EXPLOITING INTRINSIC FLASH PROPERTIES TO ENHANCE MODERN STORAGE SYSTEMS

A Dissertation
Presented to
The Academic Faculty

By

Jian Huang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

August 2017

EXPLOITING INTRINSIC FLASH PROPERTIES TO ENHANCE MODERN STORAGE SYSTEMS

Approved by:

Date Approved: July 20, 2017

Dedicated to my parents, sister, and wife

for their infinite love, encouragement, and support

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my thesis advisor, Moinuddin K. Qureshi, for his continuous support, guidance, and patience. He always inspires me to be a better researcher in many ways. I will remember many of our discussions on research in his office or the Starbucks Cafe in Clough Commons at Georgia Tech. All these bits and pieces make up a precious part of the memory of my Ph.D. life. His principles and dedication will have a great influence on me.

I owe an immeasurable debt to my another advisor, Karsten Schwan, for his kindness and mentorship. I really appreciate the opportunity to work with him to start a new journey of my Ph.D. study. His helpful nature and smile are an endless source of energy to keep me going forward. We all miss you, Karsten.

I would like to express my sincere gratitude to my mentor Anirudh Badam for his mentorship and friendship. This thesis would not have been possible without his gracious advice and encouragement. I am thankful for the opportunities to work with him at Microsoft Research and enable me to see the beauty of systems research. And most importantly, we made a friendship that would last a lifetime. I would also like to thank my other research mentors Ranveer Chandra, Ed Nightingale, Laura Caulfield, Suman Nath, Sudipta Sengupta, and Bikash Sharma at Microsoft Research for their guidance and support. The research projects we have done were always full of adventure and expectations, which made me have many wonderful summers in Seattle.

Special thanks to my recent collaborator Peng Liu for his mentorship. Peng was always available for discussions and gave me useful advice on my career development. I would also like to thank Xuechen Zhang, Nitin Agrawal, Dawei Li, Xiaolong Wang, Xiangyong Ouyang, Hao Wang, Harry Xu, Xingyu Xin, and Feng Qin for their discussions, suggestions, and feedback on my research.

I would like to thank Umakishore Ramachandran, Steven Swanson, Taesoo Kim, and

James Mickens for their time to serve on my thesis committee and their valuable feedback on my thesis proposal and the final dissertation.

Thank you to the group members of the memory systems lab, including Chia-Chen Chou, Vinson Young, Swamit Tannu, Gururaj Saileshwar, Prashant Nair, and Mohammad Arjomand. I am so proud to be part of such a talented, fun and motivated team.

Thanks also goes to my CERCS office-mates, especially Alexander Merritt, Yanwei Zhang, Chao Chen, Anshuman Goswami, Chengwei Wang, and Fang Zheng. They are always available for discussions and help. I would also like to thank Ada Gavrilovska, Greg Eisenhauer, and Susie Y. McClain for keeping the CERCS lab running during the most difficult time to provide a stable and helpful environment for our studies.

I am especially grateful to my parents, sister, and wife for their infinite love, endless patience, and unconditional support. They always encourage me in all of my pursuits and inspire me to follow my heart. For years my parents have set personal examples to show the value of hard work for me. The efforts they put in educating my sister and me always motivate us to become better persons. My sister is much more than just a sister, her continuous support and encouragement empower me to pursue my dreams. Finally, I would like to thank my wife who has made my Ph.D. journey worthwhile. We look forward to starting a new chapter in our lives together.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The longstanding goals of storage system design have been to provide simple abstractions for applications to efficiently access data while ensuring the data durability and security on a hardware device. The traditional storage system, which was designed for slow hard disk drive with little parallelism, does not fit for the new storage technologies such as the faster flash memory with high internal parallelism. The gap between the storage system software and flash device causes both resource inefficiency and sub-optimal performance. This dissertation focuses on the rethinking of the storage system design for flash memory with a holistic approach from the system level to the device level and revisits several critical aspects of the storage system design including the storage performance, performance isolation, energy-efficiency, and data security.

The traditional storage system lacks full performance isolation between applications sharing the device because it does not make the software aware of the underlying flash properties and constraints. This dissertation proposes FlashBlox, a storage virtualization system that utilizes flash parallelism to provide hardware isolation between applications by assigning them on dedicated chips. FlashBlox reduces the tail latency of storage operations dramatically compared with the existing software-based isolation techniques while achieving uniform lifetime for the flash device.

As the underlying flash device latency is reduced significantly compared to the conventional hard disk drive, the storage software overhead has become the major bottleneck. This dissertation presents FlashMap, a holistic flash-based storage stack that combines memory, storage and device-level indirections into a unified layer. By combining these layers, FlashMap reduces critical-path latency for accessing data in the flash device and improves DRAM caching efficiency significantly for flash management.

The traditional storage software incurs energy-intensive storage operations due to the need for maintaining data durability and security for personal data, which has become a significant challenge for resource-constrained devices such as mobiles and wearables. This dissertation proposes WearDrive, a fast and energy-efficient storage system for wearables. WearDrive treats the battery-backed DRAM as non-volatile memory to store personal data and trades the connected phone's battery for the wearable's by performing large and energy-intensive tasks on the phone while performing small and energy-efficient tasks locally using battery-backed DRAM. WearDrive improves wearable's battery life significantly with negligible impact to the phone's battery life.

The storage software which has been developed for decades is still vulnerable to malware attacks. For example, the encryption ransomware which is a malicious software that stealthily encrypts user files and demands a ransom to provide access to these files. Prior solutions such as ransomware detection and data backups have been proposed to defend against encryption ransomware. Unfortunately, by the time the ransomware is detected, some files already undergo encryption and the user is still required to pay a ransom to access those files. Furthermore, ransomware variants can obtain kernel privilege to terminate or destroy these software-based defense systems. This dissertation presents FlashGuard, a ransomware-tolerant SSD which has a firmware-level recovery system that allows effective data recovery from encryption ransomware. FlashGuard leverages the intrinsic flash properties to defend against the encryption ransomware and adds minimal overhead to regular storage operations.

CHAPTER 1
INTRODUCTION

For decades, storage systems have been designed for block-based devices such as hard disk drive (HDD) that has little parallelism and long access latency. The widely used solid state disk (SSD) out-performs HDD by orders of magnitude, providing up to 5000x more IOPS, at 1% of the latency [1] without any physical seek operation. Such performance characteristics make SSD more similar to DRAM than to HDD. Moreover, SSD behaves intrinsically different from HDD. For example, typical SSDs have high internal parallelism by organizing their flash blocks into a hierarchy of multiple chips; they employ a flash translation layer (FTL) for out-of-place writes in order to mitigate the long erase latency of flash memories and also ensuring uniform wear for flash blocks. These unique properties make their storage management fundamentally different from that of HDDs.

1.1 The Problem

To be compatible with the existing computer systems and simplify the software development, the storage systems that were originally made for HDD are employed for flash memory today, which causes a large gap between storage system software and hardware device. The gap not only causes resource inefficiency across the whole storage system stack but also results in the uncertainty for data security and durability, since the flash device is taken as a 'black box' under the conventional block abstraction. To fully exploit the hardware benefits, it is important to rethink the system design for flash memory with a holistic cross-layer approach.

The storage system is normally virtualized and shared by multiple applications for resource efficiency and these applications can interfere with each other. Therefore, a long-standing goal of storage virtualization has been to provide performance isolation between multiple applications sharing the device. Virtualizing SSDs, however, has traditionally been a challenge because of the fundamental tussle between resource isolation and the lifetime of the flash device. The SSDs aim to uniformly age all the regions of flash and this violates the isolation assumption in traditional storage systems. Although the storage system software has provided isolation mechanisms such as token bucket rate limiters and intelligent IO throttling [2, 3, 4], the FTL in SSDs, which is responsible for address translation and wear leveling for flash blocks, does not provide any isolation guarantee underneath the flash controller and this makes the storage performance highly unpredictable. This dissertation proposes utilizing flash parallelism to improve isolation between applications by running them on dedicated chips. It also proposes allowing the wear of different chips to diverge at fine time granularities in favor of performance isolation and adjusting the imbalance at a more coarse time granularity in a principled manner.

As flash device out-performs hard disk drive by orders of magnitude in terms of the device latency and throughput, the storage software has become the major bottleneck when accessing data on physical device. To improve the storage performance, an emerging approach to using SSDs treats them as a slower form of non-volatile memory. For example, NoSQL databases like MongoDB [5, 6], LMDB [7] which are widely deployed use SSDs via the existing memory-mapped file interface. Such an approach eases application development and automatically tiers SSD under the DRAM by the operating system (OS) manager. Using SSDs in this manner, unfortunately, is inefficient as there are three software layers with redundant functionalities between the application and flash device. The first of these, memory-level indirection, involves page table translations and sanity checks by the OS memory manager. The second of these, storage-level indirection, involves converting file offsets to blocks on the SSD and permission checks by the file system. The final one, device-level indirection, is for the FTL of the SSD. Redundant indirections and checks not only increase the latency of flash accesses but also affect performance by requiring precious DRAM space to cache indirection data across all the three layers. This dissertation proposes a holistic SSD architecture that combines memory, storage and device-level indirections into a unified layer.

Beyond the performance issues, the storage software also incurs energy-intensive storage operations which consume up to 110x more energy compared to flash hardware for accessing data [8]. These energy overheads pose a big challenge for resource-constrained platforms such as wearables because the size and weight constraints on wearables limit their battery capacity and restrict them from providing rich functionality. The need for durable and secure storage for personal data further compounds this problem as these features incur energy-intensive operations. This dissertation presents WearDrive, a fast in-memory storage system for wearables by treating the battery-backed DRAM as non-volatile memory to avoid the energy-intensive storage operations and leveraging the low-power network connectivity on wearables to trade the resources on the phone for the wearable.

Finally, this dissertation also investigates the security aspect of the storage system stack. Although the storage system has been developed for decades, it is still vulnerable to malware attacks. For instance, the encryption ransomware which is a malicious software that stealthily encrypts user files and demands a ransom to provide access to these files. Several prior studies have developed systems to detect ransomware by monitoring the activities that typically occur during a ransomware attack. Unfortunately, by the time the ransomware is detected, some files already undergo encryption and the user is still required to pay a ransom to access those files. Furthermore, ransomware variants can obtain kernel privilege, which allows them to terminate software-based defense systems, such as anti-virus. While periodic backups have been explored as a means to mitigate ransomware, such backups incur storage overheads and are still vulnerable as ransomware can obtain kernel privilege to stop or destroy backups. Therefore, it is important to defend against ransomware without relying on software-based solutions and without incurring the storage overheads of backups. This dissertation proposes FlashGuard, a ransomware-tolerant SSD which has a firmware-level recovery system that allows quick and efficient data recovery from encryption ransomware without relying on explicit software-based solutions.

1.2 Thesis Statement

Redesigning the storage system to exploit the intrinsic properties of flash memory can substantially improve the scalability, reliability, and security of flash-based storage systems.

1.3 Contributions

This dissertation makes the following contributions:

- This dissertation proposes hardware isolation for flash-based storage system by utilizing the internal parallelism of flash memory and assigning applications on dedicated chips. It also presents a new wear-leveling scheme to manage the wear of flash chips at fine time granularities. Such a new design makes SSD wears uniformly while the tail latency of storage operations are reduced significantly compared to the traditional software-based isolation techniques.

- This dissertation presents a holistic SSD architecture for scalable SSD with larger capacity. It combines memory, storage, and device-level indirections into a unified layer while preserving the properties of each layer. Such an architecture significantly improves DRAM caching efficiency and reduces the latency of accessing a page on SSD. It also presents a way of unifying the new and emerging memory and storage technologies such as 3D XPoint memory.

- This dissertation proposes an energy-efficient storage system based on the battery-backed DRAM to avoid the energy-intensive storage operations to the flash device. The DRAM on mobile and wearable devices can be treated as non-volatile memory for free because they are backed by non-removable batteries. Such a type of non-volatile memory opens a new field for persistent memory research.

- This dissertation also presents a ransomware-tolerant SSD which has the capability of defending against encryption ransomware by leveraging the intrinsic flash properties. Such an SSD can protect user data from encryption ransomware without relying on the software-based defense systems, while adding minimal overhead to regular storage operations and avoiding storage overhead of explicit data backups.

1.4 Dissertation Organization

The rest of the dissertation is organized as follows. Related work is discussed in Chapter 2. Chapter 3 describes the hardware isolation for multi-tenant applications. Chapter 4 discusses the unified address translation for memory-mapped SSDs. The energy-efficient storage system for wearables is presented in Chapter 5. Chapter 6 presents the ransomware-tolerant SSD. Finally, Chapter 7 summarizes the dissertation and discusses the directions of future work.

CHAPTER 2
BACKGROUND AND RELATED WORK

As the replacement to conventional persistent storage devices – hard disk drives (HDDs), Solid-State Drives (SSDs) have been widely used on many kinds of computing platforms, because they provide orders of magnitude better performance than HDDs while their cost is fast approaching to that of HDDs [9, 10, 11, 12]. This chapter describes the technical background of SSDs. Related work for the specific problems in the storage system stack will be further discussed in the corresponding chapter.

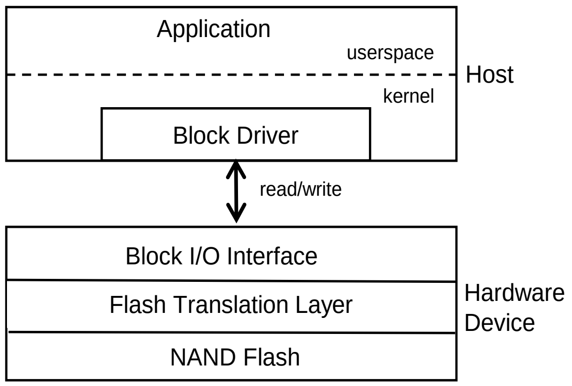


Figure 2.1: The system architecture of using an SSD with block I/O interface.

2.1 Flash Memory: Background and Terminology

Same as conventional HDDs, a commodity SSD employs a block interface to encapsulate the idiosyncrasies of flash devices as shown in Figure 2.1. As such, it gives upper-level software systems (such as file systems) an impression that both SSD and HDD perform storage operations in the same manner. At the hardware level, however, an SSD is fundamentally different from HDD.

Given an SSD, each physical page can be written only after it is erased. Unfortunately, erase operation can be performed only at block (which has multiple pages) granularity and

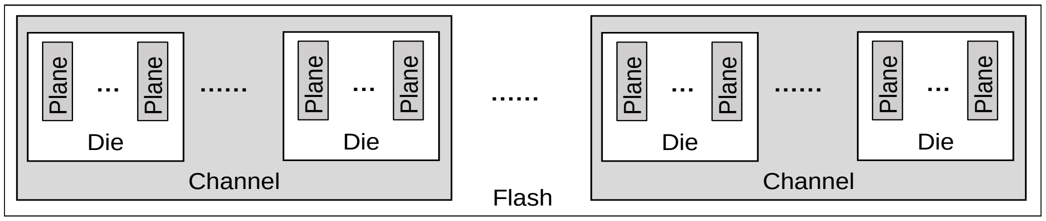


Figure 2.2: The internal parallelism in an SSD.

such operations are time-consuming. Therefore, SSDs issue the writes to free pages which have been erased in advance (i.e., out-of-place write) rather than waiting for the expensive erase operation for every write, and garbage collection (GC) will be executed later to clean the stale data on SSDs. Moreover, each flash block has limited endurance: it is rated only for a few thousand erase operations, therefore it is important for the blocks to age uniformly. SSDs employ both out-of-place write and GC to overcome the shortcomings of SSDs and maintain indirections in the Flash Translation Layer (FTL) for indexing the virtual-to-physical address mapping.

Typical SSDs organize their flash array into a hierarchy of channels, dies and planes [13, 14]. As shown in Figure 2.2, each SSD has multiple channels, each channel has multiple dies, and each die has multiple planes. The number of channels, dies and planes varies by vendor and generation. Typically, there are 2 - 4 planes per die, 4 - 8 dies per channel, and 8 - 32 channels per drive.

Channels, which share only the resources common to the whole SSD, provide the strongest isolation. Dies execute their commands with complete independence, but they must share a bus with other dies on the same channel. Planes' isolation is limited – the controller may isolate data to different planes, but operations on these data must either happen at different times or to the same address on each plane in a die [15].

2.2 Related Work in Improving Storage Isolation

SSDs have become indispensable for large-scale cloud services as their cost is fast approaching to that of HDDs. The rapidly shrinking process technology has allowed SSDs to boost their bandwidth and capacity by increasing the number of chips. However, the limitations of SSDs' management algorithms have hindered these parallelism trends from efficiently supporting multiple tenants on the same SSD.

Previous work had proposed novel techniques to help application tenants place their data such that underlying flash pages are allocated from separate blocks. This helps improve performance by reducing the write amplification factor (WAF) [16]. Lack of block sharing has the desirable side effect of clumping garbage into fewer blocks, leading to more efficient garbage collection (GC), thereby reducing tail latency of SSDs [17, 18, 19, 20].

However, significant interference still exists between tenants because when data is striped, every tenant uses every channel, die and plane for storing data and the storage operations of one tenant can delay other tenants. Software isolation techniques [21, 22, 23] split the SSD's resources fairly. However, they cannot maximally utilize the flash parallelism when resource contention exists at a layer below because of the forced sharing of independent resources such as channels, dies, and planes.

New SSD designs, such as open-channel SSDs that explicitly expose channels, dies and planes to the operating system [24, 25, 26], can help tenants avoid some of these pitfalls by using dedicated channels. However, the wear imbalance problem between channels that ensues from different tenants writing at different rates remains unsolved.

2.3 Related Work in Improving Storage Performance

To further improve the performance of storage operations against SSDs, an emerging approach to using SSDs treats them as a slower form of non-volatile memory. For example, NoSQL databases like MongoDB [5, 6], LMDB [7] (backend for OpenLDAP) and others [27, 28, 29] which are widely deployed [30] use SSDs via a memory-mapped file interface. There are three advantages to this approach. First, the virtual memory interface eases development. For example, MongoDB uses TCMalloc [31] to manage SSD-file backed memory to create data structures like its B-tree index, and for using the Boost template library. Second, SSDs are automatically tiered under DRAM by the OS's memory manager. Finally, memory that is backed by a file enables durability for data. Such hybrid memory systems have also been proposed in the academic community [32, 33, 34, 35, 36].

However, these systems do not present optimizations for reducing the address translation overhead. Nameless writes [37] and DFS [38] combine the FTL with the file system's index. But, when mapping a file that uses nameless writes or DFS into virtual memory, page tables are created separately on top which increases the address translation and other software overhead.

DFTL [39] proposes caching only the "hot" mappings in memory while other mappings can be stored on the SSD. While such techniques reduce the space requirement of RAM at the FTL, they do not reduce the indirection overheads in higher software layers.

2.4 Related Work in Improving Storage Energy Efficiency

Beyond the storage performance of flash memories, the energy efficiency of storage operations is also a major concern of system design (especially for resource-constrained platforms such as mobiles and wearables). Kim et al. [40] provided the evidence that slow flash technologies such as SD and eMMC are the primary performance bottleneck for several classes of mobile applications. Li et al. [8] studied the energy overhead of mobile storage systems and found that the mobile software stack consumes more power than storage hardware. These findings motivate the research on the energy efficiency of storage systems in this dissertation, as these overheads become more prominent on wearables where the battery is more constrained than on phones.

Recent optimizations to mobile storage [41, 42] address some of the performance problems, but flash is still 10,000x slower compared to DRAM. Emerging non-volatile memory (NVM) technologies like PCM [43, 44, 45] are not yet available in the market. Battery-backed RAM [46, 47, 48] is viable because batteries, DRAM, and flash are pervasive in mobile systems. Luo et al. [46] proposed QuasiNVRAM that is a dedicated, known, contiguous region of physical memory to provide performance benefits for phone applications that use SQLite on Android.

Rio [49], BlueFS [50], EnsemBlue [51] Simba [52], Segank [53], Bayou [54] and PersonalRAID [55] are distributed file system techniques to share personal data efficiently across mobile consumer electronic devices. For wearable devices, their workload characteristics are different. The wearable workloads like extended-display and sensor data analysis focus on the newest data. A quick and energy-efficient mechanism can be exploited to span data and computation across the wearable and the phone.

2.5 Related Work in Improving Storage Security

To ensure the data reliability and security in modern storage systems, a large number of backup systems have been proposed [56, 57, 58]. The ones that have been commonly adopted on Unix systems are `dump` and `tar` utilities. They both support full and incremental backup strategies [56]. On Microsoft Windows system, the most popular backup system is Volume Shadow Copy Service that archives user data on local and external volumes in an incremental manner [57]. Another line of work capable of achieving data recovery are log-structured file systems [59] and journaling file systems [60]. They both maintain data updates in persistent logs. Once data loss or inconsistency occurs, they can recover the data back to previous states by rolling back the logs.

Apart from the backup systems integrated into OSes, other well-developed backup systems include the IBM Tivoli Storage Manager [58] that performs selective, incremental backup in conjunction with deduplication, and those cloud based storage systems [61] that synchronize file updates and creation with the backup storage running on the cloud.

As a defense mechanism, however, none of them is sufficient and proper. To avoid loss of files newly updated or created, they have to perform backup frequently. From the perspective of efficiency, this is particularly time-consuming. Since malware has already run with the kernel privilege, the backup systems proposed can be easily disabled or circumvented. For example, a backup process that synchronizes user files with a cloud storage can be terminated by ransomware with the kernel privilege.

Looking beyond file backups, researchers proposed to integrate proactive defense mechanisms into the existing software systems recently. For example, `ShieldFS` [62] monitors the low-level file access activities to detect ransomware and implements a protection layer with the copy-on-write mechanism to recover data. `PayBreak` [63] hooks crypto functions in the standard libraries to identify the invocations from ransomware and logs the encryption key for future data decryption. Similar to the attacks against file backups, ransomware can easily undermine these mechanisms by disabling them with kernel privilege or obfuscating the execution of its critical functions.

CHAPTER 3
FLASHBLOX: HARDWARE ISOLATED VIRTUAL SSDS

A longstanding goal of storage virtualization has been to provide performance isolation between multiple tenants sharing the device. Virtualizing SSDs, however, has traditionally been a challenge because of the fundamental tussle between resource isolation and the lifetime of the device – existing SSDs aim to uniformly age all the regions of flash and this hurts performance isolation.

We propose utilizing flash parallelism to improve isolation between virtual SSDs by running them on dedicated channels and dies. Furthermore, we offer a complete solution by also managing the wear. We propose allowing the wear of different channels and dies to diverge at fine time granularities in favor of isolation and adjusting that imbalance at a coarse time granularity in a principled manner. Our experiments show that the new SSD wears uniformly while the 99th percentile latencies of storage operations in a variety of multi-tenant settings are reduced by up to 3.1x compared to software isolated virtual SSDs.

3.1 Introduction

SSDs have become indispensable for large-scale cloud services as their cost is fast approaching to that of HDDs. They outperform HDDs by orders of magnitude, providing up to 5,000x more IOPS, at 1% of the latency [64]. The rapidly shrinking process technology has allowed SSDs to boost their bandwidth and capacity by increasing the number of chips. However, the limitations of SSDs' management algorithms have hindered these parallelism trends from efficiently supporting multiple tenants on the same SSD.

The tail latency of SSDs in multi-tenant settings is one such limitation. Cloud storage systems have started collocating multiple tenants on the same SSDs [65, 66, 67] which further exacerbates the already well-known tail latency problem of SSDs [17, 68, 69, 70].

The cause of tail latency is the set of complex flash management algorithms in the SSD's controller, called the Flash Translation Layer (FTL). The fundamental goals of these algorithms are decades-old and were meant for an age when SSDs had limited capacity and little parallelism. The goals were meant to hide the idiosyncrasies of flash behind a layer of indirection and expose a block interface. These algorithms, however, conflate wear leveling (to address flash's limited lifetime) and resource utilization (to exploit parallelism) which increases interference between tenants sharing an SSD.

While application-level flash-awareness [38, 71, 72, 73, 74] improves throughput by efficiently leveraging the device level parallelism, these optimizations do not directly help reduce the interference between multiple tenants sharing an SSD. These tenants cannot effectively leverage flash parallelism for isolation even when they are individually flash-friendly because FTLs hide the parallelism. Newer SSD interfaces [25, 26] that propose exposing raw parallelism directly to higher layers provide more flexibility in obtaining isolation for tenants but they complicate the implementation of wear-leveling mechanisms across the different units of parallelism.

This dissertation proposes leveraging the inherent parallelism present in today's SSDs to increase isolation between multiple tenants sharing an SSD. We propose creating virtual SSDs that are pinned to a dedicated number of channels and dies depending on the capacity and performance needs of the tenant. The fact that the channels and dies can be more or less operated upon independently helps such virtual SSDs avoid adverse impacts on each other's performance. However, different workloads can write at different rates and in different patterns, this could age the channels and dies at different rates. For instance, a channel pinned to a TPC-C database instance wears out 12x faster than a channel pinned to a TPC-E database instance, reducing the SSD lifetime dramatically. This non-uniform aging creates an unpredictable SSD lifetime behavior that complicates both provisioning and load-balancing aspects of data center clusters.

To address this problem, we propose a two-part wear-leveling model which balances wear within each virtual SSD and across virtual SSDs using separate strategies. Intra-virtual SSD wear is managed by leveraging existing SSD wear-balancing mechanisms while inter-virtual SSD wear is balanced at *coarse-time granularities* to reduce interference by using new mechanisms. We control the wear imbalance between virtual SSDs using a mathematical model and show that the new wear-leveling model ensures near-ideal lifetime for the SSD with negligible disruption to tenants.

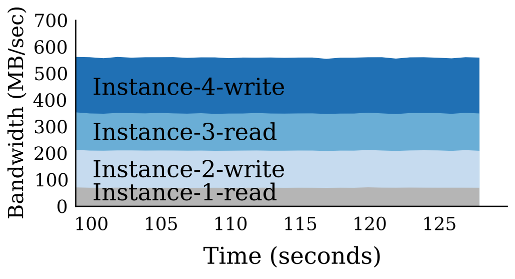
We design and implement FlashBlox and its new wear-leveling mechanisms inside an open-channel SSD stack (from CNEX labs [75]), and demonstrate benefits for a Microsoft data centers' multi-tenant storage workloads: the new SSD delivers up to 1.6x better throughput and reduces the 99th percentile latency by up to 3.1x. Furthermore, our wear leveling mechanism provides 95% of the ideal SSD lifetime even in the presence of adversarial write workloads that execute all the writes on a single channel while only reading on other channels.

3.2 Motivation

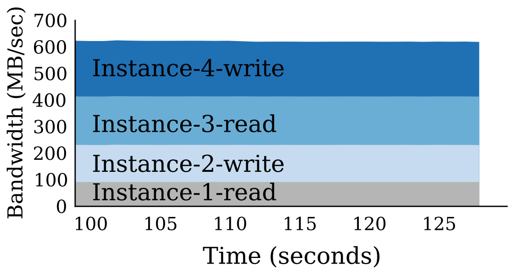
Premium storage Infrastructure-as-a-Service (IaaS) offerings [76, 77, 78], persistent Platform-as-a-Service (PaaS) systems [79] and Database-as-a-Service (DaaS) systems [80, 81, 82, 83] need SSDs to meet their service level objectives (SLO) that are usually outside the scope of HDD performance. For example, DocDB [81] guarantees 250, 1,000 and 2,500 queries per second respectively for the S1, S2 and S3 offerings [84].

Storage virtualization helps such services make efficient use of SSDs' high capacity and performance by slicing resources among multiple customers or instances. Typical database instances in DaaS systems are 10 GB – 1 TB [84, 85] whereas each server can have more than 20 TB of SSD capacity today.

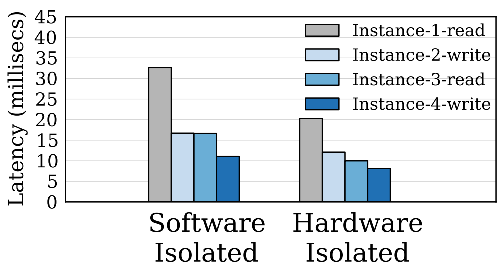
Bandwidth, IOPS [86, 87] or a convex combination of both [21, 88] is limited on a per-instance basis using token bucket rate limiters or intelligent IO throttling [2, 3, 4] to meet SLOs. However, there is no analogous mechanism for sharing the SSD while maintaining



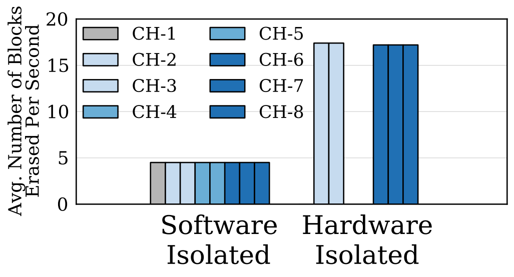
(a) Software Isolation



(b) Hardware Isolation



(c) 99th Percentile Latency



(d) Wear Leveling

Figure 3.1: Tenants sharing an SSD get better bandwidth (compare (a) vs. (b)) and tail latency as shown in (c) when using new hardware isolation. However, dedicating channels to tenants can lead to wear-imbalance between the various channels as shown in (d). A new design for addressing such a wear-imbalance is proposed.

low IO tail latency – an instance's latency still depends on the foreground reads/writes [17, 18, 20] and background garbage collection [16] of other instances.

Moreover, it is increasingly becoming necessary for diverse workloads (e.g., latency-critical applications and batch processing jobs) to be collocated for improving efficiency while being isolated from each other [89, 18]. Virtualization and container technologies are evolving to exploit hardware isolation of memory [90, 91], CPU [92, 93], caches [94, 95], and networks [96, 97] to support such scenarios. We extend this line of research to SSDs by providing hardware isolated SSDs while solving the wear-imbalance problem that arises due to the physical flash partitioning across diverse applications.

3.2.1 Hardware Isolation vs. Wear-Leveling

To understand this problem, we compare the two different approaches to hardware sharing using a representative workload. The first approach stripes data from all the workloads

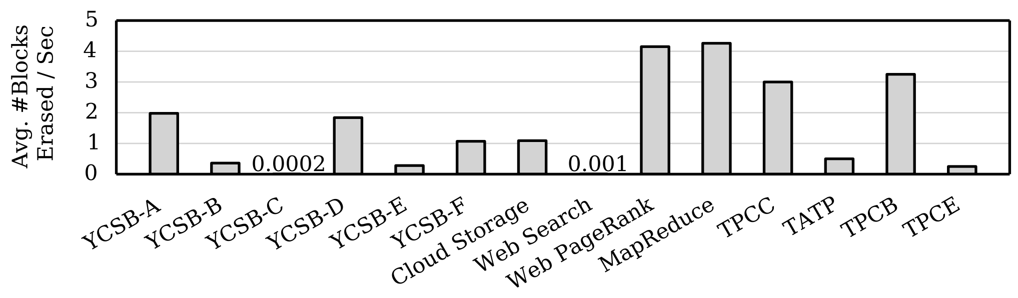


Figure 3.2: The average rate at which flash blocks are erased for various workloads, including NoSQL, SQL, and batch processing workloads.

across all the flash channels (eight total) much like existing SSDs. This scheme provides the maximum throughput for each IO and uses the software rate limiter which has been used for Linux containers and Docker [98, 99] to implement weighted fair sharing of the resources (the scenario for Figure 3.1a). Note that each instance in software isolated case does not share physical flash blocks with other collocated instances to eliminate the unpredictability stemming from SSD firmware [16]. The second approach uses a configuration from our proposed mechanism that provides the hardware isolation by assigning a certain number of channels to each instance (the scenario for Figure 3.1b).

