UNIVERSITY OF CALIFORNIA
SANTA CRUZ

IMPROVING PERFORMANCE FOR FLASH-BASED STORAGE SYSTEMS

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

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Jingpei Yang

June 2014

The Dissertation of Jingpei Yang is approved:

Table of Contents

List of Figures

List of Tables

Abstract

Improving Performance for Flash-based Storage Systems

by

Jingpei Yang

With the dramatic advances in electronic device industry, the availability of high speed non-volatile memory (NVRAM) has introduced a new tier into the storage hierarchy, and holds great promise for reduction in latency, power consumption, and improved performance. Among them, flash-memory has become a popular storage medium to replace hard disk as a permanent storage device and DRAM as a temporary storage device. Yet, despite their fast random I/O performance, the design of a flash-based storage system achieves suboptimal performance or suffers from reduced endurance due to the nature of flash memory, for example, out of place update, asymmetric read-write throughput, and limited write cycles. Lack of application aware design makes flash memory less efficient, and hence cannot meet various performance requirements. To this end, we investigate the roles flash memory plays in different storage applications and their performance and reliability requirements. By examining the behavior of these systems and their consequent data access characteristics, as well as the performance impact, we propose solutions that tradeoff performance, cost, endurance and reliability to achieve high efficiency for flash memory in different storage applications with reduced overhead.

We first explore the use of flash memory as a write-through cache in a

tiered storage. We demonstrate the individual and cumulative contributions of cache admission policy, cache eviction policy, flash garbage collection policy, and flash device configuration on a flash caching device. We show that workloads on Solid State Caches (SSCs) have significantly greater write pressures than their storage counterparts. we propose HEC, a High Endurance Cache that aims to improve overall device endurance via reduced media writes and erases while increasing or maintaining cache hit rate performance.

To further characterize the behavior of flash memory used in different storage applications, we focus on flash as a primary backing storage device in the second part of this dissertation. We explore the drawbacks of a typical log-structured file system on top of a log-structured FTL flash device. We characterize the interactions between multiple levels of independent logs, and describe several practical scenarios which arises in real log-on-log systems. We then propose a log-aware coordination to tune the layout of logs, so that when multiple layers of logs exist in the system we can still achieve high performance with minimum interference among each log.

In the third part of this dissertation, we explore the approaches to collapsing logs. While there are several popular ways that utilize the nature of flash memory translation layer to eliminate multiple layers of logs in the entire system, we focus on the benefit we can obtain from a log-less object-based flash aware system. We show from simulation experiments that advanced features could be embedded to improve overall performance for object-based flash system with low overhead through a rich interface.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor, Prof. Scott A. Brandt for his great and continuous support of my Ph.D. journey. I would like to thank his help, kindness, and understanding during the most difficult time I had in my Ph.D. His consistent encouragement and support made this journey complete. It is my honor to have him as my Ph.D. advisor and learn from him.

I would like to thank my manager at Fusion-IO, Dr. Nisha Talagala, who provided guidance and persistent help on my research. Without the opportunities she gave me and her help and thoughtfulness, I could not be able to go through those tough times. She is one of the sharpest people I have ever met and keeps bringing new ideas. I learned a lot from her enthusiasm and diligence.

I would also like to thank my committee members, Prof. Carlos Maltzahn and Prof. Jose Renau. Their time, dedication and insightful comments helped me to improve my thoughts. In addition, I would like to thank Prof. Yiming Hu who introduced me to the world of computer systems and architecture and encouraged me to continue my Ph.D. study.

My gratitude also goes to my colleagues in the Advanced Development Group at Fusion-IO and Prof. Darrell Long and Prof. Ethan Miller in the Storage Systems Research Center at UCSC. I am lucky to have them during my Ph.D. journey and work with them.

More importantly, the success of my Ph.D. depends largely on the support from Mary Fan and her family. My life in the United States could not go evenly without their help. Through them, I see God's hand and provisions of joys, challenges, and grace.

Last but not least, I would like to express my deepest gratitude to my family. The tremendous support from my parents is beyond description. Being far away from them for the past many years, their love is always with me. Thanks for the good education they gave me. They deserve it.

Chapter 1

Introduction

1.1 Flash memory in storage systems

Flash based devices are increasing in popularity for performance sensitive applications ranging from databases to key-value stores to persistent messaging, to name a few. Table 1.1 briefly depicts the characteristics of NAND flash and other traditional storage media. With flash memory's unique advantages, it shows great potential to improve performance for a wide range of storage applications. As a result, the way we designed flash-based storage systems years ago can no longer meet the needs of the fast development trend and increasing requirements on performance and reliability any more. On the other hand, a lot of designs only focus on improving performance on a single layer without considering the limitations of the other layers. For example, the space reclaiming process in a log-structured file system may be detrimental to the garbage collection efficiency in the flash device.

Figure 1.1 depicts an overview of today's storage hierarchy and shows

	DRAM	NAND Flash	Disk
Cell Size	6-8 F^2	4-5 F^2	(2/3) F^2
Read Latency	10-60 ns	25 μs	8.5 ms
Write Latency	10-60 ns	200 μs	9.5 ms
Energy per bit access	2 pJ	10 nJ	100-1000 mJ
Static Power	Yes	No	Yes
Endurance	$> 10^{15}$	10^4	$> 10^{15}$
Non-volatility	No	Yes	Yes

Table 1.1: Comparison of current memory/storage technologies. [73]

different roles of flash memory devices are playing in the system. As flash memory can deliver much better I/O performance than disk-based systems, it has been used to store some frequently access data (e.g. metadata) and used in combination with HDD to diminish the I/O latency through the block interface, or even replace the HDD as a primary storage device. As shown in Figure 1.1, some systems use flash array as a Network Attached Storage devices(NAS) [76, 26], some use it as a traditional SSD backing device [83, 82]. In addition, its non-volatile characteristic and relatively lower cost per GB makes flash memory a good candidate to extend the size of main memory using DRAM in a tired storage. Using flash as a cache enables applications with good workload locality to leverage the performance of flash while utilizing less expensive traditional disk-based storage as the backing store. A Solid-State-Cache (SSC) could be used to serve various types of backing storage devices, there are a few commercially avaliable flash based caching products to better serve large scale enterprise storage systems and provide improved QoS [1, 87, 6].

While flash device has been used in a wide range of applications, the design of a flash-based storage system still achieves suboptimal performance

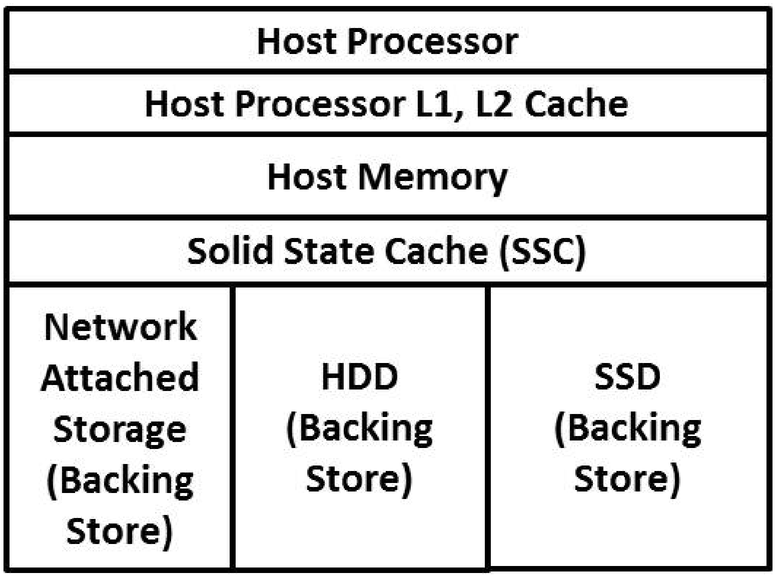


Figure 1.1: Storage system architecture

due to the nature of flash memory such as out-of-place update and limited Program/Erase cycles (P/E cycles). Minimizing unnecessary writes to flash is generally of high importance. Though a lot of research has been done to address these issues and optimize the benefit flash memory can deliver to the system, most of the designs only focus on improving performance on one layer, while running the risk of increasing overhead and defeating its benefit on the other layer. This is due to unawareness of different layers and the lack of coordinations between different system components. For example, some SSC designs minimize the garbage collection overhead by exclusively discarding data in the victim segments, but not considering the loss of valid pages may potentially results in more writes to the flash cache due to a cache miss. Some log-structured flash-aware file system separating data to different file system logs to get better data arrangement, but not be aware of the limitations of the underlying hardware that can only support fewer number of append points.

These issues lead to one of the problems our dissertation address: What algorithms and design choices are better for different flash memory applications to tradeoff performance and cost? By investigating the challenges of reducing cost while maintaining high performance and reliability issues that a flash-based storage system meets, we explore algorithms for garbage collector, data placement strategies, application-aware controllers in different flash-based storage devices. For instance, a cache device is in high demand of fast random I/O access, this forces garbage collector to reduce overhead on both selection and reclaiming processes so that it will not block the incoming requests and waste CPU cycles. Meanwhile, as write-through cache allows eviction of valid data, cache hit rate shall be maintained to a certain level. Thus, the garbage collector in device layer should be aware of such performance requirements that is required by the upper layer.

1.2 Contributions

The dissertation is a comprehensive study of flash memory used in the storage systems. We investigate the flash memory device behavior following the data path in the storage hierarchy, from caching for temporary data storage, to the store of a large volume of permanent data. Different from other flash-aware system designs that only focused on optimizing the performance of a single layer without being aware of the limitations of the other layers, for example, multiple file system logs on top of a single append point flash chip, or GC eviction policy for flash-based cache without considering the hit rate QoS, our study explores coordinations between different layers. We demonstrate the individual and cumulative effects of different storage layers on the overall systerms' performance and endurance.

While there are many new classes of materials and technologies becoming interesting in the family of NVRAM, this dissertation focuses on solving the problems in flash-based storage system. The use of flash memory in this dissertation focuses on caching, primary backing storage, and a combination of them. Flash memory used in other storage systems is simplified in our work. Besides, algorithms proposed in this dissertation that fit into flash-based system could also be applied to issues for other NVRAM candidates in the future as long as log structure is used and out-of-place update happens in those devices. The technical contributions of this dissertation include the following:

- We present the unique write pressures seen by a flash-based cache device that result in cache and flash layer write amplification, and their combined effects on hit rate and media endurance. We present methods to reduce the write amplification factors by using a combination of admission, eviction, and GC optimizations. Our combined algorithms help reduce cache writes significantly, with improved or maintained hit rate.

- We characterize the interactions between multiple levels of independent logs in a log-structured system using flash memory as a storage media, and describe several practical scenarios which arises in real systems. We show that log on log can result in highly counterintuitive behaviors. Furthermore, we recommend several design choices for log coordinations to preserve the advantages of log-structure while trailing off the disadvantages that one level of log passes to the others.

- While log coordination helps improve overall performance in a system having multiple layers of logs, collapsing logs is always beneficial as it reduces the system overhead with improved efficiency. We also explore the possibilities of utilizing the nature of flash translation layer for a log-less flash aware system. Object-based flash system is investigated in this dissertation. We explore its feasibility and the overhead of some advanced features, for example, object data grouping, and discussed its use case.

1.3 Outlines

The rest of the dissertation is organized as following: Chapter 2 discusses flash-based storage systems and the state-of-art related work. This includes introduction of current designs and problems with flash memory used as a traditional primary backing storage, as well as it being used as a caching device in a tiered storage system that resides in between DRAM main memory and HDD. In chapter 3, we present the design of a High Endurance Flash-based Cache (HEC) and the tradeoff between flash endurance and cache performance. We evaluate the proposed algorithms through an SSC simulator and discussed future directions to extend our work. Chapter 4 and chapter 5 are related to flash memory used as a primary backing storage device. Chapter 4 discussed the problem of existing log-structured file systems on top of a flash memory device. We show that log on log can result in highly counterintuitive behaviors. In addition, we recommend several design choices and solutions to improve overall performance and minimize overhead for a log-aware system while log-on-log cannot be eliminated. In chapter 5, we discussed the system design choices of collapsing logs. It utilizes the nature of a log-like flash translation layer (FTL) to reduce total number of logs in the entire system. Chapter 6 summarizes contributions of this dissertation.

Parts of the dissertation have been published as a paper in conference. Chapter 3 is based on a paper published in the 7th ACM International Systems and Storage Conference (SYSTOR) in 2013 [96]. Chapter 3 and chapter 4 are part of the projects I have done at Fusion-io. The ideas are described in United States Patent Application entitled SYSTEMS AND METHODS FOR LOG COORDINATION, and SYSTEMS AND METHODS FOR A HIGH ENDURANCE CACHE.

Chapter 2

Background and Related Work

Flash memory has attracted the attention of modern storage systems. In this chapter, we discuss flash-based storage systems and the state-of-art related work. We first introduce the fundamental of NAND flash briefly, followed by the introduction of current designs and problems with flash memory used as a primary backing storage, as well as it be used as a caching device in a tiered storage system.

2.1 NAND flash background

NAND flash memory has unique characteristics particularly with respect to writes. In absence of mechanical parts, it has less possibility of page failure, but higher chance of raw bit error as the density increases. Many studies have been done to investigate the failure behavior of SLC and MLC [31, 89, 7]. The Raw Bit Error Rate (RBER) varies over time which is affected by I/O operations, temperature, and etc. Moreover, RBER increases as the the number of

P/E cycles increases. As the capacity and density of flash chips increases, metrics such as reliability, endurance, and performance are declining [32]. As a result, longer and more complex ECCs are employed [61, 20, 45]. This brings challenges to the future of a performance aware flash memory based storage system. While high random I/O performance make them attractive in many applications, the increasing cost of maintaining reliability and endurance make it difficult to be adopted in large scale applications.

As the media must be explicitly erased between successive updates to the same location, with the erase unit being substantially larger than the update unit. NAND flash also has limited endurance, measured in P/E Cycles. For example, some 25nm multi-level cell (MLC) NAND flash devices have a rating of 3K P/E cycles per NAND flash erase block [63].

2.2 Garbage Collection, Write Amplification, and Wear Leveling

To amortize the erase cycles over as many writes as possible, FTLs perform out-of-place updates. New writes are written to empty blocks while a background garbage collector coalesces free space and performs erases. Flash P/E cycles are consumed by both user-data writes and the extra writes performed by GC. The amplification of user writes by the system is referred to as write amplification [2] or Flash-Layer Write Amplification (FLWA). The combination of user writes and FLWA result in the total Physical Bytes Written (PBW), which is the primary metric of endurance and reliability in flash devices. FLWA can be affected by user workloads and GC policies (including selection of erase blocks). Significant research has been performed on GC strategies, primarily focusing on storage usages of flash, with some focus on cache usages.

Studies have focused on analysis of write amplification and performance effects in SSDs due to various GC strategies [36, 71, 15]. In these studies, endurance impacts are assessed through measurement of write amplification or write workload effects from various victim selection strategies. Other studies have considered the effects of over-provisioning and victim selection [36, 71]. Other studies have focused on the impacts of wear leveling and its impact on SSD endurance [66, 18, 8]. Recently, some research introduced a GC selection algorithm designed for caching [85, 33]. Some other research focus on separating data of different update frequencies and append them to different points in the log during the initial writes, thus providing a better segment selection strategy for GC [64, 37, 48, 67]. While reducing GC overhead for flash-based devices used in different applications has been previously discussed [55, 51], there has been a lack of research on coordinating GC with different cache components.

In addition, some research tries to pass the file system semantic to the underlying flash memory device without much change on the block interface, thus reducing the extra amount of data copied forward by GC due to lack of knowledge from file system delete operations (TRIM) [10, 84, 42, 97, 92]. Others proposed a new design of data structure for indexing in flash-based file systems [79, 27, 43, 93] to reduce the write operations for metadata update. On the other hand, some research takes advantage of GC process to reduce file system level fragmentation by combining data with adjacent logical block addresses, thus reducing the data structure entries and write operations [21].

Recently, some researches introduced a garbage collector selection algorithm that is designed for caching [86, 34]. From a cache point of view [14], write-through policy allows eviction of valid packets silently but has high demand of cache read hit rate, while write-back policy needs to handle asynchronous update of cache and backing store. The asynchronous update to the backing store introduces many challenges to garbage collection policies, admission policies, and performance of regular operations. It can however enable more write intensive workloads to become cache candidates, making it a good target for endurance focused investigations. Thus, selecting a better segment to clean is not only a problem of finding one with fewer number of valid packets to copy forward, but also related to some other issues, for example, the hit rate of a flash-based write-through cache will be affected by the garbage collector's eviction policy.

2.3 Flash-based Cache

In all caches, hit rate is a primary performance and effectiveness metric. In DRAM based caches, since the only cost of a cache insert is the content displaced, focus has traditionally been on the eviction policy. Since SSCs generate writes upon admission of new data, a cache insert should be evaluated in terms of not just what it replaces but also the media writes. In this dissertation, we refer to these increased writes as Cache-Layer Write Amplification (CLWA), which is computed as the ratio between the cache-generated writes and the original workload writes. CLWA can result from cache misses on workload reads and from both hits and misses on workload writes. Consequently, SSCs require additional admission policy consideration specifically to help segregate or reject unnecessary disk traffic (e.g., backups) from polluting the cache and to admit only "quality" (or popular) data into cache. The size of the cache, the extant cached data, and the admission policy can therefore impact the writes generated by the cache to the underlying flash device.

Figure 2.1 shows the components of an SSC that serves varied backing storage devices. An SSC device includes several components: cache controller, flash controller, as well as banks of NAND flash. The cache controller may include different admission and eviction policies to control the data flow to the cache device. The choice of GC algorithms in the flash controller will determine how incoming writes translate into physical writes, and hence the FLWA. The combination of CLWA and FLWA will determine the PBW and the endurance of the device for the cache workload, consequently we focus on writes and erases as the flash endurance metrics.

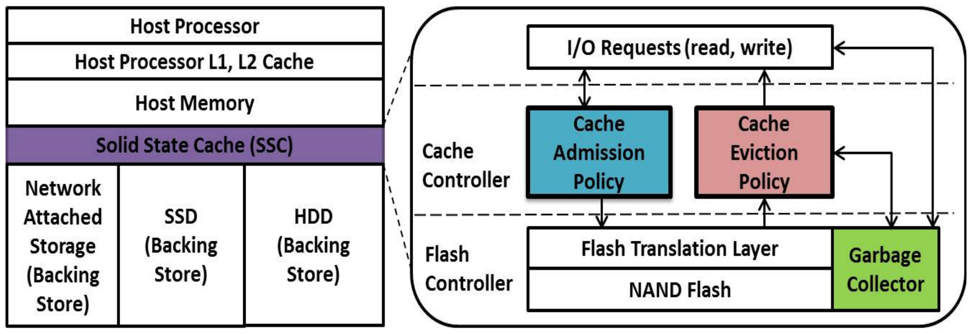


Figure 2.1: Components of an SSC.

Given that an SSC can cache various kinds of back-end storage systems, the performance of an application using an SSC is dependent on many factors: the hit rate, the performance of the SSC, the performance of the backing store, I/O patterns and bottlenecks in the application, etc. Since our goal in this paper is to improve device endurance while maximizing performance, we focus on hit rate as the performance metric, and writes and erases as the endurance metrics. Comparing hit rate ensures that we can assess the efficiency of the cache independent of the backing store and other factors. Since in most cases SSCs are deployed in write-through mode to preserve the reliability of data, we focus this study on write-through caching.

Limited studies [85, 71, 49] exist that have focused specifically on the design of a high performance SSC device with file system or firmware support. FlashTier [85] describes a system architecture that guarantees data consistency and supports sparse address space for SSCs. OP-FCL [71] trades off over-provisioned space with SSC overall performance. Other research focuses on integrating flash into disk-based systems to improve system performance [78, 16, 19]. Narayanan *et al.* [69] quantifies the benefit of using SSCs for enterprise workloads. Little is known about the combined effects of workloads, caching admission and eviction policies, and FTL enhancements on SSC write workloads. In our work, we evaluated additional SSC capabilities and characteristics and the resulting WA (CLWA, FLWA, TCWA) and cache hit rate impacts on SSCs on various traces. Additionally, we consider the impact of reserving capacity of physical usage for hit rate and the combined system effects on the SSC write workload and hit rate.

Other research outlines the design tradeoffs for SSD performance and utilizes workload trace-driven simulators on traces obtained from real operational workloads [8, 52]. Those simulation tools provide instruments to analyze SSDs as a primary backing store. Our study utilizes simulators that are trace-driven, however we have integrated cache-specific policies for admission and eviction together with FTLs that focus on log-structured devices often used in SSCs. Such integration provides us with a framework for investigating the combined affects of key SSC components and their impacts on hit rate versus endurance.

Many studies have been performed for admission control and replacement strategies of a cache device [12, 75, 95, 62]. Different from cache devices that support in-place-updates and unlimited write cycles, admission control for SSCs is even more critical to both performance and endurance. Studies have also focused on detecting sequentiality in workloads in SSCs [71]. Using techniques such as sequentiality detection and looping request detection [50] does impact the workload as seen by the SSC. In addition, techniques for identifying recency in cached data also impacts the workload seen by an SSC [41, 40]. Parallel to these research, our work provides an extendable framework to integrate these algorithms into an SSC device, and helps inform understanding of the impact of different algorithms on a solid-state NAND device.

