Beyond Storage Interfaces: Finding and Exploiting Unwritten Contracts In Storage Devices

By

Jun He

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2019

Date of final oral examination: December 13, 2018

To my family

Acknowledgments

I would not be able to get this Ph.D. without the support from my advisors, committee members, colleagues, collaborators, and family members, to whom I would like to express my gratitude.

First, I would like to thank my advisors, Andrea and Remzi, for their tireless guidance. Some people are glad they have one good advisor; I have two fascinating advisors. Andrea and Remzi pursue high quality in research, which makes me dive deep to the problems. They asked questions in great details in our weekly meetings to ensure the quality of the work and to discover exciting insights. The questions often made me nervous (later I learned that "I don't know" was a totally acceptable answer) and also encouraged me to ask similar questions while working on the problems by myself. Andrea and Remzi give the best advice. When I was stuck, I would talk to them and feel enlightened. While working with them, I learned to find and validate new idea by concrete data from measurements; I also learned to always build systems with concrete code. When it came to presenting our work, Andrea and Remzi took it very seriously and gave valuable feedback. Andrea was a "paper architect" (according to Remzi): she gave constructive comments on how a paper should be organized and how a specific idea should be presented to highlight strengths and make them easy to understand. Remzi gives excellent feedback for oral presentations. I enjoyed the moments when he precisely

articulated the problems in presentations and provided better ways of presenting (sometimes in humorous ways). I thank Andrea and Remzi for a fruitful and happy Ph.D. journey. I learned from the best.

I want to thank my thesis committee members: Michael Swift and Peter Chien. Michael, who was also in my prelim committee, asked questions that provoked deep thoughts on my work and expanded the scope of the idea. Peter, who is a world-class expert on Design of Experiments (DOE), helped my first project, which used techniques in DOE, by giving great suggestions and introducing me a great statistics student, who later became a co-author of my paper. Since then, I have been obsessed with DOE and took Peter's class on this subject.

I am grateful for having smart and productive colleagues at school: Kan Wu, Sudarsun Kannan, Duy Nguyen, Lanyue Lu, Zev Weiss, Thanh Do, Aishwarya Ganesan, Leon Yang, Thanumalayan Pillai, Tyler Caraza-Harter Yupu Zhang, Vijay Chidambaram, Yuvraj Patel, Jing Liu, Ed Oakes, Ram Alagappan, Samer Al-Kiswany, Suli Yang, Leo Arulraj, Yiying Zhang, Dennis Zhou, and Ao Ma. In particular, I would like to thank Kan Wu for his help with my last project. The discussion with Kan brought me new and deep understanding of the problem; his help with various aspects of the project significantly speeded up the progress, especially when I was busy preparing for job interviews. I would like also to thank Sudarsun, who helped me on two projects. Sudarsun was very knowledgable, encouraging, and willing to help. I am grateful to Duy Nguyen, who was a co-author of my first paper in Wisconsin. I still clearly remember the all-day discussions during a Spring break with Duy in his apartment about applying statistics to systems research; the interdisciplinary conversations with Duy, a Statistics student, was very inspiring. I thank Lanyue Lu for many joyful lunches together. I thank Zev Weiss for answering many kernel questions (and many other interesting discussions); it was very nice to have a kernel expert sitting next to me. I thank Thanh Do for his tips and great help in my job hunting.

I had an excellent opportunity to work with great researchers at Microsoft Jim Gray Systems Lab: Alan Halverson, Jignesh Patel, Willis Lang, Karthik Ramachandra, Kwanghyun Park, Justin Moeller, Rathijit Sen, Brian Kroth, and Olga Poppe. I thank Alan Halverson for his excellent suggestions and discussions; I learned a lot about production databases from him. I thank Jignesh for many technical discussions and his constructive questions, which encouraged me to deeper understandings of various problems.

I also had a great opportunity to collaborate with and learn from John Bent, Gary Grider, and Aaron Torres, who were also my mentors at Los Alamos National Lab; I thank them for the collaboration, which I enjoyed. In particular, I would like to thank John Bent, who is Andrea and Remzi's second student, for multiple life-changing opportunities. John is always fun to talk to, either about life or work. I was very fortunate to have John as my official marriage witness. I thank Aaron for many valuable discussions and comments about my work. I thank Gary for truly inspiring feedback and his support for my work.

Finally, I am grateful to my family. My parents believe that knowledge can bring a better life; they are extremely supportive of my study since I was young. When I decided to pursue a Ph.D. in the US, they would sell the house to support me financially if necessary. To allow me to focus on my study, they tried their best to create the most comfortable environment for me. After being a parent myself, I understand how hard parenting is. Also, I would like to thank Yan for being a caring sister from our childhood to adulthood. I want to thank my wife Jingjun for her love and her support for all my dreams. She is smart, curious and sympathetic. We met online when grading each other's GRE writing; after we got married, she continued to be my advisor at home: she listened to my research to allow me to clarify my thoughts. I thank her for taking care of our son

Charlie, especially in countless early mornings, so that I could have more energy to keep working the next day. Finally, I thank Charlie and the son to be born soon for bringing joy and laughter to my life.

Contents

Abstract

As data becomes bigger and more relevant to our lives, high-performance data processing systems are more critical. In such systems, storage devices play an essential role as they store data and feed data for the processing; efficient interactions between the software and the storage devices are important for achieving high performance.

Efficient interactions are beyond the storage interfaces. Storage devices present a block interface; clients must comply with the specifications of the interfaces, which we call "written contract", to communicate with the devices correctly. However, complying with the written contract is not enough to achieve high performance; clients of storage devices must follow a set of implicit rules, which we call "unwritten performance contracts", to achieve high performance. The specific contract of a device depends on its internal architecture. Violating the unwritten contract lowers the performance of the data processing system.

In this dissertation, we start by finding violations of the unwritten contracts. First, we study the methodologies for finding violations of the HDD unwritten contract. Specifically, we explore using statistical methods (e.g., Latin hypercube sampling and Sensitivity Analysis) to find violations of the HDD unwritten contract in file system block allocators. The exploration leads to discoveries and fixes of four design issues in a popular file system, Linux ext4.

In addition to finding violations of the HDD unwritten contract, we continue to study violations of the SSD unwritten contract. We begin by formalizing the SSD unwritten contract, which includes five rules that one must follow to achieve high performance on SSDs. We then conduct a vertical full-stack analysis of multiple popular applications and file systems on top of a detailed SSD simulator that we build; the analysis shows whether the combinations of applications and file systems violate the rules of the unwritten contract. As a result, we make numerous interesting observations that could shed light upon the future storage system designing.

Finally, we exploit the unwritten contract of SSDs to build a cost-effective and high-performance data processing system. As the SSD unwritten contract suggests, SSDs offer high internal I/O parallelism, which enables high bandwidth and low latency on SSDs. The high performance of SSDs brings new opportunities for a system to rely more on I/O and less on expensive memory, reducing the cost of the system. To this end, we propose Tiny-Memory Processing Systems (TMPS), which use a small amount of RAM, along with fast and cheap SSDs. The resulting system is cost-effective thanks to the limited size of expensive RAM. The major challenge of the TMPS approach is to reduce I/O traffic as most data needs to be loaded from SSDs, which offer lower data bandwidth than memory. To study the TMPS approach, we build a complete search engine that employs careful designs in data layout, early pruning, prefetching and space/traffic tradeoffs. When compared with the state-of-the-art search engine in a tiny-memory configuration, our search engine reduces I/O amplification by up to 3.2 times, query latency by up to 16 times, and increases the query throughput by up to 2.9 times.

1 Introduction

Data has been transforming many aspects of our lives. Search engines [7, 15], which collect and organize data from the web, allow us to quickly and accurately retrieve information; data-driven recommendation systems present contents that we are very likely to enjoy [2, 18, 30]; ride-share services [25, 48], which match the real-time data of riders and drivers, significantly ease our transportation. Data will continue to enhance our lives in the days yet to come in health care, business, transportation, scientific research, entertainment and so on.

To support and continue supporting the massive transformation in our lives, data is getting bigger. More data is created automatically or manually and captured by a variety of sensors, such as cameras and IoT (Internet of Things) devices. The size of global data will increase by five times from 2018 to 2025 and data will be even more critical for our lives[10].

In order to process the large volume of data to solve critical problems, a data processing system must store and retrieve data quickly; the storage stack, which involves storage hardware and software, plays a vital role in such high-performance data processing systems, as the storage stack is responsible for efficiently transferring data between the CPU and the storage hardware.

At the bottom of the storage stack are storage devices such as Hard Disk Drives (HDDs) and Solid State Drives (SSDs). HDDs have dominated the market for decades, and they will continue to play a critical role in the future due to its low cost. SSDs, which are more expensive than HDDs, become more popular in recent years thanks to their high performance. HDDs and SSDs will be the most important players in the storage market in the foreseeable future [16].

Storage devices expose interfaces that allow clients to specify operations (e.g., read, write, trim) and the range of addresses to operate on [31, 40, 43]. We call the specifications of such interfaces the *written contracts* of the storage devices, as it is indeed presented in the form of written documents. The written contracts are easy to comply with because the specifications clearly state the contract and the device will report errors when the contract is violated. For example, the SATA specification [40] clearly defines the format of commands; if a client sends a command in an incorrect format, the devices will send back an error message.

However, understanding and complying with the written contract is not enough; beyond the written contracts, there exist multiple *unwritten contracts*, which must be complied with to achieve high performance, power efficiency, high reliability, or other desired properties. A particular unwritten contract depends on the storage media and the architecture of the devices. For example, Schlosser and Ganger state in the "unwritten performance contract" of HDDs [143] that "nearby accesses are better than farther accesses" because moving disk head farther takes more time. As another example, due to the limited life span of flash, one should avoid access patterns that trigger overuses of flash-based SSDs; an "unwritten reliability contract" should describe such patterns. In this dissertation, we focus on the unwritten performance contracts, which we refer to as "unwritten contracts" hereafter for brevity. Violations of the unwritten performance contracts will not lead to errors (which can be easily detected) but to performance degradation.

In this dissertation, we investigate and find violations of the unwritten contracts of HDDs and SSDs, both of which are widely deployed in modern systems, supporting diverse workloads range from personal document editors to distributed data analytics. HDDs and SSDs present distinct unwritten contracts due to the vastly different internal structures. By investigating and contrasting these contracts, we can better understand the factors that impact performance and how the systems should be restructured.

After understanding the unwritten contracts, we study how we should exploit them to maximize their performance. Our study of the SSD unwritten contract shows that the contract is often violated, leading to insufficient systems. Therefore, we continue to study how one should exploit the SSD unwritten contract, specifically the high internal I/O parallelism suggested by the contract, to implement high-performance and cost-effective systems by reducing the memory demand.

We believe that our studies on finding violations and exploiting the unwritten contracts of storage devices, along with the tools and systems that we build, will boost harmonious interactions among the storage layers and significantly improve the performance of data processing systems that support our day-to-day life.

1.1 Do contractors comply?

In the first part of this dissertation, we investigate whether applications and file systems comply with the contracts of HDDs and SSDs, separately. Both HDDs and SSDs support a wide range of systems and thus are worth studying. HDDs have been and will continue to be widely deployed due to its low cost; flash-based SSDs have become increasingly popular thanks to its high performance and decreasing costs. Another reason to study both HDDs and SSDs is that they are distinct: HDDs employ rotatory magnetic platters but SSDs employ static flash. Comparing the unwritten contracts of these distinct devices will allow us to understand the extent of the unwritten contracts and how to discover unwritten contracts when new types of devices come.

For HDDs, we present *Chopper*, a tool that efficiently explores the vast input space of file system policies to find behaviors that lead to violations of the HDD contracts. We focus specifically on block allocation, as unexpected poor layouts can lead to high tail latencies in critical large-scale systems. Our approach utilizes sophisticated statistical methodologies, based on Latin Hypercube Sampling (LHS) and sensitivity analysis, to explore the search space efficiently and diagnose intricate design problems. We apply *Chopper* to study the behavior of two file systems, and to study Linux ext4 in depth. We identify four internal design issues in the block allocator of ext4 which form a large tail in the distribution of layout quality. By removing the underlying problems in the code, we cut the size of the tail by an order of magnitude, producing consistent and satisfactory file layouts that reduce data access latencies.

For SSDs, we perform a detailed vertical analysis of application performance atop a range of modern file systems and SSD FTLs. We formalize the "unwritten contract" of SSDs, 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.

1.1.1 Hard Disk Drives

Hard Disk Drives (HDDs) have been the most important storage devices for decades; they will continue to be popular in the future. Cost is the most important factor in the continuing adoption of HDDs. High-capacity HDDs are much cheaper than high-capacity SSDs; therefore, such HDDs take less shelf space at a lower price, which makes them suitable for data centers, cloud storage providers and backup systems.

HDDs present its unique unwritten contract, summarized by Schlosser and Ganger [143]. The unwritten contract of HDDs states that nearby accesses on the address space are faster and sequential accesses are faster than random ones. The unwritten contract comes from the physical structure of HDDs: HDDs must physically move disk heads to access data on different locations. Due to the popularity of HDDs, the unwritten contract of HDDs is crucial, as violations of the contract can lead to significant performance loss on a large population of systems.

Local file systems, such as Linux ext4 and XFS, sit on top of storage devices, such as HDDs, and interact directly with the devices; therefore, local file systems play an essential part in complying with the unwritten contract of the devices. Violations that come from the local file systems can significantly degrade the performance of the storage stack. As a result, we must find the violations from the local file systems. However, finding violations is challenging because the input space of file systems is vast, which makes it hard to identify input that triggers problematic behaviors.

We present *Chopper*, a tool that enables developers to discover (and subsequently repair) violations in local file systems. *Chopper* currently focuses on the HDD unwritten contract due to the popularity of HDDs. We use *Chopper* to investigate a file system component that is critical for contract compliance: block allocator, which can reduce file system performance by one or more orders of magnitude on hard disks due to violations of the HDD unwritten contract [8, 70, 138]. With *Chopper*, we show how to find behaviors that violate the unwritten contract of HDDs, and then how to fix them (usually through simple file-system repairs).

The key and most novel aspect of *Chopper* is its use of advanced statistical techniques to search and investigate an infinite performance space systematically. Specifically, we use Latin hypercube sampling [119] and sensitivity analysis [142], which have been proven effective in the investigation of many-factor systems in other applications [95, 124, 141]. We show how to apply such advanced techniques to the domain of file-system performance analysis, and in doing so make finding violations of the unwritten contracts tractable.

We use *Chopper* to analyze the block allocators of Linux ext4 and XFS, and then delve into a detailed analysis of ext4 as its behavior is more complex and varied. We find four subtle violations in ext4, including behaviors that spread sequentially-written files over the entire disk volume, grossly violating the HDD unwritten contract. We also show how simple fixes can remedy these problems, resulting in an order-of-magnitude improvement in the tail layout quality of the block allocator. *Chopper* and the ext4 patches are publicly available at:

`research.cs.wisc.edu/adsl/Software/chopper`

1.1.2 Solid State Drives

Solid State Drives (SSDs) are emerging, despite its higher cost than HDDs. SSDs have higher bandwidth, higher IOPS and lower latency than HDDs. SSDs perform much better than HDDs for random data accesses because SSDs do not need to move mechanical parts to access data. As a result, SSDs are suitable for OLTP (Online Transactional Processing), data analytics, data caching and many other critical data processing systems.

Due to the increasing popularity of SSDs, it is crucial to analyze to understand their properties and interactions with other layers of the storage stack. 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 [22, 35]; new file systems, such as F2FS [110], have been created for 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 [143].

We present five rules of the SSD unwritten contract 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 [66] and garbage collection [67, 121].

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 [118], XFS [150], and the flash-friendly F2FS [110]. 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 [89, 98, 104, 128]. 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/`

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" [121]).

1.2 How to exploit the unwritten contracts?

From our studies of the SSD unwritten contract, we find that the unwritten contract is often violated, leading to inefficient usage of SSDs. We believe that SSDs, which have a smaller performance gap to RAM than HDDs, could bring much more benefits to data processing systems if they are exploited better. Specifically, we think that the high internal I/O parallelism, which does not exist in HDDs, could enable new system designs that could reduce the memory needed and build cost-effective and high-performance data processing systems.

Due to changes in technology – specifically, the vast and growing performance difference between hard drives and main memories – the effectiveness of automatic paging by the VM system has been rendered largely ineffective. As a result, application and service implementation has moved, in many performance-critical cases, to a one-level memory approach: keep all relevant data in memory, and thus deliver predictable high performance. This "all memory" approach is surprisingly common, driving the design of distributed storage systems [126], processing infrastructures [6], databases [34], and many other applications and services.

Memory-only approaches are not a panacea, introducing serious cost issues – DRAM costs dollars per GB, whereas hard drives fractions of a penny – as well as energy demands. Furthermore, in the burgeoning era of cloud computing, renting systems with high amounts of memory can be cost prohibitive, costing orders of magnitude more than smaller-memory instances [1]. Thus, a question arises: can traditionally memory-only or large-memory applications and services be realized in a different more cost-effective manner, without losing their performance advantages?

We believe the answer is (in some cases) yes. Specifically, the advent of high-speed solid-state storage devices [38, 46] has radically altered the performance gap between main memory and persistent storage, thus raising the possibility that storage-oriented approaches can once again be successful. In current technology, flash-based SSDs [38, 46] are readily available (and thus our focus). Flash-based SSDs offer very high internal I/O parallelism, which brings high bandwidth and low latency and enables new design opportunities. We exploit the high internal I/O parallelism to design a cost-effective and high-performance system.

The extreme point in the design spectrum that we study is something we call an "tiny-memory" design, in which a traditional all-memory/large-memory approach is supplanted by a system with meager memory resources and an SSD-based storage system; we call such systems Tiny-

Memory Processing Systems (TMPS). The hypothesis underlying this approach is that the high bandwidth provided by modern SSDs [38, 46] can be effectively exploited to achieve application performance goals without requiring copious amounts of main memory in addition.

Tiny-memory processing systems can significantly reduce the monetary costs of critical data processing systems by reducing memory demands. The price difference between an all-memory and a tiny-memory system is significant; for example, renting a 512-GB all-memory instance costs six times more than a 1-GB memory instance with NVMe SSDs on Amazon Web Service (the cost of the 1-GB memory and SSD instance is estimated). The cost of search engines can be significantly reduced when they are deployed on tiny-memory systems as their dataset sizes are large (and growing). For example, the Wikimedia organization, which owns many wiki sites including Wikipedia, owns 6,500 shards of search indices ranging from a few MBs to 50 GBs each [12]. Github maintains search indices of 96 million code repositories [42]. The index sizes of web-scale search engines, such as Google and Bing, are hundreds of exabytes [17].

To understand tiny-memory processing systems, we perform a detailed case study, which serves as the technical focus of this paper: the modern search engine [4, 5, 12]. We focus on search engines for two reasons. First, they are important; search engines are widely used in many industrial and scientific settings today, including popular open-source offerings such as Elastisearch [12] and Solr [5]. Second, they are challenging: search engines demand high throughput and low latency and seemingly, it should be hard to match the performance of the commonly used in-memory/cache-based approach with a tiny-memory system.

In this dissertation, we present the design, implementation, and evaluation of Vacuum Search, a tiny-memory search engine. Vacuum Search reorganizes traditional search data structures so as to be amenable to flash, thus using little memory while exploiting the parallel bandwidth that modern SSDs provide. We propose several techniques to make the tiny-memory search engine possible. First, we propose a technique called data vacuuming, which produces storage-oriented data layout and significantly reduces read amplification. Second, we propose two-way cost-aware pruning by bloom filters to reduce I/O for phrase queries. Third, we use adaptive prefetch to reduce latency and improve I/O efficiency. Fourth, we trade space on flash for less I/O; for example, we compress documents individually, which consumes more space than compression in groups but allows us to retrieve documents with less overall I/O.

We show that Vacuum, which exploits the high internal I/O parallelism (the property behind the request scale rule in the unwritten contract), performs significantly better than a state-of-the-art open-source search engine (Elastisearch) in a tiny-memory system, where only a small fraction of data is stored in memory. Vacuum delivers higher query throughput (up to a factor of three) and lower tail latencies (by a factor of sixteen). In multiple cases, Vacuum matches the performance of the popular search engine, while using orders of magnitude less memory.

1.3 Contributions and Hightlights

We believe this dissertation makes the following contributions.

Finding Violations of the Unwritten Contract of HDDs.

- We propose to use statistical methods (e.g., Latin Hypercube Sample and sensitivity analysis) to efficiently explore the vast input space of file system policies, in order to find violations of the HDD contract that lead to performance problems.

- The analysis tool, *Chopper*, that we built based on the statistical methods, finds multiple design problems and bugs in one of the most popular file systems, Linux ext4.

Finding Violations of the Unwritten Contract of SSDs.

- We formalize the rules of the SSD unwritten contract. As SSDs may present different rules due to their internal designs, we study the rules individually, which allows future system designs to mix and match these rules to evaluate their new systems.

- To find violations of the SSD contract, we perform a vertical analysis of the storage stack, involving four critical applications, three popular file systems, and a detailed SSD simulator.

- From the vertical analysis, we made 24 observations; some of them are surprising, and some are re-assuring. For example, we find that the Linux page cache design, which was designed in the HDD era, limits performance when running with an SSD.

- As the interactions among the layers of the storage stack are complex, we have built tools to allow designers to easily investigate if their systems comply with the SSD contract.

Exploiting the Unwritten Contract of SSDs

- Through the case study of search engines, we demonstrate that one significantly reduces the cost of critical large-scale data processing systems by replacing RAM with fast and cheap flash.

- We identify that the high data bandwidth of RAM is largely wasted. By offloading data to flash, we can build more flexible and cost-effective systems.

- We find that fast flash makes cache misses tolerable, which allows us to build a system with new assumptions that memory is tiny and cache misses are the common case.

- To build the search engine that performs well with tiny memory, we propose four techniques in data layout, early pruning, prefetching and space/traffic tradeoffs. As a result, we built a complete search engine that reduces query latency by up to 16 times and increases query throughput by up to 3 times.

1.4 Overview

In Chapter 2 and Chapter 3, we find violations of the unwritten contracts of HDDs and SSDs, respectively. We start by adopting statistics tools, such as sensitivity analysis, to find violations of the HDD unwritten contract in Chapter 2. We find four design issues in the Linux ext4 file system that violate the HDD unwritten contract and could lead to long tail latency in large-scale systems. After studying HDDs, we continue to formalize the SSD unwritten contract and find violations to the contract in Chapter 3. We find violations in four applications and three file systems.

In Chapter 4, we exploit the unwritten contract of SSDs to achieve high performance with only tiny memory. Specifically, by careful designs in the data layout, early pruning, prefetching and space/IO tradeoffs, we build a complete search engine that achieves three times higher query throughput than the state-of-the-art search engine, Elasticsearch.

In Chapter 5, we introduce the related work. In Chapter 6, we introduce the future work and the lessons learned, and then we conclude this dissertation.

2 Finding Violations of the HDD Contract

HDDs have been the major storage devices for decades and will be a significant player in the market for a long time. As a result, understanding the unwritten contract of HDDs and finding its violations to improve the performance of HDD-based systems are critical.

Thanks to the low cost of HDDs, they are widely deployed at large-scale distributed systems, where violations of the HDD unwritten contract could lead to significant performance problems. The performance, especially the performance of interactive services, of large-scale distributed systems, is significantly impacted by tail latency, which is the longest latency of sub-requests of a big request. The impact of the tail latency is significant because the big request is not completed until all sub-requests are completed. As a critical part of a distributed system, distributed storage systems may contribute significantly to tail latency if they misbehave. The foundation of a distributed storage system is the local file system (e.g., Linux ext4 and XFS), on which the performance of the distributed system depends. As a result, violations of the HDD unwritten contract from local file systems could contribute to the tail latency of large distributed systems; finding violations from local file systems is crucial.

Contract violations increase local data access latency, which will eventually increase tail latency of distributed systems. In this chapter, we focus on finding violations in file-system block allocators, which determine the layout of data on storage devices. The layout of data is critical for the HDD unwritten contract, which specifies that the distance between data chunks accessed consecutively is the key.

Finding violations of the HDD unwritten contract in block allocators is challenging because the input space of block allocators is vast and it is difficult to identify a problematic input that triggers misbehaviors. To address this challenge, we propose to apply statistical tools, such as Latin Hypercube Sampling and Sensitivity Analysis, to efficiently explore the vast input space in order to find violations. We apply such tools to two popular file systems, Linux ext4 and XFS. We find four critical design issues in Linux ext4 that violate the unwritten contract of HDDs. Removing the violations allows ext4 to produce consistent and satisfactory file layouts that reduce data access latencies.

This chapter is organized as follows. We first describe the internal structure of HDDs in Section 2.1 and the unwritten contract of HDDs in Section 2.2. Section 2.3 introduces the experimental methodology. In Section 2.4, we evaluate ext4 and XFS as black boxes and then go further to explore ext4 as a white box since ext4 shows more problems than XFS; we present detailed analysis and fixes for internal allocator design issues of ext4. Section 2.5 concludes this chapter.

2.1 HDD Background

An HDD consists of multiple components: platters, a spindle, and disk heads. Figure 2.1 shows a simplified structure of an HDD. A platter is a circular magnetic surface on which data is stored persistently. Data is recorded on the surface of platters in concentric circles of sectors; the concentric circles are referred to as tracks. Platters are bound together around the spindle, which is attached to a motor that spins the platters. Disk

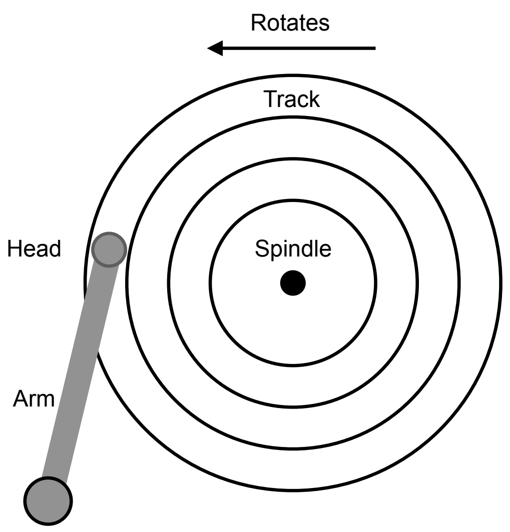


Figure 2.1: **HDD Internal Structure.** *This figure only shows one platter and a very small portion of tracks on the platter.*

heads are attached to disk arms, which position disk heads to specific locations of the platters; each side of a platter has their own disk head.

Disk heads read and write data from the surfaces of platters, which incur mechanical movements. To access a sector of data on a track, the disk arm moves the disk head to the desired track; this movement is called a "seek". After the seek, the disk head waits until the desired sector is rotated underneath the head and then reads or writes the sector; the time of waiting is called "rotational delay".

2.2 HDD Unwritten Contract

The physical structure of HDDs determines their performance properties. A significant portion of the data access time comes from seek time and rotational delay. If the sectors of data to be accessed consecutively are farther apart on the platter, the seek time and rotational delay are longer because disk heads have to move a longer distance and platters have to rotate more.