In both scenarios, there are four IO-intensive workloads. These workloads request 1/8th, 1/4th, 1/4th, and 3/8th of the shared storage resource. The rate limiter uses these as weights in the first approach, while FlashBlox assigns 1, 2, 2 and 3 channels respectively. Workloads 2 and 4 perform 100% writes and workloads 1 and 3 perform 100% reads. All workloads issue sequentially-addressed and aligned 64 KB IOs.

Hardware isolation not only reduces the 99th percentile latencies by up to 1.7x (Figure 3.1c), but also increases the aggregate throughput by up to 10.8% compared to software isolation. However, pinning instances to channels prevents the hardware from automatically leveling the wear across all the channels, as shown in Figure 3.1d. We exaggerate the variance of write rates to better motivate the problem: a need to balance wear between hardware isolated virtual SSDs. However, we will explore applications' typical write rates (see Figure 3.2) and tailor our solution to this side effect. To motivate the problem further,

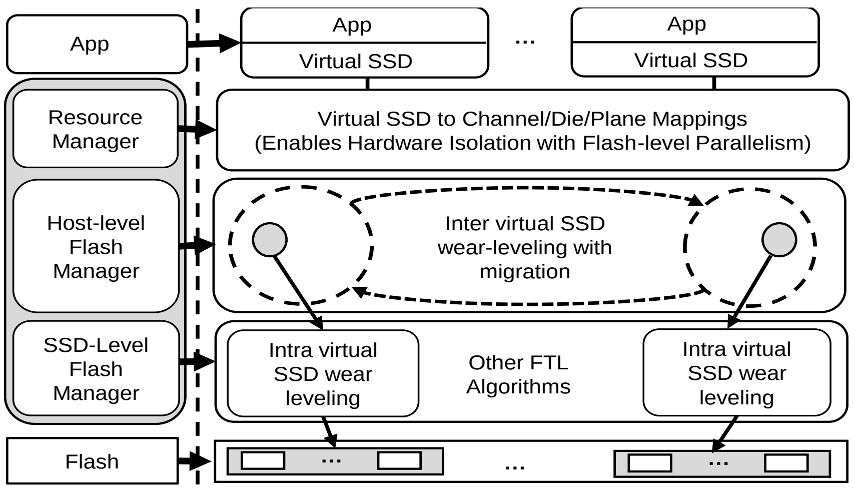

App
App
Virtual SSD
...
App
Virtual SSD
Resource Manager
Virtual SSD to Channel/Die/Plane Mappings (Enables Hardware Isolation with Flash-level Parallelism)
Host-level Flash Manager
Inter virtual SSD wear-leveling with migration
SSD-Level Flash Manager
Intra virtual SSD wear leveling
Other FTL Algorithms
Intra virtual SSD wear leveling
Flash
...

Figure 3.3: The system architecture of FlashBlox.

we must first explore the parallelism available in SSD hardware, and the aspects of FTLs which cause interference in the first approach.

3.2.2 Leveraging Parallelism for Isolation

As shown in Figure 2.2, the architecture of flash memories plays an important role in defining isolation boundaries. In current drives, none of this flexibility is exposed to the host. Drives instead optimizes for a single IO pattern: extremely large or sequential IO. The FTL logically groups all planes into an array, effectively creating large super-pages and super-blocks. Striping increases the throughput of large, sequential IOs, but introduces the negative side effect of interference between multiple tenants sharing the drive. As all data is striped, every tenant's reads, writes and erases can potentially conflict with every other tenant's operations.

3.3 FlashBlox Design

We now describe FlashBlox whose architecture is shown in Figure 3.3. At a high level, FlashBlox consists of the following three components: (1) A resource manager that allows tenants to allocate and deallocate virtual SSDs (vSSD); (2) A host-level flash manager that implements inter-vSSD wear-leveling by balancing wear across channels and dies at

Table 3.1: Virtual SSD types supported in FlashBlox.

Virtual SSD Type	Isolation Level	Allocation Granularity
Channel Isolated vSSD	High	Channel
Die Isolated vSSD	Medium	Die
Software Isolated vSSD	Low	Plane/Block
Unisolated vSSD	None	Block/Page

coarse time granularities; (3) An SSD-level flash manager that implements intra-vSSD wear-leveling and other FTL functionalities.

One of the key new abstractions provided by FlashBlox is that of a virtual SSD (vSSD) which can reduce tail latency. It uses dedicated flash hardware resources such as channels and dies that can be operated independently from each other. It can be created using the following simple API:

```
vssd_t AllocVirtualSSD(int isolationLevel,
                          int tptLevel, size_t capacity);
```

Instead of asking tenants to specify absolute numbers, FlashBlox enables them to create different types of vSSDs with different levels of isolation and throughput, and various storage capacity in GBs (see Table 3.1). These parameters are compatible with the performance and economic cost levels such as the ones [100, 84] advertised in DaaS services to ease usage and management. Tenants scale up capacity by creating multiple vSSDs of advertised sizes just as it is done in DaaS systems today. A vSSD is deallocated with `void DeallocVirtualSSD(vssd_t vSSD)`.

Channels, dies and planes are used for providing different levels of performance isolation. This brings significant performance benefits to multi-tenant scenarios because they can be operated independently from each other.

Higher levels of isolation have larger resource allocation granularities as channels are larger than dies. Therefore, channel-granular allocations can have higher internal fragmentation compared to die-granular allocations. However, this is less of a concern for FlashBlox's design for several reasons. First, capacities of flash's channels/dies/planes can

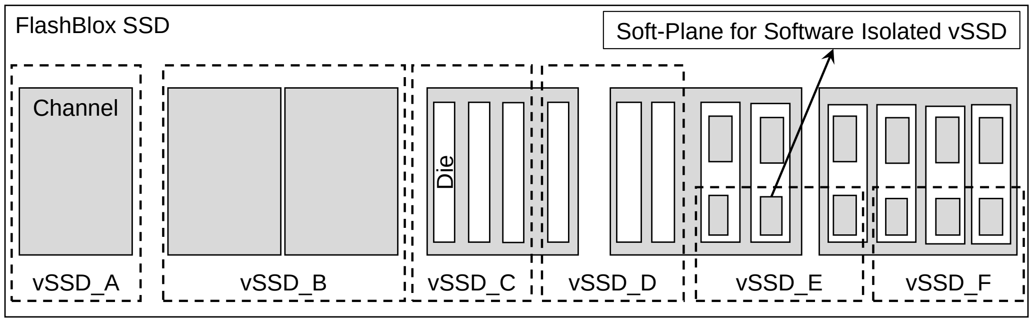


Figure 3.4: A FlashBlox SSD: vSSD_A and B use one and two channels respectively. vSSD_C and D use three dies each. vSSD_E, and F use three soft-planes each.

be modified by vendors according to the design specifications of data center operators. Moreover, the tiered storage offerings of DaaS systems [100, 84, 85] allows the flexibility for the cloud provider to choose sizes and granularities such that fragmentation is reduced. Finally, the differentiated isolation levels match with the well-known pay-as-you-go cost model for cloud platforms, in which better services are subject to increased pricing such the overhead is offset.

Beyond providing different levels of hardware isolation, FlashBlox has to overcome the unbalanced wear-leveling challenge to prolong the SSD lifetime. We describe the design of each vSSD type and its corresponding wear-leveling mechanism respectively as follows.

3.3.1 Channel Isolated Virtual SSDs

A vSSD with high isolation receives its own dedicated set of channels. For instance, the resource manager of an SSD with 16 channels can host up to 16 channel isolated vSSDs, each containing one or more channels inaccessible to any other vSSD. Figure 3.4 illustrates vSSD A and B that span one and two channels respectively.

Channel Allocation. The throughput level and target capacity determine the number of channels allocated to a channel isolated vSSD. FlashBlox allows the data center/PaaS administrator to implement the `size_t tptToChannel(int tptLevel)` function that maps between throughput levels and required number of channels. The number of channels allocated to the vSSD is, therefore, the maximum of `tptToChannel(tptLevel)` and $\lceil$`capacity` / `capacityPerChannel`$\rceil$.

Within a vSSD, the system stripes data across its allocated channels similar to traditional SSDs. This maximizes the peak throughput by operating on the channels in parallel. Thus, size of the super-block of vSSD_A in Figure 3.4 is half that of vSSD_B. Pages within the super-block are also striped across the channels similar to existing physical SSDs.

The hardware-level isolation present between the channels by virtue of hardware parallelism allows the read, program and erase operations on one vSSD to largely be unaffected by the operations on other vSSDs. Such an isolation enables latency sensitive applications to significantly reduce their tail latencies.

Compared to an SSD that stripes data from all applications across all channels, a vSSD (over fewer channels) delivers a portion of the SSD's all-channel bandwidth. Customers of DaaS systems are typically given and charged for a fixed bandwidth/IOPS level, and software rate-limiters actively keep their consumption in check. Thus, there is no loss of opportunity for not providing the peak-bandwidth capabilities for every vSSD.

Unbalanced Wear-Leveling Challenge. A significant side effect of channel isolation is the risk of uneven aging of the channels as different vSSDs may be written at different rates. Figure 3.2 shows how various storage workloads erase blocks at different rates indicating that channels pinned naively to vSSDs will age at different rates if left unchecked.

Such uneven aging may exhaust a channel's life long before other channels fail. Premature death of even a single channel would render significant capacity losses ($> 6\%$ in our SSD). Furthermore, premature death of a single channel leads to an opportunity loss of never being able to create a vSSD that spans all the 16 channels for the rest of the server's lifetime. Therefore, it is necessary to ensure that all the channels are aging at the same rate.

Inter-Channel Wear-Leveling. To ensure uniform aging of all channels, FlashBlox uses a simple yet effective wear-leveling scheme: *periodically, the channel that has incurred the maximum wearout thus far is swapped with the channel that has the minimum rate of wearout.*

A channel's wearout rate is the average rate at which it erased blocks since the last time the channel was swapped. This prevents the most-aged channels from seeing high wearout rates, thus intuitively extending their lifetime to match that of the other channels in the system. We analytically derive the minimum necessary frequency and present the design of the migration mechanism as follows.

Swap Frequency Analysis. Let σ_i denote the wear (total erase count of all the blocks till date) of the i^{th} channel. ξ = $\sigma_{max}/\sigma_{avg}$ denotes the wear imbalance[1] which must not exceed 1 + δ; where $\sigma_{max} = Max(\sigma_1, ..., \sigma_N)$, $\sigma_{avg} = Avg(\sigma_1, ..., \sigma_N)$, N is the total number of channels, and δ measures the imbalance.

When the device is new, it is obviously not possible to ensure that $\xi \leq 1 + \delta$ without aggressively swapping channels. On the other hand, it must be brought within bounds fairly early in the lifetime of the server (L = 150–250 weeks typical) such that all the channels are available for as much of the server's lifetime as possible.

SSDs are provisioned with a target erase workload and we analyze for the same – let's say M erases per week. We mathematically study the wear-imbalance vs. frequency of migration (f) tradeoff and show that manageable values of f can provide acceptable wear imbalance where ξ comes below $1 + \delta$ after αL weeks, where α is between 0 and 1.

Worst-case write workload for FlashBlox is when all the provisioned writes go to a single channel while the other channels are read-only workloads.[2] The assumption that a single channel's bandwidth can handle the entire provisioned bandwidth is valid for modern SSDs: most data center SSDs are provisioned with 3,000-10,000 erases to last 150–250 weeks. This implies that provisioned erase rate for a 1TB SSD is M=21–116 MBPS which is significantly lower than a channel's erase bandwidth (typically 64–128MBPS).

[1]The ratio of maximum to average is an effective way to quantify imbalance [101]. This is especially true in our case, as the lifetime of the new SSD will be determined by the maximum wearout of a single channel, whereas the lifetime of ideal wear-leveling is determined by the average wearout of all the channels. The ratio of maximum to average thus represents the loss of lifetime due to imperfect wear leveling.

[2]This worst-case is from a non-adversarial point of view. An adversary could change the vSSD write bandwidth at runtime such that no swapping strategy can keep up. But data center workloads are not adversarial and have predictable write patterns.

For an SSD with N channels, the wear imbalance of the ideal wear-leveling is ξ = 1, while the worst case workload for FlashBlox gives a ξ = N: $\sigma_{max}/\sigma_{avg} = M * time/(M * time/N) = N$ before any swaps. A simple swap strategy of cycling the write workload through the N channels (write workload spends 1/f weeks per channel) is analyzed. Let's assume that after K rounds of cycling through all the channels, $KN/f \geq \alpha L$ holds true – that is αL weeks have elapsed and ξ has become less than $1 + \delta$ and continues to remain there. At that very instant ξ equals 1. Therefore, σ_{max} = MK and σ_{avg} = MK, then after the next swap, σ_{max} = $MK + M$ and σ_{avg} = $MK + M/N$. In order to guarantee that the imbalance is always limited, we need:

$$\xi = \sigma_{max}/\sigma_{avg} = (MK + M)/(MK + M/N) \leq (1 + \delta)$$

This implies $K \geq (N - 1 - \delta)/(N\delta)$ which is upper bounded by $1/\delta$. Therefore, to guarantee that $\xi \leq (1 + \delta)$, it is enough to swap $NK = N/\delta$ times in the first αL weeks. This implies that, over a period of 150–250 weeks, if α were 0.9 then a swap must be performed once every 6–10 days ($= 1/f$) for a N=16 channel SSD and a $\delta = 0.1$. This also implies that $\frac{2}{16}^{th}$ of the SSD is erased to perform the swap once every 6–10 days, which is negligible compared to the 3,000–10,000 cycles that typical SSDs have. However, for realistic workloads that do not have such a skewed write pattern, swaps must be adaptively performed according to workload patterns to reduce the number of swaps needed.

Adaptive Migration Mechanism. A constant write rate of M is used for analysis purposes, but in real settings writes are bursty. High write rates trigger frequent swaps while swapping may not be needed as often during periods of low write rates. To achieve this, we maintain a counter per channel to represent the amount of space erased (MB) in each channel since the last swap. Once one of the counters goes beyond a certain threshold γ, a swap is performed and the counters are cleared. γ is set to the space the channel has erased from the constant-rate worst-case write workload between two swaps (i.e., M/f).

The rationale behind this mechanism is that the channels must always be positioned in a manner to be able to catch-up if the worst-case happens. FlashBlox then swaps the channel with σ_{max} with the channel with λ_{min} where λ_i denotes the erase-rate of the i^{th} channel and $\lambda_{min} = Min(\lambda_1, ..., \lambda_N)$.

FlashBlox uses an atomic block-swap mechanism to gradually migrate the candidate channels to their new locations without any application involvement. The migration happens in four steps. First, all the read, program and erase operations to the two erase-blocks being swapped are stopped (and queued). Second, the erase-blocks are read into a memory buffer (capacitor backed). Third, the erase-blocks are written to their new locations. Fourth, the operations stopped to these erase-blocks are dequeued. Note that only the IO operations for the candidate erase-blocks in the vSSD are queued and delayed, the IO requests for other blocks are still issued with higher priority to mitigate the migration overhead.

The migrations affect the throughput and latency of the vSSDs involved. However, they are rare (happen less than once in a month for real workloads) and take only 15 minutes to finish. As a future optimization, we wish to modify the DaaS system to perform the read operations on other replicas to further reduce the impact.

3.3.2 Die Isolated Virtual SSDs

For applications which can tolerate some interference (i.e., *medium* isolation) such as the non-premium cloud database offerings (e.g., Amazon's small database instance [100] and Azure's standard database service [102]), FlashBlox provides die-level isolation. The number of dies in such a vSSD is the maximum of `tptToDie(tptLevel)` and $\lceil$`capacity / capacityPerDie`$\rceil$. Their super-blocks and pages stripe across all the dies within the vSSD to maximize throughput. Figure 3.4 illustrates vSSD_C and D containing three dies each (vSSD_D has dies from different channels). These vSSDs, however, have weaker isolation guarantees since dies within a channel must share a bus.

The wear-leveling mechanism has to track wear at the die level as medium-level isolated vSSDs are to dies. Thus, the wear-leveling mechanism in FlashBlox is split into two sub-mechanisms: channel level and die level. The job of the channel-level wear-balancing mechanism is to ensure that all the channels are aging at roughly the same pace (see § 3.3.1). The job of the die-level wear-balancing mechanism is to ensure that all the dies within a channel are aging roughly at the same rate.

As discussed in § 3.3.1, an N channel SSD has to swap at least N/δ times to guarantee $\xi \leq (1 + \delta)$ within a target time period. This analysis also holds true for dies within a channel. For the SSDs today, in which each channel has 4 dies, FlashBlox has to swap dies in each channel 40 times in the worst case during the course of the SSD's lifetime or once every month or so.

As an optimization, we leverage the channel-level migration to opportunistically achieve the goal of die-level wear-leveling, based on the fact that dies have to migrate along with the channel-level migration. During each channel-level migration, the dies within the migrated channels with the largest wear are swapped with the dies that have the lowest write rate in the respective channels. Experiments with real workloads show that such a simple optimization can effectively provide satisfactory lifetime for SSDs.

3.3.3 Software Isolated Virtual SSDs

For applications that have even lower requirements of isolation like Azure's basic database service [102], it is natural to use the plane level isolation. However, flash planes within a die do not provide the same level of flexibility as channels and dies with respect to operating them independently from each other. Therefore, we turn to a software approach.

Each die is split into four regions of equal size called soft-planes by default, the size of each soft-plane is 4 GB in FlashBlox (other configurations are also supported). Each soft-plane obtains an equal share of the total number of blocks within a die. They also receive a *fair* share of the bandwidth of the die. The rationale behind this is to make it easier for data center/PaaS administrator to map the throughput levels required from tenants to quantified numbers of soft-planes.

vSSDs created using soft-planes are otherwise indistinguishable from traditional virtual

SSDs where software rate limiters are used to split an SSD across multiple tenants. Similar to such settings, we use an optimized version of the state-of-the-art token bucket rate-limiter [99, 22, 70] which has been widely used for Linux containers and Docker [98] to improve isolation and utilization at the same time.

The number of soft-planes used for creating these vSSDs is determined similarly to the previous cases: as the maximum of `tptToSoftPlane(tptLevel)` and $\lceil$`capacity / capacityPerSoftPlane`$\rceil$. Figure 3.4 illustrates vSSDs E and F that contain three soft-planes each. The super-block used by such vSSDs is simply striped across all the soft-planes used by the vSSD. We use such vSSDs as the baseline for our comparison of the channel and die isolated vSSDs.

The software mechanism allows the flash blocks of each vSSD to be trimmed in isolation, which can reduce the GC interference. However, it cannot address the situation where erase operations on one soft-planes occasionally block all the operations of other soft-planes on the shared die. Thus, such vSSDs can only provide software isolation which is lower than die-level isolation.

Besides these isolated vSSDs, FlashBlox also supports an **unisolated vSSD** model which is similar to software isolated vSSD, but a fair sharing mechanism is not used to isolate such vSSDs from each other. To guarantee the fairness between vSSDs in today's cloud platforms, software isolated vSSDs are enabled by default in FlashBlox to meet *low* isolation requirements.

For both software isolated and unisolated vSSDs, their wear-balancing strategy is kept the same rather than swapping soft-planes. The rationale for this is that isolation between soft-planes of a die is provided using software and not by pinning vSSDs to physical flash planes. Therefore, a more traditional wear-leveling mechanism of simply rotating blocks between soft-planes of a die is sufficient to ensure that the soft-planes within a die are all aging roughly at the same rate.

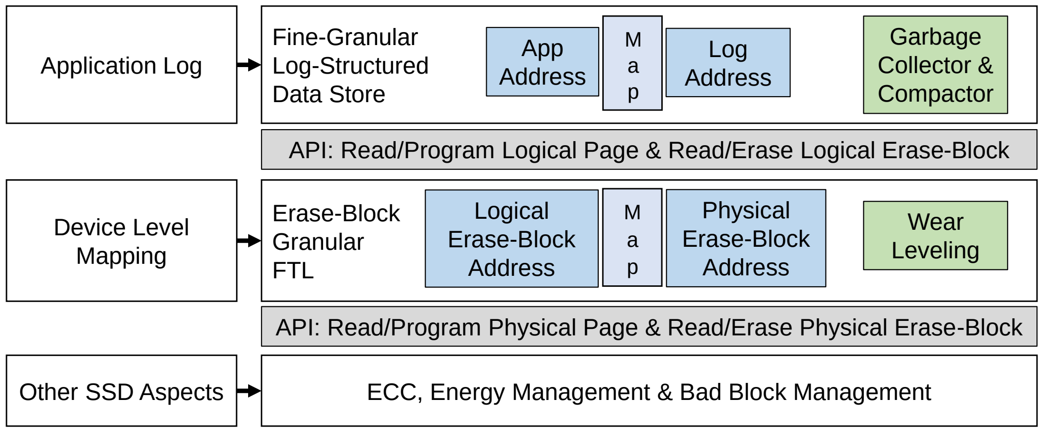

Application Log
Fine-Granular Log-Structured Data Store
App Address
Map
Log Address
Garbage Collector & Compactor
API: Read/Program Logical Page & Read/Erase Logical Erase-Block
Device Level Mapping
Erase-Block Granular FTL
Logical Erase-Block Address
Map
Physical Erase-Block Address
Wear Leveling
API: Read/Program Physical Page & Read/Erase Physical Erase-Block
Other SSD Aspects
ECC, Energy Management & Bad Block Management

Figure 3.5: Applications manage a fine-granular log-structured data store and align compaction units to erase-blocks in FlashBlox. A device level indirection layer is used to ensure all erase-blocks are aging at the same rate.

3.3.4 Intra Channel/Die Wear-Leveling

The goals of intra die wear-leveling are to ensure that the blocks in each die are aging at the same rate while enabling applications to access data efficiently by avoiding the pitfalls of multiple indirection layers and redundant functionalities across these layers [9, 41, 103, 104]. With both die-level (see § 3.3.3) and intra-die wear leveling mechanisms, FlashBlox inevitably achieves the goal of intra-channel wear-leveling as well: all the dies in each channel and all the blocks in each die age uniformly.

The intra-die wear-leveling in FlashBlox is illustrated in Figure 3.5. We leverage flash-friendly application or file system logic to perform GC and compaction, and simplify the device level mapping. We also leverage the drive's capabilities to manage bad blocks without having to burden applications with error correction, detection, and scrubbing. We base our design for intra-die wear-leveling on existing open SSDs [9, 25, 26].

3.4 FlashBlox Implementation

We implement FlashBlox using a CNEX SSD [75] which is an open-channel SSD [24] containing 1 TB Toshiba A19 flash memory and an open controller that allows physical resource access from the host. It has 16 channels, each channel has 4 dies, each die has 4 planes, each plane has 1024 blocks, each block has 256 pages with 16 KB page size. This hardware provides basic I/O control commands to issue read, write and erase operations against flash memory. We use a modified version of the CNEX firmware/driver stack that allows us to independently queue requests to each die. FlashBlox is implemented using the C programming language in 11,219 lines of code (LoC) layered on top of the CNEX stack.

We used FIO benchmarks [105] and 14 different workloads for the evaluation: six NoSQL workloads from the Yahoo Cloud Serving Benchmarks (YCSB) [106], four database workloads: TPC-C [107], TATP [108], TPC-B [109] and TPC-E [110], and four storage workload traces collected from a large data center/cloud provider.

YCSB is a framework for evaluating the performance of NoSQL stores. All of the six core workloads consisting of A, B, C, D, E and F are used for the evaluation. LevelDB [111] is modified to run using the vSSDs from FlashBlox with various isolation levels. The open-source SQL database Shore-MT [112] is modified to work over the vSSDs of FlashBlox. The table size of the four database workloads TPC-C, TATP, TPC-B and TPC-E range from 9 - 25 GB each.

Storage intensive and latency sensitive applications in the data centers of a large cloud provider are instrumented to collect traces for cloud storage, web search, PageRank and MapReduce workloads. These applications are the first-party customers of the storage IaaS system of the cloud provider.

3.5 Results and Analysis

3.5.1 Hardware Isolation vs. Software Isolation

In this experiment, the channel and die isolated vSSDs are evaluated against the software isolated vSSDs (the weighted fair sharing of storage bandwidth is enabled). We begin with a scenario of two LevelDB instances. They run on two vSSDs in three different settings each using a different isolation level: high, medium and low. The two instances run a YCSB workload each. The choice of YCSB is made for this experiment to show how removing

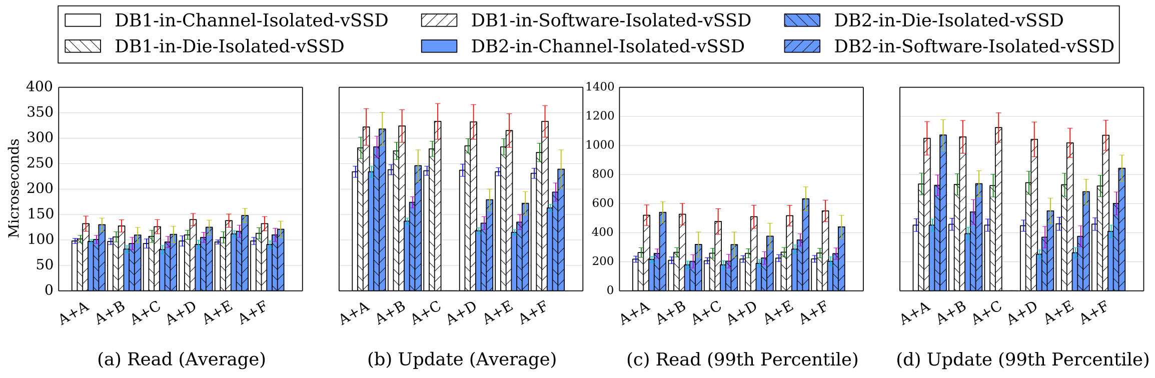


Figure 3.6: The average and 99th percentile latencies of LevelDB+YCSB workloads running at various levels of storage isolation. Compared to die and software isolated vSSDs, channel isolated vSSD reduces the average latency by 1.2x and 1.4x respectively, and decreases the 99th percentile latency by 1.2 - 1.7x and 1.9 - 2.6x respectively.

IO interference can improve the throughput and reduce latency for applications.

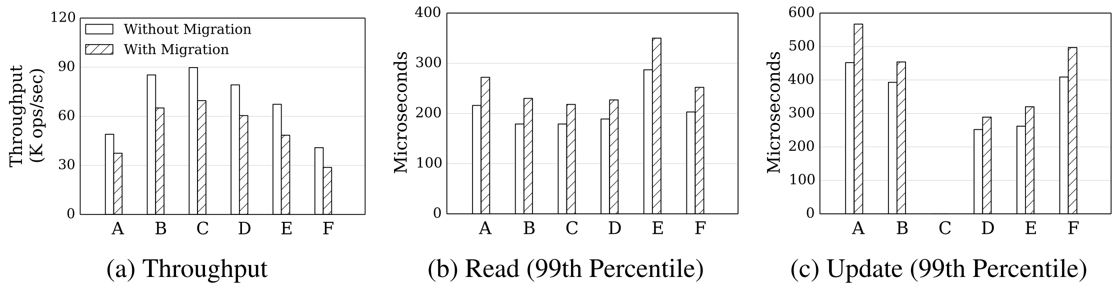
Each LevelDB instance is first populated with 32 GB of data and each key-value pair is 1 KB. 50 million CRUD (i.e., create, read, update and delete) operations are performed by the YCSB client threads against each LevelDB instance. The size of the database and number of operations are picked such that the GC is always triggered. YCSB C is read-only, thus we report only the results for read operations.

The total number of dies in each setting is the same. In the channel isolation case, two vSSDs are allocated from two different channels. In the die isolation case, both vSSDs share the channels but are isolated at the die level within the channel. In the software isolation case, both vSSDs are striped across all the dies in two channels.

As shown in Figure 3.6 (c) and Figure 3.6 (d), channel isolated vSSDs provide up to 1.7x lower tail latency compared to die isolated vSSDs and up to 2.6x lower tail latency compared to vSSDs that stripe data across all the dies akin to software isolated vSSDs whose operations are not fully isolated from each other.

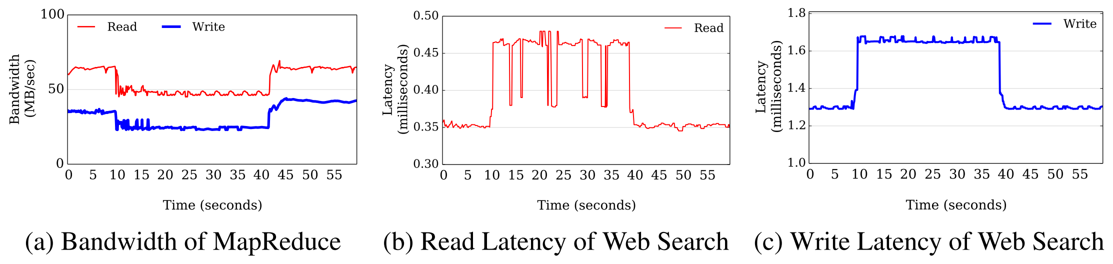
3.5.2 Migration Overhead

We first evaluate the overhead of the migration mechanism. We migrate one channel and measure the change in throughput and 99th percentile latency on a variety of YCSB work-



(a) Throughput (b) Read (99th Percentile) (c) Update (99th Percentile)

Figure 3.7: The impact of a channel migration on workloads. LevelDB's throughput falls by 33.8%, its tail percentile of reads and updates increase by 22.1% and 18.7% respectively.



(a) Bandwidth of MapReduce (b) Read Latency of Web Search (c) Write Latency of Web Search

Figure 3.8: The overhead of migrating 1GB of data as MapReduce and web search are running on the channels involved. MapReduce's bandwidth falls by up to 36.7% while web search's latency increases by up to 34.2%.

loads that are running on the channel.

The throughput of LevelDB running on that channel drops by no more than 33.8% while the tail latencies of reads and updates increase by up to 22.1% (see Figure 3.7). For simplicity, we show results for migrating 1 GB of the 64 GB channel. We use a single thread and the data moves at a rate of 78.9 MBPS. Moving all the 64 GB of data in a channel would take about 15 minutes.

