The Unwritten Contract of Solid State Drives

Jun He Sudarsun Kannan Andrea C. Arpaci-Dusseau Remzi H. Arpaci-Dusseau

Department of Computer Sciences, University of Wisconsin–Madison

Abstract

We perform a detailed vertical analysis of application performance atop a range of modern file systems and SSD FTLs. We formalize the "unwritten contract" that clients of SSDs should follow to obtain high performance, and conduct our analysis to uncover application and file system designs that violate the contract. Our analysis, which utilizes a highly detailed SSD simulation underneath traces taken from real workloads and file systems, provides insight into how to better construct applications, file systems, and FTLs to realize robust and sustainable performance.

1. Introduction

In-depth performance analysis lies at the heart of systems research. Over many years, careful and detailed analysis of memory systems [26, 81], file systems [36, 50, 51, 66, 84, 87], parallel applications [91], operating system kernel structure [35], and many other aspects of systems [25, 29, 37, 41, 65] has yielded critical, and often surprising, insights into systems design and implementation.

However, perhaps due to the rapid evolution of storage systems in recent years, there exists a large and important gap in our understanding of I/O performance across the storage stack. New data-intensive applications, such as LSM-based (Log-Structured Merge-tree) key-value stores, are increasingly common [6, 14]; new file systems, such as F2FS [62], have been created for an emerging class of flash-based Solid State Drives (SSDs); finally, the devices themselves are rapidly evolving, with aggressive flash-based translation layers (FTLs) consisting of a wide range of optimizations. How well do these applications work on these modern file systems, when running on the most recent class of SSDs? What aspects of the current stack work well, and which do not?

The goal of our work is to perform a detailed vertical analysis of the application/file-system/SSD stack to answer the aforementioned questions. We frame our study around the file-system/SSD interface, as it is critical for achieving high performance. While SSDs provide the same interface as hard drives, how higher layers utilize said interface can greatly affect overall throughput and latency.

Our first contribution is to formalize the "unwritten contract" between file systems and SSDs, detailing how upper layers must treat SSDs to extract the highest instantaneous and long-term performance. Our work here is inspired by Schlosser and Ganger's unwritten contract for hard drives [82], which includes three rules that must be tacitly followed in order to achieve high performance on Hard Disk Drives (HDDs); similar rules have been suggested for SMR (Shingled Magnetic Recording) drives [46].

We present five rules that are critical for users of SSDs. First, to exploit the internal parallelism of SSDs, SSD clients should issue large requests or many outstanding requests (*Request Scale* rule). Second, to reduce translation-cache misses in FTLs, SSDs should be accessed with locality (*Locality* rule). Third, to reduce the cost of converting page-level to block-level mappings in hybrid-mapping FTLs, clients of SSDs should start writing at the aligned beginning of a block boundary and write sequentially (*Aligned Sequentiality* rule). Fourth, to reduce the cost of garbage collection, SSD clients should group writes by the likely death time of data (*Grouping By Death Time* rule). Fifth, to reduce the cost of wear-leveling, SSD clients should create data with similar lifetimes (*Uniform Data Lifetime* rule). The SSD rules are naturally more complex than their HDD counterparts, as SSD FTLs (in their various flavors) have more subtle performance properties due to features such as wear leveling [30] and garbage collection [31, 71].

We utilize this contract to study application and file system pairings atop a range of SSDs. Specifically, we study the performance of four applications – LevelDB (a key-value store), RocksDB (a LevelDB-based store optimized for SSDs), SQLite (a more traditional embedded database), and Varmail (an email server benchmark) – running atop a range of modern file systems – Linux ext4 [69], XFS [88], and the flash-friendly F2FS [62]. To perform the study and extract the necessary level of detail our analysis requires, we build *WiscSee*, an analysis tool, along with *WiscSim*, a detailed and extensively evaluated discrete-event SSD simulator that can model a range of page-mapped and hybrid FTL designs [48, 54, 57, 74]. We extract traces from each application/file-system pairing, and then, by applying said traces to *WiscSim*, study and understand details of system performance that previously were not well understood. *WiscSee* and *WiscSim* are available at `http://research.cs.wisc.edu/adsl/Software/wiscsee/`.

EuroSys '17, April 23 - 26, 2017, Belgrade, Serbia

ISBN 978-1-4503-4938-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/3064176.3064187

Our study yields numerous results regarding how well applications and file systems adhere to the SSD contract; some results are surprising whereas others confirm commonly-held beliefs. For each of the five contract rules, our general findings are as follows. For request scale, we find that log structure techniques in both applications and file systems generally increase the scale of writes, as desired to adhere to the contract; however, frequent barriers in both applications and file systems limit performance and some applications issue only a limited number of small read requests. We find that locality is most strongly impacted by the file system; specifically, locality is improved with aggressive space reuse, but harmed by poor log structuring practices and legacy HDD block-allocation policies. I/O alignment and sequentiality are not achieved as easily as expected, despite both application and file system log structuring. For death time, we find that although applications often appropriately separate data by death time, file systems and FTLs do not always maintain this separation. Finally, applications should ensure uniform data lifetimes since in-place-update file systems preserve the lifetime of application data.

We have learned several lessons from our study. First, log structuring is helpful for generating write requests at a high scale, but it is not a panacea and sometimes hurts performance (e.g., log-structured file systems fragment application data structures, producing workloads that incur higher overhead). Second, due to subtle interactions between workloads and devices, device-specific optimizations require detailed understanding: some classic HDD optimizations perform surprisingly well on SSDs while some SSD-optimized applications and file systems perform poorly (e.g., F2FS delays trimming data, which subsequently increases SSD space utilization, leading to higher garbage collection costs). Third, simple workload classifications (e.g., random vs. sequential writes) are orthogonal to important rules of the SSD unwritten contract (e.g., grouping by death time) and are therefore not useful; irrelevant workload classifications can lead to oversimplified myths about SSDs (e.g., random writes considered harmful [71]).

This paper is organized as follows. Section 2 introduces the background of SSDs. Section 3 describes the rules of the SSD unwritten contract. Section 4 presents the methodology of our analysis. In Section 5, we conduct vertical analysis of applications, file systems and FTLs to examine behaviors that violate the contract rules. Section 6 introduces related work. Section 7 concludes this paper.

2. Background

The most popular storage technology for SSDs is NAND flash. A flash chip consists of blocks, which are typically hundreds of KBs (e.g., 128 KB), or much larger (e.g., 4 MB) for large-capacity SSDs [16, 21]. A block consists of pages, which often range from 2 KB to 16 KB [9, 10, 17]. Single-Level Cell (SLC) flash, which stores a single bit in a memory element, usually has smaller page sizes, lower latency, better endurance and higher costs than Multi-Level Cell (MLC) flash, which stores multiple bits in a memory element [28].

Flash chips support three operations: read, erase, and program (or write). Reading and programming are usually performed at the granularity of a page, whereas erasing can only be done for an entire block; one can program pages only after erasing the whole block. Reading is often much faster than programming (e.g., eight times faster [16]), and erasing is the slowest, but can have higher in-chip bandwidth than programming (e.g., 83 MB/s for erase as compared to 9.7 MB/s for write [16]). A block is usually programmed from the low page to high page to avoid program disturbance, an effect that changes nearby bits unintentionally [10, 11, 28].

Modern SSDs use a controller to connect flash chips via channels, which are the major source of parallelism. The number of channels in modern SSDs can range from a few to dozens [5, 40, 42, 73]. The controller uses RAM to store its operational data, client data, and the mapping between host logical addresses and physical addresses.

To hide the complexity of SSD internals, the controller usually contains a piece of software called an FTL (Flash Translation Layer); the FTL provides the host with a simple block interface and manages all the operations on the flash chips. FTLs can employ vastly different designs [48, 54, 57, 63, 64, 67, 74, 75, 93, 95]. Although not explicitly stated, each FTL requires clients to follow a unique set of rules in order to achieve good performance. We call these implicit rules the *unwritten contract* of SSDs.

3. Unwritten Contract

Users of an SSD often read its *written contract*, which is a specification of its interfaces. Violation of the written contract will lead to failures; for example, incorrectly formatted commands will be rejected by the receiving storage device. In contrast, the *unwritten contract* [82] of an SSD is an implicit performance specification that stems from its internal architecture and design. An unwritten contract is not enforced but violations significantly impact performance.

SSDs have different performance characteristics from HDDs in part due to unpredictable background activities such as garbage collection and wear-leveling. On an SSD, an access pattern may have excellent performance at first, but degrade due to background activities [47, 62, 71, 85]. To reflect this distinction, we call these regimes *immediate performance* and *sustainable performance*. Immediate performance is the maximum performance achievable by a workload's I/O pattern. Sustainable performance is the performance that could be maintained by an SSD given this workload in the long term.

In this section, we summarize the rules of the unwritten contract of SSDs and their impact on immediate and sustainable performance.

3.1 Request Scale

Modern SSDs have multiple independent units, such as channels, that can work in parallel. To exploit this paral-

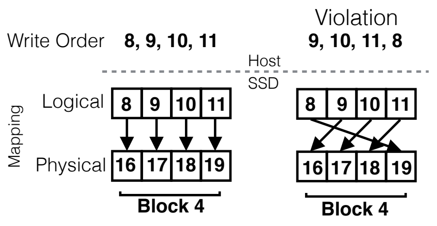


Figure 1: **Example of Aligned Sequentiality and a violation of it.** *Each block has four pages. Writes must be programmed on a flash block from low to high pages to avoid program disturbance. The page-level mapping on the left can be converted to a single block-level mapping: logical block 2 $\rightarrow$ physical block 4. The example on the right cannot be converted without re-arranging the data.*

lelism, one common technique when request sizes are large is to stripe each request into sub-requests and send them to different units [25, 34, 45, 53]. When request sizes are small, the FTL can distribute the requests to different units. To concurrently process multiple host requests, modern SSDs support Native Command Queuing (NCQ)[1] or similar features [12, 19]; a typical maximum queue depth of modern SATA SSDs is 32 requests [4, 34].

To capture the importance of exploiting internal parallelism in an SSD, the first rule of our unwritten contract is **Request Scale:** *SSD clients should issue large data requests or multiple concurrent requests*. A small request scale leads to low resource utilization and reduces immediate and sustainable performance [34, 53].

