UNIVERSITY OF CALIFORNIA, SAN DIEGO

Modernizing Storage Device Interface for Performance and Reliability

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

in

Computer Science

by

Yanqin Jin

Committee in charge:

Professor Yannis Papakonstantinou, Co-Chair
Professor Steven Swanson, Co-Chair
Professor Pamela Cosman
Professor Alin Deutsch
Professor Geoffrey Voelker

2017

The dissertation of Yanqin Jin is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____
Co-Chair

_____
Co-Chair

University of California, San Diego

2017

DEDICATION

To my parents who have always supported me.

EPIGRAPH

Science is a way of thinking much more than it is a body of knowledge.

– Carl Sagan

TABLE OF CONTENTS

Signature Page . . . iii

Dedication . . . iv

Epigraph . . . v

Table of Contents . . . vi

List of Figures . . . ix

List of Tables . . . x

Acknowledgements . . . xi

Vita . . . xiii

Abstract of the Dissertation . . . xiv

Chapter 1 Introduction . . . 1

Chapter 2 Motivation and background . . . 9
2.1 Flash memory . . . 11
2.2 Modern SSD architecture . . . 12
2.2.1 Flash translation layer . . . 12
2.2.2 SSD hardware architecture . . . 14
2.2.3 Host interface and protocol . . . 14
2.3 SSDs and transactions . . . 14
2.3.1 Concepts of transactions . . . 15
2.3.2 Native support for transaction in SSDs . . . 17
2.4 Log-structured storage . . . 19

Chapter 3 KAML: a flexible, high-performance key-value SSD . . . 22
3.1 System overview . . . 24
3.1.1 Keys, values, and namespaces . . . 26
3.1.2 Atomicity and durability . . . 27
3.1.3 Fine-grained locking . . . 27
3.1.4 KAML caching layer . . . 28
3.2 KAML . . . 30
3.2.1 Hardware architecture . . . 31
3.2.2 Namespace management . . . 32
3.2.3 Mapping tables . . . 34
3.2.4 Transaction support . . . 35

3.2.5 Wear-leveling and GC . . . 36
3.3 Evaluation . . . 37
3.3.1 Experimental setup . . . 38
3.3.2 Microbenchmarks . . . 39
3.3.3 Effect of number of logs . . . 41
3.3.4 OLTP . . . 42
3.3.5 NoSQL key-value store . . . 47
3.4 Related work . . . 47
3.4.1 Innovations in SSD architecture . . . 48
3.4.2 Key-Value stores in SSDs . . . 50
3.5 Summary . . . 51

Chapter 4 Improving SSD lifetime with byte-addressable metadata . . . 53
4.1 System overview . . . 55
4.1.1 PebbleSSD interface . . . 55
4.1.2 PebbleSSD applications . . . 58
4.2 PebbleSSD . . . 59
4.2.1 Hardware architecture . . . 59
4.2.2 BAM . . . 61
4.2.3 Block layer and device driver . . . 63
4.3 File systems customization . . . 64
4.3.1 Log cleaning . . . 64
4.3.2 Data block writing . . . 65
4.4 Evaluation . . . 66
4.4.1 Experimental setup . . . 66
4.4.2 Microbenchmarks . . . 68
4.4.3 Efficient log cleaning . . . 70
4.4.4 Write-optimized file block index . . . 71
4.4.5 BAM space utilization . . . 73
4.5 Related work . . . 75
4.5.1 Backward pointer . . . 75
4.5.2 Address map manipulation . . . 75
4.5.3 Coordinated garbage collection . . . 77
4.5.4 Metadata caching . . . 78
4.6 Summary . . . 78

Chapter 5 Willow: a user-programmable SSD . . . 80
5.1 System design . . . 82
5.1.1 Willow system components . . . 83
5.1.2 The Willow usage model . . . 84
5.1.3 Building an SSD App . . . 86
5.1.4 The SPU architecture . . . 87
5.1.5 The RPC interface . . . 89

5.1.6 Protection and sharing in Willow 90
5.2 The Willow prototype 93
5.3 Case study: AtomicWrites 94
5.4 Related work 97
5.5 Summary 98

Chapter 6 Conclusion 100

Bibliography 102

LIST OF FIGURES

Figure 2.1: Architecture of an example SSD 15
Figure 2.2: Page-level `atomic-write` with record-level locks 18
Figure 2.3: Wandering tree problem 21