2.4 Log-structured system

As flash based devices are increasing in popularity for performance sensitive applications ranging from databases to key-value stores to persistent messaging, to name a few. In many or most of these cases, applications started by using flash as a fast disk and then made optimizations within the application itself to better leverage the flash. Since flash devices are known for asymmetric write performance, a common design pattern for applications is to use a log structure to optimize for flash - twitter fatcache [4], NILFS [53], SILT [59], FlashStore [23], and so forth.

Applications have moved to a log structure to remove dependencies on the underlying Flash Translation Layer of the flash device. Since flash cannot be updated in place, every incoming write is directed to a fresh location in flash. As the device runs out of physical space, a garbage collection process compacts valid content and frees up erase blocks for incoming writes. The performance of the flash device is as such heavily dependent on the garbage collection intelligence and efficiency as well as the incoming workload. Since early flash devices, and still some low end flash devices, demonstrate poor performance for random writes, applications have evolved to contain their own dynamic mapping scheme in an effort to limit I/O to the flash device to its ideal workload, large sequential writes.

There is a significant history of log structured stores of various kinds ranging from storage systems [22] to file systems [80] to databases [91] and other applications. Some stores are strictly log structured in that no update in place mechanisms are allowed, while other stores are more write-anywhere in nature [35]. All log structured stores allow new writes to be directed to free space in the device, and all contain some form of garbage collection (frequently called cleaning), which allows invalid space to be compacted and reused. While some strict log schemes force all new writes to the head of the log, others allow hole-plugging, enabling some invalid space to be reclaimed directly without compaction.

Prior to the arrival of flash, the motivation for log structured stores was to acclerate write performance while allowing random reads to be serviced from DRAM cache. Log structured stores allow additional advantages, such as enabling snapshots, transactional updates and eliminating the small write performance problem when run above RAID 5 layouts [35, 65].

Flash inserts a new dimension since flash is by nature not update in place at a physical level. Flash can only be erased in the units of erases blocks, which are typically an order of magnitude of more larger than the unit of write (e.g. 64 or 128 write pages per erase block). For easier management, many flash devices group multiple erase blocks into larger Virtual Erase Blocks, which are treated conceptually similarly to cleaner segments in a log structured file system. Since flash has a limited number of program/erase cycles, flash garbage collection has to balance both the efficiency of cleaning and the reliabilty requirements of balancing the amount of writes to each virtual erase block. Flash has additional requirements, such as read disturb handling, which require erases and rewrites to maintain data integrity. As such, while some factors that drive flash garbage collection are similar to those driving cleaning at higher level log stores, others are media specific.

Both flash based logs and higher level logs maintain the notion of append points, which are sequential streams within the architecture. While some architectures are strictly single append point, implying that all writes, incoming, cleaning, metadata, are driven to the same append point, others separate these. F2FS [39], for example, has six logical append points. Similarly, the FTL within a flash device may have one or more append points depending on the design. Similarly, applications that are log structires may have one or more append points (twitter fatcache has one, SILT has several).

This technique however has implications that are not fully understood. The extra operations performed by the flash layer for garbage collection result in additional write operations for every incoming write. The ratio of physical writes to incoming writes is Write Amplification, which impacts performance as well as endurance. When applications above the FTL perform their own log structured writing, an additional garbage collection process needs to occur at the application level, which some have termed Auxilliary Write Amplification [59], which is a multiplicative effect over Write Amplification. The intent of the application level log is to reduce the device WA to as close to 1 as possible, however the ability to realize this is sometimes very dependent on configuration and interactions between the device FTL and the application. Recent research also uses log-structure in a DRAM-based system [81], though coordinations between logs and garbage collectors are explored, the lack of information of the underlying hardware layouts will still be detrimental to the overall performance. Same for other log-structured system that uses emerging NVRAM devices as a storage media.

While user space applications have adopted the log structured model, file systems within the kernel have done the same [53, 39]. As a result, it is possible that two or more log structured workloads may become stacked on a flash device, with the application writing in log structured fashion over a file system writing in log structured fashion over an FTL writing in log structured fashion to the physical flash.

Comparing with other log-structured systems, this dissertation focuses on Log on Log, outlining and characterizing the interactions between multiple levels of independent logs. We describe several practical scenarios which arise in real systems and show that the log on log strategy can provide benefits in some situations but can result in highly counterintuitive behaviors in other situations for flash-based storage systems. We show results from a log on log scenario and compare it to an alternate architecture where the application operates in coordination with the FTL to leverage its log rather than build an independent log, and discuss the tradeoffs therein.

2.5 Approaches to access flash memory

Many researches have been done to explore the use of flash memory and the best way to adopt them into the storage hierarchy. Currently, these approaches can be categorized into three classes. Direct access model with a flash-aware file system design on the host side, FTL-based model that embeds the flash translation layer into the underlying device, and breaking the traditional block interface with a more flexible object-based interface using a quite different new design on both file system and the device part.

The direct access model supports direct access to a raw flash device by a specific designed file system [9, 38, 46, 39]. Such system can usually achieve performance efficiency by considering the characteristics of flash memory, such as out-of-place update and wear-leveling. In addition, address translation and garbage collectoin is done at the host side, giving the system more flexibility and resource to maximize performance by balancing the workload from different components. However, once designed, file system could not be optimized for a specific type of hardware, and scalability is another problem.

The FTL-based model is a more popular approach in the current flash world. With FTL embedded in the underlying device - flash-based Solid State Disk, it allows a traditional disk-based file system to have flash memory as a primary storage device without any change on the host side. This provides high compatibility that file system designers do not have to worry about different flash characteristics, while the SSD vendors design their own FTL algorithms for the specific hardware. Many researches have been done to improve the performance for FTL [56, 77, 44, 57]. However, the improvement is limited as this approach introduces higher overhead with two duplicate layers mapping, one in the file system and one in the FTL layer, especially when log-structured file system is used on top of an SSD, it will be more detrimental to the overall performance

(see detailed discussion in Section 4)

The third approach - object-based model is previously developed in distributed disk-based storage systems [17, 29, 94]. It offloads the traditional file system's storage management layer to the underlying device - Object Storage Device (OSD), while the file system becomes lightweighted by keeping only the name resolution layer on the host side. Through object-based interface that standardized in the ANSI T10 [68], for example, read/write/create object(), data blocks belonging to each object are now closely associated with its metadata and maintained by the OSD. With this approach, file system is unaware of the hardware characteristics and no duplicated mapping is necessary. This allows a generic file system to be able to manage the host side regardless of the type of the underlying device, while having more flexibility and resources of advanced features on the device side, especially in large scale distributed systems that no central controller is available. The benefit of object-based storage in a disk-based system is not remarkable as file system does not need any complex mapping mechanisms to map logical addresses to physical space on disk, it shows promising benefit on the Storage Class Memory devices, large scale distributed storage systems [24, 54, 90]. Recent research has investigated its use in the emerging technology for Storage Class Memories (SCM) as well as flash memory [48, 47, 60, 58]. Other efforts are done to explore more efficient interface for flash memory [72, 42, 88].

Chapter 3

Flash as a Cache in Tiered Storage

3.1 Flash-based Caching

In recent years, deployment of flash memory as a cache has rapidly accelerated. Flash memory's fast random I/O performance and non-volatile characteristic make it a good candidate as a Solid State Cache device (SSC). Using flash as a cache enables applications with good workload locality to leverage the performance of flash while utilizing less expensive traditional disk-based storage as the backing store.

Performance of cache is measured through read hit rate. Where, hit rate is expressed as a percentage of number of reads serviced from the cache compared to all read accesses. Cache admission and eviction policies are added to the cache to admit quality data and evict less critical data, respectively. These policies help improve read hit rate and hence the effectiveness of the cache.

While improving hit rate is critical for overall cache performance, SSCs experience unique pressures that are not present in traditional DRAM caches.

Due to the characteristics of flash memory, such as limited Program/Erase cycles (P/E Cycles) and out-of-place updates with subsequent Garbage Collection (GC), minimizing unnecessary writes to flash is generally of high importance.

SSCs are more write intensive than their storage counterparts. From a SSC point of view, writes to SSC are comprised of: user writes, writes caused by admissions of cache misses (of new or previously evicted data), and writes due to GC. Thus, the cumulative choices of cache admission policy, cache eviction policy, GC strategy, and device configuration (e.g. system reserve for over-provisioning) impact the overall SSC performance and endurance precisely because they arbitrate writes to flash.

In this work, High Endurance Cache (HEC), we develop and validate techniques for improving SSC performance and endurance by providing insights and answers to the following questions:

- How are the workload characteristics that a flash-based cache is likely to encounter different from the workloads encountered by flash when used as a storage device?

- How do different cache admission and eviction policies affect read cache hit rate and device endurance?

- How do various GC strategies in the Flash Translation Layer (FTL) impact writes/erases to media for cache workloads?

- What is the impact of the combination of cache admission and eviction, GC policies and device configuration on hit rate and writes/erases to the

SSC?

To our best knowledge, this is the first study of a flash-based cache device that focuses on improving performance and endurance through both cache and flash layer optimizations. The contributions of our HEC work follow:

- We present the unique write pressures seen by an SSC that result in Cache-Layer Write Amplification (CLWA) and Flash-Layer Write Amplification (FLWA), and their combined effects on hit rate and media writes and erases.
- At the cache layer, we present several cache admission policies and demonstrate that CLWA can be significantly reduced (up to 8x). At the flash layer, we present several GC policies and demonstrate that FLWA can also be significantly reduced (up to 6.17x).
- We show that our combined admission, eviction, and GC optimizations can reduce writes by up to 20x, reduce erases by up to 6x, with improved or maintained hit rate.
- Finally, we show the implications of QoS with virtual storage containers in cache for both performance and endurance of SSC.

3.2 Problem Analysis

We highlight the effects of CLWA and FLWA via a simple example. We use a polluted TPC-E trace, which contains traffic from a database TPC-E run combined with a backup workload. We execute the TPC-E Polluted trace (tpcep) and gather the hit rates, read/write counts and flash management statistics from our simulation framework (described in Section 3.3). We configure the cache to admit all misses. In this example, the cache and the flash management layers operate independently, with the cache performing LRW-based eviction (Least Recently Written) and informing the FTL to invalidate evicted blocks one-by-one as needed (see Sections 3.3.2.1 and 3.5.1), and the FTL performing an *OLDEST* garbage selection policy, targeting the least recently written segments for copying forward the valid data. The flash device is configured with 20% reserve for use by the FTL GC. The cache is warmed with the workload prior to data collection, processing the entire trace prior to executing the workload a second time during which the data are gathered. Measurements are summarized after the warming stage and are shown in Table 3.1.

Original		Cache	Cache	GC	Total	CLWA	FLWA	TCWA	Hit rate
r GB	w GB	size GB	w GB	w GB	w GB				%
331.90	36.8	80	322.13	1553.98	1876.11	8.75	5.82	50.93	14.03
		100	300.11	1459.13	1759.24	8.16	5.86	47.82	20.67
		120	275.83	1352.01	1627.84	7.50	5.90	44.25	27.98

Table 3.1: TPC-E Polluted: Combined write amplification under ADMIT ALL and OLDEST *cache-based* eviction mode, the workload is more write intensive than the original one due to extra writes from cache misses and from GC reclaiming process.

CLWA: From Table 3.1, we see that the workload presented to the flash device through the cache (Cache Writes) is far more write intensive than the original workload (Original Writes). The original workload's writes as well as cache misses become writes to the flash device. As cache size increases, the relative Cache Writes to Original Writes ratio decreases. However, the increase of hit rate is suboptimal due to low quality writes. The CLWA can be as high as 7-8x.

FLWA: Flash performs out-of-place updates, and writes are directed to new locations with invalidation of the past instance. Invalidated pages will be reclaimed for free space by GC. Reclaiming space may require that some valid data be copied forward. In Table 3.1, 'GC-Writes' shows the extra writes performed by the flash layer. The FLWA for this workload ranges from 5-6x for different cache sizes.

Collective Endurance Impact: The 'TCWA' column in Table 3.1 shows the Total Combined Write Amplification effects of FLWA and CLWA. Since the two operate independently, the original workload's writes are amplified by the cache and each cache write is further amplified by the flash device. The result is a 40-50x amplification in writes as compared to the original workload. Left unaddressed, these effects can render the endurance of SSC devices unacceptably low - limiting how long they are useful as accelerators for storage systems.

3.3 Simulator and Analysis Methodology

In this section, we describe the traces used in our analysis, the tools we built to characterize traces, and the simulator we built to evaluate our cache and flash layer optimizations.

3.3.1 Block Traces and Workload Analysis

The block traces represent various possible cache workloads. To ensure robustness and broad applicability of our results, we have selected a variety of traces with differing characteristics of cacheability, deployment scenario, trace size, and unique data size.

For example, sequential requests are normally not cache worthy as they cause cache pollution, however they are good candidates to aide gauging the effectiveness of a selective cache admission policy. Another example is that of larger read/write ratios, which indicate stronger desire to cache and are more likely cases for a cache deployment. We looked at 35 traces from Microsoft Research [5] and several TPC-E workload traces generated from database runs [11].

We built a Workload Analyzer (WLA) to characterize block traces and help inform trace selection for our experiments. The metrics reported by the WLA include unique sector counts, critical cache size, read/write ratio, and histograms for read/write Logical Block Address (LBA), sequential and non-sequential addresses, request sizes, and sequential access lengths. The *Critical Cache Size* is the minimal cache size that is required to cache the admitted trace data without a single eviction. Critical cache size is therefore highly dependent on the specific admission algorithm. When all cache misses are admitted by default, the critical cache size is the same as the unique overall sector count. As the admission policy becomes more restrictive, the actual cache size necessary to cache all admitted sectors decreases. Table 3.2 presents a partial WLA summary for the six traces used in our analysis.

Trace Name	Description	Total R GB	Total W GB	R/W Ratio %	Unique GB	Total req (K)	Non Seq req (K)	Seq req (K)
stg_1	Web staging	79.5	5.99	93.0	80	2197	1077	1120
web_2	Web server	262.8	0.78	99.7	66	5175	1990	3185
hm_0	HW monitor	9.96	20.48	32.7	2.31	3993	2560	1432
prn_0	Print server	13.12	45.97	0.29	14.80	5586	2761	2824
prn_1	Print server	181.35	30.78	85.5	80.90	11233	8098	3134
tpce	Database	241.61	88.96	73.1	127.25	19086	18366	719
tpcep	DB+backup	331.90	37.81	89.8	226.52	9442	8120	1321

Table 3.2: Trace Characteristics

3.3.2 SSC Simulator

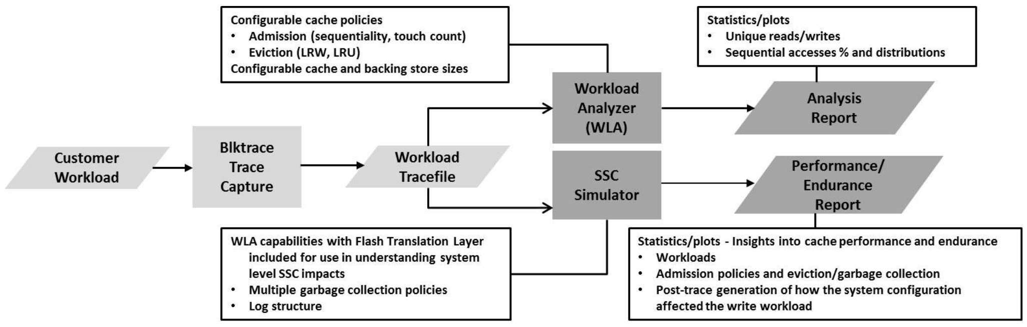

Configurable cache policies
• Admission (sequentiality, touch count)
• Eviction (LRW, LRU)
Configurable cache and backing store sizes
Statistics/plots
• Unique reads/writes
• Sequential accesses % and distributions
Workload Analyzer (WLA)
Analysis Report
Customer Workload
Blktrace Trace Capture
Workload Tracefile
SSC Simulator
Performance/ Endurance Report
WLA capabilities with Flash Translation Layer included for use in understanding system level SSC impacts
• Multiple garbage collection policies
• Log structure
Statistics/plots - Insights into cache performance and endurance
• Workloads
• Admission policies and eviction/garbage collection
• Post-trace generation of how the system configuration affected the write workload

Figure 3.1: HEC Analysis and Simulation Environment

The SSC simulator mimics the behavior of an SSC and contains two major components: the cache controller and flash controller. The cache controller simulates cache behavior. It reads the input block trace, and depending on the user-chosen cache mode and the admission policy, performs the caching actions. The flash controller simulates a NAND flash device. Similar to a solid-state storage (SSD) device, it includes an FTL, GC, wear-leveling and other flash-related capabilities. We validated our SSC simulator by comparing its results against an SSC product - directCache [1], for a subset of the traces.

Figure 3.1 shows the overall workflow of our trace analysis and simulation environment. We perform a workload analysis, where we determine the characteristics of each trace. Based on the analysis report, we select traces possessing a spectrum of interesting characteristics.

The internal view of a typical SSC is introduced in previous section 2.1. While cache controller maintains the components of a generic cache deivce, flash controller consists of all modules that are needed to make a raw flash memory device work properly in the system. Then we run these selected traces through our SSC simulator, varying both admission and eviction policies, and evaluate the workloads' behavior under different cache and flash configurations. We describe each of these components in turn below.

3.3.2.1 Cache Controller

The cache controller supports two modes of operation, as is shown in Figure 3.2. The first mode consists of *Cache-based* eviction, which utilizes both an in-memory fixed-size valid-data-only cache and an FTL log-structured cache. Both embodiments maintain the same fixed valid sector count *in this mode.* When the in-memory cache becomes full, a least recently written (LRW) or least recently used (LRU) element is evicted, one sector at a time, and a signal is sent to the FTL log to invalidate that sector (one at a time) using the industry standard TRIM operation [3]. The Cache Controller may then overwrite the in-memory block with the newly admitted sector, and the flash controller is now free to reclaim this sector under GC. In this mode, GC copies-forward all valid data, as the eviction in the upper cache controller is invisible to the lower flash controller. This allows the cache controller to be more flexible on which sector to evict once it is full, for example, based on the hotness of data in the cache.

The second mode consists of *GC-based* eviction, which utilizes only an FTL log-structured cache, and flash-based GC algorithms to perform cache eviction. In this mode, the cache controller performs no eviction, and the flash controller evicts data during its GC process. The cache discovers the presence or absence of data by querying the FTL via an EXISTS operation. In the *GC-based* eviction model, the GC can use cache-specific intelligence (eviction) to improve endurance.

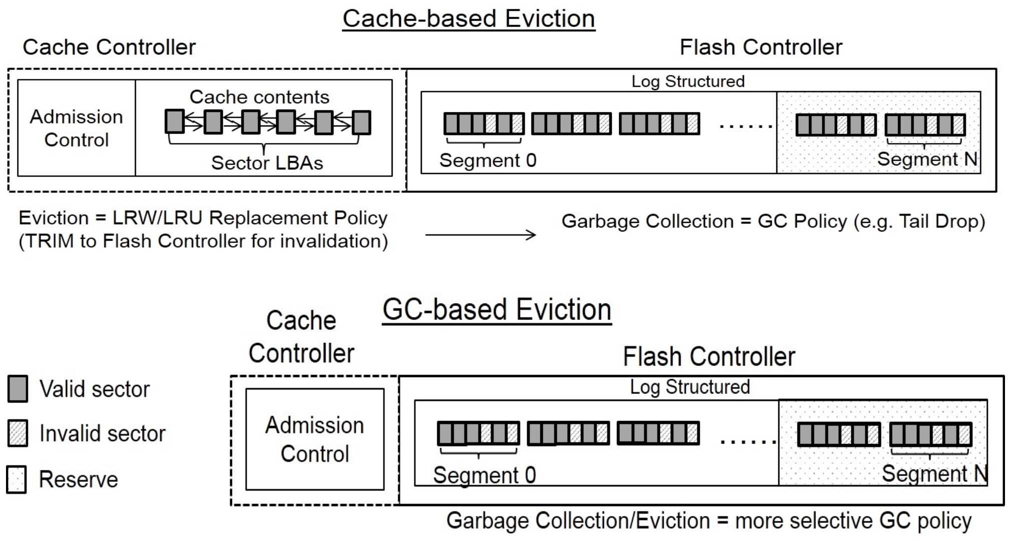


Figure 3.2: Two eviction Modes

3.3.2.2 Flash Controller

The flash controller implements a log structure that continually appends new data at the head of a log, similar to existing FTLs. Metadata for the flash controller, such as maps of valid and invalid blocks and erase counts, are kept in-memory for fast searching. The controller also supports various GC algorithms to free-up erasure blocks or segments for further cache usage when the device fills up. We use the term *GC Segment* to refer to an erasure block that consists of a fixed number of flash pages for a specific hardware configuration.

3.4 Reducing CLWA

In this section, we explore techniques to reduce writes at the cache layer by selectively admitting cache misses. We describe and evaluate a series of cache admission policies (see details in previous work [30]). The settings for the policies mentioned below were chosen precisely because they perform the best across all the workloads we consider in this paper.

3.4.1 Cache Admission Control

We investigate two classes of admission policies, one focused towards excluding sequential traffic and the other focused towards admitting data based soley on it's access frequency. The default policy is *ADMIT ALL* where every miss is admitted to cache. *ADMIT ALL* has drawbacks as it does not filter cache pollution (such as backup operations or sequential scans) nor evaluate the quality of the request data, thus being detrimental to hit rate and impacting endurance by generating large numbers of cache-miss writes. On the other hand, for random workloads, the absence of large-scale sequentiality (only small looping requests), and with somewhat high read recurrence (read before eviction), this policy can be beneficial. As a policy for the generic workload that has random workloads mixed with sequential requests, clearly *ADMIT ALL* is suboptimal. Such lack of discrimination leads to cache flooding, cache pollution, increased WA, and shorter NAND life.