3.2 Locality

Because flash chips do not allow in-place updates, an FTL must maintain a dynamic mapping between logical[2] and physical pages. A natural choice is a page-level mapping, which maintains a one-to-one mapping between logical and physical pages. Unfortunately, such a mapping requires a large amount of RAM, which is scarce due to its high price, relatively high power consumption, and competing demands for mapping and data caching [48]. With a page size of 2 KB, a 512-GB SSD would require 2 GB of RAM.[3] Having larger pages, such as those ($\geq$ 4 KB) of popular MLC flash, will reduce the required space. However, SLC flash, which often has pages that are not larger than 2 KB, is still often used to cache bursty writes because of its lower latency [23]. The use of SLC flash increases the demand for RAM.

On-demand FTLs, which store mappings in flash and cache them in RAM, reduce the RAM needed for mappings. The mapping for a translation is loaded only when needed and may be evicted to make room for new translations. Locality is needed for such a translation cache to work; some FTLs exploit only temporal locality [48], while others exploit both temporal and spatial locality [54, 63, 64].

Thus the contract has a **Locality** rule: *SSD clients should access with locality*. Workloads without locality can incur a poor immediate performance because frequent cache misses lead to many translation-related reads and writes [48, 54]. Poor locality also impacts sustainable performance because data movement during garbage collection and wear-leveling requires translations and mapping updates.

Locality is not only valuable for reducing required RAM for translations, but also for other purposes. For example, all types of SSDs are sensitive to locality due to their data cache. In addition, for SSDs that arrange flash chips in a RAID-like fashion, writes with good locality are more likely to update the same stripe and the parity calculation can thus be batched and written concurrently [92], improving performance.

3.3 Aligned Sequentiality

Another choice for reducing memory requirements is hybrid mapping [57, 63, 64, 74], in which part of the address space is covered by page-level mappings and the rest by block-level mappings. Since one entry of a block-level map can cover much more space than a page-level mapping, the memory requirements are significantly reduced. For example, if 90% of a 512-GB SSD (128 KB block) is covered by block-level mapping, the hybrid FTL only needs 233 MB.[4] A hybrid FTL uses page-level mappings for new data and converts them to block-level mappings when it runs out of mapping cache. The cost of such conversions (also known as merges) depends on the existing page-level mapping, which in turn depends on the alignment and sequentiality of writes. The example in Figure 1 demonstrates aligned and sequential writes and an example of the opposite. To convert the aligned mapping to block-level, the FTL can simply remove all page-level mappings and add a block-level mapping. To convert the unaligned mapping, the FTL has to read all the data, reorder, and write the data to a new block.

Due to the high cost of moving data, *clients of SSDs with hybrid FTLs should start writing at the aligned beginning of a block boundary and write sequentially*. This **Aligned Sequentiality** rule does not affect immediate performance since the conversion happens later, but violating this rule degrades sustainable performance because of costly data movement during the delayed conversion.

3.4 Grouping by Death Time

The death time of a page is the time the page is discarded or overwritten by the host. If a block has data with different death times, then there is a time window between the first and last page invalidations within which both live and dead data reside in the block. We call such a time window a

[1] NCQ technology was proposed to allow sending multiple requests to an HDD so the HDD can reorder them to reduce seek time. Modern SSDs employ NCQ to increase the concurrency of requests.

[2] We call the address space exposed by the SSD the logical space; a unit in the logical space is a logical page.

[3] $2\,GB = (512\,GB/2\,KB) * 8\,bytes$, where the $8\,bytes$ include $4\,bytes$ for each logical and physical page number.

[4] $233\,MB = (512\,GB \times 0.9/128\,KB + 512\,GB \times 0.1/2\,KB) \times 8\,bytes$.

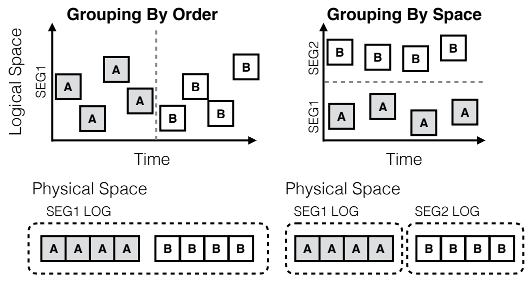


Figure 2: **Demonstration of Grouping by Death Time.** *Data A and B have different death times. In the figure, Vertical locations of the data in the same group are randomized to emphasize its irrelevance to this rule. Note that grouping by space is not available in non-segmented FTLs [48, 54, 95].*

zombie window and a block in a zombie window a *zombie block*. In general, larger zombie windows lead to increased odds of a block being selected for garbage collection and incurring costly data movement, as the FTL must move the live data to a new block and erase the victim block.

Zombie windows can be reduced if data with similar death times are placed in the same block [38, 44, 56, 76]. There are two practical ways to achieve this. First, the host can order the writes, so data with similar death times are gathered in the write sequence. Because many FTLs append data to a log, the consecutively written data is physically clustered, as demonstrated in Figure 2 (left). We call this *grouping by order*.

Second, the host can place different death groups in different portions of space. This approach relies on logical space segmentation, which is a popular technique in FTLs [57, 63, 74]. Because FTLs place data written in different segments to different logs, placing death groups to different logical segments isolates them physically, as demonstrated in Figure 2 (right). We call this *grouping by space*. Grouping by order and grouping by space both help to conform to the **Grouping By Death Time** rule. Note that clients of segmented FTLs can group by order or space. However, on non-segmented FTLs, grouping by space does not have any effect.

The rule of grouping by death time is often misunderstood as separating hot and cold data [75, 86, 94], which essentially can be described as grouping by *lifetime*. Note that two pieces of data can have the same lifetime (i.e., hotness) but distant death times. The advice of separating hot and cold data is inaccurate and misleading.

Grouping by death time does not affect immediate performance in page-level FTLs or hybrid FTLs, because in both cases data is simply appended to the log block. Violation of this rule impacts sustainable performance due to increasing the cost of garbage collection.

3.5 Uniform Data Lifetime

Flash cells can endure a limited number of program/erase (P/E) cycles before wearing out [55, 72]. The number of P/E cycles is on the order of 10^3 P/E cycles for recent commercial SSDs [10, 70, 83] and is expected to decrease in the future [68]. A cell that is worn out becomes unstable or completely unusable. Uneven block wearout can lead to loss of the over-provisioning area of an SSD, which is critical for performance. Severely uneven wearout can lead to premature loss of device capacity.

	Immediate Performance					**Sustainable Performance**				
Type	**RS**	**LC**	**AL**	**GP**	**LT**	**RS**	**LC**	**AL**	**GP**	**LT**
Page	✓	✓				✓	✓		✓	✓
Hybrid	✓					✓		✓		✓

Table 1: **The contract rules of on-demand page-level FTLs and hybrid FTLs.** *RS: Request Scale, LC: Locality, AL: Aligned Sequentiality, GP: Grouping By Death Time, LT: Uniform Data Lifetime. A check mark (✓) indicates that the immediate or sustainable performance of a particular type of FTL is sensitive to a rule.*

To prevent uneven wear out, FTLs conduct wear-leveling, which can be dynamic or static [30]. Dynamic wear-leveling evens the P/E count by using a less-used block when a new block is needed. Static wear-leveling is often done by copying data in a rarely-used block to a new location so the block can be used for more active data. Static wear-leveling can be done periodically or triggered with a threshold. Since static wear-leveling incurs costly data movement, which interferes foreground traffic and increases the total wear of the device, it is preferable to avoid.

To reduce wear-leveling cost, we introduce the **Uniform Lifetime** rule: *clients of SSDs should create data with similar lifetimes*. Data with relatively long lifetimes utilize blocks for long periods, during which data with shorter lifetimes quickly use and reduce the available P/E cycles of other blocks, leading to uneven wearout. If client data have more uniform lifetimes, blocks will be released for reuse after roughly the same amount of time. Lack of lifetime uniformity does not directly impact immediate performance, but impacts sustainable performance as it necessitates wear-leveling and leads to loss of capacity.

3.6 Discussion

The unwritten contract of SSDs is summarized in Table 1. Some rules in the contract are independent, and others are implicitly correlated. For example, Request Scale does not conflict with the other rules, as it specifies the count and size of requests, while the rest of the rules specifies the address and time of data operations. However, some rules are interdependent in subtle ways; for example, data writes with aligned sequentiality imply good locality; but good locality does not imply aligned sequentiality.

The performance impact of rule violations depends on the characteristics of the FTL and the architecture of the SSD. For example, violating the request scale rule can have limited performance impact if the SSD has only one channel and thus is insensitive to request scale; however, the violation may significantly reduce performance on an SSD with many

Rule	Impact	Metric	
Request Scale	7.2×, 18×	Read bandwidth	A
	10×, 4×	Write bandwidth	B
Locality	1.6×	Average response time	C
	2.2×	Average response time	D
Aligned Sequentiality	2.5×	Execution time	E
	2.4×	Erasure count	F
Grouping by Death Time	4.8×	Write bandwidth	G
	1.6×	Throughput (ops/sec)	H
	1.8×	Erasure count	I
Uniform Data Lifetime	1.6×	Write latency	J

Table 2: **Empirical Performance Impact of Rule Violations.** *This table shows the max impact of rule violations reported, directly or indirectly, by each related paper.* **A** *(from Figure 5 of [34]): 7.2× and 18× was obtained by varying the number of concurrent requests and request size, respectively.* **B** *is the same as A, but for write bandwidth.* **C** *(from Figure 11 of [48]): 1.6× was obtained by varying translation cache size and subsequently cache hit ratio. Thus it demonstrates the impact of violating the locality rule, which reduces hit ratio.* **D** *(from Figure 9 of [95]) is similar to C.* **E** *and* **F** *(from Figure 10(e) of [64]): 2.5× and 2.4× are obtained by varying the number of blocks with page-level mapping in a hybrid FTL, which leads to different amounts of merge operations.* **G** *(from Figure 8(b) of [62]): it is obtained by running a synthetic benchmark on ext4 for multiple times on a full SSD.* **H** *(from Figure 3(a) of [56]) and* **I** *(from Figure 8(a) of [38]): the difference is between grouping and non-grouping workloads.* **J** *(from Figure 12 of [30]): the difference is due to static wear-leveling activities.*

channels. Although we do not focus on quantifying such performance impact in this paper, in Table 2 we present the empirical performance impact of rule violations reported, either directly or indirectly, by existing literature.