Figure 3.1: KAML system architecture 25
Figure 3.2: State transition graph of a transaction 30
Figure 3.3: KAML SSD architecture 31
Figure 3.4: In-storage log and records 33
Figure 3.5: Bandwidth comparison between KAML and block I/O 37
Figure 3.6: Latency comparison between KAML and block I/O 37
Figure 3.7: Effect of batch size on KAML bandwidth 41
Figure 3.8: Effect of multiple KAML logs 42
Figure 3.9: Throughput of OLTP workloads on KAML 43
Figure 3.10: YCSB throughput on KAML 48

Figure 4.1: Pebble system architecture 56
Figure 4.2: Log cleaning 57
Figure 4.3: PebbleSSD architecture 60
Figure 4.4: PebbleSSD write-optimized file block mapping 62
Figure 4.5: Performance comparison of `fs_write` and `write` 67
Figure 4.6: Performance of `MOVE` with conventional SSD and PebbleSSD 68
Figure 4.7: Improvement on log cleaning efficiency due to `remap` 69
Figure 4.8: Improvement on writing to files with `fs_write` 70
Figure 4.9: Performance comparison between `F2FS-opt`, `NILFS2-opt` and their baseline implementations 72
Figure 4.10: Improvement on file write amplification for F2FS and NILFS2 . . . 74

Figure 5.1: A conventional SSD vs. Willow 82
Figure 5.2: The anatomy of an SSD App 83
Figure 5.3: READ() implementation for `Base-IO` 88
Figure 5.4: TPC-B throughput 95

LIST OF TABLES

Table 2.1: Flash memory opreations 12

Table 3.1: KAML commands 26
Table 3.2: KAML caching layer API 29
Table 3.3: YCSB workloads summary 47

Table 4.1: PebbleSSD commands 58

Table 5.1: RPCs for `Atomic-Writes` 96

ACKNOWLEDGEMENTS

I'm grateful for the support from many wonderful people during the long adventure that has led me to this milestone in my life.

I would like to acknowledge my advisors Profesor Steven Swanson and Professor Yannis Papakonstantinou for their kind support as co-chairs of my committee. Their guidance has helped me overcome the obstacles and achieve the goals throughout my graduate school time. Their mentorship in both research and career planning has proved to be invaluable to me.

Besides, I want to express my gratitude to Professor Alin Deutsch, Professor Geoffrey Voelker and Professor Pamela Cosman for their valuable comments and feedback as my committee members.

I'm also thankful for the help of other members of the Non-volatile Systems Laboratory. Especially I want to acknowledge Hung-Wei Tseng for his suggestions and encouragement during the long discussions in his office. I would like to thank Meenakshi Sundaram Bhaskaran, Jian Yang and Mark Gahagan for their assistance in solving the technical issues in my research. I would like to thank my office mates, Jing Li and Yuliang Li for those useful conversations and insights.

I wish to whole-heartedly thank all my friends who make me not feel too lonely during my graduate school years. I will never forget the happy time we have had together.

I must also thank my parents to whom I owe everything. Ever since I was a young boy, they taught me to be a good person and helped me develop a life-long passion for knowledge. Their love and optimism have given me the strength to survive the gloomy and frustrating periods during my entire PhD study.

Chapter 1, Chapter 2 and Chapter 3 contain material from "KAML: A Flexible, High-Performance Key-Value SSD", by Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou and Steven Swanson, which appears in the 23rd IEEE Symposium on High

Performance Computer Architecture. The dissertation author was the primary investigator and first author of this paper. The material in this chapter is copyright ©2017 by the Institute of Electrical and Electronics Engineers (IEEE).

Chapter 1, Chapter 2 and Chapter 4 contain material from "Improving SSD Lifetime with Byte-Addressable Metadata", by Yanqin Jin, Yannis Papakonstantinou and Steven Swanson, which has been submitted for publication. The dissertation author is the primary investigator and first author of this paper.

Chapter 1, Chapter 2 and Chapter 5 contain material from "Willow: A User-Programmable SSD", by Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu and Steven Swanson, which appears in the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, (OSDI 2014). The dissertation author was the sixth investigator and author of the paper. The material in this chapter is copyright ©2014 by USENIX *The Advanced Computing Systems Association*.

