SSD becomes more irregular due to frequent buffer flush and GC. In this case, SSD-only PAS shows marginal enhancement due to few possible re-orderings. To reduce this overhead recent storage systems [2]–[4] have adopted a non-volatile memory (NVM) to absorb the writes in the fast memory temporary. However, these systems obliviously send all writes into the NVM, so the small-capacity NVM becomes full shortly and flushes the contents into an SSD. Thus, they still suffer from irregularity.

We propose Hybrid PAS for the systems with SSDs and NVMs to provide consistent write performance using SSD-check. The key idea of Hybrid PAS is *selective delivery* that forwards the write requests based on the internal status of the SSD. First, Hybrid PAS asks SSDcheck for the latency prediction of an incoming write. Next, Hybrid PAS classifies the write into *NL* or *HL* based on the specified threshold. Hybrid PAS sends the *HL* request to the NVM. For the NL request, it randomly selects the destination depending on the specified buffer weight *W* (0-100%); that is, the NVM and the SSD handle *W*% and (100 - *W*)% of *NL* requests, respectively. Lastly, the contents in the NVM are periodically flushed to the SSD by a background thread. In this way, Hybrid PAS provides consistent performance by reducing the NVM burden from the *NL* writes, which account for the most write requests.

We implement both versions of PAS as linux I/O scheduling modules. Currently, a user specifies which version to use, but the automatic runtime adaptation could be done easily by monitoring I/O requests. When the strict order is necessary (e.g, barrier), PAS enforces the request order by recording it.

TABLE I: Extracted internal features in various SSDs.

Vendor	SSD	# of Internal Volumes (Idx)	Write Buffer		
			Size	Type	Flush
W	A	1 (None)	248KB	back	Full
X	B	1 (None)	248KB	back	Full
Y	C	1 (None)	256KB	back	Full
Z	D	2 (17)	128KB	back	Full
	E	4 (17, 18)	128KB	back	Full
	F	1 (None)	128KB	fore	Full & Read†
	G	1 (None)	128KB	fore	Full & Read†

† A read request explicitly triggers buffer flush.

TABLE II: Characteristics of various real workloads.

Trace Name (Abbr.)	# of requests	Writes	Random
TPCE (TPCE)	1.3M	92.4%	99.9%
Homes (Homes)	2.0M	90.4%	53.8%
Web (Web)	2.0M	91.5%	14.8%
Exchange (Exch)	7.6M	9.4%	99.8%
LiveMapsBackEnd (Live)	3.6M	22.2%	50.5%
BuildServer (Build)	0.6M	53.9%	85.6%

V. Evaluation

This section describes our evaluation environment (See §V-A), prediction accuracy of our proposed module (See §V-B), and potential benefits of our engine. (See §V-C, §V-D)

A. Experimental Setup

We implemented SSDcheck and the two example use cases in the real system to evaluate the prediction accuracy and the performance improvement, respectively. We used a Dell PowerEdge R730 server with two Intel Xeon E5-2640 v3 and seven SSDs selected from four major vendors (listed in Table I) with the linux kernel version 3.19 patch.

SSDcheck implementation. We implemented SSDcheck in a device driver communicating with MegaRAID SAS device driver to monitor both I/O submission and completion. We provide new interfaces for the host system to communicate with our prediction engine. The host asks the driver via the interfaces to check whether slowdown might occur or not. The interfaces are designed to support both user-level (e.g., ioctl) and kernel-level (e.g., exported functions) accesses.

Target SSDs. We choose seven SSDs from four major vendors. Table I shows the target SSDs for our evaluation and their internal features are extracted from *feature extraction* (See §III-B). We observe that the internal features of each SSD have different configurations. For each experiment, we measure the performance in the steady state to capture the real-world behaviors following the SNIA performance test standard [60]. We first purge the entire SSD using TRIM, then do pre-condition it through dummy random write requests. Note that SSDcheck without pre-condition performs better than with it because an SSD rarely calls GC.