The fact that an SSD presents certain rules does not necessarily mean that SSD clients can always comply. For example, an SSD may require a client to group data with the same death time by order, but this requirement may conflict with the durability needs of the client; specifically, a client that needs durability may frequently flush metadata and data together that have different death times. Generally, a client should not choose an SSD with rules that the client violates. However, due to the multi-dimensional requirements of the rules, such an SSD may not be available. To achieve high performance in such an environment, one must carefully study the workloads of clients and the reactions of SSDs.

4. Methodology

The contractors of an SSD are the applications and file systems, which generate the SSD workload, i.e., a sequence of I/O operations on the logical space. Both applications and file systems play important roles in determining the I/O pattern. Application developers choose data structures for various purposes, producing different I/O patterns; for example, for searchable data records, using a B-tree to layout data in a file can reduce the number of I/O transfers, compared with an array or a linked list. File systems, residing between applications and the SSD, may alter the access pattern of the workload; for example, a log-structured file system can turn random writes of applications into sequential ones [80], which may make workloads comply with the contract of HDDs.

How to analyze SSD behaviors? We run combinations of applications and file systems on a commodity SSD, collect block traces and feed the traces to our discrete-event SSD simulator, *WiscSim*.[5] *WiscSim* allows us to investigate the internal behaviors of SSDs. To the best of our knowledge, *WiscSim* is the first SSD simulator that supports NCQ [25, 48, 53, 59]. *WiscSim* is fully functional, supporting multiple mapping and page allocation schemes, garbage collection, and wear-leveling. The input trace of *WiscSim* is collected on a 32-core machine with a modern SATA SSD with a maximum NCQ depth of 32 [4], which allows concurrent processing of up to 32 requests. We use a 1-GB partition of the 480 GB available space, which allows us to simulate quickly. Our findings hold for larger devices as our analysis will demonstrate.

Why design a new simulator? We develop a new simulator instead of extending an existing one (e.g., FlashSim [48, 59], SSDsim [53], and SSD extension for DiskSim [25]). One reason is that most existing simulators (FlashSim and SSDsim) do not implement discrete-event simulation[6] [48, 53, 59], a method for simulating queuing systems like SSDs [79]. Without discrete-event simulation, we found it challenging to implement critical functionality such as concurrent request handling (as mandated by NCQ). The second reason is that existing simulators do not have comprehensive tests to ensure correctness. As a result, we concluded that the amount of work to extend existing platforms exceeded the implementation of a new simulator.

Why focus on internal metrics instead of end-to-end performance? Our analysis in this paper is based not on end-to-end performance[7], but on the internal states that impact end-to-end performance. The internal states (e.g., cache miss ratio, zombie block states, misaligned block states) are fundamental sources of performance change. We have validated the correctness of the internal states with 350 unit tests; some of the tests examine the end-to-end data integrity to ensure that all components (e.g., address translation, garbage collection) work as expected.

What are the applications studied? We study a variety of applications, including LevelDB (version 1.18) [6], RocksDB (version 4.11.2) [14], SQLite (Roll Back mode and Write-Ahead-Logging mode, version 3.8.2) [18] and

[5] *WiscSim* has 10,000 lines of code for SSD simulation core. *WiscSee*, which includes *WiscSim*, has 32,000 lines of code in total. It is well tested with 350 tests, including end-to-end data integrity tests.

[6] They are not discrete-event simulations, even though the source code contains data structures with name "`event`".

[7] Our simulator does show reasonable end-to-end performance.

the Varmail benchmark [2]. LevelDB is a popular NoSQL database based on log-structured merge trees; log-structured merge trees are designed to avoid random writes through log structuring and occasional compaction and garbage collection. Its periodic background compaction operations read key-value pairs from multiple files and write them to new files. RocksDB is based on LevelDB but optimizes its operations for SSDs by (among other things) increasing the concurrency of its compaction operations. SQLite is a popular B-tree based database widely used on mobile devices, desktops and cloud servers. The default consistency implementation is roll-back journaling (hereafter refer to as RB), in which a journal file, containing data before a transaction, is frequently created and deleted. More recent versions of SQLite also support write-ahead logging (hereafter referred to as WAL), in which a log file is used to keep data to be committed to the database. The WAL mode typically performs less data flushing and more sequential writes. Varmail is a benchmark that mimics the behavior of email servers which append and read many small (tens of KBs) files using 16 threads. SSDs are often used to improve the performance of such workloads.

Why are these applications chosen? Through these applications, we examine how well application designs comply with the SSD contract in interesting ways. With the collection of databases, we can study the differences of access patterns between essential data structures: B-tree and LSM-Tree. We can also study the effectiveness of SSD optimizations by comparing LevelDB and RocksDB. Additionally, we can investigate differences between mechanisms implementing the same functionality – implementing data consistency by Write-Ahead Logging and Roll Back journaling in SQLite. Besides databases, we choose Varmail to represent a large class of applications that operate on multiple files, flush frequently, and request small data. These applications perform poorly on HDDs and demand SSDs for high performance. The applications that we chose cover a limited space of the population of existing applications, but we believe that our findings can be generalized and the tools we designed are helpful in analyzing other applications.

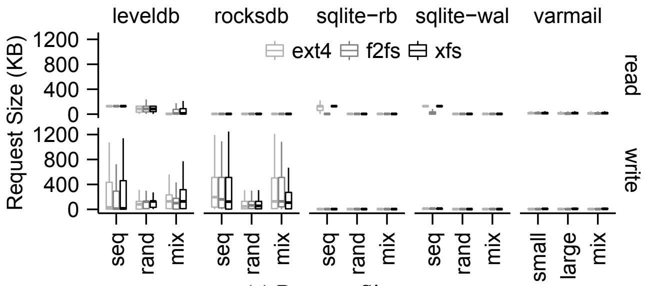
What access patterns are studied? We study a variety of usage patterns for each application. These patterns include sequential, random insertions and queries, as well as their mix, for all database applications. Sequential and random queries are conducted on sequentially and randomly inserted databases, respectively. LevelDB and RocksDB are driven by their built-in benchmark *db_bench*, using 16 byte keys and 100 byte values. SQLite is driven by a simple microbenchmark that we developed to perform basic operations; we commit a transaction after every 10 operations. The SQLite database has the same key and value sizes as LevelDB and RocksDB. The exact number of operations (insertions, updates or queries) performed on these database applications depend on the goal of the experiment. For example, to evaluate the Uniform Data Lifetime rule, we insert and update key-value records for hundreds of millions of times. For Varmail, we study small, large, and mixed (i.e., both small and large) collections of files, which reflect small, large, and mixed email workloads on a single server. We limit the memory usage of each application with Linux control groups [7] to avoid large cache effects.

What file systems are studied? We study two traditional file systems (ext4 and XFS) and a newer one that is designed for SSDs (F2FS), all on Linux 4.5.4. Both ext4 and XFS are among the most mature and popular Linux file systems. Although originally designed for HDDs, the fact that they are stable and well-established has caused them to be widely used for SSDs [1, 3]. F2FS (Flash-Friendly File System) is a log-structured file system that is claimed to be optimized for modern SSDs. F2FS is a part of the mainline Linux kernel and is under active development. In our evaluation, we enable discard (also known as trim) support for all file systems, as suggested by some major cloud providers [1, 3].

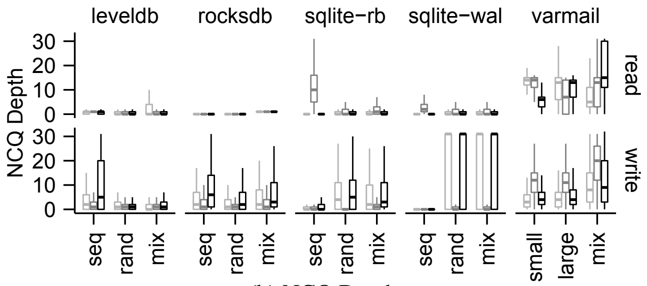
How to evaluate rule violations? We evaluate how well the contractors conform to each rule with the help of *WiscSee*. *WiscSee* automatically executes, traces and analyzes combinations of applications and file systems. To examine request scale and uniform data lifetime, *WiscSee* analyzes the traces directly; for locality, aligned sequentiality, and grouping by death time, *WiscSee* feeds the traces through *WiscSim* as these items require understanding the internal states of the SSD. The best metrics for evaluations are often suggested by the rule. For example, to evaluate locality we examine miss ratio curves [89, 90], whereas to understand death time, we introduce a new metric, zombie curves.

We evaluate rules individually, which has several benefits. First, it makes the analysis relevant to a wide spectrum of FTLs. An FTL is sensitive to a subset of rules; understanding each rule separately allows us to mix and match the rules and understand new FTLs. Second, it prevents the effects of rules from confounding each other. For example, analyzing a workload on an FTL that is sensitive to two rules can make it difficult to determine the source of performance degradation.

How to identify the root of a violation? Although *WiscSee* shows the performance problems, it does not directly reveal their root causes. However, using the hint from *WiscSee*, we can find out their causes by examining the internals of applications, file systems, and the SSD simulator. Because the source code of the applications, file systems and *WiscSim* are all available, we can understand them by freely tracing their behaviors or making experimental changes. For example, with the information provided by applications and file systems, we can investigate *WiscSim* and find the semantics of garbage data, why the garbage data was generated, and who is responsible.



(a) Request Size



(b) NCQ Depth

Figure 3: **Request Scale - Distributions of Request Size and NCQ Depth.** *Data of different applications is shown in columns. The top (read) and bottom (write) panels show the read and write results, respectively; the X axis indicates I/O patterns. Inside each panel, the top and bottom border of the box show the third and first quartile; the heavy lines in the middle indicate the medians. The whiskers indicate roughly how far data points extend [13]. Note that Linux block layer splits requests if they exceed a maximum size limit (1280 KB in our case) [8].*

5. The Contractors

In this section, we present our observations based on vertical analysis of applications, file systems, and FTLs. The observations are categorized and labeled by their focuses. Category **App** presents general behaviors and differences across applications. Category **FS** presents general behaviors and differences across file systems.

We will show, by analyzing how well each workload conforms to or violates each rule of the contract, that we can understand its performance characteristics. Additionally, by vertical analysis of these applications and file systems with FTLs, we hope to provide insights about their interactions and shed light on future designs in these layers.