Selective Sequential Rejection (SSEQR): All non-sequential requests are admitted to cache, and short sequential strings are admitted to cache. We will explain the main concept first, then describe the selective portion of the filter. We keep an array W (the window), of request beginning and ending addresses $we_k = (ba_k, ea_k)$ (LBA's in our implementation), for the most recent k requests: $W = \{we_1, we_2, ..., we_k\}$.

On a request r, its start address ba_r is compared against the ending addresses in W. Each request's address data-pair is always inserted into W as a new window entry we. If ba_r is arithmetically subsequent to any of the end address entries in W, within an allowable gap G, then r is *sequential* and is denied admittance into cache.

$$\text{If } \exists ea \in W | 0 \leqslant ba_r - ea \leqslant G \Rightarrow \text{reject } r$$

We now explain the selective part of this filter. Employing the same framework just mentioned, we track the length of each sequential string and allow cache admissions of strings up to a configurable length L in megabytes. In short, this policy admits small sequential strings into cache, and is our replacement to a looping sequential request detector [50]. Typically, values satisfying $L \leqslant 4$ MB are best, though in our implementation we chose $L = 4$ MB. In our experiments, the default settings are: $(k, G) = (8, 4)$.

Touch Count (TC): Sectors or collections of sectors are admitted only after they have been accessed a certain number of times, thus earning their way into the cache. Since this technique requires the ability to track activity to not just cached sectors but to the entire backing storage device, we employ memory efficient ways to track accesses. To accomplish this, we employ a bitmap B representing the backing store, together with a hashing scheme (think of the bitmap as a touch count ledger - tracking how many times backing store addresses have been touched). We also define a *segment* to be a contiguous string of sectors, typically a power of two in length, i.e., 2^a. At the initialization stage, knowing the size of the backing store, we create the bitmap array. This bitmap consists of storage for the touch-count counts for all segments, employing j bits per each segments touch count.

As requests are tendered, the bitmap is modified to keep track of which segments have been touched and how many times. When a segment has attained a touch count equal to the threshold, then it is admitted to cache.

When a request is tendered, the touch count is incremented for each segment that the requests addresses intersect. It is possible (commonly) that a request not begin or end strictly on segment boundaries (recall that a is configurable) and can span segment boundaries. Thus, we increment the touch count for each segment that intersects the request.

Checking and incrementing a touch count value is fast and is performed via masking the bits of B that correspond to each segment intersecting the request. If any of the segments intersecting a request has attained the threshold

t (user configurable), then the request is admitted. Furthermore, if any portion of a requests touch-count has attained the threshold then it is admitted even though other portions of the requests intersecting segments have not yet reached the threshold. We call this *touch count promotion.*

As workloads progress throughout the day, the touch counts become stale as they represent earlier activity that may not be currently valid. To ameliorate this, we employ multiple bitmaps Bi, for i = 1,2,3,..., and either: (1) a time-based rotation schedule, or (2) a data-driven rotation schedule R where the bitmaps are rotated to a fresh (zeroed) map every R GB of data. The other bitmaps are used as a bitmap history to help workloads and touch counts blend throughout continued activity over time. For this experiment, we employed one bitmap with one bit per touch count entry, and segment size is $4MB$.

Sequentiality and Touch Count (SSEQR+TC): If a request is not deemed sequential, and if its constituent segments have attained the touch-count threshold t, then admit.

3.4.2 Admission Policy Analysis

Table 3.3 shows the hit rates and writes that result from each admission control algorithm being applied to the TPC-E Polluted trace. Given the large backup-based read content in this trace, under *ADMIT ALL* a significant amount of sequential writes are generated, leading to a CLWA of 2.5-8x. However, when admission policies are applied, both the CLWA and the effective cache size required to hold the cached data decrease. For example, when *SSEQR* is applied, a hit rate of 79% is achievable with a 138GB cache, with a CLWA of 1.7. In comparison, *ADMIT ALL* achieves a lower hit rate (41%) with a similar cache size (136GB), but much larger CLWA (7.73). In fact, the *ADMIT ALL* policy is only able to achieve high hit rates when the cache is large enough to hold the backup traffic as well as the database traffic.

At smaller cache sizes, the admission policies achieve higher hit rates and lower CLWA than *ADMIT ALL*, while generating fewer writes to the flash layer. As the physical cache size increases, the more restrictive admission policies become limited - *ADMIT ALL* can achieve a higher hit rate since it *does* make use of the extra capacity.

Policy	Orig		Cache	Phy	Post Trace					Hit	CLWA
	R	W	Size	Size*	Read	Write	R-miss	W-hit	W-miss	Rate	
	GiB	GiB	%	GiB	GiB	GiB	w GB	w GB	w GB	%	
ADMIT ALL	331.9	36.8	50	135.91	84.40	284.31	247.50	32.70	4.10	41.26	7.73
			70	190.28	170.40	198.31	161.50	34.21	2.59	62.05	5.39
			90	244.64	279.50	89.22	52.41	35.87	0.93	87.08	2.42
SSEQR	331.9	36.8	50	76.66	71.71	161.07	124.27	32.22	4.59	20.86	4.38
			70	107.33	132.21	116.64	79.84	34.04	2.77	47.89	3.17
			90	138.0	200.38	62.60	25.80	35.88	0.92	79.01	1.70
TC	331.9	36.8	50	64.83	44.33	169.28	134.56	30.93	3.80	19.22	4.60
			70	90.76	85.87	142.66	107.94	32.27	2.45	34.66	3.88
			90	116.69	149.21	95.60	60.88	33.39	1.34	55.66	2.60
SSEQR +TC	331.9	36.8	50	53.57	56.75	136.78	101.87	31.27	3.64	15.21	3.72
			70	75.0	98.43	111.22	76.31	32.67	2.24	32.61	3.02
			90	96.43	150.94	72.33	37.42	33.85	1.06	57.34	1.97

Table 3.3: Evaluation of different cache admission policies (TPC-E Polluted): restricted admission policies achieve higher hit rates with reduced writes to the flash media when cache size is smaller.

Figure 3.3 depicts the total number of bytes written to the flash, the erase counts, and the hit rates for four admission policies on four traces. From the figure, we see that sequentiality and touch count detection can reduce the total number of writes to the flash media (up to 60% and 80% for prn_1 and tpcep traces, respectively, at larger cache sizes). The erase counts are also reduced, leading to extended device life. Overall, the *SSEQR* policy is able to filter the backup pollution traffic from the TPC-E Polluted trace, but the *TC* approach is able to be generally discriminating of quality traffic across a variety of trace types (observe the relationships in all graphs in Figure 3.3). The hybrid combination of the two polices, *SSEQR+TC*, generates high hit rates with substantially lower CLWA values than *ADMIT ALL*.

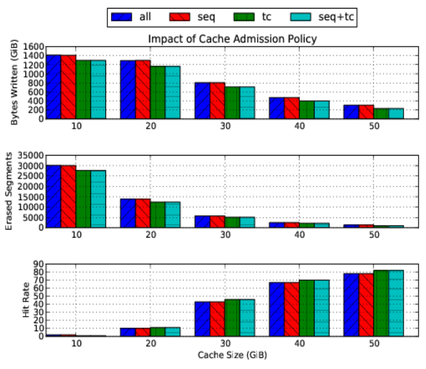
3.5 Reducing FLWA

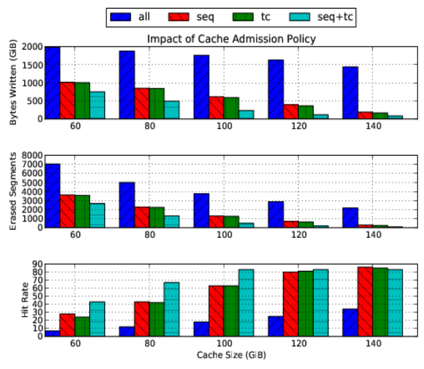
In this section we describe and evaluate FTL techniques for reducing FLWA by way of various GC algorithms and *Cache-based* versus *GC-based* eviction

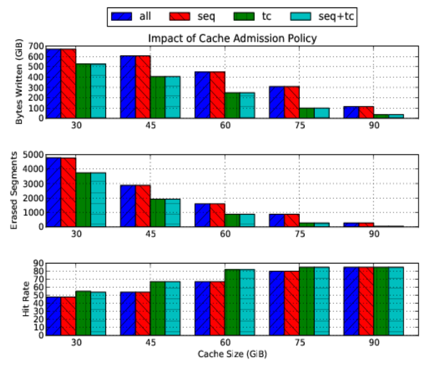
3.5.1 Cache-based vs. GC-based Eviction

While traditional SSD devices can only reclaim invalidated pages, an SSC device can evict valid data based on the cache controller's knowledge of data usage. Eviction of valid data improves GC efficiency with less data being copied forward. As part of our quest to reduce FLWA, we investigate GC algorithm effectiveness under both *Cache-based* and *GC-based* eviction.

Cache-based: In this mode, each GC algorithm only considers the invalid space as seen by the FTL. When the cache is full of valid data (as perceived


(a) web_2 trace under different admission policies


(b) TPC-E Polluted trace under different admission policies


(c) prn_1 trace under different admission policies

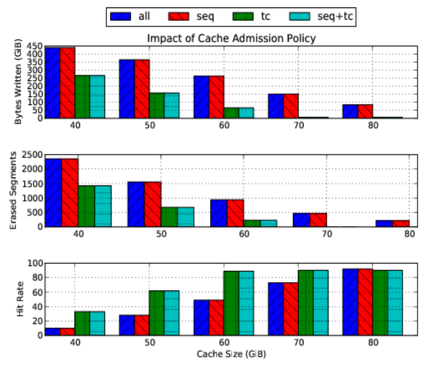

(d) stg_1 trace under different admission policies

Figure 3.3: Effects of different admission policies: total number of bytes written to flash are reduced by up to 80% through sequentiality tracking and touch count detection for selected traces, leading to longer device life with improved hit rates.

by the cache controller), it signals the FTL to invalidate one sector at a time, based on the cache's replacement policy. GC then utilizes victim-selection algorithms (such as OLDEST) to reclaim space, no valid data is ever discarded in this mode. It is important to note that the cache layer is not FTL-aware and that the LRW or LRU list maintained by the cache controller is not representative of the FTL view. Furthermore the cache controller LRW or LRU list represents the valid data view of the SSC size which is the $total\_device\_physical\_size - system\_reserved\_capacity$ or 80% of the total device physical size.

GC-based: In this mode, the GC algorithm usurps the eviction responsibilities from the cache controller, and valid data in the victim segment are discarded by the flash controller. Some applications require a certain number of valid sectors to be preserved in the device to guarantee a certain hit rate without aggressively evicting hot cache entries. This is achieved by tracking the number of valid pages in the media and letting GC perform eviction or copying forward based on the usage. Note that in this mode, the GC algorithms valid data view is the view over the *total_device_physical_size*. As will be shown in Section 3.5.2.3 these different validity views has an impact on hit rate, FLWA, and segments erased between the two eviction modes.

3.5.2 GC Algorithms

We implemented and evaluated three commonly-used GC algorithms for victim segment selection. Their descriptions and impacts on FLWA are described below. These three algorithms can be used in both *Cache-based* and

GC-based eviction modes.

Greedy Segment Selection: Selects victim segments with the most invalid pages. From both cache and storage usage viewpoints, this algorithm is the most space efficient in that it frees more usable space by erasing the least number of segments. However, it runs the risk of wearing out some segments earlier than others as some hot segments may be selected more frequently without concern for erasure count.

Cost-benefit: Employs a utilization function that considers both *space* and *age* [80] during victim segment selection. This algorithm provides improved wear-leveling over the greedy algorithm and attempts to group data with similar access frequencies during GC, which can then result in lower FLWA by segregating cold data.

OLDEST Segment (Tail-drop): GC chooses the oldest segment programmed in the log (similar to the LRW policy at the cache layer). This provides low victim-segment search overhead, and potentially chooses one that is less active. However, this can result in copying forward large amounts of data (i.e. high FLWA) if the segment in question does not possess many invalidated sectors.

3.5.2.1 Configurations

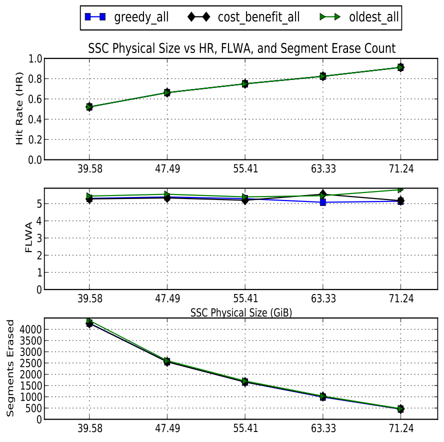
To isolate the impact of GC policy, we use the *ADMIT ALL* admission policy in all our experiments and only vary the GC policy for different SSC physical sizes. We determine the impacts of *Cache-based* eviction relative to

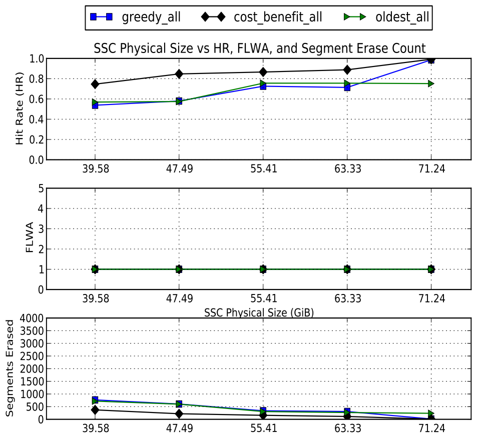
GC-based eviction. We evaluate the effectiveness of each algorithm with regards to hit rates, FLWA, and segment erase counts. For an accurate comparison between GC-based and cache-based eviction, we force the SSC to keep the same amount of valid data (i.e., 80% of physical device capacity as 20% is reserved capacity) in both modes in order to guarantee the amount of cacheable data remains the same.

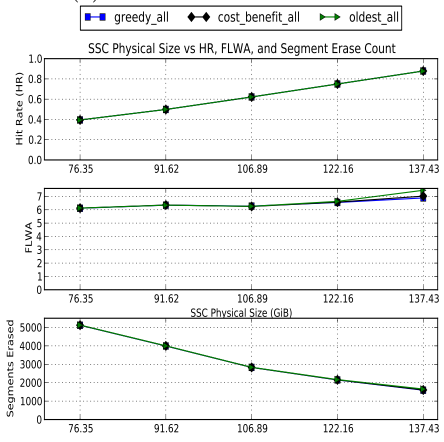
3.5.2.2 Eviction Mode and GC Policy Analysis

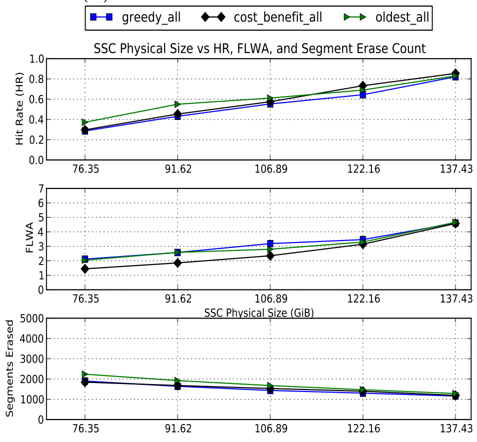
Figure 3.4 shows the test results under four different workloads for both *Cache-based* and *GC-based* eviction approaches. The left graphs (i.e. 3.4(a), 3.4(c), and 3.4(e)) show the performance of *Cache-based* eviction while those on the right (i.e. 3.4(b), 3.4(d), and 3.4(f)) display *GC-based* eviction. While comparing *Cache-based* with *GC-based* eviction approaches, we see that the hit rate and FLWA vary between different eviction modes, even though both modes guarantee the same amount of valid sectors in the SSC.

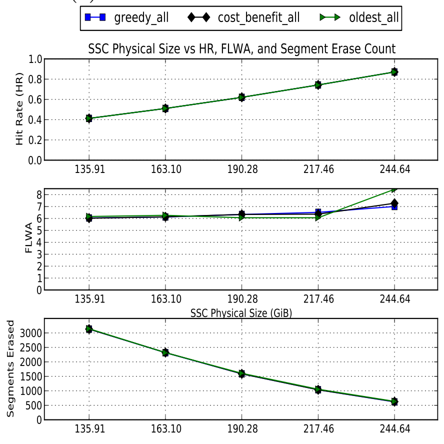
Some broad trends are as follows: first, the *GC-based* eviction has broadly lower FLWA than *Cache-based* eviction, regardless of workload or GC algorithm. The improvements in FLWA range from 1.5x to approximately 5x. Second, the hit rates of *GC-based* eviction appear to be roughly similar to that of *Cache-based* eviction, with a difference of 5-10% under different GC selection algorithms. Third, the FLWA of *Cache-based* eviction does not appear to improve with any particular GC algorithm. For all cases except the stg trace, the results of all three GC algorithms is roughly the same. Fourth, reductions in segment


(a) web_2: Cache-based eviction


(b) web_2: GC-based eviction


(c) TPC-E: cache-based eviction


(d) TPC-E: GC-based eviction


(e) TPC-E Polluted: Cache-based eviction

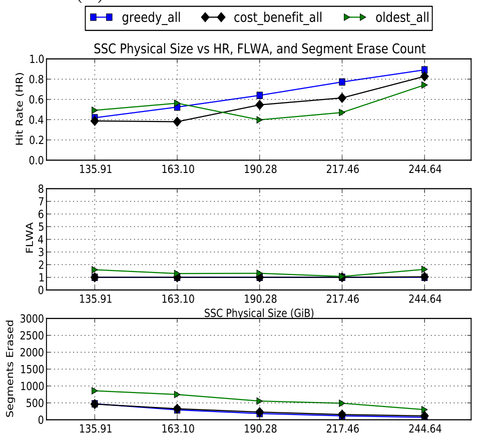

(f) TPC-E Polluted: GC-based eviction

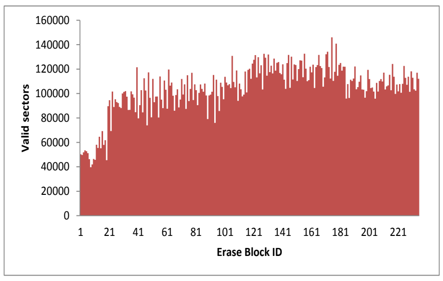
Figure 3.4: Effects of eviction modes and GC policies varying SSC sizes.

erase counts up to 6x were seen using *GC-based.* Finally, where there is a discrepancy in the results among different GC algorithms, in the *GC-based* eviction cases, the *Greedy* and *Cost-benefit* algorithms appear to outperform *OLDEST*. We observe that a more careful victim-selection algorithm, such as *Cost-benefit*, yields up to 60% FLWA improvement and up to 50% erase counts improvement over a simplistic SSC algorithm like *OLDEST*.

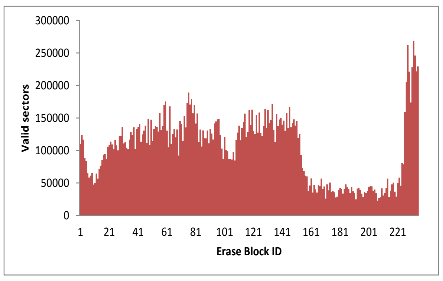
3.5.2.3 Interactions between Cache and Flash Policies

Figure 3.4 clearly shows that cooperative GC, as performed under *GC-based* eviction, results in lower FLWA than independent GC as performed by *Cache-based* eviction. For example, FLWA is reduced for the TPC-E trace using *GC-based* eviction with the *Cost-benefit* policy by 1.5x to 4x depending on the physical cache size. A reason for this is the difference between the distributions of invalid data in the segments between the two eviction modes.

For *Cache-based* eviction, candidate segments are chosen based on recency (or frequency) from the cache's perspective (i.e. in-memory data). In contrast, *GC-based* eviction predominantly looks at invalidity (or recency) from a storage perspective (i.e. on media). As a result, the valid sectors evicted by the cache controller are different from the ones evicted by GC, even though both use similar GC policies such as *OLDEST*. This results in a different view of the valid sectors' distribution throughout the log. Figure 3.5 represents the distribution of invalid sectors in each segment under the two eviction modes for the TPC-E trace. While the total number of invalid sectors in the SSC remains the same, the deviations are large. As shown in Figure 3.5(a), *Cache-based* eviction created a uniform distribution of invalid data across the segments for the TPC-E workload. Such uniformity in the distribution did not provide advantage to any of the GC algorithms investigated under *Cache-based* eviction and resulted in similar FLWA and segments erased for each; see Figure 3.4(c).



(a) Cache-based eviction



(b) GC-based eviction

Figure 3.5: TPC-E: valid sectors distribution in log for different eviction modes, GC-based eviction shows higher deviation of valid data thoughout log than Cache-based eviction, providing more flexibility on victim-segment selection.

As shown in Figure 3.5(b) the distribution in invalid data amongst the erasure segments is non-uniform in *GC-based* eviction. Non-uniform invalidity is desirable for GC since it enables algorithms to select segments with large numbers of invalid blocks. The non-uniform distribution in *GC-based* eviction demonstrates that a more selective GC policy can yield better effectiveness with regards to reducing FLWA. The benefits include generation of additional free space while maintaining a lower segment erased count, and the provision of greater room for the upcoming hot write-data with higher efficiency.