VITA

2010	Bachelor of Engineering, Tsinghua University
2010-2017	Graduate Student Researcher, University of California, San Diego
2011	Internship, Oracle Corporation
2012	Internship, Twitter Inc.
2013	Master of Science, University of California, San Diego
2014	Candidate of Philosophy, University of California, San Diego
2017	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

Jin, Yanqin, Hung-Wei Tseng, Yannis Papakonstantinou, Steven Swanson. "KAML: a flexible, high-performance key-value ssd." In 23rd IEEE Symposium on High Performance Computer Architecture, 2017

Liu, Yang, Hung-Wei Tseng, Mark Gahagan, Jing Li, Yanqin Jin, and Steven Swanson. "Hippogriff: Efficiently moving data in heterogeneous computing systems." In Computer Design (ICCD), 2016 IEEE 34th International Conference on, pp. 376-379. IEEE, 2016

Seshadri, Sudharsan, Mark Gahagan, Meenakshi Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. "Willow: A User-Programmable SSD." In OSDI, pp. 67-80. 2014

Chan, Christine S., Yanqin Jin, Yen-Kuan Wu, Kenny Gross, Kalyan Vaidyanathan, and Tajana Rosing. "Fan-speed-aware scheduling of data intensive jobs." In Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design, pp. 409-414. ACM, 2012

Ayoub, Raid Zuhair, Umit Ogras, Eugene Gorbatov, Yanqin Jin, Timothy Kam, Paul Diefenbaugh, and Tajana Rosing. "Os-level power minimization under tight performance constraints in general purpose systems." In Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design, pp. 321-326. IEEE Press, 2011

ABSTRACT OF THE DISSERTATION

Modernizing Storage Device Interface for Performance and Reliability

by

Yanqin Jin

Doctor of Philosophy in Computer Science

University of California, San Diego, 2017

Professor Yannis Papakonstantinou, Co-Chair
Professor Steven Swanson, Co-Chair

Modern solid state drives (SSDs) unnecessarily confine applications to the conventional block I/O interface, which under-utilizes SSD's internal resources, leading to suboptimal performance and unsatisfactory lifetime.

This thesis first presents the *key-addressable, multi-log SSD* (*KAML*), an SSD with a key-value interface that uses a novel multi-log architecture and stores data as variable-sized records rather than fixed-sized sectors. Exposing a key-value interface allows applications to remove a layer of indirection between application-level keys and data stored in the SSD. KAML also provides native support for fine-grained atomicity

and isolation. We have implemented a prototype of KAML on an SSD development platform, and results show that KAML outperforms conventional systems by up to 4.0×.

Existing SSDs also provide flash-based out-of-band (OOB) data that can only be updated on a conventional write. Consequently, the metadata stored in their OOB region lack flexibility due to the idiosyncrasies of flash memory, incurring unnecessary flash write operations detrimental to device lifetime. This thesis also presents PebbleSSD, an SSD with *byte-addressable metadata*, or BAM, as a mechanism exploiting the non-volatile, byte-addressable random access memory (NVRAM) inside the SSD. With BAM, PebbleSSD can support a range of useful features to improve its lifetime by reducing redundant flash writes. We have implemented a prototype of PebbleSSD on an SSD development platform, and experimental results demonstrate that PebbleSSD can reduce the amount of data written by log-structured file systems during log cleaning by up to 99%, and reduce file-system-level write amplification by up to 33% for a number of workloads.

Finally, previous proposals for SSDs with new interfaces suffer from the limitation caused by one-at-a-time design approach. To overcome this limitation, the thesis presents Willow, a user-programmable SSD with programmability as a central feature. Willow allows programmers to augment and extend the semantics of an SSD with application-specific features without compromising file system protections. We demonstrate the effectiveness and flexibility of Willow by implementing support for atomicity as an example. We find that defining SSD semantics in software is easy and beneficial, and that Willow makes it feasible for database transaction processing workload to benefit from a customized SSD interface.

Chapter 1

Introduction

Today the era of big data has arrived, and the ability to efficiently store and process huge volumes of data is crucial. Financial firms and e-commerce websites accept and process endless streams of requests from customers. People post articles, tweets, photos, videos, etc. on social network websites and update their personal pages frequently. Companies hosting web services not only dump user activities to logs, but also perform analysis on the stored data to asist decision making, provide customized user experience, and diagnose causes for system inefficiencies. To keep up with the rapid growth of data, the world is in need of storage systems with fast speed and large capacity. Few people would argue for the end of this trend in the near future.