According to these properties of HDDs, Schlosser and Ganger [143] summarize the unwritten contract of HDDs. The unwritten contract states that two data accesses near one-another within the drive's address space are faster than two accesses that are far apart; sequential accesses are faster than random accesses.

Because the address of data is ultimately important in the unwritten contract, the block allocator, which determines the location of data, is a key for the contract: if data accessed at the same time are placed far apart on the disk, the contract will be violated.

2.3 Diagnosis Methodology

We now describe our methodology for discovering violations of the HDD unwritten contract, particularly as related to block allocation in local file systems. The file system input space is vast, and thus cannot be explored exhaustively; we thus treat each file system experiment as a simulation, and apply a sophisticated sampling technique to ensure that the large input space is explored carefully.

In this section, we first describe our general experimental approach, the inputs we use, and the output metric of choice. We conclude by presenting our implementation.

2.3.1 Experimental Framework

The Monte Carlo method is a process of exploring simulation by obtaining numeric results through repeated random sampling of inputs [140, 142, 149]. Here, we treat the file system itself as a simulator, thus placing it into the Monte Carlo framework. Each run of the file system, given a set of inputs, produces a single output, and we use this framework to explore the file system as a black box.

Each input factor X_i $(i = 1, 2, ..., K)$ (described further in Section 2.3.2) is estimated to follow a distribution. For example, if small files are of particular interest, one can utilize a distribution that skews toward small file sizes. In the experiments of this paper, we use a uniform distribution for fair searching. For each factor X_i, we draw a sample from its distribution and get a vector of values $(X_i^1, X_i^2, X_i^3, .., X_i^N)$. Collecting samples of all the factors, we obtain a matrix M.

$$M = \begin{bmatrix} X_1^1 & X_2^1 & ... & X_K^1 \\ X_1^2 & X_2^2 & ... & X_K^2 \\ ... & & & \\ X_1^N & X_2^N & ... & X_K^N \end{bmatrix} \qquad Y = \begin{bmatrix} Y^1 \\ Y^2 \\ ... \\ Y^N \end{bmatrix}$$

Each row in M, i.e., a *treatment*, is a vector to be used as input of one run, which produces one row in vector Y. In our experiment, M consists of columns such as the size of the file system and how much of it is currently in use. Y is a vector of the output metric; as described below, we use a metric that captures how much a file is spread out over the disk called *d-span*. M and Y are used for exploratory data analysis.

The framework described above allows us to explore file systems over different combinations of values for uncertain inputs. This is valuable for file system studies where the access patterns are uncertain. With the framework, block allocator designers can explore the consequences of design decisions and users can examine the allocator for their workload.

In the experiment framework, M is a set of treatments we would like to test, which is called an *experimental plan* (or *experimental design*). With a large input space, it is essential to pick input values of each factor and organize them in a way to efficiently explore the space in a limited number of runs. For example, even with our refined space in Table 2.1 (introduced in detail later), there are about 8×10^9 combinations to explore. With an overly optimistic speed of one treatment per second, it still would take 250 compute-years to finish just one such exploration.

Latin Hypercube Sampling (LHS) is a sampling method that efficiently explores many-factor systems with a large input space and helps discover surprising behaviors [97, 119, 142]. A *Latin hypercube* is a generalization of a *Latin square*, which is a square grid with only one sample point in each row and each column, to an arbitrary number of dimensions [69]. LHS is very effective in examining the influence of each factor when the number of runs in the experiment is much larger than the number of factors. It aids visual analysis as it exercises the system over the entire range of each input factor and ensures all levels of it are explored evenly [140]. LHS can effectively discover which factors and which combinations of factors have a large influence on the response. A poor sampling method, such as a completely random one, could have input points clustered in the input space, leaving large unexplored gaps in-between [140]. Our experimental plan, based on LHS, contains 16384 runs, large enough to discover subtle behaviors but not so large as to require an impractical amount of time.

2.3.2 Factors to Explore

	Factor	Description	Presented Space
FS	DiskSize	Size of disk the file system is mounted on.	1,2,4,...,64GB
	UsedRatio	Ratio of used disk.	0, 0.2, 0.4, 0.6
	FreeSpaceLayout	Small number indicates high fragmentation.	1,2,...,6
OS	CPUCount	Number of CPUs available.	1,2
Workload	FileSize	Size of file.	8,16,24,...,256KB
	ChunkCount	Number of chunks each file is evenly divided into.	4
	InternalDensity	Degree of sparseness or overwriting.	0.2,0.4,...,2.0
	ChunkOrder	Order of writing the chunks.	permutation(0,1,2,3)
	Fsync	Pattern of `fsync()`.	****, *=0 or 1
	Sync	Pattern of `close()`, `sync()`, and `open()`.	***1, *=0 or 1
	FileCount	Number of files to be written.	1,2
	DirectorySpan	Distance of files in the directory tree.	1,2,3,...,12

Table 2.1: **Factors in Experiment.**

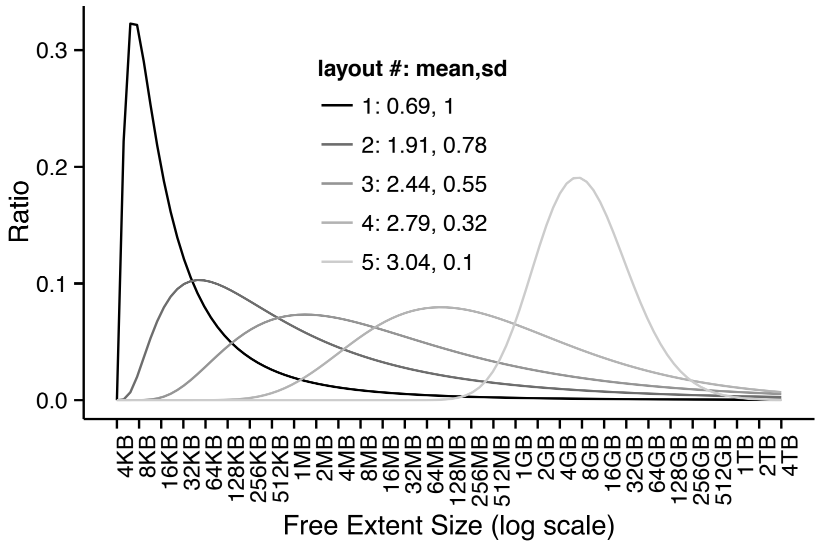


Figure 2.2: **LayoutNumber.** *Degree of fragmentation represented as lognormal distribution.*

File systems are complex. It is virtually impossible to study all possible factors influencing performance. For example, the various file system formatting and mounting options alone yield a large number of combinations. In addition, the run-time environment is complex; for example, file system data is often buffered in OS page caches in memory, and differences in memory size can dramatically change file system behavior.

In this study, we choose to focus on a subset of factors that we believe are most relevant to allocation behavior. As we will see, these factors are broad enough to discover interesting performance oddities; they are also not so broad as to make a thorough exploration intractable.

There are three categories of input factors in *Chopper*. The first category of factors describes the initial state of the file system. The second category includes a relevant OS state. The third category includes factors describing the workload itself. All factors are picked to reveal potentially interesting design issues. In the rest of this paper, a value picked for a factor is called a *level*. A set of levels, each of which is selected for a factor, is called a *treatment*. One execution of a treatment is called a *run*. We picked twelve factors, which are summarized in Table 2.1 and introduced as follows.

We create a virtual disk of **DiskSize** bytes, because block allocators may have different space management policies for disks of different sizes.

The **UsedRatio** factor describes the ratio of disk that has been used. *Chopper* includes it because block allocators may allocate blocks differently when the availability of free space is different.

The **FreeSpaceLayout** factor describes the contiguity of free space on disk. Obtaining satisfactory layouts despite a paucity of free space, which often arises when file systems are aged, is an important task for block allocators. Because enumerating all fragmentation states is impossible, we use six numbers to represent degrees from extremely fragmented to generally contiguous. We use the distribution of free extent sizes to describe the degree of fragmentations; the extent sizes follow lognormal distributions. Distributions of layout 1 to 5 are shown in Figure 2.2. For example, if layout is number 2, about $0.1 \times \mathtt{DiskSize} \times (1 - \mathtt{UsedRatio})$ bytes will consist of 32KB extents, which are placed randomly in the free space. Layout 6 is not manually fragmented, in order to have the most contiguous free extents possible.

The **CPUCount** factor controls the number of CPUs the OS runs on. It can be used to discover scalability issues of block allocators.

The **FileSize** factor represents the size of the file to be written, as allocators may behave differently when different sized files are allocated. For simplicity, if there is more than one file in a treatment, all of them have the same size.

A chunk is the data written by a `write()` call. A file is often not written by only one call, but a series of writes. Thus, it is interesting to see how block allocators act with different numbers of chunks, which **ChunkCount** factor captures. In our experiments, a file is divided into multiple chunks of equal sizes. They are named by their positions in file, e.g., if there are four chunks, chunk-0 is at the head of the file and chunk-3 is at the end.

Sparse files, such as virtual machine images [100], are commonly-used and important. Files written non-sequentially are sparse at some point in their life, although the final state is not. On the other hand, overwriting is also common and can have effect if any copy-on-write strategy is adopted [132]. The **InternalDensity** factor describes the degree of coverage (e.g. sparseness or overwriting) of a file. For example, if InternalDensity is 0.2 and chunk size is 10KB, only the 2KB at the end of each chunk will be written. If InternalDensity is 1.2, there will be two writes for each chunk; the first write of this chunk will be 10KB and the second one will be 2KB at the end of the chunk.

The **ChunkOrder** factor defines the order in which the chunks are written. It explores sequential and random write patterns, but with more control. For example, if a file has four chunks, ChunkOrder=0123 specifies that the file is written from the beginning to the end; ChunkOrder=3210 specifies that the file is written backwards.

The **Fsync** factor is defined as a bitmap describing whether *Chopper* performs an `fsync()` call after each chunk is written. Applications, such as databases, often use `fsync()` to force data durability immediately [74, 91]. This factor explores how `fsync()` may interplay with allocator features (e.g., delayed allocation in Linux ext4 [117]). In the experiment, if ChunkOrder=1230 and Fsync=1100, *Chopper* will perform an `fsync()` after chunk-1 and chunk-2 are written, but not otherwise.

The **Sync** factor defines how we open, close, or sync the file system with each write. For example, if ChunkOrder=1230 and Sync=0011, *Chopper* will perform the three calls after chunk-3 and perform `close()` and `sync()` after chunk-0; `open()` is not called after the last chunk is written. All Sync bitmaps end with 1, in order to place data on disk before we inquire about layout information. *Chopper* performs `fsync()` before `sync()` if they both are requested for a chunk.

The **FileCount** factor describes the number of files written, which is used to explore how block allocators preserve spatial locality for one file and for multiple files. In the experiment, if there is more than one file, the chunks of each file will be written in an interleaved fashion. The Chunk-Order, Fsync, and Sync for all the files in a single treatment are identical.

Chopper places files in different nodes of a directory tree to study how parent directories can affect the data layouts. The **DirectorySpan** factor describes the distance between parent directories of the first and last files in a breadth-first traversal of the tree. If FileCount=1, DirectorySpan is the index of the parent directory in the breadth-first traversal sequence. If FileCount=2, the first file will be placed in the first directory, and the second one will be at the *DirectorySpan*-th position of the traversal sequence.

In summary, the input space of the experiments presented is described in Table 2.1. The choice is based on efficiency and simplicity. For example, we study relatively small file sizes because past studies of file systems indicates most files are relatively small [54, 64, 136]. Specifically, Agrawal et. al. found that over 90% of the files are below 256 KB across a wide range of systems [54]. Our results reveal many interesting behaviors, many of which also apply to larger files. In addition, we study relatively small disk sizes as large ones slow down experiments and prevent broad explorations in limited time. The file system problems we found with small disk sizes are also present with large disks.

Simplicity is also critical. For example, we use at most two files in these experiments. Writing to just two files, we have found, can reveal interesting nuances in block allocation. Exploring more files make the results more challenging to interpret. We leave further exploration of the file system input space to future work.

2.3.3 Layout Diagnosis Response

To find contract violations from block allocators, which aim to place data compactly to avoid time-consuming seeking on HDDs [58, 138], we need an intuitive metric reflecting data layout quality. To this end, we define *d-span*, the distance in bytes between the first and last physical block of a file. In other words, *d-span* measures the worst allocation decision the allocator makes in terms of spreading data. As desired, *d-span* is an *indirect* performance metric, and, more importantly, an intuitive diagnostic signal that helps us find unexpected file-system behaviors. These behaviors may produce poor layouts that eventually induce long data access latencies. *d-span* captures subtle problematic behaviors which would be hidden if end-to-end performance metrics were used. Ideally, *d-span* should be the same size as the file.

d-span is not intended to be an one-size-fits-all metric. Being simple, it has its weaknesses. For example, it cannot distinguish cases that have the same span but different internal layouts. An alternative of *d-span* that we have investigated is to model data blocks as vertices in a graph and use *average path length* [84] as the metric. The minimum distance between two vertices in the graph is their corresponding distance on disk. Although this metric is able to distinguish between various internal layouts, we have found that it is often confusing. In contrast, *d-span* contains less information but is much easier to interpret.

In addition to the metrics above, we have also explored metrics such as number of data extents, layout score (fraction of contiguous blocks) [148], and normalized versions of each metric (e.g. *d-span*/`ideal` *d-span*). One can even create a metric by plugging in a disk model to measure quality. Our diagnostic framework works with all of these metrics, each of which allows us to view the system from a different angle. However, *d-span* has the best trade-off between information gain and simplicity.

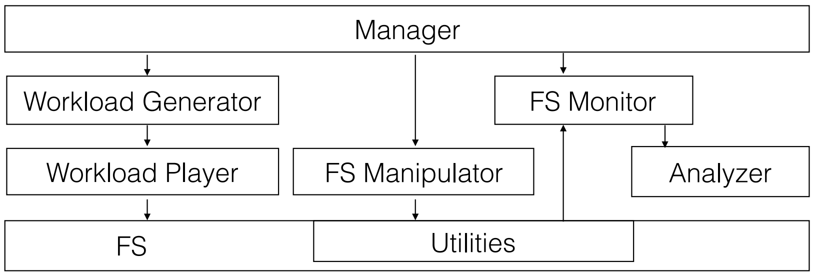


Figure 2.3: ***Chopper*** **components.**

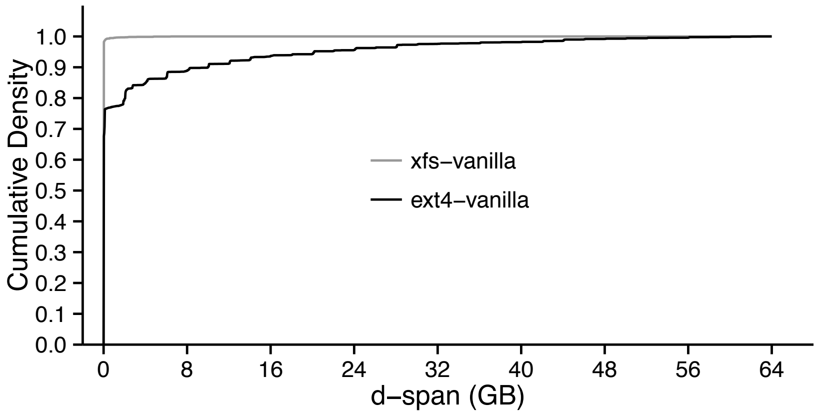


Figure 2.4: ***d-span*** **CDFs of ext4 and XFS.** *The 90th%, 95th%, and max d-spans of ext4 are 10GB, 20GB, and 63GB, respectively. The 90th%, 95th%, and max d-spans of XFS are 2MB, 4MB, and 6GB, respectively.*

2.3.4 Implementation

The components of *Chopper* are presented in Figure 2.3. The **Manager** builds an experimental plan and conducts the plan using the other components. The **FS Manipulator** prepares the file system for subsequent workloads. In order to speed up the experiments, the file system is mounted on an in-memory virtual disk, which is implemented as a loop-back device backed by a file in a RAM file system. The initial disk images are re-used whenever needed, thus speeding up experimentation and providing reproducibility. After the image is ready, the **Workload Generator** produces a workload description, which is then fed into the **Workload**

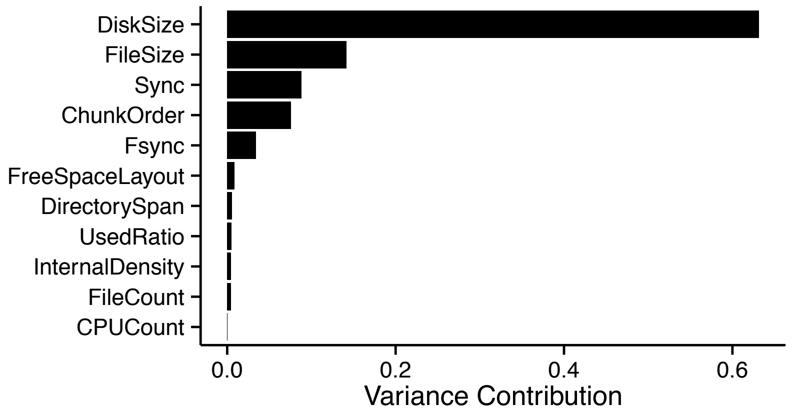


Figure 2.5: **Contribution to *d-span* variance.** *It shows contributions calculated by factor prioritization of sensitivity analysis.*

Player for running. After playing the workload, the Manager informs the **FS Monitor**, which invokes existing system utilities, such as *debugfs* and *xfs_db*, to collect layout information. No kernel changes are needed. Finally, layout information is merged with workload and system information and fed into the **Analyzer**. The experiment runs can be executed in parallel to significantly reduce time.

2.4 The Analysis of Violations

We use *Chopper* to help understand the policies of file system block allocators, to achieve more predictable and consistent data layouts, and to reduce the chances of performance fluctuations. We focus on Linux ext4 [117] and XFS [145], which are among the most popular local file systems [29, 33, 49, 131].

For each file system, we begin in Section 2.4.1 by asking whether or not it provides robust file layout in the presence of uncertain workloads (i.e., whether the unwritten contract is violated). If the file system is robust (i.e., XFS), then we claim success; however, if it is not (i.e., ext4), then we delve further into understanding the workload and environment factors that cause the unpredictable layouts. Once we understand the combination of factors that are problematic, in Section 2.4.2, we search for the responsible policies in the file system source code and improve those policies.

2.4.1 File System as a Black Box

2.4.1.1 Does a Tail Exist?

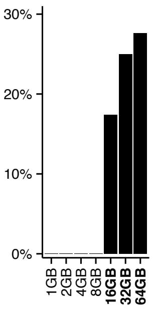
The first question we ask is whether or not the file allocation policies in Linux ext4 and XFS are robust to the input space introduced in Table 2.1.

To find out if there are tails (i.e., violations) in the resulting allocations, we conducted experiments with 16,384 runs using *Chopper*. The experiments were conducted on a cluster of nodes with 16 GB RAM and two Opteron-242 CPUs [87]. The nodes ran Linux v3.12.5. Exploiting *Chopper*'s parallelism and optimizations, one full experiment on each file system took about 30 minutes with 32 nodes.

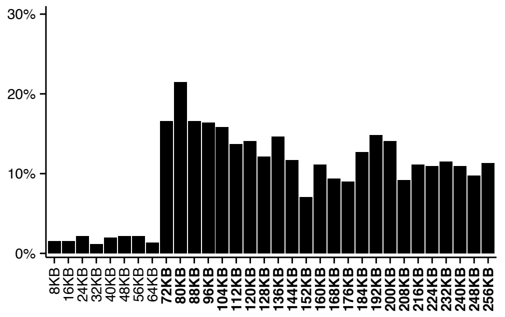
Figure 2.4 presents the empirical CDF of the resulting *d-span*s for each file system over all the runs; in runs with multiple files, the reported *d-span* is the maximum *d-span* of the allocated files. A large *d-span* value indicates a file with poor locality (i.e., a violation of the HDD unwritten contract). Note that the file sizes are never larger than 256KB, so *d-span* with optimal allocation would be only 256KB as well.

The figure shows that the CDF line for XFS is nearly vertical; thus, XFS allocates files with relatively little variation in the *d-span* metric, even with widely differing workloads and environmental factors. While XFS may not be ideal, this CDF indicates that its block allocation policy is relatively robust.

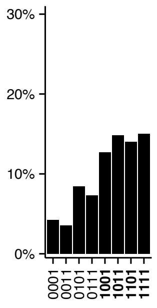
In contrast, the CDF for ext4 has a significant tail. Specifically, 10% of the runs in ext4 have at least one file spreading over 10GB. This tail indicates instability in the ext4 block allocation policy that could produce



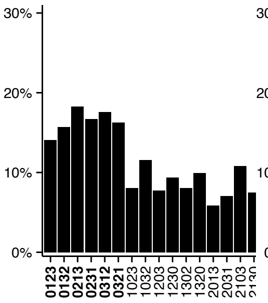
(a) DiskSize



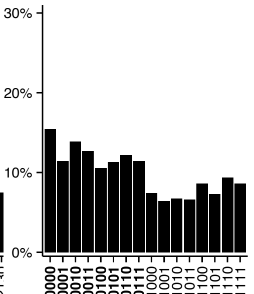
(b) FileSize



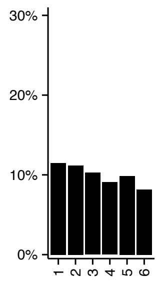
(c) Sync



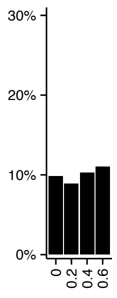
(d) ChunkOrder



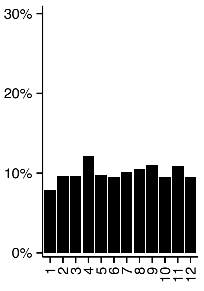
(e) Fsync



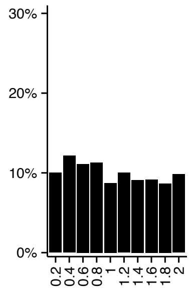
(f) Freespace-layout



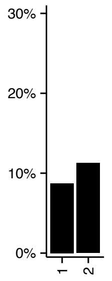
(g) Used-Ratio



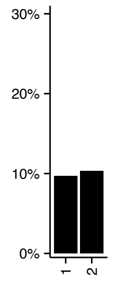
(h) DirectorySpan



(i) InternalDensity



(j) FileCount



(k) CPU-Count

Figure 2.6: **Tail Distribution of 11 Factors.** *In the figure, we can find what levels of each factor have tail runs and percentage of tail runs in each level. Regions with significantly more tail runs are marked bold. Note that the number of total runs of each level is identical for each factor. Therefore, the percentages between levels of a factor are comparable. For example, (a) shows all tail runs in the experiment have disk sizes $\geqslant$ 16GB. In addition, when DiskSize=16GB, 17% of runs are in the tail (d-span$\geqslant$10GB) which is less than DiskSize=32GB.*

poor layouts inducing long access latencies.

2.4.1.2 Which factors contribute to the tail?

We next investigate which workload and environment factors contribute most to the variation seen in ext4 layout. Understanding these factors is important for two reasons. First, it can help file system users to see which workloads run best on a given file system and to avoid those which do not run well; second, it can help file system developers track down the source of internal policy problems.

The contribution of a factor to variation can be calculated by variance-based *factor prioritization*, a technique in sensitivity analysis [140]. Specifically, the contribution of factor X_i is calculated by:

$$S_i = \frac{V_{X_i}(E_{X_{\sim i}}(Y|X_i = x_i^*))}{V(Y)}$$

S_i is always smaller than 1 and reports the ratio of the contribution by factor X_i to the overall variation. In more detail, if factor X_i is fixed at a particular level x_i^*, then $E_{X_{\sim i}}(Y|X_i = x_i^*)$ is the resulting mean of response values for that level, $V_{X_i}(E_{X_{\sim i}}(Y|X_i = x_i^*))$ is the variance among level means of X_i, and $V(Y)$ is the variance of all response values for an experiment. Intuitively, S_i indicates how much changing a factor can affect the response.

Figure 2.5 presents the contribution of each factor for ext4; again, the metric indicates the contribution of each factor to the variation of *d-span* in the experiment. The figure shows that the most significant factors are DiskSize, FileSize, Sync, ChunkOrder, and Fsync; that is, changing any one of those factors may significantly affect *d-span* and layout quality. DiskSize is the most sensitive factor, indicating that ext4 does not have stable layout quality with different disk sizes. It is not surprising that FileSize affects *d-span* considering that the definition *d-span* depends on the size of the file; however, the variance contributed by FileSize (0.14 $\times$

$V(\mathrm{dspan}_{real}) = 3\times10^{18}$) is much larger than ideally expected ($V(\mathrm{dspan}_{ideal}) = 6 \times 10^{10}$, $\mathrm{dspan}_{ideal} = \mathrm{FileSize}$). The significance of Sync, ChunkOrder, and Fsync imply that certain write patterns are much worse than others for ext4 allocator.

Factor prioritization gives us an overview of the importance of each factor and guides further exploration. We would also like to know which factors and which levels of a factor are most responsible for the tail. This can be determined with *factor mapping* [140]; factor mapping uses a threshold value to group responses (i.e., *d-span* values) into tail and non-tail categories and finds the input space of factors that drive the system into each category. We define the threshold value as the 90th% (10GB in this case) of all *d-span*s in the experiment. We say that a run is a *tail run* if its response is in the tail category.

Factor mapping visualization in Figure 2.6 shows how the tails are distributed to the levels of each factor. Thanks to the balanced Latin hypercube design with large sample size, the difference between any two levels of a factor is likely to be attributed to the level change of this factor and not due to chance.

Figure 2.6a shows that all tail runs lay on disk sizes over 8GB because the threshold *d-span* (10GB) is only possible when the disk size exceeds that size. This result implies that blocks are spread farther as the capacity of the disk increases, possibly due to poor allocation polices in ext4. Figure 2.6b shows a surprising result: there are significantly more tail runs when the file size is larger than 64KB. This reveals that ext4 uses very different block allocation polices for files below and above 64KB.