The impact of migration on web search and MapReduce workloads is shown in Figure 3.8. During migration, the bandwidth of the MapReduce job decreases by 36.7%, the tail latencies of reads and writes of the web search increase by 34.2%. These performance slowdowns bring channel-isolation numbers on par with the software isolation. This implies that a 36.7% drop for 15 minutes when amortized over our recommended swap rate represents a 0.04% overall drop.

Table 3.2: Monte Carlo simulation (10K runs) of SSD lifetime with randomly sampled workloads on the channels.

#vSSD	NoSwap Lifetime (Years)		Ideal vs. FlashBlox Lifetime (Years)		Wear Imbalance	Swap Once in Days (Avg)
	99th	50th	99th	50th		
4	1.2	1.6	6.2/6.1	13.8/13.5	1.02	94
8	1.2	1.3	3.7/3.6	6.7/6.6	1.02	22
16	1.2	1.2	2.1/2.1	3.4/3.3	1.01	19

3.5.3 Migration Frequency Analysis

To evaluate FlashBlox's wear-leveling efficacy, we run a Monte Carlo simulation (10K runs) of the SSD lifetime. We create a various number of vSSDs and assign them uniformly at random to one of the fourteen workloads. The SSD is then simulated to end-of-life.

We report the 99th and 50th percentile lifetime of ideal SSD, SSD without swapping (NoSwap) and FlashBlox in Table 3.2. For the case of running 16 instances, 99% of the ideal SSDs last 2.1 years, and half of them can work for 3.4 years. With adaptive wear-leveling scheme, FlashBlox's lifetime is close to ideal and its wear imbalance is close to the ideal case. In the real world, where not all applications are adversarial, the swap frequency automatically increases.

3.6 Summary

In FlashBlox, we propose leveraging channel and die-level parallelism present in SSDs to provide hardware isolation for latency sensitive applications sharing an SSD. Furthermore, FlashBlox provides near-ideal lifetime despite the fact that individual applications write at different rates to their respective channels and dies. FlashBlox achieves this by migrating applications between channels and dies at coarse time granularities. Our experiments show that FlashBlox can improve throughput by 1.6x and reduce tail latency by up to 3.1x. We also show that migrations are rare for real world workloads and do not adversely impact applications' performance.

CHAPTER 4
FLASHMAP: UNIFYING INDIRECTION LAYERS ACROSS SYSTEM STACK

Applications can map data on SSDs into virtual memory to transparently scale beyond DRAM capacity, permitting them to leverage high SSD capacities with few code changes. Obtaining good performance for memory-mapped SSD content, however, is hard because the virtual memory layer, the file system and the flash translation layer (FTL) perform address translations, sanity and permission check independently from each other.

This chapter introduces FlashMap, an SSD interface that is optimized for memory-mapped SSD-files to further improve the performance of flash-based storage system. The proposed solution combines all the address translations into page tables that are used to index files and also to store the FTL-level mappings without altering the guarantees of the file system or the FTL. It uses the state in the OS memory manager and the page tables to perform sanity and permission checks respectively. By combining these layers, FlashMap reduces critical-path latency and improves DRAM caching efficiency. We find that this increases performance for applications by up to 3.32x compared to state-of-the-art SSD file-mapping mechanisms. Additionally, the latency of SSD accesses reduces by up to 53.2%.

4.1 Introduction

A growing number of data-intensive applications use solid state disks (SSDs) to bridge the capacity and performance gaps between main memory (DRAM) and magnetic disk drives (disks). SSDs provide up to 5000x more IOPS and up to 100x better latency than disks [1]. SSDs provide up to 20 TB capacity per rack-unit (RU) [113], whereas DRAM scales only to a few hundred GBs per RU [114, 115]. SSDs are used today as a fast storage medium to replace or augment disks.

An emerging approach to using SSDs treats them as a slower form of non-volatile memory. For example, NoSQL databases like MongoDB [5, 6], LMDB [7] (backend for OpenLDAP) and others [27, 28, 29] which are widely deployed [30] use SSDs via a memory-mapped file interface. There are three advantages to this approach. First, the virtual memory interface eases development. For example, MongoDB uses TCMalloc [31] to manage SSD-file backed memory to create data structures like its B-tree index, and for using the Boost template library. Second, SSDs are automatically tiered under DRAM by the OS's memory manager and finally, memory that is backed by a file enables durability for data. Such hybrid memory systems have also been proposed in the academic community [32, 33, 34, 35, 36].

Using SSDs in this manner, unfortunately, is inefficient as there are three layers with redundant functionalities between the application and NAND-Flash. The first layer, memory-level indirection, involves page table translations and sanity checks by the OS memory manager. The second layer, storage-level indirection, involves converting file offsets to blocks on the SSD and permission checks by the file system. The final one, device-level indirection, is for the flash translation layer (FTL) of the SSD. Redundant indirections and checks not only increase the latency of NAND-Flash accesses, but also affect performance by requiring precious DRAM space to cache indirection data across all the three layers.

In this dissertation, we present FlashMap, a holistic SSD design that combines memory, storage, and device-level indirections and checks into one level. This is a challenging problem because page table pages and OS memory manager state that form the memory-level indirection are process-specific, private, and non-shared resources while direct/indirect blocks (file index) that form the storage-level indirection in a file system, and the FTL that converts the logical block addresses to physical block addresses are shared resources. FlashMap introduces the following three new techniques to combine these layers without losing their functionality:

- FlashMap redesigns a file as a contiguous global virtual memory region accessible to all eligible processes. It uses page table pages as the indirect block index for such files where these page table pages are shared across processes mapping the same file.

- FlashMap enables sharing of page table pages across processes while preserving semantics of virtual memory protections and file system permissions by creating private page table pages only on demand.
- FlashMap introduces a new SSD interface with a sparse address space to enable storing of the FTL's mappings inside page table pages.

More importantly, FlashMap preserves the guarantees of all the layers in spite of combining them into virtual memory. We implement FlashMap in Linux for EXT4 on top of a functional SSD-emulator with the new interface proposed. Experimental results demonstrate that data intensive applications like NoSQL stores (Redis [116]), SQL databases (Shore-MT [112]) and graph analytic software (GraphChi [117]) obtain up to 3.32x better performance and up to 53.2% less latency.

4.2 Motivation

To meet the high-capacity needs of data-intensive applications, system-designers typically do one of two things. They either scale up the amount of DRAM in a single system [118, 119] or scale out the application and utilize the collective DRAM of a cluster [120, 121, 122]. When more capacity is needed, SSDs are useful for scale-up scenarios [123, 124, 125, 126, 127] and for scale-out scenarios [128, 129, 130]. Adding SSDs not only improves performance normalized for cost but also improves the absolute capacity.

Today SSDs scale up to 10 TB per system slot [1] (PCIe slot) while DRAM scales only to 64 GB per slot (DIMM slot). Even though systems have more DIMM slots than PCIe slots, one would require an impractically high (160) number of DIMM slots to match per-slot SSD density. SSDs provide up to a million IOPS at less than 100 μsec and lack seek latencies. Such performance characteristics make SSDs more similar to DRAM than to disks. Moreover, SSDs are non-volatile. Therefore, data-intensive applications are using SSDs as

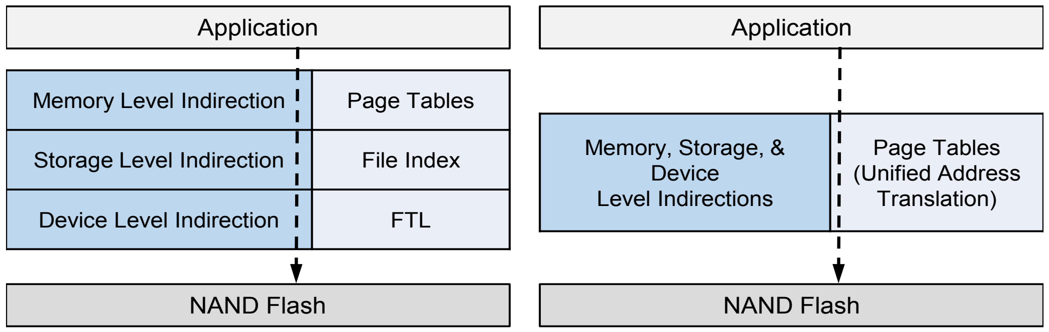


Figure 4.1: Comparison of (a) conventional memory-mapped SSD-file's IO stack and (b) FlashMap that combines all the address translations for mapping files into page tables.

slow, but high-capacity non-volatile memory [5, 7, 29, 32, 34] via memory-mapped files. Such an approach has the following three advantages.

First, virtual memory interface helps existing in-memory applications adopt SSDs with only a few code changes. For example, we find that less than 1% of the code has to be changed in Redis (an in-memory NoSQL store built for DRAM) to use SSD-backed memory instead of DRAM. On the other hand, more than 10% of the code has to be modified if SSDs are used via read/write system calls. Prior work [32] is also evident from the fact that widely deployed databases like MongoDB map SSD-files and build data structures such as BTree indexes using TCMalloc [31] and Boost [131].

Second, memory-mapped access ensures that hot data is automatically cached in DRAM by the OS and is directly available to applications via `load/store` instructions. To get such a benefit without memory-mapping, an application would have to design and implement custom tiering of data between DRAM and SSD which can take months to years depending on application complexity.

Finally, file-backed memory as opposed to anonymously-mapped memory (SSD as a swap space) allows applications to store data in the file durably and exploit other file system features such as atomicity (via transactions and journaling), backup, space management, naming, and sharing.

Unfortunately, existing OSes and SSDs are not optimized for this style of using the SSD. There are three distinct software layers with separate indirections between the application and NAND-Flash. The separation of the virtual memory, the file system, and the FTL as shown in Figure 4.1(a) reduces throughput and increases latency. We present a holistic SSD designed for memory-mapped SSD-files that combines these layers.

4.2.1 Overhead from Redundant Software

To service a page fault in a memory mapped region from a file on an SSD, three types of address translations are required. First, the CPU traverses the page tables (memory-level indirection) to trigger the page fault. Later, the file system uses indirect blocks to translate the faulting file offset to a block on the SSD (storage-level indirection). Finally, the FTL converts this block address to an actual NAND-Flash level address (device-level indirection). Multiple layers of indirection not only increase the latency but also decrease the performance of applications. Each layer wastes precious DRAM space for caching address translation data. The latency increases further if the required address translation is not cached in DRAM. Locking all the address translation data in DRAM is not a feasible option as one would require as much as 60 GB of DRAM for a 10 TB SSD[1].

Even if the address translation data is cached in DRAM, the software latency is still significant (5–15 μsec in each layer) as each layer performs other expensive operations. For example, the memory manager has to check if the faulting address is in an allocated range, the file system has to check if the process has access to the file (checks that can be efficiently enforced using permission bits in page tables that are always enforced by the CPU) and FTLs with sparse address spaces [132] have to check allocation boundaries (checks that can be efficiently performed by the memory manager itself). There is a need for a new SSD design that is optimized for memory-mapped SSD-files, one that uses a single software layer (virtual memory), one address translation, one permission check (in page tables), and one sanity check (in memory manager).

[1]A minimum of eight bytes per indirection layer per 4 KB page

4.2.2 Challenges for Combining Indirection Layers

Memory-level indirection. SSD-files have to be mapped with small pages (4 KB) as large pages are detrimental to the performance of SSDs. An x86_64 CPU provides memory usage (read/fault and write/dirty) information only at the granularity of a page. Therefore, smaller page sizes are better for reducing the read/write traffic to the SSD. For example, our enterprise class SSD provides 700K 4KB random reads per second, while it provides only 1,300 2MB random reads per second. Thus using a 4KB page size means that the size of the page tables would be about 20 GB for a 10 TB dataset. While keeping page table pages only for the pages in DRAM can reduce the space required, it does not reduce the software-latency of handing a page fault which is dominated by the layers below the memory manager. To reduce this latency, we propose performing all the address translations with *page table pages* as they are an x86_64 necessity for file-mapping and cannot be changed or removed. Moreover, the virtual memory protection bits can be exploited for *all* permission checks and the memory allocation metadata in the OS memory manager can be exploited to perform *all* sanity checks.

Storage-level indirection. In a file system, for typical block sizes (2–8KB), the indirection layer requires 10–40 GB of space for a 10 TB SSD. Larger blocks, unfortunately, decrease DRAM caching efficiency and increase the traffic from/to the SSD. Extent-based file indexes such as the one used in EXT4 [133] can reduce this overhead. However, using such an index does not remove all the file system overhead from the IO-path. It is well known that permission checks of file systems increase latency in the IO-path by 10–15μs [134, 26]. A combined memory and storage-level indirection layer would not only eliminate a level of indirection but would also perform all the necessary checks efficiently by using the protection bits in the page tables.

Device-level indirection. NAND-Flash supports three operations – read, write, and erase. Reads can be performed at a granularity of a page (4KB), which can be written only after they are erased. Unfortunately, erases can be performed only at a large granularity of a block (eight or more pages at a time). Moreover, each block is rated only for a few thousand erases and therefore it is vital for the blocks to age uniformly. SSDs employ a log-structured data store with garbage collection (GC) [13] using indirections in the FTL for out-of-place writes and ensuring uniform wear. To improve performance and lifetime, high-performance SSDs implement such logs by employing a fine-granular and fully-associative page-level index [132, 135]. Such an index at a granularity of a page (4KB) requires more than 20 GB of space for a 10 TB SSD.

Traditionally, FTLs have cached their mappings in embedded SRAM/DRAM to provide predictable performance. Unfortunately, it is not possible to provision large amounts of RAM inside SSDs. Therefore, high-capacity SSDs store the mappings in the host [132] where DRAM scales better. FlashMap leverages this SSD design pattern to combine the FTL mappings with indirections in the higher layers.

Combining page tables with storage and device-level indirections, however, is challenging because page table *pages* are process specific and private entities while the remaining indirections are system-wide entities that are shared by all processes. Furthermore, page table pages cannot be shared frivolously across processes because it may lead to false sharing of memory and violate permissions and protections. To address these problems, FlashMap introduces a new virtual memory design where the page table pages needed for mapping a file belong to the file system and are system wide resources shared across processes mapping the same file. FlashMap enforces file permissions and virtual memory protections as required at a low-overhead by creating process-specific private page table pages only on demand. This design helps FlashMap unify the memory, storage and device interfaces (Figure 4.1(b)) without changing *any* guarantees from virtual memory, file system, or FTL.

4.3 FlashMap Design

To combine the memory and storage-level indirections, we re-imagine a file as a contiguous virtual memory region of an abstract process. The region is indexed by an x86_64 style

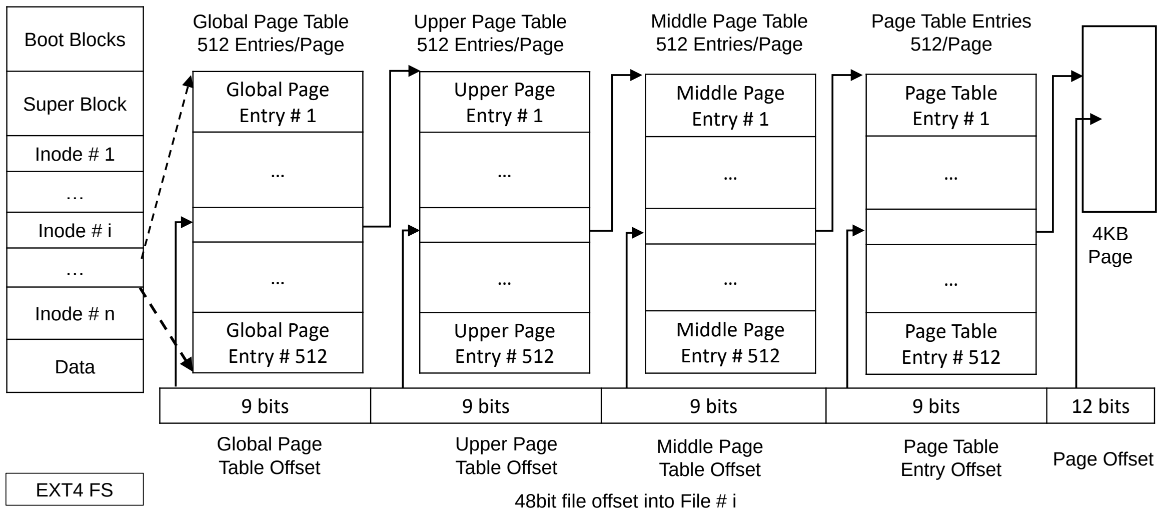


Figure 4.2: FlashMap uses a page table design for indexing files.

four-level page table as opposed to the traditional direct/indirect block/extent representations that file systems use. Figure 4.2 illustrates the indirect blocks in our file design – *the rest of the file system, however, remains unaltered.* The 48-bit physical frame numbers (PFN) in this page table are the block pointers that the file system uses to index the file. Since we treat a file as a region of memory of an abstract process, we will use page and block interchangeably. We will refer to the file index as shared page tables.

Such files can be accessed via the `POSIX` file API without any application changes. Most file systems are designed to abstract away the indexing from the API and the rest of the file system. We leverage this abstraction to preserve the rest of the file system and `POSIX` API. When mapping such files, however, *necessary shared page table pages have to be borrowed by the process* in contrast to traditional file-mapping and memory management techniques where private page table pages are created for the process.

At a first glance, this problem can be solved by grafting the process's page table with as much of the shared page table pages as possible. However, the solution is not so simple. If two processes map the same file with different permissions (e.g., READ_ONLY vs. READ_WRITE), then sharing page table pages by frivolous grafting can violate file system permissions. To address these problems in a systematic manner, we introduce the notion of

Semi-Virtual Memory which is a mechanism to share some of the shared page table pages across processes that map the same file with different file-level permissions.

4.3.1 Preserving File System Permissions

Semi-Virtual Memory is a method to share only the leaf-level page table pages (LL-PTP) that contain page table entries (PTE) across processes that map the same file. When a process maps a file, only the LL-PTPs of the shared page table of the file are borrowed and grafted into the process's page table. The rest of the page table pages (page directory pages) needed for the mapping are created for the process afresh and deleted when the process unmaps the file.

The permissions set in the higher level page table entries (page global, middle, and upper directory entries) override the permissions set at the page table entries in x86_64. Therefore, private copies of higher-level page table pages can be exploited to implement custom file-level permissions during mapping. It helps to think of this memory virtualization at the granularity of a file rather than a page, hence the name Semi-Virtual Memory. Not sharing higher-level page tables would increase the memory overhead of page tables by only a negligible amount. The branching factor of x86_64 page tables is 512 (512 entries per 4KB page), and the overhead is less than 0.5%. Figure 4.3 shows how only the LL-PTP of the shared page tables are borrowed from the file system when a file is mapped into a process. Rest of the higher-level page table pages are created for the process like the way they are in traditional OSes.

FlashMap avoids false sharing of file permissions by design. Two or more files are mapped to the same process such that the page table entries required for mapping these files are never on the same LL-PTP. This requires that file boundaries be aligned to 2MB (512x4KB span) in virtual memory space. FlashMap is designed for x86_64 where there is ample virtual memory available and therefore, we do not see this requirement as a major overhead. This design helps separate permissions for each file mapped to the same process.

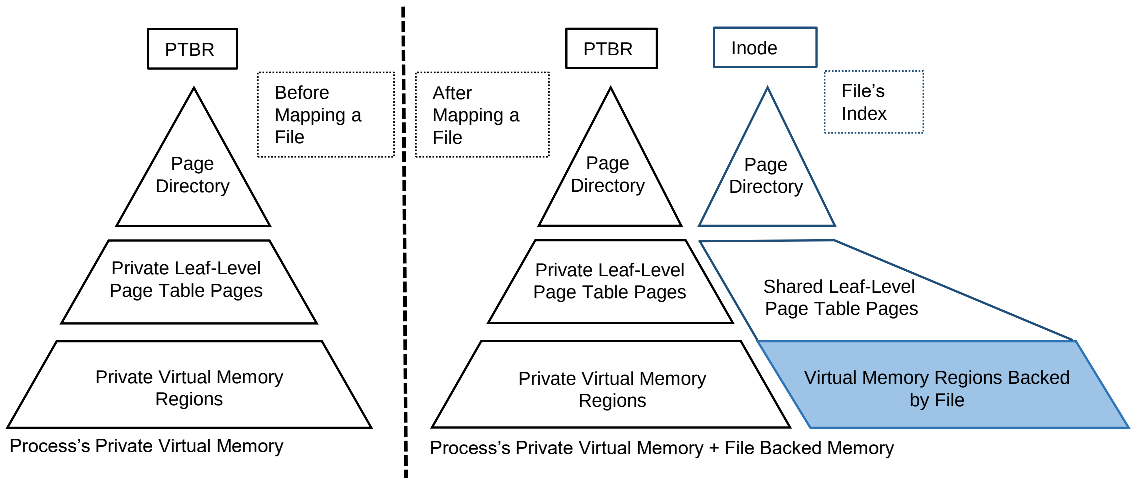

PTBR
Before Mapping a File
Page Directory
Private Leaf-Level Page Table Pages
Private Virtual Memory Regions
Process's Private Virtual Memory
After Mapping a File
PTBR
Inode
File's Index
Page Directory
Page Directory
Private Leaf-Level Page Table Pages
Shared Leaf-Level Page Table Pages
Private Virtual Memory Regions
Virtual Memory Regions Backed by File
Process's Private Virtual Memory + File Backed Memory

Figure 4.3: File brings leaf-level page tables with itself to a process that maps it. Higher-level page table pages are not shared, they are created on-demand for mapping the file.

This enforcement does not violate the `POSIX` compliance of `mmap`. `POSIX mmap` specifies that the OS may pick an address of its choosing to map the file if the address requested by the caller is not available. However, as this scheme itself is deterministic, it is still possible for a process to map a file to the same address across reboots.

In traditional OSes, the separation of the memory and storage-level indexes meant that the memory manager and the file system needed to interact with each other only for data. In a system like FlashMap, they have to additionally collaborate to manage the shared LL-PTPs of the files that are currently mapped. PTE behavior must remain the same for user space in spite of this dual role of LL-PTPs

4.3.2 Preserving PTE Behavior

FlashMap overloads the PTE. When a page of a file is in DRAM, the corresponding PTE in the LL-PTP is marked as resident and contains the address of the physical page in DRAM where the page resides. When the page is not cached in DRAM, the PTE in the shared LL-PTP is marked as not-resident, and contains the address of the page on the SSD.

The SSD-location of the page must be stored elsewhere while the page is cached in DRAM. We design an auxiliary index to store the SSD-locations of all the pages cached

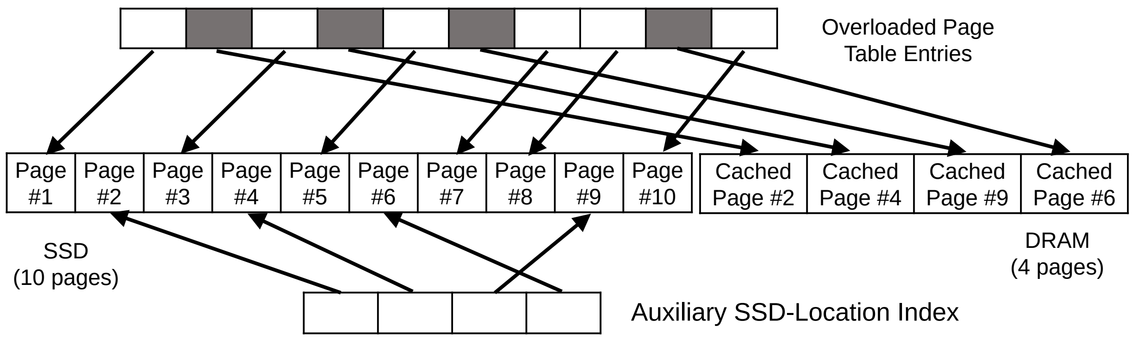


Figure 4.4: Page table entries are overloaded to store both SSD and DRAM locations of pages. When a page table entry stores the DRAM location of a cached page, the corresponding auxiliary SSD-location entry remembers it's SSD-location.

in DRAM. The auxiliary index is implemented using a simple one-to-one correspondence between DRAM pages and the SSD-Location of the block that the DRAM page may hold – a simple array of 8 byte values. It must be noted that the size of this array is the same as the number of pages in DRAM and therefore is not significant. For example, for a typical server with 64 GB DRAM, this auxiliary index would require only 128MB and can be stored in DRAM – an overhead of less than 0.25%. Figure 4.4 demonstrates how the overloaded page table entries and auxiliary SSD-location index remember the location of all the pages (on SSD or cached in DRAM).

While Semi-Virtual Memory preserves the file-level permissions, it does not provide one of the crucial properties of virtual memory – page-granular memory protection via `mprotect`. The sharing of LL-PTPs between processes means that the protection status of individual pages is also shared. This can violate the semantics of memory protection.

4.3.3 Preserving Memory Protection Behavior

To preserve the `POSIX` memory protection (`mprotect`) behavior, FlashMap simply disables Semi-Virtual Memory for the LL-PTPs of *only the virtual memory ranges that require custom access permissions only for the requesting process.* If a process requires custom memory protection for a single page, then the OS creates a private LL-PTP on demand for the encompassing virtual memory range that contains this page – the minimum size of

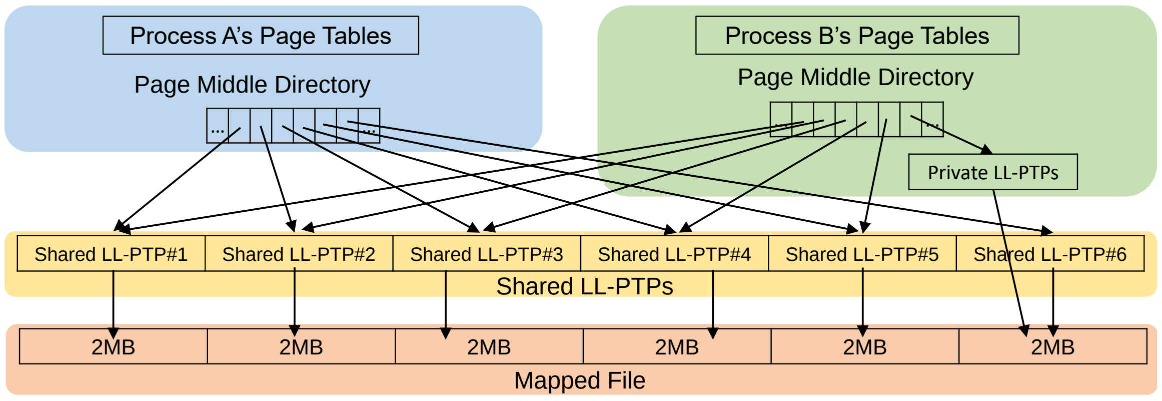


Figure 4.5: Processes A and B map the same file. However, process B has custom memory protections for a small memory region with private leaf-level page-table pages (LL-PTP).

such a region would be the span of an LL-PTP in x86_64 (2MB = 512x4KB span). These regions of memory are managed similarly to shared memory in operating systems where the memory-level and storage-level indexes are separate. We call these regions "saturated virtual memory". We believe that saturated virtual memory regions will not increase the memory overhead of address translation significantly. Basu et al. [115] report that for many popular memory-intensive applications, less than 1% of memory requires custom per-page protections. Moreover, our focus is on high-performance, in-memory applications where sharing files across processes with differing protections is a rare scenario.

Saturated virtual memory is depicted in Figure 4.5. Processes A and B map the same file, however, Process B requires custom protection status for a page. FlashMap creates a private LL-PTP for the encompassing 2MB region to enforce the protection.

4.3.4 Preserving the FTL Properties

FlashMap only changes the interface between the OS and the SSD. The rest of the SSD remains intact. FlashMap requires an SSD that can store only the mappings of the blocks of a file on the host in the shared LL-PTPs. The rest of the mappings – of file system metadata and other metadata – are managed by the SSD itself. Data dominates metadata in our applications which usually map large files to memory. Therefore, having separate translations inside the SSD for metadata of the file system does not add significant overhead.

We propose a virtualized SSD architecture, where the SSD and the file system share indirections *only* for data blocks of files. For each file, FlashMap creates a virtual SSD (48-bit address space) whose blocks have a one-to-one correspondence with the blocks of the file. The mappings of this address space are stored by FlashMap in shared LL-PTPs. Most of the space and IOPS of an SSD, in our scenario, will be spent towards data in large memory-mapped files, and very little on file system metadata (boot blocks, super blocks, inodes and etc.). A one-to-one mapping between FlashMap files and virtual SSDs allows the SSD driver to have a trivial way to store performance-critical mappings in the shared page table without affecting the metadata design of a file system.

Virtual SSDs are carved out of a larger virtual address space that the SSD driver implements. Similar to several high-performance SSDs [132], we design an SSD with a 64-bit address space and carve out smaller virtual SSDs with 48-bit contiguous address spaces from it. The first 48-bits worth of this address space is used for implementing a virtual SSD whose FTL indirections are managed by the SSD driver itself. The file system can use this as a traditional block device for storing non-performance-critical data: including metadata and the on-SSD copy of the shared LL-PTPs. We call this the proto-SSD.

For each 48-bit address range, the file system must remember its allocation status so that they can be recycled and reused as files are created and deleted. The file system maintains a simple one-to-one table to represent this information using some space in the proto-SSD. A 64-bit address space allows 2^{16} 48-bit contiguous address spaces. Therefore, this table requires only tens of megabytes of space. Directories and smaller files are also stored on the proto-SSD.

4.4 FlashMap Implementation

We modify EXT4 and the memory manager in Linux to combine their memory and storage level indexes using Semi-Virtual Memory in combination with overloaded page tables, auxiliary SSD-Location index, and saturated virtual memory. We also implement a wrapper that can be used to convert any legacy SSD into one that provides virtual SSDs and a proto SSD. FlashMap is the combination of these two systems.

Modified EXT4 Index. We implement the shared page tables as the file index in EXT4 by replacing the default indirect (single/double/triple) block representation as depicted in Figure 4.2. We populate the page table of each file as the file grows contiguously in a virtual SSD's address space. The index implementation is abstracted from the rest of the file system in such a way that traditional POSIX file APIs, page replacement, and other file system entities work seamlessly.

Augmented Memory Manager. We implement a special memory manager helper kernel module that manages physical memory for all the virtual memory regions that map FlashMap files. This contains a page fault handler that brings the relevant data/page tables into DRAM. The module also manages the LL-PTPs. It also maintains the auxiliary SSD-Location index. This module interacts with Linux's page replacement scheme to identify least recently used pages [136] and LL-PTPs of the file that can be sent back to the SSD.