The data also shows that workloads with higher relative reads, Figure 3.4(b, f), generate lower FLWA under *GC-based* eviction. The lower FLWA is caused by the fact that invalidity across the log in *GC-based* eviction mode is caused by workload writes that caused overwrites of logical addresses. These overwrites result in physical locations in the FTL log becoming invalid. As a result, the log has less valid data. When a valid data capacity reservation is enforced, the amount of freedom the *GC-based* eviction algorithm has to evict data is dependent on the difference between the valid data in the log and the value of the capacity reservation. The more invalidity the log possesses, the lower this gap and the greater the likelihood that *GC-based* eviction will be forced to copy data forward, thereby generating higher FLWA.

3.6 Combined Benefit

In this section we put it all together and demonstrate the combined benefit of increasing hit rate, reducing CLWA and FLWA, and any unexpected interactions between admission control and eviction schemes.

Original W (GB)	Cache Size (GB)	Cache W (GB)	GC W (GB)	Total W (GB)	CLWA	FLWA	TCWA	Hit Rate %
36.8	80	70.10	169.19	239.29	1.90	3.41	6.50	46.59
	100	37.65	169.05	206.70	1.02	5.49	5.62	59.78
	120	37.65	44.80	82.45	1.02	2.19	2.24	59.78

Table 3.4: TPC-E Polluted: Collective endurance impact under SSEQR+TC and cost-benefit GC-based eviction mode, with well-selected cache and flash layer algorithms, TCWA can be reduced by up to 20x with improved hit rates.

Table 3.4 shows the combined effect of deploying an admission control policy *SSEQR+TC* and a *GC-based* eviction scheme using *Cost-benefit* segment selection, two techniques that individually generated good results for CLWA and FLWA reduction, respectively. The combined solution generates TCWAs of between 2-7, which is 8x-20x better than the original TCWA generated by the combination of *ADMIT ALL* at the cache controller layer and conventional (non-cache aware) GC at the flash controller layer (see Table 3.1).

The summarized data for the other five traces are shown in Figure 3.6. Due to space limitations, we present one set of cache sizes for each trace, and show the TCWA and hit rate before and after the cache and flash layer optimizations, with TCWA normalized to the "before" state. Fifteen (15) of 18 simulations show increased or maintained hit rate with reduced TCWA. Three (3) of them showed reduction in TCWA at the expense of hit rate (9-25%). These results demonstrate that significant benefits in device endurance can be achieved for cache workloads when flash-aware policies are used at both the cache and the flash controller layers. They also demonstrates the value of cache-aware GC policies at the flash controller layer.

Tables 3.1 and 3.4 show that hit rate can be significantly improved (2-3x) while still reducing CLWA and FLWA. Such hit rate improvement is workload dependent (e.g. TPC-E Polluted trace), however we found the results to be generally applicable to the workloads we have tested.

We also looked at what improvements could be gained if we made changes independently to the cache controller policies and GC algorithms. When

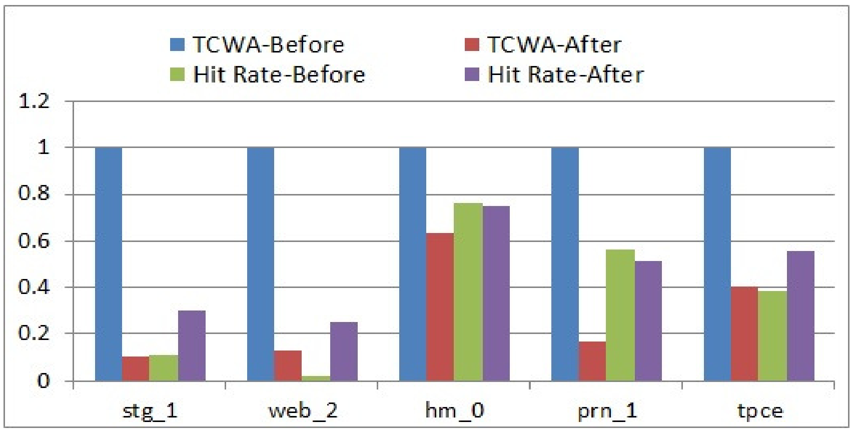


Figure 3.6: Comparison of TCWA and hit rate before and after cache and flash layer optimizations: TCWA is significantly reduced with well-selected algorithms. Hit rate maintains a certain level and is improved under some workloads.

we implemented changes to the flash controller layer only we observed a 1.1x to 6.17x FLWA reduction. We observed CLWA decrease in 71% of the simulations by 1.02x to 2.19x, 24% remained the same, and 5% increased by less than 10%. TCWA decreased by 1.11x to 9.61x or remained the same. We also observed hit rate improvements up to 10% in 71% of the simulations while 29% remained the same.

When we implemented changes to the cache controller layer only we observed a 1.08x to 8x reduction in CLWA. We observed FLWA increase in 41% of the simulations from 1.05x to 1.58x with 18% reducing by 1.17x to 2.82x and 41% with the same FLWA . Overall, TCWA decreased by 1.09x to 13x or remained the same for all simulations. We also observed hit rate improvements in 58% of the simulations, 29% remained the same, and 12% reduced by less than 10%.

We performed similar tests across all the workloads using an LRU replacement instead of LRW replacement eviction policy. The results for LRU and LRW were similar with respect to the benefits achieved by *GC-based* eviction over *Cache-based* eviction. We believe the cause to be the uncoordinated result of GC operating at a different segment size than the cache eviction. GC moves data around in physical flash, causing the evictions performed by the cache layer in *Cache-based* eviction mode to become uniformly distributed across the flash in both LRU and LRW scenarios.

3.7 QoS with Virtual Storage Containers

As virtualization technologies become involved, such as Virtual Storage Containers (VSC), a single physical device could be partitioned into several cache instances of different sizes to favor various user application requirements. Cache and flash controller design choices, when optimized for virtualization technologies, present potentially further complications (ensemble settings) because of a single shared log instance. Indeed, for example, to guarantee a certain Quality of Service (QoS) on the hit rate for one application on a specific partition, the GC may have to invalidate log data belonging to other VSCs, resulting in more data copied forward during garbage collection. In this case, admission, eviction, and garbage collection will have to be aware of these requirements for improved performance.

When Virtual Storage Container (VSC) is turned on, I/O requests with different LBA ranges are put into different logical containers. Only one log is maintained by VSC-aware FTL for mapping between logical and physical addresses on the device.If virtualization technology is introduced where a single physical device is divided into partitions, active data from one partition may be obliterated by data from another partition due to their positions in the log. This will result in a dramatic decrease in performance as the eviction component is oblivious to the QoS requirement on each logical instance.

In todays commonly available SSC devices, silent eviction happens at the tail of the log when the device is running out of space. This improves garbage collection efficiency with no data being copied forward. However, due to flash memorys out-of-place update nature, each segment usually contains a large number of valid and invalid pages. Evicting valid sectors reduces the amount of useful sectors in the log and produces negative effects on the read hit rate, especially under a random workload where the usage of the log is very low, making the hit rate QoS far from satisfactory.

In addition to SSD-aware algorithms as discussed in Section 3.5, we seek solutions to SSC-aware algorithms that leverage cache eviction strategies as well as SSD-based metrics.

$\frac{benefit}{cost} = space * w1 + age * w2 + VSC * w3$

As is shown above, *space*, *age*, and *VSC* are metrics in different dimensions to indicate the GC efficiency, forexample, the number of valid pages, the last time each segment is modified, and different VSC activeness and usage when multiple VSCs exist. Meanwhile, different weights w1,w2,w3 could be embedded based on SSC design choices. In this dissertation, we only focus on the combined effects of these factors on one cache instance. Intelligent cache-aware algorithms with interactions between cache and flash controllers, as well as multiple VSCs will be explored in the future.

For a single VSC instance in our framework, it allows a user-defined QoS setting that is known by the flash controller. To measure the performance and endurance, we conduct our experiments under various QoS settings, which forces the GC to perform copies forward so that the log can keep a minimum quantity of valid sectors and allow GC to perform evictions once the amount of valid data satisfies a certain threshold. This precipitates further design choices in the tradeoff between SSC performance and endurance - by adjusting the amount of valid data kept in the log.

In order to provide better QoS on cache hit rate, we measure the performance of different VSC configurations, with 0%, 50%, and 100%. When VSC is configured to be 0, there is no guarantee on the number of valid sectors the log has to keep. At the time GC starts, it can silently evict all valid sectors in the victim segment. When VSC is configured to be 100, flash controller will force the log to keep at least 80% of valid data while the rest of 20% is reserved for garbage collection. As a result, GC copies forward a large amount of data to satisfy this QoS setting.

Similar to other experiments, we conduct the tests with cache size from 50% to 100OLEST, and 8% of GC threshold. Is is shown in Figure reffig:vsu that as different QoS settings force different number of valid sectors to be stored in the log, this result in an increase of cache hit rate as vsc qos min increases.

On the other hand, in order to guarantee a certain usage level, GC has to do valid data copying forward when the usage is below the required vsc qos min instead of silent eviction. As vsc qos min increases, write amplification factor increases accordingly, causing additional writes to the flash. There is a tradeoff between performance and endurance of SSC, design choices could be made to minimize the WA if hit rate is guaranteed within a certain range.

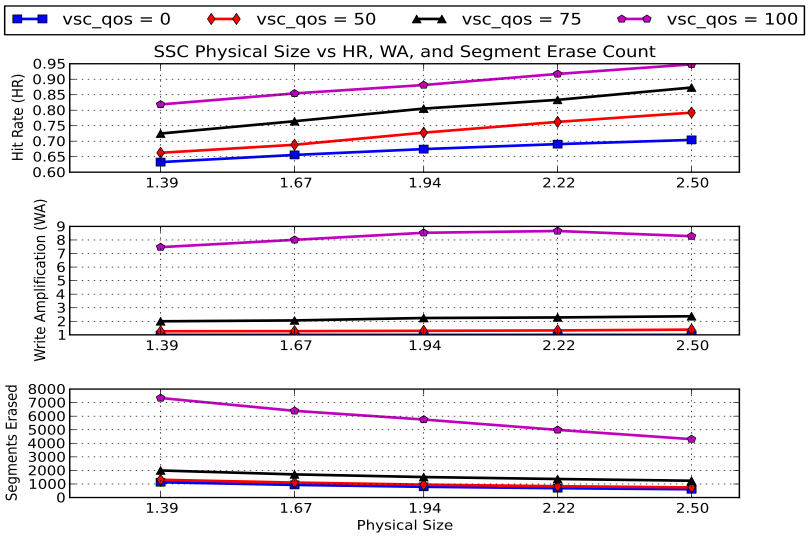


Figure 3.7: hm_0 performance with different VSC QoS configurations.

3.8 Future Work

In this section, we have exclusively explored write-through as the caching mode for an SSC. In the future we plan to extend our work to write-back caching. Unlike write-through mode, in write-back caching mode the most-recent copy of the data would not always be in the backing store but in the SSC itself, needing to be asynchronously written to the backing store (destaging). The asynchronous update to the backing store introduces many challenges to GC policies, admission policies, and performance of regular operations. It can however enable more write-intensive workloads to become cache candidates, making it a good target for endurance-focused investigations.

In this section, we have looked at only one cache instance within a single SSC. In many data center deployments, due to virtualization, more than one cache instance would be running within the same device. We intend to explore the impacts of multiple independent cache instances on the same FTL. In our work we focused on comparing FTL and cache layer policies and GC versus cache layer eviction with configurations where the amount of valid data guaranteed (e.g. hit rate QoS capacity reservation) in the SSC was fixed at 100%. Preliminary work was performed with varying hit rate QoS percentages (e.g. 0%, 50%, 75%) and that work will be the basis for future investigations on impacts to hit rate, CLWA, and FLWA in such configurations. We intend to explore the impacts of multiple independent cache instances on the same FTL, extending the work presented so far on QoS and its interaction with GC and cache algorithms. We also intend to explore the implication of running multiple instance of write-back, write-through, and a combination of write-back and write-through caches within the same flash device. The performance-endurance tradeoffs in the multi-cache scenario will be challenging and could require global admission and eviction policies in addition to the local admission/eviction policies for individual caches.

3.9 Summary

The focus of our high endurance flash based cache was to analyze the individual contributions and cummulative choices of cache admission policy, cache eviction policy, GC strategy, and device configurations on the overall endurance and performance of a high endurance SSC device.

Through our experiments we have the following observations.

- Workloads from write-through caching software products with admission policies yield much different workloads as seen by SSCs. The resulting changes in workloads, due to such configurations, can result in cache layer write amplification (CLWA).

- *Cache-based* vs. *GC-based* victim selection, using differing GC policies, under various cache workloads, demonstrates much different hit rates, FLWA, and segment erases for differing cache sizes (different percentages of the critical cache size for each trace). These differences produce impacts on write workload, P/E cycles, and overall performance. We recognized varying data validity distributions in each approach which in turn had impacts on victim selection, CLWA, FLWA, and segment erases. We identified that such choices must be considered carefully to balance endurance with performance in SSCs.

- Use of capacity reservation with QoS enforcement on SSCs provides for improved hit rate at the expense of FLWA. We identified that when using capacity reservation and QoS, the hit rates approached that of cache-based eviction with improvements seen in both write amplification and segment erases as contrasted in GC-based eviction.

In addition we identified a number of areas for further research. The use of VSCs and QoS has shown improvements in balancing endurance versus performance. Additional study is required to understand the effects of multiple VSCs with QoS residing on a single SSC and the effects of garbage collection victim selection strategies on such configurations. As our study focused on SSCs configured as write-through devices further study is required on the impacts of VSCs with QoS in write-back cache configurations.

Chapter 4

Log-structured Flash-based Storage Systems

The use of log-structured in storage systems is to maximize write throughput by aggregating small random writes to large sequential ones. With logging, storage medium is easier to manage, for example, free space management becomes straightforward regardless of the fragmentation inside the log. Recovery mechanism could be simple and fast as newer data is always appended to the tail of the log. This is also good for keeping transactional consistency. Even crash occurs in the middle of a transaction, the last uncommitted message could be discarded with little overhead. Though garbage collcetion process takes up most of the resouce in a log-structured system, it provides more flexibility on reclaiming space based on different performance requirements, hardware configurations, and reliability needs.

With this said, individual logging design is always optimized for its own performance. Multiple logs within one application is even btter. For example, some log-structured system place hot and cold in separate logs, or store data and metadata in different logs. This groups data of silimilar update frequencies, making them be invalidated together within a short period of time, and hence minimize the GC overhead. In addition, independent logging applications achieve better isolation so that application-specific data and storage management could be feasible.

However, when considering flash memory as a primary stoarge medium in these systems, due to the nature of NAND flash that does not support in-place update, another log-like Flash Translation Layer (FTL) to map from logical to physical addresses must be adopted. Such duplicated logging on file system (or application) and device layers suffers suboptimal performance and reduced reliability. Data layouts become intermixed defeating the benefit of original log structure and reducing flash memory life time due to uncoordinated log activities.

4.1 Experimental Methodology

We pick F2FS and NILFS as examples of two real systems for experimental environment to describe the log-on-log architecture. NILFS is a newly designed log-structured system to improve data consistency with features like "continuous snapshotting". It is not primarily written to be flash friendly, so we use it as a baseline test for comparison with the flash-friendly log-structured F2FS design. Samsung's recently developed flash-aware file system - F2FS [39] also uses a log structure on the host side, the file system layer can have upto 6 logs that separates data and metadata based on their hotness. The redirection of data to different system logs aims to aggregate data of similar update frequency so that they have high chance to be invalidated together within a certain period of time, and hence reduce the GC overhead by copying forward less amount of live data. Such technology has been proposed for a while in several systems, such as DualFS [74] and hFS [98]. However, the benefit of such systems can only be obtained with a device that supports multi-append point technology - in which the underlying device can maintain more than one active segments at a time, or where the device log can have more than one append points other than the tail. There is limitation to do so on the hardware side, thus defeating the benefit of the original multi-log design. In addition, even if the FTL is able to support multiple number of logs with hardware support, the number of upper layer logs tends to grow faster than that of the underlying physical logs. With this said, we conduct our experiments as a more general case that uses F2FS or Nilfs on the host side and a Fusion-io's ioDrive2 [25] as a primary storage device. The ioDrive2 we use is a 785GB NAND-based MLC SSD with a single append point (single log) in the FTL layer.

For each problem described in the following sections, we measured the performance and collected several system statistics, for example, file system writes to the flash media, actual physical media writes, device garbage collection write amplification factor, and etc.

4.1.1 Log-on-Log simulator

In order to quantify the issues with stacking logs, in addition to experiments on real existing file systems, we also build a log-on-log simulator to mimic a system configuration that having one log stacking on another. The basic structure of our simulator could be depicted in Figure 4.1. Initially, there are two layers of logs, the upper layer log can be treated as a log-structured file system that maintians its own log, and does address translation from virtual to logical space. While the lower layer log can be treated as a physical device that stores data on the media, with a log-like structure, for example, FTL in solid-state disks, logical addresses are translated to actual physical addresses. Data could be passed across logs by a standard I/O interface through operations such as READ, WRITE, TRIM, and etc.

Same as a typical log-structured system, each log alone has its own components, such as metadata management that maintains log metadata, storage management that controls data placement, garbage collector to reclaim free space on demand, and etc. These components' activities within one layer are invisible to the logs in the other layers.

The simulator can stack more than two layers log together, each with its own management mechanism. In addition, we can extend more than one log in each layer. Each log appends data to the tail, and garbage collector starts to work independently when the log is out of space. In our simulation experiments, we only focus on the two-layer log scenario, with each layer up to two logs. Multiple layers and more than two logs will be further investigated in the future.

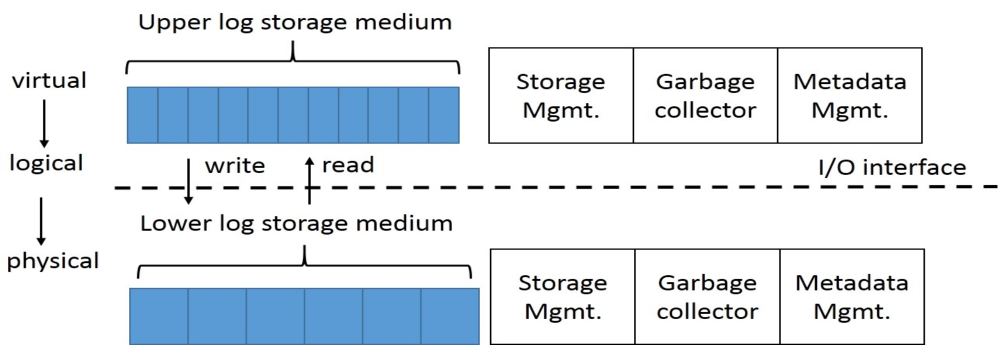


Figure 4.1: Log-on-log simulator

4.2 Issues with stacking logs

In this section, we show the problem with a log-on-log structured system. To simplify the issue, we start with a structure having one log stacking on another FTL log, we present the counter-intuitive behavior of such systems. We describe the system's deficiency based on several of its log activities, and then propagate to a more general two-layer log structured system where both file system or application and the hardware device has more than one logs on their own layer.

4.2.1 Metadata footprint increase

Increase in metadata footprint is unavoidable when logs are stacked on each other. At each log layer, one needs to maintain metadata that not only describes the log but also ensures that it always represents a consistent state. For example, in NILFS, the metadata blocks consists of file b-tree, inodes, inode b-tree, and checkpoint [53]. The on-disk space required by metadata for NILFS could be even higher to hundred KB for every file update, especially with direct

I/O. Moreover, most log systems also cache a copy of the on-media metadata in OS memory which increases with the number of log layers. When file system logs keep growing, the metadata needed to maintain each log increases. As an example, F2FS is designed to support up to 6 logs, making it possible for the file system to identify hot/warm/cold data and separate them to different segments. However, each log requires a nonignorable amount of metadata to manage its activity. These metadata not only increases the memory consumption, but also generates more writes to the device, and hence would be further amplified by the underlying log activities, for example, lower log garbage collection.

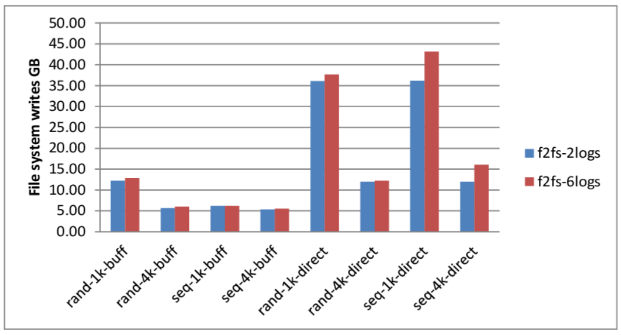


Figure 4.2: Metadata foot print increases as more logs are introduced on file system, defeats the benefit of multiple logs.

We measured the total file system write bytes issued to the device under different workloads varies number of file system logs. We configured F2FS to have 2 and 6 logs. With 2 logs, F2FS separates user data and metadata, while 6 logs further separate each type of data into 3 segments, hot, warm and cold. We use FIO benchmark tool to generate workloads with different I/O size.

The workloads include random and sequential read/write operaitons, with and without caching (O_DIRECT), I/O size of 1KB and 4KB.