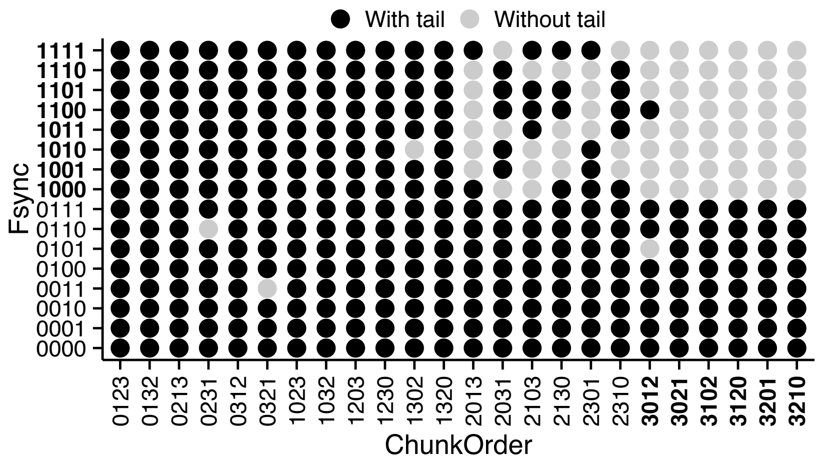
Sync, ChunkOrder, and Fsync also present interesting behaviors, in which the first written chunk plays an important role in deciding the tail. Figure 2.6c shows that closing and sync-ing after the first written chunk (coded 1***) causes more tail runs than otherwise. Figure 2.6d shows that writing chunk-0 of a file first (coded 0***), including sequential writes (coded 0123) which are usually preferred, leads to more tail runs. Figure 2.6e shows that, on average, not fsync-ing the first written chunk (coded 0***) leads to more tail runs than otherwise.

The rest of the factors are less significant, but still reveal interesting observations. Figure 2.6f and Figure 2.6g show that tail runs are always present and not strongly correlated with free space layout or the amount of free space, even given the small file sizes in our workloads (below 256KB). Even with layout number 6 (not manually fragmented), there are still many tail runs. Similarly, having more free spaces does not reduce tail cases. These facts indicate that many tail runs do not depend on the disk state and instead it is the ext4 block allocation policy itself causing these tail runs. After we fix the ext4 allocation polices in the next section, the DiskUsed and FreespaceLayout factors will have a much stronger impact.

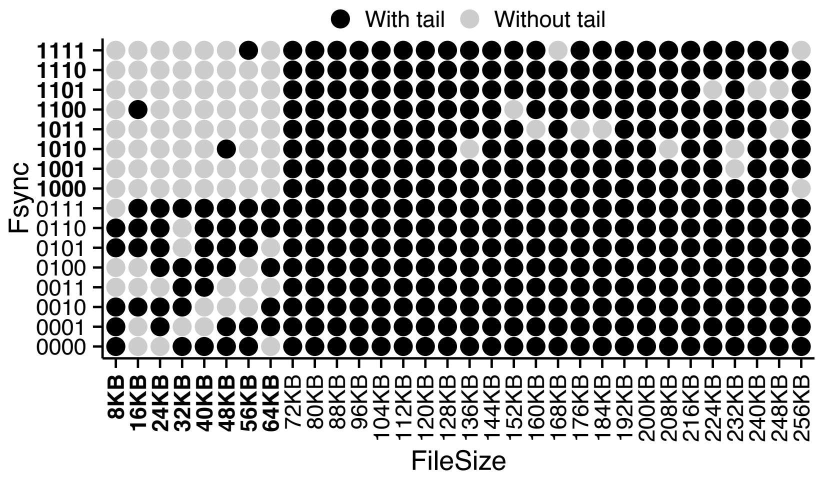
Finally, Figure 2.6h and Figure 2.6i show that tail runs are generally not affected by DirectorySpan and InternalDensity. Figure 2.6j shows that having more files leads to 29% more tail cases, indicating potential layout problems in production systems where multi-file operations are common. Figure 2.6k shows that there are 6% more tail cases when there are two CPUs.

2.4.1.3 Which factors interact in the tail?

In a complex system such as ext4 block allocator, performance may depend on more than one factor. We have inspected all two-factor interactions and select two cases in Figure 2.7 that present clear patterns. The figures show how pairwise interactions may lead to tail runs, revealing both dangerous and low-danger zones in the workload space; these zones give us hints about the causes of the tail, which will be investigated in Section 2.4.2. Figure 2.7a shows that, writing and fsync-ing chunk-3 first significantly reduces tail cases. In Figure 2.7b, we see that, for files not larger



(a) ChunkOrder and Fsync.



(b) FileSize and Fsync.

Figure 2.7: **Tail Runs in the Interactions of Factors.** *Note that each interaction data point corresponds to multiple runs with other factors varying. A black dot means that there is at least one tail case in that interaction. Low-danger zones are marked with bold labels.*

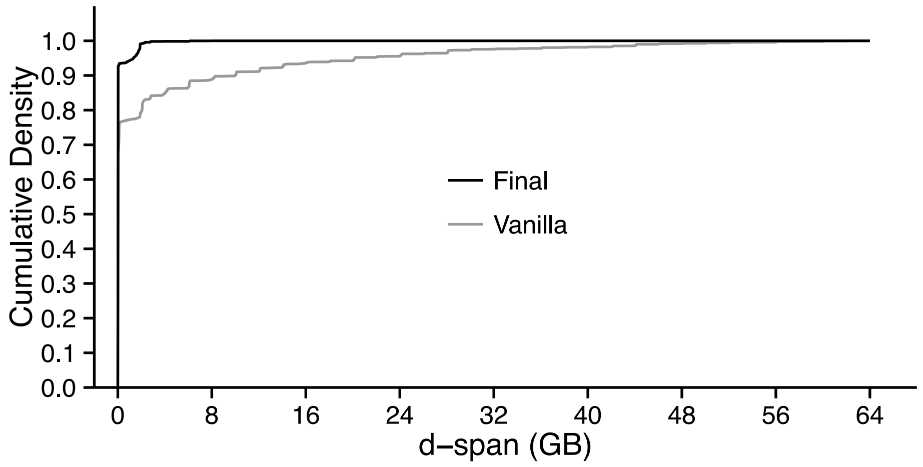


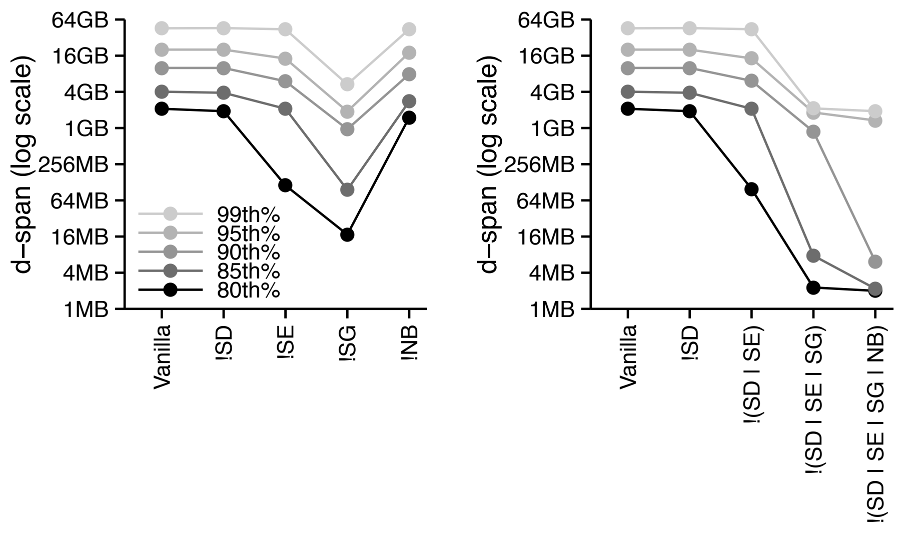
Figure 2.8: ***d-span* CDF of vanilla and final versions of ext4.** *The final version reduces the 80th, 90th, and 99th percentiles by* 1.0×10^3, 1.6×10^3, *and* 24 *times, respectively.*

than 64KB, fsync-ing the first written chunk significantly reduces the possibility of producing tail runs. These two figures do not conflict with each other; in fact, they indicate a low-danger zone in a three-dimension space.

Evaluating ext4 as black box, we have shown that ext4 does not consistently provide good layouts given diverse inputs. Our results show that unstable performance with ext4 is not due to the external state of the disk (e.g., fragmentation or utilization), but to the internal policies of ext4. To understand and fix the problems with ext4 allocation, we use detailed results from *Chopper* to guide our search through ext4 documentation and source code.

2.4.2 File System as a White Box

Our previous analysis uncovered a number of violations from the layout policies of ext4, but it did not pinpoint the location of those policies within the ext4 source code. We now use the hints provided by our previous data analysis to narrow down the sources of problems and to perform detailed source code tracing given the set of workloads suggested by *Chopper*. In



(a) Effect of Single Fix (b) Cumulative Effect of Fixes

Figure 2.9: **Effect of fixing issues..** *Vanilla: Linux v3.12.5. "!" means "without". SD: Scheduler Dependency; SE: Special End; SG: Shared Goal; NB: Normalization Bug. !(X | Y) means X and Y are both removed in this version.*

this manner, we are able to fix a series of violations in the ext4 layout policies and show that each fix reduces the tail cases in ext4 layout.

Figure 2.8 compares the original version of ext4 and our final version that has four violations of the HDD unwritten contract removed. We can see that the fixes significantly reduce the size of the tail, providing better and more consistent layout quality. We now connect the symptoms of problems shown by *Chopper* to their root causes in the code.

2.4.2.1 Randomness → Scheduler Dependency

Our first step is to remove non-determinism for experiments with the same treatment. Our previous experiments corresponded to a single run for each treatment; this approach was acceptable for summarizing from a large sample space, but cannot show intra-treatment variation. After we identify and remove this intra-treatment variation, it will be more straightforward to remove other tail effects.

We conducted two repeated experiments with the same input space as in Table 2.1 and found that 6% of the runs have different *d-span*s for the same treatment; thus, ext4 can produce different layouts for the same controlled input. Figure 2.10a shows the distribution of the *d-span* differences for those 6% of runs. The graph indicates that the physical data layout can differ by as much as 46GB for the same workload.

Examining the full set of factors responsible for this variation, we found interesting interactions between FileSize, CPUCount, and ChunkOrder. Figure 2.10b shows the count of runs in which *d-span* changed between identical treatments as a function of CPUCount and FileSize. This figure gives us the hint that *small* files in multiple-CPU systems may suffer from unpredictable layouts. Figure 2.10c shows the number of runs with changed *d-span* as a function of ChunkOrder and FileSize. This figure indicates that most small files and those large files written with more sequential patterns are affected.

Root Cause: With these symptoms as hints we focused on the interaction between small files and the CPU scheduler. Linux ext4 has an allocation policy such that files not larger than 64KB (*small* files) are allocated from *locality group (LG) preallocations*; further, the block allocator associates each LG preallocation with a CPU, in order to avoid contention. Thus, for *small* files, the layout location is based solely on which CPU the flusher thread is running. Since the flusher thread can be scheduled on different CPUs, the same small file can use different LG preallocations spread across the entire disk.

This policy is also the cause of the variation seen by some large files written sequentially: large files written sequentially begin as small files and are subject to LG preallocation; large files written backwards have large sizes from the beginning and never trigger this scheduling depen-

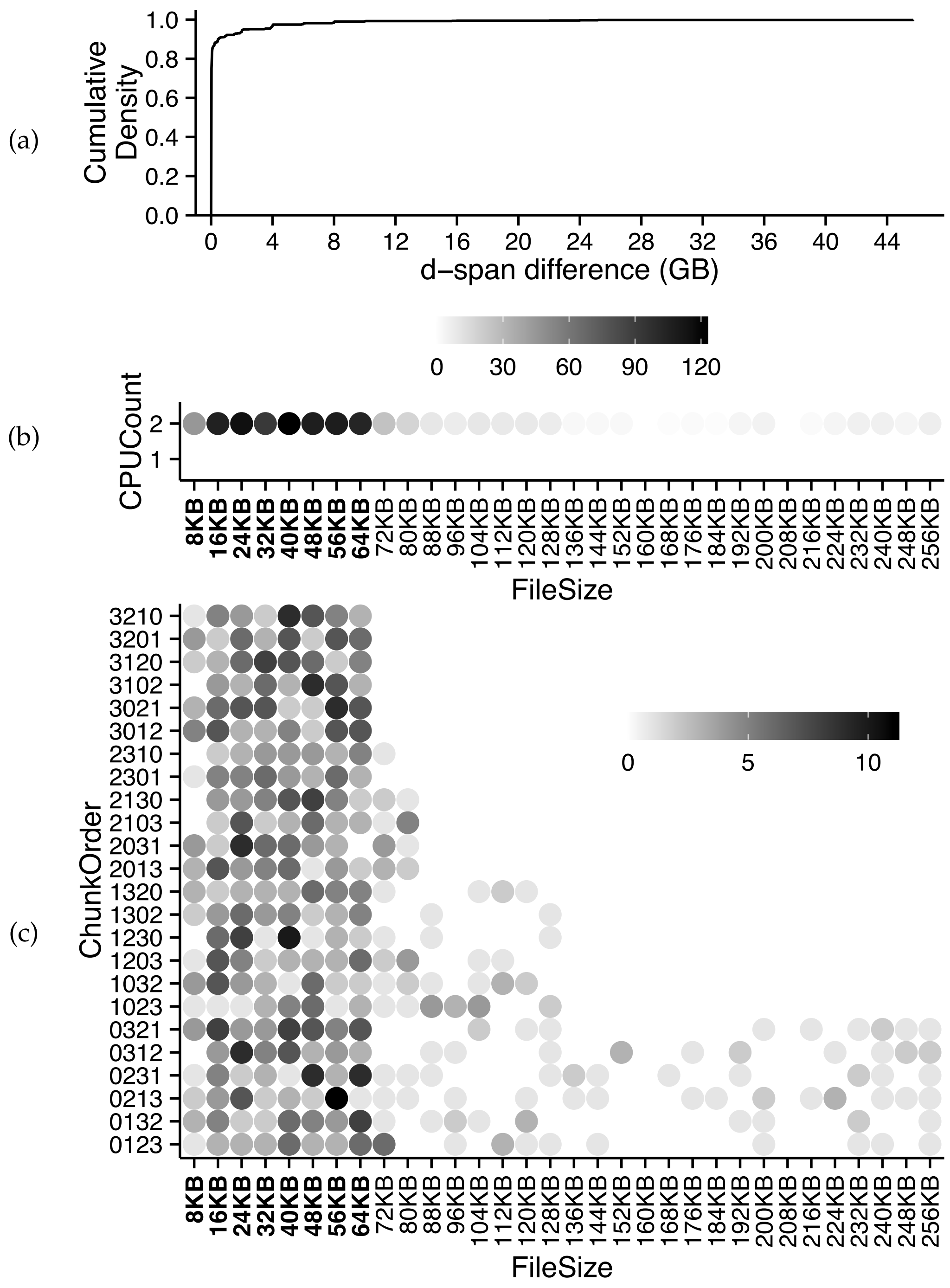


Figure 2.10: **Symptoms of Randomness.** *(a): CDF of d-span variations between two experiments. The median is 1.9MB. The max is 46GB. (b): Number of runs with changed d-span, shown as the interaction of FileSize and CPUCount. (c): Number of runs with changed d-span, shown as the interaction of FileSize and ChunkOrder. Regions with considerable tail runs are marked with bold labels.*

dency[1]. In production systems with heavy loads, more cores, and more files, we expect more unexpected poor layouts due to this effect.

Fix: We remove the problem of random layout by choosing the locality group for a *small* file based on its i-number range instead of the CPU. Using the i-number not only removes the dependency on the scheduler, but also ensures that *small* files with close i-numbers are likely to be placed close together. We refer to the ext4 version with this new policy as *!SD*, for no Scheduler Dependency.

Figure 2.9a compares vanilla ext4 and *!SD*. The graph shows that the new version slightly reduces the size of the tail. Further analysis shows that in total *d-span* is reduced by 1.4 TB in 7% of the runs but is increased by 0.8 TB in 3% of runs. These mixed results occur because this first fix unmasks other problems which can lead to larger *d-span*s. In complex systems such as ext4, performance problems interact in surprising ways; we will progressively work to remove three more problems.

2.4.2.2 Allocating Last Chunk $\rightarrow$ Special End

We now return to the interesting behaviors originally shown in Figure 2.7a, which showed that allocating chunk-3 first (Fsync=1*** and ChunkOrder=3***) helps to avoid tail runs. To determine the cause of poor allocations, we compared traces from selected workloads in which a tail occurs to similar workloads in which tails do not occur.

Root Cause: Linux ext4 uses a *Special End* policy to allocate the last extent of a file when the file is no longer open; specifically, the last extent does not trigger preallocation. The Special End policy is implemented by checking three conditions - *Condition 1*: the extent is at the end of the file; *Condition 2*: the file system is not busy; *Condition 3*: the file is not open. If

[1]Note that file size in ext4 is calculated by the ending logical block number of the file, not the sum of physical blocks occupied.

all conditions are satisfied, this request is marked with the hint "do not preallocate", which is different from other parts of the file[2].

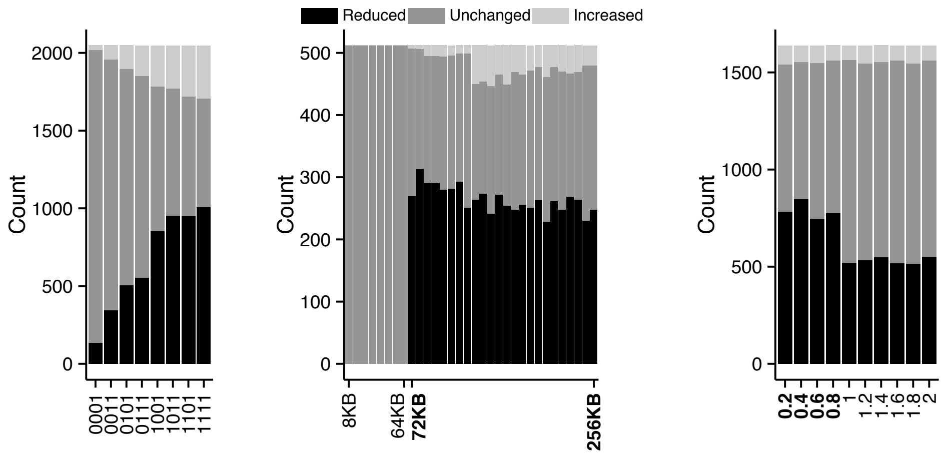
The motivation is that, since the status of a file is final (i.e., no process can change the file until the next open), there is no need to reserve additional space. While this motivation is valid, the implementation causes an inconsistent allocation for the last extent of the file compared to the rest; the consequence is that blocks can be spread far apart. For example, a small file may be inadvertently split because non-ending extents are allocated with LG preallocations while the ending extent is not; thus, these conflicting policies drag the extents of the file apart.

This policy explains the tail-free zone (Fsync=1*** and ChunkOrder=3***) in Figure 2.7a. In these tail-free zones, the three conditions cannot be simultaneously satisfied since fsync-ing chunk-3 causes the last extent to be allocated, while the file is still open; thus, the Special End policy is not triggered.

Fix: To reduce the layout variability, we have removed the Special End policy from ext4; in this version named *!SE*, the ending extent is treated like all other parts of the file. Figure 2.9 shows that *!SE* reduces the size of the tail. Further analysis of the results show that removing Special End policy reduces *d-span*s for 32% of the runs by a total of 21TB, but increases *d-span*s for 14% of the runs by a total of 9TB. The increasing of *d-span* is primarily because removing this policy unmasks inconsistent policies in File Size Dependency, which we will discuss next.

Figure 2.11a examines the benefits of the *!SE* policy compared to vanilla ext4 in more detail; to compare only deterministic results, we set CPU-Count=1. The graph shows that the *!SE* policy significantly reduces tail runs when the workload begins with sync operations (combination of `close()`, `sync()`, and `open()`); this is because the Special End policy is

[2]In fact, this hint is vague. It means: 1. if there is a preallocation solely for this file (i.e., *i-node preallocation*), use it; 2. do not use LG preallocations, even they are available 3. do not create any new preallocations.



(a) Sync (b) FileSize (c) InternalDensity

Figure 2.11: **Effects of removing problematic policies.** *The d-spans could be 'Reduced', 'Unchanged' or 'Increased' due to the removal. (a): removing Special End; (b) and (c): removing Shared Global.*

more likely to be triggered when the file is temporarily closed.

2.4.2.3 File Size Dependency → Shared Global

After removing the Scheduler Dependency and Special End policies, ext4 layout still presents a significant tail. Experimenting with these two fixes, we observe a new symptom that occurs due to the interaction of FileSize and ChunkOrder, as shown in Figure 2.12. The stair shape of the tail runs across workloads indicates that this policy only affects *large* files and it depends upon the first written chunk.

Root Cause: Traces of several representative data points reveal the source of the 'stair' symptom, which we call *File Size Dependency*. In ext4, one of the design goals is to place small files (less than 64KB, which is tunable) close and big files apart [58]. Blocks for *small* files are allocated from *LG preallocations*, which are shared by all *small* files; blocks in *large* files are allocated from per-file *inode preallocations* (except for the ending extent of a closed file, due to the Special End policy).

This file-size-dependant policy ignores the activeness of files, since the dynamically changing size of a file may trigger inconsistent allocation policies for the same file. In other words, blocks of a file larger than 64KB can be allocated with two distinct policies as the file grows from *small* to *large*. This changing policy explains why FileSize is the most significant workload factor, as seen in Figure 2.5, and why Figure 2.6b shows such a dramatic change at 64KB.

Sequential writes are likely to trigger this problem. For example, the first 36KB extent of a 72KB file will be allocated from the LG preallocation; the next 36KB extent will be allocated from a new i-node preallocation (since the file is now classified as *large* with 72KB $>$ 64KB). The allocator will try to allocate the second extent next to the first, but the preferred location is already occupied by the LG preallocation; the next choice is to use the block group where the last big file in the whole file system was allocated (Shared Global policy, coded *SG*), which can be far away. Growing a file often triggers this problem. File Size Dependency is the reason why runs with ChunkOrder=0*** in Figure 2.6d and Figure 2.12 have relatively more tail runs than other orders. Writing Chunk-0 first makes the file grow from a *small* size and increases the chance of triggering two distinct policies.

Fix: Placing extents of *large* files together with a shared global policy violates the initial design goal of placing big files apart and deteriorates the consequences of File Size Dependency. To mitigate the problem, we implemented a new policy (coded *!SG*) that tries to place extents of *large* files close to existing extents of that file. Figure 2.9a shows that *!SG* significantly reduces the size of the tail. In more detail, *!SG* reduces *d-span* in 35% of the runs by a total of 45TB.

To demonstrate the effectiveness of the *!SG* version, we compare the number of tail cases with it and vanilla ext4 for deterministic scenarios (CPUCount=1). Figure 2.11b shows that the layout of *large* files (>64KB)

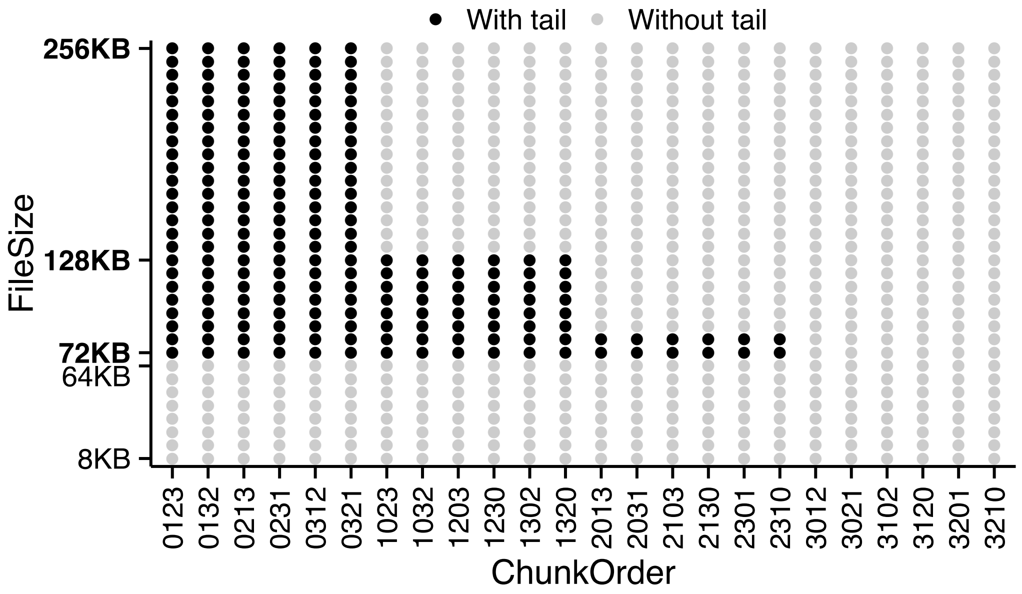


Figure 2.12: **Tail Runs in *!(SD | SE)*.** *The figure shows tail runs in the interaction of ChunkOrder and FileSize, after removing Scheduler Dependency and Special End.*

is significantly improved with this fix. Figure 2.11c shows that the layout of sparse files (with InternalDensity < 1) is also improved; the new policy is able to separately allocate each extent while still keeping them near one another.

2.4.2.4 Sparse Files $\rightarrow$ Normalization Bug

With three problems fixed in version *!(SD | SE | SG)*, we show an interesting interaction that still remains between ChunkOrder and InternalDensity. Figure 2.13 shows that while most of the workloads exhibit tails, several workloads do not, specifically, all "solid" (InternalDensity$\geqslant$1) files with ChunkOrder=3012. To identify the root cause, we focus only on workloads with ChunkOrder=3012 and compare solid and sparse patterns.

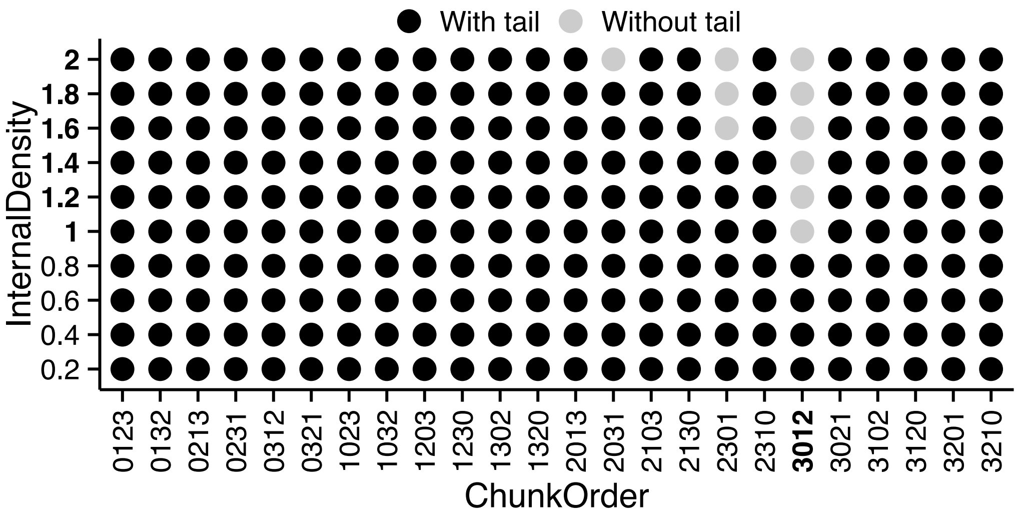


Figure 2.13: **Tail Runs in *!(SD | SE | SG)*.** *This figure shows tail runs in the interaction of ChunkOrder and InternalDensity on version !(SD | SE | SG).*

Root Cause: Comparing solid and sparse runs with ChunkOrder=3012 shows that the source of the tail is a bug in ext4 normalization; normalization enlarges requests so that the extra space can be used for a similar extent later. The normalization function should update the request's logical starting block number, corresponding physical block number, and size; however, with the bug, the physical block number is not updated and the old value is used later for allocation[3].