Recent high-performance and efficient storage systems usually target modern flash-memory-based storage devices, e.g. solid state drives (SSDs) that already have large-scale deployment. Flash memory is a non-volatile storage medium with drastically lower latency than previous magnetic storage medium, higher density than existing DRAM and better cost-efficiency than emerging non-volatile main memories. Flash-based SSDs offer low latency, huge bandwidth and high concurrency compared with conventional hard disks. These performance features make SSDs attractive for modern, data-intensive

applications. Therefore they will continue to exist and play a significant role in data storage systems.

Since flash-based SSDs have significantly different characteristics from hard disks, existing storage system software calls for a thorough rethinking of its architecture and design decisions. Some trade-offs were made decades ago with the assumption of slow storage device that performs extremely poorly in random I/O and much better, yet still far from perfection in large, sequential I/O. Moreover, previous storage software does not have the urgent need to optimize the lifetime of hard disks that are considered to be very stable and reliable. Flash memory used by SSDs suffer from limited lifetime and thus has to be treated differently with extra care by software.

The design and implementation of storage systems with flash-based devices need to address two challenges. First, they should allow applications to fully unleash the potential of modern SSDs made possible by multiple flash chips connected via parallel channels, general-purpose embedded processors and large RAM. Second, they should accommodate the physical characteristics of flash memory, e.g. limited lifetime so that flash memory wears out slowly and evenly.

The I/O interface determines how the host application can access the storage device, therefore the design of I/O interface is very important in storage systems. An ideal interface not only presents an elegant abstraction of the underlying storage device, but also exposes internal resources for the benefit of applications. I/O interface should match the characteristics of storage devices. Therefore it has profound impact on system-level perfomance, efficiency and reliability.

Existing block-based, disk-centric I/O interface prevents the applications from fully utilizing the resources inside modern SSDs. The flash memory that SSDs use to store data is a poor match for the block-based I/O interface that dominates present storage devices. In fact, the block I/O interface is a legacy inherited from the design of

conventional hard disks and present SSDs have chosen to conform to this interface for the purpose of backward compatibility. Modern SSD vendors usually provision their SSDs with more resources than required to process simple I/O requests e.g. `read` and `write`, and such resources are unfortunately hidden from applications.

Other than the conventional block I/O interface, the key-value interface is an attractive alternative, since the latter is naturally in alignment with human perception of the world. Ultimately, users of storage system software are more concerned about collections of variable-sized, application-level objects than about fixed-sized blocks and sectors.

Furthermore, flash-based SSDs already use their internal computation and memory resources to maintain a layer of indirection between logical and physical storage address space. With powerful (by the standard of embedded system) general purpose processors and increased memory density, the indirection can be generalized to a higher level. Instead of having a fixed semantic of mapping logical to physical address space, the indirection can become the bridge between arbitrary application-specific domain and physical storage. This allows for simpler implementation of storage software and makes it possible to provide additional services like snapshots or multi-part atomic writes.

However, current proposals for key-value interfaces for SSDs fall short in several aspects. First, they provide a single, shared map for all the key-value pairs in the SSD, forcing the SSD to use uniform management policies to manage the key-value mapping across all applications. Second, the proposals stop at the SSD and ignore the rest of the storage stack. In particular, they do not provide generic caching facilities to improve the performance of key-value accesses.

Write-ahead logging (WAL) is another existing software-based technique that requires careful scrutiny of block-based I/O in the age of modern SSDs. Widely-used algorithms, e.g. ARIES [68] use WAL to provide support for fine-grained locking and

atomicity that are critical to system performance and data integrity. ARIES optimizes for hard disks and fails to fully utilize the I/O bandwidth offered by SSDs. To make the matter worse, software systems using WAL incur extra block writes to persistent storage. Before updating original data, they have to write the changes to the log first. Excessive writes consume flash memory rapidly and have negative impact on the lifetime and reliability of flash-based SSDs.