The memory manager also implements the modified versions of `mmap`, `munmap`, and `mprotect` by intercepting these system calls. It implements semi-virtual memory and saturated virtual memory. We use a red-black tree to remember the virtual memory regions that require saturated virtual memory and handle the page faults in these regions as if the memory and storage-level indirections are separate. Finally, this module implements the functions required by the SSD for atomically updating and reading the shared LL-PTPs and the auxiliary SSD-index.

4.5 Results and Analysis

4.5.1 Experimental Methodology

We compare FlashMap with the following systems:

Unoptimized Software: Unmodified `mmap` is used to map a file on a unmodified EXT4 file system without extents on the proto-SSD. Private page tables are created by `mmap`. The

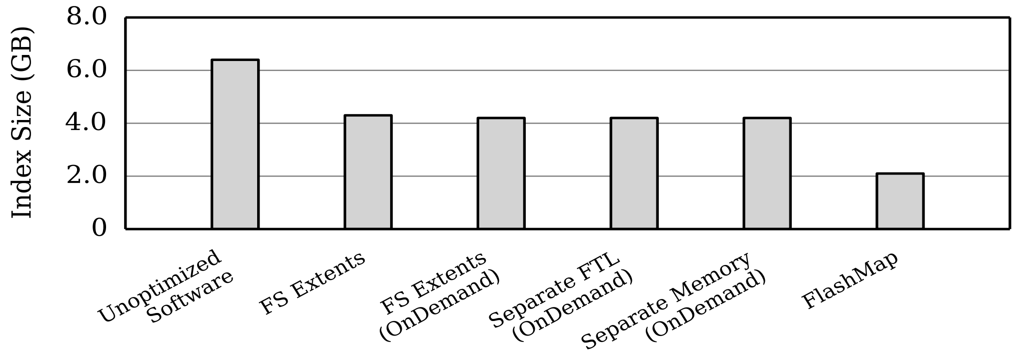


Figure 4.6: Index size for 1 TB SSD.

proto-SSD has its FTL locked in DRAM similar to high-performance FTLs such as Fusion-io's ioMemory [132, 135]. Rest of the indirections are fetched on demand.

FS Extents: This is the same software and hardware stack as described above but with file system extents (128MB) enabled in EXT4. An improved version is **FS Extents (OnDemand)** in which the FTL mappings are fetched to DRAM from flash on-demand similar to DFTL [39].

Separate FTL (OnDemand): Page tables and the file system index are combined but the FTL remains separate. All the indirections are, however, fetched on demand.

Separate Memory (OnDemand): The file system and the FTL are combined similar to existing systems such as Nameless Writes [37] and DFS [38] where the memory-level indirections remain separate. However, all the indirections are fetched on demand to DRAM.

The physical machine used for experiments has two Intel Xeon processors each with 6 cores running at 2.1 GHz and 64 GB DRAM. Samsung 840 EVO SSDs are used for experimentation. For the experiments in Section 4.5.3, we emulate high-end SSDs (e.g., PCM-based SSDs) using DRAM with added latency.

4.5.2 Benefits from Saving DRAM

We first examine the total size of indirections in each system per TB of SSD used. The results are shown in Figure 4.6. FlashMap requires close to 2 GB while unoptimized software

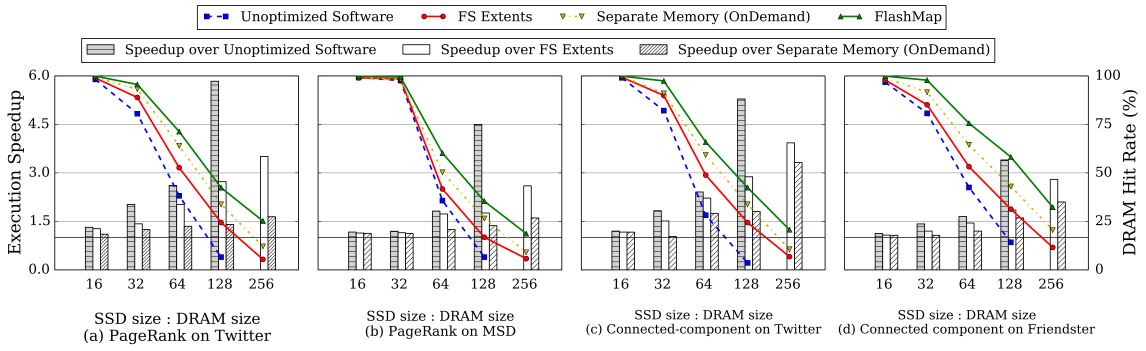


Figure 4.7: Improvements for analytics on Twitter, Friendster and MSD dataset, with varied DRAM size. Compared to Separate Memory (OnDemand), FlashMap performs 1.15–1.64x better for PageRank, and up to 3.32x better for the connected component labeling.

requires more than 6 GB of indirections per TB of SSD. For 10 TB SSDs available today, unmodified software would require more than 60 GB of metadata. While more efficient than Unoptimized Software, FS Extents and Separate Memory (OnDemand) are still more than twice as expensive as FlahsMap in terms of metadata overhead. This indicates that for a given working set, other methods would require 2-3x more amount of DRAM to cache all the indirections needed for the data in a given working set. For large working sets, the DRAM savings translate to higher performance.

We now turn our attention towards more computationally intensive applications like PageRank and connected-component labeling inside graphs. The aim of these experiments is to demonstrate that the additional DRAM provides performance benefits not just for IOPS driven applications, but also for computationally intensive applications. We find that FlashMap reduces the execution time of these memory intensive workloads significantly.

GraphChi [117] is a graph computation toolkit that enables analysis of large graphs from social networks and genomics on a single PC. GraphChi is primarily a disk based graph computation mechanism that makes efficient usage of available memory. It partitions the graph such that each partition can fit in memory while it can work on it at memory speeds. We modify GraphChi to use SSDs as memory and run graph algorithms for various DRAM sizes. We increase the memory budget of GraphChi beyond the amount of DRAM available in the system. This means that the entire SSD is the working set of GraphChi. We run two graph computational algorithms on different graphs with various memory budgets to demonstrate the benefits of FlashMap over existing techniques.

First, we use GraphChi to find the PageRank of a real social networking graph. The graphs we use in our experiments are from Twitter (61.5 million vertices and 1.5 billion edges) [137, 138], Friendster (65.6 million vertices and 1.8 billion edges) and MSD (1 billion records of songs' metadata). Figure 4.7 (a) and (b) demonstrate that FlashMap obtains 1.27–3.51x, 1.10–1.65x, 1.31–5.83x speedup in execution time of the PageRank algorithm than FS Extents, Separate Memory (OnDemand) and Unoptimized Software respectively, across different DRAM sizes.

The execution times of the PageRank algorithm on MSD at 1:64 DRAM:SSD ratio are 589.85 and 471.88 seconds respectively for Separate Memory (OnDemand) and FlashMap. For GraphChi, the working set spans the entire SSD. As the size of the SSD increases, the effective amount of DRAM available for GraphChi runtime decreases. It is natural that this increases the execution time. However, by combining indirections, FlashMap helps speed up the graph computation by freeing up memory. As shown in Figure 4.7 (b), the DRAM hit rates are 18.7% and 9.1% for FlashMap and Separate Memory (OnDemand) respectively as DRAM:SSD ratio is 1:256. We next evaluate the FlashMap with the `connected-component labeling` algorithm in GraphChi. We run it on the Twitter and Friendster dataset and the results are shown in Figure 4.7 (c) and (d): FlashMap outperforms FS Extents by up to 3.93x, and Separate Memory (OnDemand) by up to 3.32x.

4.5.3 Latency Benefits

The aim of the experiments in this section is to demonstrate that FlashMap also brings benefits for high-end SSDs with much lower device latencies, as it performs single address translation, single sanity and permission check in the critical path to reduce latency.

PCM-based SSDs and PCM-based caches on NAND-Flash based SSDs are on the horizon [134, 139, 140]. With much smaller PCM access latencies, FlashMap can provide tan-

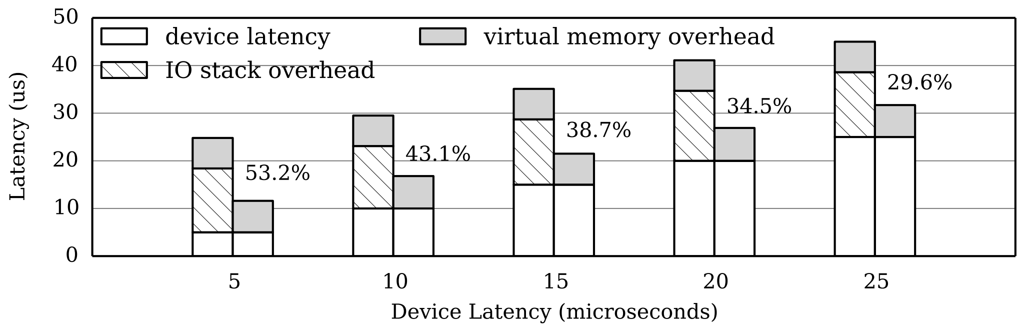


Figure 4.8: For faster SSDs, FlashMap provides tangible improvements in the latency over Unoptimized Software.

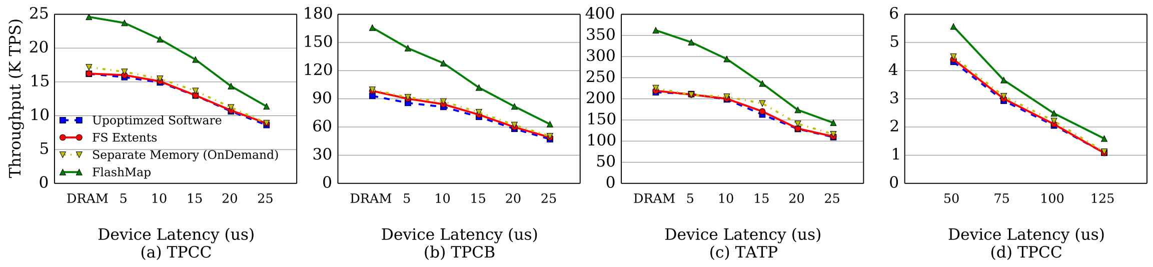


Figure 4.9: For faster SSDs, FlashMap provides up to 1.78x improvements on throughput over Unoptimized Software with TPCC, TPCB and TATP benchmarks. As for Flash with 100 μs device latency, FlashMap still performs 1.21x more TPS than others.

gible latency improvements. We emulate such SSDs with various latencies using DRAM and show how the software overhead decreases because of FlashMap. 4KB pages from the SSD are accessed at random in a memory-mapped file and the average latencies are reported. The results are presented in Figure 4.8 and it shows how FlashMap can improve latency by up to 53.2% for faster SSDs. Note that these SSDs do not have a device-level indirection layer, therefore these benefits are purely from combining the translations and checks in file systems with those in virtual memory.

We further break down the software overheads of FlashMap by intercepting and timing each indirection layer. Table 4.1 lists the overhead of the key operations in FlashMap and we find that these are comparable to the latencies for each component in unmodified Linux.

Furthermore, we investigate how the latency benefits of FlashMap improves the performance of applications with concurrency. Faster access to data often translates to locks

Table 4.1: FlashMap's Overhead

Overhead Source	Average Latency (μsec)
Walking the page table	0.85
Sanity checks	2.49
Updating memory manager state	1.66
Context switches	0.75

Table 4.2: Cost-effectiveness of FlashMap for 1 TB workload sizes, compared with the ideal large DRAM-only system.

Application	Benchmark	Settings	Bottleneck	Performance/$	Expected Life
NoSQL Store (Redis)	YCSB	DRAM vs SATA SSD (1GigE switch)	Wide-area latency & router throughput	26.6x	33.2 years
		DRAM vs PCIe SSD (10GigE switch)	Wide-area latency & router throughput	11.1x	10.8 years
SQL Database (MySQL)	TPCC	DRAM vs PCIe SSD	Concurrency	1.27x	3.8 years
Graph Engine (GraphChi)	PageRank	DRAM vs SATA SSD	Memory bandwidth	1.89x	2.9 years

being released faster in transactional applications and this translates to higher application-level throughput.

We modify a widely used database manager (Shore-MT [112]) to `mmap` its database and log files with various techniques (i.e., FlashMap, FS Extents, Separate Memory (On-Demand) and Unoptimized Software). We use TPC-C [107], TPC-B [109] and TATP [108] benchmark in our experiments. Their dataset sizes are 32-48 GB and the footprint of address translation data is small. The memory configured for the database manager is 6 GB. As shown in Figure 4.9, for SSDs with low latency, FlashMap provides up to 1.78x more throughput because of its latency reductions. For SSDs with higher hardware latency, FlashMap provides more than 1.21x improvement on throughput over Separate Memory (OnDemand), as the latency reduction (even small) can relieve the lock contentions in software significantly [141, 142]. We find similar trends for TPC-B and TATP workloads.

4.5.4 DRAM vs. SSD

In this section, we analyze the cost effectiveness of using SSD as slow non-volatile memory compared to using DRAM with the aim of demonstrating FlashMap's practical impact on data-intensive applications. We survey three large-scale memory intensive applications (as shown in Table 4.2) to conduct the cost-effectiveness analysis. For this evaluation, we ignore the benefits of non-volatility that SSDs have and purely analyze from the perspective of cost vs performance for workloads that can fit in DRAM today. Additionally, we analyze how real-world workloads affect the wear of SSDs used as memory.

We use three systems for the analysis: Redis which is an in-memory NoSQL database, MySQL with "MEMORY" engine to run the entire DB in memory and graph processing using the GraphChi library. We use YCSB for evaluating Redis, TPC-C [107] for evaluating MySQL, and page-rank and connected-component labeling on a Twitter social graph dataset for evaluating GraphChi. We modify these systems to use SSDs as memory in less than 50 lines of code each. The results are shown in Table 4.2. The expected life is calculated assuming 3,000 P/E and 10,000 P/E cycles respectively for the SATA and PCIe SSDs, and a write-amplification factor of two. The results show that write traffic from real-world workloads is not a problem with respect to wear of the SSD.

SSDs match DRAM performance for NoSQL stores. We find that the bottleneck to performance for NoSQL stores like Redis is the wide-area network latency and the router throughput. Redis with SATA SSD is able to saturate a 1GigE network router and match the performance of Redis with DRAM. Redis with PCIe SSD is able to saturate a 10GigE router and match the performance of Redis with DRAM. The added latency from the SSDs was negligible compared to the wide-area latency.

SSD-memory is cost-competitive when normalized for the performance of key-value stores. For a 1 TB workload, the SATA setup and PCIe setup cost 26.3x and 11.1x less compared to the DRAM setup ($30/GB for 32 GB DIMMs, $2/GB for PCIe SSDs, $0.5/GB for SATA SSDs). The base cost of the DRAM setup is $1,500 higher as the server needs 32 DIMM slots and such servers are usually expensive because of specialized logic boards designed to accommodate a high density of DIMM slots.

SSDs provide competitive advantage for SQL stores and graph workloads. We find that the bottleneck of performance for MySQL is concurrency. MySQL on PCIe SSD's Tpm-C was 8.7x lower compared to MySQL on DRAM for a 480 GB TPCC database. However, the SSD setup cost 11.1x less compared to the DRAM setup that makes the SSD setup 1.27x better when performance is normalized by cost. Processing graphs in DRAM is up to 14.1x faster than processing them on the SSD while the SSD setup used is 26.3x cheaper than DRAM system. However, the advantage of SSDs are not based on cost alone. The ability to use large SSDs as slow-memory allows such applications to handle workloads (up to 20 TB/RU) beyond DRAM's capacity limitations (1 TB/RU) with very few code modifications.

4.6 Summary

Using SSDs as memory helps applications leverage the large capacity of SSDs with minimal code modifications. However, redundant address translations and checks in virtual memory, file system, and flash translation layer reduce performance and increase latency. FlashMap consolidates all the necessary address translation functionalities and checks required for memory-mapping of files on SSDs into page tables and the memory manager. FlashMaps design combines these layers but does not lose their guarantees. Experiments show that with FlashMap the performance of applications increases by up to 3.32x, and the latency of SSD-accesses reduces by up to 53.2% compared to other mapping mechanisms.

CHAPTER 5
WEARDRIVE: ENERGY-EFFICIENT STORAGE FOR WEARABLES

The need for durable and secure storage for personal data incurs energy-intensive operations, which is especially a significant challenge for resource-constrained devices such as wearables. The size and weight constraints on wearables limit their battery capacity, which further compounds the energy problem.

This chapter presents WearDrive, a fast storage system for wearables based on battery-backed RAM and an efficient means to offload energy intensive tasks to the phone. It leverages low-power network connectivity available on wearables to trade the phone's battery for the wearable's by performing large and energy-intensive tasks on the phone while performing small and energy-efficient tasks locally using battery-backed RAM. WearDrive improves the performance of wearable applications by up to 8.85x and improves battery life up to 3.69x with negligible impact on the phone's battery life.

5.1 Introduction

The utility of a mobile device has long depended upon the tension between the device's size, weight and its battery lifetime. Smaller, lighter devices tend to be easier to carry. However, battery lifetime is mainly a function of size. A smaller device must therefore contain a smaller battery making energy a precious resource. The need for durable storage further compounds this problem. Slow flash storage wastes energy by keeping the CPU active for longer period of time [40, 41, 143], yet the use of a battery dictates that durable storage is vital to a device's utility. Likewise, data encryption is energy-intensive [8], but the sensitive nature of personal information that devices collect dictates using appropriate protection mechanism over a durable medium like flash that can be easily detached from a stolen device to retrieve personal data.

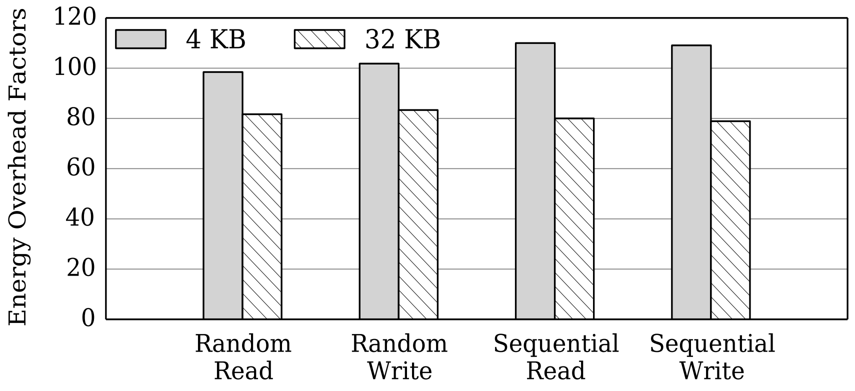
On wearables [144, 145, 146, 147], these trade-offs are magnified. Size matters even more since the device is worn on the body, therefore these devices have a very precious energy reserve. A watch that must be charged after a few hours is not very useful. Likewise, these devices generate precious sensor data (e.g., body sensor readings and location) that must be guaranteed against loss and theft.

In this dissertation, we explore a new approach to storage on wearable devices that do away with local durable storage while leveraging a nearby phone to protect against data loss and theft in an energy efficient manner. The system, called WearDrive, uses only memory on wearables for storage operations to provide performance and energy improvements. It exploits the battery in mobile devices to provide durability for the data in memory. It leverages low-power network connectivity available on wearables to exploit the capabilities of the phone. New data is asynchronously transmitted to the phone, which ultimately performs the energy-intensive operations of storing data with encryption in its local flash.

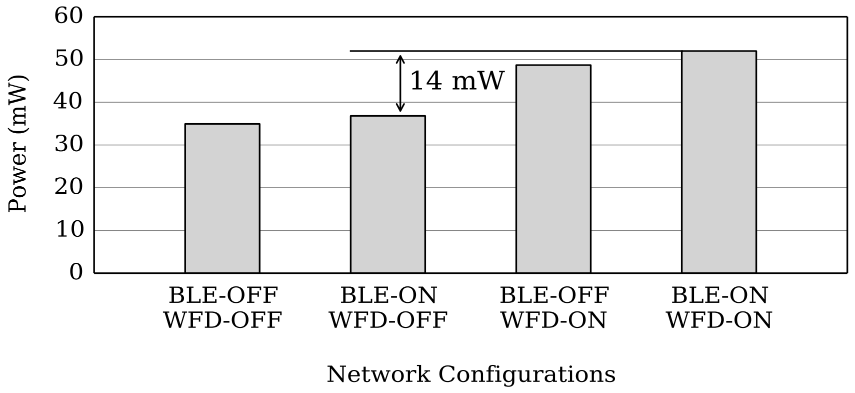
WearDrive targets the two most important application scenarios of wearables. The first scenario is the "extended display" that uses the wearable as a second display to allow applications on a nearby phone to run interactive but less-featured companion applications. Examples include companions that provide notifications for emails, social networks, etc. Providing *fast and durable storage* to such applications helps wearables conserve battery while remaining interactive.

The second scenario is sensor data analysis. Wearables are packed with sensors that take advantage of their location on a person's body. Exposing this data to the applications on the phone with *low-energy data sharing* can open up powerful applications. WearDrive targets these scenarios and reduces the need for a large battery and eliminates the need for flash on wearables.

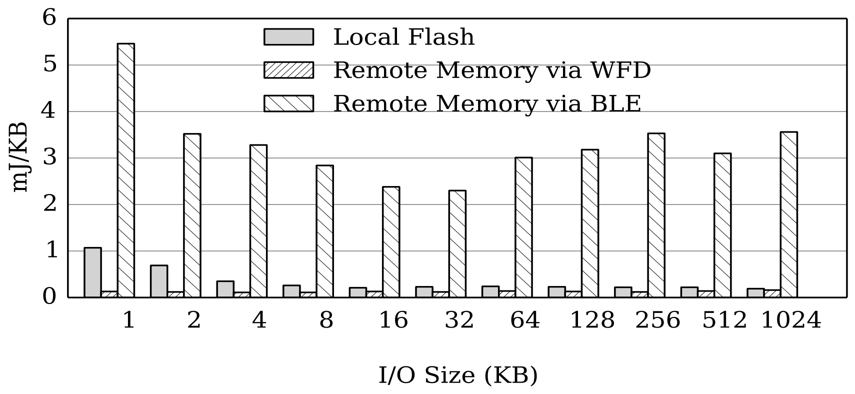
Experimental results show WearDrive helps applications obtain up to 8.85x better performance and consume up to 3.69x less energy compared to the state-of-the-art systems with little impact on the phone.



(a) Software Storage Stack Vs. Storage Hardware



(b) Standby Power Consumption



(c) Flash Vs. Wireless Network

Figure 5.1: Motivating scenarios for WearDrive: (a) Mobile storage stacks are energy-intensive because storage software consumes 80–110x more energy than flash. (b) To maintain a connection to the phone for the wearable, WiFi-Direct consumes 10–15mW extra power, while Bluetooth Low-Energy requires only 1–2mW. (c) In terms of energy consumption of the whole system when sequentially writing 32 MB data set with various I/O granularities, it is more energy efficient to write to remote phone's memory via WiFi-Direct than to write data locally to flash on the wearable.

5.2 Motivation

Wearables present a new challenge for mobile system design. Constraints on size and weight limit the battery capacity, but their location on the body and proximity to the phone create new opportunities.

5.2.1 Small Batteries

Li-ion battery metrics like gravimetric energy density (Watt-Hours/kg) and volumetric energy density (Watt-Hours/liter) take more than ten years to double [148]. Therefore, wearables will still be restricted to battery capacities of 1–2 Watt-Hours for the next several years because of their size and weight constraints; today's phones have 7–11 Watt-Hours batteries [149, 150, 151]. Therefore, we propose that the battery on the phone is traded for that on the wearable.

5.2.2 Energy Overhead of Legacy Platforms

To simplify the hardware and software development of wearables, manufacturers have chosen to reuse the system-on-a-chip (SoC) design and mobile operating systems that were originally made for phones and tablets. For example, most smart-watches and smart-glasses [144, 152, 145, 146] follow this approach to reduce cost and accelerate development of the platform and the applications. The focus of this paper is on such wearable devices. This means that wearables face a larger energy challenge compared to phones, because of their smaller batteries.

Our prior work [8] identified that mobile storage software consumes up to 110x more energy compared to flash hardware for accessing data as shown in Figure 5.1a. The energy overheads are caused by three factors. First, mobile flash is slow and increases CPU idle time while waiting for IO completion [40]. Second, storage on mobile devices is accessed via managed runtime environments like the Darwin engine on Android and the CLR engine on Windows that add additional CPU overhead. Finally, encryption of data that happens using special CPU instructions is also energy intensive. Therefore, a fast and energy-efficient storage system with security and privacy guarantees is needed for wearables.

5.2.3 New Applications on Wearables

Nearly all existing applications of wearables fall into two categories: extended display and sensor analysis. Using a wearable as an extended display requires arbitrary mobile application states are shared across the wearable and phone. And for wearables, the users focus more on *new content* from contextual applications like email, messaging, social networks, calendar events, music controls, navigation companion and etc.

Wearables are rich sources of sensor data. For example, watches can better monitor heart-rate and glasses can provide better video sensing. These sensors pave the way for a wide variety of useful applications including long term fitness/wellness tracking, detecting chronic health conditions like sleep-disorder, heart conditions, etc. Unfortunately, existing wearables are severely crippled in terms of battery size and provide only limited data analytics. A storage system capable of supporting these wearable workloads and exploiting their characteristics for performance and energy-savings is needed.

5.2.4 Low-Power Connectivity to the Phone

Bluetooth Low Energy (BLE) enables wearables to maintain a constant connection to the phone at a low-energy cost (Figure 5.1b). However, its low modulation rate imposes a large energy tax on large data transfers. An alternative is WiFi-Direct (WFD) which requires higher constant power to maintain a connection but supports low-energy large data transfers with high modulation rates. Figure 5.1c shows the average energy per KB consumed by the whole system of the wearable as it sequentially writes data to local flash or remotely to the phone via BLE/WFD. The results indicate that the energy overhead of writing data to remote memory via WFD is comparable to that of writing data to flash on the wearable.

The challenge is to build a mechanism to connect the wearable to the phone with a constant low-power connection overhead with a means to transfer data energy-efficiently. A hybrid connection and data-transfer mechanism can be built using BLE and WFD so that data sharing between wearable and phone can be enabled at a low-energy cost.

5.2.5 Slow flash on Wearables

The flash device for mobiles and wearables is slow and energy-intensive [40]. Faster flash technologies like SSDs require 25–100% more $/GB and 5x more energy per operation, and have a controller alone that is bigger than an entire SD card. Moreover, even SSDs are 10,000x slower than DRAM [1]. Furthermore, we demonstrate that data transfers over WiFi-Direct between two mobile devices consumes less energy than writing the same data to flash (Figure 5.1c). We propose that wearables actively use only DRAM (local and remote) to drastically speed up storage operations.

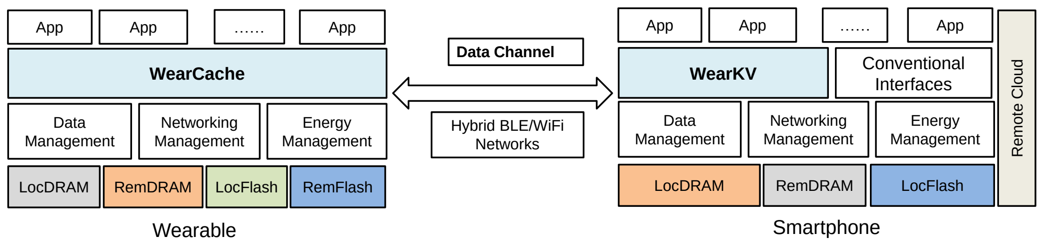
5.3 WearDrive Design

We begin by showing how applications minimize using flash and use mostly DRAM for *fast and durable* storage operations on wearables. We then present a new data management system that helps applications span extended-display and sensor data across the wearable and the phone. A new hybrid BLE/WFD data transfer mechanism is then described which helps WearDrive transmit data at a low-energy cost to the phone.

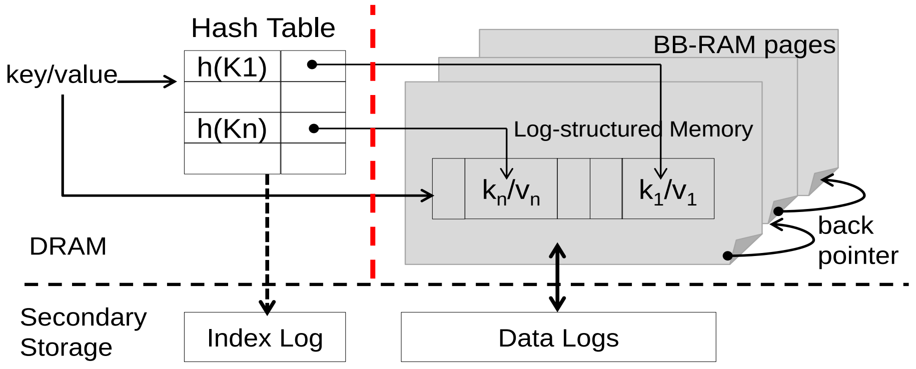
5.3.1 Storage with Battery-Backed RAM

To speed up storage operations, WearDrive actively uses DRAM as storage. However, WearDrive guarantees durability in spite of DRAM's volatility. DRAM on mobile platforms is continuously refreshed. The only time when the DRAM refresh stops is when the device is shutdown, the battery runs out of energy or it is removed. The first two scenarios

[1]Data surveyed from samsung.com, newegg.com and amazon.com



(a) WearDrive Overview



(b) KV-Store Design

Figure 5.2: (a) WearDrive expands wearable's memory and storage capacity by leveraging phone's capabilities. LocDRAM/RemDRAM represents local/remote DRAM, LocFlash/RemFlash are local/remote Flash. (b) BB-RAM pages are held in a linked list. The pages contain a sequential log of key-value pairs as they arrive. The hashtable stored in regular DRAM contains the index for the key-value store whose state can be efficiently recovered after failures.

provide an early warning sign allowing data in DRAM to be flushed to flash just in time before the refresh stops. Removing the battery while the system is running can lead to data loss even in today's systems. Moreover, most wearables' batteries are not removable. Therefore, we assume that DRAM can be treated as non-volatile on such devices. We call such DRAM as battery-backed RAM (BB-RAM).

BB-RAM coexists with DRAM to minimize OS changes. It grows and shrinks dynamically according to the memory pressure in the rest of the OS. DRAM is a precious resource on wearable devices. Most of the wearables we surveyed have less than 0.5GB of DRAM. While reserving a known and fixed region of physical memory as BB-RAM simplifies the implementation, it leads to fragmentation of DRAM and does not allow BB-RAM to dynamically expand and contract in accordance with application/OS requirements.

WearDrive's BB-RAM design adapts to memory pressure and spans across non-contiguous physical memory pages.

WearDrive uses BB-RAM both on the wearable and phone to ensure high-performance of applications spanning both the wearable and the phone. Wearable uses the phone as the secondary storage for its data. All old data on the wearable's BB-RAM is retired to the phone's BB-RAM. All dirty data in wearable's BB-RAM is also sent to the phone when the wearable needs to shutdown. Likewise, the phone uses its flash as the secondary storage for its data in BB-RAM.