Figure 2.14 illustrates how this bug can lead to poor layout. In this scenario, an ill-normalized request is started (incorrectly) at the original physical block number, but is of a new (correct) larger size; as a result, the request will not fit in the desired gap within this file. Therefore, ext4 may fail to allocate blocks from preferred locations and will perform a desperate search for free space elsewhere, spreading blocks. The solid files with ChunkOrder of 3012 in Figure 2.13 avoid this bug because if chunks-0,1,2 are written sequentially after chunk-3 exists, then the physical block

[3]This bug is present even in the currently latest version of Linux, Linux v3.17-rc6. It has been confirmed by an ext4 developer and is waiting for further tests.

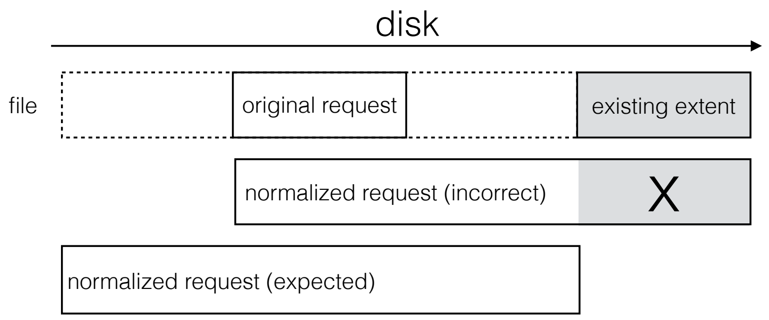


Figure 2.14: **Ill Implementation of Request Normalization.** *In this case, the normalized request overlaps with the existing extent of the file, making it impossible to fulfill the request at the preferred location.*

number of the request does not need to be updated.

Fix: We fix the bug by correctly updating the physical starting block of the request in version *!NB*. Figure 2.15 shows that *large* files were particularly susceptible to this bug, as were sparse files ($\mathtt{InternalDensity} < 1$). Figure 2.9a shows that fixing this bug reduces the tail cases, as desired. In more detail, *!NB* reduces *d-span* for 19% of runs by 8.3 TB in total. Surprisingly, fixing the bug increases *d-span* for 5% of runs by 1.5 TB in total. Trace analysis reveals that, by pure luck, the mis-implemented normalization sometimes sets the request to nearby space which happened to be free, while the correct request fell in space occupied by another file; thus, with the correct request, ext4 sometimes performs a desperate search and chooses a more distant location.

Figure 2.9 summarizes the benefits of these four fixes. Overall, with all four fixes, the 90th-percentile for *d-span* values is dramatically reduced from well over 4GB to close to 4MB. Thus, as originally shown in Figure 2.8, our final version of ext4 has a much less significant tail than the original ext4.

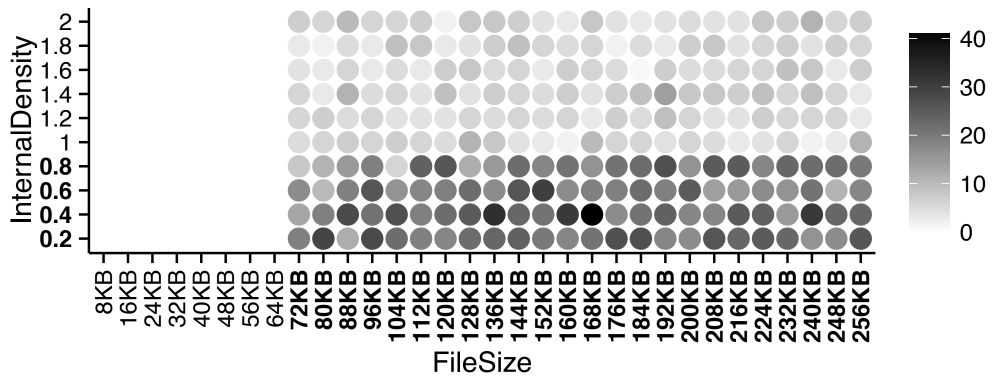


Figure 2.15: **Impact of Normalization Bug.** *This figure shows the count of runs affected by Normalization Bug in the interaction of FileSize and Internal-Density. The count is obtained by comparing experimental results ran with and without the bug.*

2.4.3 Latencies Reduced

Chopper uses *d-span* as a diagnostic signal to find violations of the HDD unwritten contract that produce poor data layouts. The poor layouts, which incur costly disk seeks on HDDs [138], and even CPU spikes [8], can in turn result in long data access latencies. Our repairs based on *Chopper*'s findings reduce latencies caused by the problematic designs.

For example, Figure 2.16 demonstrates how Scheduler Dependency incurs long latencies and how our repaired version, *!SD*, reduces latencies on an HDD (Hitachi HUA723030ALA640: 3.0 TB, 7200 RPM). In the experiment, files were created by multiple *creating threads* residing on different CPUs; each of the threads wrote a part of a 64KB file. We then measured file access time by reading and over-writing with one thread, which avoids resource contentions and maximizes performance. To obtain application-disk data transfer performance, OS and disk cache effects were circumvented. Figure 2.16 shows that with the *SD* version, access time increases with more creating threads because SD splits each file into more and potentially distant physical data pieces. Our fixed version, *!SD*,

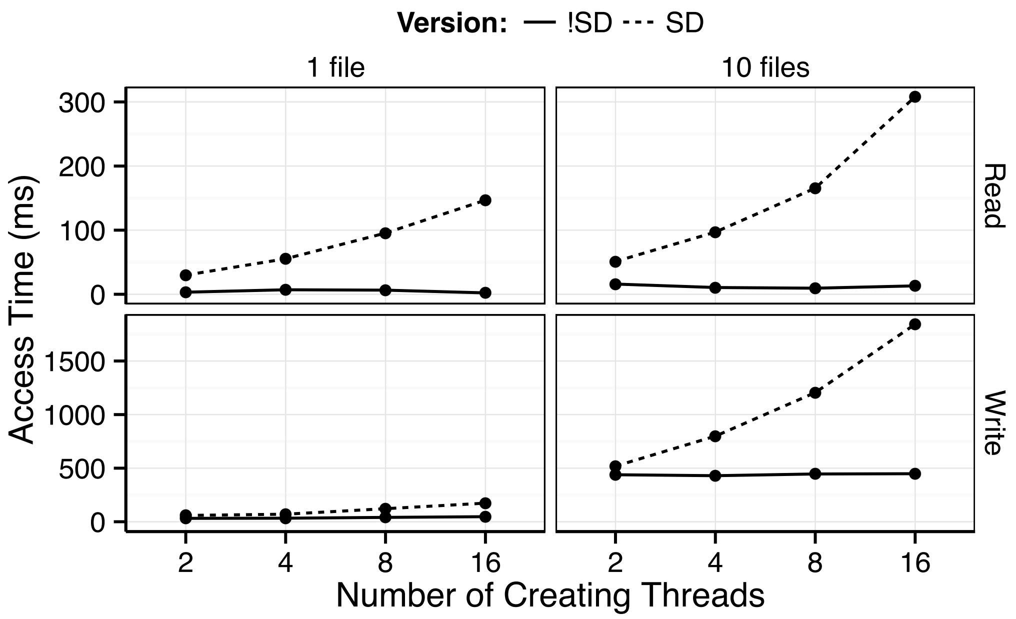


Figure 2.16: **Latency Reduction.** *This figure shows that !SD significantly reduces average data access time comparing with SD. All experiments were repeated 5 times. Standard errors are small and thus hidden for clarity.*

reduced read and write time by up to 67 and 4 times proportionally, and by up to 300 and 1400 ms. The reductions in this experiment, as well as expected greater ones with more creating threads and files, are significant – as a comparison, a round trip between US and Europe for a network packet takes 150 ms and a round trip within the same data center takes 0.5 ms [88, 125]. The time increase caused by Scheduler Dependency, as well as other issues, may translate to long latencies in high-level data center operations [80]. *Chopper* is able to find such issues, leading to fixes reducing latencies.

2.4.4 Discussion

With the help of exploratory data analysis, we have found and removed four violations of HDD unwritten contract in ext4 that can lead to unex-

Issue	Description
Scheduler Dependency	Choice of preallocation group for small files depends on CPU of flushing thread.
Special End	The last extent of a closed file may be rejected to allocate from preallocated spaces.
File Size Dependency	Preferred target locations depend on file size which may dynamically change.
Normalization Bug	Block allocation requests for large files are not correctly adjusted, causing the allocator to examine mis-aligned locations for free space.

Table 2.2: **Linux ext4 Issues.** *This table summarizes issues we have found and fixed.*

pected tail latencies; these issues are summarized in Table 2.2. We have made the patches for these issues publicly available with *Chopper*.

While these fixes do significantly reduce the tail behaviors, they have several potential limitations. First, without the Scheduler Dependency policy, flusher threads running on different CPUs may contend for the same preallocation groups. We believe that the contention degree is acceptable, since allocation within a preallocation is fast and files are distributed across many preallocations; if contention is found to be a problem, more preallocations can be added (the current ext4 creates preallocations lazily, one for each CPU). Second, removing the Shared Global policy mitigates but does not eliminate the layout problem for files with dynamically changing sizes; choosing policies based on dynamic properties such as file size is complicated and requires more fundamental policy revisions. Third, our final version, as shown in Figure 2.8, still contains a small tail. This tail is due to the disk state (DiskUsed and Freespace-Layout); as expected, when the file system is run on a disk that is more heavily used and is more fragmented, the layout for new files suffers.

The symptoms of violations revealed by *Chopper* drive us to reason about their causes. In this process, time-consuming tracing is often necessary to pinpoint a particular problematic code line as the code makes complex decisions based on environmental factors. Fortunately, analyzing and visualizing the data sets produced by *Chopper* enabled us to focus on several representative runs. In addition, we can easily reproduce and trace any runs in the controlled environmental provided by *Chopper*, without worrying about confounding noises.

With *Chopper*, we have learned several lessons from our experience with ext4 that may help build file systems that are robust to uncertain workload and environmental factors in the future. First, policies for different circumstances should be harmonious with one another. For example, ext4 tries to optimize allocation for different scenarios and as a result has a different policy for each case (e.g., the ending extent, *small* and *large* files); when multiple policies are triggered for the same file, the policies conflict and the file is dragged apart. Second, policies should not depend on environmental factors that may change and are outside the control of the file system. In contrast, data layout in ext4 depends on the OS scheduler, which makes layout quality unpredictable. By simplifying the layout policies in ext4 to avoid special cases and to be independent of environmental factors, we have shown that file layout is much more compact and predictable.

2.5 Conclusions

Violations of the HDD unwritten contract have high consequences and cause unexpected system fluctuations. Removing such violations will lead to a system with more consistent performance. However, identifying violations and finding their sources are challenging in complex systems because the input space can be infinite and exhaustive search is impossible. To study the violations in block allocations of XFS and ext4, we built *Chopper* to facilitate carefully designed experiments to effectively explore the input space of more than ten factors. We used Latin hypercube design and sensitivity analysis to uncover unexpected violations among many of those factors. Analysis with *Chopper* helped us pinpoint and remove four critical designs issues in ext4, which violates the unwritten contract of HDDs. Our improvements significantly reduce the problematic behaviors causing violations.

We believe that the application of established statistical methodologies to system analysis can have a tremendous impact on system design and implementation. We encourage developers and researchers alike to make systems amenable to such experimentation, as experiments are essential in the analysis and construction of robust systems. Rigorous statistics will help to reduce unexpected issues caused by intuitive but unreliable design decisions.

3 Finding Violations of the SSD Contract

The storage stack has been rapidly shifting from an HDD-based stack to an SSD-based one, thanks to the advance in flash technology. Flash-based SSDs store data on non-mechanical flash, which is very distinct from the magnetic platters in HDDs. Consequently, SSDs employ very different architectures to HDDs and present a very different unwritten performance contract.

As the storage stack shifts from HDDs to SSDs, many questions arise regarding their distinct unwritten contracts. As many applications and file systems that were designed in the HDD era but now run on SSDs, do these old applications and file systems comply with the SSD contract? Do SSDs work well with these old applications and file systems? As SSDs become more popular, new applications and file systems are designed for SSDs. Do these new applications and file systems really comply with the contract of SSDs? Do SSDs work well for them?

In this chapter, we first formalize the unwritten contract of SSDs, which was not well summarized before. Then, we conduct a vertical analysis of the storage stack, involving four applications, three file systems, and a detailed SSD simulator that we developed. Our vertical analysis finds many surprising violations of the SSD unwritten contract.

3.1 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 [37, 45]. A block consists of pages, which often range from 2 KB to 16 KB [26, 27, 39]. 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 [59].

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 [37]), 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 [37]). A block is usually programmed from the low page to high page to avoid program disturbance, an effect that changes nearby bits unintentionally [27, 28, 59].

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 [21, 75, 79, 127]. 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 [89, 98, 104, 111, 112, 115, 128, 129, 160, 162]. 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.

3.2 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* [143] 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 [86, 110, 121, 146]. 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.

3.2.1 Request Scale

Modern SSDs have multiple independent units, such as channels, that can work in parallel. To exploit this parallelism, one common technique when request sizes are large is to stripe each request into sub-requests and send them to different units [55, 71, 83, 96]. 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

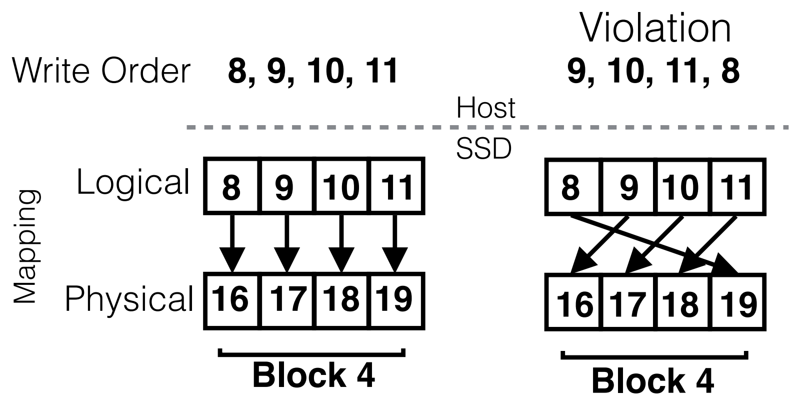


Figure 3.1: **Example of Aligned Sequentiality and a violation of it.** *Each block has four pages. Writes must be programmed on a flash block from low to high pages to avoid program disturbance. The page-level mapping on the left can be converted to a single block-level mapping: logical block 2 $\rightarrow$ physical block 4. The example on the right cannot be converted without re-arranging the data.*

Queuing (NCQ)[1] or similar features [31, 43]; a typical maximum queue depth of modern SATA SSDs is 32 requests [20, 71].

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 [71, 96].

3.2.2 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

[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.

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 [89]. With a page size of 2 KB, a 512-GB SSD would require 2 GB of RAM.[3] Having larger pages, such as those ($\geqslant$ 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 [50]. 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 [89], while others exploit both temporal and spatial locality [98, 111, 112].

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 [89, 98]. 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 [158], improving performance.

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

3.2.3 Aligned Sequentiality

Another choice for reducing memory requirements is hybrid mapping [104, 111, 112, 128], 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 3.1 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.

3.2.4 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

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

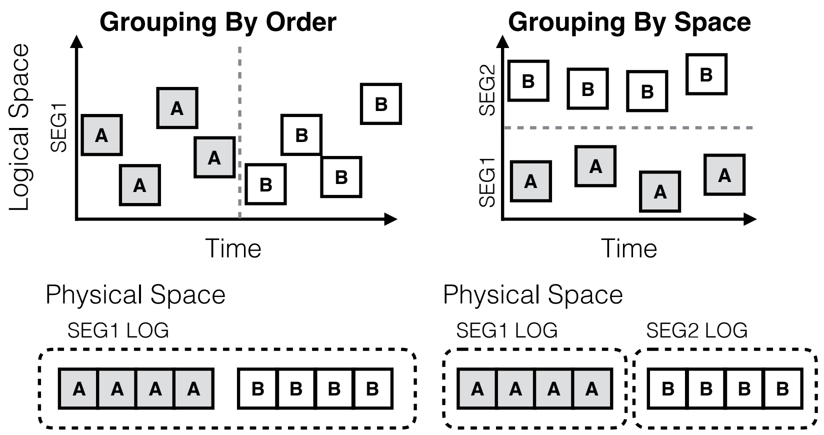


Figure 3.2: **Demonstration of Grouping by Death Time.** *Data A and B have different death times. In the figure, Vertical locations of the data in the same group are randomized to emphasize its irrelevance to this rule. Note that grouping by space is not available in non-segmented FTLs [89, 98, 162].*

both live and dead data reside in the block. We call such a time window a *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 [73, 82, 103, 130]. 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 3.2 (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 [104, 112, 128]. 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 3.2 (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 [129, 147, 161], 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.

3.2.5 Uniform Data Lifetime

Flash cells can endure a limited number of program/erase (P/E) cycles before wearing out [99, 122]. The number of P/E cycles is on the order of 10^3 P/E cycles for recent commercial SSDs [27, 120, 144] and is expected to decrease in the future [116]. 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.

To prevent uneven wear out, FTLs conduct wear-leveling, which can be dynamic or static [66]. 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

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

Table 3.1: **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.*

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.

3.2.6 Discussion

The unwritten contract of SSDs is summarized in Table 3.1. 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 channels. Although we do not focus on quantifying such performance impact in this dissertation, in Table 3.2 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.

3.3 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

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 3.2: **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 [71]): 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 [89]): 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 [162]) is similar to C.* **E** *and* **F** *(from Figure 10(e) of [111]): 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 [110]): it is obtained by running a synthetic benchmark on ext4 for multiple times on a full SSD.* **H** *(from Figure 3(a) of [103]) and* **I** *(from Figure 8(a) of [73]): the difference is between grouping and non-grouping workloads.* **J** *(from Figure 12 of [66]): the difference is due to static wear-leveling activities.*

can turn random writes of applications into sequential ones [137], 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,

[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.

WiscSim is the first SSD simulator that supports NCQ [55, 89, 96, 107]. *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 [20], 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 [89, 107], SSDsim [96], and SSD extension for DiskSim [55]). One reason is that most existing simulators (FlashSim and SSDsim) do not implement discrete-event simulation[6] [89, 96, 107], a method for simulating queuing systems like SSDs [135]. 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 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-

[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.

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) [22], RocksDB (version 4.11.2) [35], SQLite (Roll Back mode and Write-Ahead-Logging mode, version 3.8.2) [41] and the Varmail benchmark [13]. 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 [23] to avoid large cache effects.

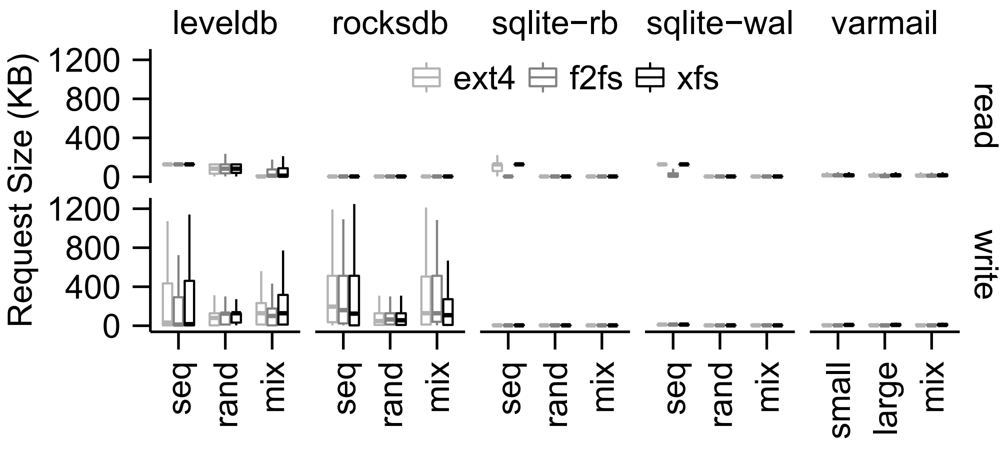
What file systems are studied? We study two traditional file systems (ext4 and XFS) and a newer one that is designed for SSDs (F2FS), all on Linux 4.5.4. Both ext4 and XFS are among the most mature and popular

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 [3, 14]. 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 [3, 14].

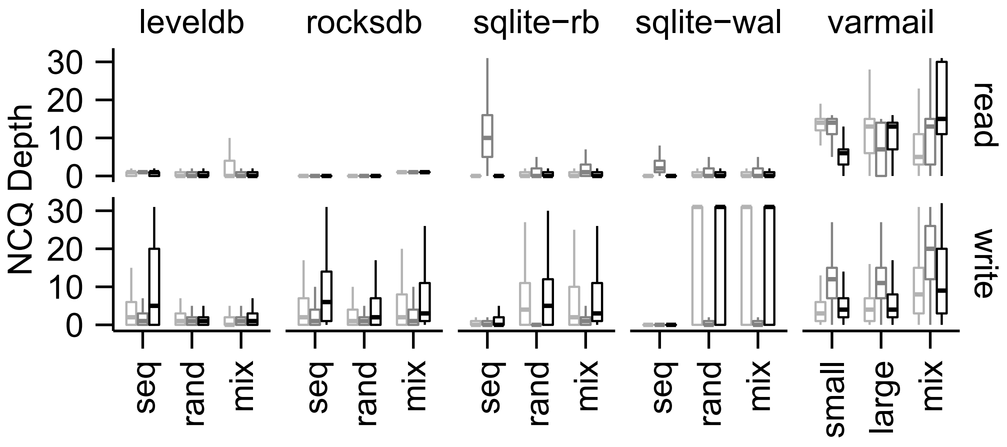
How to evaluate rule violations? We evaluate how well the contractors conform to each rule with the help of *WiscSee*. *WiscSee* automatically executes, traces and analyzes combinations of applications and file systems. To examine request scale and uniform data lifetime, *WiscSee* analyzes the traces 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 [155, 156], 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



(a) Request Size



(b) NCQ Depth

Figure 3.3: **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 [32]. Note that Linux block layer splits requests if they exceed a maximum size limit (1280 KB in our case) [24].*

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.

3.4 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.

3.4.1 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 3.3 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 [109]. 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 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 3.3a 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 3.3b 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 [35].[8] We do see the number of writing processes increase, but the write concurrency does not increase and device bandwidth is underutilized. For example, Figure 3.4 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.

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

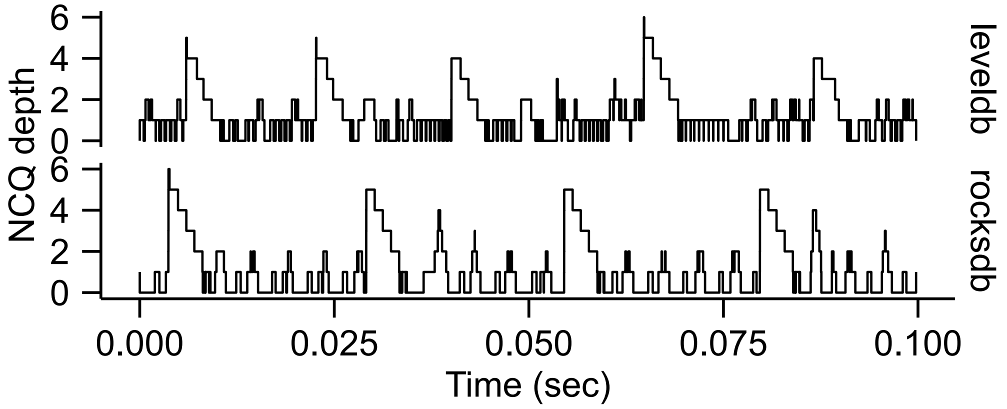


Figure 3.4: **Request Scale - NCQ Depths During Compaction.** *The lower data points are from read requests; higher ones are from writes.*

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 relatively higher queue depth shown in Figure 3.3b 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 3.4 (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 3.4 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" [36]), 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

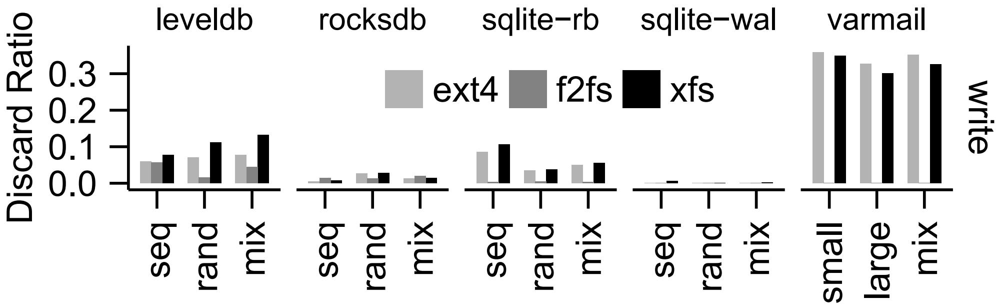


Figure 3.5: **Request Scale - Ratio of Discard Operations.** *The discard ratios of read workloads are not shown because they issue a negligible number of discards (or none at all).*

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.

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 [74, 133]. 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 immediately-valuable read and write operations. We present the ratio of discard operations to all operations for all write workloads in Figure 3.5. 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 [47, 94]). 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

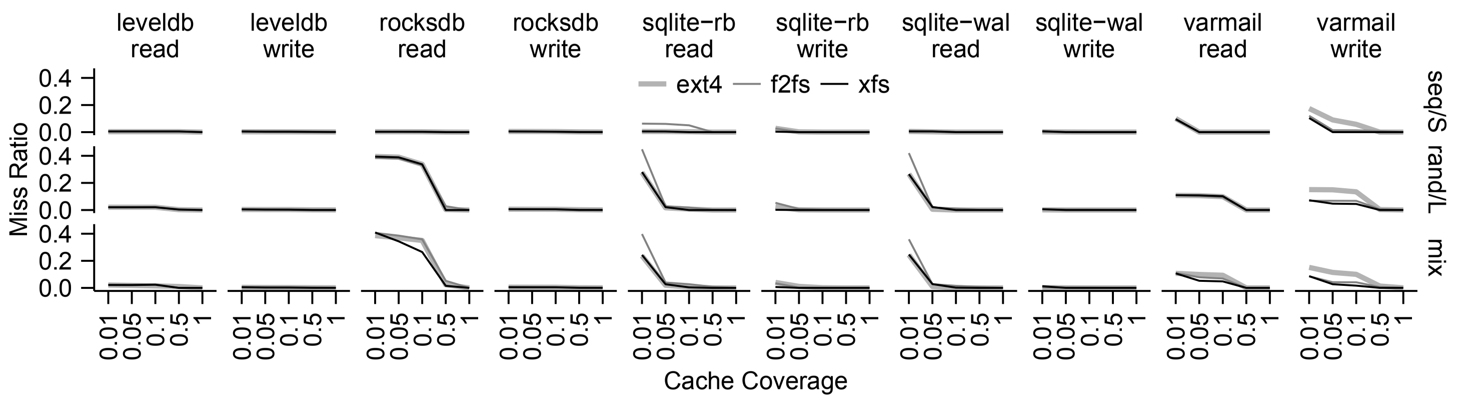


Figure 3.6: **Locality - Miss Ratio Curves.** *We set the cache to cover 1%, 5%, 10%, 50%, and 100% of the logical space.* `seq/S` *indicates sequential for databases and small set of files for Varmail.* `rand/L` *indicates random for databases and large set of files for Varmail.*

discards can significantly degrade sustainable performance.