As is show in Figure 4.2, under the same application workload that writes 8GB totally, the file system generally writes more data to the device when it grows from 2 to 6 logs. For example, the first column set shows the total number of file system writes issued to the device with an original 8GB random writes, buffered IO and 1k IO size. File system amplifies the original writes due to file metadata and log metadata. Since the user writes are the same for 2 and 6 logs, the file metadata used to maintain the file status remains the same. Thus, the increased writes from 2 to 6 logs are generated by the additional log metadata that is used to maintain the log status. In this chapter, we refer to the increased writes generated by file system from the original workload as File System Write Amplification (FSWA), which is computed as the ratio of file system writes to the device and the original workload writes. From our test, the FSWA varies under differnet number of file system logs, and increase from 1.5 to 2.0 (up to 33%) when file system grows from 2 to 6 logs in the test of sequential write, 4k IO size, and direct IO. This difference is larger comparing to the random workload (the last two column set in Figure 4.2). Under sequential workload, as each write operation extends the size of the file and requires ionode updates, as a result each write generates an additional metadata write which is not true for random workload. Thus in sequential writes that extending file size, metadata is more active and hence requires more log metadata to maintain the metadata log. With more frequent metadata operations, separating data and metadata introduces higher metadata footprint for those metadata logs. Generally, if some logs are more active than others in the system, they generate higher log metadata overhead.

As more logs are invoked on the upper layer, more metadata writes will be generated to maintain the log, and hence increasing FSWA. While separating data of differnet attributes or from different applications provides more design flexibility and aims to fully utilize the system resources, the increased performance overhead defeats such benefit and the additional writes reduce the flash device endurance due to the nonignorable amount of metadata foorprint increase.

4.2.2 Mixed workload

The nature of a log structure is to guarantee that data is written sequentially and appended at the tail of the log internally. In the upper layer log, file system or application writes sequentially within their own logs. However, the workload seen by the underlying device is quite different, with only one physical device and single lower log, workloads from logs and other non-log traffice get mixed, mostly likely to be random, especially when more than one logs are active at the same time - which usually happens in the real world. In addition, as disscussed above that lower log writes its own metadata, where the characteristics of metadata vary depending on the hardware design. Thus the actual traffic seen by the device is hard to predict. As is shown in Figure 4.3, IO requests from different upper logs intermingle with each other before reaching the device and results in the chaos especially under intensive workload.

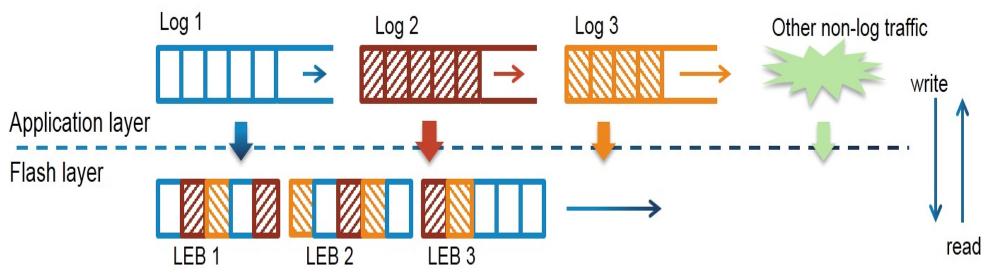
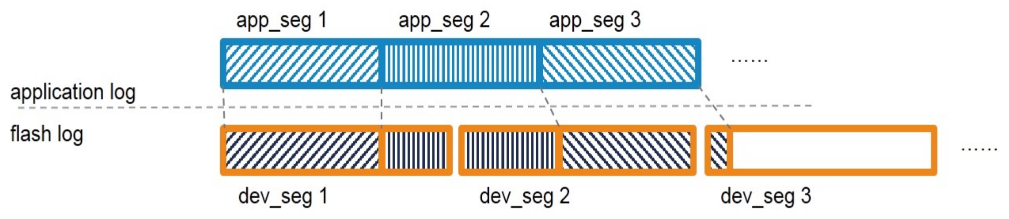


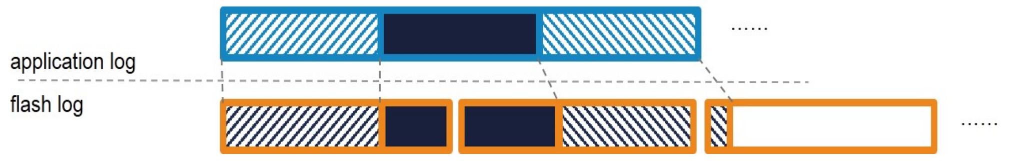
Figure 4.3: Mixed workload from logs and other traffic.

4.2.3 Unaligned segment size

Many of the advantages of sequentializing (or collocating) writes in large segments is lost when logs are stacked on each other. Among them, segment size mismatching causes most performance deficiency. First, segment size mismatch (i.e., unequal segment size) in logs leads to data spread across segment boundaries in the lower log. This results in data of upper log fragmented in the lower log. The degree of fragmentation gets amplified when each of these layers performs segment cleaning leading to increased write amplification. As an example shown in Figure 4.4, data from the same upper log segment is spread across segments in the lower device log, regardless of the ratio of segment size on each layer. At the time when an entire upper layer segment is deleted (fsys_seg 2), the invalidation is across two segments in the underlying device. Reclaiming physical space may have to do garbage collection of two dev_seg, and results in high device GC WA (see Section 4.2.5 for more detailed discussion on segment cleaning). The fragmentation becomes worse with multiple logs are stacked on top of one device log.



(a) Initially, application writes segment 1, 2, 3 sequentially to the device log



(b) Delete segment 2: upper log cleans one segment, underlying log cleans two segments

Figure 4.4: Unaligned segment size results in fragmented device space caused by garbage collection.

Second, matching segment sizes between logs does not prevent data fragmentation. Different logs have different behaviors and attributes, and hence different metadata objects and sizes. They write metadata objects at different conditions and frequencies. This results in variability in the capacity of segments within and across logs. Moreover, underlying logs have their own metadata which is invisible to the upper log, and intermixed the data from upper logs. It is impossible to align data perfectly based on their attributes (e.g. types of data, origin of data). Hence cleaning of segments at one log layer doesn't preclude the need to clean the segments at another layer. Even if the segment sizes exactly matches, the metadata written by each log layer would make it impossible to align data perfectly. As is discussed in the previous section that metadata for each layer log is nonignorable. Hence cleaning of segments at one log layer doesn't preclude the need to clean the segments at another layer.

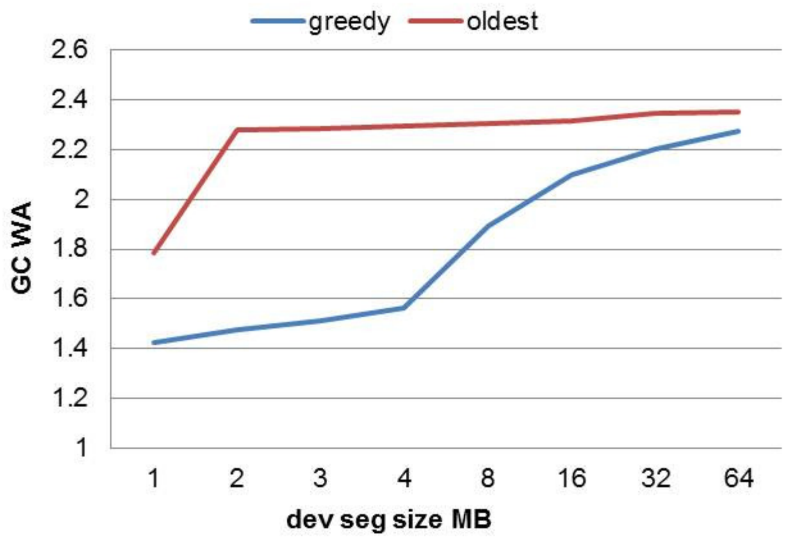


Figure 4.5: Device GC Write Amplification factor varies segment sizes.

We measured the impact of different segment sizes on device garbage collection by running the block trace captured in Figure 4.2 through our single-log SSD simulator. We picked block trace of random writes, 6 file system logs, 1k IO size, and buffered IO. The SSD is configured to be of size 6GB. As F2FS has a default segment size of 2MB, we vary the device segment size from 1MB to 64MB. As shown in Figure 4.5, GC WA increases as segment size increases. For example, under greedy policy that selects the victim segment with most invalid data, as segment size increases, GC WA increases dramatically, especially when device segment size is larger than the file system segment size. The increase of WA can be caused by both the unmatched segment size from two logs, and the lack of flexibility of choosing segments having less valid data with larger device segment size. In other words, if file system segment size matches the device segment size well or larger than the underlying segment, the chance of fragmented logs could be reduced. However, this is usually not feasible in the real system as device segment sizes vary by different vendors design and usually not easy to be configured.

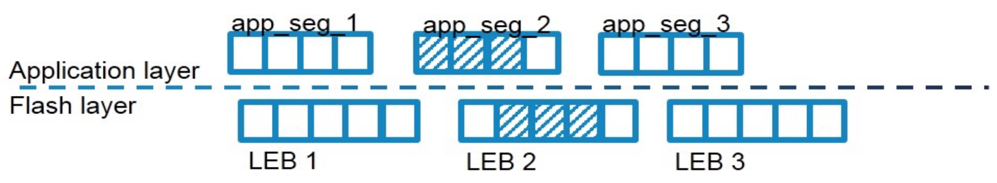
4.2.4 Different log activeness

In the case of multiple logs within one layer, if application separates data to different logs, logs will have different activeness based on the data type it stores. For example, metadata log is usually more active and smaller than the data log. As a result, each log triggers the data invalidation and garbage collection with different frequencies. Since the original workload remains the same, separating data to multiple upper logs tends to make the invalidation and garbage collection on one original log be spread across multiple upper logs. Once be propagated to lower log, it is likely to be across even more segments. The more number of upper logs are, the more dispersing lower level data is stored. More application or file system logs is not always good for overall performance, the reduced upper layer WA sometimes results in higher device WA, thus making the combined WA higher.

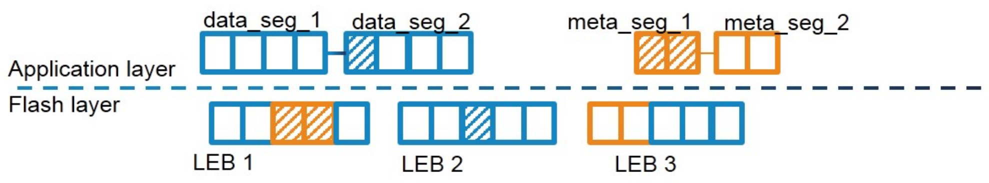
Separating data of different update frequencies in uppper log can help reduce the file system garbage collection overhead, and better organize the data layouts on the lower log if with multi-append point on the device side. This is achieved by maintaining more than one active logs in the file system and keeping more than one active physical segments at a time on flash, e.g. F2FS uses one log for user data and one for metadata. As an example shown in Figure 4.6, if the underlying hardware does not support multi-logs which is common in the current SSD products, user data and metadata will be intermixed in the device. Usually, metadata is invalidated more often than user data. Thus, the upper metadata log will be garbage collected sooner and more frequently. Either TRIM or overwriting the metadata blocks will cause invalidation of blocks on the device log (Figure 4.6(a)). However, if the file system supports advanced capabilities that further separates metadata to hot and cold one, which is even better for the upper logs' data arragements. Hot metadata log will be the one that has most frequent invalidations. As the total amount of file metadata remains the same, the separation of data to multiple upper logs will disperse data to more lower logs or segments, and hence the invalidations, as is shown in Figure 4.6(b). Thus, invalidation tends to be distributed to more lower logs or segments with smaller size of extents, resulting in high degree of fragmentation and higher device GC WA.

If there are m upper logs stacking on n lower logs, then the ratio of (m/n) can indicate the degree of data fragmentation to some extent. The higher the ratio is, it is more likely that a single underlying log may contain data from more different upper logs. With different log activeness, such as update, invalidation, and GC activities, the lower log will suffer from high degree of fragmentation, and hence higher WA as the system is aging.

We conducted an experiment of 2 upper logs vs. 6 upper logs using



(a) With fewer application logs, upper layer segment invalidation will be aggregated to fewer device segments.



(b) More application logs tends to distribute invalidation across more flash log segments, resulting in higher WA.

Figure 4.6: Different log activeness results in high degree of fragmentation.

F2FS. Under the same workload of 60GB random writes, 4k IO size and direct IO, the file system writes issued to the device is much closer (see column 3 - fsys W GB in Table 4.1). However, due to the different distributions of hot and cold data within the same device log and the decreased size of invalidated extents, 6-log case gets higher fragmentation, and hence much higher devce GC WA than 2-log case. As more valid data is copied forward during GC, it has to reclaim more segments at once to free same amount of space, as a result, more erase counts in more logs case. (see column 5 and 7 in Table 4.1). If the file system has more hot files on a smaller portion, more logs gets worse as the invalidation of data and metadata blocks will be spread to more device segments with smaller extent size, as is shown in Table 4.1 zipf0.8 vs. zipf1.1.

Though in the 6-log case, the increased log metadata also contributes to more device writes and higher GC (See section 4.2.1), in this buffered I/O experiments, we can still separate their effect in a coarse manner by assuming the resulting device writes are from both original workload and increased log metadata in a linear scale. Thus, still the much higher WA and erase counts are mostly caused by the effect of data activities distributed across logs.

Generally, if upper logs have a diverse range of activeness, for example, some applications may be more active than others at a certain period of time, or some logs are much smaller than others, it increases the degree of data fragmentation presented on the lower logs and results in high device GC overhead.

random distrbution	log #	fsys W GB	device W GB	erase count	GC data copied GB	GC WA
zipf:0.8	2	120.55	221.02	370	98.28	1.82
zipf:0.8	6	121.08	267.88	473	144.58	2.19
zipf:1.1	2	122.05	222.78	374	98.52	1.81
zipf:1.1	6	122.53	277.10	493	152.35	2.24
uniform	2	96.32	137.94	188	39.96	1.41
uniform	6	96.42	141.25	195	43.15	1.45

Table 4.1: Device WA varies with different number of upper logs.

4.2.5 Decoupled segment cleaning

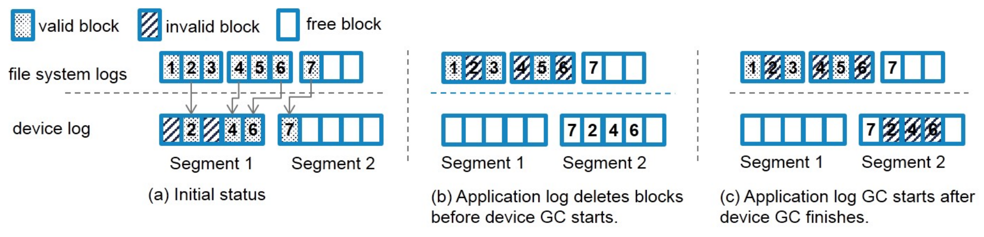
Coordinated segment cleaning is almost impossible when logs are stacked on each other. Garbage collection is a side effect of log structuring, where invalidated blocks within segments need to be reclaimed to generate free space [80]. By design, logs work in isolation (i.e., manage their free space themselves) and are oblivious of other logs underneath them, which results in an inefficient implementation of logs.

First, data that is presumed to be valid at the lower layer need not be valid at the upper layer if without TRIM support. Invalidations at the upper logs need not trickle down to the lower log, as a result, lower logs works with outdated validity information. Invalidations in the lower logs is normally inferred by overwrites performed by the upper logs. The lower-log layers performs segment cleaning operations, move data (invalid at the upper log but valid in its level) that would never be accessed but invalidated by the upper layer at some point in time. In the worst case, such data can be moved multiple times within the lower log before the upper log overwrites it due to re-allocation of logical block addresses (LBA).

Second, data could be moved multiple times across log layers due to uncoordinated segment cleaning. Each log layer performs segment cleaning while being agnostic of the activities in the other log layer. Considering the case where the segment cleaner of the lower log runs ahead of the segment cleaners of the upper logs segment. In this situation, the lower log could clean a segment that contains one or more segments from the upper logs. After the cleaning is done, the segment cleaner of the upper logs will move the data (as part of segment cleaning) and rewrite the segments in the lower log again. The process of uncoordinated segment cleaning creates more invalidation at the lower log's segment that was used to store data from upper logs. The unnecessary additional writes caused due to uncoordinated garbage collection increases the WA, and if running on a flash device it also decreases the device's lifetime.

Figure 4.7 depicts an example of decoupled cleaning without TRIM.

Initially, data are written sequentially to the device. At a certain point, file system deletes some data blocks without informing the underlying FTL (Block 2, 4 and 6 in this example). When the device garbage collector wakes up, those data are still valid from the lower log and device GC point of view, so it still copies them forward to reclaim device segment 1. This increases the device GC write amplification (WA) by copying invalid data seen by upper layer. As a result, invalidated blocks 2, 4 and 6 will stay in the flash media until file system GC cleans segments containing those LBAs and overwrite those LBAs with new data. Copying of these blocks by device GC could happen several times until the file system re-allocates those LBAs.



valid block
invalid block
free block
file system logs
device log
Segment 1
Segment 2
(a) Initial status
(b) Application log deletes blocks before device GC starts.
(c) Application log GC starts after device GC finishes.

Figure 4.7: Decoupled segment cleaning without TRIM

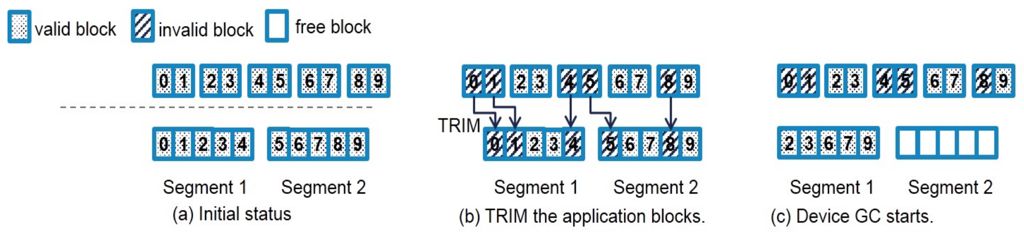


Figure 4.8: Decoupled segment cleaning with TRIM.

Third, high degree of data fragmentation due to different log characteristics and activities (discussed in Section 4.2.3) increases device GC WA, and meanwhile garbage collection on both layers further fragments the data layouts. While file system makes effort to write sequentially on its own segment basis, even with the effort of invalidating and overwriting sequetially, holes could be made at the layer with larger segment size due to the unmatched segment size or page size as well as the timing issue for both layers' GC. Without TRIM, as an example shown in Figure 4.7, when file sytem GC invokes after device GC has copied block 2, 4 and 6, the overwrite operation sent to the device will invalidate those blocks, and causes fragmentation on the media. As a result, when device GC wakes up again, valid data (Block 7 in this case) will be copied forward. This further increases the WA due to the mixed placement of valid data and invalid data. Even with TRIM, as is shown in Figure 4.8, fragmentation problem still exists. When file sytem TRIMs the entire segment (Figure 4.8), the underlying device segments only invalidate a portion of them. Moreover, if file system GC happens after device GC, those valid blocks will be copied at least twice. In addition, device GC process usually involves several stages including segment scan, victim segment selection, and valid data re-read and re-write. Not to block the incoming requests in a high performance system, this is multi-threaded and alternate with new IO requests. Data written by GC and by new write operations will be mixed, and further enhances the degree of fragmentation.

Fourth, many of the optimizations in segment cleaning algorithms are no longer applicable when logs are stacked on top of each other. For example, grooming based on hold cold segments needs not hold true at lower log layers. As mentioned earlier, the notion of hold/cold data at the lower log need not be accurate and representative at the higher layer. Reasons include interspersed data from segments across (and within) logs, segment cleaning at an upper log layer translates to "hot" data to the lower log layer, segment cleaning at an lower log moves "cold" data from the upper log (which could have been invalidated) making it hot on a overwrite, and assumptions about availability of multiple append points.

4.3 Experimental evaluation

In this section, we quantify the problems discussed in previous sections with our log-on-log simulator. The log-on-log simulator presents an ideal condition which keeps all metadata in memory. Our focus is on measuring the write amplification factor in both log layers, and their combined WA. Since the cost of GC is one of the dominant factors that affect both performance and device endruance, we focus on exploring solutions to improve performance and endurance by reducing GC cost. As both the single log and multi-log experiments show the deficiency of stacking logs in an uncoordinated configurations in the ideal environment, in real system, the condition will get worse.

We picked several traces from Microsoft Research [5] and several TPC-E workload traces generated from database runs [11] to run through our simulator. Though these are block level traces, as our two-layer log simulator is independent of platforms and system configurations, it could be abstracted as any two-layer modules reside in the entire system hierarchy. In this file system to device case, we mimic them to be requests sent to the higher level log. The LBAs in the trace are treated similar to virtual addresses in the VFS layer, and be translated to the logical addresses by upper layer log, then be passed to the lower log through a block-like interface, and then be converted to physical addresses by flash FTL.

4.3.1 Single log

Generally, each log needs to reserve a portion of its free space for reclamation process. Logical addresses in upper log can only be reused after the upper layer GC reclaims that address space which is in a segment unit in this case, thus lower layer log only gets "update" operations on the same logcial address after the upper layer GC runs for a while. The actual total usable logical space (or total unique logical addresses that are used) in the upper log could not exceed the physical size of the lower log. In other words, upper log size is always smaller than the lower logs. For example, if the physical device is 1TB, it reserves 20the upper layer also reserved 20exposed to the user or application is only 640GB then.