Workloads. Table II shows the characteristics of workloads used to evaluate SSDcheck and the use cases. We use various

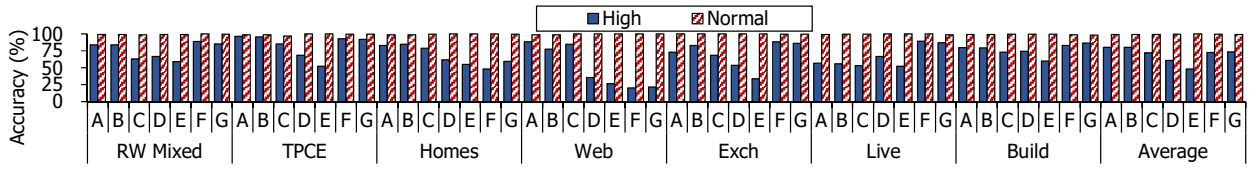


Fig. 11: The prediction accuracy of SSDcheck. We run six real-world and one synthetic workloads on seven commodity SSDs.

TABLE III: The latency distribution of *Web* on SSD A.

	Latency		
	<250 us	**<3500 us**	**<10 ms**
Read	99.12%	0.87%	0.01%
Write	98.43%	1.53%	0.04%

real block I/O traces from SNIA IOTTA Repository [61] and categorize the workloads into two groups: write-intensive (i.e., *TPCE*, *Homes*, *Web*) and read-intensive (i.e., *Exch*, *Live*, *Build*). We calculate a ratio between sequential (adjacent) and random accesses, and the access patterns of the workloads in each group show a different degree of randomness.

B. SSDcheck

We first analyze the latency distribution of each workload. Table III shows that the latency distribution of Web workload on SSD A. Other pairs of SSDs and workloads show similar behaviors, so we only show this case. We set the latency threshold to 250 us to distinguish NL requests from HL requests. The *NL* requests account for 99.12% of the write requests and 98.43% of the read requests.

To evaluate an accuracy of SSDcheck's prediction engine, we use modified fio to replay the workloads (listed in Table II). The fio asks SSDcheck for a predicted latency before issuing a request into SSDs. We calculate the prediction accuracy of *NL* and *HL* requests by comparing the measured latency with the corresponding predicted latency.

Fig. 11 shows the prediction accuracy of *NL* and *HL* requests. In additional to the six workloads at Table II, we also run a randomly generated trace (RW Mixed) to evaluate SSDcheck under the extreme read/write requests mixed situation. In all workloads, the prediction accuracy of *HL* requests is on average 80.0%, 79.8%, 72.3%, 61.1%, 48.4%, 72.7%, 73.7% for SSD A–G. The allocation volume model substantially increases SSDcheck's accuracy on SSD D and E compared to extremely low accuracy of SSDcheck without the model. Unfortunately, the secondary features make their accuracy slightly lower than SSDcheck on other SSDs (See VI).

Calibration engine, however, quickly resolves the discrepancy; therefore, SSDcheck achieves reasonable accuracy. The accuracy of *NL* requests is on average 99.0%, 99.0%, 99.0%, 99.7%, 99.7%, 99.5%, 99.1% for SSD A–G in all workloads. SSDcheck shows high accuracy of both *NL* and *HL* requests in the whole evaluation because of its accurate performance model. Note that SSDcheck's prediction overheads are negligible (i.e., a few nanoseconds), so it can be used for many effective use cases (e.g., scheduler) without any performance degradation.

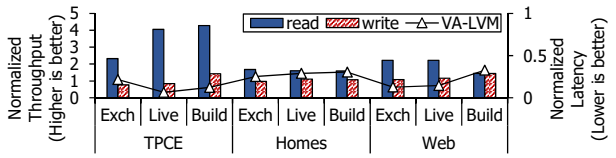


Fig. 12: The throughput and the 99.5th latency of VA-LVM for all workload combinations on SSD D. Both throughput (left-axis) and latency (right-axis) are normalized to Linear-LVM.