5.1 Request scale

We evaluate and pinpoint request scale violations from applications and file systems by analyzing block traces, which include the type (read, write, and discard), size, and time (issue and completion) of each request. Since the trace is collected using a small portion of a large SSD, the traced behaviors are unlikely to be affected by SSD background activities, which should occur at a negligible frequency.

Figure 3 shows the distributions of request sizes and NCQ depths. As we can see from the figures, the request scale varies significantly between different applications, as well as file systems. The difference between traditional file systems (i.e. ext4 and XFS) and log-structured file system (i.e. F2FS) is often significant.

Observation #1 (App): ***Log structure increases the scale of write size for applications, as expected.*** LevelDB and RocksDB are both log-structured, generating larger write requests than SQLiteRB, SQLiteWAL, and Varmail, in which write requests are limited by transaction size (10 insertions of 116-byte key-value pairs) or flush size (average 16 KB). However, the often large write amplification introduced by a log structure is harmful for SSDs [61]. We do not discuss this issue here as we focus on SSD interface usage.

Observation #2 (App): ***The scale of read requests is often low.*** Unlike write requests, which can be buffered and

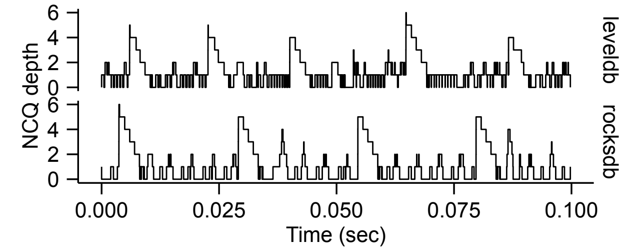


Figure 4: **Request Scale - NCQ Depths During Compaction.** *The lower data points are from read requests; higher ones are from writes.*

enlarged, the scale of read requests is harder to increase. Small requests, such as the database entries used in our evaluation, cannot be batched or concurrently issued due to dependencies. Users may need to query one key before another, or the database may need to read an index before reading data from the location given by the index. Figure 3a also shows that LevelDB issues larger requests than RocksDB because RocksDB disables Linux's default readahead behavior so the OS cache contains only explicitly requested data.

Observation #3 (App): ***SSD-conscious optimizations have room for improvements.*** Neither RocksDB, which is optimized for SSDs, nor LevelDB is able to saturate device resources. Figure 3b shows that RocksDB is only able to use a few more NCQ slots than LevelDB, despite RocksDB's use of multi-threaded compaction to increase SSD parallelism [14].[8] We do see the number of writing processes increase, but the write concurrency does not increase and device bandwidth is underutilized (bandwidth results are not shown). For example, Figure 4 shows a snippet of NCQ depth over time on ext4 for compaction operations in LevelDB and RocksDB. RocksDB does not appear to use NCQ slots more efficiently than LevelDB during compaction. One obvious optimization would be to perform reads in parallel, as the figure shows that RocksDB reads several files serially, indicated by the short spikes. The rela-

[8] We have set the number of compaction threads to be 16.

tively higher queue depth shown in Figure 3b is due to higher concurrency during flushing memory contents.

Observation #4 (App): ***Frequent data barriers in applications limit request scale.*** Data barriers are often created by synchronous data-related system calls such as `fsync()` and `read()`, which LevelDB, RocksDB, SQLite and Varmail all frequently use. Since a barrier has to wait for all previous requests to finish, the longest request wait time determines the time between barriers. For example, the writes in Figure 4 (higher depth) are sent to the SSD (by `fdatasync()`) at the same time but complete at different times. While waiting, the SSD bandwidth is wasted. As a result, frequent application data barriers significantly reduce the number of requests that can be concurrently issued. Although the write data size between barriers in LevelDB and RocksDB is about 2 MB on average (which is much larger than the sizes of SQLiteRB, SQLiteWAL, and Varmail), barriers still degrade performance. As Figure 4 shows, the write and read barriers frequently drain the NCQ depth to 0, underutilizing the SSD.

Observation #5 (FS): ***Linux buffered I/O implementation limits request scale.*** Even though LevelDB and RocksDB read 2 MB files during compactions, which are relatively large reads, their request scales to the SSD are still small. In addition to previously mentioned reasons, the request scale is small because the LevelDB compaction, as well as the RocksDB compaction from the first to the second level ("the only compaction running in the system most of the time" [15]), are single-threaded and use buffered reads.

The Linux buffered read implementation splits and serializes requests before sending to the block layer and subsequently the SSD. If buffered `read()` is used, Linux will form requests of `read_ahead_kb` (default: 128) KB, send them to the block layer and wait for data one at a time. If buffered `mmap()` is used, a request, which is up to `read_ahead_kb` KB, is sent to the block layer only when the application thread reads a memory address that triggers a page fault. In both buffered `read()` and `mmap()`, only a small request is sent to the SSD at a time, which cannot exploit the full capability of the SSD. In contrast to buffered reads, direct I/O produces much larger request scale. The direct I/O implementation sends application requests in whole to the block layer. Then, the block layer splits the large requests into smaller ones and sends them asynchronously to the SSD.

Application developers may think reading 2 MB of data is large enough and should achieve high performance on SSDs. Surprisingly, the performance is low because the request scale is limited by a seemingly irrelevant setting for readahead. To mitigate the problem, one may set `read_ahead_kb` to a higher value. However, such setting may force other applications to unnecessarily read more data. In addition, the request to the block layer is limited up to a hard-coded size (2 MB), to avoid pinning too much memory on less capable machines. We believe this size should be tunable so that one can achieve larger request scale on more capable machines and storage devices. Other ways to avoid the buffered read problem include reading by multiple threads or by small asynchronous I/Os. However, these approaches unnecessarily complicate programming. We believe that the Linux I/O path should be re-examined to find and fix similar problems when we transit from the HDD to the SSD era.

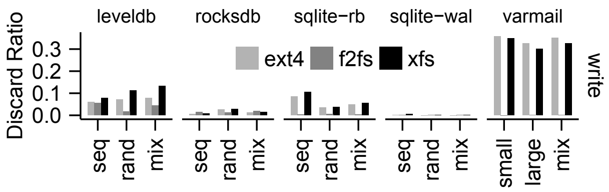


Figure 5: **Request Scale - Ratio of Discard Operations.** *The discard ratios of read workloads are not shown because they issue a negligible number of discards (or none at all).*

Observation #6 (FS): ***Frequent data barriers in file systems limit request scale.*** File systems also issue barriers, which are often caused by applications and affect all data in the file system [39, 77]. Journaling in ext4 and XFS, often triggered by data flushing, is a frequent cause of barriers. Checkpointing in F2FS, which is often triggered by `fsync`-ing directories for consistency in LevelDB, RocksDB, and SQLiteRB, suspends all operations. Barriers in file systems, as well as in applications (Observation 4), limit the benefit of multi-process/thread data access.

Observation #7 (FS): ***File system log structuring fragments application data structures.*** F2FS issues smaller reads and unnecessarily uses more NCQ slots than ext4 and XFS for sequential queries of SQLiteRB and SQLiteWAL. This performance problem arises because F2FS breaks the assumption made by SQLiteRB and SQLiteWAL that file systems keep the B-tree format intact. For SQLiteRB, F2FS appends both the database data and the database journal to the same log in an interleaved fashion, which fragments the database. For SQLiteWAL, F2FS also breaks the B-tree structure, because F2FS is log-structured, causing file data layout in logical space to depend on the time of writing, not its offset in the file. Due to the broken B-tree structure, F2FS has to read discontiguous small pieces to serve sequential queries, unnecessarily occupying more NCQ slots, while ext4 and XFS can read from their more intact B-tree files.

When the number of available NCQ slots is limited or the number of running applications is large, workloads that require more NCQ slots are more likely to occupy all slots, causing congestion at the SSD interface. In addition, for the same amount of data, the increased number of requests incur more per-request overhead.

Observation #8 (FS): ***Delaying and merging slow non-data operations could boost immediate performance.*** Non-data discard operations occupy SSD resources, including NCQ slots, and therefore can reduce the scale of more

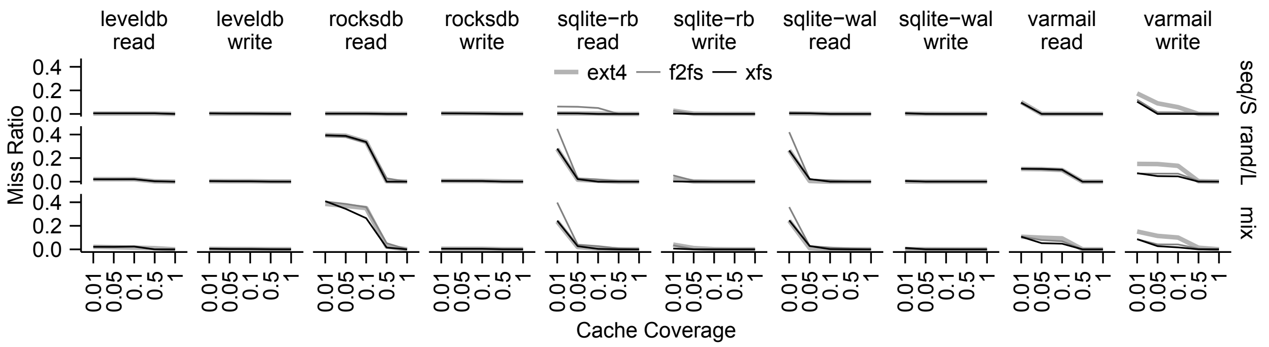


Figure 6: **Locality - Miss Ratio Curves.** *We set the cache to cover 1%, 5%, 10%, 50%, and 100% of the logical space.* `seq/S` *indicates sequential for databases and small set of files for Varmail.* `rand/L` *indicates random for databases and large set of files for Varmail.*

immediately-valuable read and write operations. We present the ratio of discard operations to all operations for all write workloads in Figure 5. As we can see, ext4 and XFS often issue more discard requests than F2FS, because ext4 and XFS both immediately discard the logical space of a file when it is deleted. SQLiteWAL reuses its write-ahead log file instead of deleting it and thus incurs very few discard operations. On the other hand, SQLiteRB and Varmail frequently create and delete small files, leading to many small discard operations. Such behavior may lead to severe performance degradation on SSDs that do not handle discard operations quickly (a common problem in modern SSDs [22, 52]). In contrast to ext4 and XFS, F2FS attempts to delay and merge discard operations, which boosts immediate performance by reducing the frequency and increasing the size of discard operations. However, later we will show that this (sometimes infinite) delay in performing discards can significantly degrade sustainable performance.