The higher the size ratio of upper vs. lower log is, the higher utilization the lower log will be, and hence the more aggresive lower log GC will be. In this experiment, we fix the upper log size and conduct the uppper/lower log size ratio from 40% to 90% by changing the lower log size. Each layer has 10% reserved space for GC and uses *greedy* policy to choose victim segment. We vary each layers segment size from 2MB upto 512MB so that we can get a variety of segment size ratios between two logs.

We measure the upper log write amplification as the ratio of total upper layer writes to the original writes in the trace (FSWA) and the lower log write amplification which is actually caused by the flash translation layer's GC activity as the ratio of total lower layer writes to the upper layer writes (FLWA). Similar to the TCWA in Chapter 3, the total combined write amplification represents the effect of FSWA and FLWA.

4.3.1.1 Capacity ratio

Usually, the physical and logical (virtual) space each layer log exposes is different, this depends on system design, configurations, and hardware characteristics. And hence the capacity ratio of upper to lower log varies. If the capacity ratio is high which means the size of each layer is close to each other, then the lower log will be running out of space soon after the upper log is. That means, both layers GC starts within a short period of time. In this case, it has high chance that a lot of logical addresses have not be reused or reclaimed by upper log, and thus less data invalidation seen from the lower log. When lower GC starts, most of the pages in the victim segment are still alive, more data being copied forward resulting in higher GC WA. In other words, lower log GC will be more aggressvie to reclaim a certain amount of free sapce. In such systems, usually FLWA is much higher than FSWA, and dominant the combined TCWA. Reducing FLWA is more critical to the overall performance.

On the other hand, if the capacity ratio is lower, which means upper log GC starts much earlier than the lower log GC. It allows more invalidation in the lower layer, and hence reduces the FLWA. This tradeoff the FSWA for lower

FLWA, and hence the TCWA would also be reduced.

Table 4.2 shows an example of our experiments with tpce trace. We fixed the lower physical log size and vary the upper log size. Under the same workload, when the size of upper log is larger and the capacity ratio is relatively higher, we see a lower FSWA and less amount of data be written to the device. However, this increases the FSWA by copying more valid pages during device GC. When lower GC is dominant in determining TCWA - when lower segment size is larger than 8MB in this case, less writes from the upper layer results in higher total device writes. On the other hand, if we configure the upper log to be smaller (up/low size ratio of 0.8), though this makes file system GC more aggresive and issuing more writes to the device, those increased writes contribute to update/rewrite operations on the same logical addresses, and hence more invalidation occurs within each segment to favor lower FLWA. When the order of FLWA reduction surpasses that of the FSWA rise, the combined overhead will be decreased.

Figure 4.9 depicts the results in another dimension where lower log size varies under a fixed-sized upper log. In such conditions, higher capacity ratio presents more physical space in the lower log. Lower log GC can be less active, and hence the FSWA dominant the TCWA. In such case, choosing a segment size that achieves lower FSWA will be more effective to help improve the overall performance. We will discussed the segment size ratio in details in the following section.

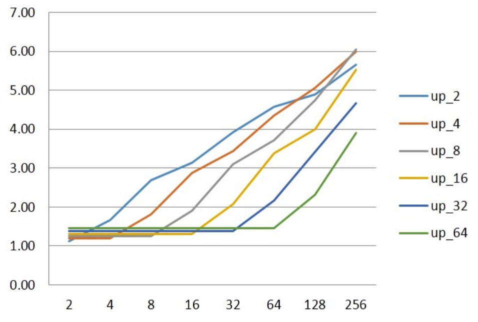
up/low s ratio	low seg MB	orig w GB	up GC W GB	FSWA	low GC W GB	FLWA	TCWA	erase count
0.9	2	88.96	32.63	1.37	0	1.00	1.37	46693
0.9	4	88.96	32.63	1.37	72.38	1.60	2.18	41877
0.9	8	88.96	32.63	1.37	175.00	2.44	3.33	34073
0.9	16	88.96	32.63	1.37	250.26	3.06	4.18	21854
0.9	32	88.96	32.63	1.37	339.46	3.79	5.18	13782
0.9	64	88.96	32.63	1.37	404.02	4.32	5.91	7924
0.9	128	88.96	32.63	1.37	442.03	4.64	6.34	4266
0.9	256	88.96	32.63	1.37	494.39	5.07	6.92	2343
0.8	2	88.96	45.86	1.52	0	1.00	1.52	53468
0.8	4	88.96	45.86	1.52	61.93	1.46	2.21	42587
0.8	8	88.96	45.86	1.52	102.08	1.76	2.66	26434
0.8	16	88.96	45.86	1.52	167.25	2.24	3.40	17388
0.8	32	88.96	45.86	1.52	216.08	2.60	3.94	10257
0.8	64	88.96	45.86	1.52	244.54	2.81	4.26	5584
0.8	128	88.96	45.86	1.52	261.66	2.94	4.46	2929
0.8	256	88.96	45.86	1.52	281.62	3.09	4.68	1545

Table 4.2: TPCE: Lower capacity ratio provides device GC more invalidated pages. TCWA shows reduction with more aggressive upper log GC under same lower log physical size.

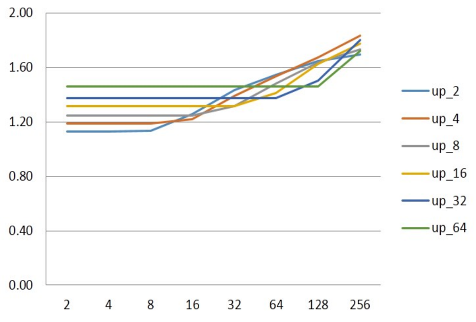
4.3.1.2 Segment size ratio

Another factor that affects both layers WA is the segment size - which is a smallest unit during the GC process. Generally in a single log case, smaller segment size can provide more flesibility to the GC selection algorithm as it arranges the storage space in a fine-grained manner so that the selection mechanism can find a segment with fewer valid pages. Thus, the smaller segment size, the lower WA. Usually due to the metadata overhead of maintaining segments and some hardware limitations, segment size is designed in an order of MB to achieve balance between GC efficiency and maintenance overhead.

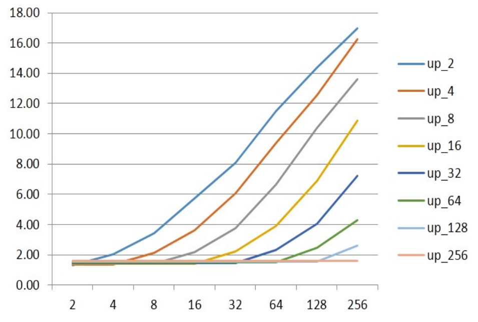
When one log is stacking on another, simply choosing a smaller seg-



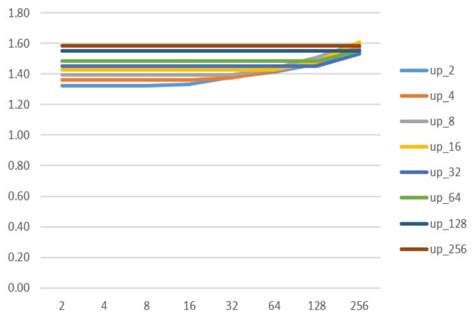
(a) prn_0: upper/lower size ratio 90%.



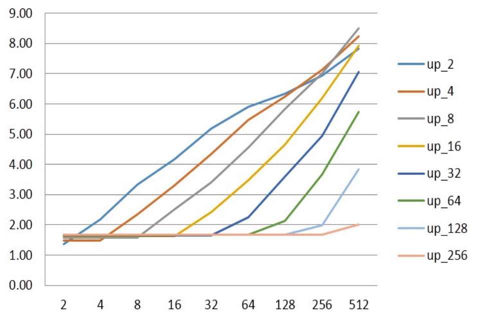
(b) prn_0: upper/lower size ratio 70%



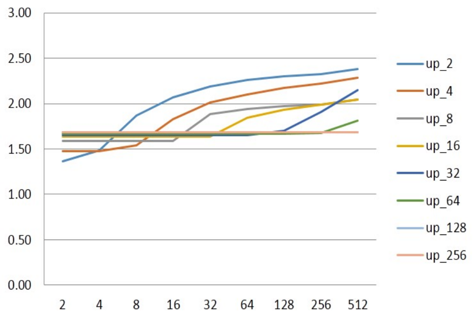
(c) prn_1: upper/lower size ratio 90%.



(d) prn_1: upper/lower size ratio 70%



(e) tpce: upper/lower size ratio 90%



(f) tpce: upper/lower size ratio 70%

Figure 4.9: Single log experiments varying the capacity ratio and segment size.

ment size to favor lower WA in its own log sometimes is harmful to other layers GC performance. We have discussed the problem of unaligned segment size in Section 4.2.3. Here we further explore the impact of different segment size ratio on the TCWA. Figure 4.9 shows the TCWA of three of the traces: prn_0, prn_1, and tpce (See detailed trace characteristics in Table 3.2). prn_0 is write intensive, prn_1 and tpce is read intensive workload. We fix the upper log size while configuring the lower log size to make the up/low size ratio from 40% to 90%. For each size ratio, we varying both layers segment size and measure the FSWA, FLWA and TCWA.

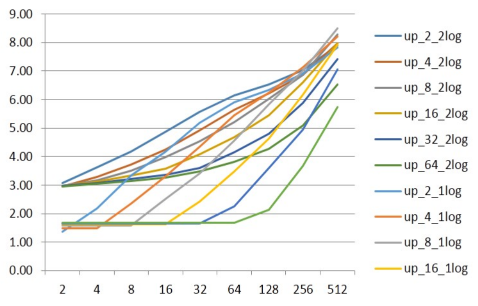
For each segment size configure, the TCWA starts to increases dramatically once the lower segment size exceeds the upper segment size. This is because upper GC reclaims space in a unit of upper segment size, at the time these logical addresses are reused - when same upper segments are written again, this is seen as invalidation by the lower log. If these invalidation can cover the entire lower segment at once, then the whole lower segment can be reclaimed with no data copied forward, and as a result FLWA is 1. If these invalidations only cover a part of the lower segment, the more random upper GC chooses victim segments, the higher degree of fragmentation lower log will suffer. This becomes worse as the up/low segment size ratio decreases. On the other hand, if upper segment size is larger, re-writing one upper segment can always conver at least one lower segment, no matter whether the size is a multiple of lower segment size. In this case, lower GC can always find segment without or with fewer valid pages.

While smaller upper segment size can help reduce FSWA and hence issue fewer writes to the device, it gets lower TCWA when device log segment size is small. Once device segment size gets larger and FLWA becomes dominant, the increased FLWA due to the segment fragmentation will make TCWA even higher. As is shown in Figre 4.9(a)(c)(e), the very left part shows smallest upper segment size behaves best, however, the increasing ratio of them is also larger. Once beyond a certain point, larger upper segment size gets lower TCWA regard less of the size of the lower segment size.

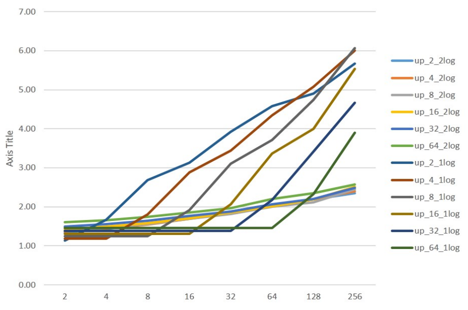
In addition, if the capacity ratio is lower that device GC does not have to work aggresively and delays its wake up time to allow more invalidations, smaller upper segment size gets lower TCWA as long as the FSWA is dominant. For example, Figure 4.9(c)(d) shows the TCWA of prn_1 with 90% and 70% capacity ratio. Though they both show the same trend at the macrolevel, the turnning point in the 70% case is closer to a larger device segment size, which means, with larger physical device, smaller upper segment size is better if the lower segment size is not too big (e.g. in an order of hundred MB). That is said, the choice of each layers segmeng size to achieve smallest overhead depends on the segment size ratio and the log capacity ratio.

4.3.2 Multiple logs

In order to explore the impact of multiple logs, we configure our simulator to have 2 upper logs and 1 lower log to mimic a system having more file system logs on top of a device with fewer append point (e.g. F2FS on a commercially available SSD). We use the same traces as above, each upper log has their own address ranges so that incoming requests are redirected to different logs based on the request's address. Once a log is running out of space, the GC starts reclaiming space internally. To simplify the problem, we configure each upper log having same segment size, same GC threshold, and same GC selection algorithms. As each log has different capacity and activeness, the workload seen by the lower log will be random and unpredictable.



(a) tpce: TCWA gets higher with 2 upper logs.



(b) prn_0: TCWA is reduced with 2 upper logs

Figure 4.10: Directing data to two separated logs can reduce WA in its own layer, but potentially increases WA in the others, and results in higher TCWA under some workloads or configurations.

Figure 4.10 presents results of tpce and prn_0 under one configuration, both with a 90% up/low capacity ratio. We put the single-log case and 2-log case together for easy comparison. For prn_0, FSWA is reduced with two upper logs in all segment size configurations, this is in line with the purpose of mult-log design. While it generates fewer writes to the device, and makes the device GC less active meanwhile, TCWA is reduced. As device segment size increases, the inceasing ratio of TCWA is also lower than that in the single log case. This is an optimal condiftion for multiple logs. On the other hand, in the tpce trace where workload is random and with no distinct difference of access frequencies on the address ranges, for example, in the workload that data and metadata are usually updated together, separating data based on types does not help reduce WA. Figure 4.10(a) shows a higher TCWA in the 2-log configuration especially when device segment size is smaller. Though the incrasing raio of TCWA is lower in 2-log case, in most of the case, single log configuration works better.

While the workload is hard to predict in the real world, separating data based on types is not always beneficial. This rises more difficulties to the file system or application designers. With underlying hardware layouts unknown, they need to be more cautious on directing data to different logs. We intend to further explore the log coordinations between logs within and across multiple layers.

4.4 Methods for log coordination

We have characterized the interactions between multiple levels of independent logs, and describe several practical scenarios which arises in real systems. Stacking logs on each other defeats the benefit of original logs due to lack of log coordinations. In the system where log structure can still show its superior performance and reliability, we recommend several design choices to reduce the overhead of multiple levels of logs.

4.4.1 Size coordination

In a typical log-on-log system, two size-related configurations will affect the overall system performance, and each of them are not independant on its own layer, which is quite different from a single log architecture. We have analyzed the impact of different capacity and segment size ratios on the overall TCWA in the previous section. We will summarize the design suggestions regarding size coordination here.

Capacity: Larger capacity is not always beneficial. For larger upper log, it provides more virtual space, delays the file system GC start time, and lowers the upper GC activeness and hence lower FSWA. As a result, file system writes less data to the device. However, as device log gets less invalidation from writes, more valid pages will be in the victim segments when device GC is activated, and hence higher FLWA. Moving more physical data forward is harmful to performance.

Generally, the cost of moving virtual/logical pages around in file system or other application logs is lower than that of moving physical pages in the device log. In addition, the device endurance is directly related to the total writes to the flash which is a factor of TCWA (FSWA * FLWA). While reducing both layers WA is not easy to achieve, slightly increasing the upper layer FSWA by making upper log GC working more aggressively to tradeoff a lower FLWA could be a better choice. With a fixed sized physical device, configuring a smaller size of upper log can help reduce the total writes to the device as long as it can hold the workload. In other words, smaller up/low log capacity ratio improves the overall performance. This also stands for the condition when upper log size is fixed, smaller up/low log capacity ratio means a larger physical device, which can always gets lower FLWA, and hence reduced TCWA.

Segment Size: The common sense that smaller segment size can always gets lower WA is no longer true in the log-on-log scenario. We have shown the segment size impact on the individual logs as well as the entire log-on-log. When TCWA is determined by the combination of FSWA and FLWA, we should be very carful when choosing each layers segment size.

Generally, in the system that upper log is not the only consumer of the underlying physical log and other upper level applications are less active, or the lower log has more spare capacity so that device GC is less aggresive than file system GC, it is better to choose an upper segment size that can achieve lowest FSWA without bringing too much metadata overhead, such as segment metadata, and etc. On the other hand, when both layers have heavy workload and the capacity is close to each other, it is better to have the upper segment size match or larger than the lower segment size. Though larger upper segment size run the risk of increasing FSWA, the large range of invalidation it passes to the device can help reduce device GC WA dramatically.

File system designers do not know the underlying hardware layouts, and vice versa. Besides, due to the hardware limitations, normally it is hard to change the device segment size after it leaves the factory. The best way to do segment coordination is to configure the file system or application layers segment size during mounting. By getting lower segment size information, file system can configure its own layer's segment size based on the capacity ratio and lower segment size. The advantange of doing this is pretty straightforward and only needs to be done at the startup time with no overhead.

4.4.2 Virtual log defragger

Recently, several log-structured file systems are designed as flash-aware and aim to optimize SSD performance. However, the underlying log (FTL) is usually lacking information about the upper layer log, e.g. the block and segment sizes, the number of upper logs, etc. Passing these information through a standard interface is not easy to achieve in many systems. In addition, even if the FTL is able to support multiple number of logs (multi-append points) with hardware support, the number of upper layer virtual logs tends to grow faster than that of the underlying physical logs. One of the main reasons for this is due to the fact that the upper layers assume that they manage their own logs and does not propagate or inform invalidations at its layer to the FTLs log layer. What we really need is a separate log at the FTL layer to store/append data from the upper log. This would provide isolation for the individual upper logs and prevents intermixing of upper log-data, which will enable the upper logs to retain all the good properties of maintain a log.

Unfortunately, it is easy to increase the number of application logs, but is impossible for the FTL to support an arbitrary number of active append points all the time due to hardware restrictions to match the number of upper logs. Thus, the data layouts on the physical device usually present a quite different view than that on the upper layer virtual logs. The mismatch in the number of logs in the upper and lower layer (especially when the number of logs are greater in the upper layer) defeats the benefit of maintaining logs at the upper layer, and potentially increases write-amplification during garbage collection at the SSDs FTL (or lower) layer. To solve this problem, we propose a Virtual Log Defragger that can be invoked at system idle time. The main idea can be described in Figure 4.11.

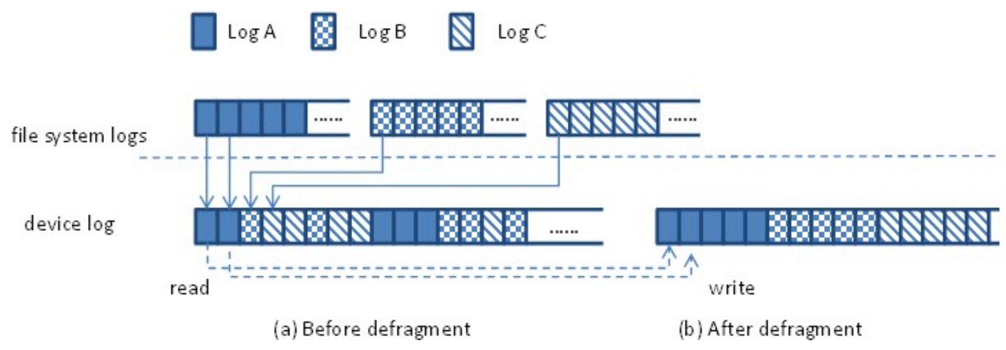


Figure 4.11: Virtual Log Defragger.

The defragger (we refer to Virtual Log Defragger in the rest of this section) works for any m-to-n logs configuraitons. We show an example of 3-to-1 case in Figure 4.11. Before defragment, upper logs write sequentially within their own log, and the lower log receives intermixed requests from the upper layer concurrently. Most of the time the order each of them writes to the physical device cannot be predicted, and the requests size vary. This makes it difficult to re-organize data from different upper logs on-line without blocking other incoming requests. While there is only a single log (or append point) in the lower log, it is also not possible to separating data from different logs during device garbage collection. To solve this problem, we propose a Virtual Log Defragger at system idle time.

Defragger wakes up when system is idle, or when a file system log or application log is idle, or a defragger request is issued to a specific upper log. After selecting the target log, the upper layer reads the entire log and write pages back to the device. By doing so, virtual/logcial attributes of data blocks are not changed, while blocks from same virtual layer will be physically contiguous on the device. The selection of upper log to be defragged can be based on several factors, for example, relatively static logs could be a good candidate.

The defragger works without restrictions on the number of underlying logs. It makes up for a lack of multi-append points at the device level, or alleviate the deficiency when file system has more logs than FTL does. It also helps reduce metadata overhead for FTL mapping by grouping data from the same upper log and merging adjacent blocks, e.g. index entries can be reduced with several small pieces of data chunks are integrated by the extent-based indexing. Moreover, the GC performance can be improved if data of similar update frequencies from the same log have already be grouped in the device segment. Besides, aligning segment size for each layer is no longer critical. If the defragger can detect the activeness of each upper log or even each segment, it can arrange the defragmentation order and group multiple upper segments having similar update frequencies to the same device segment.

4.5 Future work

We focus on single log case in our experiments. The capacity ratio as well as the segment size ratio helps reduce overall TCWA when a single log stacks on another log. In a system with multiple logs in each layer, or even with multiple instances of log-structure, things will be more complicated. We intend to explore the impacts of multiple logs with different configurations and activeness.

We propose a virtual log defragger to group data from same upper log during system idle time. The behavior of the defragger is not fully undertood yet, nor its possible collaboration with garbage collection process. Several things remain undetermined, for example, the best time to start the defragger, the interface lower layer exposes to the defragger to informs the status of device GC, and etc. We will further investigate its characteristics in our future work.