C. Use Case 1: Volume-aware Logical Volume Manager

We compare a volume-aware logical volume manager (VA-LVM) with a typical volume manager (Linear-LVM) which maps a continuous range of an SSD into a logical volume. We configure two logical volumes for both VA-LVM and Linear-LVM as explained in Fig. 9. To create a multi-tenant environment, we run read-intensive (*Exch*, *Live*, *Build*) and write-intensive (*TPCE*, *Homes*, *Web*) workloads concurrently. We run all nine combinations of a read- and write-intensive workloads on both VA-LVM and Linear-LVM, and measure the performance impact of the proposed scheme. We evaluate both LVM on SSD D, which has two internal volumes.

We compare VA-LVM with Linear-LVM on both the throughput and the tail latency. Fig. 12 shows the results of the experiments. VA-LVM improves the throughput of read-intensive workloads by up to 4.29× (or 2.38× on average), while retaining the throughput of write-intensive workloads. In addition to the throughput improvement, VA-LVM also reduces the tail latency at 99.5th percentile down to 6.53% (or 20.3% on average) for read-intensive workloads.

The huge throughput improvement and tail latency reduction originate from a performance isolation feature of VA-LVM, which eliminates interference between the logical volumes. In Linear-LVM, which is oblivious of the internal volumes in SSDs, high latencies caused by the write-intensive workload significantly degrade the performance of the read-intensive workload. On the other hand, VA-LVM successfully removes the interference among logical volumes; therefore, the read-intensive workload can be more efficiently executed.

D. Use Case 2: Prediction-aware Scheduling

In this experiment, we present how PAS reduces the tail latency and improves throughput. First, we evaluate SSD-only PAS and show the effectiveness of the buffer flush latency hiding. Next, we demonstrate how Hybrid PAS leverages an NVM for write-intensive workloads whose buffer flush latency cannot be hidden by SSD-only PAS.

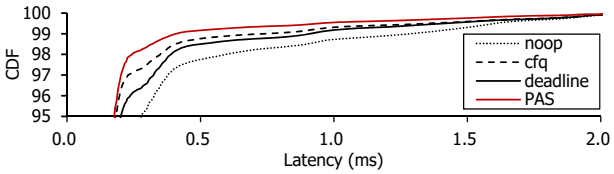


Fig. 13: The tail latency distribution of Build on SSD G.

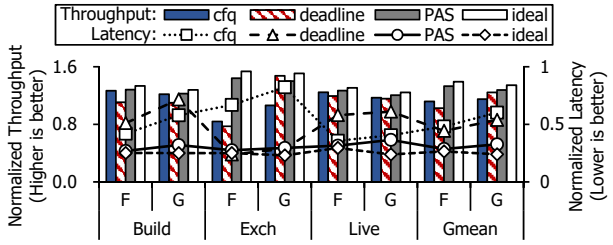


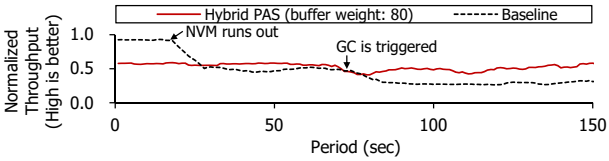
Fig. 14: The tail latency and the throughput of Build, Exch and Live on SSD F and G. Both the tail latency and the throughputs are normalized to `noop`.

SSD-only PAS. To evaluate the effectiveness of SSD-only PAS, we use three workloads which consist of read and write requests randomly: Build, Exch and Live. For baseline schedulers, we use `noop`, `deadline`, and `cfq`, the mainstream I/O schedulers in the linux kernel. `noop` serves I/O requests in FIFO-order, and `cfq` and `deadline` adopt their own distinct priority policies for read and write requests. As representative examples, we show the results on SSD F and G whose write buffer flush overhead is high to highlight the flush-induced tail latency.

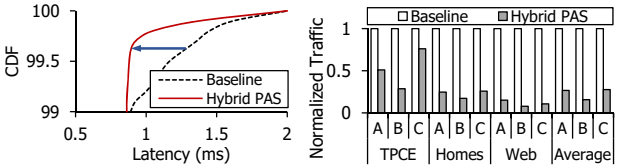
Here, we focus on the tail latency distribution for SSD G on Build in Fig. 13 since other cases show a similar trend. `noop` shows the longest tail latency since it processes the requests in strict order. `cfq` and `deadline` show the shorter tail latency than `noop`, because of their own priority policy. However, since they are unaware of buffer flush prediction, their chance to hide long latency is small and many delayed reads form a long tail. SSD-only PAS shows the shortest tail latency, thanks to heavily reduced the number of delayed reads.