5.2 Locality

We study locality by examining miss ratio curves [89, 90], obtained from *WiscSim* with an on-demand page-level mapping scheme based on DFTL [48]. We revise the cache replacement policy of the mapping scheme to be aware of spatial locality, as it is a common and important attribute of many workloads [43, 89, 90]. In this FTL, one translation page contains entries that cover 1 MB of contiguous logical space. Our replacement policy prefers to evict clean entries rather than dirty ones. Our locality study here is applicable in general, as locality is a valuable property in storage systems.

Figure 6 presents the miss ratios curves. Columns indicate different combinations of applications and read/write modes. For example, `leveldb.read` and `level.write` indicate LevelDB query and insertion workloads, respectively. Rows indicate workload I/O patterns. The x-axis shows the fractions of logical space that can be covered by the different cache sizes. Intuitively, small and distant requests tend to have poor locality and thus higher miss ratios. As we can see for writes, log-structured applications (LevelDB, RocksDB, and SQLiteWAL) have better locality than others, as log-structured writing produces a more sequential I/O pattern. Read locality, on the other hand, is highly dependent on the pattern of requests.

Observation #9 (App): ***SSDs demand aggressive and accurate prefetching.*** RocksDB queries (`rocksdb.read`) experience much higher miss ratios than LevelDB, because RocksDB disables Linux's readahead and thus issues much smaller requests (as discussed in Section 5.1). The high miss ratio, as well as low request scale, leads to low utilization of the SSD. On the other hand, LevelDB enables readahead, which naively prefetches data nearby; the prefetched data could go unused and unnecessarily occupy host memory. We believe that, with powerful SSDs, aggressive and accurate prefetching should be used to boost SSD utilization and application performance.

Observation #10 (FS): ***Aggressively reusing space improves locality.*** XFS achieves the best locality on all workloads, because XFS aggressively reuses space from deleted files by always searching free space from the beginning of a large region (XFS allocation group). In contrast, other file systems delay reuse. F2FS delays discarding space of deleted files (Observation 8) and therefore delays their reuse; ext4 prefers to allocate new space near recent allocated space, which implicitly avoids immediate space reuses.

Observation #11 (FS): ***Legacy policies could break locality.*** For the Varmail write workload, ext4 has much higher miss ratios than XFS and F2FS, because (ironically) an allocation policy called Locality Grouping breaks locality. Locality grouping was originally designed to optimize small file handling for HDDs by storing them in globally shared preallocated regions to avoid long seeks between small files [27, 51]. However, the small writes of Varmail in fact spread across a large area and increase cache misses. Data is spread for several reasons. First, ext4 pre-allocates a group of 2 MB regions (a locality group) for each core repeatedly as they are filled. Second, the large number of CPU cores in modern machines lead to a large number of locality groups. Third, writes can go to any place within the locality groups depending on which core performs the write. The combination of these factors leads to small and scattered write requests for ext4 running Varmail. Similarly to Varmail, SQLiteRB read, which also frequently creates and deletes files, suffers slightly degraded performance as a result of locality grouping.

Observation #12 (FS): ***Log structuring is not always log-structured.*** For the Varmail write workload, F2FS often suffers from larger miss ratios than XFS, despite its log-structured design which should lead to good spatial locality and thus very low miss ratios. F2FS has high miss ratios because it frequently switches to a special mode called in-place-update mode, which issues many small, random writes over a large portion of the device. The reason for F2FS to switch to in-place-update mode is to reduce the metadata overhead of keeping track of new blocks. In-place-update mode is triggered if the following two conditions are satisfied. First, flush size must be less than 32 KB. Second, the workload must be overwriting data. Surprisingly, despite its append-only nature, Varmail still triggers F2FS's in-place update. It satisfies the first condition easily, because it flushes data in random small quantities (16 KB on average). More subtly, while Varmail is only *appending* to each file, if the previous append to the file only partially occupies its last sector (4 KB), the current append operation will read, modify and *overwrite* the last sector of the file, satisfying the second condition. We call such behavior *partial sector use*. Because the two conditions are satisfied, partial sector use in Varmail triggers in-place-update mode of F2FS, which results in small, scattered write requests among many previously written files. It is easy to see how such behavior can also break the Aligned Sequentiality rule, which we will discuss in Section 5.3. Note that SQLite can also incur partial sector use since its default operation unit size is 1 KB. [9]

Observation #13 (FS): ***Log structuring can spread data widely across a device and thus reduce locality.*** F2FS has the highest miss ratios for most SQLiteRB and SQLiteWAL query workload. This poor cache performance arises because F2FS spreads database data across logical space as it appends data to its log and it breaks the B-tree structure of SQLite, as we have mentioned in Section 5.1.

5.3 Aligned Sequentiality

We study aligned sequentiality by examining the unaligned ratio, which is the ratio of the size of data in the SSD with misaligned mappings (see Figure 1) to total application file size. The data pages with misaligned mapping are potential victims of expensive merging. The unaligned ratio is obtained using a hybrid mapping scheme [74] in *WiscSim*. Requests are not striped across channels in our multi-channel implementation of this scheme as striping immediately fragments mappings.

[9] Beginning with SQLite version 3.12.0 (2016-03-29), the default unit size has been increased to 4 KB.

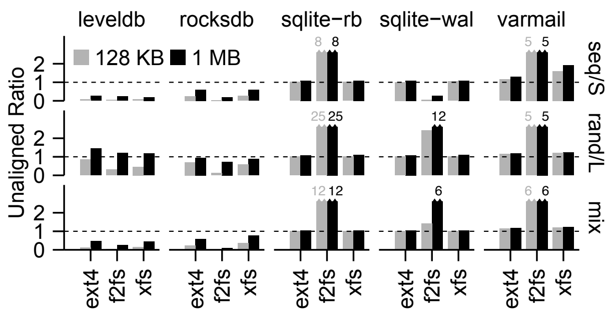


Figure 7: **Aligned Sequentiality - Unaligned Ratios.** *The dashed line indicates unaligned ratio of 1.0: the SSD internally has the same amount of data with unaligned mapping as the amount of application file data. Different colors indicate different block sizes.*

Flash block size is an important factor that affects aligned sequentiality. If the block size is large, logical writes must be aligned and sequential in a wider logical area in order to maintain logical-to-physical address alignment. Therefore, smaller block sizes make an SSD less sensitive to alignment. Unfortunately, block sizes tend to be large [20, 21]. We analyze different block sizes to understand how it impacts different combinations of applications and file systems.

Figure 7 shows the unaligned ratios of different combinations of applications, I/O patterns, file systems and block sizes. Read workloads do not create new data mappings and thus are not shown. As we can see, applications with large log-structured files (LevelDB and RocksDB) often have lower unaligned ratios than other applications. In addition, unaligned ratios can be greater than 1, indicating that there is more unaligned data inside the SSD than the application file data. The figure also shows that larger block sizes lead to higher unaligned rations.

Observation #14 (App): ***Application log structuring does not guarantee alignment.*** The log-structured LevelDB and RocksDB can also have high unaligned ratios, despite writing their files sequentially. On ext4 and XFS, the misalignment comes from aggressive reuse of space from deleted files, which can cause the file system to partially overwrite a region that is mapped to a flash block and break the aligned mapping for that block. F2FS often has smaller unaligned ratios than ext4 and XFS because it prefers to use clean F2FS segments that contain no valid data, where F2FS can write sequentially; however, there is still misalignment because if a clean F2FS segment is not available, F2FS triggers threaded logging (filling holes in dirty segments).

Observation #15 (FS): ***Log-structured file systems may not be as sequential as commonly expected.*** Except for sequential insertion (`seq/S`) on SQLiteWAL, both SQLiteRB and SQLiteWAL have very high unaligned ratios on F2FS, which is supposed to be low [62]. For SQLiteRB, F2FS sees high unaligned ratios for two reasons. First, in every transaction, SQLiteRB overwrites the small journal file header and

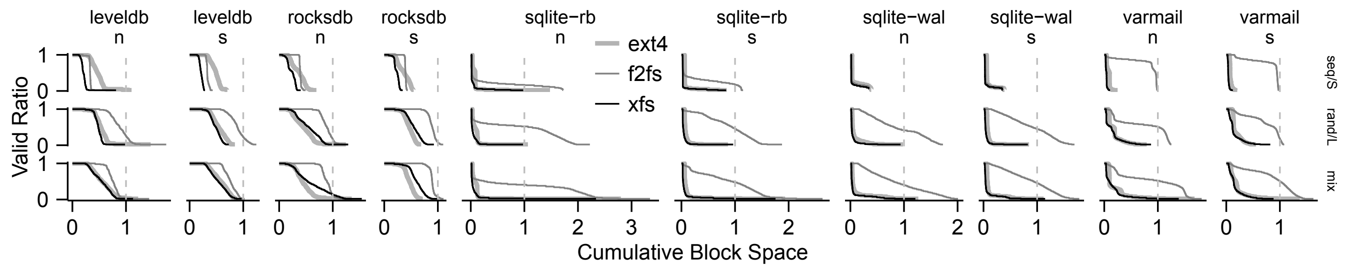


Figure 8: **Grouping by Death Time - Zombie Curves (Stable Valid Ratios).** *A zombie curve shows the sorted non-zero valid ratios of blocks. Labels* `'n'` *and* `'s'` *indicate non-segmented and segmented FTLs, respectively. The x-axis is normalized to logical space size. Zombie curves show important usage characteristics of SSDs. A curve that is much shorter than 1 on x-axis indicates that only a small portion of blocks are occupied (i.e., fully or partially valid); a curve reaches beyond 1 on x-axis indicates that the total size of occupied blocks is larger than the logical space size (some over-provisioned blocks are occupied). The area size under a curve is the size of valid data in proportion to the logical space size. An under-curve area with size close to 0 (e.g., sqlite-wal with seq/S on ext4) indicates a small amount of valid data in the simulated SSD; an under-curve area with size close to 1 (e.g., Varmail on F2FS) suggests that almost all the data on logical space is considered valid by the SSD.*

a small amount of data inside the database file. Both cases trigger in-place updates, which break sequentiality. Second, the FTL has to keep a large amount of *ghost data* that is deleted by the application but not discarded by the file system, which is one of the major reasons that unaligned ratio can be greater than 1. Ghost data is produced because F2FS delays discard operations until an F2FS segment (2 MB) contains no valid application data and thus becomes "clean". However, SQLiteRB's valid data is spread across many segments, which are considered "dirty" by F2FS. F2FS does not clean these dirty segments as it would interfere with current application traffic. Thus, many segments are not discarded, leaving a large amount of ghost data. The write pattern of SQLiteWAL is different, however. The combination of SQLiteWAL and F2FS violates aligned sequentiality because merging data from the write-ahead log to the database also incurs small discrete updates of the database file, triggering in-place updates in F2FS; it also has a large amount of unaligned ghost data due to the delayed discarding of dirty F2FS segments.