4.6 Summary

As flash memory has its own log-like mapping mechanism, a log-structured system on top of a NAND flash-based device will causes performance degradation and decreased device endurance due to duplicated loggings and uncoordinated activities on each log layer. In this chapter, we characterize the interactions between multiple levels of independent logs, and describe several practical scenarios which arises in real systems. We show that log on log can result in highly counterintuitive behaviors. While applications and file systems continue to use log-structure for their own performance and reliability purposes, we recommend several design choices to minimize the side effect that multiple layers logs bring to the entire system due to lack of coordinations. Through our real system and simulation experiments, we suggest a better choice of capacity ratio and segment size ratio. In addition, we also propose a virtual log defragger at system idle time. The benefit and ovehead for the defragger will be further investigated in our future work.

Chapter 5

Log-less Flash-aware Storage Systems

5.1 Collapsing logs

We have discussed the problem of a log-structured file system or applications on top of a log-like FTL flash device in the previous chapter. Though log-structured system can achieve higher performance and better system reliability, and log-aware optimaztion can help coordinate activities between different layers, stacking logs on logs is not always a good idea. In real systems, the structure could be more complicated that makes things even worse. For example, a log-structured application stacks on a log-structured file system, and with a log-structured flash device underneath. Light-weighted log coordination between any two layers may result in negative impact on performance of other layers, making the overall system behavior hard to control, and hence unpredictable.

There are several classes of alternatives to collapse file system level logs by utilizing the nature of flash memory and device embedded FTL. In this chapter, we will discuss the design of two log-less flash-aware system - DirectFS and object-based system. We focus on object-based flash system, its performance, portability, feasibility, and show from simulation experiments that such system can provide light-weighted advanced features to improve overall performance as well as reliability with no interference to other system layers.

5.1.1 DirectFS using sparse space

One approach to breaking logs is to utilize the rich 64-bit address space in modern computer systems to do direct mapping, and leverages the FTL mapping mechanism. DirectFS is an extension of the ideas presented in[42]. It leverages the underlying FTL log structure via four primitives. A sparse address space represented by the FTL enables DirectFS to eliminate its own mappings of ¡file, offset¿ to physical block. Rather, directFS maps each file to a small number of virtual extents, and relies upon the FTL to allocate physical blocks to these virtual addresses as blocks are consumed. Different from conventional file block allocation in a typical log-structured system, DirectFS offloads these capabilities to the underlying FTL, as FTL by nature is responsible of doing mapping between logical to physical addresses.

In place of a journal, directFS uses two primitives, atomic writes and persistent TRIMs [70] provided by the underlying FTL. The atomic writes are executed entirely or not at all, and persistent TRIMs (also atomic) provide transactional deletes of virtual address ranges. By using these in combination as a group of transactional updates and deletes, DirectFS can move from transactionally consistent state to transactionally consistent state without an independent journal. Finally, directFS uses statistics exported by the FTL on allocated block counts to maintain accurate counts of physical space consumption, which limits updates of superblocks when files are extended. DirectFS also exports the same primitives to user space applications as capabilities of DirectFS files.

Without the overhead to maintain a log, DirectFS outperforms the conventional log-structured file system on a flash-base storage media. e.g. Nilfs, F2FS. In addition, with large virtual address space, more add-on could be introduced that performs the same capability such as atomic transaction in a journaling file system, but with less overhead. The performance and advanced features of DirectFS is beyond this dissertation's scope, and will be discussed in the future.

5.1.2 Object-based flash-aware system

Different from DirectFS, another approach to collapsing logs in the file system layer is to utilize an object-based interface, so that file system semantics could be passed to the underlying device to let the storage device deal with all the complicated jobs such as data placement, indexing, garbage collection, and etc. Meanwhile file system could be very simple and portable.

The design of an object-based storage system has been discussed many years ago [17, 29], however its use is not widely popular in the world of disk-based storage system since disk does not require complicated data remapping, which makes the intelligent object-based storage device (osd) lose its attraction. Not until the emerging technology of non-volatile memory, has object-based storage started to attract attention in the new class of storage medium, not only in the large distributed systems, but in a single node host machine as well.

In an object-based design, file system only takes care of name resolution and offloads the storage management layer to the object-based device. As the storage management layer is device specific that has to serve the characteristics and limitations of different hardware, for example, flash memory's out-of-place update, and limited endurance of different NVRAM devices, maintaining such mechanism in the device layer would be more beneficial. Meanwhile, hardware layout is transparent to the object-based file system so that regardless of the types of the underlying NVRAM, the upper layer could be generic.

On the other hand, since osd gets more information from upper layer through the rich object-based interface, a lot of advanced features could be introduced, while it could not be easily achieved in a block-based interface. For example, separating data of different types (data and metadata) and redirecting them to different active segments [48] can help better organize data layouts. Previously many efforts have been done to redirect data of various access frequencies to different append point so that garbage collector can choose a victim segment with less amount of valid data. However the performance improvement is usually suboptimal due to the lack of file system information passed through a standard block interface, as it is hard to tell from the LBAs the types of data. Maintaining a buffer to store LBA information such as touch counts in the device layer is even more infeasible.

Other features, such as object-level reliability and object-level compression and encryption can be achieved on the osd device layer without consuming a lot of host side resources [48]. In this dissertation, we provide a design choice of object-aware data placement that aggregates data based on object size and remaps them to different segments. We show from simulation results that such method is easy to approach with little overhead, but can provide great benefit on garbage collection efficiency and more balanced wear-leveling.

Typically, the objects in an object-based system could be divided into two categories, regular user data objects, and file system metadata objects. osd does not differentiate them, nor through the standard object interface. So both user data objects and file system metadata objects are treated as regular objects by osd. On the other hand, osd creates its own object metadata, such as indexing entries, object size, and etc. To simplify the structure, we keep all osd created metadata in memory in our simulator.

5.1.2.1 Object-aware data grouping

A lot of research has been done to explore the relationship between data file sizes and their access patterns [28, 13]. It is said that though most of the data accessed are in large files, "the majority of file accesses are to small files" (e.g.about 80% of the accesses are on files smaller than 10KB). However, the implementation of these approaches to grouping smaller files or objects usually end up less efficient, and sometimes results in negative impact on performance due to incorrect grouping. Even if file system groups data optimally, it is hard to pass these information through a standard block interface. When flash memory is used as a primary storage media, data is remapped to different physial locations by FTL and will be moved multiple times during garbage collection, which results in quite different views from two layers that makes the grouping of files inefficient. If we maintain an additional layer of data access information in the device, this will cause problems similar to log-on-log structure as we discussed in the previous chapter. Thus, flash-based storage system needs a more intelligent but light-weighted design.

An object-based storage system meets these requirements. File system does not have to deal with complicated data management like object grouping. Meanwhile, it can utilize the sparse address space to do easy name resolution. Data managment such as grouping and placing are maintained by osd alone, so there is only one consistent view of virtual to physical data in the entire system. Thus, osd can have more flexibility on managing the objects based on their characteristics, such as access frequencies, object size, request size, so on and so forth.

In our design, we use a simple scenario by grouping objects based on their sizes. As is said that a large portion of data accesses are on files smaller than 10KB [13], most of such files or objects could be well fit into a single physical flash page in most of today's NAND flash device (4KB or 8KB). Once be read or written, the entire objects could be retrieved through a single I/O. When object-based flash device supports multiple logs (or multiple append points), osd could

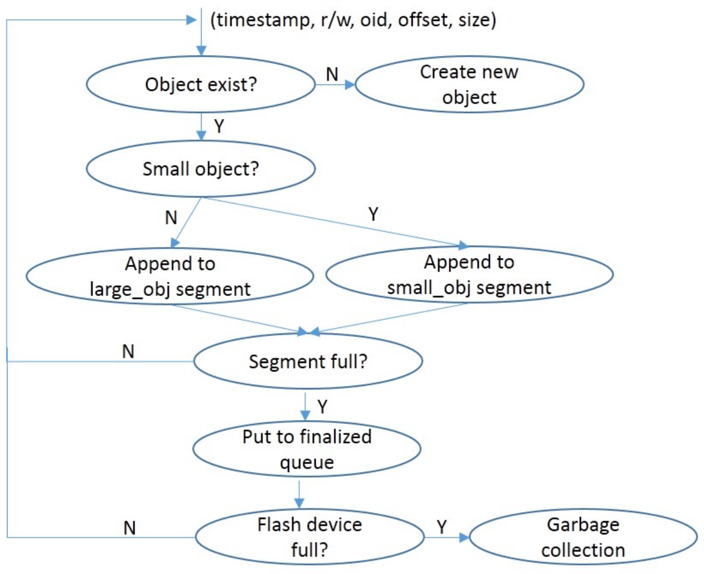


Figure 5.1: The object-aware data placement framwork.

group these small objects together into one log, and redirect objects larger than 10KB to the other log. As small objects are acceesed more frequently, segments in the small objects log are more like to be invalidated together so that GC overhead could be reduced dramatically. On the other hand, larger objects are more like to be accessed sequentially but only once in a certain long period of time (e.g. backup process), segments in the larger objects' log can remain relatively stable. This approach could not be easily achieved through block interface as the underlying device has no idea of which LBAs belong to which files or objects. Through object interface, grouping data is simple, flexible, and straightforward.

Based on these observation, we trigger the osd to maintain two logs, one for objects smaller than a flash page, and one for objects larger than a flash page. The framework of object-aware data placement is shown in Figure 5.1. In some flash device that supports two logs, the incoming data is treated to be hot and put into the "hot" log, while the data moved by garbage collector is assumed to be relatively static and be appended to the other "cold" log. We will evaluate the performance of these methods on an object-based osd in the following section.

5.1.2.2 Object-aware features

As object-based interface provides much feasibility and flexibility on data management, a lot of other features could be explored without consuming too much host side resource. Meanwhile, the change could be done only on the device layer, without interfering the host side file system design which provides better portability and compatibility. Several examples are listed as follows:

Object-aware reliability: Current design of data reliability relies on per-page basis error detection and correction. Such mechanisms are limited on the ability to correct a certain number of bit errors, but cannot deal with errors such as misdirected writes where pages are stored at the wrong place with no failure return message [10]. By offloading data management layer to the device which is responsible for the actual physical location of the data, such failures could be detected and corrected through objec-level reliability. In addition, different level of reliability could be achieved on different objects, for example, higher reliability on objects' metadata. In addition, failures that occur in the middle of a transaction could also be detected on the affected objects. The overhead of the additional object-level reliability could be mantained at a certain portion to the total metadata overhead if with well design of indexing and buffering [48].

Object-aware eviction and admission: When object-based flash is used as a caching device, it is possible to detect the hotness on an object base, for example, touch count per object. Such engine could help decide object-level eviction and destaging for both write-back and write-through caches when device is full. Similar to the components discussed in Chapter 3 that using LRU combined with advanced GC algorithms, improved cache hit rate with reduced GC overhead could be achieved if the cache has more knowledge of objects. This could outperform a sector-level cache eviction or destaging by better knowing the temporal and spacial locality of data access. In addition, object-level data prefetching in a tiered storage is also possible. Data belonging to one object could be prefetched together from disk to flash cache and to the main memory. This could help fully utilize the device bandwidth by obtaining an entire object, and meanwhile improve memory or cache efficiency without reading ahead useless data.

5.2 Simulator and analysis methodology

In this section, we describe the traces used in our analysis, the tools we built to characterize traces, and the simulator we built to evaluate our object-aware data placement and garbage collection performance.

Since currently there is no real world object-level trace available for our analysis, we choose several block-level traces available in the public domain and convert them into object-level traces synthetically. Our trace converter works similar to a simplified object-based file system, that by receiving a read/write requests, it treats the original block level requests as system calls from VFS layer, and converts the original block read/write operations into object-level requests. The converter internally maintains a one-one mapping table that translates each LBA into a $< oid, offset >$ pair. If the incoming request's LBA does not yet exist in the table, the converter assigns a new object ID (oid) with offset 0 to the new request. Otherwise, it retrieves the corresponding objects's id and starting offset, and convert the request to an object-level request $< r/w, oid, offset, size >$. In both cases, the requests' size remain unchanged. In addition, it does not change the original trace's characteristics, such as read/write ratio, and total unique data. The main components of the trace converter is depicted in Figure 5.2.

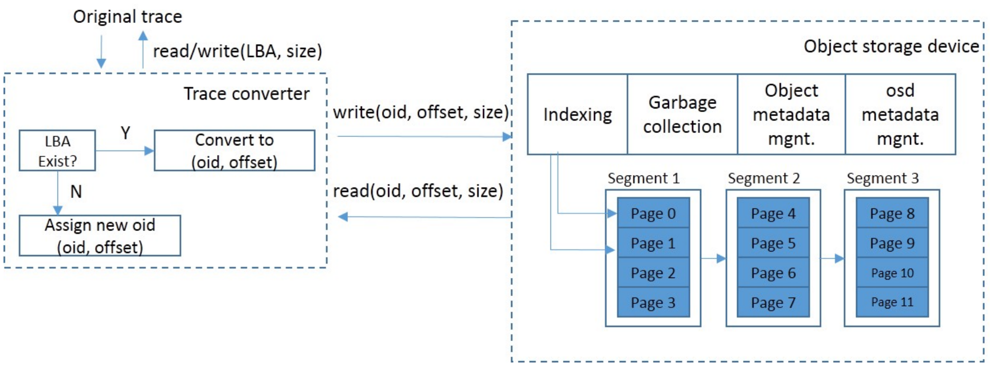


Figure 5.2: Compenents of object-based flash device.

While the object-level requests are sent to the object storage device (osd) through an object interface, the osd will do object data placement, object metadata maintenance, and etc. Same as an object-based system design that file system only does name resolution and let the OSD finishes the rest of the tasks, our OSD simulator manages the objects' physical locations, osd level metadata, device garbage collection, and etc. Internally, osd uses a log-structure to append new data to the end of the physical log, data mapping is in a unit of sector (512 bytes or 4096 bytes) while erasing is in a unit of segment. For each read/write request passed from the upper layer, osd looks up its mapping table, and finds the old location of the object if it exists. Otherwise, it will treat the operation as a create() request. By doing this, we have a variety of request types including object create/read/write, and the write requests will do overwrite operations or extend the existing object's size based on its starting offset and size.

In addition, our osd simulator can maintain 1 or 2 active segments at a time, which is similar to the concept of two active logs or append points. With one log, osd appends objects to the tail as a conventional log-structured FTL device. By activating two logs, osd has more flexibility on placing data to either log, depending on the pre-defined object placement policies. For example, appending new requests' data to one log and garbage collection data to the other, or aggregating small objects to one log while large objects to the other. We will discuss the performance of different scenarios in the following section.

In our following experiments, we pick two block traces prn_0 and prn_1 in Table 3.2 for our analysis of object-aware data placement. We first run these traces through the trace converter, and output object level traces in a format of (timestamp, read/write, oid, offset, size). We then run through the converted traces against our osd simulator, and output the statistics. The characteristics of two converted traces are shown in Table 5.1. Trace1 is write-intensive and trace

2 is read-intensive, both with a lot of read/write/create/update operations.

5.3 Experimental evaluation

We have discussed the effect of separating data and redirect them to different logs in the previous chapter and showed the importance of grouping data correctly. In this section, we evaluate the garbage collection overhead on an object storage device.

5.3.1 Experimental setup

Our simulation experiments run against semi-synthetic object traces that converted from block traces. The characteristics of two traces are shown in Table 5.1. Trace 1 is write-heavy and trace 2 is read-heavy. We configure the osd to be able to hold all unique data in the trace, meanwhile trigger enough GC activities for our performance and endurance analysis. For each test, we ran the traces for two loops, the first round is to warm up the osd device so that all data could be admitted to the flash, while the second round is to generate more activities. We show the efficiency of object-level data grouping in the osd layer, and its benefit of reducing GC overhead, and hence improved endurance.

There are three data placement modes that we are investigating. ***Single append point***: the conventional data placement strategy with a single append point. All dats is appended to the tail of the log, and osd only maintains one active segment at a time. ***Two append points***: osd supports two append points that incoming new data is appended to the active (hot) segment,

	Trace 1 (w)	Trace 2 (r)
total objects	307236	2048255
unique size GB	17.93	84.06
max object size MB	4.25	2.07
read GB	13.12	181.78
write GB	45.97	30.78
workload	write intensive	read intensive
configured flash size GB	21	95
segment size MB	21	48
# of segments	1024	2048
GC threshold	10	10

Table 5.1: Object-level trace characteristics and experimental configurations.

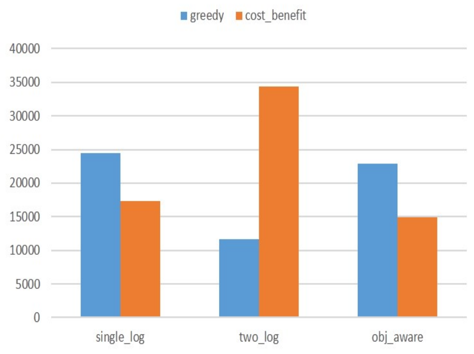
while valid data copied forward by garbage collector is appended to the inactive (cold) segment. This approach is popular in many SSDs that support two append points, but with no clue of file system information through a standard block interface, nor do they maintain additional data access statistics inside the devie layer. ***Object-aware grouping***: data requests belonging to small objects (smaller than 8KB in our configuration) are appended to hot segment, larger objects as well as garbage collection data are appended to cold segment. Finalized segments are put into one single list for GC selection. Once the free space is below the GC threshold, we trigger GC to select a victim segment based on either greedy or cost-benefit algorithms.

5.3.2 Performance evaluation

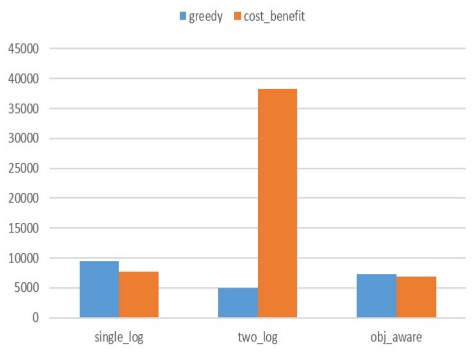
Figure 5.3 depicts the garbage collection efficiency of three data placement methods in terms of erase counts. It is shown from both workloads that greedy policy in two append points mode achieves lowest erase counts. This is because the relatively colder data during garbage collection are separated to the cold segment efficiently, so that GC algorithm was able to choose segment with least amount of valid data in the hot segments pool. However, this runs the risk of wearing out some of the segments sooner than others especially in the *two_log* mode as colder segments have less chance to be erased.

Such impact is shown when cost-benefit policy is used which tries to balance the hotness and age of the segments. As is shown in the Figure 5.3, in *two_log* mode where greedy policy performs best among three modes, cost-benefit suffers from much higher GC overhead than others. This happens because most of the data in the cold segments is relatively stable and lives longer in the device. When GC is not working aggressively in a fresh device, it is able to find hot segments with less valid data to reclaim. As device becomes full and aged, the need of reclaiming any free space hole gets higher, those cold and old segments with more valid data also need to be copied out to avoid potential retention errors. As a result, cost-benefit GC algorithms may move more data in the case of two_log mode and hence higher erase counts to reclaim a certain amount of free space.

However, endurance not only depends on erase counts, but also the distribution of erase counts throughout the entire device - here we use standard deviation of erase counts to measure the wear-leveling factor. As is shown in Figure 5.4, greedy policy often gets lower WA but higher erase counts deviation, while cost-benefit sacrifices the GC efficiency to some extent but provides improved erase counts balance in the single_log object placement mode. With object-aware grouping strategy, data could be better organized on the device.



(a) Write intensive workload - trace 1



(b) Read intensive workload - trace 2

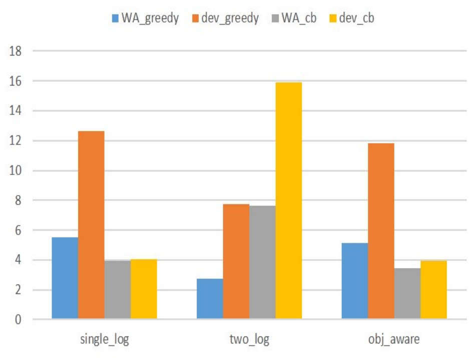
Figure 5.3: Erase counts varies different data placement strategies. Object aware data grouping achieves lower erase counts under both GC selection algorithms.

Under both read-heavy and write-heavy workload, object-aware data grouping achieves lower erase counts as well as lower WA. Though the erase counts is slightly higher than greedy policy in two_log case, the wear-leveling factor is the lowest among all three strategies. When cost-benefit algorithm is triggered, it provides better balance between GC efficiency and device overall endurance.

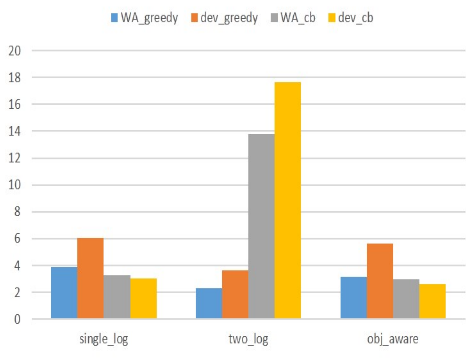
trace	# of logs	obj aware	GC policy	start validity dev	end validity dev
1 w heavy	2	No	cost_ ben	14271.81	4919.87
	2	Yes	cost_ ben	14363.74	2731.61
2 r heavy	2	No	cost_ ben	25830.18	5578.80
	2	Yes	cost_ ben	26275.24	3490.15

Table 5.2: Initial and final status of valid sectors distribution in different object placing modes.Higher validity deviation in cost-benefit GC algorithm does not always lead to lower WA.