Fig. 14 presents the tail latency reduction and the throughput improvement for the representative workload and SSD pairs. Because the buffer flush latency appears at the different percentile of the distribution among the different SSD and workload pairs, we show the distinct points to see the improvements clearly. The measurement points are 97.6% (Build-SSD F), 99.0% (Build-SSD G), 94.0% (Exch-SSD F), 97.2% (Exch-SSD G), 97.6% (Live-SSD F), and 97.9% (Live-SSD G). In all workloads, compared with `noop`, SSD-only PAS reduces tail latency by 71% and 67% for SSD F and G on average. Since SSDcheck predicts latency accurately, SSD-only PAS shows short tail latency.

SSD-only PAS also improves the throughput by efficiently handling reads. Compared with `noop`, SSD-only PAS improves the throughput by 32% and 27% for SSD F and G on average. On the other hand, `cfq` and `deadline` suffer from longer tail latency and more inconstant throughput than



(a) Performance timeline of the baseline and Hybrid PAS



(b) Tail latency reduction. (c) Pressure on the NVM.

Fig. 15: Performance analysis of *baseline* and Hybrid PAS (buffer weight: 80). (a) shows the throughput variations. (b) shows the latency distribution of *Web* on SSD C. (c) shows an amount of pressures on the NVM, normalized to the baseline.

SSD-only PAS.

To present the maximum potential of SSD-only PAS, we calculate the expected bandwidth and latency of ideal SSD-only PAS, whose prediction accuracy is 100% (`ideal` in Fig. 14). PAS shows 7.9% and 36% higher latency and 4.4% and 4.9% lower throughput for SSD F and G compared to ideal. These are equal to the cost of misprediction by PAS.

Hybrid PAS. We compare Hybrid PAS with *baseline*, a typical I/O scheduler for the multi-tier storage systems. *Baseline* always forwards incoming write requests into an NVM to hide the write latency of an SSD, and the contents in the NVM are periodically flushed to the SSD by a background thread. If the NVM is full, backpressure is exposed and all write requests need to access the irregular SSD. For this experiment, we use SSD A-C which show more irregular write behaviors.

Fig. 15a shows the performance of *baseline* and Hybrid PAS in a timeline for a synthetic write-intensive benchmark on SSD C. Initially, *baseline* shows higher throughput than Hybrid PAS because the NVM should handle all writes. However, the NVM runs out quickly, so the baseline throughput is severely degraded (15 sec). *Baseline* throughput is further degraded (76 sec) because of GC overheads are exposed to the host system. In contrast, Hybrid PAS shows persistent performance thanks to reduced NVM backpressure and extremely little HL writes. On average, Hybrid PAS outperforms *baseline* throughput by up to 2.1x in the steady state.

In addition, Hybrid PAS reduces long tail latency. Fig. 15b shows the latency distribution for SSD C on workload *Web*. *Baseline* cannot avoid long tail latency after the NVM runs out. Hybrid PAS, however, can reduce long tail latency thanks to the accurate prediction of SSDcheck. Overall, Hybrid PAS reduces the tail latency by 1.46x at 99.7^{th} percentile.

Since an NVM is a limited resource, we should efficiently use the NVM to avoid performance degradation due to the capacity shortage [62]. However, all writes in *baseline* go to the NVM and cause the significant pressure on it. By

comparison, Hybrid PAS selectively issues only some writes to the NVM, so it can effectively reduce the pressure. Fig. 15c shows the normalized write traffic that represents the NVM pressure. Hybrid PAS reduces the pressure on the NVM on average 16.7%, 27.8%, 28.7% for SSD A–C, respectively, for real-world write-intensive workloads.