3.4.2 Locality

We study locality by examining miss ratio curves [155, 156], obtained from *WiscSim* with an on-demand page-level mapping scheme based on DFTL [89]. 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 [81, 155, 156]. 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 3.6 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 3.4.1). 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 [58, 93]. 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 3.4.3. 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 3.4.1.

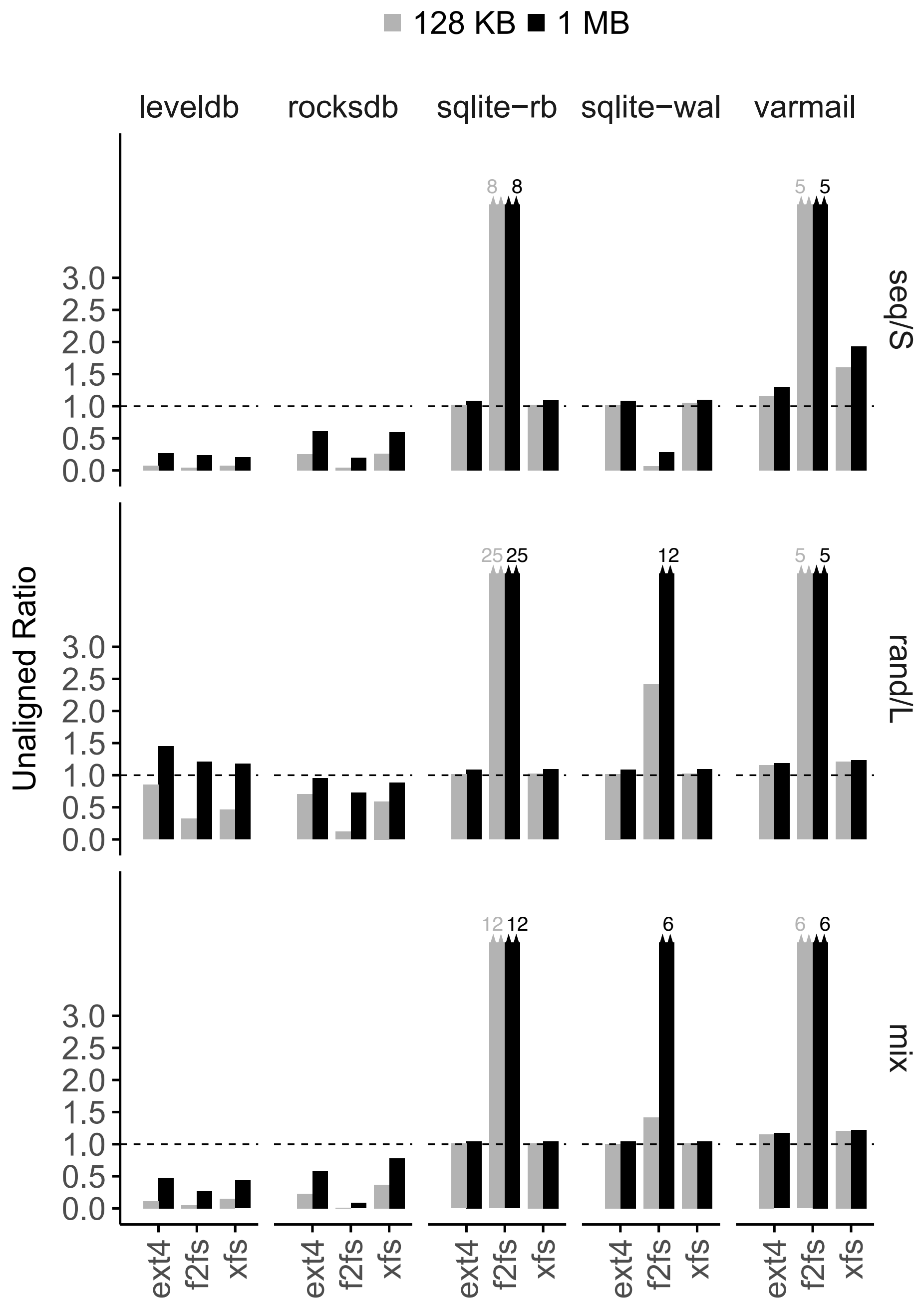


Figure 3.7: **Aligned Sequentiality - Unaligned Ratios.** *The dashed line indicates unaligned ratio of 1.0: the SSD internally has the same amount of data with unaligned mapping as the amount of application file data. Different colors indicate different block sizes.*

3.4.3 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 3.1) 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 [128] in *WiscSim*. Requests are not striped across channels in our multi-channel implementation of this scheme as striping immediately fragments mappings.

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 [44, 45]. We analyze different block sizes to understand how it impacts different combinations of applications and file systems.

Figure 3.7 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

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

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 [110]. For SQLiteRB, F2FS sees high unaligned ratios for two reasons. First, in every transaction, SQLiteRB overwrites the small journal file header and 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 3.4.2) 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.

3.4.4 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 [89], with added support for logical space segmentation and multiple channels.

Figure 3.8 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***

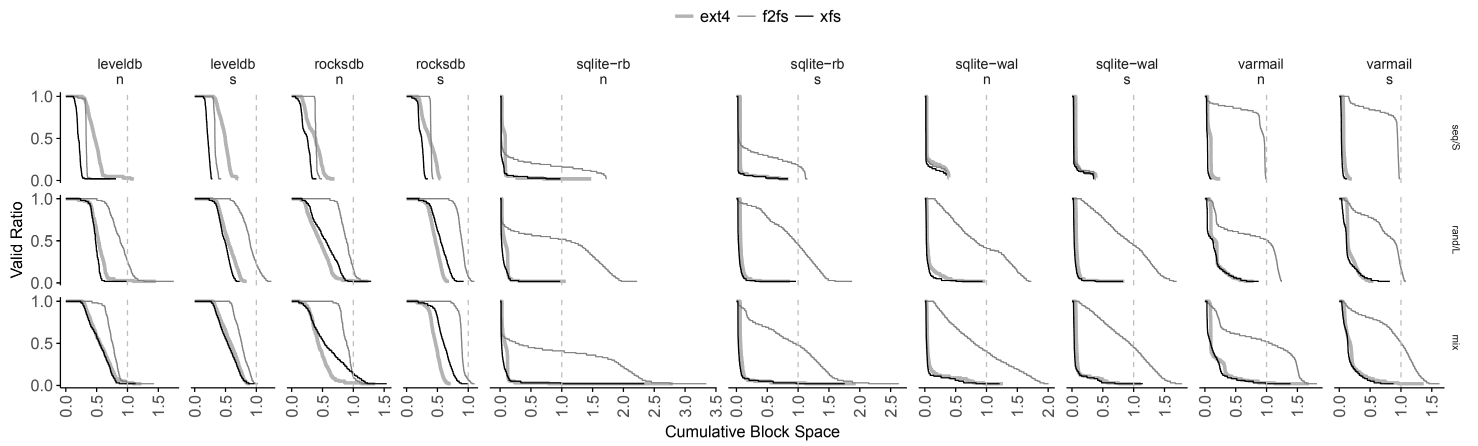


Figure 3.8: **Grouping by Death Time - Zombie Curves (Stable Valid Ratios).** *A zombie curve shows the sorted non-zero valid ratios of blocks. Labels* `'n'` *and* `'s'` *indicate non-segmented and segmented FTLs, respectively. The x-axis is normalized to logical space size. Zombie curves show important 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.*

garbage collection. It has long been believed that application-level log structure reduces garbage collection, but it does not. As shown in Figure 3.8, 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 [71]. 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 3.4.2) mixes all small files (â‰¤ 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) [93]. 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.

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 3.4.1 and 3.4.3. 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.

3.4.5 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 [66, 99, 122]. 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,

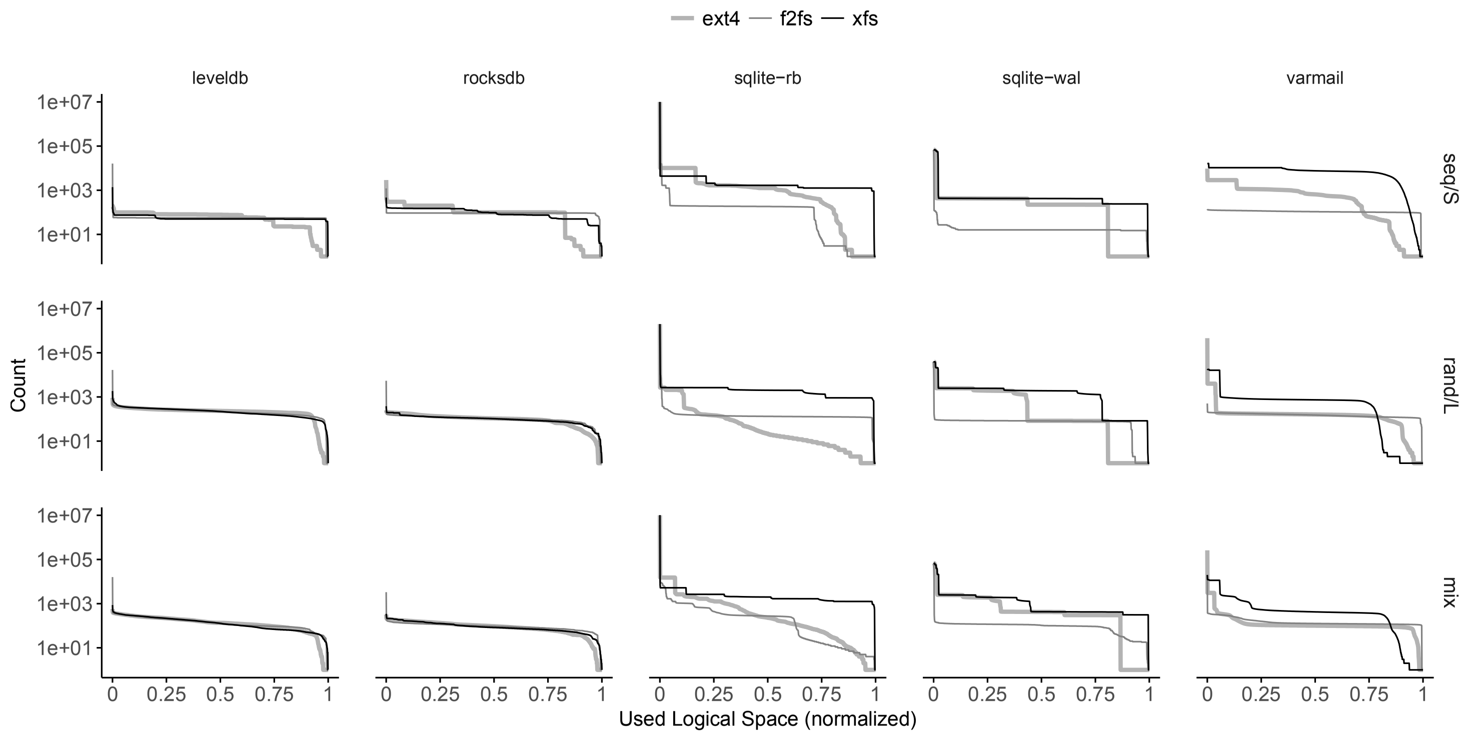


Figure 3.9: **Uniform Data Lifetime - Sorted Write Count of Logical Pages.** *Y axis scale is logarithmic; pages without data are excluded.*

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 3.9, 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 3.4.2).

3.4.6 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 [67, 70, 110, 121]. As a result, pessimistic views have spread and systems are built or optimized based on this assumption [35, 53, 68, 108].

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 [67, 106, 134], 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 [143]. 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.

3.5 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; the tool we have developed could benefit future SSD workload analysis.

4 Exploiting the SSD Contract

In our previous studies, we have found many violations to the unwritten contract of SSDs. As a result, a question arises: how can we design a system to avoid violations and exploit SSDs better?

One SSD feature that is worth exploiting is the rich internal I/O parallelism, which is behind the request scale rule in the SSD unwritten contract. According to our study, I/O parallelism has the most significant impact among all the rules and thus is of great importance. The rich internal I/O parallelism, which does not exist in HDDs, enables high bandwidth and low latency in SSDs. The high I/O performance brought by SSDs allows systems to rely more on the I/O and subsequently less on the expensive memory, reducing the cost of data processing systems. As a result, by exploiting SSDs well, we can build high-performance and cost-effective systems.

In this chapter, we demonstrate how to exploit SSDs by building high-performance and cost-effective systems that we refer to as a Tiny-Memory Processing System (TMPS). As a case study, we have built a complete search engine that only uses a small memory and a fast SSD, by applying our TMPS approach. Search engines naturally demand high throughput and low latency to support a large amount of concurrent and interactive queries; however, by carefully designs in data layout, early pruning, prefetching and space tradeoffs, we show that our search engine with tiny memory, named Vacuum, can effectively exploit SSDs and perform significantly better than the state-of-the-art search engine, Elasticsearch.

4.1 Search Engine Background

Search engines play a crucial role in retrieving information from large data collections. Although search engines are designed for text search, they are increasingly being used for data analytics because search engines do not need fixed data schemes and allow flexible data exploration. Popular modern search engines, which share similar features, include Elasticsearch, Solr, and Splunk [11]. Elasticsearch and Solr are open-source projects based on Lucene [4], an information retrieval library. Elasticsearch and Solr wrap Lucene by implementing practical features such as sharding, replication, and network capability.

We use Elasticsearch as our baseline as it is the most popular [11] and well-maintained project. Elasticsearch is used at Wikipedia and Github to power text (edited contents or source code) search [12, 52]. It is also widely used for data analytics [12]; for example, Uber uses Elasticsearch to generate dynamic prices in real time based on users' locations. Although we only study Elasticsearch, our results also apply to other engines, which share the same underlying library (i.e., Lucene) or key data structures. We also believe what we have found for building low-memory high-I/O search engines can apply to systems beyond search engines.

4.1.1 Data Structures

Search engines allow users to quickly find documents (e.g., text files, web pages) that contain desired information. Documents must be indexed to allow fast searches; the core index structure in popular engines is the *inverted index*, which stores a mapping from a term (e.g., a word) to all the documents that contain the term.

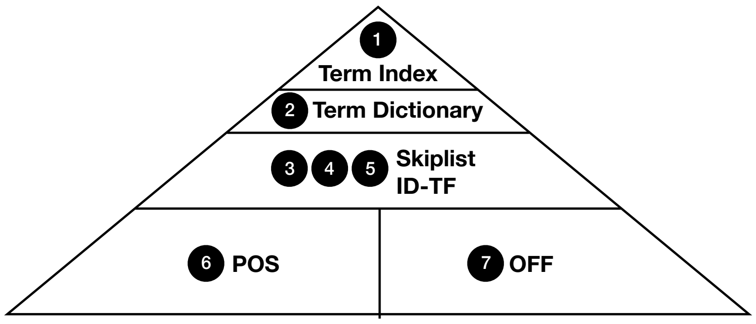


Figure 4.1: **Inverted Index in Elasticsearch.** *Term Index maps a term to an entry in Term Dictionary. A Term Dictionary entry contains metadata about a term (e.g., doc frequency) and multiple pointers pointing to files that contain document IDs and Term Frequencies (ID-TF), positions (POS), and byte offsets (OFF). The number in the figure indicates a typical data access sequence to serve a query. No.3, 4, and 5 indicate the access of skip lists, document ID and Term Frequencies. For Wikipedia, the sizes of each component are Term Index: 4 MB, Term Dictionary: 200 MB, Skiplist.ID.TF: 2.7 GB, POS: 4.8 GB, OFF: 2.8 GB.*

ID	**Text**
1	cheese is a dairy product. i am a dairy king.
2	office products
3	the products of dairy farms

Table 4.1: **An Example of Documents.** *An indexer parses the documents to build an inverted index; a document store will keep the original text.*

Figure 4.1 shows the data structures of Elasticsearch. To serve a query with a single term, Elasticsearch follows these steps. First, Elasticsearch locates a postings list by Term Index (1) and a Term Dictionary (2). The Term Index and Term Dictionary contain location information of the skip lists, document IDs, positions, and offsets (details below). Second, the engine will load the skip list, which contains more information for navigating document IDs, term frequencies, positions, and offsets. Third, it will iterate through the document IDs and use the corresponding term

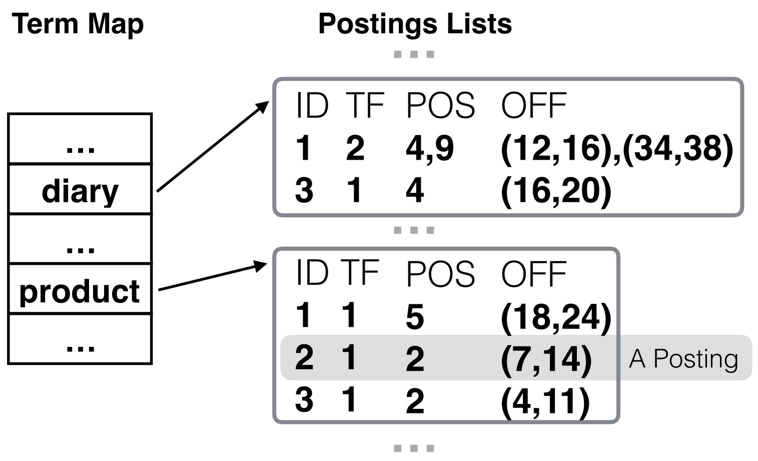


Figure 4.2: **An Example of Inverted Index.** *This figure shows the general contents of inverted index without specific layout information. Term Map allows one to look up the location of the postings list of a term.*

frequencies to rank documents. Fourth, after finding the documents with top scores, it will read offsets and document text to generate snippets.

An indexer in a search engine builds the inverted index. Table 4.1 shows an example of documents to be indexed. First, the indexer splits a document into tokens by separators such as space and punctuation marks. Second, the indexer transforms the tokens. A common transformation is stemming, which unifies tokens (e.g., worlds) to their roots (e.g., world). The transformed tokens are usually referred to as terms. Finally, the location information of the term is inserted to a list, called a postings list. The resulting inverted is shown in Figure 4.2.

A posting, as shown in Figure 4.2, contains the location information of a term in a particular document. Such information often includes a document ID, positions, and byte offsets of the term in the document. A position records, for example, that term "dairy" is the 4-th and 9-th token in document 1. Positions enable the processing of phrase queries: given a phrase such as "dairy product", we know that a document contains the phrase if the first term, "dairy", is the x-th token and the second term “product” is the $x + 1$-th one. An offset pair records the start and end byte address of a term in the original document. Offsets enable quick highlighting; the engine presents the most relevant parts (with the queried terms highlighted) of a document to the user. A posting also contains information such as term frequency for ranking the corresponding document.

Query processing includes multiple stages: document matching, ranking, phrase matching, highlighting; different types of queries go through different stages. For queries with a single term (e.g., “dairy”), an engine executes the following stages: iterating through document IDs in a term’s postings list (document matching); calculating the relevance score of each document, which usually uses term frequencies, and finding the top documents (ranking); and highlighting queried terms in the top documents (highlighting). For *AND* queries such as “dairy AND product”, which look for documents containing multiple terms, document matching includes intersecting the document IDs in multiple postings lists. For the example in Figure 4.2, intersecting $[1, 3]$ and $[1, 2, 3]$ produces $[1, 3]$, which are the IDs of documents that contain both “dairy” and “product”. For phrase queries, a search engine needs to use positions to perform phrase matching after document matching.

4.1.2 Performance Problems of Cache-Oriented Systems

Cache-oriented systems, such as Elasticsearch, cannot achieve high performance on a tiny-memory configuration because cache-oriented data structures introduce significant read amplification if memory is tiny.

Elasticsearch divides data of different stages into multiple locations and uses memory to cache the data in early stages, which are smaller and accessed more frequently than later stages (i.e., a cache-oriented design). Therefore, data for a particular stage is grouped; data of different stages is stored separately. Such cache-oriented systems are justified for slow

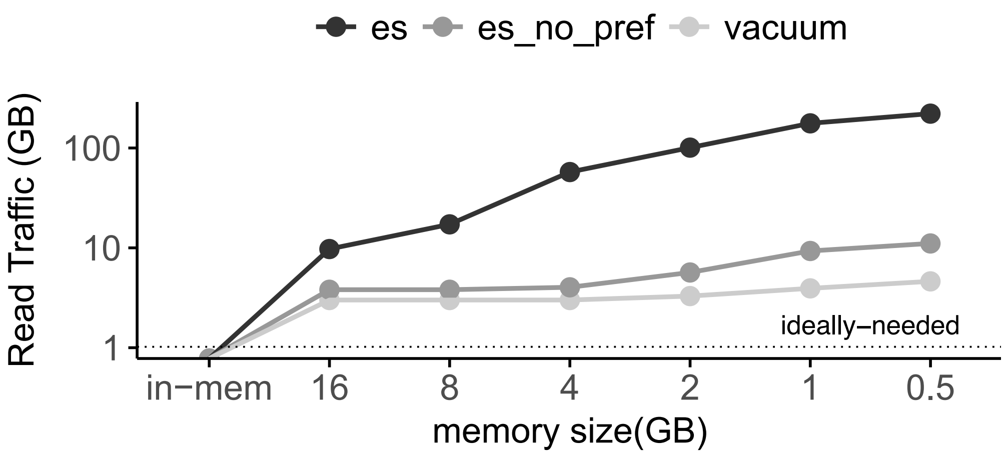


Figure 4.3: **Read Traffic of Search Engines.** *This figure shows read I/O traffic of various search engines as the size of memory decreases. Note that search engines only read while serving queries. The ideally-needed traffic is calculated by assuming an unrealistic byte-addressable storage device.*

storage, where *frequent cache misses are intolerable*; they assume that the system will run with a reasonable amount of memory as a cache.

If we run the cache-oriented Elasticsearch on a tiny-memory system, read amplification will be high. Figure 4.3 shows the I/O traffic of a realistic query workload on Wikipedia; as we decrease the size of memory, the I/O traffic increases significantly. The traffic increases because cache hits become cache misses, as expected. However, we find that putting a cache-oriented design on a tiny-memory system incurs unexpected high read amplification; we may reduce read amplification considerably if we redesign a search engine for tiny-memory systems.

4.2 Vacuum: A Tiny-Memory Search Engine

A tiny-memory processing system works on a large dataset with a small fraction of memory relative to the dataset size. As a result, the I/O traffic becomes large because every data access is a cache miss. Therefore, *reducing read amplification* is of the highest priority in a tiny-memory system. Another critical task to realize a tiny-memory system is to *hide I/O latency*, as retrieving data from SSDs may incur long latency. Finally, a tiny-memory system needs to *maximize I/O efficiency*; for example, issuing large I/O requests increases I/O efficiency because data bandwidth is higher with large requests [92].

We introduce several techniques that allow Vacuum to achieve high performance in tiny-memory configurations. First, *cross-stage data vacuuming* creates a data layout that combines data of different stages and stores it compactly. Second, we propose *two-way cost-aware filtering*, which employs special Bloom filters to prune early and reduce I/O for positions in the inverted index. The proposed Bloom filters are different to traditional Bloom filters; our Bloom filters are novel in that they are tightly integrated to the query processing pipeline and exploits unique properties of search engines. Third, we *adaptively prefetch* data to increase effective bandwidth (for I/O efficiency) and reduce query latency (for low latency). Unlike the prefetch (i.e., OS readahead) employed by Elasticsearch, our prefetch mechanism dynamically adjusts prefetch size to avoid reading unnecessary data. Fourth, we take advantage of the cheap and ample space of SSDs by *trading disk space for I/O speed*. For example, Vacuum aligns document to file system blocks to prevent the data from crossing multiple blocks unnecessarily, which speeds up I/O.

4.2.1 Cross-Stage Data Vacuuming

The key to building a low-memory high-I/O search engine is to reduce read amplification because the memory (cache) size is small and cache misses become the common (or only) case. As a result, we can only afford to read data that is needed definitely and immediately, as we have no memory for caching potential future data. In other words, we optimize

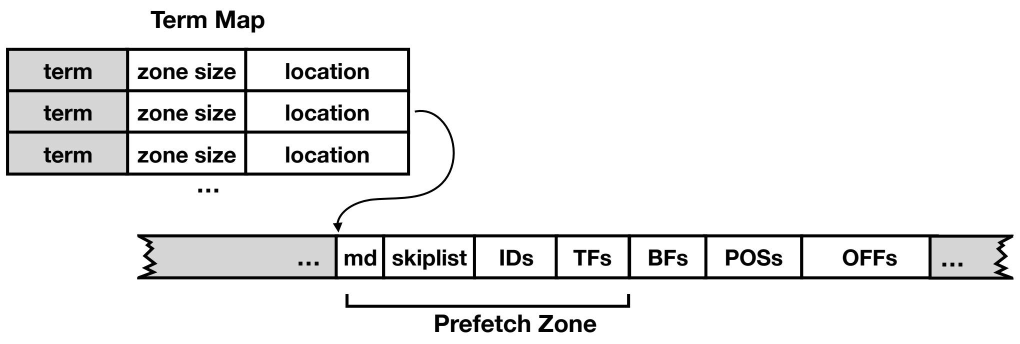


Figure 4.4: **Structure of Vacuum's Inverted Index.** *Contents of each postings list are placed together. Within each postings list, IDs, TFs and so on are also placed together to maximize compression ratio, which is the same as Elasticsearch.*

for now, not the future, in such tiny-memory systems. In tiny-memory systems, memory is not used as a cache, but rather a buffer for the CPU.

We propose a technique, called data vacuuming, to reduce read amplification for posting lists of small or medium sizes. The processing of such postings lists is critical because usually most of the postings lists fall into this category. For example, 99% of the postings lists in the representative Wikipedia are in this category (less than 10,000 documents are in the postings list). Also, search engines often divide large postings lists to smaller ones to reduce processing latency, which increase the quantity of postings lists in this category.

Vacuuming is similar in spirit to what a vacuum can do: pulling data (or dirt for a real vacuum) into a container. In our case, the containers are file system blocks. The vacuuming process groups data needed for different stages of a query into one or more continuous and compact blocks on the storage device, which increases block utilization when transferring data for a query. Figure 4.4 shows the resulting data structures after vacuuming; data needed for a query is placed in one place and in the order that they will be accessed. Essentially, the vacuumed data becomes a stream of data, in which each piece of data is expected to be used at most once. Such expectation matches the query processing in a search engine, in which multiple iterators iterate over lists of data. Such streams can flow through a small buffer efficiently with high block utilization and low read amplification.