We compare the distribution of valid sectors across segments, under two_log mode, with and without object-aware grouping, both of them use cost-



(a) Write intensive workload - trace 1



(b) Read intensive workload - trace 2

Figure 5.4: Write amplification factor and wear-leveling varies different data placement strategies. With object aware grouping, there is a better balance between lower WA and improved wear-leveling factor.

benefit GC selection algorithms. We show the initial status of such distribution when the device becomes full and before the first GC activity is triggered, and the final status when the entire test run finishes. As is shown in Table 5.2, the initial validity deviation for these two modes are very close under both workloads. However, as the device becomes more active, different data is being moved around due to the way they are grouped and placed on device, which results in a different view of validity distribution. Though under non-object aware two_log mode, the valid sectors deviation becomes higher, it does not provide a positive impact on the selection of victim segments. In contrast, it suffers from higher WA. One of the reasons is cost-benefit algorithm selects victim segment based on the number of valid data and the age of the segment. If data is not well grouped, for example, if segment is mixed with some very hot and relatively cold data, it is still treated as hot segment based on the latest time it was accessed by hot data. This provides some illusive information to the algorithm, and results in more valid data being copied forward, sometimes being moved multiple times. With object-aware data grouping, the chance of objects with similar update frequencies being grouped together becomes higher, and hence lower WA and better wear- leveling.

We show this as an example of object aware data grouping that provides better GC efficiency with lower overhead invoked. More intelligent mechanisms could be designed, for example, application specific objects grouping and data placing in the case of single or multiple instances on a single device, or with multiple flash memory devices. Other advanced GC algorithms could be further explored such as grouping data belonging to one objects during garbage collection so that number of indexing records could be reduced, and hence lower metadata overhead.

5.4 Future work

As object-based flash memory is one of the approaches to collapsing multiple layer of logs in the storage hierarchy, we discussed several design choices that could be made to achieve improved performance as well as better endurance through the rich object-based interface. We present one solution to reduce GC overhead by grouping data on object-level with the support of two append points on the device. In the future, other advanced features could be furthre explored. We plan to extend our work of object-level data grouping to a more mature mode. Currently we keep all object related metadata such as indexing and objects' access time in memory, as a result, the overhead of metadata is not thoroughly investigated when they are flushed to the device. By flushing metadata to the device, different object grouping and placing strategies will be considered as metadata itself has its unique characteristics and reliability requirements. In addition, the grouping of data will also affect the overhead of total metadata, its access patterns, and hence the GC efficiency.

We have mentioned that several advanced object aware features could be designed for improved system performance and reliability. We also plan to explore these features in the future. When object-based flash device is used for caching in between DRAM and hard disks, it would be interesting to see the effects of object-aware data eviction and admission. We believe with careful design choices, object-aware data grouping could help improve cache hit rates as well as reduce GC ovehread.

5.5 Summary

In this chapter, we present the approaches to collapsing multiple layers of logs in storage systems. These approaches do not have to construct any log-like mechanism in the file system layer to work efficiently with flash device. Instead, they utilize the log-like FTL on the flash device to achieve light-weighted data management. The focus of our work was to explore one of the approaches - object-based storage system. We discussed its superiority in flash-based device, which could be further extended to other NVRAM technologies.

We built an object-based flash device simulator, and embedded the object-aware add-on features - object-aware data grouping when the device supports two-append ponts. Through our simulation experiments against two converted object-level traces, the object-aware data grouping outperforms two other data placing strategies with reduced erase counts and balanced wear-leveling factor. In addition, we identified a number of areas for further reserach. The impact of object metadata grouping and placing is an interesting area to explore. Using object-based flash memory as a caching device is another interesting area, for example, the effect of object-aware eviction and admission.

Chapter 6

Conclusions

Modern storage systems have moved to a non-volatile world, in which, flash memory is one of the most promising technologies that providing high throughput and stability with relatively low cost. In this chapter, we conclude contributions of our study in flashs-based storage systems, and discussed the limitation of the work as well as future research directions.

6.1 Flash-based caching device

We have exclusively explored write-through as the caching mode for an SSC in chapter 3. we demonstrate some performance and endurance issues that arise in flash-based caches. We show that workloads on SSCs have significantly greater write pressures than their storage counterparts (as misses become writes). We analyzed the individual contributions and cumulative choices of cache admission and eviction policies, and GC strategy on the overall hit rate and endurance. We show uncoordinated algorithms at Cache and FTL layers are less efficient than coordinated policies (where GC assists cache eviction). With advances in NAND technology, P/E cycles are decreasing and finding ways to reduce P/E cycle consumption is critical. Combined reduction of CLWA and FLWA through our techniques enables SSCs to maintain endurance.

Also, cache hit rate is a critical performance parameter and improving it must be balanced with approaches to reducing CLWA and FLWA. Our analysis showed that cache admission algorithms (for improving hit rate) can be used cooperatively with GC and cache eviction algorithms with improved endurance. Through our experiments, 83% of 18 simulations showed increased or maintained hit rate with significantly improved endurance. Moreover, use of Virtual Storage Containers (VSCs) with quality of service (QoS) enforcement on SSCs provides for improved hit rate at the expense of WA. We identified that when using VSCs and QoS, the hit rates approached that of cache-based eviction with improvements seen in both write amplification and wear leveling as contrasted in GC-based eviction.

In the future we plan to extend our work to write-back caching which has higher requirements on data consistency and integrity. In addition, multiple instances of caching on a single device will be futher investigated. The impact of partitioning an SSC on GC, hit rates, write endurance and so forth and how these affect the application-level response latency is an interesting topic to explore.

6.2 Flash-based primary storage

In chapter 4 and 5, we explore the best way to use flash memory as a primary storage device, both on the file system design and on the device part. We use log-structured system as an example to demonstrate its deficiency in flash-based systems though it has been very widely used in many applications today.

In chapter 4, we characterize the interactions between multiple levels of independent logs, and describe several practical scenarios which arises in real systems. We also built a log-on-log simulator to quantify the overhead that multiple layers of logs brought in to the system, more specifically on the write amplifications and its impact on system performance and flash endurance. We show that log on log can result in highly counterintuitive behaviors. However, log-structured system can still bring benefit to a wide rang of applications from database, key-value caching, to all-flash backing storage, for example, efficient space management, fast recovery, and easy transactional consistency, and so forth. Given this context, we recommend some design choices to reduce the overall overhead by coordinating different log layers activities, including tuning segment size and a lightweighted defragger. Though the focus of our study is on a single log in each layer, the problems we described can be propagated to any number of logs, and the design choices we recommend is general to m-on-n logs. We intend to exclusively explore the impacts of multiple logs for various applications in the future.

In chapter 5, we discuss the methods of collapsing logs. In applications that the cost of maintaining a log structure exeeds its benefit or there are other light-weighted methods to achieve these beneift, the best way is to collapse logs on the file system layer while a log-like mapping in flash cannot be eliminated. While there are two methods that utilize the nature of flash memory to avoid duplicated logs - DirectFS and object-based flash system, we focus on exploring the feasibility and flexibility of designing an object-based flash systems with improved performance and reduced GC overhead. We show through simulation results the benefit of adding advanced features to an object-based flash system.

Bibliography

[1] http://www.fusionio.com/products/directcache/.

[2] http://en.wikepedia.org/wiki/Write_amplification.

[3] http://en.wikepedia.org/wiki/TRIM.

[4] FatCache. https://github.com/twitter/fatcache.

[5] MSR Cambridge Traces. http://iotta.snia.org/traces/388.

[6] NetApp flash cache. http://www.netapp.com/us/products/storage-systems/flash-cache/.

[7] Trends in nand flash memory error correction. http://www.cyclicdesign.com/whitepapers/Cyclic_Design_NAND_ECC.pdf, 2009.

[8] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 57–70, Berkeley, CA, USA, 2008. USENIX Association.

[9] Aleph One Ltd. YAFFS: Yet another flash file system. http://www.yaffs.net.

[10] Andrea C. Arpaci-dusseau, Remzi H. Arpaci-dusseau, and Vijayan Prabhakaran. Removing the costs of indirection in flash-based ssds with nameless writes. In *Proceedings of the 2nd USENIX conference on Hot topics in storage and file systems*, 2010.

[11] Storage Networking Industry Association. `http://www.tpc.org/tpce/default.asp`.

[12] Ricardo Baeza-Yates, Flavio Junqueira, Vassilis Plachouras, and Hans Friedrich Witschel. Admission policies for caches of search engine results. In *Proceedings of the 14th international conference on String processing and information retrieval*, SPIRE'07, pages 74–85, Berlin, Heidelberg, 2007. Springer-Verlag.

[13] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 198–212, New York, NY, USA, 1991. ACM.

[14] Timothy Bisson and Scott A. Brandt. Reducing hybrid disk write latency with flash-backed i/o requests.

[15] Werner Bux and Ilias Iliadis. Performance of greedy garbage collection in flash-based solid-state drives. *Perform. Eval.*, 67(11):1172–1186, November 2010.

[16] Steve Byan, James Lentini, Anshul Madan, and Luis Pabon. Mercury: Host-side flash caching for the data center. In *MSST*, pages 1–12. IEEE, 2012.

[17] Luis Cabrera and Darrell D.E. Long. Swift: Using distributed disk striping to provide high i/o data rates. Technical report, Santa Cruz, CA, USA, 1991.

[18] Li-Pin Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 1126–1130, New York, NY, USA, 2007. ACM.

[19] 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 eleventh international joint conference on Measurement and modeling of computer systems*, pages 181–192, 2009.

[20] Shih-Liang Chen, Bo-Ru Ke, Jian-Nan Chen, and Chih-Tsun Huang. Reliability analysis and improvement for multi-level non-volatile memories with soft information. In *Proceedings of the 48th Design Automation Conference*, pages 753–758, 2011.

[21] Mei-Ling Chiang, Paul C. H. Lee, and Ruei-Chuan Chang. Using data clustering to improve cleaning performance for plash memory. *Softw. Pract. Exper.*, 29(3):267–290, March 1999.

[22] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The logical

disk: a new approach to improving file systems. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, 1993.

[23] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: High throughput persistent key-value store. *Proc. VLDB Endow.*, 3(1-2):1414–1425, September 2010.

[24] Exablox. OneBlox. `https://www.exablox.com/products/`.

[25] Fusion-io. Fusion-io ioDriveII. `http://www.fusionio.com/products/iodrive2/`.

[26] Fusion-io. Fusion-io ioN. `http://www.fusionio.com/products/ion-accelerator/`.

[27] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2):138–163, June 2005.

[28] Gregory R. Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '97, pages 1–1, Berkeley, CA, USA, 1997. USENIX Association.

[29] Garth A. Gibson and Rodney Van Meter. Network attached storage architecture. *Commun. ACM*, 43(11):37–45, November 2000.

[30] Greg Gillis, Swaminathan Sundararaman, Nisha Talagala, Amar Mudrankit, and Jonathan Ludwig. Admission polices for solid state cache devices. In *Proceedings of 2013 Non-Volatile Memories Workshop*, 2013.

[31] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. Characterizing flash memory: anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 24–33, 2009.

[32] Laura M. Grupp, John D. Davis, and Steven Swanson. The bleak future of nand flash memory. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 2–2, 2012.

[33] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Leveraging value locality in optimizing nand flash-based ssds. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pages 7–7, Berkeley, CA, USA, 2011. USENIX Association.

[34] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Leveraging value locality in optimizing nand flash-based ssds. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, pages 7–7, 2011.

[35] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94.

[36] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. In *Pro-*

ceedings of SYSTOR 2009: The Israeli Experimental Systems Conference, SYSTOR '09, pages 10:1–10:9, New York, NY, USA, 2009. ACM.

[37] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 10:1–10:9, 2009.

[38] Adrian Hunter. A brief introduction to the design of UBIFS. http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf.

[39] Kim Jaegeuk. F2FS:flash-friendly file system. http://en.wikipedia.org/wiki/F2FS.

[40] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). *SIGARCH Comput. Archit. News*, 38(3):60–71, June 2010.

[41] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '02, pages 31–42, New York, NY, USA, 2002. ACM.

[42] William K. Josephson, Lars A. Bongo, David Flynn, and Kai Li. DFS: A file system for virtualized flash storage. In *FAST '10: Proccedings of the 8th conference on File and storage technologies.* USENIX Association, 2010.

[43] Dongwon Kang, Dawoon Jung, Jeong-Uk Kang, and Jin-Soo Kim. μ-tree : An ordered index structure for nand flash memory. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 144–153, 2007.

[44] 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*, pages 161–170, 2006.

[45] Yangwook Kang and Ethan L. Miller. Adding aggressive error correction to a high-performance compressing flash file system. In *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*, pages 285–294, New York, NY, USA, 2009. ACM.

[46] Yangwook Kang and Ethan L. Miller. Adding aggressive error correction to a high-performance flash file system. October 2009.

[47] Yangwook Kang, Jingpei Yang, and Ethan L. Miller. Efficient storage management for object-based flash memory. August 2010.

[48] Yangwook Kang, Jingpei Yang, and Ethan L. Miller. Object-based SCM: An efficient interface for storage class memories. May 2011.

[49] Taeho Kgil, David Roberts, and Trevor Mudge. Improving NAND flash based disk caches. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 327–338, Washington, DC, USA, 2008. IEEE Computer Society.

[50] Jong Min Kim, Jongmoo Choi, Jesung Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 9–9, Berkeley, CA, USA, 2000. USENIX Association.

[51] Youngjae Kim, Sarp Oral, Galen M. Shipman, Junghee Lee, David A. Dillow, and Feiyi Wang. Harmonia: A globally coordinated garbage collector for arrays of solid-state drives. In *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies*, MSST '11, pages 1–12, Washington, DC, USA, 2011. IEEE Computer Society.

[52] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. FlashSim: A simulator for NAND flash-based solid-state drives. In *Proceedings of the 2009 First International Conference on Advances in System Simulation*, SIMUL '09, pages 125–131, Washington, DC, USA, 2009. IEEE Computer Society.

[53] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev.*, 40(3):102–107, July 2006.

[54] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An archi-

tecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11):190–201, November 2000.

[55] Junghee Lee, Youngjae Kim, Galen M. Shipman, Sarp Oral, Feiyi Wang, and Jongman Kim. A semi-preemptive garbage collector for solid state drives. In *ISPASS*, pages 12–21. IEEE Computer Society, 2011.

[56] 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. on Embedded Computing Systems*, 6(3), 2007.

[57] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. LAST: locality-aware sector translation for nand flash memory-based storage systems. *SIGOPS Oper. Syst. Rev.*, 42(6):36–42, October 2008.

[58] Young-Sik Lee, Sang-Hoon Kim, Jin-Soo Kim, Jaesoo Lee, Chanik Park, and Seungryoul Maeng. Ossd: A case for object-based solid state drives. *Mass Storage Systems and Technologies, IEEE / NASA Goddard Conference on*, 0:1–13, 2013.

[59] 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*, SOSP '11, 2011.

[60] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In

Proceedings of the 11th USENIX Conference on File and Storage Technologies, FAST'13, pages 257–270, Berkeley, CA, USA, 2013. USENIX Association.

[61] Yuu Maeda and Haruhiko Kaneko. Error control coding for multilevel cell flash memories using nonbinary low-density parity-check codes. In *Proceedings of the 2009 24th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 367–375, 2009.

[62] Carlos Maltzahn, Kathy J. Richardson, and Dirk Grunwald. Reducing the disk i/o of web proxy server caches. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 17–17, Berkeley, CA, USA, 1999. USENIX Association.

[63] Micron. `http://www.micron.com/~/media/Documents/Products/White%20Paper/nand_201.pdf`.

[64] 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 conference on File and Storage Technologies*, pages 12–12, 2012.

[65] Ian Murdock and John H. Hartman. Swarm: a log-structured storage system for linux. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '00, pages 28–28, Berkeley, CA, USA, 2000. USENIX Association.

[66] Muthukumar Murugan and David. H. C. Du. Rejuvenator: A static wear

leveling algorithm for NAND flash memory with minimized overhead. In *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies*, MSST '11, pages 1–12, Washington, DC, USA, 2011. IEEE Computer Society.

[67] Muthukumar Murugan and David. H. C. Du. Rejuvenator: A static wear leveling algorithm for nand flash memory with minimized overhead. In *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies*, pages 1–12, 2011.

[68] D. Nagle, M. E. Factor, S. Iren, D. Naor, E. Riedel, O. Rodeh, and J. Satran. The ANSI t10 object-based storage standard and current implementations. *IBM J. Res. Dev.*, 52(4):401–411, July 2008.

[69] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to ssds: analysis of trade-offs. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 145–158, 2009.

[70] David Nellans, Michael Zappe, Jens Axboe, and David Flynn. PTRIM + EXISTS: Exposing new FTL primitives to applications. In *2nd Annual Non-Volatile Memories Workshop*, 2011.

[71] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems. In *Proceedings of the 10th USENIX*

conference on File and Storage Technologies, FAST'12, pages 25–25, Berkeley, CA, USA, 2012. USENIX Association.

[72] Percona. Tuning for speed: Percona server and fusion-io. `http://www.percona.com/files/presentations/percona-live/nyc-2011/PerconaLiveNYC2011-Tuning-For-Speed-Percona-Server-and-Fusion-io.pdf`.

[73] Taciano Perez and Cesar A. F. De Rose. Non-volatile memory: Emerging technologies and their impacts on memory systems. In *Technical Report*, 2010.

[74] Juan Piernas, Toni Cortes, and José M. García. DualFS: a new journaling file system without meta-data duplication. In *Proceedings of the 16th international conference on Supercomputing*, pages 137–146, 2002.

[75] Konstantinos Psounis and Balaji Prabhakar. Efficient randomized web-cache replacement schemes using samples from past eviction times. *IEEE/ACM Trans. Netw.*, 10(4):441–455, August 2002.

[76] PureStorage. Purestorage flash array. `http://www.purestorage.com/flash-array/`.

[77] Abhishek Rajimwale, Vijayan Prabhakaran, and John D. Davis. Block management in solid-state devices. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, pages 21–21, 2009.

[78] David Roberts, Taeho Kgil, and Trevor Mudge. Integrating NAND flash devices onto servers. *Commun. ACM*, 52(4):98–103, April 2009.

[79] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. *Proc. VLDB Endow.*, 5(4), December 2011.

[80] Mendel Rosenblum. The design and implementation of a log-structured file system. Technical report, Berkeley, CA, USA, 1992.

[81] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for dram-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 1–16, Santa Clara, CA, 2014. USENIX.

[82] SamSung. `http://www.samsung.com/global/business/semiconductor/minisite/SSD/us/html/about/overview.html?gclid=CL764c3A1r0CFQ5gMgodHWIACQ`.

[83] SanDisk. `http://www.sandisk.com/products/ssd/`.

[84] Mohit Saxena and Michael M. Swift. FlashVM: Revisiting the virtual memory hierarchy. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS-XII)*, 2009.

[85] Mohit Saxena, Michael M. Swift, and Yiying Zhang. FlashTier: a lightweight, consistent and durable storage cache. In *Proceedings of the*

7th ACM european conference on Computer Systems, EuroSys '12, pages 267–280, New York, NY, USA, 2012. ACM.

[86] Mohit Saxena, Michael M. Swift, and Yiying Zhang. FlashTier: a lightweight, consistent and durable storage cache. In *EuroSys*, pages 267–280, 2012.

[87] Nimble Storage. CASL hybrid flash storage architecture. `http://www.nimblestorage.com/products/architecture.php`.

[88] Sriram Subramanian. *Beyond the Block-based Interface for Flash-based Storage*. PhD thesis, Madison, WI, USA, 2013. AAI3560183.

[89] Hairong Sun, Pete Grayson, and Bob Wood. Quantifying reliability of solid-state storage from multiple aspects, 2011.

[90] Hitachi Data Systems. An evolution in storage: The object store. `http://www.hds.com/assets/pdf/hitachi-solution-profile-object-based-storage.pdf`.

[91] TokuTek. Tokudb. http://www.tokutek.com/.

[92] Michael Wei, Laura M. Grupp, Frederick E. Spada, and Steven Swanson. Reliably erasing data from flash-based solid state drives. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, 2011.

[93] Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang. An efficient b-tree layer

implementation for flash-memory storage systems. *ACM Trans. Embed. Comput. Syst.*, 6(3), July 2007.

[94] Yulai Xie, Kiran-Kumar Muniswamy-Reddy, Dan Feng, Darrell D. E. Long, Yangwook Kang, Zhongying Niu, and Zhipeng Tan. Design and evaluation of oasis: An active storage framework based on t10 osd standard. In *Proceedings of the 27th IEEE Symposium on Massive Storage Systems and Technologies (MSST 2011)*, May 2011.

[95] Chul-Woong Yang, Ki Yong Lee, Yon Dohn Chung, Myoung-Ho Kim, and Yoon-Joon Lee. An effective self-adaptive admission control algorithm for large web caches. *IEICE Transactions*, 92-D(4):732–735, 2009.

[96] Jingpei Yang, Ned Plasson, Greg Gillis, and Nisha Talagala. HEC: Improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 10:1–10:11, New York, NY, USA, 2013. ACM.

[97] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, Berkeley, CA, USA, 2012.

[98] Zhihui Zhang and Kanad Ghose. hFS: a hybrid file system prototype for improving small file and metadata performance. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, 2007.