Observation #16 (FS): ***sequential $+$ sequential $\neq$ sequential.*** Surprisingly, the append-only Varmail with log-structured F2FS produces non-sequential writes and has high unaligned ratios. This non-sequentiality is caused by partial-sector usage (discussed in Section 5.2) which triggers F2FS in-place update mode, and F2FS produces ghost data from delayed discards. Varmail has high unaligned ratios on ext4 and XFS as it appends to random small files.

5.4 Grouping by Death Time

We introduce *zombie curve analysis* to study Grouping By Death Time. We set the space over-provisioning of *WiscSim* to be infinitely large and ran workloads on it for a long time. While running, we periodically take a snapshot of the valid ratios of the used flash blocks, which is the ratio of valid pages to all pages inside a block. The valid ratios provide useful information about zombie (partially valid) blocks, which are the major contributors to SSD garbage collection overhead. We find that the distribution of valid ratios quickly reaches a stable state. The *zombie curve* formed by stable sorted valid ratios can be used to study how well each file system groups data by death time. The *WiscSim* FTL used for this study is based on DFTL [48], with added support for logical space segmentation and multiple channels.

Figure 8 presents the zombie curve for each workload. An ideally grouped workload would be shown as a vertical cliff, indicating zero zombie blocks. In such a case, the garbage collector can simply erase and reuse blocks without moving any data. A workload that grossly violates grouping by death time has a curve with a large and long tail, which is more likely to incur data movement during garbage collection. This large and long tail arises because garbage collection must move the data in zombie blocks to make free space if available flash space is limited. Analysis by zombie curves is generally applicable because it is independent of any particular garbage collection algorithm or parameter.

Observation #17 (App): ***Application log structuring does not reduce garbage collection.*** It has long been believed that application-level log structure reduces garbage collection, but it does not. As shown in Figure 8, LevelDB and RocksDB have large tails (gradual slopes), especially for `rand/L` and `mix` patterns. Sequential writes within individual files, which are enabled by log structuring, do not reduce garbage collection overhead as indicated by the zombie curves.

The fundamental requirement to reduce garbage collection overhead is to group data by death time, which both LevelDB and RocksDB do not satisfy. First, both LevelDB and RocksDB have many files that die at different times because compactions delete files at unpredictable times. Second, data of different files are often mixed in the same block. When LevelDB and RocksDB flush a file (about 2 MB), the file data will be striped across many channels to exploit the internal parallelism of the SSD [34]. Since each block receives a small piece of a file, data from multiple files will be mixed in the same flash block. Our 128-KB block in the simulation may mix data from two files since the 2-MB file data is striped across 16 channels and each channel receives 128 KB of data, which may land on two blocks. As blocks are often bigger in modern SSDs, more files are likely to be mixed together. Third, files flushed together by foreground insertions (from memory to files) and background compactions are also mixed in the same block, because the large application flush is split into smaller ones and sent to the SSD in a mixed fashion by the Linux block layer.

Another problem of LevelDB and RocksDB is that they both keep ghost data, which increases garbage collection overhead. Even if users of LevelDB and RocksDB delete or overwrite a key-value pair, the pair (i.e., ghost data) can still exist in a file for a long time until the compaction process removes it. Such ghost data increases the tail size of a zombie curve and the burden of garbage collection.

Observation #18 (FS): ***Applications often separate data of different death time and file systems mix them.*** Both ext4 and XFS have thin long tails for SQLiteRB and SQLiteWAL. These long tails occur because ext4 and XFS mix database data with the database journal and write-ahead log for SQLiteRB and SQLiteWAL, respectively. Since database journal and write-ahead log die sooner than the database data, only the database data is left valid in zombie blocks. Note that our experiments involve only up to two SQLite instances. Production systems running many instances could end up with a fat, long tail.

Observation #19 (FS): ***All file systems typically have shorter tails with segmented FTLs than they have with non-segmented FTLs, suggesting that FTLs should always be segmented.*** Segmentation in logical space enables grouping by space. Since file systems often place different types of data to different locations, segmentation is often beneficial. The longer tail of non-segmented FTL is due to mixing more data of different death times, such as application data and the file system journal. The difference between segmented and non-segmented FTLs are most visible with SQLiteRB.

Observation #20 (FS): ***All file systems fail to group data from different directories to prevent them from being mixed in the SSD.*** Data belonging to different users or application instances, which are often stored in different file system directories, usually have different death times. For example, one database may be populated with long-lived data while another is populated with short-lived data. Linux ext4 fails to group data from different directories because its locality group design (Section 5.2) mixes all small files (64 KB) and its stream optimization appends chunks of large files ($> 64\,KB$) next to each other (stream optimization was originally designed to avoid long seeks while streaming multiple files on HDDs) [51]. XFS groups data from different directories in different regions, known as allocation groups. However, data from different directories may still end up mixed when allocation groups overflow or XFS runs out of empty allocation groups. F2FS tries to isolate different types of data, such as file data, file inodes, directory entries, and directory inodes. Unfortunately, F2FS mixes data from all files (except files with a few specific extensions such as `mp3`) written to the file system into a single log regardless of their parent directories, grossly violating grouping by death time.

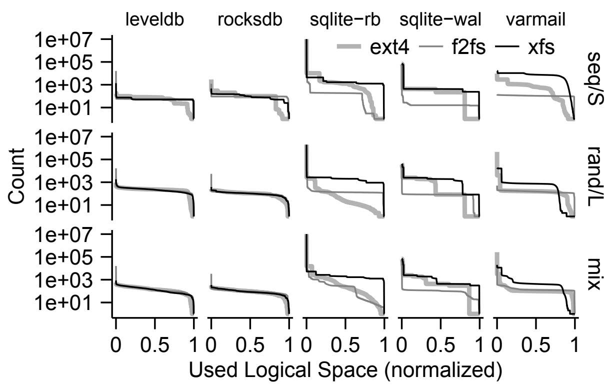


Figure 9: **Uniform Data Lifetime - Sorted Write Count of Logical Pages.** *Y axis scale is logarithmic; pages without data are excluded.*

Observation #21 (FS): ***F2FS sacrifices too much sustainable performance for immediate performance.*** F2FS exhibits a much larger tail than ext4 and XFS for SQLiteRB, SQLiteWAL, Varmail, and non-sequential patterns of LevelDB and RocksDB. This poor behaviors materializes because F2FS delays discards, sometimes infinitely, of data that is already overwritten or deleted by applications, whereas ext4 and XFS discard them immediately, as discussed in Section 5.1 and 5.3. We determine that F2FS sacrifices sustainable performance for immediate performance because the un-discarded data becomes ghost data which produces a large number of zombie blocks and increases garbage collection overhead. The large amount of zombie blocks makes the pair of SQLiteRB and F2FS very sensitive to SSD garbage collection capabilities. On SSDs with fast discard operations this trade-off could be counterproductive, because it may not improve immediate performance but could degrade sustainable performance significantly.

5.5 Uniform Data Lifetime

The uniform data lifetime rule is for reducing flash cell program/erase variance and wear-leveling cost. The exact cost of wear-leveling varies significantly between different algorithms and architectures [30, 55, 72]. Data lifetime variance is the fundamental source of program/erase variance and wear-leveling cost.

We use the write count of logical pages to estimate data lifetime. The lifetime of a piece of data starts when it is written to a logical page address, and ends when the physical page address is discarded or overwritten by new data. If one logical page address has many more writes than another during the same time period, the data written on the former logical page address has much shorter lifetime than the later on average. Therefore, a higher write count on a logical page address indicates that the address is written with data of shorter lifetime and with more data.

Since program/erase variance is a long-term effect, we run our applications much longer to gather statistics. Each combination of application and file system generates at least 50 times and up to 800 times more data traffic than the logical capacity. Based on our analysis, we believe our results are representative even for much larger traffic sizes.

In Figure 9, we show the logical space write count of different combinations of applications and file systems. For ease of comparison, the x-axis (logical pages) is sorted by write count and normalized to the written area. Ideally, the curve should be flat. We can see that log-structured applications are more robust than others across different file systems. Also, small portions of the logical space tend to be written at vastly higher or lower frequencies than the rest.

Observation #22 (FS): ***Application and file system data lifetimes differ significantly.*** The write counts of application data and the file system journal often vary significantly. For example, when running SQLiteWAL (`rand/L`) on XFS, the database data is written 20 times more than the journal on average (the flat line on the right side of the curve is the journal). In addition, when running Varmail (`rand/L`) on ext4, the journal is on average written 23 times more than Varmail data. The most frequently written data (the journal superblock) is written 2600 times more than average Varmail data. Such a difference can lead to significant variance in flash cell wear.

F2FS shows high peaks on LevelDB, RocksDB, and SQLiteRB, which is due to F2FS checkpointing. F2FS conducts checkpointing, which writes to fixed locations (checkpoint segment, segment information table, etc.), when applications call `fsync` on directories. Since LevelDB, RocksDB and SQLiteRB call `fsync` on directories for crash consistency, they frequently trigger F2FS checkpointing, which writes to fixed locations. Such high peaks are particularly harmful because the large amount of short-lived traffic frequently programs and erases a small set of blocks, while most blocks are held by the long-lived data.