Vacuumed data introduces significantly less read amplification than Elasticsearch in tiny-memory configurations for postings lists of small and medium sizes. Due to highly efficient compression, the space consumed by each postings list is often small; however, due to Elasticsearch's cache-oriented layout, the data required to serve a query is spread across multiple distant locations (Term Dictionary, ID-TF, POS, and OFF) as shown in Figure 4.1, increasing the I/O traffic and also the number of I/O operations. On the other hand, as shown in Figure 4.4, the vacuumed data can often be stored in the one block (e.g., 99% of the terms in Wikipedia can be stored in a block), incurring only one I/O.

As expected, the vacuumed data is not cache-friendly. Vacuum's data layout is inefficient for systems with a relatively large cache because the layout would waste a large amount of cache. The vacuumed layout uses cache inefficiently because the layout mixes frequently used data (e.g., skip lists) with a large amount of infrequently used data (e.g., offsets) in the same page. The cache space occupied by infrequently used data is better used for more frequently used data, as what is done by Elasticsearch. However, as we have stated, the vacuumed data layout is friendly to tiny-memory systems.

4.2.2 Two-way Cost-aware Filters

Phrase queries are pervasive and are often used to improve search precision [76]. Unfortunately, phrase queries put great pressure on a search engine's storage system, as they require retrieving a large amount of positions data (as described in Section 4.1). To build a low-memory search engine, we must reduce the I/O traffic of phrase queries.

Bloom filters, which can test if an element is a member of a set, are often used to reduce I/O; however, we find that plain Bloom filters, which are often directly used in data stores [22, 114, 123], surprisingly increases I/O traffic for phrase queries because individual positions data is relatively small due to compression and the corresponding Bloom filter can be larger than the positions data, which is the data to be avoided.

As a result, we propose special *two-way cost-aware* Bloom filters by exploiting unique properties of search engines to reduce I/O. The design of such Bloom filters is based on the observation that the data sizes of two (or more) terms in a phrase query are often very different. Therefore, we construct the Bloom filters in a way to allow us to pick the smaller Bloom filter to filter out a larger amount of positions data. In addition, we design special bitmap-based structure to store Bloom filters in order to further reduce I/O. This section gives more details on our Bloom filters.

Plain Bloom filter set contains terms that are after a particular term; for example, the `set.after` of term "dairy" in document 1 of Table 4.1 contains "product" and "king". To test the existence of a phrase "dairy product", an engine can simply test if "product" is in `set.after` of "dairy", without reading any positions. If the test result is negative, we stop and consequently avoid reading the corresponding positions. If the test result is positive, we must confirm the existence of the phrase by checking positions because false positives are possible in Bloom filters; also, we may have to use positions to locate the phrase within a document for highlighting.

However, we must satisfy the following two conditions for Bloom filters to reduce I/O. First, the ratio of negative tests must be high because in this case we only need to read Bloom filters, which can potentially reduce I/O. If the test is positive (a phrase may exist), we have to read both Bloom filters and positions, which increases I/O. Statistically, the ratio of the positive result is low for real phrase queries to Wikipedia: only 12% of

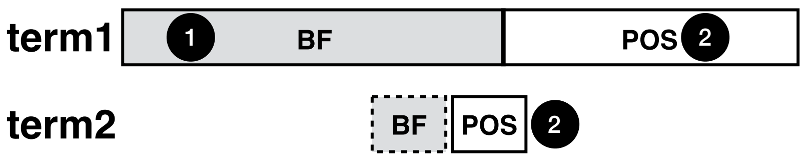


Figure 4.5: **Phrase Processing With Bloom Filters.** *Vacuum uses one of the Bloom filters to test the existence of a phrase (step 1) and then read positions for the positive tests to confirm (step 2).*

the tests are positive. Intuitively, the probability for two random terms to form a phrase in a document is also low due to a large number of terms in a regular document. The second condition is that the I/O traffic to the Bloom filters must be smaller than the traffic to positions needed to identify a phrase; otherwise, we would just use the positions.

Meeting the second condition is challenging because the sizes of Bloom filters are too large in our case, although they are considered space-efficient in other uses [22, 114]. Bloom filters can be larger than their corresponding positions because positions are already space efficient after compression (delta encoding and bit packing). In addition, Bloom filters are configured to be relatively large because their false positive ratios must be low. The first reason to reduce false positive is to increase negative test results, as mentioned above. The second reason is to avoid reading unnecessary blocks. Note that a 4-KB I/O block contains positions of hundreds of postings. If any of the positions are requested due to false positive tests, the whole 4-KB block must be read; however, none of the data in the block is useful. Through our evaluation, we find that storing 5 entries with a false positive probability of 0.0009 (9 bytes) in the Bloom filter offers a balanced tradeoff between space and test accuracy.

We now show how we can reduce I/O traffic to both Bloom filters and positions with cost-aware pruning and a bitmap-based store. To realize it, first we estimate I/O cost and use Bloom filters conditionally (i.e., cost-aware): we only use Bloom filters when the I/O cost of reading Bloom

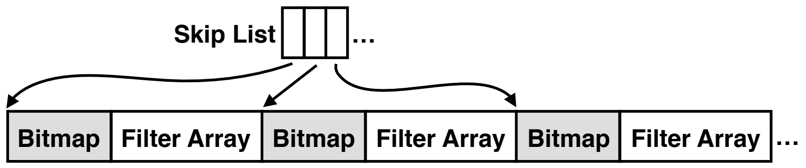


Figure 4.6: **Bloom Filter Store.** *The sizes of the arrays may vary because some Bloom filters contain no entries and thus are not stored in the array.*

filters is much smaller than the cost of reading positions if Bloom filters are not used. For example, in Figure 4.5, we will not use Bloom filters for query "term1 term2" as the I/O cost of reading the Bloom filters is too large. We estimate the relative I/O costs of Bloom filter and positions among different terms by the popularity of the terms, which is proportional to the sizes of Bloom filters and positions.

Second, we build two Bloom filters for each term to allow filtering in either direction (i.e., two-way): a set for all following terms and another set for all preceding terms of each term. This design is based on the observation that the data sizes of terms in a query are often very different. With these two Bloom filters, we can apply filters in forward or backward directions, whichever can reduce I/O. For example, in Figure 4.5, instead of using Bloom filters of term1 to test if term2 is *after* term1, we can now use Bloom filters of term2 to test if term1 is *before* term2. Because the Bloom filters of term2 are much smaller, we can apply it to significantly reduce I/O.

To further reduce the size of Bloom filters, we employ a *bitmap-based data layout* to store Bloom filters. Figure 4.6 shows the data layout. Bloom filters are separated into groups, each of which contains a fixed number of filters (e.g., 128); the groups are indexed by a skip list to allow skipping reading large chunks of filters. In each group, we use a bitmap to mark the empty filters and only store non-empty filters in the array; thus, empty Bloom filters only take one bit of space (in the bitmap). Reducing the space usage of empty filters can significantly reduce overall space usage of Bloom filters because empty filters are very common. For instance, about one-third of the filters for Wikipedia are empty. Empty filters of a term come from surrounding punctuations and stop words (e.g., “a”, “is”, “the”), which are not added to filters.

4.2.3 Adaptive Prefetching

Although the latency of NVMe devices is low, it is still much higher than that of memory. The relatively high latency of NVMe devices adds to query processing time, especially the processing of long postings lists, which demands a large amount of I/O. If we load one page at a time as needed, query processing will frequently stop and wait for data, which also increases system overhead. In addition, the I/O efficiency will be low due to small request sizes [92].

To mitigate the impact of high I/O latency and improve the I/O efficiency, we propose adaptive prefetching. Prefetching, a commonly used technique, can reduce I/O stall time, increase the size of I/O requests, and reduce the number of requests, which boosts the efficiency of flash devices and reduces system overhead. However, naive prefetching, such as the Linux readahead [19], used by Elasticsearch, will suffer from significant read amplification in a tiny-memory system. Linux unconditionally prefetches data of a fixed size (default: 128 KB), which causes high read amplification due to the diverse data sizes needed.

Prefetching in a tiny-memory system must adapt to the queries and the structures of persistent data. Among all data in the inverted index, the most commonly accessed data includes metadata, skip lists, document IDs, and term frequencies, which are often accessed together and sequentially; thus we place them together in an area called the *prefetch zone*. Our adaptive approach prefetches data when doing so can bring significant benefits. We prefetch when all prefetch zones involved in a query are larger than a threshold (e.g., 128 KB); we divide the prefetch zone into small prefetch segments to avoid accessing too much data at a time.

To enable adaptive prefetch, Vacuum employs prefetch-friendly data structures, as shown in Figure 4.4. A search engine should know the size of the prefetch zone before reading the posting list (so the prefetch size can be adapted). To enable such prefetching, we hide the size of the prefetch zone in the highest 16 bits of the offset in Vacuum's Term Map (The 48 bits left is more than enough to address large files). In addition, the structure in the prefetch zone is also prefetch-friendly. Data in the prefetch zone is placed by the order they are used, which avoid jumping ahead and waiting for data that has not been prefetched. Finally, compressed data is naturally prefetch-friendly. Even if there are data "holes" in the prefetch zone that are unnecessary for some queries, prefetching data with such holes is still beneficial because these holes are usually small due to compression and the improved I/O efficiency can well offset the cost of such small read amplification.

4.2.4 Trade Disk Space for Less I/O

Because tiny-memory systems reduce the amount of memory and the saved budget can be used to increase flash size substantially, tiny-memory systems can tolerate storage space amplification. Therefore, we can relax the space constraints and utilize the additional space to speed up the system. We apply this technique to Vacuum's document store, which is frequently accessed to generate snippets.

We compress each document individually in Vacuum, which often increases space usage but avoids reading and decompressing unnecessary documents. Compression algorithms, such as LZ4, achieve higher compression ratios when more data is compressed together. As a result, when compressing documents, engines like Elasticsearch put documents into a buffer (default size: 16 KB) and compresses all data in the buffer together. Unfortunately, decompressing a document requires reading and decompressing all documents before the document, leading to more I/O and computation. In Vacuum, we trade space for less I/O by using more space but reducing the I/O while processing queries.

In addition, we align compressed documents to the boundaries of file system blocks if the unaligned data would incur more I/O. A document may suffer from the block-crossing problem, where a document is unnecessarily placed across two (or more) file system blocks and requires reading two blocks during decompression. For example, a 3-KB data chunk has a 75% chance of spanning across two 4-KB file system blocks. To avoid this problem, Vacuum applies a simple heuristic to trade a small amount of space for I/O reduction: if aligning a compressed document reduces the block span, align it.

4.2.5 Implementation

We have implemented Vacuum with 11,000 lines of core code in C++, which allows us to interact with OS more directly than higher-level languages. We also rigorously conducted hundreds of unit tests to ensure the correctness of our code. The query processor in Vacuum is carefully optimized for high performance. For example, we switched from C++ virtualization to metaprogramming to avoid runtime costs. Data files are mapped by `mmap()` to avoid complex buffer management. Prefetch is implemented by `madvise()` with the `MADV_SEQUENTIAL` hint.

4.3 Evaluation

In this section, we evaluate Vacuum with WSBench, a benchmark suite we built, which includes synthetic workloads and a realistic workload.

First, we will analyze in detail how each of the proposed techniques in Vacuum is able to improve performance. Most importantly, we will show that the techniques are able to significantly reduce read amplification, which is key to building a tiny-memory processing system as data is offloaded to SSDs but SSDs offer limited bandwidth. We will show that: cross-stage data vacuuming reduces I/O traffic by 2.9 times by comparing Elasticsearch's data layout and the vacuumed data layout; two-way cost-aware Bloom filters reduce I/O traffic by 3.2 times by comparing Vacuum with and without Bloom filters; adaptive prefetching is able to prefetch only necessary data; trading disk space for less I/O reduces I/O traffic by 1.7 times by comparing the document store with and without such tradeoffs.

Second, we will show that the techniques that we propose improve end-to-end performance. For example, we compare Vacuum (with Bloom filters) and Vacuum (without Bloom filters) to show that our Bloom filters increase query throughput by up to 2.6x. We will also show that Vacuum delivers higher end-to-end performance than Elasticsearch, thanks to the combined effect of all our techniques. Specifically, Vaccum delivers up to 2.7 times higher query throughput and up to 16 times lower latency than Elasticsearch.

We conduct our experiments on machines with 16 CPU cores, 64-GB RAM and NVMe SSD (peak read bandwidth is 2.0 GB/s; peak IOPS is 200,000)[9]. Our OS is Ubuntu 16.04 with Linux 4.4.0. Our tiny-memory machine configuration is with 512-MB memory, which is the smallest memory size of AWS instances (t3.nano). 512-MB memory is truly tiny as the footprint of multi-thread applications can easily approach 512 MB, and we observe that the OS page cache has little cache effect with such small memory. We choose such a small memory to demonstrate the irrelevance of memory in our system and consequently the ability to scale with little memory. We use Cgroup tools to limit memory size for experiments.

4.3.1 WSBench

We create WSBench, a benchmark based on the Wikipedia corpus, to evaluate Vacuum and Elasticsearch. The corpus is a snapshot of English Wikipedia in May 2018, which includes 5.8 million documents and 5,785,696 unique terms (excluding stop words).

WSBench contains 24 workloads with various I/O characteristics, including 21 synthetic query workloads with different types of queries, e.g., single-term, two-term, and phrase queries. For each type of queries, WSBench produces different workloads by varying the popularity level (also known as document frequency: the number of documents in which a term appears) of the queried terms. In general, a high popularity level indicates a long postings list and large data size per query. WSBench also includes one realistic query workload extracted from production Wikipedia servers [51], as well as three workloads with different query types derived from the original realistic workload.

4.3.2 Impact of Proposed Techniques

We evaluate the proposed techniques in Vacuum by three types of synthetic workloads: single-term queries, two-term queries, and phrase queries. Such evaluations allow us to investigate how the proposed techniques impact various aspects of the system as different techniques have different impacts on workloads. We investigate low-level metrics such as traffic sizes to precisely show the impact and why the proposed techniques could lead to end-to-end performance gain.

4.3.2.1 Cross-stage Data Vacuuming

Cross-stage vacuuming can reduce the read amplification for all types of queries. Here we show its impact on single-term and two-term queries where vacuuming plays the most important role; phrase queries are dominated by positions data where two-way cost-aware Bloom filters play a more important role (we will soon show).

Figure 4.7 shows the decomposed read traffic for single-term queries. The figure shows that Vacuum can significantly reduce read amplification (indicated by lower `waste` than Elasticsearch); the reduction is up to 2.9 times. The reduction is more when the popularity level is lower because the block utilization is lower with lower popularity levels. To process queries with low-popularity terms, a search engine only needs a very small amount of data; for example, an engine only needs approximately 30 bytes of data to process the term "tugman" (popularity=8). To retrieve such small data, read amplification is inevitable as the minimal I/O request size is 4 KB, which leads to higher waste when the popularity level is low. However, we can minimize the read amplification. Elasticsearch, which employs cache-oriented a design, often needs three separate I/O requests for such queries: one to term index, one to document IDs/term frequency, and one to offsets. In contrast, Vacuum only needs one I/O request because the data is 'vacuumed' to one block.

For high popularity levels (popularity=100,000), the traffic reduction is inconspicuous because queries with very popular terms require a large amount of data for each stage (KBs or even MBs). In that case, the waste from splitting data into different stages in Elasticsearch is negligible.

Figure 4.8 shows the aggregated I/O traffic for two-term queries, which read two postings lists. Similar to Figure 4.7, we can see that Vacuum (`vacuum`) incurs significantly less traffic than Elasticsearch. In this figure, we show two configurations of Elasticsearch: one with prefetch (`es`) and one without prefetch (`es_no_pref`). Prefetch is a common technique to

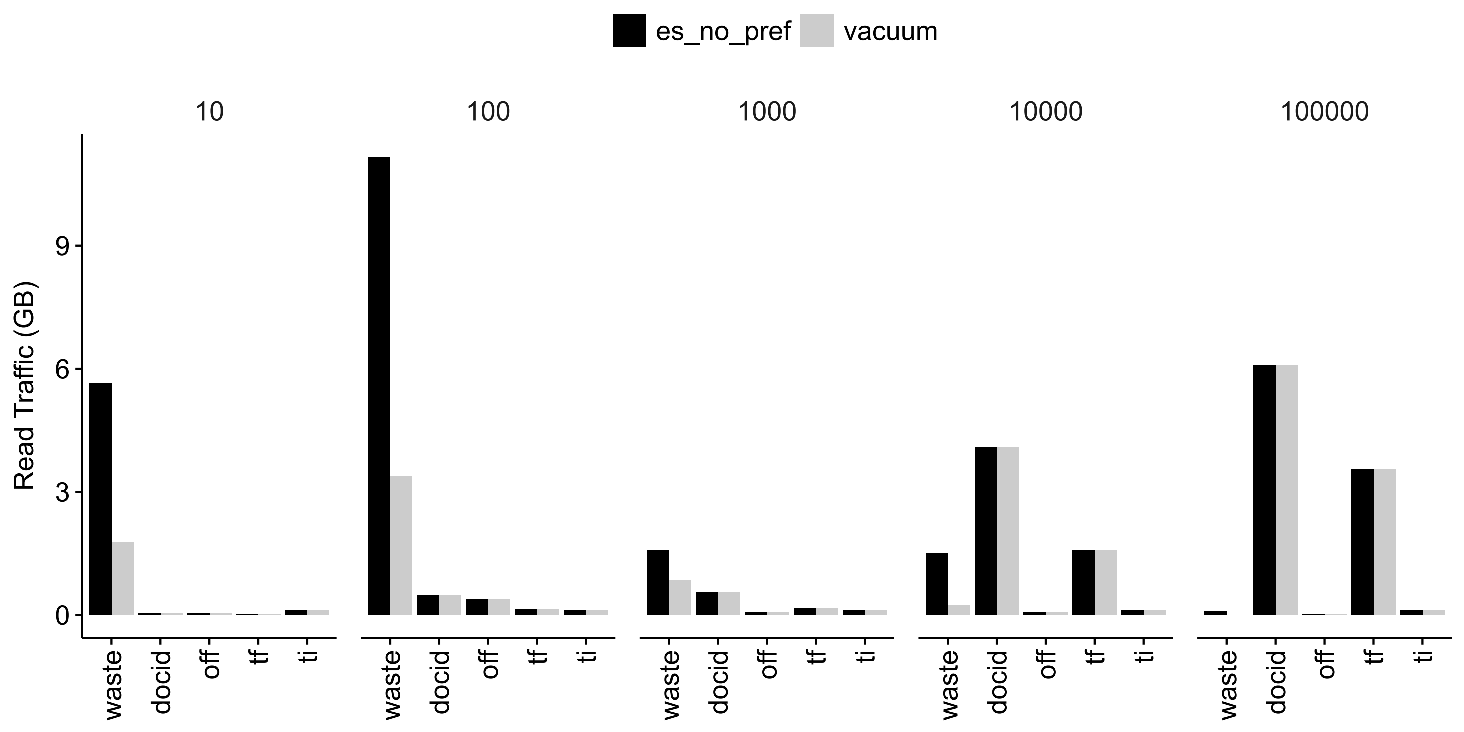


Figure 4.7: **Decomposed Traffic of Single-Term Queries.** `waste` *represents the data that is unnecessarily read;* `docid`, `off`, `skiplist`, `tf`, *and* `ti` *represents the ideally needed data of document ID, offset, skip list, term frequency, term index/dictionary. Positions is not needed in match queries and thus not shown. This figure only shows the traffic fromt inverted index, which relates to cross-stage data vacuuming; we investigate the rest of the traffic (document store) later.*

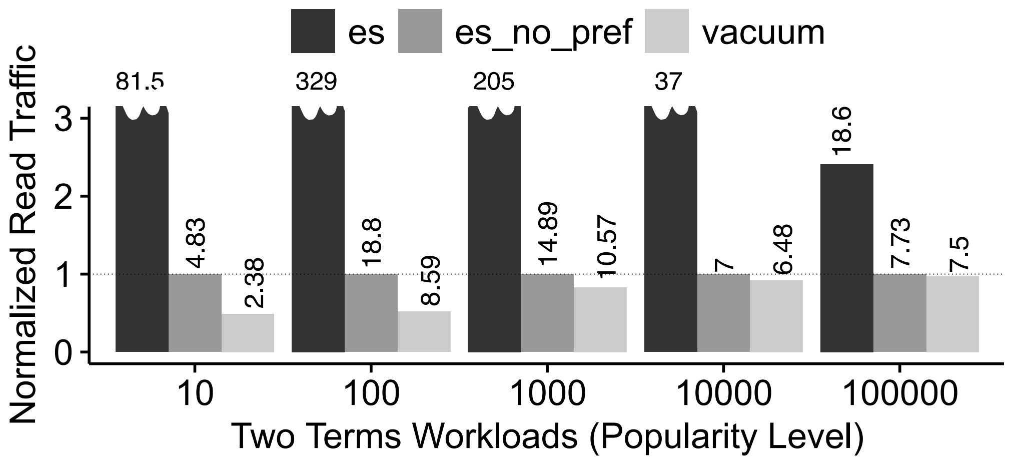


Figure 4.8: **I/O Traffic of Two-term Match Queries.** *The size (GB) is normalized to the traffic size of Elasticsearch without prefetching.*

boost performance in systems with ample memory; however, as shown in Figure 4.8, naive prefetch in Elasticsearch can increase read amplification significantly. Such a dilemma motivates our adaptive prefetch.

4.3.2.2 Two-Way Cost-Aware Bloom Filters

Two-way cost-aware Bloom filters only affect phrase queries as filters are only used to avoid positions data, which is used for phrase queries. We will show that Vacuum without two-way cost-aware Bloom filters demands a similar amount of data as Elasticsearch; the Vacuum with two-way cost-aware Bloom filters incurs much less I/O traffic than Vacuum without them and Elasticsearch, which demonstrates the effect of our novel Bloom filters.

Figure 4.9 shows the read amplification by the decomposed traffic in Elasticsearch, Vacuum without Bloom filters, and Vacuum with Bloom filters. The bars labeled with data type names show the data needed ideally; the `waste` bars show the data that is unnecessarily read (e.g., undesired positions in a 4-KB I/O block). First, we can see that applying filters sig-

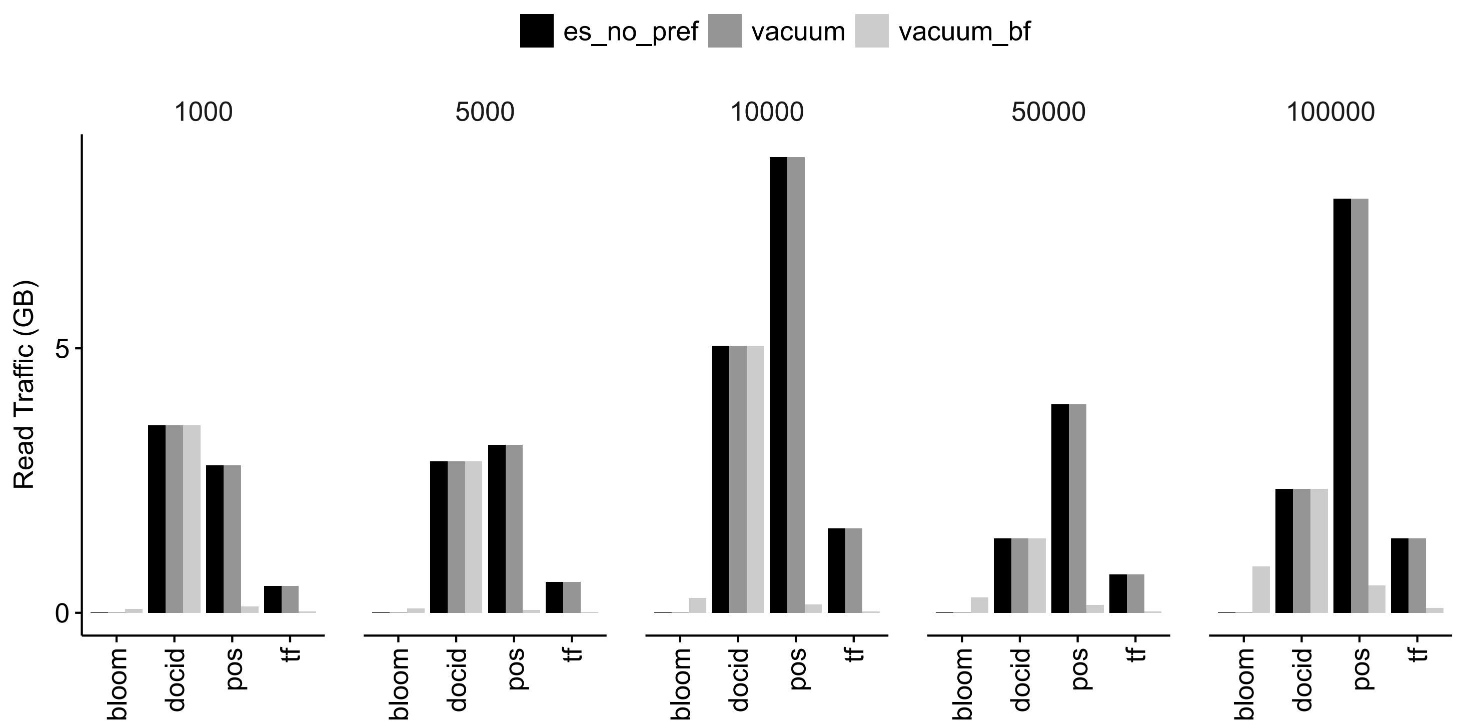


Figure 4.9: **Decomposed Traffic Analysis of Phrase Queries.** `waste` *bars show the data that is unnecessarily read; the rest of the bars show the size of data that is ideally needed. Results of Elasticsearch with prefetch is not shown as it is always much worse than Elasticsearch without prefetch.*

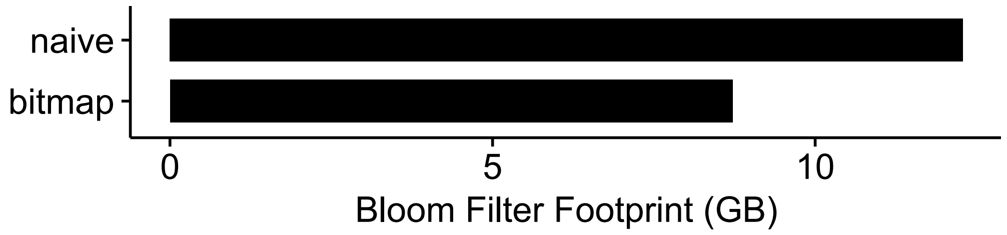


Figure 4.10: **Bloom Filter Footprint.** *Our bitmap-based layout reduces footprint by 29%*

nificantly reduce the data needed ideally (the size is calculated by assuming a byte-addressable SSD) which is shown by the reduced aggregated size of all the bars except `waste`. As shown in the figure, both Elasticsearch (`es`) and Vacuum without filters (`vacuum`) demands a very large amount of position data; in contrast, Vacuum with our two-way cost-aware filters (`vacuum_bf`) significantly reduces positions needed in all workloads. Surprisingly, we find that our filters also significantly reduce the traffic from term frequencies (`tf`), which is used to iterate positions (an engine needs to know the number of positions in each document in order to iterate to the positions of the destination document). The traffic to term frequencies is reduced because the engine no longer need to iterate many positions.