Data in BB-RAM is not lost even after an OS crash. WearDrive uses a firmware component to ensure that BB-RAM is backed to flash in case of an OS crash. Firmware needs additional support to identify the physical pages that are used as BB-RAM. For this purpose, WearDrive reserves a small known region of physical memory to store a bitmap in DRAM to represent whether that physical page belongs to BB-RAM or not. The firmware uses these bits to identify BB-RAM pages after an OS crash (before shutdown) and writes them to a reserved region on flash. This simple design allows BB-RAM to coexist with DRAM and also enables a firmware without any OS state awareness to ensure data durability. Recovering WearDrive's state after a crash solely from the set of BB-RAM pages that it spans across is a harder problem and we present its design in the next sections.

WearDrive uses BB-RAM only as long as there is enough battery life left to ensure the durability of data in case of a crash. When battery level reaches a threshold, WearDrive stops using BB-RAM and treats all of DRAM as volatile. New and dirty data is first written to local flash to ensure durability. We set the threshold to 7% in WearDrive based on the observation that flushing 512 MB data from memory to flash sequentially costs about 5% of wearable's battery life on our reference wearable platform. However, this value can be adapted according to the hardware.

Warm reset. WearDrive is optimized for warm resets of the OS. If the available energy is above 7%, the firmware continues to refresh DRAM without scrubbing or cleaning any data. The OS then separates the pages in BB-RAM from regular DRAM using the bitmap and continues the boot process.

OS Deadlock. In case of a deadlock there is a chance that the data in BB-RAM will permanently be lost as the phone is completely drained out of battery. WearDrive uses a watchdog timer to detect if the OS is hung. When the battery life reaches the threshold, firmware schedules a BIOS-context process that wakes up once every sixty seconds and sets a bit in a known portion of memory that it expects the OS to reset every sixty seconds. If the OS fails to reset it during an iteration then the firmware assumes that the OS has hanged and flushes the data to flash by itself and disables the watchdog timer. The watchdog timer is also disabled as soon as the OS starts using DRAM as volatile.

5.3.2 Storing Data Across Devices

Since extended display and sensor data analysis scenarios need to span data across wearables and phone, we design WearDrive as a distributed storage system spanning across all devices. We find that in most extended-display scenarios, the wearable is treated as a helper for the full application on the phone because of the smaller screen size and small battery size on wearables. For this reason, we design the component of WearDrive on the wearable as a cache (WearCache) and the component of WearDrive on the phone (WearKV) as the main storage of data (see Figure 5.2a). WearKV and WearCache both have a key-value store interface that mobile application developers are familiar with. We use the same KV-store system to implement both WearCache and WearKV.

5.3.3 KV-store Design

KV-store is optimized for BB-RAM. This ensures fast and durable operations for WearCache and WearKV when inserting new data. KV-store prioritizes new data. The focus of wearable applications is on the latest data generated by phone applications and also by the sensors. Examples include the user's interest in latest notifications and most recent sensor values that can provide statistics about a run or a workout session. Therefore, the KV-store is implemented as a sequential log of key-value pairs in BB-RAM with FIFO replacement. Figure 5.2b illustrates the design. Keys and values are arbitrary length data blobs. New values are inserted by appending the KV-pair to the head of the log and adding a hash table entry with pointers to the key and the value in the log.

KV-store stores data in BB-RAM and metadata in DRAM. The log of KV-pairs is stored in BB-RAM and the hash table is stored in regular DRAM. The rationale for this is that the hash table can be recovered from BB-RAM in case of a crash by scanning through the BB-RAM pages in the right order. In case of a clean shutdown, the hash table is serialized to secondary storage (Index Log in Figure 5.2b). This design choice makes effective use of the precious BB-RAM space.

KV-store can recover BB-RAM and DRAM state after a crash. Recall that the firmware flushes BB-RAM pages to local flash in case of a crash. To recover the hash table and the correct head of the log of KV-pairs, ordering of the BB-RAM pages is needed. The ordering of the BB-RAM pages in the log is determined by a four-byte pointer stored at the tail of every BB-RAM page to the next BB-RAM page in the log as shown in Figure 5.2b. Each KV-pair in BB-RAM is a sequence of five fields: four bytes length of the key followed by the key, followed by eight bytes of application identifier (described later) and then four bytes length of the value followed by the value. This FIFO of BB-RAM pages allows the KV-store to arbitrarily increase its size by appending new pages and decrease the size of the log by purging the KV-pairs at the tail to secondary storage. Moreover, the firmware remains simple, precious BB-RAM space is best utilized and recent data that is of interest for applications is prioritized during page replacement.

WearCache is the KV-store instance that lives on the wearable and caches all the latest data from applications and sensors. New data arrives in WearCache via two methods: when phone applications push data to their companion applications and when sensors generate new values. When WearCache runs out of BB-RAM, it flushes old data to WearKV on the phone in FIFO order as the focus of the wearable is always on new data. It does so by simply moving the tail forward in the log of KV-pairs on BB-RAM several KV-pairs at a time. This provides the functionalities of having recent data on the wearable, adapting to memory pressure, and providing an efficient replacement policy. An example application on today's watches that can leverage this storage model is a notification center for recent emails. The user's focus will be on the most recent emails while the older emails may be safely flushed to WearKV as the user may not access them on the wearable. Complex functionalities are implemented by the email application on the phone while a companion email application on the wearable keeps the design/UI simple with a focus on latest data.

WearCache removes flash I/O overhead from the critical path of applications. The OS, application binaries and other application metadata continue to reside on local flash. However, data accessed in critical path resides in WearDrive. The key-value interface to WearDrive eases development as wearable applications already use the key-value interface for sharing data between the phone and wearable [153]. As future work, we wish to provide filesystem and database interfaces using BB-RAM.

WearDrive supports simple sensor data analytics on the wearable and complex data analytics on the phone. Small battery restricts wearables to analyzing sensor logs from short activities like the latest run/workout-session or other short activity. However, applications can perform rigorous analytics on the phone (several days worth of sensor logs at a time). Applications on the phone can proactively pull the sensor data from WearCache as and when a certain number of samples are available. For example, a fitness tracker on the phone can register with WearCache that the heart-rate logs from the wearable be pushed to the phone once every ten minutes. WearCache implements these requests in the following manner. For each sensor, WearCache pre-allocates a KV-pair. A certain amount of space is reserved for the value upfront. The sensor samples (configurable sampling rate) are gradually added to the pre-allocated value as they become available. Data is pushed to the phone and phone-applications are notified accordingly.

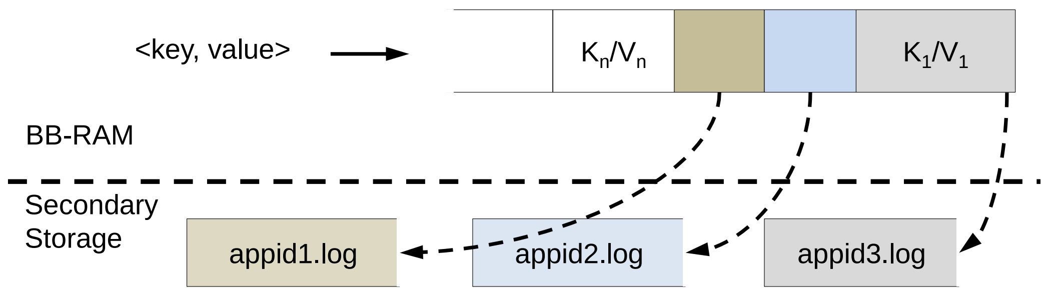


Figure 5.3: WearDrive creates individual logs per application and per sensor to isolate on secondary storage.

WearKV is the KV-store that resides on the phone and contains all the data of the wearable. It contains old extended display data and the entire log of sensor values. Old extended display data is fetched back to WearCache on demand (this is a rare event as wearables focus on new data). The phone with its larger battery can use the full sensor log to perform rigorous sensor data analysis. When WearKV runs out of BB-RAM, it flushes old data to flash where it creates a per-application and per-sensor sequential log as shown in Figure 5.3. It does so by leveraging the metadata information stored in the values where it records the device-ID, application-ID, sensor-ID and time stamp of creation.

Data in WearDrive crosses the memory/flash boundary only on the phone. Data encryption and other mechanisms put in place to ensure security and privacy of data are needed only for “truly” non-volatile media like flash that can be detached from the rest of the phone and have unprotected data stolen in a straightforward manner. Therefore, the heavy software cost [8] of storage is offloaded to the phone. Note that treating DRAM as non-volatile by using it as BB-RAM is at least as secure as the previous model where data was not encrypted in DRAM as DRAM which is part of the SOC is hard to detach from a device. BB-RAM is a mechanism to ensure that data in DRAM in never lost as opposed to making DRAM “truly” non-volatile.

Offline Capabilities. WearCache can function without the phone. WearCache can lock data on the wearable based on time of arrival such that it is not purged to the phone until explicitly deleted. Offline capabilities allow applications to lock data to be available locally so that functionality can be provided without the phone. An example is when the email companion application imposes a restriction that emails from last three days are locked locally. KV-pairs are written to flash on the wearable only if WearCache runs out of BB-RAM and the applications impose an offline availability restriction. Offline requirements are specified in WearCache using time cutoffs per applications and per sensor. We compare the specified time with the timestamp stored in KV-pair's metadata. The qualified offline data is written to its local flash's logs. As time passes, WearCache will move the tail closer to the head on the flash log and overwrites older data that the application does not need.

5.3.4 Communication

Efficient reachability to the phone allows the wearable to be designed with less DRAM and slower flash thereby reducing their cost. Moreover, it allows the wearable to offload storage and computations to the phone. BLE 4.1 and 802.11a/b/g/n/ac are the network connectivity options for wearables. While a few smart-watches only have BLE, we envision that Wi-Fi will make it to all wearables as it enables efficient large data transfer.

Standalone BLE or WFD is not an ideal network connection. BLE consumes low power (1–3mW) for staying always connected to the phone while using a WFD to stay connected to the phone consumes 5x extra power (10–14mW) (Figure 5.1b). On the other hand, BLE consumes 10–20x extra energy for transmitting data when compared to WFD (Figure 5.1c). A mechanism to minimize the total energy of always staying connected and for transferring data is required.

Using BLE for staying connected and short data transfers, and turning on WFD solely for large data transfers is a hybrid solution. This is practical because WearCache and WearKV know how much data is to be pushed. If it is beneficial then a control signal over BLE is sent to the other side to turn on WFD. Data transmission begins on BLE and switches over to WFD when it is available.

Knowing the right data transfer size for switching on WFD is crucial. To estimate the

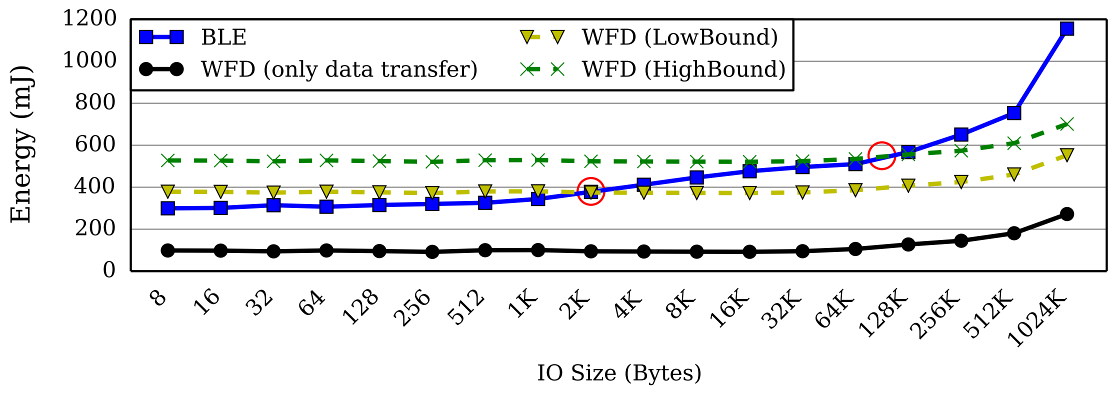


Figure 5.4: Energy consumption of data transfer via BLE and WFD. WFD is efficient if connection establishment, tail latency, and connection-teardown are not included.

transfer size at which it pays-off to turn on the WFD, we conduct the following experiment: transferring data of various sizes on BLE and WFD. We keep BLE always on and send data of various sizes between two mobile devices whose power consumption is monitored using the Monsoon power monitor [154]. We then estimate the energy required for transferring the data via WFD. The energy estimates for WFD contains the energy needed for turning the WiFi chipset on and off. Figure 5.4 shows the transfer size at which using the hybrid protocol pays off.

The pay-off point for switching to WFD depends on signal quality. We present the results for two extreme modulation rates in 802.11n: the highest modulation rate and the lowest modulation rate. A crossover-point database is built for various modulation rates of BLE and WFD. We use the BLE signal strength to estimate the WiFi signal strength as they use the same band and radio over the same distance.

Picking the right time to turn off WFD is important. WFD consumes more power than BLE in the idle state (i.e., standby power gap). However, network discovery, connection, and powering-down are expensive, frequently turning WFD on/off would incur more energy usage than keeping it in the idle state for workloads with small inter-arrival times. We use two solutions to solve this problem. The first is to have a running average of inter-arrival times and predict on the basis of the average-value if it is worth keeping the WFD

Table 5.1: WearDrive API

API	Description
OpenWearDrive (FileName)	open a connection to WearDrive and obtains a handle, the data is represented using an opaque *FileName*.
CloseWearDrive (handle)	close the connection to WearDrive and flush any data from BB-RAM in the process to an appropriate location.
InsertKV (handle, key, value)	insert the new key/value to the *FileName* corresponding to the handle.
ReadKV (handle, key)	provide the value corresponding to the key in the *FileName* file.
MakeOffline (handle, date)	make all data of this file that arrived after a certain date available on the wearable even when the phone is not reachable. Date is specified relatively to the current time. This function is available only to WearCache.
DeleteOldData (handle, date)	provide a hint to WearDrive that data beyond a certain date can be deleted. Date is an absolute value. This function is available only to WearKV.
RegisterForSensor (DeviceID, SensorID)	register an application for values from the sensor represented by *(DeviceID, SensorID)*.
UnregisterFromSensor (DeviceID, SensorID)	unregister the application from a sensor.
RegisterCallBack (TimeGap, CallBackFunction)	make WearDrive issue the *CallBackFunction* in the context of registering application every *TimeGap* seconds with the newly available sensor values.
Compute (DeviceID1, SensorID1, ..., DeviceIDN, SensorIDN, TimeGap)	a function that does not access any global variables but accesses data in sensor logs that are accessible to the application. It can be executed on both wearable and phone.

on. The second is to explicitly help applications that can tolerate delay to batch data for further energy saving.

5.4 WearDrive Implementation

We implement WearDrive on Android 4.4 using Java, C and JNI [155]. It consists of the KV-store, the data transfer library and the code needed for ensuring the durability of BB-RAM. WearDrive is accessed via the calls on *all devices* as shown in Table 5.1. `InsertKV` and `ReadKV` always append the application ID (stored in `handle`) to the key for inserting and reading data. This helps WearDrive isolate data between applications. Privacy is protected by not providing user-space access to BB-RAM. All data is accessed through user space buffers provided to the system calls.

Sensor values are aggregated by WearDrive on a per-sensor basis. Applications can

Table 5.2: Workloads included in WearBench.

Workload	Parameters	Application examples
Extended Display	Size and inter-arrival time distribution of data	Email, news, instant messages, status updates from social networks, etc.
Sensors	sampling rate, monitoring period	Physical fitness, sleep quality, heart health monitoring, elder care, etc.
Audio/ Video	Encoding rate, quality, monitoring period	Dash-cam using glasses, sleep quality monitoring.

register sensor logs for each sensor. WearDrive directly appends sensor samples to the pre-allocated KV-pair that is buffering the current set of sensor samples. When enough samples are available, WearDrive notifies the corresponding applications.

5.5 Experimental Methodology

Evaluating wearable applications is hard because of the lack of a standard benchmarking tool that can generate representative workloads that span across wearables and phone. We present WearBench, a framework that is intended to test the impact of data generated by such wearable workloads on performance and energy.

5.5.1 WearBench

WearBench is an Android app that runs on the phone/wearable for generating the extended-display data and sensor data which represent wearable applications. WearBench runs on the phone when testing the wearable and vice versa so that WearBench does not interfere with the measurements. WearBench defines synthetic data-analytics that can be executed on sensor logs like calculation of running statistical features including average, standard-deviation, k-means, and hourly/diurnal/weekly pattern recognition algorithms – sampling rate and timeliness are configurable. WearBench can create notifications of varying sizes and different inter arrival time distributions. To the best of our knowledge, WearBench is the first framework for benchmarking wearable systems.

We identify several typical data-intensive workloads running on smart wearables (see

Table 5.3: Reference wearable device used for evaluation.

Type	Our Reference Wearable	Samsung Gear
Processor	1.2 GHz dual-core	1.2 GHz dual-core
Memory	1 GB RAM	512 MB RAM
Storage	4 GB eMMC flash	4 GB eMMC flash
Network	Bluetooth 4.0 LE, WiFi 802.11 b/g/n	Bluetooth 4.0 LE, WiFi 802.11 b/g/n
Sensors	accelerometer, barometer, compass, GPS, gyroscope, heart rate monitor, magnetometer, altimeter, barometer, UV light sensor, ambient light sensor, BLE and WiFi events, camera, microphone	accelerometer, gyroscope, compass, heart rate monitor, ambient light, UV light, barometer, GPS, microphone, BLE and WiFi events
OS	Android 4.4	Android 4.3+/Tizen

Table 5.2). In order to cover a wide variety of users, we abstract the usage pattern as configurable parameters in WearBench.

5.5.2 Experimental Setup

We use a low-end mobile platform as a reference wearable device that runs Android 4.4. As shown in Table 5.3, our reference wearable device compares to Samsung Galaxy Gear smart-watches which have similar hardware and software configurations. While our reference has 1 GB RAM, we use only 512 MB on it for the system to match the amount of RAM on state-of-the-art wearables.

Monsoon power monitor [154] is used to profile energy consumption of the device. We instrument the reference wearable device's battery-leads such that it draws power from the Monsoon power meter instead of a battery. We perform comparative energy calculations by subtracting the base power of the system from the power used when a workload is executed. However, when reporting absolute energy required for a workload we include the base power of the system. We compare WearDrive with the following state-of-the-art storage solutions:

WearableOnly: The wearable applications use the capabilities on the wearable for storage. The phone is used only for Internet connection via tethering. All the computation is performed locally and all data is durably written to local flash. This is the way most fitness/health trackers are implemented on today's wearables.

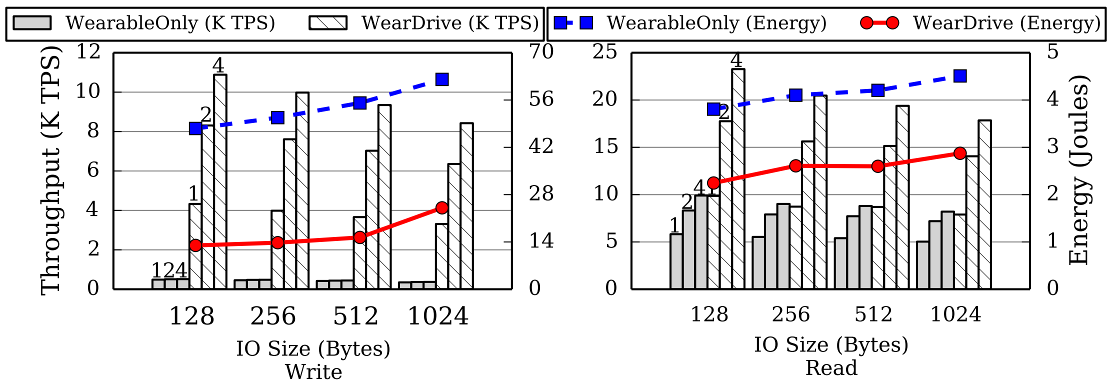


Figure 5.5: Performance and energy comparison of WearableOnly and WearDrive with a varied number (1, 2, 4) of threads.

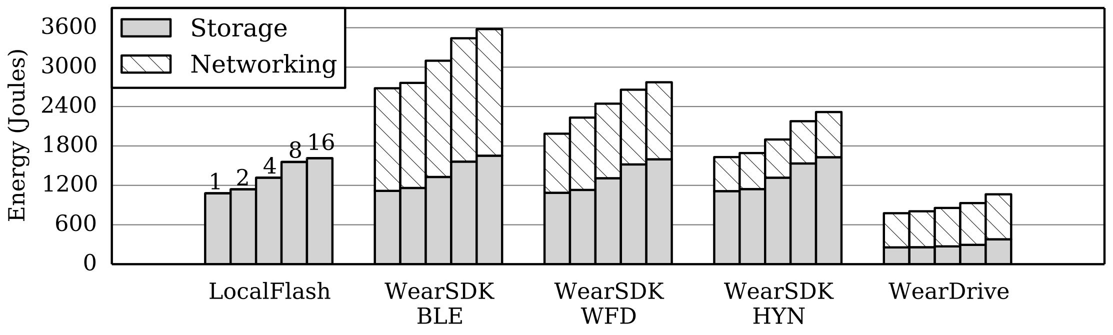


Figure 5.6: Energy used by various storage systems with varied number (1–16) of sensors sampling values continuously at 1Hz for 24 hours. A typical smart-watch battery contains between 3000–6000 Joules of energy.

WearSDK: Android Wear SDK released by Google [153] is one way to span data across wearable and phone. However, this SDK uses flash synchronously on either one of the devices to ensure durability. WearSDK provides a data layer for data synchronization between paired wearable and phone via BLE (i.e., WearSDK-BLE). We extend the data layer and make it support WFD (i.e., WearSDK-WFD) and our hybrid network protocol (i.e., WearSDK-HYN).

5.6 Results and Analysis

5.6.1 Local Memory vs. Local Flash

We first examine the advantages of BB-RAM over local flash with a set of microbenchmarks. We configure WearBench to issue 100 K `InsertKV` and `ReadKV` operations. The size of the data written or read is varied uniformly from 128 bytes to 1 KB. Figure 5.5 compares the throughput for different data sizes. WearDrive outperforms WearableOnly by 6.65–8.85x on inserts where storage I/O from flash becomes the bottleneck, and 1.57–1.69x on reads where the CPU becomes (single thread) the bottleneck for our system and the flash IOPS for WearableOnly. Moreover, WearDrive's throughput scales linearly till four threads while WearableOnly is saturated by a single thread. Figure 5.5 also shows the total energy usage of these write/read operations. WearDrive consumes 2.58–3.69x and 1.57–1.70x less power than WearableOnly on inserts and reads respectively, as slow I/O operations on flash cause more CPU cycle wastage, and further increase the energy usage.

5.6.2 Passive Sensor Data Aggregation

We demonstrate the benefits of using local and remote BB-RAM for providing durability for sensor data recording over flash.

Fitness/health tracking applications collect sensor values on a periodic basis and update statistics [156]. We use a fitness tracker application that samples various sensors at 1Hz and stores them to local flash periodically. We record the storage calls that this application makes for storing sensor logs, and incorporate the workload into WearBench for replaying.

WearDrive aggregates sensor data in BB-RAM and ensures their durability. WearableOnly and WearSDK unfortunately cannot provide such guarantees unless they write every sensor sample through to flash, but they suffer severe performance losses in doing so. As a tradeoff between durability and performance, for these methods, we write the sensor samples to flash when data fills a sector (512 bytes). Every five minutes, all the new data is sent to the phone as sending data to the phone at 1Hz leads to significant energy wastage because the network chip would never go into low power mode. Figure 5.6 shows the total amount of energy in Joules required each day only recording the sensor values. The overall trend across all the systems show that the number of sensors sampled does not severely impact the energy consumption of storage, indicating that the setup costs inside storage stack are the dominant factors for this workload.

WearDrive outperforms the other systems by up to 3.31x and provides better durability. When sampling 16 sensors every second for the whole day and writing them to flash, the storage system (hardware and software) requires 1760 Joules. Considering a typical smart-watch battery that contains 4000 Joules (1.1 Watt-Hour) of energy, writing sensor data to flash requires 44% of total battery life each day. WearDrive on the other hand consumes 28.25%, which is 1.54x more efficient. Moreover, we find that 89.5%, 68.1% and 58.75% of the battery life is respectively required by WearSDK-BLE, WearSDK-WFD and WearSDK-HYN. While HYN reduces the cost of transmitting data over the network to the phone, the bulk of the cost for these systems is still from using slow flash which wastes energy by delaying CPU and network from going to sleep sooner.

5.6.3 Extended Display Workload

In this experiment, we demonstrate the benefits of WearDrive to efficiently store extended-display data durably. We use WearBench to emulate application patterns from representative workloads of Twitter [138], Instagram [157], and email [143] applications with various parameters (size and interarrival time).

Varying inter-arrival times. In order to model more notification workload patterns, we vary the interval between tweets from 5 to 60 seconds and measure the energy-impact from storing them durably on the wearable. Figure 5.7 shows these results.

WearDrive reduces energy usage by 1.2–2.9x compared with the default option of WearSDK-BLE for today's wearable applications. The benefits are made possible not only

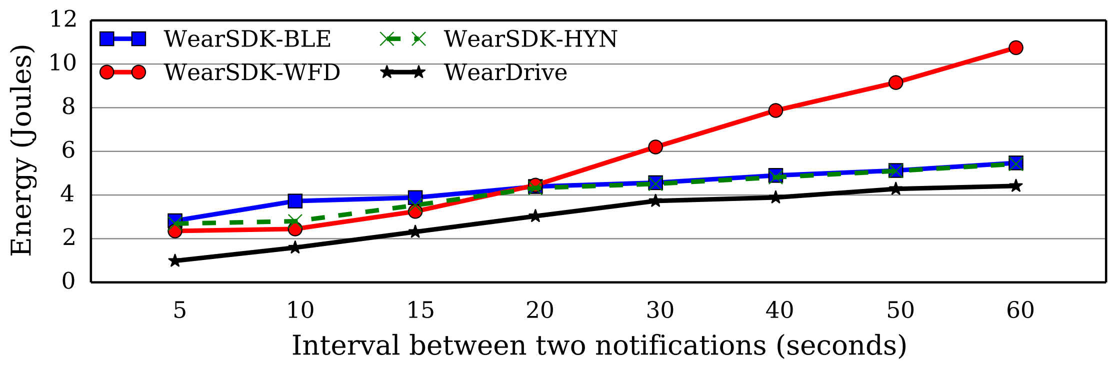


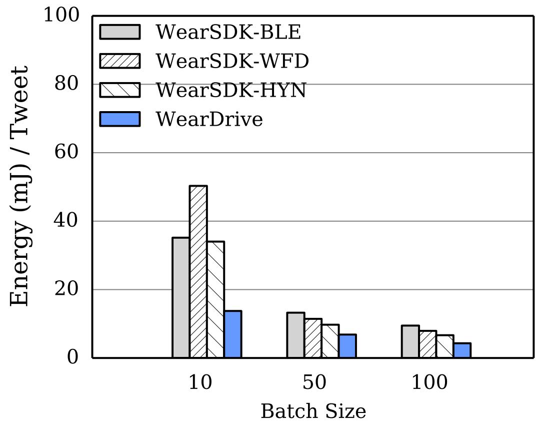
Figure 5.7: Energy usage of receiving 10 notifications (10KB size) with the varied interval between notifications.

because of the performance benefits of BB-RAM, but also because of the energy-benefits of HYN. Faster storage operations help the CPU and network go back to sleep faster and reduce the energy footprint.

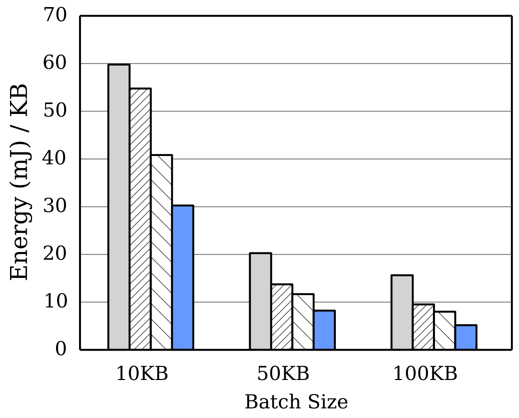
With HYN, WearDrive uses WFD when the average interval between notifications is small enough to warrant keeping WFD active (20 seconds for our hardware). When the interval is further increased, WearDrive will intelligently turn off WFD and use BLE to send notifications. The hybrid networking protocol also brings benefit to WearSDK (see WearSDK-HYN in Figure 5.7). For long intervals, WearDrive still performs better than WearSDK-BLE, because of its faster storage.

Effects of batching notifications. Buffering data on the phone gives HYN more opportunity to exploit the energy efficiency of the WFD protocol. We vary the size of the notifications pushed by the phone to wearable from 128 bytes to 1KB. The batch size that the data is sent ranges from 10 to 100. This experiment allows us to study the energy-benefits of delaying notifications from applications that are less interactive than instant messages, such as social networking updates and even email in some cases.

Figure 5.8a shows the results for tweets which are short social networking messages that can tolerate delay. Compared to WearSDK-BLE, WearDrive takes 2.93x less time, while saving energy by 2.23x. The overhead of WearSDK is reduced with WFD and HYN for a large number of notifications. For small number of notifications such as 10 notifications,



(a) Batching Tweets



(b) Batching Email

Figure 5.8: Performance and energy usage of notification workload with different data size.

HYN will use BLE instead of WFD for data transfer. The execution time of WearSDK-WFD is less than WearSDK-BLE and WearSDK-HYN, but its energy usage is larger as the overhead on WiFi discovery and connection offsets its benefit on data transfer. WearDrive is 1.81x more energy efficient than WearSDK-HYN because of BB-RAM's fast durability.

Likewise for email, as shown in Figure 5.8b, the benefits of HYN when batching when possible are apparent. However, WearDrive is 2.49x more energy efficient than WearSDK-HYN because of the fast durability guarantee provided by BB-RAM. Overall, WearDrive helps extended-display applications not only by making the energy-batching tradeoff straightforward to exploit but also by providing benefits for applications that are interactive by enabling fast durability.

5.6.4 Impact on Smart-phone

In this section, we evaluate the energy usage on the phone and show how WearDrive can improve the lifetime of wearables by leveraging only a negligible portion of phone's larger battery capacity. To understand the energy impact on the phone accurately in this context, we use the same reference hardware in Table 5.3 as a phone. Note that this is a hardware specification similar to most low-end phones on the market today. However, we use a

Table 5.4: WearDrive saves wearable's battery by trading it with the phone's battery.The battery capacities of the wearable and phone used in the experiments are 300 mAh and 2000 mAh respectively.