VI. Discussion

Secondary Features. As we do not model all internal features of SSDs, our prediction engine mispredicts some HL requests. We expect more feature extractions and performance models (e.g., wear-leveling, ECC, SLC caching) can improve the accuracy. For example, in MLC-based SSDs, some vendors adopt an optimization technique, SLC caching, which uses a portion of MLC cells as SLC cells to improve write performance. Incoming requests are first stored in a SLC region, and are flushed to a MLC region later; therefore, SLC caching can hide long write latency of MLC cells. If we can find the size of the SLC region and conditions of when SSDs flush data from SLC to MLC region, we can further improve the model correctness. We plan to add these models in the future work.

NVM-based SSDs. We expect SSDcheck can be applied to NVM-based SSDs consisting of the internal write buffer and the backend storage medium (e.g., 3D Xpoint, PRAM, RRAM). Those SSDs usually have the internal write buffer and often rely on GC to provide the consistent high throughput. Since SSDcheck is a software-based approach, it can be extended easily to cover medium-specific designs and optimizations by adding new performance models.

VII. Related Work

SSD performance modeling. One branch of SSD modeling is garbage collection modeling [20], [28]–[32]. Using Markov chain model and mean-field approach, [29] builds a GC model and proposes a randomized greedy algorithm. Their purposes are efficient GC algorithms, so they are orthogonal to our work.

Others [35], [36] provide analytical performance models for SSDs. For given per-workload characteristics (e.g., randomness, read write ratio), their models give quantitative performance numbers (e.g., overall throughput). However, their model predicts the performance in coarse granularity (i.e., per-workload) while we predict the latency much finer granularity (i.e., per-access) to predict the irregular behaviors.

The closest research is internal feature extraction work [33], [34], [63] that uses diagnosis code snippets to extract the internal features (e.g., chunk size [33], operation units [34], write buffer [63]). However, their goals are to find the internal features to set the system parameters (e.g., write granularity), and they do not provide any performance model and richful use cases. In contrast, we propose a performance model and capture the irregular behaviors of an SSD.

Device-level modifications. Many studies modify SSDs to address their irregular behaviors. One group of research relieves the irregularity by modifying SSD's firmware [10], [37]–[40]. HIOS [10] extends host-interface layer (HIL) to schedule host I/Os being aware of underlying layers' status. Preemptive GCs [38], [39] make GC-related operations preemptible by incoming I/O requests to reduce the GC overhead.

The other group makes an SSD more predictable by opening a new interface [41]–[46]. Multi-stream [43] provides "stream" primitives and guarantees that different applications can run on a single SSD without huge interference. Fstream [44] is a file system that takes advantage of this. [45] provides an explicit GC triggering command, so the host can know when GC is running. To this end, innovative movements offloading most of FTL's functions to the host are emerging [23], [47], [48]. SDF [47] moves page allocation and GC to the host for more available space, higher throughput, and predictable latency. However, all these approaches require changes on the structure and/or the interfaces of SSDs which the device vendors are not willing to do.

Performance isolation. Some work proposes a virtual SSD concept to run various workloads concurrently [59], [64]–[67]. The NVMe community suggests *I/O determinism* [66], [67] that splits an NVMe SSD into user-exposed virtual volumes. Flashbox [59] exposes multiple virtual SSDs and provides a new API to isolate the performance among tasks. However, they are limited to FTL modifications while VA-LVM exploits the built-in features of commodity SSDs.

System software. Many efforts are made to exploit SSD's characteristics for I/O scheduler [27], [68]–[70], RAID [6], [8], [71], [72], file systems [73]–[75], and storage systems [3], [4]. They are orthogonal to our work and can use SSDcheck for better performance. For example, FIOS [27] is an I/O scheduler for fairness among tasks and assumes read requests after write requests are always slow. By mitigating such strong assumption with the help of SSDcheck, FIOS can improve the responsiveness.

VIII. Conclusion

We propose SSDcheck, a novel methodology to accurately predict the timing of future high latency without hardware modifications. SSDcheck extracts the device-specific features and provides a performance model of black-box SSDs by using the extracted features. SSDcheck achieves a high prediction accuracy and performance improvement in the two use cases.