Observation #23 (FS): ***All file systems have allocation biases.*** Both ext4 and XFS prefer to allocate logical space from low to high addresses due to their implementations of space search. F2FS prefers to put user data at the beginning of logical space and node data (e.g. inode, directory entries) at the end. These biases could lead to significant lifetime differences. For example, on Varmail (`seq/S`) ext4 touches a large area but also prefers to allocate from low addresses, incurring significant data lifetime variance.

Observation #24 (FS): ***In-place-update file systems preserve data lifetime of applications.*** Both ext4 and XFS show high peaks with SQLiteRB and SQLiteWAL because SQLite creates data of different lifetimes and both file systems preserve it. The high peak of SQLiteRB is due to overwriting the database header on every transaction. For SQLiteWAL, the high peak is due to repeatedly overwriting the write-ahead log. The large portion of ext4 logical space with low write count is the inode table, which is initialized and not written again for this workload (`sqlite-wal` column). The inode table of SQLiteRB is also only written once, but it accounts for a much smaller portion because SQLiteRB touches more logical space (as discussed in Section 5.2).

5.6 Discussion

By analyzing the interactions between applications, file systems and FTLs, we have learned the following lessons.

Being friendly to one rule is not enough: the SSD contract is multi-dimensional. Log-structured file systems such as F2FS are not a silver bullet. Although pure log-structured file systems conform to the aligned sequentiality rule well, it suffers from other drawbacks. First, it breaks apart application data structures, such as SQLite's B-tree structure, and thus breaks optimizations based on the structures. Second, log-structured file systems usually mix data of different death times and generate ghost data, making garbage collection very costly.

Although not perfect, traditional file systems still perform well upon SSDs. Traditional file systems have HDD-oriented optimizations that can violate the SSD contract. For example, locality grouping and stream optimization in ext4 were designed to avoid long seeks. Unfortunately, they now violate the grouping by death time rule of the SSD contract as we have shown. However, these traditional file systems continue to work well on SSDs, often better than the log-structured F2FS. This surprising result occurs because the HDD unwritten contract shares some similarity with the SSD contract. For example, HDDs also require large requests and strong locality to achieve good performance.

The complex interactions between applications, file systems, and FTLs demand tooling for analysis. There are countless applications and dozens of file systems, all of which behave differently with different inputs or configurations. The interactions between layers are often difficult to understand and we have often found them surprising. For example, running append-only Varmail on log-structured F2FS produces non-sequential patterns. To help deal with the diversity and complexity of applications and file systems, we provide an easy-to-use toolkit, *WiscSee*, to simplify the examination of arbitrary workloads and aid in understanding the performance robustness of applications and file systems on different SSDs. Practically, *WiscSee* could also be used to find the appropriate provisioning ratio before deployment, using visualizations such as the zombie curves.

Myths spread if the unwritten contract is not clarified. Random writes are often considered harmful for SSDs because they are believed to increase garbage collection overhead [31, 33, 62, 71]. As a result, pessimistic views have spread and systems are built or optimized based on this assumption [14, 24, 32, 60].

We advocate an optimistic view for random writes. Random writes often show low sustainable performance because benchmarks spread writes across a large portion of the logical space without discarding them and effectively create a large amount of valid data without grouping by death time [31, 58, 78], which would show as a large tail in our zombie curve. However, random writes can perform well as long as they produce a good zombie curve so that SSDs do not need to move data before reusing a flash block. For example, random writes perform well if they spread only across a small logical region, or data that is randomly written together is discarded together to comply with the rule of grouping by death time. Essentially, write randomness is not correlated with the rule of grouping by death time and garbage collection overhead.

Sequential writes, often enabled by log structure, are believed to reduce garbage collection overhead inside SSDs. Sequential writes across a large logical space, as often produced by benchmarks, show high sustainable performance because data that is written together will be later overwritten at the same time, implicitly complying with the rule of grouping by death time. However, as we have observed, log-structured writes in applications such as LevelDB and RocksDB often do not turn into repeated sequential writes across a large logical space, but sporadic sequential writes (often 2 MB) at different locations with data that die at different times. These writes violate the rule of grouping by death time and do not help garbage collection. In addition, log-structured file systems, such as F2FS, may not produce large sequential writes as we have often observed.

We advocate dropping the terms “random write” and “sequential write” for discussing SSD workloads regarding garbage collections. Instead, one should study death time and use zombie curves as a graphic tool to characterize workloads. The terms “random” and “sequential” are fine for HDDs as HDD performance is impacted only by the characteristic of two consecutive accesses [82]. However, SSDs are very different as their performance relies also on accesses that are long before the most recent ones. Such out-of-date and overly simplified terms bring misconceptions and suboptimal system designs for SSDs.

6. Related Work

Our paper uncovers the unwritten contract of SSDs and analyzes application and file system behaviors with the contract. We believe it is novel in several aspects.

Previous work often offers incomplete pictures of SSD performance [31, 58, 62, 71]. A recent study by Yadgar et al. [92] analyzes multiple aspects of SSD performance such as spatial locality, but omits critical components such as the concurrency of requests. The study by Lee et al. evaluates only the immediate performance of F2FS and real applications, but neglects sustainable performance problems [62]. Our investigation analyzes five dimensions of the SSD contract for both immediate and sustainable performance, providing a more complete view of SSD performance.

Previous studies fail to connect applications, file systems and FTLs. Studies using existing block traces [48, 54, 95], including the recent study by Yadgar et al. [92], cannot reason about the behaviors of applications and file systems because the semantics of the data is lost and it is impossible to re-create the same environment for further investigation. Another problem of such traces is that they are not appropriate for SSD related studies, because they were collected in old HDD environments which are optimized for HDDs. Studies that evaluate applications and file systems on black-box SSDs [59, 62] cannot accurately link the application and file system to hidden internal behaviors of the SSD. In addition, studies that benchmark black-box SSDs [31, 34] or SSD simulators [49] provide few insights about applications and file systems, which can use SSDs in surprising ways. In contrast, we conduct full-stack analysis with diverse applications, file systems, and a fully functioning modern SSD simulator, which allows us to investigate not only *what* happened, but *why* it happened.

7. Conclusions

Due to the sophisticated nature of modern FTLs, SSD performance is a complex subject. To better understand SSD performance, we formalize the rules that SSD clients need to follow and evaluate how well four applications (one with two configurations) and three file systems (two traditional and one flash-friendly) conform to these rules on a full-function SSD simulator that we have developed. This simulation-based analysis allows us to not only pinpoint rule violations, but also the root causes in all layers, including the SSD itself. We have found multiple rule violations in applications, file systems, and from the interactions between them. We believe our analysis here can shed light on design and optimization across applications, file systems and FTLs, and the tool we have developed could benefit future SSD workload analysis.

Acknowledgments

We thank the anonymous reviewers, Ant Rowstron (our shepherd), Joel Dedrick and Eunji Lee for their insightful comments. We thank the members of the ADSL for their valuable discussions. This material was supported by funding from Microsoft Jim Gray Systems Lab, NSF grants CNS-1419199, CNS-1421033, CNS-1319405, and CNS-1218405, DOE grant DE-SC0014935, as well as donations from EMC, Facebook, Google, Huawei, NetApp, Samsung, Seagate, Veritas, and VMware. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE, or other institutions.

References

[1] Amazon Web Service SSD Instance Store Volumes. `http://docs.aws.amazon.com/AWSEC2/latest/`

UserGuide/ssd-instance-store.html.

[2] Filebench. https://github.com/filebench/filebench.

[3] Google Cloud Platform: Adding Local SSDs. https://cloud.google.com/compute/docs/disks/local-ssd.

[4] Intel DC3500 Data Center SSD. http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-dc-s3500-series.html.

[5] Intel SSD X25-M Series. http://www.intel.com/content/www/us/en/support/solid-state-drives/legacy-consumer-ssds/intel-ssd-x25-m-series.html.

[6] LevelDB. https://github.com/google/leveldb.

[7] Linux Control Group. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[8] Linux Generic Block Layer. https://www.kernel.org/doc/Documentation/block/biodoc.txt.

[9] Micron 3D NAND Status Update. http://www.anandtech.com/show/10028/micron-3d-nand-status-update.

[10] Micron NAND Flash Datasheets. https://www.micron.com/products/nand-flash.

[11] Micron Technical Note: Design and Use Considerations for NAND Flash Memory. https://www.micron.com/~/media/documents/products/technical-note/nand-flash/tn2917.pdf.

[12] NVMe Specification. http://www.nvmexpress.org/.

[13] R Manual: Box Plot Statistics. http://stat.ethz.ch/R-manual/R-devel/RHOME/library/grDevices/html/boxplot.stats.html.

[14] RocksDB. https://rocksdb.org.

[15] RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[16] Samsung K9XXG08UXA Flash Datasheet. http://www.samsung.com/semiconductor/.

[17] Samsung V-NAND technology white paper. http://www.samsung.com/us/business/oem-solutions/pdfs/V-NAND_technology_WP.pdf.

[18] Sqlite. https://sqlite.org.

[19] Technical Commitee T10. http://t10.org.

[20] Technical Note (TN-29-07): Small-Block vs. Large-Block NAND Flash Devices. https://www.micron.com/~/media/documents/products/technical-note/nand-flash/tn2907.pdf/.

[21] Toshiba Semiconductor Catalog Mar. 2016. https://toshiba.semicon-storage.com/info/docget.jsp?did=12587.

[22] fstrim manual page. http://man7.org/linux/man-pages/man8/fstrim.8.html.

[23] White Paper of Samsung Solid State Drive TurboWrite Technology. http://www.samsung.com/hu/business-images/resource/white-paper/2014/01/Whitepaper-Samsung_SSD_TurboWrite-0.pdf.

[24] Wikipedia: write amplification. https://en.wikipedia.org/wiki/Write_amplification.

[25] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, pages 57–70, Boston, Massachusetts, June 2008.

[26] A. R. Alameldeen and D. A. Wood. Variability in Architectural Simulations of Multi-threaded Workloads. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture (HPCA-11)*, Anaheim, California, February 2003.

[27] K. Aneesh Kumar, M. Cao, J. R. Santos, and A. Dilger. Ext4 block and Inode Allocator Improvements. In *Ottawa Linux Symposium (OLS '08)*, volume 1, pages 263–274, Ottawa, Canada, July 2008.

[28] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[29] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, California, February 2008.

[30] S. Boboila and P. Desnoyers. Write Endurance in Flash Drives: Measurements and Analysis. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.