Algorithms	Mean		k-NN		ID3		k-means	
Schemes	% of battery life on		% of battery life on		% of battery life on		% of battery life on	
	wearable	phone	wearable	phone	wearable	phone	wearable	phone
WearableOnly	14.72%	-	18.85%	-	20.24%	-	27.12%	-
WearableOnly+InMem	0.83%	-	4.96%	-	6.56%	-	13.23%	-
WearDrive	0.87%	0.21%	0.87%	0.83%	0.87%	1.08%	0.87%	2.09%

2000mAh battery as the reference battery when evaluating the energy impact on the phone.

Energy cost of storage: We reuse the fitness monitoring application workload from Section 5.6.2. Recall that for recording 16 sensors at 1Hz for 24 hours requires 28.25% of the battery life on the wearable instead of 44.0% when writing the data to the flash on the wearable. For this experiment, we find that the phone requires 1369 Joules of energy. This energy accounts for 5.1% of the battery on the phone but this leads to savings of 16% of the battery on the wearable. Considering the fact that the batteries on wearables are usually 5–7x smaller than on the low-end phone, this is a valuable tradeoff to make. Moreover, having the data on the phone enables the phone to perform analytics and provide more energy savings for the wearable device.

Energy cost of computation: We implement `Mean` and three commonly used data mining algorithms in WearBench: `k-NN` (k-Nearest Neighbor) for classification [158], `ID3` (Iterative Dichotomiser 3) for generating decision tree [159], and `k-means` for cluster analysis [160] for detecting patterns in streams of sensor data to find out when user's heart rate is high [161], when a user snores during the night [162], the levels of UV exposure [163], etc. WearableOnly refers to the baseline, in which records are stored in SQLite and data analytics run on wearables. WearDrive performs computation on the phone with the data in WearKV. The sensor data are aggregated over three days.

Table 5.4 shows that WearableOnly method of storing and computing on the wearable consumes a significant portion of wearable's battery life, ranging from 14.72% to 27.12%. For smaller data sets the data can be read into memory all at once and computed over as opposed to reading data from flash in batches. We refer to this solution as WearableOnly+InMem. It reduces the energy usage dramatically, but it works only for small workloads that fit in memory. However, when sampled at a higher rate (required usually when the user is running or biking) of over 10Hz, sensor data beyond a few hours will not fit in the memory of the wearable. While such workloads may not fit in the phone's memory either, the phone's larger battery takes much smaller impact.

When the computation is shifted to the phone by WearDrive, it consumes a trivial portion (0.21%–2.09%) of phone's battery life, but reduces the energy usage on wearables to be only 0.87% of wearable's battery life for issuing the arithmetic functions. As future work, we wish to explore when offloading computation to the cloud pays-off with respect to energy. Offloading to the cloud incurs more energy overhead due to data transmission across a wide area with WiFi or LTE. For instance, uploading 8 MB data to Google Drive [164] consumes 3.14x more power than writing to local flash in our experiment setup (with perfect WiFi conditions).

5.7 Summary

WearDrive demonstrates that battery-backed RAM (BB-RAM) can provide significant performance and energy benefits for wearable applications. It also shows how Bluetooth and WiFi can be used in combination to provide a low-energy communication link (HYN) between the wearables and the phone. BB-RAM in combination with HYN provides a quick and energy-efficient way for wearable applications to span data across all the devices on the body enabling new functionalities for users. We validate these benefits with various typical wearable applications using a new wearable benchmarking suite that we develop, and show that WearDrive is 1.16-1.55x more energy-efficient compared to existing solutions. WearDrive can leverage phone's capabilities to reduce energy usage of wearables by up to 15.21x, with trivial impact on the phone for realistic wearable workloads.

CHAPTER 6
FLASHGUARD: HARDWARE-ASSISTED DEFENSE AGAINST RANSOMWARE

The storage system software has been developed for decades, but it is still vulnerable to malware attacks. Taking the encryption ransomware for example, it is a malicious software that stealthily encrypts user files and demands a ransom to provide access to these files. Several prior studies have been proposed to detect ransomware. However, by the time the ransomware is detected, some files already undergo encryption and the user is still required to pay a ransom to access those files. Furthermore, ransomware variants can obtain kernel privilege, which allows them to terminate software-based defense systems, such as anti-virus and data backups. Ideally, we would like to defend against ransomware without relying on software-based solutions and without incurring additional storage overheads.

To that end, this dissertation proposes FlashGuard, a ransomware-tolerant Solid State Drive (SSD) which has a firmware-level recovery system that allows quick and effective recovery from encryption ransomware without relying on explicit backups. FlashGuard leverages the observation that the existing SSD already performs out-of-place writes in order to mitigate the long erase latency of flash memories. Therefore, when a page is updated or deleted, the older copy of that page is anyway present in the SSD. FlashGuard slightly modifies the garbage collection mechanism of the SSD to retain the copies of the data encrypted by ransomware and ensure effective data recovery. Our experiments with 1,447 manually labeled ransomware samples show that FlashGuard can efficiently restore files encrypted by ransomware. In addition, we demonstrate that FlashGuard has a negligible impact on the performance and lifetime of the SSD.

6.1 Introduction

Recently, criminals are unleashing brash attacks on users' machines through a new type of malicious software called encryption ransomware [165, 166, 167]. For example, the WannaCry ransomware [165] launched on May 12, 2017 has infected more than 230,000 computers across 150 countries. Among the victims are government agencies, schools, hospitals, and police departments.

Different from traditional malware which typically disrupts computer operations and gathers sensitive information, encryption ransomware stealthily encrypts the files on user's machine and demands users pay a ransom to restore the files. Since the operations performed by ransomware are indistinguishable from benign software, ransomware can easily bypass various antivirus, making it increasingly prevalent in cyber criminals [168, 169]. According to a study from IBM Security [170], the number of users who came across encryption ransomware in 2016 increased by more than 6,000% over the previous year. The ransomware attacks cost their victims about a billion dollars in 2016 which is a 41x increase compared to the cost in all of 2015 [171].

To counteract ransomware, researchers have proposed several detection systems that use file access patterns [172, 173] or features of cryptographic algorithms [174] to identify ransomware. However, these detection mechanisms still cannot prevent ransomware from locking up user data. First, existing ransomware detection occurs only after observing the actual damage. Given that the encrypted data may contain the files considered to be valuable, victims still have to shoulder the burden of paying the ransom. Second, some ransomware can run with kernel privileges, which allow them to carry out kernel-level attacks. Therefore, ransomware can easily disable or work around the aforementioned detection mechanisms.

To address these issues, one instinctive solution would be to enable file backup on local persistent storage (e.g., journaling and log-structured file systems [59, 60]) or remote machines (e.g., NFS [175] and cloud-based storage [61]). However, this is insufficient at guarding against ransomware. First, any file backup mechanisms inevitably impose storage overhead. Second, ransomware may find and jump to the backup and encrypt it regardless of whether it is on shared network drives, local hard disk drives, external storage devices, or plugged-in USB sticks [176]. Third, ransomware with the kernel privilege can also terminate backup processes, making them futile against ransomware defense.

As the replacement to conventional persistent storage devices – hard disk drives (HDDs), Solid-State Drives (SSDs) have been widely used on many kinds of computing platforms, because they provide orders of magnitude better performance than HDDs while their cost is fast approaching to that of HDDs [9, 10, 11, 12]. A unique property of SSDs is that a physical page cannot be written until it is erased, however, the erase operation incurs significantly longer latency. To overcome such a shortcoming, modern SSD performs out-of-place write for every write. Therefore, SSDs intrinsically support the logging functionality without requiring an explicit backup. Such a feature will naturally preserve the old copies of overwritten or deleted files for a period of time before they are reclaimed by the process of garbage collection. Moreover, the firmware-level logging could isolate the data protection and recovery from operating system (OS) kernels and upper-layer software.

Unlike existing ransomware detection systems [177, 173] and explicit file backups [56, 57], we take advantage of the intrinsic flash properties and build a ransomware-tolerant SSD named FlashGuard, which has a *lightweight hardware-assisted* data recovery system. It allows users to reinstate the data held in captivity by ransomware.

While the proposed system is based on the out-of-place write characteristic of an SSD, it is challenging to leverage such a feature for data recovery for two major reasons. First, once data is deleted or overwritten but gets left behind on the drive, SSD controller may perform garbage collection (GC) to erase the blocks taken up by such stale data for free space. Given that stale data may contain the original data copies "deleted" or "overwritten" by ransomware, FlashGuard needs to hold stale data and prevent GC from discarding them.

Since holding too much stale data could increase the GC overhead, which further affects the performance of regular storage operations significantly [178] and even jeopardizes the SSD lifetime [179], an efficient GC mechanism is desirable. Second, we must guarantee that the change to the GC is resistant to the potential attacks against SSDs from the ransomware running with the kernel or administrator privilege.

To tackle these challenges, we implemented FlashGuard's data recovery system in SSD firmware by augmenting GC mechanism with the ability to only hold the data potentially deleted or overwritten by ransomware. We prototyped FlashGuard on a 1 TB programmable SSD with minimal modifications to the existing SSD design. Using a real world set of 1,477 distinct ransomware samples covering 13 families, we show FlashGuard can quickly recover the files held by ransomware. For example, we demonstrate FlashGuard can restore 4 GB of encrypted data in 30 seconds. Using a set of publicly available storage traces, we extensively evaluated the impact of FlashGuard upon the storage performance. Our experimental results show that FlashGuard incurs negligible performance overhead (up to 6%) and has a trivial impact (less than 4%) on SSD lifetime.

To the best of our knowledge, FlashGuard is the first defense scheme that can efficiently offset the damage of ransomware to user data even if ransomware run with administrator privileges to load kernel code or exploits a kernel vulnerability.

6.2 Ransomware Study

Among various strains of ransomware, encryption ransomware is the most common type that encrypts user data and demands money in exchange for decrypting them. The objective of this work is to design and develop a ransomware-tolerant SSD which has the data-recovery capability to offset the damage to user data resulting from encryption ransomware. To achieve this, we first analyze the behaviors of encryption ransomware and understand how they interact with user data by conducting a study on a large number of ransomware samples. Different from prior studies on ransomware [172, 173, 180], our

Table 6.1: Ransomware families, their encryption time, and behaviors of deleting backup files (backup spoliation).

Family	Samples		Encryption		Backup spoliation
	Num	%	Target	T (*mins*)	
Petya[1]	14	0.95	MFT	2	✯
CTB-Locker	119	8.05	Files	14	✗
JigSaw	5	0.34	Files	16	✗
Mobef	7	0.47	Files	16	✗
Maktub	10	0.68	Files	22	✓
Stampado	42	2.84	Files	27	✗
cerber	29	1.96	Files	37	✓
Locky	344	23.29	Files	43	✓
7ev3n	16	1.08	Files	44	✓
TeslaCrypt	75	5.08	Files	44	✓
HydraCrypt	13	0.88	Files	70	✓
CryptoFortress	4	0.27	Files	75	✓
CryptoWall	799	54.10	Files	75	✓
Total	1477	100	–	–	–

study focuses on two aspects – *encryption time* and *backup spoliation*.

6.2.1 Study Methodology

We gathered 1,477 encryption ransomware samples from VirusTotal [181] and classified them into 13 distinct ransomware families based on the ransom notes they present to victims. Table 6.1 illustrates these families, their encryption strategies and the number of samples in each ransomware family.

Following the common scientific guidelines [182], we executed each ransomware sample within a virtual machine (VM) running 64-bit Windows 7 SP1 with 2 CPU cores and 4 GB main memory on a host machine (configured with 2.67 GHz Intel quad-core Xeon processor and 8 GB DRAM). We removed the barriers of ransomware execution by disabling protection services such as firewall, Microsoft security protection, and user account control. Moreover, we granted all ransomware samples the administrator privilege. Since ransomware might perform key-exchange with the control server and establish those en-

[1]we do not deem `Petya` ransomware that it deletes backups because `Petya` demolishes and replaces Windows file system.

Table 6.2: File distribution in a normal user's computer.

Type	Number		Size		
	Num	%	Avg (KB)	Total (MB)	%
pdf	2378	24.08	565.27	1312.70	30.28
html	2117	21.43	59.15	122.29	2.82
jpg	1073	10.86	335.08	351.12	8.10
doc	797	8.07	361.92	281.69	6.50
txt	788	7.98	553.89	426.23	9.83
xls	584	5.91	587.68	335.16	7.73
ppt	501	5.07	2110.94	1032.80	23.82
xml	353	3.57	132.59	45.71	1.10
gif	349	3.53	81.64	27.83	0.64
ps	208	2.11	764.85	155.36	3.58
csv	188	1.90	202.77	37.23	0.86
gz	128	1.30	628.64	78.58	1.81
log	99	1.00	170.80	16.51	3.81
unk	59	0.60	358.53	20.66	4.77
eps	40	0.41	516.59	20.18	4.66
png	39	0.39	312.85	11.92	2.75
others	141	1.77	343.62	58.72	1.35
Total	9876	100	449.44	4334.67	100

cryption keys used for locking up user data, we enabled the access to the Internet. However, considering ransomware may attempt to propagate themselves, we used a filtered host-only adapter to control their traffic and minimize their impact upon the host. After executing each ransomware, we revert the VM to a clean snapshot.

We conduct two experiments to measure ransomware's encryption time and examine whether ransomware attacks backup files (e.g., Volume Shadow Copies [183]) respectively. We describe their experimental setups as follows:

Encryption time. We placed a set of files (9,876 files in total) following the file-type distribution in a normal user's computer [184] in each VM. Table 6.2 shows the distribution of these files covering more than 18 unique file types. We run each ransomware sample and use the screenshot method described in [172] to examine their execution time. Specifically, we detect the changes to the screen of the virtual machine, screenshot the ransom notifications, and calculate the time it took for a ransomware to encrypt files and display a message on the screen to notify victim. To avoid false positives, we disabled Windows notification and manually examined each screenshotted notification.

Backup spoliation. To determine whether a ransomware also attacks file backups (especially the volume shadow copies), we created and enclosed several volume shadow copies on VMs. We deem a ransomware sample targets at backups if we observe the disappearance of these shadow copies.

6.2.2 Our Findings

Table 6.1 describes how fast ransomware encrypts data and notifies victim with a ransom screen (the 5th column), and whether ransomware attacks file backups (the last column). According to our study, ransomware typically displays ransom screen immediately after the encryption (sometimes even before the encryption has been completed). The notification procedure takes little time and most of the execution time of encryption ransomware is spent on the encryption part.

We observed that ten families complete the file encryption in less than an hour. For ransomware `CTB-Locker`, `JigSaw`, `Mobef` and `Petya`, their encryption takes even less than 20 minutes. Moreover, we discovered that some ransomware encrypt only small files or files with certain extensions. For example, `JigSaw` encrypts only files smaller than 10 MB, `CTB-Locker` only locks up files with certain extensions and `Petya` only encrypts a system's Master File Table (MFT) [185].

Observation 1: Ransomware typically locks up data rapidly and the size of the data encrypted is relatively small.

Implication: Ransomware would like to minimize the chances of being terminated and caught, or ransomware authors may want to collect ransom quickly.

Table 6.1 also shows that eight ransomware families attempt to delete backup files. Recall that we assigned ransomware samples the administrator privilege, which grants the ransomware the permission to destroy backups. We observed that some ransomware families attempt to bypass User Access Control if the privilege of deleting the backup files is not given. For instance, `cerber` [186] firstly escalates its privilege and then deletes Shadow Copies using the WMIC utility [187].

Observation 2: Ransomware variants proactively try to remove any means that victims could have to recover from the attack without paying the ransom.
Implication: Ransomware can obtain kernel privilege to terminate or destroy software-based defense systems such as explicit data backups.

6.3 Threat Model

As this work focuses on defending against encryption ransomware, we exclude the damage caused by non-encryption ransomware because they typically lock a computer system in a way which is not difficult for a knowledgeable person to reverse. For example, the ransomware `Trojan WinLock` trivially restricts access to the computer system and asks users to pay ransom to receive a code for unlocking their machines. In addition, we assume that encryption ransomware must be capable of restoring user data because inaccessibility and non-recoverability after paying ransom can significantly influence the rewards of ransomware attacks.

In this work, we only consider the situation where data on persistent storage are overwritten or deleted by ransomware. The targets not only include the files created by user-level applications (e.g., `.docx` and `.zip`) but also the metadata files that are required for file systems (e.g., Master File Table).

As discussed in § 6.2, some ransomware (e.g., `cerber`) will try to elevate its privileges to run as administrator. Once the privilege is given, the ransomware can disable or terminate any kernel-level defense mechanisms. As such, we do not assume the OS is trustworthy. Rather, we trust the SSD firmware. We believe this is a realistic assumption because (1) firmware is located within a storage controller, making it hardware-isolated to ransomware processes; (2) in comparison with the OS kernel, firmware has a small Trusted Computing Base (TCB) typically less vulnerable to malware attacks.

Overall, we believe this is a realistic threat model. First, it considers all types of ransomware attacks that aim to encrypt user data. Second, this threat model covers the cases in which the OS kernel is compromised such as WannaCry [165]. With the advance in ransomware defense, we believe ransomware authors will also actively exploit the vulnerabilities in the OS kernel. To the best of knowledge, this is the first work that explores malware defense solutions at the firmware level.

6.4 FlashGuard Design

6.4.1 Approach overview

To demand ransom, ransomware typically overwrites user files with encrypted contents. As described in § 2.1, SSDs naturally hold the old copies of the data overwritten by upper-level programs. As such, SSDs can be devised as a recovery system that holds data potentially manipulated by ransomware. Moreover, SSDs have an indirection layer at the firmware level to manage data. Building a recovery system on top of it, we can naturally isolate our recovery system from the OS, making it resistant to the attacks typically launched by malware to evade anti-virus. Taking advantage of the intrinsic characteristics of SSDs, we can also minimize the code space of our recovery system. As a result, SSDs naturally reduce the attack surface of our recovery system.

FlashGuard consists of two major components: a Ransomware-aware Flash Translation Layer (RFTL) and a tool for data recovery. The RFTL is designed for holding data potentially overwritten by encryption ransomware. The recovery tool is for victims to offset the damage to their files when they are aware of ransomware infection.

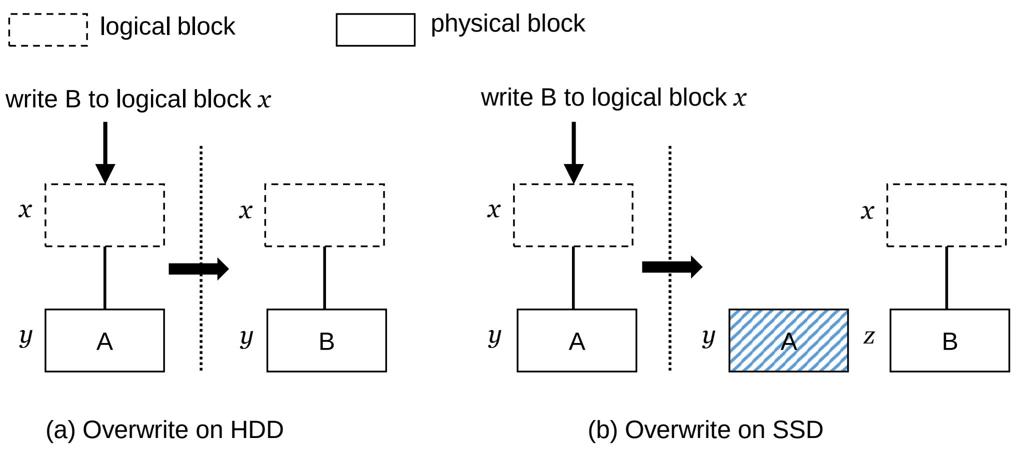


Figure 6.1: The fundamental difference between HDD and SSD for an overwrite operation. When a logical block x is overwritten, HDD will update the mapped physical block y with the new data B, while SSD will place the new data B on a free block z and garbage collect the block y later.

6.4.2 Ransom-Aware FTL

The FTL in modern SSDs maintains four data structures (see ①②③④ in Figure 6.2) to support out-of-place write and GC functionalities in practice. For each I/O access, the address mapping table ① is checked to translate the logical page address (LPA) to physical page address (PPA)[2]. For performance reason, the recently accessed mapping table entries ① are stored in a cache (using LRU policy in RFTL) located in a small and fast SRAM. If a mapping entry is not cached, FTL will check the Global Mapping Directory (GMD) ② to locate the corresponding translation page, and place the mapping entry in the address-mapping cache.

After certain storage operations, some pages in flash blocks may become invalid. To assist the GC operation, FTL usually uses the Block Validity Table (BVT) ③ to track the number of the valid pages in each block and to determine whether the block should be garbage collected or not. Since BVT is indexed in block-level granularity, it is small and can be fully stored in SRAM. Once a block is selected as the GC candidate, the Page

[2]The mapping table can be managed in page-level, block-level or hybrid block/page granularity. FlashGuard uses fine-granular and fully-associative page-level mapping.We believe it also works for other two mapping schemes.

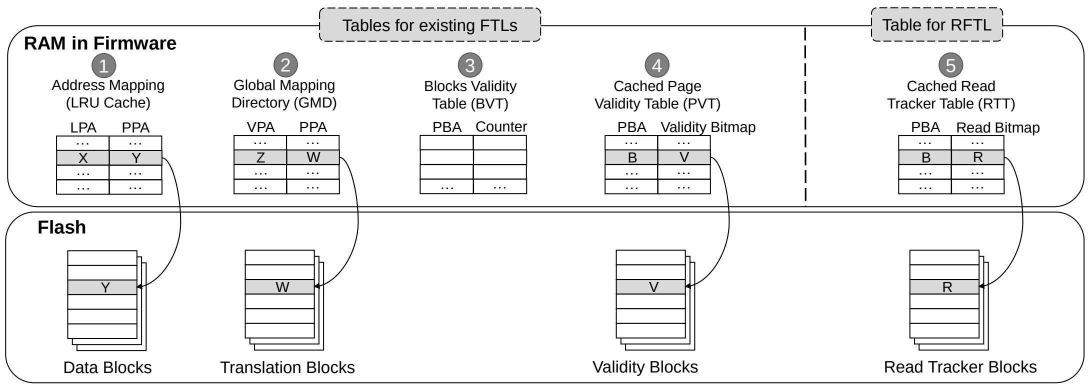


Figure 6.2: Overview of RFTL in FlashGuard. RFTL slightly modifies the existing FTLs by adding a read tracker table (RTT) to track whether a page has been read. Cooperating with other tables, RTT helps RFTL track the pages that could be encrypted by ransomware. LPA*: logical page address,* PPA*: physical page address,* VPA*: virtual page address,* PBA*: physical block address*.

Validity Table (PVT) 4 will be accessed to check which pages are valid and should be moved to a new flash block. The PVT could be a conventional page validity bitmap (PVB) or a recent optimized version which uses a log-structured merge-tree to reduce the space requirement of indexing the bitmap for each physical block [188]. In this work, we adopt the latter optimized design. We will use examples (see § 6.4.3 and § 6.4.4) to illustrate how these data structures work collaboratively with other components in FlashGuard.

To augment an SSD with the capability of counteracting ransomware attacks, a straightforward solution is to keep all the invalid pages in the physical device until ransomware is detected. This is infeasible for two major reasons. First, an SSD would quickly fill up with stale data, making the SSD unusable and causing unacceptable resource inefficiency. Second, the GC operations will be executed much more frequently to compact and collect free blocks, which affects the storage performance significantly.

Therefore, it is desirable that SSDs only hold the invalid pages having the old versions of the data manipulated by encrypted ransomware. According to our study (§ 6.2) and CryptoDrop [173], the size of the data encrypted by ransomware is typically less than a gigabyte. Holding such a small dataset will have negligible impact on a commodity SSD which usually has TBs of storage capacity.

However, it is challenging to track the pages manipulated by encryption ransomware since the underlying FTL does not have any semantic information of the received storage commands. To overcome this, we propose the Ransomware-aware FTL to track the invalid pages that could result from ransomware. RFTL augments the conventional FTLs with only one additional data structure: the **Read Tracker Table (RTT)** ⑤, which requires minimal modification to the existing firmware implementation.

We propose RTT based on the insights that ransomware typically read user data from disk, encrypt it and then overwrite or delete the original copy [172, 173]. Therefore, if a page has been read and then become invalid later, it could be a victim page encrypted by ransomware. We use the RTT ⑤ to track the page that has been read and leverage the PVT ④ to check whether it is valid or not, they provide us the hints to decide whether the page should be retained or not.

The RTT organizes entries in the way of the PVT ④, except that each entry in the RTT is a read bitmap[3] indexed by a block address. With the same optimization used in PVT, RTT enables RFTL to access and update the bitmap in an efficient manner. We use a buffer (4 KB in RFTL) to cache the frequently accessed RTT entries, which introduces only a small storage overhead in SRAM.

6.4.3 Read and Write Operations in RFTL

In this section, we describe how RFTL performs I/O requests in cooperation with the data structures discussed in § 6.4.2.

Read operation: When a read request to page *X* is received, RFTL first looks up the LPA in the cached address mapping table ①. If it is a cache miss, it searches the corresponding translation page in the GMD ② to locate the mapping entry for *X* in the translation page. During this process, the RFTL also places the mapping entry in the LRU cache for

[3]In FlashGuard, we use a bitmap carries 64 bits because each block contains 64 pages.

the address mapping ①. If it is a cache hit when accessing the cached address mapping table ①, the read operation will be issued directly. After locating the PPA of page *X* for serving the read operation, the RFTL updates the read bitmap in RTT ⑤ and sets the corresponding bit to 1 to indicate that the corresponding physical page has been read.

Write operation: When receiving a write request, RFTL performs the same address lookup procedure as for read in the cached address mapping table ①. If the mapping entry exists in the LRU cache, the data is written to a new free page, the address mapping entry is updated with the new PPA. Otherwise, a new mapping entry is created. The updated or newly created mapping entries are propagated to the translation pages and GMD ② when they are evicted from the cached address mapping table ①.

To enable the reverse mapping from the physical page in SSDs to logical page in file systems for data recovery, RFTL stores the metadata information of a page in its out-of-band (OOB) metadata. The commodity SSDs typically reserve 16-64 bytes OOB metadata for each physical page. FlashGuard leverages this space to store the metadata information about a page as shown in Figure 6.3.

The OOB metadata includes (1) the LPA mapped to this physical page, (2) the previous PPA (P-PPA) mapped to the current LPA (it is used when a page is overwritten and it enables FlashGuard to identify all the old pages mapped to the same LPA), (3) the timestamp when the page is written, and (4) a **Retained Invalid Page (RIP)** bit to indicate whether this page is invalid and also potentially manipulated by encryption ransomware. We will discuss how these metadata can be leveraged for data recovery in § 6.4.5.

6.4.4 Garbage Collection in RFTL

Garbage collection is an essential component in SSDs to provide free blocks for future use by compacting the used flash blocks and also guarantee all the flash blocks age uniformly to extend SSD lifetime. It also plays a critical role in preserving the old copies (invalid pages) of the data manipulated by ransomware. When GC executes, it first selects the

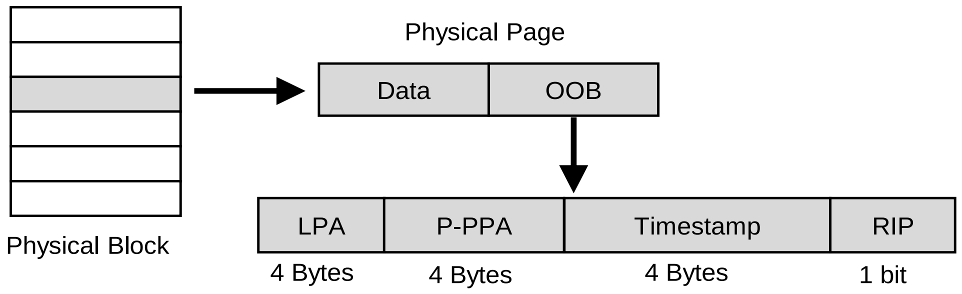


Figure 6.3: The out-of-band (OOB) metadata in each physical page. It includes the LPA mapped to this physical page, the previous physical page address (P-PPA) mapped to the current LPA, the timestamp when the page is written, the retained invalid page (RIP) bit indicating whether this page should be retained if it becomes invalid.

candidate blocks, move the valid pages in those blocks to new free blocks and then erases these candidate blocks for future use.

Key idea: To make an SSD capable of holding data for recovery, we propose a new GC scheme in RFTL. In particular, RFTL examines whether an invalid page in a GC candidate block has been read. The GC will retain those pages. The invalid pages that have never been read will be discarded/erased. The intuition behind this is that ransomware needs to read data from an SSD before performing encryption, the pages that have never been read cannot be a piece of damaged data caused by ransomware.

We describe the new GC scheme in Algorithm 1 and discuss its procedure as follows.

GC procedure: When the number of free blocks in an SSD is below a threshold (10% - 40% of all the flash blocks in commodity SSDs), GC will be triggered to free space. The existing GC typically employs a greedy algorithm for selecting the GC candidate blocks. More specifically, it chooses the block with the least number of valid pages. This selection procedure can be quickly completed by looking up the BVT ③ that tracks the number of valid pages for each block.

Different from the current block selection scheme for GC candidate, RFTL takes those retained invalid pages (RIP has set to be *Reserved* in Algorithm 1) as valid pages. Therefore, the GC in RFTL selects the block with the least number of both valid pages and retained invalid pages. Such a GC scheme implies that a block with multiple invalid pages retained for recovery may delay its collection (see Figure 6.4), which could reduce the additional GC overhead caused by copying retained invalid pages to new free blocks.