Note that the introduction of our Bloom filters only adds a very small amount of traffic to the Bloom filters (Figure 4.9), thanks to our two-way cost-aware design and the bitmap-based data layout of Bloom filters. The two-way cost-aware design allows us to prune by the smaller Bloom filter between the two Bloom filters of the two terms in the query. The bitmap-based layout, which uses only one bit to store an empty Bloom filter, significantly compresses Bloom filters, reducing traffic. We observe that 32% of the Bloom filters for Wikipedia are empty, which motivates the bitmap-based layout. Figure 4.10 shows that using bitmap-based layout reduces the Bloom filter footprint by 29%.

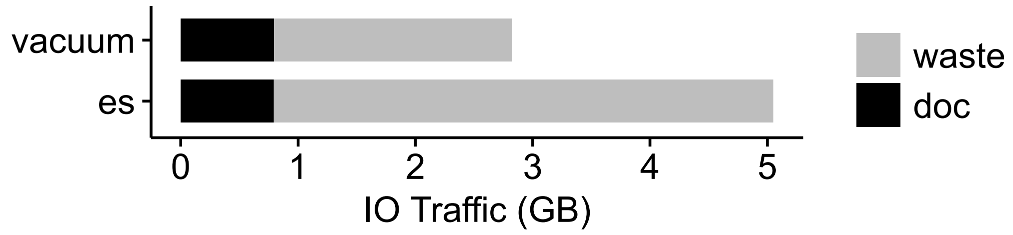


Figure 4.11: **Document Store Traffic.** `doc` *indicates ideal traffic size. The relative quantity between Elasticsearch and Vacuum is the same across different workloads; therefore, we show the result of one workload here for brevity (single-term queries with the popularity level = 10).*

4.3.2.3 Adaptive Prefetching

Adaptive prefetching aims to avoid the frequent wait for I/O and, in the meantime, reduce read amplification by prefetching only the data needed for the current queries. As shown in Figure 4.7 and Figure 4.8, Vacuum incurs less traffic than Elasticsearch that is with and without prefetching. As expected, by taking advantage of the information embedded in the in-memory data structure (Section 4.2.3), Vacuum only prefetches the necessary data. Later in this section, we will show that adaptive prefetching is able to avoid frequent I/O wait and improve end-to-end performance.

4.3.2.4 Trade Disk Space for Less I/O

The process of highlighting, which is the last step of all common queries, reads documents from the document store and produces snippets of the documents. Figure 4.11 show that Vacuum's highlighting incurs significantly less I/O traffic (42%) to the document store than Elasticsearch because in Vacuum documents are decompressed individually and are aligned, whereas Elasticsearch may have to decompress irrelevant documents and read more I/O blocks due to misalignment. The size of Vacuum's document store (9.5 GB) is 25% larger than that of Elasticsearch (7.6 GB); however, we argue that the space amplification can be well justified by the 42% I/O traffic reduction. Vacuum still wastes much traffic because the compressed documents in Wikipedia is small (average: 1.44 KB) but Vacuum has to read at least 4 KB.

4.3.3 End-to-end Performance

We examine various types of workloads in this section, including match queries (single-term and multi-term), phrase queries, and real workloads. For match queries, Vacuum achieves 2.5 times higher throughput than Elasticsearch with tiny memory, thanks to cross-stage data vacuuming. For phrase queries, Vacuum achieves 2.7 times higher throughput than Elasticsearch, thanks to our two-way cost-aware pruning. Vacuum achieves consistently higher performance than Elasticsearch for real-world queries.

4.3.3.1 Match Queries

We now describe the results for the single-term and multi-term queries. Because queries that match more than two terms share similar characteristics with two-terms queries, we only present the results of two-term queries here.

Figure 4.12 presents the single-term match QPS (Queries Per Second) of Vacuum, which is normalized to that of Elasticsearch with default prefetch (i.e., 128 KB). The default Elasticsearch is much worse than other engines when the popular levels are low because Elasticsearch incurs significant read amplification: it reads 128 KB of data when only a much smaller amount is needed (e.g., dozens of bytes). Elasticsearch without prefetch (`es_no_pref`) performs much better than `es_default`, thanks for much less read amplification.

Vacuum achieves higher throughput than Elasticsearch without prefetch (`es_no_pref`) for low/medium popularity levels, which accounts for a large portion of the postings lists; the speedup is up to 2.5 times. When

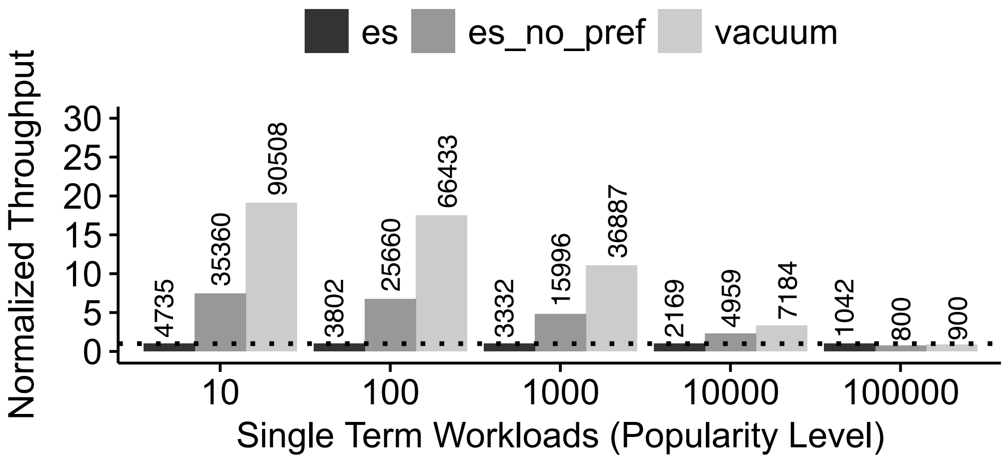


Figure 4.12: **Single Term Matching Throughput.** *The throughput (QPS) is normalized to the performance of Elasticsearch with prefetch (the default).*

popularity level is 100,000, the query throughput of Vacuum is 14% worse than Elasticsearch with default prefetching. We found that the difference is related to Vacuum's less efficient score calculation, which is not directly linked to I/O.

The query throughput improvement largely comes from the reduced I/O traffic as queries with low popularity levels are I/O intensive and I/O is the system bottleneck. Indeed, we see that the query throughput improvement is highly correlated with the I/O reduction. For example, Vacuum's query throughput for popularity level 10 is 2.6 times higher than Elasticsearch's; Vacuum's I/O traffic for the same workload is 2.9 times lower than Elasticsearch's.

Vacuum achieves better median latency and tail latency than Elasticsearch, thanks to adaptive prefetch and cross-stage data vacuuming. Figure 4.13 shows that Vacuum achieves up to 16x and 11x lower median latency than Elasticsearch, for median and tail latency respectively. The latency of Elasticsearch is longer due to similar reasons for its low query throughput. Elasticsearch's data layout incurs more I/O requests than

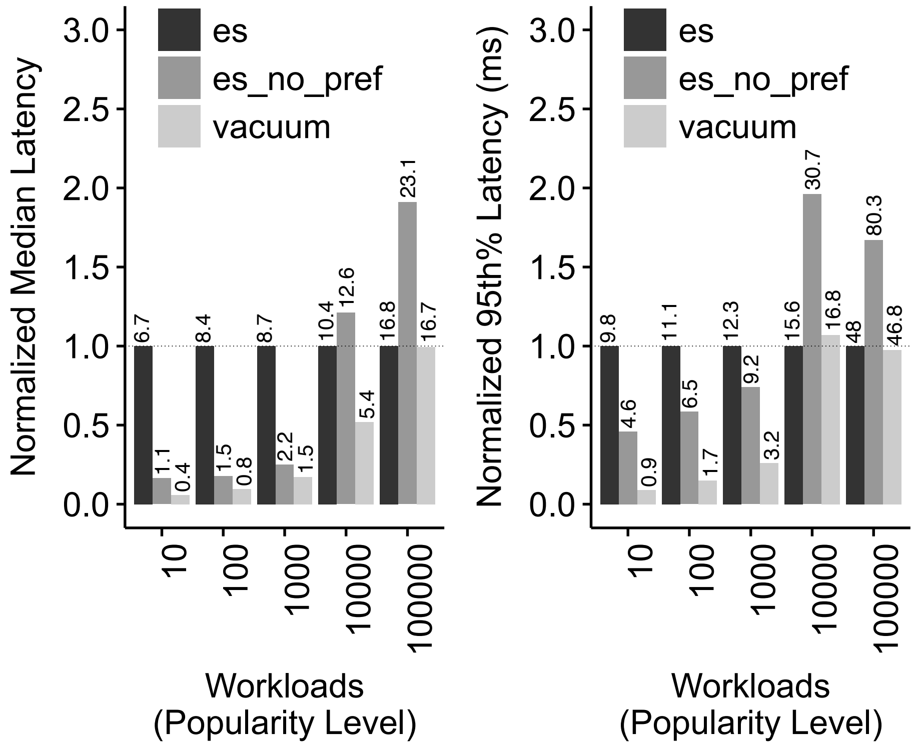


Figure 4.13: **Single Term Matching Latency.** *The latency (ms) is normalized to the median latency of Elasticsearch with default prefetching.*

Vacuum; the time of waiting for page faults adds to the query latency. In contrast, Vacuum's more compact data layout and adaptive prefetch incur minimal I/O requests, eliminating unnecessary I/O wait.

Vacuuming data layout also benefits two-term match queries. Figure 4.14. presents results for two-term match queries, which are similar to single-term match queries. As shown in the figure, Vacuum reduces by 17% to 51% of I/O traffic for workloads with popularity levels no more than 1,000. As a result, Vacuum achieves 1.5x to 2.6x higher query throughput compared with Elasticsearch. When a workload in-

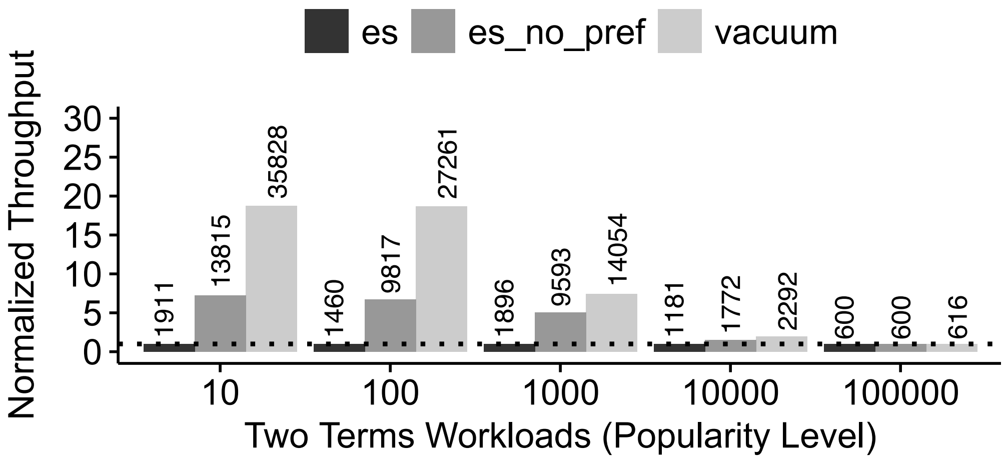


Figure 4.14: **Two Terms Intersecting Performance.** *The throughput (QPS) is normalized to the performance of Elasticsearch.*

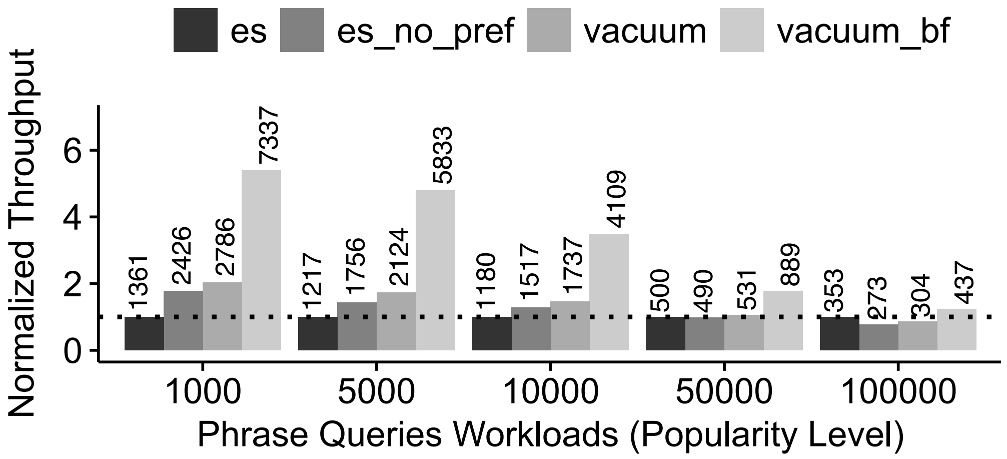


Figure 4.15: **Phrase Query QPS.** *The throughput (QPS) is normalized to the performance of Elasticsearch.*

cludes very popular terms (popularity=100,000), Vacuum's traffic reduction becomes negligible since data vacuuming has little effects.

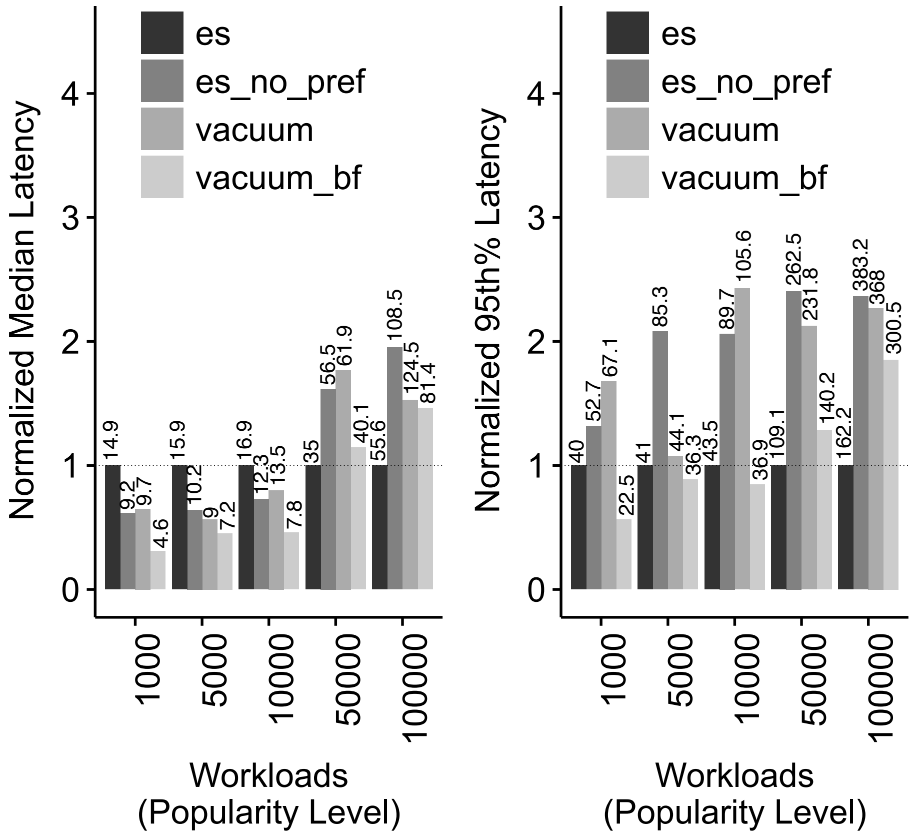


Figure 4.16: **Phrase Queries Latency.** *The latency (ms) is normalized to the corresponding latency of Elasticsearch with default prefetching.*

4.3.3.2 Phrase Queries

In this section, we will show that our two-way cost-aware Bloom filters make fast phrase query processing possible on tiny-memory systems. Specifically, Vacuum can achieve up to 2.7 times higher query throughput and up to 8 times lower latency, relative to Elasticsearch with tiny memory. To support early pruning, Vacuum needs to store 9 GB of Bloom filters (the overall index size increases from 18GB to 27GB, a 50% increase). We believe such space amplification is reasonable because flash is an order of magnitude cheaper than RAM.

WSBench produces the phrase query workloads by varying the probability that the two terms in a phrase query become a phrase. WSBench chooses one term from popular terms (popularity level is larger than 10,000); then it varies the popularity level of another term from low to high. As the popularity increases, the two terms are more likely to co-exist in the same document, also more likely to appear as a phrase in the document.

Figure 4.15 presents the phrase queries results among workloads in Elasticsearch (`es` and `es_no_pref`), Vacuum (`vacuum`), and Vacuum with two-way cost-aware pruning (`vacuum_bf`). The QPS of the basic Vacuum (`vacuum` and Elasticsearch with no prefetch (`es_no_pref`) are very similar because our techniques in the basic Vacuum (cross-stage data vacuuming, adaptive prefetch, trading space for less I/O) have little effect for phrase query. Vacuum with two-way cost-aware Bloom filters achieves from 1.3x to 2.7x higher query throughput than that of basic Vacuum, thanks to significantly lower read amplification brought by the filters.

Figure 4.16 shows that Bloom filters can significantly reduce latency (`vacuum` vs. `vacuum_bf`); also Vacuum reduces median and tail latency by up to 3.2x and 8.7x respectively, compared to Elasticsearch. The reduction is more evident when the probability of forming a phrase is lower (low popularity level) because the Bloom filters are smaller in that case. Interestingly, Elasticsearch with OS prefetch (`es`) achieves the lowest latency when the probability of forming phrases is higher. The latency is lower because the OS prefetches 128 KB of positions data and avoids waiting for many page faults, although the large prefetch increases read amplification. In contrast, Elasticsearch without prefetch (`es_no_pref`) and Vacuum do not prefetch; thus they may have to frequently stop query processing to wait for data (Vacuum's adaptive prefetch does not prefetch position data due to fragments of unnecessary data.). However, the reduction of latency comes at a cost: although the latency of individual queries is lower, the query throughput is also lower due to the read am-

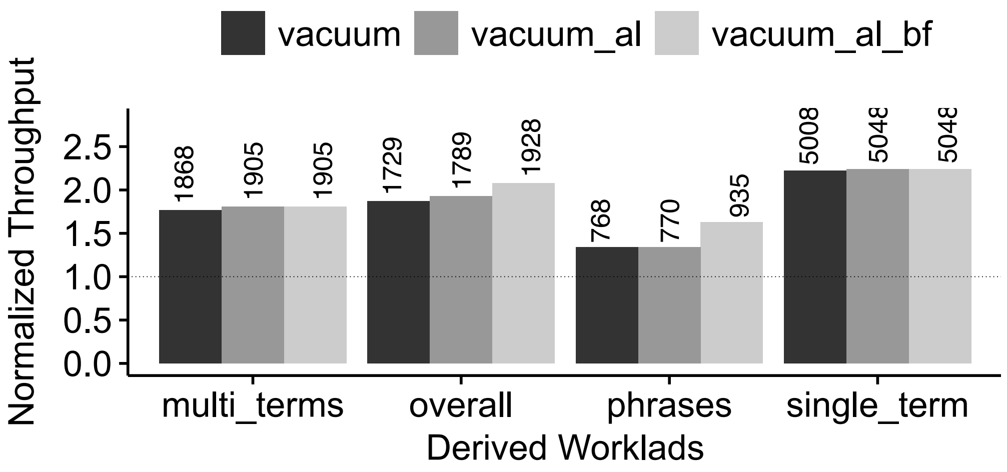


Figure 4.17: **Throughput of Derived Workloads.** *The throughput (QPS) is normalized to the throughput of Elasticsearch without Prefetching*

plification caused by prefetch (Figure 4.15).

Interestingly, our Bloom filters also speed up the computation of phrase checking. The search engine checks if a phrase in a document by testing the Bloom filter, which is done mathematically. In the common case that the Bloom filter is empty, Vacuum can even check faster because an empty Bloom filter is marked as 0 in the bitmap and we only need to check this bit. As a result, in addition to avoiding reading positions, Bloom filters also allow us to avoid the subsequent computations on the positions.

4.3.3.3 Real Workload

WSBench also includes realistic workloads. We compare Elasticsearch and Vacuum's query throughput on the real query log; we also split the query log into different types of query workloads to examine the performance closely.

Vacuum performs significantly better than Elasticsearch as shown in Figure 4.17. For example, for single-term queries, Vacuum achieves as high as 2.2x throughput compared to Elasticsearch. We observe that around

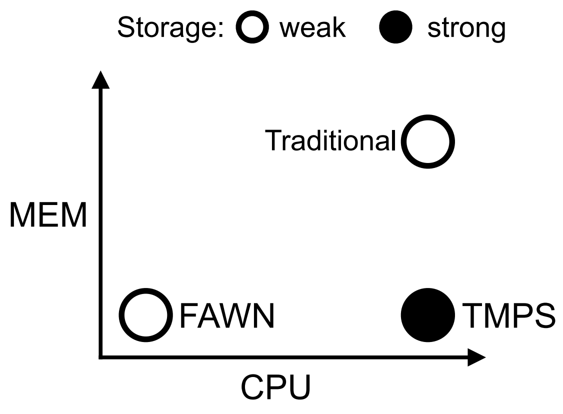


Figure 4.18: **Classes of Systems.** *The axes indicate the capability of memory and CPU.*

60% queries in the real workload are of popularity less than 10,000, which benefits from our cross-stage data vacuuming. For multi-term match queries, vacuumed data layout also helps to increase throughput by more than 60%. For phrase queries extracted from the realistic workload, Vacuum with Bloom filters increases throughput by more than 60% compared to Elasticsearch. Note that Vacuum cannot achieve 2.7x higher throughout as shown in our synthetic phrase queries because, in this real workload, many phrases are the names of people, brand, or events and so on. Among these names, many terms are unpopular terms that are not I/O intensive, where pruning has limited effect. Finally, the overall performance of Vacuum is similar to that of real single-term query log because single-term queries occupy half of the overall query log.

4.4 Discussion

In this section, we discuss several questions regarding tiny-memory search engines and tiny-memory processing systems.

How do the proposed techniques affect indexing? The proposed techniques, which optimize query processing, can add overhead to engine indexing; however, the effect is limited. Cross-stage data vacuuming does not add overhead as it simply places the same data in different places. Building two-way cost-aware Bloom filters requires additional computation, which is very manageable as we have seen in our experiments. Adaptive prefetching only employs existing information and thus does not add computation overhead. Aligning document store and individually compressing documents have little effect on computation. Overall, indexing is a far less frequent operation than query processing in a search engine and we believe the overhead can be well justified by the significant performance improvements on query processing.

Why can tiny-memory systems (low-memory high-I/O) achieve high performance? There are several reasons. First, thanks to modern high-bandwidth storage devices, frequent cache misses are now tolerable. Previously, slow devices such HDDs can only process one request at a time and may incur high (millisecond) latencies. Now, fast devices such as NVMe SSDs can process hundreds of requests at a time with latency on the order of microseconds, feeding GB/s of data to CPU. Second, some applications can be re-constructed specifically to demand less I/O bandwidth so that they can run on tiny-memory systems well, as we have done for search engines in this paper. In addition, I/O latency can be hidden by overlapping computation and I/O.

What is the role of memory in a tiny-memory system? In a tiny-memory processing system, we downplay the role of memory significantly, keep strong CPUs, and employ high-performance SSDs; in TMPS, memory no longer serves as a cache, but rather a buffer that allows data to flow through. Figure 4.18 shows a comparison with relevant systems. Traditional systems equipped with a weak storage system (i.e., HDDs) rely on large memory as a cache to achieve high performance. Low-cost systems, exemplified by FAWN (Fast Array of Wimpy Nodes) [57], use weak hardware components to achieve high power efficiency. TMPS aims to achieve high performance and low cost by exploiting modern SSDs with high bandwidth and reducing memory.

4.5 Conclusions

In this chapter, we propose to exploit SSDs to build tiny-memory processing systems to reduce the cost of search engines; we propose four techniques to reduce read amplification, latency and increase I/O efficiency, in order to take full advantage of SSDs. We design and implement a new search engine, Vacuum, that reduces read amplification by up to 3.2 times, reduces latency by up to 16 times, and increases query throughput by up to 2.7 times, when compared to a state-of-the-art engine. Vacuum exemplifies an approach that allows us to efficiently exploit SSDs to build better modern systems.

5 Related Work

5.1 Finding Violations of the HDD Contract

Chopper is a comprehensive diagnostic tool that provides techniques to explore file system block allocation designs and find the violations of the HDD unwritten contract. It shares similarities and has notable differences with traditional benchmarks and with model checkers.

File system benchmarks have been criticized for decades [151, 152, 154]. Many file system benchmarks target many aspects of file system performance and thus include many factors that affect the results in unpredictable ways. In contrast, *Chopper* leverages well-developed statistical techniques [139, 140, 157] to isolate the impact of various factors and avoid noise. With its sole focus on block allocation, *Chopper* is able to isolate its behavior and reveal problems with data layout quality.

The self-scaling I/O benchmark [72] is similar to *Chopper*, but the self-scaling benchmark searches a five-dimension workload parameter space by dynamically adjusting *one parameter* at a time while keeping the rest constant; its goal is to converge all parameters to values that uniformly achieve a specific percentage of max performance, which is called a *focal point*. This approach was able to find interesting behaviors, but it is limited and has several problems. First, the experiments may never find such a focal point. Second, the approach is not feasible given a large number of parameters. Third, changing one parameter at a time may miss interesting points in the space and interactions between parameters. In contrast, *Chopper* has been designed to systematically extract the maximum amount of information from limited runs.

Model checking is a verification process that explores system state space [77]; it has also been used to diagnose latent performance bugs. For example, MacePC [105] can identify bad performance and pinpoint the causing state. One problem with this approach is that it requires a simulation which may not perfectly match the desired implementation. Implementation-level model checkers, such as FiSC [159], address this problem by checking the actual system. FiSC checks a real Linux kernel in a customized environment to find file system bugs; however, FiSC needs to run the whole OS in the model checker and intercept calls. In contrast, *Chopper* can run in an unmodified, low-overhead environment. In addition, *Chopper* explores the input space differently; model checkers consider transitions between states and often use tree search algorithms, which may have clustered exploration states and leave gaps unexplored. In *Chopper*, we precisely define a large number of factors and ensure the effects and interactions of these factors are evenly explored by statistical experimental design [119, 139, 140, 157].