Acknowledgment

This work was partly supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (MSIT) (NRF-2015M3C4A7065647, NRF-2017R1A2B3011038), Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. R0190-15-2012, No. 1711073912), and Creative Pioneering Researchers Program through Seoul National University. We also appreciate the support from Automation and Systems Research Institute (ASRI), Inter-university Semiconductor Research Center (ISRC), and Neural Processing Research Center (NPRC) at Seoul National University.

REFERENCES

[1] J. Gray and B. Fitzgerald, "Flash disk opportunity for server applications," *ACM Queue, Vol.6(4), July/August 2008*.

[2] "Pure storage," https://www.purestorage.com/.

[3] J. Colgrove, J. D. Davis, J. Hayes, E. L. Miller, C. Sandvig, R. Sears, A. Tamches, N. Vachharajani, and F. Wang, "Purity: Building fast, highly-available enterprise flash storage from commodity components," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*.

[4] "Netapp solidfire," https://www.solidfire.com/.

[5] "Dell emc xtremio," https://www.dellemc.com/en-us/storage/xtremio-all-flash.htm.

[6] M. Hao, G. Soundararajan, D. Kenchammana-Hosekote, A. A. Chien, and H. S. Gunawi, "The tail at store: A revelation from millions of hours of disk and SSD deployments," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*.

[7] J. He, D. Nguyen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Reducing file system tail latencies with chopper." in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*.

[8] D. Skourtis, D. Achlioptas, N. Watkins, C. Maltzahn, and S. A. Brandt, "Flash on rails: Consistent flash performance through redundancy." in *Proceedings of the 2014 conference on USENIX Annual technical conference (ATC '14)*.

[9] S. Yang, T. Harter, N. Agrawal, S. S. Kowsalya, A. Krishnamurthy, S. Al-Kiswany, R. T. Kaushik, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Split-level i/o scheduling," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*.

[10] M. Jung, W. Choi, S. Srikantaiah, J. Yoo, and M. T. Kandemir, "Hios: a host interface i/o scheduler for solid state disks," in *Proceeding of the 41st annual international symposium on Computer architecuture (ISCA '14)*.

[11] M. Jung and M. Kandemir, "Revisiting widely held SSD expectations and rethinking system-level implications," in *Proceedings of the ACM SIGMETRICS/international conference on Measurement and modeling of computer systems (SIGMETRICS '13)*.

[12] L. M. Grupp, J. D. Davis, and S. Swanson, "The harey tortoise: Managing heterogeneous write performance in ssds." in *Proceedings of the 2008 conference on USENIX Annual technical conference (ATC '13)*.

[13] E. K. Lee and R. H. Katz, "An analytic performance model of disk arrays," in *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems (SIGMETRICS '93)*.

[14] S. Chen and D. Towsley, "A performance evaluation of raid architectures," *IEEE Transactions on computers*, vol. 45, 1996.

[15] J. B. Chen and B. N. Bershad, "The impact of operating system structure on memory system performance," in *Proceedings of the fourteenth ACM symposium on Operating systems principles (SOSP '93)*.

[16] M. Y. Kim and A. N. Tantawi, "Asynchronous disk interleaving: Approximating access delays," *IEEE Transactions on Computers*, vol. 40, 1991.

[17] M. Uysal, G. A. Alvarez, and A. Merchant, "A modular, analytical throughput model for modern disk arrays," in *Proceedings. Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '17)*.

[18] "Zynq-7000 xc7z045," https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.

[19] S.-W. Lee, B. Moon, and C. Park, "Advances in flash memory ssd technology for enterprise database applications," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data (SIGMOD '09)*. ACM, 2009.

[20] P. Desnoyers, "Analytic modeling of ssd write performance," *Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR '12)*.

[21] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for ssd performance." in *USENIX Annual Technical Conference (ATC '08)*.

[22] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim, "A reconfigurable ftl (flash translation layer) architecture for nand flash-based applications," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, 2008.

[23] M. Bjørling, J. Gonzalez, and P. Bonnet, "LightNVM: the linux open-channel SSD subsystem," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*.

[24] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proceedings of the ACM SIGMETRICS/international conference on Measurement and modeling of computer systems (SIGMETRICS '09)*.