Algorithm 1 Garbage Collection in RFTL

```
Require: $ReserveTime$ = the time threshold for retaining invalid pages
         $Reserved$ = the bit flag indicating a page is invalid but retained

1: Select the candidate block for GC  ▷
   the candidate block has the least number of valid pages and retained invalid pages.
2: Check PVT to find valid pages in candidate block
3: for each valid page do
4:     Check page's OOB metadata
5:     Verify page's validity
6:     if page is valid then
7:         Copy page to a new free page
8:         Update address mapping entry
9: for each invalid page do
10:    Check read tracker table (RTT)
11:    if page has been read then
12:        Check page's RIP bit
13:        if RIP == Reserved then
14:            page_timestamp← timestamp in page's OOB metadata
15:            if current_time - page_timestamp < ReserveTime then
16:                Clear this page's read bit in the bitmap of RTT
17:                Copy page and its OOB metadata to a new free page
18:                Set the new page's read bit in RTT to 1 (Read)
19:            else
20:                Discard and reclaim this page
21:                Clear this page's read bit in the bitmaps of RTT
22:        else
23:            Set metadata (timestamp←current_time, RIP←Reserved)
24:            Clear this page's read bit in the bitmaps of RTT
25:            Copy page and its OOB metadata to a new free page
26:            Set the new page's read bit in RTT to 1 (Read)
27:    else
28:        Discard and reclaim this page
```

Once a candidate flash block is selected, RFTL checks the PVT (4) and searches the valid pages in that block. Since lazy policies are usually adopted to update the PVT, the information in PVT might be outdated. To address this issue, RFTL double checks each valid page indicated by PVT by looking at its OOB metadata. It retrieves the LPA from the OOB metadata and looks up the corresponding PPA through the address mapping table

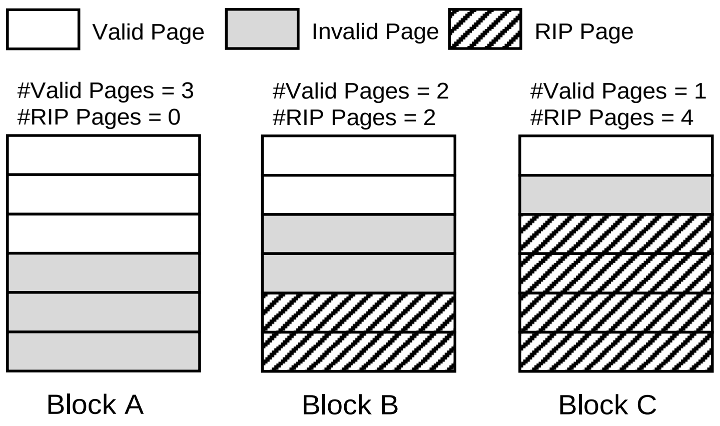


Figure 6.4: An example of candidate block selection in state-of-the-art GC vs. RFTL's GC. Traditionally, block *C* is selected, as the number of the valid pages is the least. In RFTL, block *A* is selected, since RFTL counts the retained invalid pages (RIP) as valid pages.

1. If the PPA retrieved is the same as the PPA of the page, RFTL deems it valid.

Given a candidate flash block, RFTL migrates its valid pages and retained invalid pages to new free blocks. For those valid pages, their corresponding mapping entries in 1 are updated and pointed to the new PPAs. The retained invalid pages will be kept in the flash device for a certain time (a configurable threshold, 20 days in FlashGuard by default).

RFTL uses the timestamp stored in the page's OOB metadata to calculate how long this page has been retained. Once the interval between the timestamp in the OOB metadata and the current time is larger than the configured threshold, the page will be erased and reclaimed. Otherwise, both the page and its OOB metadata are copied to a free page, so that RFTL will keep retaining this invalid page in the SSD until it is expired when it is selected by GC next time (see line 13-21 in Algorithm 1).

For an invalid page *X* whose RIP bit is not set and has been read as indicated in RTT 5, it is treated as a page to be retained and will be copied to a free page *Y*. RFTL runs the GC procedure for this type of invalid pages as follows:

First, RFTL prepares the OOB metadata for the new page *Y*: the RIP bit is set to be *Reserved*, the timestamp is set to the current time (so that the content of this page will be conservatively retained for a certain period of time), the LPA and P-PPA are kept the same as in page *X*'s OOB metadata. Second, RFTL copies the page *X* and its OOB metadata into the free page *Y* in a new free block. Third, page *X*'s read bit in RTT is cleared and page *Y*'s read bit in RTT is set to 1 (indicating the content of this page has been read). Finally, page *X* is garbage collected (see line 23-26 in Algorithm 1). After this procedure, RFTL moves the retained invalid page to a new location and keeps holding it in the flash device.

For an invalid page which has never been read, RFTL will discard and garbage collect it (see line 28 in Algorithm 1), which is handled in the same way as in traditional SSDs.

Impact on SSD performance: The GC scheme in RFTL keeps the basic and essential procedures in the state-of-the-art FTLs, including candidate block selection and valid page movement. In our design, the overhead is introduced by copying retained invalid pages. The RFTL takes retained invalid pages as valid pages and the GC on the blocks carrying these pages will be delayed (see Figure 6.4). Meantime, RFTL also needs to ensure all the blocks age at the same rate (i.e., wear leveling) to extend the lifetime of the SSD. The blocks that have retained invalid pages, would still be selected as candidate blocks for GC, thus additional overhead would be introduced.

However, these blocks will not be frequently garbage collected because of the throttling and swapping mechanisms in the existing GC design: cold data (i.e., not frequently accessed data) is migrated to old blocks (i.e., blocks that experience more wear). The blocks which have many retained invalid pages will be accessed less frequently, and the chance that they will be collected shortly is small. In addition, if all the pages in a GC candidate block are invalid and will be retained, RFTL does not garbage collect them.

Impact on SSD lifetime: The SSD lifetime is determined by the wear-leveling and write traffic to the device. The GC in existing FTLs uses a greedy policy for candidate block selection, which always selects the block having the least number of valid pages. Such a GC policy provides maximal GC efficiency (i.e., the least number of page migrations), and the throttling and swapping mechanisms are used to balance the wear between blocks. RFTL employs these techniques. Moreover, recent research [189, 10] on SSDs discloses that a

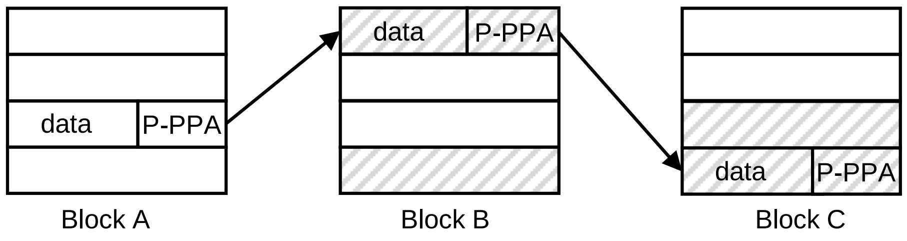


Figure 6.5: FlashGuard restores all the overwritten pages by travelling back to their previous versions with the previous physical page address stored in each page's OOB metadata.

relaxed wear-leveling can provide guaranteed SSD lifetime. Experiments with a variety of real-world workloads demonstrate that RFTL has minimal impact on SSD lifetime in § 6.6.

6.4.5 Data Recovery

To restore the invalid pages retained in an SSD when victims are aware of the ransomware infection, users can remove the SSD device and plug it into another clean and isolated computer for data recovery in case ransomware would attack the data recovery procedure. FlashGuard first checks the RTT ⑤ to locate all the pages that have been read recently. These pages are the candidate pages that may contain the user's stale data. As the RTT is cached in firmware RAM, this checking procedure is fast. To read the retained invalid page, FlashGuard checks the RIP bit in OOB metadata of each candidate page. If the RIP bit flag is set, the page is read from flash. Otherwise, RFTL will check the address mapping table ① to figure out whether this page is valid or not. If it is invalid, the page is read from flash as well, since it is possible that this page is also a victim page.

FlashGuard accelerates the procedure of reading invalid pages retained from a flash drive by leveraging the internal parallelism in an SSD. Parallelizing the read of pages from the flash drive, the recovery will not take too much time (see the evaluation in § 6.6).

Once these invalid pages retained are read from a flash drive, the LPAs, P-PPAs, and timestamps stored in these pages' OOB metadata will be used to reconstruct the user files. FlashGuard can use the previous physical page address (P-PPA) stored in each page's OOB metadata to reverse an invalid page to its previous versions as shown in Figure 6.5. In order to maintain data locality for performance reasons, modern file systems usually manage the logical address space in a contiguous manner, and also flash controllers buffer storage operations to exploit temporal and spatial locality [190]. With these insights, the recovery tool in FlashGuard sorts the retained invalid pages with their LBAs and timestamps to reconstruct the original file. As a page could have been overwritten several times by either ransomware or trusted users, the recovery tool can reverse it to any older versions and allow users to verify the content.

Since FlashGuard retains all the versions of the invalid pages in flash device, many other existing data recovery tools can also be leveraged to reconstruct user files (if there is no information available for data locality). For example, some recovery tools can read the first few bytes in each page to figure out the file type (e.g., `.ppt` or `.doc` file), and then use the defined layout for the file type to recover the data [191].

6.4.6 Metadata Recovery

As all the data structures (see Figure 6.2) are cached in firmware RAM, the cached data could be lost if a power failure happens. FlashGuard maintains their durability by leveraging the metadata recovery and check-pointing techniques that have been adopted in the state-of-the-art FTLs [188, 9]. RFTL identifies the recently written flash block by checking its OOB metadata (which includes timestamp as shown in Figure 6.3) and use the metadata information to recover the cached entries such as the address mapping table ①. For the data structure RTT ⑤ that tracks the recent reads, RFTL recovers it to the latest checkpointed states. For the blocks that have been written after the checkpoint, RFTL identifies their older versions (with P-PPA in OOB metadata) and conservatively marks them as 'read' in RTT.

An alternative solution is to use a battery or large capacitor to preserve the cached entries and persist them before power turns off, which simplifies the metadata recovery procedure significantly. We wish to take this solution as the future work.

6.5 FlashGuard Implementation

We implement FlashGuard on a 1TB programmable SSD with a state-of-the-art page-level FTL. Each block in the SSD has 64 pages and each page is 4 KB with 16 bytes of OOB metdata. The programmable SSD provides basic I/O control commands to issue read, write and erase operations against the physical flash device. The RFTL for FlashGuard is implemented based on the page-level FTL. FlashGuard is implemented with 5,718 lines of `C` code on top of the flash device. The SSD is over-provisioned with 15% of its full capacity by default, and the garbage collection is running in the background.

We also implement a recovery tool that can read all the retained invalid pages from the flash device and organize them in the manner as discussed in 6.4.5. The recovered data will be written back to SSD after having verified by users.

6.6 Results and Analysis

6.6.1 Experimental Setup

To evaluate the capability of FlashGuard to recover data encrypted by ransomware, we use the 1,477 ransomware samples from 13 families as shown in Table 6.1. These samples are executed with the same experimental setup as described in § 6.2.1. Once a ransom screen appears, we start to run the recovery tool to recover encrypted data.

To evaluate the impact of FlashGuard on storage performance and SSD lifetime, we reply five sets of I/O traces collected from a variety of real-world applications (see Table 6.3): (1) the storage traces collected from enterprise servers running different applications (e.g., media server, research project management systems, and print server) in Microsoft Research at Cambridge for one week [192]; (2) the storage traces collected from machines running in a department at FIU for twenty days [193]; (3) the database workload traces of running TPC-C benchmark and TPC-E benchmark for eight days [107]; (4) the storage traces of running IOZone benchmark [194] for ten days; (5) the storage traces of running

Table 6.3: A variety of real-world application workloads used for evaluating FlashGuard. R: *Read*, W: *Write*.

Workload		Description	IO Pattern
FIU IO Trace	online-course	course management system of a department using Moodle	R:22.3%, W:77.7%
	webmail	web interface to the mail server	R:18.0%, W:82.0%
	home	research group activities: developing, testing, experiments, etc.	R:0.9%, W:99.1%
	mailserver	department mail server traces	R:8.6%, W:91.4%
	web-research	research projects management using Apache web server	R:0.001%, W:99.999%
	web-users	web server hosting faculty, staff and graduate student web sites	R:10.0%, W:90.0%
Microsoft Servers	hm	hardware monitoring	R:35.5%, W:64.5%
	mds	media server	R:11.9%, W:88.1%
	prn	print server	R:10.8%, W:89.2%
	proj	project directories	R:12.5%, W:87.5%
	prxy	firewall/web proxy	R:3.1%, W:96.9%
	rsrch	research projects	R:9.3%, W:90.7%
	src	source control	R:56.4%, W:43.6%
	stg	web staging	R:15.2%, W:84.8%
	ts	terminal server	R:17.6%, W:82.4%
	usr	user home directories	R:40.4%, W:59.6%
	wdev	test web server	R:20.1%, W:79.9%
	web	web/SQL server	R:29.9%, W:70.1%
Others	postmark	mail servers	R:83.2%, W:16.8%
	IOZone	filesystem benchmark	R:0.0%, W:100.0%
	TPC-C	online transaction processing	R:75.1%, W:24.9%
	TPC-E	OLTP of a brokerage firm	R:91.8%, W:8.2%

the Postmark benchmark [195] for ten days. For each experiment, we first run 50 million mixed read and write operations to warm up the system and then replay each trace to collect the performance results.

6.6.2 Efficiency on Data Recovery

FlashGuard performs the procedure of data recovery following the approaches discussed in § 6.4.5. Once the recovery procedure is finished, we manually verify the pages that have been read from the flash device. All the old versions of the encrypted data can be found in the flash pages recovered by FlashGuard. Figure 6.6 displays the average size of the data

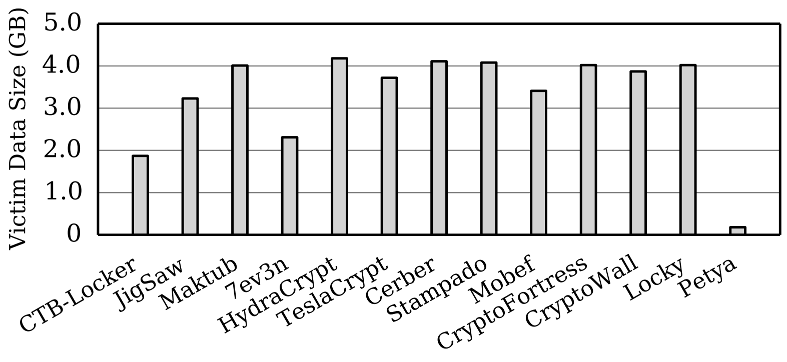


Figure 6.6: The total size of the data encrypted by each ransomware family.

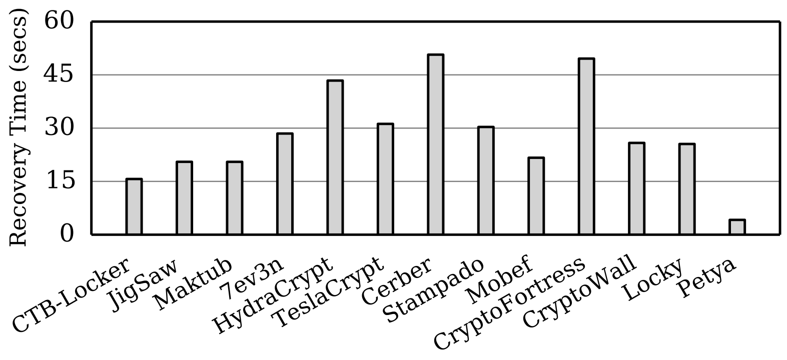
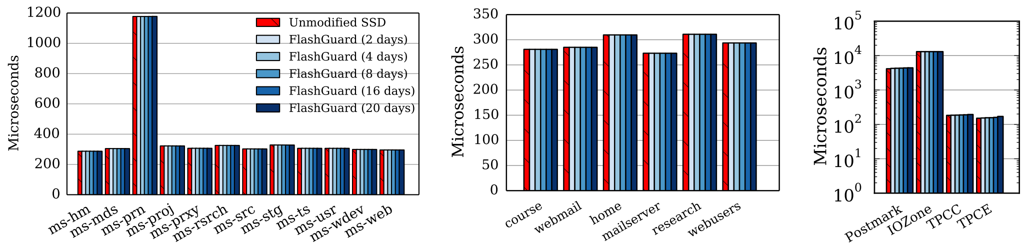


Figure 6.7: The time of restoring the data that have been encrypted by ransomware.

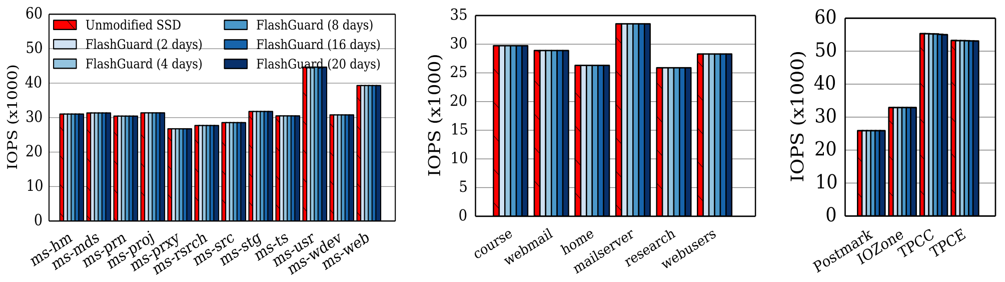


(a) Server Storage in Enterprise (b) Server Storage in University (c) Other Workloads

Figure 6.8: The average latency of running real-world workloads with FlashGuard vs. Unmodified SSD. The time of holding retained invalid pages in FlashGuard ranges from 2 days to 20 days. FlashGuard's average latency is almost the same as that of the unmodified SSD for a variety of workloads.

recovered from infection by different families, which ranges from 0.2 GB to 4.1 GB.

The execution time of restoring the encrypted data ranges from 4.2 seconds to 49.6 seconds as shown in Figure 6.7. FlashGuard leverages the internal parallelism in flash



(a) Server Storage in Enterprise (b) Server Storage in University (c) Other Workloads

Figure 6.9: The average throughput of running real-world workloads with FlashGuard vs. Unmodified SSD. FlashGuard has negligible impact on the I/O throughput for most of these workloads.

device to access the retained invalid pages in parallels. It is noted that the recovery time is not proportional to the victim data size, as the retained invalid pages are not evenly distributed across the parallel elements (i.e., chip-level packages) in flash device. However, the current recovery approach used in FlashGuard is much faster than the naive approach that scans the whole flash device (which takes 707.7 seconds).

Most of the ransomware samples do not read and overwrite user data many times, it takes little time for FlashGuard to reconstruct the original files. Although encryption ransomware would attack user data with the knowledge of SSD properties, for instance, a ransomware can keep reading and overwriting user data to an SSD, FlashGuard can still restore the encrypted data since it retains all their older versions.

6.6.3 Impact on Storage Performance

To understand the impact of FlashGuard on storage performance, we begin with the default over-provisioning (15% of the SSD's full capacity) and run the acknowledged storage traces collected from real-world applications (see Table 6.3). We assume all the writes are encrypted, which means all the invalid pages that have been read will be retained in SSD. The time of holding these invalid pages ranges from 2 days to 20 days, the storage latency and throughput are reported in Figure 6.8 and Figure 6.9.

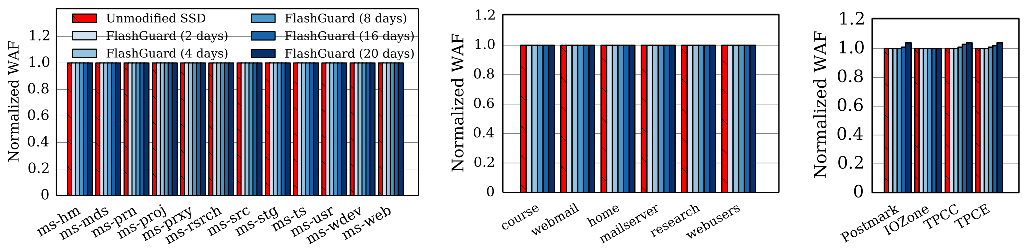
Table 6.4: The additional page movements (%) for retaining invalid pages in FlashGuard over the time period from 2 to 20 days. For the workloads that do not incur additional page movements, they are not shown in the table.

	Enterprise Storage			Other Workloads		
days	hm	usr	web	Postmark	TPCC	TPCE
2	0.0	0.0	0.1	8.5	8.1	5.3
4	0.0	0.1	0.1	9.1	8.3	5.5
8	0.5	0.1	0.4	9.7	8.8	5.7
16	0.5	0.3	0.6	9.7	8.8	5.7
20	0.5	0.3	0.8	9.7	8.8	5.7

For most of the workloads, the average latency of running them on FlashGuard is almost the same as that of running them on the unmodified SSD as shown in Figure 6.8. For I/O-intensive workloads including Postmark, TPCC, and TPCE, FlashGuard increases the average latency by up to 6.1%. As the time of holding retained invalid pages is increased, the average latency is slightly increased. In terms of I/O throughput, FlashGuard has trivial impact (up to 0.6%) as shown in Figure 6.9. FlashGuard does not introduce much performance overhead for three reasons:

First, the RFTL in FlashGuard delays the GC execution on the flash blocks having retained invalid pages by counting them as valid pages, which reduces the chances of moving retained invalid pages. Second, the GC is executed in the background, which allows FTLs schedule GC during the idle time of flash controller, further reducing the performance interference caused by GC. Third, the existing I/O schedulers and FTLs provide decent GC efficiency (i.e., the valid page movements during GC procedure) for many workloads. When all the pages on a flash block are invalid, the flash block will be erased without incurring any page movement. In FlashGuard, no additional page movement is required for a flash block whose pages are all retained invalid pages.

To further understand the performance overhead of FlashGuard, we profile the GC events and collect statistics on the number of additional page movements. As shown in Table 6.4, all the FIU workloads incur no additional page movements, although the time



(a) Server Storage in Enterprise (b) Server Storage in University (c) Other Workloads

Figure 6.10: The normalized write amplification factor (WAF) of FlashGuard compared to Unmodified SSD (lower is better).

of holding the retained invalid pages is set to be 20 days. For the workloads running in enterprise servers, up to 0.8% of the page movements are contributed by retaining invalid pages. For these I/O intensive workloads such as Postmark, TPCC and TPCE, more page movements are introduced. Since the IOZone traces are write-only, no pages are required to be retained in FlashGuard.

6.6.4 Impact on SSD Lifetime

As flash block has limited endurance, it is necessary to ensure FlashGuard offers acceptable SSD lifetime. We use the write amplification factor (WAF) [196] to evaluate the actual amount of physical write traffic to that of logical write traffic. Larger WAF means that SSD suffers from more write traffic, indicating that the SSD would last for a shorter time.

As shown in Figure 6.10, For the storage workloads running in enterprise and university, the WAF of FlashGuard is the same as that of unmodified SSD. For IO-intensive workloads, the WAF is increased by up to 4%, this is because FlashGuard incurs additional page movements for retaining invalid pages. As the time of holding the retaining invalid pages in the flash device is increased, the WAF is slightly increased. However, this is less of a concern. For an SSD that usually has a lifetime of 160 - 250 weeks, the slightly increased WAF reduces its lifetime by only one or two weeks, which is acceptable in practice.

6.7 Security Analysis and Discussion

According to our study in § 6.2, few encryption ransomware was developed considering the SSD characteristics. In this section, we discuss the possible ransomware attacks against FlashGuard and potential research directions in the future.

6.7.1 Exploiting storage capacity

To support data recovery, FlashGuard holds the data potentially encrypted by ransomware and prevents them from being discarded by garbage collection. Intuition suggests an attacker can exploit storage capacity and keep writing to occupy the available space in SSD, forcing FlashGuard to release its hold. Another potential attack is that a ransomware keeps reading and overwriting data to the SSD in order to cause FlashGuard to retain a large amount of garbage data. In practice, such attacks are in vain. FlashGuard refuses to release data hold if the lifespan of the holding data has not yet expired, even though the SSD is fully occupied. When such an incident happens, FlashGuard will stop issuing IO requests when the SSD is full, resulting in the failure of filesystem operations in OS. Therefore, even though ransomware has the kernel privilege, it cannot prevent a user from noticing abnormal events.

6.7.2 Timing attacks

Time is critical for both security and performance of FlashGuard. The longer FlashGuard holds stale data, the more overhead it might impose on I/O operations. To obtain high storage performance, a user might set the lifespan of holding data relatively short. In this way, the user is exposed to the threat of ransomware attacks in that ransomware could slow down the pace of encrypting data and notifying victims.

As discussed in § 6.2, ransomware variants have been evolving to lock up user data and collect ransom rapidly to prevent from being caught. In § 6.6, we have already demonstrated that FlashGuard typically incurs only negligible overhead to regular I/O operations, even though we set the lifespan of holding data for 20 days. This implies FlashGuard is effective in defending against the aforementioned ransomware attacks. This is because it not only significantly increases the risk of ransomware of being caught but also thwarts ransomware authors from gaining rewards rapidly. In the future, we wish to explore new detection and defense mechanisms against timing attacks.

6.7.3 Secure deletion

FlashGuard retains overwritten contents for the sake of recovery. Intuitively, this design contradicts to the objective of secure deletion [11, 197, 198, 199, 200], which requires irrecoverable data deletion from a physical medium. However, we believe FlashGuard is compatible with secure deletion. In particular, FlashGuard can use a user-specified encryption key to encrypt the stale data potentially overwritten by ransomware. In this way, a user can still perform data recovery but not worrying about data leakage because adversaries cannot restore “securely deleted data” without the encryption key. As future work, we will develop this solution, making FlashGuard compatible with secure deletion.

6.8 Summary

In this chapter, we present FlashGuard, a ransomware-tolerant SSD that retains the data potentially encrypted by ransomware in SSD. With FlashGuard, we demonstrate that victims can efficiently reinstate the damage to their files caused by encryption ransomware. The design of FlashGuard takes advantage of the intrinsic flash properties. We show FlashGuard only introduces negligible overhead to regular storage operations and has a trivial impact on SSD lifetime. In comparison with existing detection mechanisms against ransomware, FlashGuard is the first firmware-level defense system, it is naturally resistant to the ransomware that exploits kernel vulnerabilities or runs with the kernel privilege.

CHAPTER 7
CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

Flash memories have been developed to bridge the gap between DRAM and hard disk drive. However, the system software which performs as the abstraction layer between hardware and applications has not adapted rapidly to the hardware evolution. Such a gap not only hinders the exploitation of the power and flexibility of hardware devices but also causes resource inefficiency and sub-optimal performance. This dissertation revisits the storage system design for flash memory with a holistic approach from the system level to the device level and proposes new solutions to improve the performance isolation, software latency, energy-efficiency, and system security of the flash-based storage systems respectively.

Chapter 3 investigates the problem of performance isolation between multiple tenants sharing the SSD in modern data centers and identifies that there is a fundamental tussle between resource isolation and the lifetime of the flash device – existing SSDs aim to uniformly age all the regions of flash and this hurts isolation. To improve performance isolation, we propose utilizing flash parallelism to provide hardware isolation between applications by running them on dedicated flash chips. Moreover, we propose allowing the wear of different flash chips to diverge at fine time granularities in favor of isolation and adjusting the wear imbalance at a coarse time granularity in a principled manner. The experimental results show that the new SSD wears uniformly while the tail latency of storage operations is decreased significantly.

Chapter 4 further discusses improving the storage performance with the new ways of using SSDs. It presents a solution that combines all the indirection layers of the virtual memory system, file system, and the flash translation layer into a unified layer while preserving the properties of each layer. Specifically, it combines all the address translation into page tables in virtual memory system without altering the guarantees of the file systems and the flash translation layer. It uses the state in the memory manager and the page tables to perform sanity and permission checks. Such a unified address translation layer reduces critical-path latency and improves DRAM caching efficiency.

Chapter 5 focuses on improving the energy-efficiency of the flash-based storage system on resource-constrained devices such as wearables. It proposes a fast and energy-efficient in-memory storage system based on battery-backed DRAM and an efficient means to offload energy-intensive tasks to the connected phone. This dissertation uses the battery-backed DRAM as non-volatile memory to avoid energy-intensive storage operations and leverages the low-power network connectivity available on wearables to trade the resources on the phone for the wearable.

Chapter 6 targets the security aspect of the modern storage systems with a focus on the defense against encryption ransomware. This dissertation presents a ransomware-tolerant SSD which has a firmware-level recovery system that allows quick and effective recovery from encryption ransomware. It leverages the intrinsic flash properties to present a malware defense solution at the firmware level. Based on the observation that the existing SSD already performs out-of-place write in order to mitigate the long erase latency of flash memories, therefore, when a page is updated or deleted, the older copy of that page is anyways preserved in the SSD. The proposed solution requires minimal modification to the firmware implementation.

7.2 Future Work

7.2.1 Achieving Predictable Storage Performance

FlashBlox has demonstrated that leveraging the flash parallelism to provide hardware isolation can significantly reduce the tail latency for multi-tenant applications. As the underlying hardware components and their management are exposed to upper system software and applications, all the storage events such as IO scheduling and garbage collection can be handled in a transparent manner. Therefore, it is feasible to achieve highly predictable storage performance in flash-based storage systems. Furthermore, FlashBlox discusses the scenario that multiple applications share the same SSD on a single machine. It can be extended and integrated with multi-resource data center schedulers to help applications obtain predictable end-to-end performance.

7.2.2 Unifying Management for Memory and Storage

We are at the cusp of memory and storage technology revolution; increasing diversity of memory and storage devices is becoming evident which brings challenges to the memory and storage management. FlashMap took Flash as an example and has shown the benefits of unifying the memory and storage software. The OS memory manager transparently manages data placement between flash and DRAM by discerning access pattern information from the virtual memory system. We envision future hardware and software systems that can enable intelligent data placement across heterogeneous memory technologies. Similar to the approach discussed in FlashMap, page tables can be leveraged for moving physical pages around memory technologies for two benefits: the application's view via virtual memory is not changed and more importantly, expensive hardware-level mapping tables are not needed.

7.2.3 Improving Storage Security on Various Platforms

FlashGuard leverages the intrinsic properties of Flash to protect against encryption ransomware. This approach can be applied to any kind of flash-based storage devices to protect different computing platforms against encryption ransomware. A typical example is the mobile device which has used Flash to store personal user data for decades. As the flash devices used on mobiles (e.g., eMMC) share the same intrinsic properties as that on personal computers and enterprise servers (e.g., SSDs) [8, 40, 201], our approach can be deployed on the mobile platform to enhance its storage system and protect users against the ever-increasing threat of mobile ransomware such as `Simplocker` [202, 203, 204, 205].

FlashGuard retains overwritten contents for the sake of recovery. Intuitively, this design contradicts with the objective of secure deletion [11, 197, 198, 199, 200], which requires irrecoverable data deletion from a physical medium. However, we believe FlashGuard is compatible with secure deletion. In particular, FlashGuard can use a user-specified encryption key to encrypt the stale data potentially overwritten by ransomware. In this way, a user can still perform data recovery but not worrying about data leakage because adversaries cannot restore "securely deleted data" without the encryption key. As future work, we wish to develop this solution, making FlashGuard compatible with secure deletion.

REFERENCES

[1] Fusion-io: ioDrive Octal,
http://www.fusionio.com/products/iodrive-octal/.

[2] N. Li, H. Jiang, D. Feng, and Z. Shi, "PSLO: Enforcing the Xth Percentile Latency and Throughput SLOs for Consolidated VM Storage," in *Proc. EuroSys'16*, London, United Kingdom, Apr. 2016.

[3] A. Singh, M. Korupolu, and D. Mohapatra, "Server-Storage Virtualization: Integration and Load Balancing in Data Centers," in *Proc. SC'08*, Austin, Texas, Nov. 2008.

[4] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu, "IOFlow: A Software-Defined Storage Architecture," in *Proc. SOSP'13*, Farmington, PA, Nov. 2013.

[5] MongoDB, http://mongodb.org.

[6] MongoDB: Memory Mapped File Usage,
http://docs.mongodb.org/manual/faq/storage/.

[7] LMDB, http://symas.com/mdb/.