[31] L. Bouganim, B. T. Jónsson, P. Bonnet, et al. uFLIP: Understanding Flash IO Patterns. In *Proceedings of the fourth Conference on Innovative Data Systems Research (CIDR '09)*, Pacific Grove, California, January 2009.

[32] R. Branson. Presentation: Cassandra and Solid State Drives. https://www.slideshare.net/rbranson/cassandra-and-solid-state-drives.

[33] 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 2009 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance '09)*, pages 181–192, Seattle, Washington, June 2009.

[34] 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 *Proceedings of the 17th International Symposium on High Performance Computer Architecture (HPCA-11)*, pages 266–277, San Antonio, Texas, February 2011.

[35] J. B. Chen and B. Bershad. The Impact of Operating System Structure on Memory System Performance. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, Asheville, North Carolina, December 1993.

[36] P. M. Chen and D. A. Patterson. A New Approach to I/O Performance Evaluation–Self-Scaling I/O Benchmarks, Predicted I/O Performance. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of*

Computer Systems (SIGMETRICS '93), pages 1–12, Santa Clara, California, May 1993.

[37] Y. Chen, K. Srinivasan, G. Goodson, and R. Katz. Design Implications for Enterprise Storage Systems via Multi-dimensional Trace Analysis. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[38] Y. Cheng, F. Douglis, P. Shilane, G. Wallace, P. Desnoyers, and K. Li. Erasing Belady's Limitations: In Search of Flash Cache Offline Optimality. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 379–392, Denver, CO, 2016. USENIX Association.

[39] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[40] S. Cho, C. Park, Y. Won, S. Kang, J. Cha, S. Yoon, and J. Choi. Design Tradeoffs of SSDs: From Energy Consumptions Perspective. *ACM Transactions on Storage*, 11(2):8:1–8:24, Mar. 2015.

[41] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '10)*, pages 143–154, Indianapolis, IN, June 2010.

[42] M. Cornwell. Anatomy of a Solid-state Drive. *Commun. ACM*, 55(12):59–63, Dec. 2012.

[43] P. J. Denning. The Locality Principle. *Commun. ACM*, 48(7), July 2005.

[44] P. Desnoyers. Analytic Modeling of SSD Write Performance. In *Proceedings of the 5th International Systems and Storage Conference (SYSTOR '12)*, Haifa, Israel, June 2013.

[45] 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 36nd Annual International Symposium on Computer Architecture (ISCA '09)*, Austin, Texas, June 2009.

[46] M. Dunn and T. Feldman. Shingled Magnetic Recording Models, Standardization, and Applications. `http://www.snia.org/sites/default/files/Dunn-Feldman_SNIA_Tutorial_Shingled_Magnetic_Recording-r7_Final.pdf`.

[47] G. Gasior. The SSD Endurance Experiment. `http://techreport.com/review/27062`.

[48] 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 XIV)*, Washington, DC, March 2009.

[49] X. Haas and X. Hu. The Fundamental Limit of Flash Random Write Performance: Understanding, Analysis and Performance Modelling. *IBM Research Report, 2010/3/31, Tech. Rep*, 2010.

[50] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A File Is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[51] 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)*, Santa Clara, California, February 2015.

[52] C. Hellwig. Online TRIM/discard performance impact. `http://oss.sgi.com/pipermail/xfs/2011-November/015329.html`.

[53] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang. Performance Impact and Interplay of SSD Parallelism Through Advanced Commands, Allocation Strategy and Data Granularity. In *Proceedings of the International Conference on Supercomputing (ICS '11)*, Tucson, Arizona, May 2011.

[54] S. Jiang, L. Zhang, X. Yuan, H. Hu, and Y. Chen. S-FTL: An Efficient Address Translation for Flash Memory by Exploiting Spatial Locality. In *Proceedings of the 27th IEEE Symposium on Mass Storage Systems and Technologies (MSST '11)*, Denver, Colorado, May 2011.

[55] X. Jimenez, D. Novo, and P. Ienne. Wear Unleveling: Improving NAND Flash Lifetime by Balancing Page Endurance. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST '14)*, Santa Clara, California, February 2014.

[56] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho. The Multi-streamed Solid-State Drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '14)*, Philadelphia, PA, June 2014.

[57] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee. A Superblock-based Flash Translation Layer for NAND Flash Memory. In *Proceedings of the 6th ACM IEEE International Conference on Embedded software (EMSOFT '09)*, Grenoble, France, October 2009.

[58] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting Storage for Smartphones. *ACM Transactions on Storage*, 8(4):14, 2012.

[59] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar. Flashsim: A Simulator for Nand Flash-based Solid-State Drives. In *Proceedings of the First International Conference on Advances in System Simulation (SIMUL '09)*, Porto, Portugal, September 2009.

[60] J. Kreps. SSDs and Distributed Data Systems. `http://blog.empathybox.com/post/24415262152/ssds-and-distributed-data-systems`.

[61] Lanyue Lu and Thanumalayan Sankaranarayana Pillai and Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, pages 133–148, Santa Clara, California, February 2016.

[62] 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)*, Santa Clara, California, February 2015.

[63] S. Lee, D. Shin, Y.-J. Kim, and J. Kim. LAST: Locality-aware Sector Translation for NAND Flash Memory-based Storage Systems. *Operating Systems Review*, 42(6):36–42, Oct. 2008.

[64] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A Log Buffer-based Flash Translation Layer Using Fully-associative Sector Translation. *ACM Trans. Embed. Comput. Syst.*, 6(3), July 2007.

[65] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Comparing Public Cloud Providers. In *Proceedings of the Internet Measurement Conference (IMC '10)*, Melbourne, Australia, November 2010.

[66] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu. A Study of Linux File System Evolution. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, San Jose, California, February 2013.

[67] D. Ma, J. Feng, and G. Li. A Survey of Address Translation Technologies for Flash Memories. *ACM Comput. Surv.*, 46(3):36:1–36:39, Jan. 2014.

[68] A. Maislos. A New Era in Embedded Flash Memory. Presentation at Flash Memory Summit. `http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2011/20110810_T1A_Maislos.pdf`.

[69] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, L. Vivier, and B. S.A.S. The New Ext4 Filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, July 2007.

[70] J. Meza, Q. Wu, S. Kumar, and O. Mutlu. A Large-Scale Study of Flash Memory Failures in The Field. In *Proceedings of the 2015 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '15)*, Portland, Oregon, June 2015.

[71] 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 Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[72] V. Mohan, T. Siddiqua, S. Gurumurthi, and M. R. Stan. How I Learned to Stop Worrying and Love Flash Endurance. In *2nd Workshop on Hot Topics in Storage and File Systems (HotStorage '10)*, Boston, Massachussetts, June 2010.

[73] 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 XIV)*, Salt Lake City, Utah, March 2014.

[74] 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 Trans. Embed. Comput. Syst.*, 7(4):38:1–38:23, Aug. 2008.

[75] D. Park, B. Debnath, and D. Du. CFTL: A Convertible Flash Translation Layer Adaptive to Data Access Patterns. In *Proceedings of the 2010 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '10)*, pages 365–366, New York, NY, June 2010.

[76] D. Park and D. H. Du. Hot Data Identification for Flash-Based Storage Systems Using Multiple Bloom Filters. In *Proceedings of the 27th IEEE Symposium on Mass Storage Systems and Technologies (MSST '11)*, Denver, Colorado, May 2011.

[77] T. S. Pillai, R. Alagappan, L. Lu, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Application Crash Consistency and Performance with CCFS. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, Santa Clara, California, February 2017.

[78] M. Polte, J. Simsa, and G. Gibson. Enabling Enterprise Solid State Disks Performance. In *Proceedings of the First Workshop on Integrating Solid-state Memory into the Storage Hierarchy (WISH), held in conjunction with ASPLOS*, 2009.

[79] S. Robinson. *Simulation: The Practice of Model Development and Use*. Palgrave Macmillan, 2014.

[80] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992.

[81] R. H. Saavedra and A. J. Smith. Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes. *IEEE Transactions on Computers*, 44(10):1223–1235, 1995.

[82] S. W. Schlosser and G. R. Ganger. MEMS-based Storage Devices and Standard Disk Interfaces: A Square Peg in a Round Hole? In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 87–100, San Francisco, California, April 2004.

[83] B. Schroeder, R. Lagisetty, and A. Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, page 67, Santa Clara, California, February 2016.

[84] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 71–84, San Diego, California, June 2000.

[85] A. L. Shimpi. The Seagate 600 and 600 Pro SSD Review. `http://www.anandtech.com/show/6935/seagate-600-ssd-review`.

[86] J.-Y. Shin, Z.-L. Xia, N.-Y. Xu, R. Gao, X.-F. Cai, S. Maeng, and F.-H. Hsu. FTL Design Exploration in Reconfigurable High-performance SSD for Server Applications. In *Proceedings of the International Conference on Supercomputing (ICS '09)*, pages 338–349, Yorktown Heights, NY, June 2009.

[87] K. Smith and M. I. Seltzer. File System Aging. In *Proceedings of the 1997 Sigmetrics Conference*, Seattle, WA, June 1997.

[88] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[89] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad. Efficient MRC Construction with SHARDS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, Santa Clara, California, February 2015.

[90] J. Wires, S. Ingram, Z. Drudi, N. J. A. Harvey, and A. Warfield. Characterizing Storage Workloads with Counter Stacks. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 335–349, Broomfield, Colorado, October 2014.

[91] S. C. Woo, M. Ohara, E. Torrie, J. P. Shingh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[92] G. Yadgar and M. Gabel. Avoiding the Streetlight Effect: I/O Workload Analysis with SSDs in Mind. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '16)*, Denver, CO, June 2016.

[93] Yiying Zhang, Leo Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. De-indirection for Flash-based SSDs with Nameless Writes. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[94] J. Zhang, J. Shu, and Y. Lu. ParaFS: A Log-structured File System to Exploit the Internal Parallelism of Flash Devices. In *Proceedings of the USENIX Annual Technical Conference (USENIX '16)*, pages 87–100, Denver, CO, June 2016.

[95] Y. Zhou, F. Wu, P. Huang, X. He, C. Xie, and J. Zhou. An Efficient Page-level FTL to Optimize Address Translation in Flash Memory. In *Proceedings of the EuroSys Conference (EuroSys '15)*, Bordeaux, France, April 2015.