[25] C. Dirik and B. Jacob, "The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization," in *Proceedings of the 36th annual international symposium on Computer architecture (ISCA '09)*.

[26] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren, "Exploring and exploiting the multilevel parallelism inside ssds for improved performance and endurance," *IEEE Transactions on Computers*, vol. 62, 2013.

[27] S. Park and K. Shen, "Fios: a fair, efficient flash i/o scheduler." in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)*.

[28] B. Van Houdt, "A mean field model for a class of garbage collection algorithms in flash-based solid state drives," in *ACM SIGMETRICS Performance Evaluation Review - Performance evaluation review Volume 41 Issue 1, June 2013*.

[29] Y. Li, P. P. C. Lee, and J. C. S. Lui, "Stochastic modeling of large-scale solid-state storage systems: Analysis, design tradeoffs and optimization," in *Proceedings of the ACM SIGMETRICS/international conference on Measurement and modeling of computer systems (SIGMETRICS '13)*.

[30] Y. Li, P. P. Lee, J. Lui, and Y. Xu, "Impact of data locality on garbage collection in ssds: a general analytical study," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE '15)*.

[31] S. Boboila and P. Desnoyers, "Performance models of flash-based solid-state drives for real workloads," in *IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST '11)*.

[32] W. Bux and I. Iliadis, "Performance of greedy garbage collection in flash-based solid-state drives," *Performance Evaluation 2010*.

[33] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*.

[34] J. Kim, S. Seo, D. Jung, J.-S. Kim, and J. Huh, "Parameter-aware I/O management for solid state disks (SSDs)," *IEEE Transactions on Computers, Vol.61(5), May 2012*.

[35] H. H. Huang, S. Li, A. Szalay, and A. Terzis, "Performance modeling and analysis of flash-based storage devices," in *IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST '11)*.

[36] S. Li and H. H. Huang, "Black-box performance modeling for solid-state drives," in *IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS '10)*.

[37] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi, "Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds," *ACM Transactions on Storage (TOS) - Special Issue on FAST 2017 and Regular Papers TOS Volume 13 Issue 3, October 2017*.

[38] G. Wu and X. He, "Reducing ssd read latency via nand flash program and erase suspension." in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)*.

[39] J. Lee, Y. Kim, G. M. Shipman, S. Oral, F. Wang, and J. Kim, "A semi-preemptive garbage collector for solid state drives," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '11)*.

[40] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "De-indirection for flash-based SSDs with nameless writes," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)*.

[41] F. Shu and N. Obr, "Data set management commands proposal for ata8-acs2," *Management 2007*.

[42] M. Saxena and M. M. Swift, "Flashvm: Virtual memory management on flash." in *Proceedings of the 2010 conference on USENIX Annual technical conference (ATC '10)*.

[43] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The multi-streamed solid-state drive," in *HotStorage '14*.

[44] E. Rho, K. Joshi, S.-U. Shin, N. J. Shetty, J.-Y. Hwang, S. Cho, D. D. Lee, and J. Jeong, "Fstream: managing flash streams in the file system," in *16th USENIX Conference on File and Storage Technologies (FAST '18)*.

[45] M. Jung, R. Prabhakar, and M. T. Kandemir, "Taking garbage collection overheads off the critical path in SSDs," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '12)*.

[46] S. S. Hahn, S. Lee, and J. Kim, "Sos: Software-based out-of-order scheduling for high-performance nand flash-based ssds," in *IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST '13)*.

[47] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang, "Sdf: software-defined flash for web-scale internet storage systems," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems (ASPLOS '14)*.

[48] S. Lee, M. Liu, S. W. Jun, S. Xu, J. Kim, and A. Arvind, "Application-managed flash." in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*.

[49] D. Ma, J. Feng, and G. Li, "Lazyftl: a page-level flash translation layer optimized for nand flash memory," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD '11)*.

[50] A. Gupta, Y. Kim, and B. Urgaonkar, "Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings," in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS '09)*.

[51] L.-P. Chang and T.-W. Kuo, "Efficient management for large-scale flash-memory storage systems with resource conservation," *ACM Transactions on Storage (TOS) Volume 1 Issue 4, Nov. 2005*.