[8] J. Li, A. Badam, R. Chandra, S. Swanson, B. Worthington, and Q. Zhang, "On the Energy Overhead of Mobile Storage Systems," in *FAST'14*, Santa Clara, CA, Feb. 2014.

[9] J. Huang, A. Badam, M. K. Qureshi, and K. Schwan, "Unified Address Translation for Memory-Mapped SSD with FlashMap," in *Proc. ISCA'15*, Portland, OR, Jun. 2015.

[10] J. Huang, A. Badam, L. Caulfield, S. Nath, S. Sengupta, B. Sharma, and M. K. Qureshi, "FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs," in *Proc. FAST'17*, Santa Clara, CA, Feb. 2017.

[11] M. Y. C. Wei, L. M. Grupp, F. E. Spada, and S. Swanson, "Reliably erasing data from flash-based solid state drives.," in *Proc. 9th USENIX Conference on File and Storage Technologies (FAST'11)*, 2011.

[12] SSD prices plummet again, Close in on HDDs,
`http://www.pcworld.com/article/3040591/storage/ssd-prices-plummet-again-close-in-on-hdds.html`.

[13] N. Agarwal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," in *Proc. USENIX ATC*, Boston, MA, Jun. 2008.

[14] F. Chen, R. Lee, and X. Zhang, "Essential Roles of Exploiting Internal Parallelism of Flash Memory based Solid State Drives in High-Speed Data Processing," in *Proc. HPCA'11*, San Antonio, Texas, Feb. 2011.

[15] M. Jung, E. H. W. III, and M. Kandemir, "Physically addressed queueing (paq): Improving parallelism in solid state disks," in *ISCA'12*, Portland, OR, 2012.

[16] J. Kim, D. Lee, and S. H. Noh, "Towards SLO Complying SSDs Through OPS Isolation," in *Proc. FAST'15*, Santa Clara, CA, Feb. 2015.

[17] M. Hao, G. Soundararajan, D. Kenchammana-Hosekote, A. A. Chien, and H. S. Gunawi, "The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments," in *Proc. FAST'16*, Santa Clara, CA, Feb. 2016.

[18] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving Resource Efficiency at Scale," in *Proc. ISCA'15*, Portland, OR, Jun. 2015.

[19] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations," in *Proc. MICRO'11*, Porto Alegre, Brazil, Dec. 2011.

[20] B. Trushkowsky, P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "The SCADS Director: Scaling a Distributed Storage System Under Stringent Performance Requirements," in *Proc. FAST'11*, Santa Clara, CA, Feb. 2016.

[21] D. Shue, M. J. Freedman, and A. Shaikh, "Performance Isolation and Fairness for Multi-Tenant Cloud Storage," in *Proc. OSDI'12*, Hollywood, CA, Oct. 2012.

[22] Throtting IO with Linux,
`https://fritshoogland.wordpress.com/2012/12/15/throttling-io-with-linux`.

[23] Token Bucket Algorithm,
`https://en.wikipedia.org/wiki/token_bucket`.

[24] Matias Bjorling and Javier Gonzalez and Philippe Bonnet, "LightNVM: The Linux Open-Channel SSD Subsystem," in *Proc. USENIX FAST'17*, Santa Clara, CA, Feb. 2016.

[25] S. Lee, M. Liu, S. Jun, S. Xu, J. Kim, and Arvind, "Application-Managed Flash," in *Proc. FAST'16*, Santa Clara, CA, Feb. 2016.

[26] J. Ouyang, S. Lin, S. Jiang, Y. Wang, W. Qi, J. Cong, and Y. Wang, "SDF: Software-Defined Flash for Web-Scale Internet Storage Systems," in *Proc. ACM ASPLOS*, 2014.

[27] ArangoDB, `https://www.arangodb.com/`.

[28] MapDB, `http://www.mapdb.org/`.

[29] MonetDB, `http://www.monetdb.org`.

[30] MongoDB Deployment, `http://lineofthought.com/tools/mongodb`.

[31] TCMalloc Memory Allocator, `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[32] A. Badam and V. S. Pai, "SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy," in *Proc. 8th USENIX NSDI*, 2011.

[33] X. Ouyang, N. S. Islam, R. Rajachandrasekar, J. Jose, M. Luo, H. Wang, and D. K. Panda, "SSD-Assited Hybrid Memory to Accelerate Memcached over High Performance Networks," in *Proc. 41st ICPP*, Sep. 2012.

[34] R. Pearce, M. Ghokale, and N. M. Amato, "Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory," in *Proc. SC'10*, New Orleans, LA, Nov. 2010.

[35] B. Van Essen, R. Pearce, S. Ames, and M. Gokhale, "On the Role of NVRAM in Data-Intensive Architectures: An Evaluation," in *Proc. IPDPS'12*, Shanghai, China, May 2012.

[36] C. Wang, S. S. Vazhkudai, X. Ma, F. Mang, Y. kim, and C. Engelmann, "NVMalloc: Exposing an Aggregate SSD Store as a Memory Parition in Extreme-Scale Machines," in *Proc. IPDPS'12*, Shanghai, China, May 2012.

[37] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "De-indirection for Flash-based SSDs with Nameless Writes," in *Proc. 10th USENIX FAST*, Feb. 2012.

[38] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn, "DFS: A File System for Virtualized Flash Storage," *ACM Trans. on Storage*, vol. 6, no. 3, 14:1–14:25, 2010.

[39] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings," in *Proc. ASPLOS'09*, Washington, DC, 2009.

[40] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting Storage for Smartphones," in *FAST'12*, San Jose, CA, Feb. 2012.

[41] W.-H. Kim, B. Nam, D. Park, and Y. Won, "Resolving Journaling of Journal Anomaly in Android IO: Multi-version B-tree with Lazy Split," in *FAST'14*, Santa Clara, CA, Feb. 2014.

[42] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/O Stack Optimization for Smartphones," in *Proc. USENIX ATC'13*, San Jose, CA, Jun. 2013.

[43] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory," in *Proc. FAST'13*, San Jose, CA, Feb. 2013.

[44] K. Lee, J. J. Kan, and S. H. Kang, "Unified Embedded Non-Volatile Memory for Emerging Mobile Markets," in *Proc. ISPLED'14*, La Jolla, CA, Aug. 2014.

[45] A. Badam, "Impact of Persistent Random Access Memory on Software Systems," *IEEE Computer Society*, May 2013.

[46] H. Luo, L. Tian, and H. Jiang, "qNVRAM: quasi Non-Volatile RAM for Low Overhead Persistency Enforcement in Smartphones," in *HotStorage'14*, Philadelphia, PA, Jun. 2014.

[47] D. Narayanan and O. Hodson, "Whole-system Persistence with Non-volatile Memories," in *Proc. ASPLOS'12*, London, UK, 2012.

[48] D. E. Lowell and P. M. Chen, "Free Transactions with RioVista," in *Proc. 16th ACM SOSP*, Saint-Malô, France, Oct. 1997.

[49] A. A. Sani, K. Boos, M. H. Yun, and L. Zhong, "Rio: A System Solution for Sharing I/O Between Mobile Systems," in *Proc. 12th ACM MobiSys*, Bretton Woods, NH, Jun. 2014.

[50] E. B. Nightingale and J. Flinn, "Energy-efficiency and Storage Flexibility in the Blue File System," in *Proc. OSDI'04*, San Francisco, CA, Dec. 2004.

[51] D. Peek and J. Flinn, "EnsemBlue: Integrating Distributed Storage and Consumer Electronics," in *Proc. OSDI'06*, Seattle, WA, Nov. 2006.

[52] N. Agrawal, A. Aranya, and C. Ungureanu, "Mobile data sync in a blink," in *Proc. HotStorage'13*, San Jose, CA, Jun. 2013.

[53] S. Sobti, N. Garg, F. Zheng, J. Lai, Y. Shao, C. Zhang, E. Ziskind, A. Krishnamurthy, and R. Y. Wang, "Segank: A distributed mobile storage system," in *Proc. FAST'04*, San Francisco, CA, Mar. 2004.

[54] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *Proc. SOSP'95*, Copper Mountain Resort, Colorado, Dec. 1995.

[55] S. Sobti, N. Garg, C. Zhang, X. Yu, A. Krishnamurthy, and R. Y. Wang, "Personalraid: Mobile storage for distributed and disconnected computers," in *Proc. FAST'02*, Monterey, CA, Jan. 2002.

[56] C. Preston, *Backup & recovery: Inexpensive backup solutions for open systems*. O'Reilly Media, Inc., 2007.

[57] M. E. Russinovich, D. A. Solomon, and J. Allchin, *Microsoft windows internals: Microsoft windows server 2003, windows xp, and windows 2000*. Microsoft Press Redmond, 2005, vol. 4.

[58] I. Comparing, "Tivoli storage manager and veritas netbackup in real-world environments," *A summary by IBM of the whitepaper and benchmark written by Progressive Strategies*, 2002.

[59] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 26–52, 1992.

[60] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems.," in *USENIX Annual Technical Conference, General Track*, 2005, pp. 105–120.

[61] M. Virable, S. Savage, and G. M. Voelker, "BlueSky: A Cloud-Backed File System for the Enterprise," in *Proc. 10th USENIX conference on File and Storage Technologies (FAST'12)*, San Jose, CA, Feb. 2012.

[62] A. Continella, A. Guagneli, G. Zingaro, G. D. Pasquale, A. Barenghi, S. Zanero, and F. Maggi, "ShieldFS: A Self-healing, Ransomware-aware Filesystem," in *Proc. the 32nd Annual Conference on Computer Security Applications (ACSAC'16)*, Los Angeles, CA, Dec. 2016.

[63] E. Kolodenker, W. Koch, G. Stringhini, and M. Egele, “PayBreak: Defense Against Cryptographic Ransomware,” in *Proc. the 2017 ACM on Asia Conference on Computer and Communications Security*, Abu Dhabi, United Arab Emirates, Apr. 2017.

[64] Fusion-io ioDrive, `http://www.fusionio.com/data-sheets/iomemory-px600-atomic-series/.`

[65] Y. Bu, H. Lee, and J. Madhavan, “Comparing SSD-placement Strategies to scale a Database-in-the-Cloud,” in *Proc. SoCC’13*, Santa Clara, CA, Oct. 2013.

[66] D. Shukla, S. Thota, K. Raman, M. Gajendran, A. Shah, S. Ziuzin, K. Sundama, M. G. Guajardo, A. Wawrzyniak, S. Boshra, R. Ferreira, M. Nassar, M. Koltachev, J. Huang, S. Sengupta, J. Levandoski, and D. Lomet, “Schema-agnostic indexing with azure documentdb,” in *Proc. VLDB’15*, Kohala Coast, Hawaii, Sep. 2015.

[67] N. Zhang, J. Tatemura, J. M. Patel, and H. Hacigumus, “Re-evaluating Designs for Multi-Tenant OLTP Workloads on SSD-based I/O Subsystems,” in *Proc. SIGMOD’14*, Snowbird, UT, Jun. 2014.

[68] J. He, D. Nguyen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Reducing File System Tail Latencies with Chopper,” in *Proc. FAST’15*, Santa Clara, CA, Feb. 2015.

[69] D. Skourtis, D. Achlioptas, N. Watkins, C. Maltzahn, and S. Brandt, “Flash on rails: consistent flash performance through redundancy,” in *Proc. USENIX ATC’14*, Philadelphia, PA, Jun. 2014.

[70] S. Yang, T. Harter, N. Agrawal, S. S. Kowsalya, A. Krishnamurthy, S. AI-Kiswany, R. T. Kaushik, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Split-Level I/O Scheduling,” in *Proc. SOSP’15*, Monterey, CA, Oct. 2015.

[71] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai, “The Linux implementation of a log-structured file system,” *SIGOPS OSR*, vol. 40, no. 3, 2006.

[72] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho, “F2FS: A New File System for Flash Storage,” in *Proc. FAST’15*, Santa Clara, CA, Feb. 2015.

[73] M. Rosenblum and J. K. Ousterhout, “The Design and Implementation of a Log-Structured File System,” *ACM Trans. on Computer Systems*, vol. 10, no. 1, pp. 26–52, Feb. 1992.

[74] J. Wang and Y. Hu, “WOLF: A Novel reordering write buffer to boost the performance of log-structured file systems,” in *Proc. FAST’02*, Monterey, CA, Jan. 2002.

[75] CNEX Labs,
`http://www.cnexlabs.com/index.php`.

[76] Amazon's SSD Backed EBS,
`https://aws.amazon.com/blogs/aws/new-ssd-backed-elastic-block-storage/`.

[77] Azure Premium Storage,
`https://azure.microsoft.com/en-us/documentation/articles/storage-premium-storage/`.

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

[79] Azure Service Fabric,
`https://azure.microsoft.com/en-us/services/service-fabric/`.

[80] Amazon Relational Database Service,
`https://aws.amazon.com/rds/`.

[81] Azure DocumentDB,
`https://azure.microsoft.com/en-us/services/documentdb/`.

[82] Azure SQL Database,
`https://azure.microsoft.com/en-us/services/sql-database/`.

[83] Google Cloud SQL,
`https://cloud.google.com/sql/`.

[84] Azure DocumentDB Pricing,
`https://azure.microsoft.com/en-us/pricing/details/documentdb/`.

[85] Azure SQL Database Pricing,
`https://azure.microsoft.com/en-us/pricing/details/sql-database/`.

[86] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds," in *Proc. EuroSys'12*, Paris, France, Apr. 2010.

[87] D. Shue and M. J. Freedman, "From Application Requests to Virtual IOPs: Provisioned Key-Value Storage with Libra," in *Proc. EuroSys'14*, Amsterdam, Netherlands, Apr. 2014.

[88] H. Wang and P. Varman, “Balancing Fairness and Efficiency in Tiered Storage Systems with Bottleneck-Aware Allocation,” in *Proc. FAST’14*, Santa Clara, CA, Feb. 2014.

[89] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, “The Multi-Streamed Solid-State Drive,” in *Proc. HotStorage’14*, Philadelphia, PA, Jun. 2014.

[90] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, “A Case for NUMA-aware Contention Management on Multicore Systems,” in *Proc. USENIX ATC’11*, Berkeley, CA, Jun. 2011.

[91] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, “Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning,” in *Proc. MICRO’11*, Porto Alegre, Brazil, Dec. 2011.

[92] CGROUPS,
`https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`.

[93] J. Leverich and C. Kozyrakis, “Reconciling High Server Utilization and Sub-millisecond Quality-of-Service,” in *Proc. EuroSys’14*, Amsterdam, Netherlands, Apr. 2014.

[94] Intel Inc., “Improving Real-Time Performance by Utilizing Cache Allocation Technology,” *White Paper*, 2015.

[95] D. Sanchez and C. Kozyrakis, “Vantage: Scalable and Efficient Fine-Grain Cache Partitioning,” in *Proc. ISCA’11*, San Jose, CA, Jun. 2011.

[96] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, C. Kim, and A. Greenberg, “EyeQ: Practical Network Performance Isolation at the Edge,” in *Proc. NSDI’13*, Berkeley, CA, Apr. 2013.

[97] Traffic Control HOWTO,
`http://linux-ip.net/articles/Traffic-Control-HOWTO/`.

[98] Block IO Bandwidth (Blkio) in Docker,
`https://docs.docker.com/engine/reference/run/#block-io-bandwidth-blkio-constraint`.

[99] Block IO Controller,
`https://www.kernel.org/doc/Documentation/cgroup-v1/blkio-controller.txt`.

[100] Amazon Relational Database Service Pricing,
`https://aws.amazon.com/rds/pricing/`.

[101] H. Menon and L. Kale, “A Distributed Dynamic Load Balancer for Iterative Applications,” in *Proc. SC’13*, Denver, Colorado, Nov. 2013.

[102] SQL Database Options and Performance: Understand What’s Available in Each Service Tier,
`https://azure.microsoft.com/en-us/documentation/articles/sql-database-service-tiers/#understanding-dtus`.

[103] K. Shen, S. Park, and M. Zhu, “Journaling of journal is (almost) free,” in *Proc. FAST’14*, Berkeley, CA, 2014.

[104] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman, “Don’t stack your Log on my Log,” in *Proc. INFLOW’14*, Broomfield, CO, Oct. 2014.

[105] FIO Benchmarks,
`https://linux.die.net/man/1/fio`.

[106] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with ycsb,” in *Proc. SoCC’12*, Indianapolis, IN, Jun. 2010.

[107] TPCC Benchmark, `http://www.tpc.org/tpcc/`.

[108] TATP Benchmark, `http://tatpbenchmark.sourceforge.net/`.

[109] TPCB Benchmark, `http://www.tpc.org/tpcb/`.

[110] TPCE Benchmark,
`http://www.tpc.org/tpce/`.

[111] LevelDB,
`https://github.com/google/leveldb`.

[112] Shore-MT, `https://sites.google.com/site/shoremt/`.

[113] Violin Memory 6000 Series Flash Memory Arrays,
`http://violin-memory.com/products/6000-flash-memory-array`.

[114] A. Badam, V. S. Pai, and D. W. Nellans, “Better Flash Access via Shapeshifting Virtual Memory Pages,” in *Proc. ACM TRIOS*, Nov. 2013.

[115] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, “Efficient Virtual Memory for Big Memory Servers,” in *Proc. ISCA’13*, Tel-Aviv, Israel, 2013.

[116] Redis, `http://redis.io`.

[117] GraphChi, `http://graphlab.org/graphchi/`.

[118] M. Grund, J. Krueger, H. Plattner, A. Zeier, and P. C.-M. S. Madden, "HYRISE–A Main Memory Hybrid Storage Engine," *PVLDB*, vol. 4, no. 2, pp. 105–116, 2010.

[119] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling, "High-Performance Concurrency Control Mechanisms for Main-Memory Databases," *PVLDB*, vol. 5, no. 4, 2012.

[120] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "PACMan: Coordinated Memory Caching for Parallel Jobs," in *Proc. NSDI'12*, 2012.

[121] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM," *SIGOPS OSR*, vol. 43, no. 4, 2010.

[122] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstractions for In-Memory Cluster Computing," in *Proc. NSDI*, 2012.

[123] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang, "SSD Bufferpool Extensions for Database Systems," *PVLDB*, vol. 3, no. 1-2, pp. 1435–1446, 2010.

[124] B. Debnath, S. Sengupta, and J. Li, "FlashStore: High Throughput Persistent Key-Value Store," *PVLDB*, vol. 3, no. 1-2, 2010.

[125] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson, "Turbocharging DBMS Buffer Pool Using SSDs," in *Proc 30th ACM SIGMOD*, Jun. 2011.

[126] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Proc. 10th USENIX OSDI*, Hollywood, CA, Oct. 2012.

[127] H. Lim, B. Fan, D. Andersen, and M. Kaminsky, "SILT: A Memory-Efficient, High-Performance Key-Value Store," in *Proc. 23rd ACM SOSP*, Cascais, Portugal, Oct. 2011.

[128] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "FAWN: A Fast Array of Wimpy Nodes," in *Proc. 22nd ACM SOSP*, Oct. 2009.

[129] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis, "CORFU: A Shared Log Design for Flash Clusters," in *Proc. 9th USENIX NSDI*, 2012.

[130] A. M. Caulfield, L. M. Grupp, and S. Swanson, "Gordon: Using Flash Memory to Build Fast, Power-Efficient Clusters for Data-Intensive Applications," in *Proc. ASPLOS'09*, Washington, DC, 2009.

[131] Boost Template Library, `http://www.boost.org/`.

[132] Fusion-io: ioMemory Virtual Storage Layer, `http://www.fusionio.com/overviews/vsl-technical-overview`.

[133] EXT4 File Index, `https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout#Extent_Tree`.

[134] A. M. Caulfield, T. I. Mollov, L. Eisner, A. De, J. Coburn, and S. Swanson, "Providing safe, user space access to fast, solid state disks," in *Proc. ASPLOS'12*, London, UK, 2012.

[135] Jonathan Thatcher and David Flynn, US Patent # 8,578,127, `http://www.faqs.org/patents/app/20110060887`.

[136] M. Gorman, *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004.

[137] Twitter: A Real Time Information Network, `https://twitter.com/about`.

[138] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a Social Network or a News Media?" In *Proc. WWW'10*, Raleigh, NC, Apr. 2010.

[139] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *Micro'10*, Atlanta, GA, 2010.

[140] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu, "Evaluating Phase Change Memory for Enterprise Storage Systems: A Study of Caching and Tiering Approaches," in *FAST'14*, 2014.

[141] J. Huang, K. Schwan, and M. K. Qureshi, "NVRAM-aware Logging in Transaction Systems," in *VLDB'15*, 2015.

[142] R. Johnson, I. Pandis, R. Stoica, and M. Athanassoulis, "Aether: A scalable approach to logging," in *VLDB'10*, 2010.

[143] F. Xu, Y. Liu, T. Moscibroda, R. Chandra, L. Jin, Y. Zhang, and Q. Li, "Optimizing Background Email Sync on Smartphones," in *Proc. 11th ACM MobiSys*, Taipei, Taiwan, Jun. 2013.

[144] Samsung Galaxy Gear Specs, `http://www.samsung.com/us/mobile/wearable-tech/SM-V7000ZKAXAR`.

[145] Google Glass Hardware, `https://support.google.com/glass/answer/3064128?hl=en`.

[146] Apple Watch, `http://www.apple.com/watch/apple-watch/`.

[147] Microsoft HoloLens, `http://www.microsoft.com/microsoft-hololens/en-us`.

[148] Battery Density Trends Research by Argonne National Laboratory, `http://ec.europa.eu/dgs/jrc/downloads/events/20130926-eco-industries/20130926-eco-industries-miller.pdf`.

[149] iPhone Technical Specifications, `http://www.gsmarena.com/apple_iphone_6_plus-6665.php`.

[150] Samsung Phone Technical Specifications, `http://www.samsung.com/us/mobile/cell-phones/all-products`.

[151] Lumia 930 Technical Specifications, `http://www.microsoft.com/en/mobile/phone/lumia930/specifications/`.

[152] Google Glass Software, `https://developers.google.com/glass/tools-downloads/system`.

[153] Android Wear API, `https://developer.android.com/design/wear/`.

[154] Monsoon Power Monitor, `http://www.msoon.com/LabEquipment/PowerMonitor/`.

[155] Java Native Interface,
http://developer.android.com/training/articles/perf-jni.html.

[156] D. Barua, J. Kay, and C. Paris, "Viewing and Controlling Personal Sensor Data: What Do Users Want?" *Persuasive*, pp. 15–26, 2013.

[157] N. Hochman and R. Schwartz, "Visualizing Instagram: Tracing Cultural Visual Rhythms," in *Proc. Sixth International AAAI Conference on Weblogs and Social Media*, Jun. 2012.

[158] k-Nearest Neighbors Algorithm,
http://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm.

[159] Iterative Dichotomiser 3 Algorithm,
http://en.wikipedia.org/wiki/ID3_algorithm.

[160] k-means Clustering Algorithm,
http://en.wikipedia.org/wiki/K-means_clustering.

[161] Apple HealthKit,
https://developer.apple.com/healthkit/.

[162] Zeo Mobile Sleep Manager,
http://www.engadget.com/products/zeo/sleep-manager/mobile/.

[163] UVeBand,
http://www.uveband.com/.

[164] Google Drive,
http://drive.google.com.

[165] WannaCry Ransomware Attack,
https://en.wikipedia.org/wiki/WannaCry_ransomware_attack, 2017.

[166] Ransomware Definition,
http://www.trendmicro.com/vinfo/us/security/definition/ransomware, 2016.

[167] Ransomware: the Tool of Choice for Cyber Extortion,
https://www.fireeye.com/current-threats/what-is-cyber-security/ransomware.html, 2016.

[168] G. O'Gorman and G. McDonald, *Ransomware: A growing menace*. Symantec Corporation, 2012.

[169] Special Report: Ransomware and Businesses 2016, `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/ISTR2016_Ransomware_and_Businesses.pdf`.

[170] IBM X-Force Research, "Ransomware: How consumers and businesses value their data," *Technical Report*, 2016.

[171] How Ransomware Became a Billion-Dollar Nightmare for Business, `https://www.theatlantic.com/business/archive/2016/09/ransomware-us/498602/`, 2016.

[172] A. Kharraz, S. Arshad, C. Mulliner, W. Robertson, and E. Kirda, "UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware," in *25th USENIX Security Symposium*, Austin, Texas, Aug. 2016.

[173] N. Scaife, H. Carter, P. Traynor, K. R. B. Butler, undefined, undefined, undefined, and undefined, "Cryptolock (and drop it): Stopping ransomware attacks on user data," *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, vol. 00, 2016.

[174] J. M. Dongpeng Xu and D. Wu, "Cryptographic Function Detection in Obfuscated Binaries via Bit-precise Symbolic Loop Mapping," in *Proc. 38th IEEE Symposium on Security and Privacy (Oakland'17)*, San Jose, CA, May 2017.

[175] J. H. Palevich and M. Taillefer, *Network file system*, US Patent 7,441,012, 2008.

[176] Will you backups protect you against ransomware, `http://www.csoonline.com/article/3075385/backup-recovery/will-your-backups-protect-you-against-ransomware.html`.

[177] D. Sgandurra, L. Muñoz-González, R. Mohsen, and E. C. Lupu, "Automated Dynamic Analysis of Ransomware: Benefits, Limitations and use for Detection," *ArXiv e-prints*, Sep. 2016. eprint: `1609.03020` (cs.CR).

[178] W. Bux and I. Iliadis, "Performance of greedy garbage collection in flash-based solid-state drives," *Performance Evaluation*, vol. 67, no. 11, pp. 1172–1186, 2010.

[179] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for ssd performance.," in *USENIX Annual Technical Conference*, 2008, pp. 57–70.

[180] A. Kharraz, W. Robertson, D. Balzarotti, L. Bilge, and E. Kirda, “Cutting the Gordian Knot: A Look Under the Hood of Ransomware Attacks,” in *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, Milan, IT, Jul. 2015.

[181] VirusTotal - Free Online Virus, Malware and URL Scanner, `https://www.virustotal.com/`.

[182] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, “A survey on automated dynamic malware-analysis techniques and tools,” *ACM Computing Surveys (CSUR)*, vol. 44, no. 2, p. 6, 2012.

[183] A. Sankaran, K. Guinn, and D. Nguyen, “Volume shadow copy service,” *Power Solutions, March*, 2004.

[184] S. Garfinkel, P. Farrell, V. Roussev, and G. Dinolt, “Bringing science to digital forensics with standardized forensic corpora,” *digital investigation*, vol. 6, S2–S11, 2009.

[185] Master File Table (Windows), `https://msdn.microsoft.com/en-us/library/windows/desktop/aa365230(v=vs.85).aspx`.

[186] Cerber Ransomware - New, But Mature, `https://blog.malwarebytes.com/threat-analysis/2016/03/cerber-ransomware-new-but-mature/`.

[187] WMIC - Take Command-line Control over WMI, `https://msdn.microsoft.com/en-us/library/bb742610.aspx`.

[188] N. Dayan, P. Bonnet, and S. Idreos, “GeckoFTL: Scalable Flash Translation Techniques For Very Large Flash Devices,” in *Proc. SIGMOD’16*, San Francisco, CA, Jun. 2016.

[189] R. Verschoren and B. V. Houdt, “On the Impact of Garbage Collection on Flash-Based SSD Endurance,” in *Proc. 4th Workshop on Interactions of NVM/Flash with Operating System and Workloads (INFLOW’16)*, Savannah, GA, Nov. 2016.

[190] H. Kim and S. Ahn, “BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage,” in *FAST’08*, San Jose, CA, Feb. 2008.

[191] Data Recovery, `https://support.microsoft.com/en-us/help/835840/data-recovery`, 2014.

[192] D. Narayanan, A. Donnelly, and A. Rowstron, “Write Off-Loading: Practical Power Management for Enterprise Storage,” in *Proc. FAST’08*, San Jose, CA, Feb. 2008.

[193] A. Verma, R. Koller, L. Useche, and R. Rangaswami, “SRCMap: Energy Proportional Storage Using Dynamic Consolidation,” in *Proc. 6th USENIX on File and Storage Technologies (FAST’10)*, San Jose, CA, Feb. 2010.

[194] IOzone Lab,
`http://www.iozone.org/`.

[195] NetApp,
`http://www.shub-internet.org/brad/FreeBSD/postmark.html`.

[196] Write Amplification Factor,
`https://en.wikipedia.org/wiki/Write_amplification`.

[197] J. Reardon, S. Capkun, and D. Basin, “Data Node Encrypted File System: Efficient Secure Deletion for Flash Memory,” in *Proc. USENIX Security’12*, Bellevue, WA, Aug. 2012.

[198] R. L. Rivest, “All-or-nothing encryption and the package transform,” in *International Workshop on Fast Software Encryption*, Springer, 1997, pp. 210–218.

[199] J. Lee, S. Yi, J. Heo, H. Park, S. Y. Shin, and Y. Cho, “An efficient secure deletion scheme for flash file systems.,” *Journal of Information Science and Engineering*, vol. 26, no. 1, pp. 27–38, 2010.

[200] J. Reardon, D. Basin, and S. Capkun, “Sok: Secure data deletion,” in *Proc. 2013 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2013, pp. 301–315.

[201] J. Huang, A. Badam, R. Chandra, and E. B. Nightingale, “WearDrive: Fast and Energy-Efficient Storage for Wearables,” in *Proc. 2015 USENIX Annual Technical Conference (USENIX ATC’15)*, Santa Clara, CA, Jul. 2015.

[202] Ransomware on Mobile Devices,
`http://www.hipaajournal.com/ransomware-mobile-devices/`.

[203] R. Lipovsky, L. Stefanko, and G. Branisa, “The Rise of Android Ransomware,” *White Paper*, 2016.

[204] K. Savage, P. Coogan, and H. Lau, “The Evolution of Ransomware,” *Symantec Technical Report*, 2015.

[205] N. Andronio, S. Zanero, and F. Maggi, “HelDroid: Dissecting and Detecting Mobile Ransomware,” in *Proc. International Symposium on Research in Attacks, Intrusion and Detection (RAID’15)*, Kyoto, Japan, Nov. 2015.

VITA

Jian Huang is a Ph.D. candidate in the School of Computer Science at Georgia Institute of Technology. His research interests lie in the areas of computer systems, including operating systems, systems architecture, distributed systems, systems reliability and security, and especially the intersections of them. He enjoys building systems. His recent research contributions have been published at ASPLOS, FAST, USENIX ATC, ISCA, VLDB, and SoCC. His work WearDrive won the Best Paper Award at USENIX ATC in 2015 and attracted popular press coverage in more than eight countries. His work FlashMap won the IEEE Micro Top Picks Honorable Mention in 2016. He received the Master of Science degree in Computer Science from the Huazhong University of Science and Technology, the Bachelor of Engineering degree in Computer Science and the Bachelor of Arts degree in Engineering Management from the Taiyuan University of Technology, China.