5.2 Finding Violations of the SSD Contract

Our dissertation 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 [67, 106, 110, 121]. A recent study by Yadgar et al. [158] 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 [110]. 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 [89, 98, 162], including the recent study by Yadgar et al. [158], 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 [107, 110] cannot accurately link the application and file system to hidden internal behaviors of the SSD. In addition, studies that benchmark black-box SSDs [67, 71] or SSD simulators [90] 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.

5.3 Exploiting the SSD Contract

Much work has gone into building cost-effective storage systems by coupling SSDs with reduced RAM. FAWN-KV [56] is a power-efficient key-value store with wimpy CPUs, small RAM and some flash. SILT [114] is another SSD-based key-value store, which employs innovative in-memory index and persistent data layout to reduce memory requirement. Facebook [85] proposes yet another SSD-based key-value store to reduce the consumption of DRAM by small block sizes with partitioned index, aligning blocks with physical NVM pages and reducing the interrupt latency with adaptive polling. Vacuum shares the same general goal of reducing system cost with the above systems; however, Vacuum is a complete data processing system with more complex data manipulation and data structures than data stores (e.g., key-value store). Such complete and holistic system designing exposes new opportunities and interesting insights; for example, while using Bloom filters is straightforward in a key-value store, using them in Vacuum requires understanding the search engine pipeline, which leads us to the novel two-way cost-aware Bloom filter for search engines.

Accelerated flash storage is also proposed to support external processing, such as BlueDBM[101] and GraFBoost [102]. While these solutions require specific hardware support, Vacuum only focuses on optimizations from application data layout and processing strategy. A study of the memory hierarchy for a search engine is presented in [63]. Many proposed techniques for search engines seek to reduce the overhead/-cost of query processing for different workloads or different scenarios[60–62, 65, 78, 113, 153]. These techniques may be adapted for Vacuum to further improve its performance.

6 Conclusions

The volume of data is becoming more substantial and more relevant to our lives; therefore, fast data processing is critical. Data processing relies on the storage stack to store and retrieve data; one key component of the storage stack is the storage device, where data is stored. Although storage devices present the same block interface, different ways of using the interface could lead to vastly different performance because performance relies on the "unwritten performance contracts", which are rules one should follow to achieve high performance.

HDDs and SSDs present very different contracts due to their distinct internal structures. HDDs physically move disk heads to access data on magnetic platters. On the other hand, SSDs do not contain moving parts; data is stored on an array of NAND flash, which presents special properties such as "erasing flash block before rewriting". To manage such special media, SSDs employ a complex piece of software, FTL, which makes SSDs more complex. As a result, SSDs has more intrinsic performance properties.

Although the same software can interact with both HDDs and SSDs via the same block interface, the performance utilization can be significantly different due to the different unwritten contracts of HDDs and SSDs. On HDDs only the locations of consecutive accesses by the software matter; however, on SSDs, the contract-violating writes made a few days ago may trigger background activities (e.g., garbage collections) today, leading to unexpected low performance.

Software (i.e., contractors of storage devices) should be constructed differently to honor the differences between HDDs and SSDs. For example, we find that Linux is conservative in sending requests to storage devices, probably due to the decade-long assumption of HDDs underneath. With SSDs, operating systems can be more aggressive as SSDs are much more capable, thanks to the high internal I/O parallelism. In addition, because I/O becomes faster, the software overhead will occupy a large portion of the end-to-end data access time. Consequently, we must rethink the software design.

In this dissertation, we show how software should be constructed to maximize the performance of the underlying storage devices, by finding the violations of the unwritten contracts and by building a new system to better exploit the contracts. Experimenting is one key method of our studies as it allows us to explore and understand the complex interactions between software and the storage hardware; without the carefully designed experiments, it would be impossible to systematically reveal the subtle interactions. To accelerate the explorations, we build tools, which can be applied to future system designs. To demonstrate the exploitation of the unwritten contract, we build a new system (i.e., Vacuum) that optimizes the interaction between software and SSDs.

In this chapter, we first summarize our work on finding violations of the unwritten contracts and exploiting the contracts. We then introduce future work and the lessons learned from the studies. Finally, we will conclude.

6.1 Summary

This dissertation consists of two parts. In the first part, we find violations of the HDD unwritten contract and the SSD unwritten contract. In the second part, we exploit the SSD unwritten contract to build a high-performance and cost-effective system. We now summarize these two parts.

6.1.1 Finding Violations of Unwritten Contracts

In the first part of the dissertation, we systematically find violations of the unwritten contracts of HDDs and SSDs.

We begin by identifying the violations of HDDs. Thanks to their low costs, HDDs are widely deployed in large-scale systems, where tail latency is critical for overall performance. Violations of the HDD unwritten contract could contribute to the tail latency; however finding rare, corner-case behaviors that lead to long tail latency is challenging. To effectively find the violations, we propose to use statistical tools, such as sensitivity analysis; we focus on the block allocators of file systems, which determines the data layout and thus how data is accessed on disk. We find four design issues in the popular Linux ext4 file system, which could lead to long tail latency. Our subsequent fixes of the issues cut the size of the tail by an order of magnitude, producing consistent and satisfactory file layout that reduce data access latency.

After finding violations of HDD unwritten contract, we continue to find violations of the SSD unwritten contract. We start by clearly defining the five rules of the unwritten contract. Then, we conduct a vertical analysis of various applications, file systems atop a detailed SSD simulator, in order to find the violations of the SSD unwritten contract and the root causes of the violations. As a result, we find 24 observations. Some of the observations confirm common beliefs; some are surprising. For example, the Linux page cache throttles read bandwidth on SSDs due to legacy designs. We hope that our observations from the thorough vertical analysis can shed lights into future storage system design.

6.1.2 Exploiting Unwritten Contracts

From our studies of the violations to the SSD unwritten contract, we find that the contract is often violated, underutilizing the SSD. Among the five rules of the unwritten contract of the SSD, request scale has the most significant impact on immediate performance; violating the request scale rule can lead to underutilized I/O parallelism in the SSD and significant performance loss. Therefore, we explore ways to effectively take advantage of the internal I/O parallelism of SSDs.

Interestingly, if the I/O parallelism is well exploited, the resulting high data bandwidth will make cache misses tolerable, which allows us to build systems with tiny cache and still achieve high performance. Such systems, which we refer to as Tiny Memory Processing Systems (TMPS), significantly reduce memory usage and make large-scale systems more cost-effective. We demonstrate the TMPS approach through a case study of search engines; we build a search engine, Vaccum, that exploits SSDs efficiently.

Vacuum employs several carefully designed techniques. First, we design a new data layout that is based on the assumption that cache misses are common. The new data layout significantly reduce read amplification and thus make tiny-memory systems feasible. Second, we propose two-way cost-aware bloom filters, which prunes data by checking multiple bloom filters in a cost-aware fashion and reduce the read traffic of phrase queries. Third, Vacuum performs adaptive prefetching that adjusts the size of prefetches by the specific query. Adaptive prefetch increases I/O efficiency and reduces read amplification. Finally, we trade space for less read traffic, taking advantage of the large space of SSDs with the budget.

For example, we compress documents individually, which lead to large space usage, but small read traffic, as we can avoiding entangled documents.

Vacuum is very efficient at exploiting SSDs. Comparing with Elasticsearch, a very popular open-source search engine, Vacuum reduces read amplification by up to 3.2 times, query latency by up to 16 times, and increases query throughput by up to 2.7 times.

6.2 Future Work

We believe that system builders should apply more statistics tools to more components of computer systems. Statistics tools such as the ones used in this dissertation (e.g., Latin hypercube design and Sensitivity Analysis) and in a bigger scope (e.g., Design of Experiments and Uncertainty Quantification) have been proved to help building better products, such as cars, by quantitatively studying how products should be built so that they can perform well in an uncertain environment, where temperature and road conditions may vary. Computer systems are the same as such products: we build computer systems so that they perform well when they are deployed in an uncertain environment, where the workloads, the components interacted and hardware may vary. More rigorous studies using statistics will allow us to build systems with better tradeoffs and more predictable performance.

We believe that the process of diagnosing performance problems could be further automated by better tools. Currently, we often pinpoint the root cause of a problem by manually designing and running additional experiments to “zoom in”. The process is tedious, ad hoc and requires strong domain knowledge in related components. We think that there could be a triage framework with built-in domain knowledge that can automatically plan and run experiments to narrow down the problem to a sufficiently small component, or even to a specific line of code.

More data processing systems could benefit from the TMPS (Tiny-Memory Processing System) approach, which redesigns a system to offload data from RAM to SSDs to reduce the cost of the system. Although the bandwidth of SSDs is much lower than RAM, the bandwidth of SSDs is sufficient for many data processing systems. Many systems seem to demand high data bandwidth because they are evaluated in an isolated way; for example, evaluating a key-value store, which is only one of many components of a system, may show high data bandwidth demand. However, if we take a holistic view, a full system usually conducts computation, or machines in the system communicate over the slow network, which limits the internal bandwidth demand to an amount that can be satisfied by SSDs. Even if one SSD is not enough, one can design the system in a way to exploit the higher aggregated bandwidth from multiple SSDs. As data becomes more substantial and requires faster processing, the TMPS approach encourages system designers to make new tradeoffs with new hardware, to reduce the cost of large-scale systems.

6.3 Lessons Learned

In this section, we present the general lessons that we learned from analyzing and building complex systems.

6.3.1 Systems studies need to be more systematic

System designing is often considered as art instead of rigorous science probably because systems are complex “living organs”. There are many components interacting with each other in a system; a system will also interact with other systems. A great deal of uncertainty is involved in designing a system. As a result, system designers often rely on experiments to discover unpredicted system behaviors. However, the experiments used are often unmethodical and inefficient.

We realize that computer systems are very similar to simulators, such as aircraft engine simulators, which have established methodologies of experimenting to find performance problems. Essentially, both computer systems and simulators are computer programs that take some inputs and produce some outputs. For example, the inputs of a computer file system can be the position and size of data; one example of the outputs is the quality of data placement. In the example of aircraft engine simulator, the inputs can be the altitude, temperature, and wind speed at which the engine is simulated to run; the output can be the performance of the engine. By systematically studying the inputs and outputs the engine simulator, one can revise the design of the engine to improve the performance.

Because computer systems are very similar to simulators, we can take advantage of systematic methodologies for studying simulators to study computer systems. In our Chopper work, we applied Latin hypercube design (a method in Experimental Design), which is often used to study simulations, to systematically and efficiently explore the vast input space of file system block allocators. In addition, we applied Sensitivity Analysis, also a common method used in simulator studies, to analyze the relationship between the input of block allocators and the quality of data layout. With Experimental Design and Sensitivity Analysis, we were able to identify issues in a popular file system and subsequently fix them. Rigorous and systematic methodologies will help to reduce unexpected issues caused by intuitive but unreliable design decisions.

6.3.2 Analysis of complex systems demands sophisticated tools.

Methodological ideas are more useful when they are built into tools that can be readily used. With well-built tools, one can immediately apply the idea to solve critical problems, instead of implementing the idea.

In this dissertation, we built Chopper, a tool to explore the design issues in file system block allocators. The idea of Latin hypercube design is built into Chopper. Chopper automatically plans and runs the experiments and collects the results for further analysis. Although we only studied XFS and ext4 in this dissertation, Chopper can be used for other file systems, such as Btrfs.

We also built WiscSee, a vertical analysis tool that examines if applications and file systems comply with the unwritten contract of SSDs. With a few lines of configuration scripts, one can run a new application and a new file system in WiscSee. WiscSee will produce well-formatted data, showing the metrics for all of the five rules of SSD unwritten contract. One can then tell how well the combination of the particular application and file system comply with the contract. WiscSee allows us to find many surprising interactions among the layers of the storage stack.

6.3.3 Be aware of and validate assumptions.

When improving existing systems or designing a new system, we must be aware of the assumptions made in the system and validate them in a case-by-case fashion. As components or the running environment of a system change, invalid assumptions may limit the performance of the system. In addition, we may fail to exploit new opportunities that require removing constraints.

In this dissertation, we found many invalid assumptions as systems evolve. For example, we found that the design of Linux page cache assumes that the underlying storage device is an HDD, which is slow and can only process requests serially. With this assumption, Linux page cache serially issues relatively small read requests even when a user requests a large chunk of data. Such an assumption is invalid when the underlying device is an SSD, which is fast and can process requests concurrently. The old design of Linux page cache limits performance on SSDs. Being aware of the assumption will allow one to work around the issue or invalidate the assumption to design new systems.

By re-thinking assumptions, one can find new opportunities to improve systems. For example, we found that many systems rely on using RAM (as cache) because storage devices (e.g., HDDs) are assumed to be slow. As a result, systems are designed to be cache-oriented; cache misses are intolerable as they incur slow I/O operations. However, such an assumption is invalid when the system runs on SSDs, which can offer much higher bandwidth and lower latency than HDDs. Thanks to SSDs, cache misses are much more tolerable. With the new assumption that storage devices are fast, we build a storage-oriented search engine, which employs careful designs in data layout, early pruning, prefetching and space/traffic tradeoffs to exploit the fast storage. The resulting storage-oriented search engine demands much less RAM than the existing engines to achieve the same performance.

6.4 Closing Words

As the size of data keeps growing, efficiently processing data becomes more critical to support decisions in our lives and businesses. A vital component of a data processing system is the storage device, which presents *written* contracts one must follow to use the device correctly. However, storage devices also present *unwritten* performance contracts one must follow to achieve high performance.

In this dissertation, we found violations of the HDD and SSD unwritten contracts and exploit the SSD contract to build a high-performance and cost-effective data processing system. We proposed to use statistical tools to explore violations of HDD unwritten contract. We conducted a full-stack vertical analysis to investigate the violations of the SSD unwritten contract. To exploit the high internal I/O parallelism suggested by the unwritten contract of SSDs, we built a high-performance and cost-effective data processing system (a search engine) that takes advantage of fast and cheap SSDs, while using a small amount of RAM.

This dissertation presents the study, design, and implementation of high-performance storage stacks. We hope that the methodologies we proposed, the observations we found, the tools we built, and the designs we made can help system designers better understand the complex interactions between the software and storage devices.

Bibliography

[1] Amazon aws. https://aws.amazon.com/.

[2] Amazon. https://www.amazon.com/.

[3] Amazon Web Service SSD Instance Store Volumes. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ssd-instance-store.html.

[4] Apache lucene. https://lucene.apache.org/.

[5] Apache solr. lucene.apache.org/solr/.

[6] Apache spark. https://spark.apache.org/.

[7] Bing. http://www.bing.com.

[8] Btrfs Issues. https://btrfs.wiki.kernel.org/index.php/Gotchas.

[9] Cloudlab. http://www.cloudlab.us.

[10] Data age 2025: The evolution of data to life-critical. https://www.seagate.com/files/www-content/our-story/trends/files/seagate-wp-dataage2025-march-2017.pdf.

[11] Db-engines ranking. https://db-engines.com/en/ranking/.

[12] Elasticsearch. https://www.elastic.co/.

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

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

[15] Google. http://www.google.com.

[16] Hdd vs ssd: What does the future for storage hold? https://www.backblaze.com/blog/ssd-vs-hdd-future-of-storage/.

[17] How search organizes information. https://www.google.com/search/howsearchworks/crawling-indexing/.

[18] Hulu. https://www.hulu.com/.

[19] Improving readahead. https://lwn.net/articles/372384/.

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

[21] 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.

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

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

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

[25] Lyft. https://www.lyft.com/.

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

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

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

[29] NASA Archival Storage System. http://www.nas.nasa.gov/hecc/resources/storage_systems.html.

[30] Netflix. https://www.netflix.com/.

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

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

[33] Red Hat Enterprise Linux 7 Press Release. http://www.redhat.com/en/about/press-releases/red-hat-unveils-rhel-7.

[34] Redis. https://redis.io/.

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

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

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

[38] Samsung semiconductor. http://www.samsung.com/semiconductor/.

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

[40] SATA Specification. https://sata-io.org.

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

[42] The state of the octoverse. https://octoverse.github.com/.

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

[44] 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/.

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

[46] Toshiba semiconductor. https://toshiba.semicon-storage.com/ap-en/top.html.

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

[48] Uber. https://www.uber.com/.

[49] Ubuntu. http://www.ubuntu.com.

[50] 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.

[51] Wikibench. http://www.wikibench.eu/.

[52] Wikimedia moving to elasticsearch. .https://blog.wikimedia.org/2014/01/06/wikimedia-moving-to-elasticsearch/.

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

[54] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A Five-Year Study of File-System Metadata. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, San Jose, California, February 2007.

[55] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, pages 57–70, Boston, Massachusetts, June 2008.

[56] David Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Big Sky, Montana, October 2009.

[57] David Andersen, Jason Franklin, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. CMU PDL Technical Report CMU-PDL-08-108, May 2008.

[58] KV Aneesh Kumar, Mingming Cao, Jose R Santos, and Andreas Dilger. Ext4 block and Inode Allocator Improvements. In *Ottawa Linux Symposium (OLS '08)*, volume 1, pages 263–274, Ottawa, Canada, July 2008.

[59] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[60] Nima Asadi and Jimmy Lin. Fast candidate generation for two-phase document ranking: postings list intersection with bloom filters. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 2419–2422. ACM, 2012.

[61] Nima Asadi and Jimmy Lin. Effectiveness/efficiency tradeoffs for candidate generation in multi-stage retrieval architectures. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*, pages 997–1000. ACM, 2013.

[62] Nima Asadi and Jimmy Lin. Fast candidate generation for real-time tweet search with bloom filter chains. *ACM Transactions on Information Systems (TOIS)*, 31(3):13, 2013.

[63] Grant Ayers, Jung Ho Ahn, Christos Kozyrakis, and Parthasarathy Ranganathan. Memory hierarchy for web search. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pages 643–656. IEEE, 2018.

[64] Mary Baker, John Hartman, Martin Kupfer, Ken Shirriff, and John Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 198–212, Pacific Grove, California, October 1991.

[65] Aruna Balasubramanian, Niranjan Balasubramanian, Samuel J Huston, Donald Metzler, and David J Wetherall. Findall: a local search engine for mobile phones. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 277–288. ACM, 2012.

[66] Simona Boboila and Peter 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.

[67] Luc Bouganim, Björn Thór Jónsson, Philippe 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.

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

[69] Rob Carnell. lhs package manual. `http://cran.r-project.org/web/packages/lhs/lhs.pdf`.

[70] Feng Chen, David A. Koufaty, and Xiaodong 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.

[71] Feng Chen, Rubao Lee, and Xiaodong 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.

[72] Peter M. Chen and David 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.

[73] Yue Cheng, Fred Douglis, Philip Shilane, Grant Wallace, Peter Desnoyers, and Kai 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.

[74] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi 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.

[75] Seokhei Cho, Changhyun Park, Youjip Won, Sooyong Kang, Jaehyuk Cha, Sungroh Yoon, and Jongmoo Choi. Design Tradeoffs of SSDs: From Energy ConsumptionâĽ™s Perspective. *ACM Transactions on Storage*, 11(2):8:1–8:24, March 2015.

[76] G. G. Chowdhury. *Introduction to Modern Information Retrieval*. Neal-Schuman, 2003.

[77] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[78] Austin T Clements, Dan RK Ports, and David R Karger. Arpeggio: Metadata searching and content sharing with chord. In *International Workshop on Peer-To-Peer Systems*, pages 58–68. Springer, 2005.

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

[80] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[81] Peter J Denning. The Locality Principle. *Communications of the ACM*, 48(7), July 2005.

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

[83] Cagdas Dirik and Bruce 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.

[84] Nykamp DQ. Mean path length definition. `http://mathinsight.org/network_mean_path_length_definition`.

[85] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference*, page 42. ACM, 2018.

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

[87] Garth Gibson, Gary Grider, Andree Jacobson, and Wyatt Lloyd. PRObE: A Thousand-Node Experimental Cluster for Computer Systems Research. *USENIX ;login:*, 38(3), June 2013.

[88] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*, page 20. 2013.

[89] Aayush Gupta, Youngjae Kim, and Bhuvan 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.

[90] XYHR 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.

[91] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi 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.

[92] Jun He, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The unwritten contract of solid state drives. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 127–144. ACM, 2017.

[93] Jun He, Duy Nguyen, Andrea C. Arpaci-Dusseau, and Remzi 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.

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

[95] Jon C. Helton and Freddie J. Davis. Latin Hypercube Sampling And The Propagation Of Uncertainty In Analyses Of Complex Systems. *Reliability Engineering & System Safety*, 81(1):23–69, 2003.

[96] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping 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.

[97] Ronald L. Iman, Jon C. Helton, and James E. Campbell. An Approach to Sensitivity Analysis of Computer Models. Part I - Introduction, Input, Variable Selection and Preliminary Variable Assessment. *Journal of Quality Technology*, 13:174–183, 1981.

[98] Song Jiang, Lei Zhang, XinHao Yuan, Hao Hu, and Yu 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.

[99] Xavier Jimenez, David Novo, and Paolo 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.

[100] Keren Jin and Ethan L. Miller. The Effectiveness of Deduplication on Virtual Machine Disk Images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 7:1–7:12, 2009.

[101] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, et al. Bluedbm: An appliance for big data analytics. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pages 1–13. IEEE, 2015.

[102] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, et al. Grafboost: Using accelerated flash storage for external graph an-

alytics. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 411–424. IEEE, 2018.

[103] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun 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.

[104] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon 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.

[105] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. Finding Latent Performance Bugs in Systems Implementations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 17–26, New York, NY, USA, 2010. ACM.

[106] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting Storage for Smartphones. *ACM Transactions on Storage*, 8(4):14, 2012.

[107] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan 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.

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

[109] 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.

[110] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun 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.

[111] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A Log Buffer-based Flash Translation Layer Using Fully-associative Sector Translation. *ACM Trans. Embed. Comput. Syst.*, 6(3), July 2007.

[112] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. LAST: Locality-aware Sector Translation for NAND Flash Memory-based Storage Systems. *Operating Systems Review*, 42(6):36–42, October 2008.

[113] Jinyang Li, Boon Thau Loo, Joseph M Hellerstein, M Frans Kaashoek, David R Karger, and Robert Morris. On the feasibility of peer-to-peer web indexing and search. In *International Workshop on Peer-to-Peer Systems*, pages 207–215. Springer, 2003.

[114] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 1–13. ACM, 2011.

[115] Dongzhe Ma, Jianhua Feng, and Guoliang Li. A Survey of Address Translation Technologies for Flash Memories. *ACM Comput. Surv.*, 46(3):36:1–36:39, January 2014.

[116] Ariel 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.

[117] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Alex Tomas Andreas Dilge and, and Laurent Vivier. The New Ext4 filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, July 2007.

[118] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, Laurent Vivier, and Bull S.A.S. The New Ext4 Filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, July 2007.

[119] Michael D. McKay, Richard J. Beckman, and William J. Conover. Comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, 1979.

[120] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur 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.

[121] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik 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.

[122] Vidyabhushan Mohan, Taniya Siddiqua, Sudhanva Gurumurthi, and Mircea 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.

[123] James K. Mullin. Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering*, (5):558–560, 1990.

[124] V. N. Nair, D. A. James, W. K. Ehrlich, and J. Zevallos. A Statistical Assessment of Some Software Testing Strategies and Application of Experimental Design Techniques. *Statistica Sinica*, 8(1):165–184, 1998.

[125] Peter Norvig. Teach Yourself Programming in Ten Years. `http://norvig.com/21-days.html`.

[126] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[127] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng 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.

[128] Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh, Wonhee Cho, and Jin-Soo Kim. A Reconfigurable FTL (Flash Translation Layer) Architecture for NAND Flash-based Applications. *ACM Trans. Embed. Comput. Syst.*, 7(4):38:1–38:23, August 2008.

[129] Dongchul Park, Biplob Debnath, and David 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.

[130] Dongchul Park and David HC 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.

[131] Ryan Paul. Google upgrading to Ext4. `arstechnica.com/information-technology/2010/01/google-upgrading-to-ext4-hires-former-linux-foundation-cto/`.

[132] Zachary N. J. Peterson. Data Placement for Copy-on-write Using Virtual Contiguity. Master's thesis, U.C. Santa Cruz, 2002.

[133] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi 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.

[134] Milo Polte, Jiri Simsa, and Garth 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.

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

[136] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A Comparison of File System Workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 41–54, San Diego, California, June 2000.

[137] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[138] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. 27(3):17–28, March 1994.

[139] Jerome Sacks, William J Welch, Toby J Mitchell, and Henry P Wynn. Design and analysis of computer experiments. *Statistical science*, pages 409–423, 1989.

[140] Andrea Saltelli, Marco Ratto, Terry Andres, Francesca Campolongo, Jessica Cariboni, Debora Gatelli, Michaela Saisana, and Stefano Tarantola. *Global sensitivity analysis: the primer*. John Wiley & Sons, 2008.

[141] Andrea Saltelli, Stefano Tarantola, Francesca Campolongo, and Marco Ratto. *Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models*. John Wiley & Sons, 2004.

[142] Thomas J Santner, Brian J Williams, and William Notz. *The design and analysis of computer experiments*. Springer, 2003.

[143] Steven W. Schlosser and Gregory 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.

[144] Bianca Schroeder, Raghav Lagisetty, and Arif 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.

[145] SGI. XFS Filesystem Structure. `http://oss.sgi.com/projects/xfs/papers/xfs_filesystem_structure.pdf`.

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

[147] Ji-Yong Shin, Zeng-Lin Xia, Ning-Yi Xu, Rui Gao, Xiong-Fei Cai, Seungryoul Maeng, and Feng-Hsiung 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.

[148] Keith A. Smith and Margo I. Seltzer. File System Aging – Increasing the Relevance of File System Benchmarks. In *Proceedings of the 1997 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/PERFORMANCE '97)*, Seattle, Washington, June 1997.

[149] IM Soboĺ. Quasi-Monte Carlo methods. *Progress in Nuclear Energy*, 24(1):55–61, 1990.

[150] Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[151] Diane Tang and Margo Seltzer. Lies, damned lies, and file system benchmarks. *VINO: The 1994 Fall Harvest*, 1994.

[152] Vasily Tarasov, Saumitra Bhanage, Erez Zadok, and Margo Seltzer. Benchmarking File System Benchmarking: It *IS* Rocket Science. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[153] Nicola Tonellotto, Craig Macdonald, and Iadh Ounis. Efficient and effective retrieval using selective pruning. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 63–72. ACM, 2013.

[154] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P. Wright. A Nine Year Study of File System and Storage Benchmarking. *ACM Transactions on Storage (TOS)*, 4(2), May 2008.

[155] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan 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.

[156] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, and Andrew 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.

[157] CF Jeff Wu and Michael S Hamada. *Experiments: planning, analysis, and optimization*, volume 552. John Wiley & Sons, 2011.

[158] Gala Yadgar and Moshe 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.

[159] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.

[160] 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.

[161] Jiacheng Zhang, Jiwu Shu, and Youyou 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.

[162] You Zhou, Fei Wu, Ping Huang, Xubin He, Changsheng Xie, and Jian 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.