[52] Y. Cai, Y. Luo, S. Ghose, and O. Mutlu, "Read disturb errors in mlc nand flash memory: Characterization, mitigation, and recovery," in *45th Annual IEEE/IFIP International Conference on Systems and Networks (DSN '15)*.

[53] P. Huang, P. Subedi, X. He, S. He, and K. Zhou, "Flexecc: Partially relaxing ecc of mlc ssd for better cache performance." in *Proceedings of the 2014 conference on USENIX Annual technical conference (ATC '14)*.

[54] C. Zambelli, M. Indaco, M. Fabiano, S. Di Carlo, P. Prinetto, P. Olivo, and D. Bertozzi, "A cross-layer approach for new reliability-performance trade-offs in mlc nand flash memories," in *Proceedings of the conference on Design, automation and test in Europe (DATE '12)*.

[55] K. Zhao, W. Zhao, H. Sun, T. Zhang, X. Zhang, and N. Zheng, "Ldpc-in-ssd: making advanced error correction codes work effectively in solid state drives." in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST '13)*.

[56] "Fio benchmark," https://github.com/axboe/fio.

[57] G. Wu and X. He, "Delta-ftl: improving SSD lifetime via exploiting content locality," in *Proceedings of the 7th ACM european conference on Computer Systems (EuroSys '12)*.

[58] A. Tavakkol, M. Arjomand, and H. Sarbazi-Azad, "Design for scalability in enterprise SSDs," in *Proceedings of the 23rd international conference on Parallel architectures and compilation (PACT '14)*.

[59] J. Huang, A. Badam, L. Caulfield, S. Nath, S. Sengupta, B. Sharma, and M. K. Qureshi, "Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds." in *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*.

[60] E. Ho, "Snia solid state storage performance test specification," *SNIA*.

[61] "SNIA IOTTA Repository," http://iotta.snia.org/.

[62] Y. Oh, J. Choi, D. Lee, and S. H. Noh, "Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST '14)*.

[63] J. Kim, J. Kim, P. Park, J. Kim, and J. Kim, "Ssd performance modeling using bottleneck analysis," *IEEE Computer Architecture Letters*, vol. 17, 2018.

[64] X. Song, J. Yang, and H. Chen, "Architecting flash-based solid-state drive for high-performance i/o virtualization," *IEEE Computer Architecture Letters*, vol. 13, 2014.

[65] J. Kim, D. Lee, and S. H. Noh, "Towards slo complying ssds through ops isolation." in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*.

[66] J. M. David Allen, "The evolution and future of nvme," *Webminar, Jan 2018*.

[67] A. H. Chris Petersen, "Solving latency challenges with nvm express ssds at scale," *Flash Memory Summit, Aug 2017*.

[68] K. Shen and S. Park, "Flashfq: A fair queueing i/o scheduler for flash-based ssds." in *Proceedings of the 2013 conference on USENIX Annual technical conference (ATC '13)*.

[69] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Disk schedulers for solid state drivers," in *Proceedings of the seventh ACM international conference on Embedded software (EMSOFT '09)*.

[70] H. Wang, P. Huang, S. He, K. Zhou, C. Li, and X. He, "A novel i/o scheduler for ssd with improved performance and lifetime," in *IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST '13)*.

[71] J. Lee, Y. Kim, J. Kim, and G. M. Shipman, "Synchronous i/o scheduling of independent write caches for an array of ssds," *IEEE Computer Architecture Letters (Volume: 14, Issue: 1, Jan.-June 1 2015)*.

[72] Y. Kim, J. Lee, S. Oral, D. A. Dillow, F. Wang, and G. M. Shipman, "Coordinating garbage collectionfor arrays of solid-state drives," *IEEE Transactions on Computers, Vol.63(4), April 2014*.

[73] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*.

[74] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "SFS: random write considered harmful in solid state drives," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)*.

[75] S. Qiu and A. N. Reddy, "Nvmfs: A hybrid file system for improving random write in nand-flash ssd," in *IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST '13)*.