To fully utilize the I/O bandwidth of SSDs and avoid the extra writes to flash memory, many researchers have proposed native support for atomicity and durability inside the SSD. In addition to the original block-based `read` and `write`, they suggest that SSDs expose an `atomic-write` interface to applications, allowing applications to write multiple pages atomically and durably using a single command. Most of these proposals for atomic writes take advantage of the log-structured design of flash translation layer (FTL) and suffer from limitations caused by the mapping granularity of the FTL. They support only page-level atomic updates for transactions. While the page-level mapping that many FTLs employ makes this a convenient choice for the implementors, it is a poor match for the requirements of many applications. If the application uses the page-level `atomic-write` to update data directly, it has to perform page-level locking to protect a page from being modified by multiple concurrent transactions. For applications that deal with smaller units of data (e.g. database records), page-level locking has been shown to deliver much lower performance than finer-grained approaches due to excessive lock contention.

Being confined to conventional block-based I/O interface, applications performing even simple tasks may potentially suffer from inefficiencies, leading to sub-optimal performance and reliability. For example, an application, e.g. file system, database,etc. has to read data from its original location into host main memory and then write it out to its (new) destination location. The data movement task has to incur actual data write to

flash memory even if the application does not update the data at all.

Since existing SSDs already implement a mapping layer between logical and physical address space, applications can benefit from a new interface named `remap` that maps flash memory from source logical address (LBA) to destination LBA by modifying the mapping table. In reality, this feature requires additional architectural support that is missing in present SSDs. Current SSDs use the out-of-band (OOB) region on each flash page to store (some) FTL metadata. Storing the metadata in flash limits its flexibility since the OOB region is subject to the same idiosyncrasies as the primary, flash-based "in-band" data. Consequently updating the metadata itself will lead to flash writes, rendering `remap` as useless for existing SSDs.

The advent of non-volatile byte-addressable memories promises a solution to this limitation caused by flash-based metadata. By using non-volatile, byte-addressable memory to store FTL medata, the FTL can support dynamic, fine-grained update of the mapping between logical and physical address space. This will enable `remap` to support fast and efficient data movement performed by many applications.

Furthermore, host applications, e.g. file systems and databases can use the in-SSD non-volatile, byte-addressable memory to store their own metadata such as the file block indices in order to to reduce the amount of metadata that has to be written to flash when the file system updates the data. This has been proven to be especially useful to log-structured file systems [84, 7, 53].

There have been a number of proposals adding novel interfaces to modern SSDs. However, each of these proposals target one or several features. Although these features are all useful, the current one-at-a-time approach suffers from several limitations. First, adding features is complex and requires access to SSD internals, so only the SSD manufacturers can add them. Second, the code must be trusted, since it can access or destroy any of the data in the SSD. Third, to be cost-effective for manufacturers to

develop, market, and maintain, the new features must be useful to many users and/or across many applications. Selecting widely applicable interfaces for complex use cases is very difficult.

This thesis includes the following chapters in an attempt to explore solutions to the issues raised here. In Chapter 2 we survey the technological opportunities and challenges that motivate the research efforts in this thesis, including flash memory, SSD, FTL, transaction support in SSDs and log-structured file systems.

In Chapter 3 we present the design, implementation and evaluation of key-addressable, multi-log SSD (KAML), an SSD with a key-value interface that employs a novel multi-log architecture and stores data as variable-sized records rather than fixed-sized sctors. Exposing a key-value interface allows applications to remove a layer of indirection between application-level keys (e.g. database record IDs or file inode numbers) and data stored in the SSD. KAML also provides native transaction support used to support fine-grained locking, achieving improved performance compared to previous designs that require page-level locking. Finally, KAML includes a host caching layer analogous to a conventional page cache that leverages host DRAM to improve performance and provides additional transactional features. We demonstrate that our KAML prototype can improve the throughput of online transaction processing (OLTP) workloads by $1.1\times - 4.0\times$, and NoSQL key-value store applications by $1.1\times - 3.0\times$.

In Chapter 4 we present the design, implementation, application and evaluation of PebbleSSD, an SSD with *byte-addressable metadata*, or BAM, as a mechanism exploiting the non-volatile, byte-addressable random access memory (NVRAM) inside the SSD. With BAM, PebbleSSD can support a range of useful features to improve its lifetime by reducing redundant flash writes. Specifically, PebbleSSD supports a write-optimized, BAM-based file block mapping to prevent excessive updates of file system index blocks. Furthermore, PebbleSSD allows log-structured file systems to perform fast and efficient

log cleaning with minimal flash writes. We demonstrate that our PebbleSSD prototype can reduce the amount of data written by log-structured file systems during log cleaning by up to 99%, and PebbleSSD's BAM-based file block mapping can reduce flash writes by up to 33% for a number of workloads.

In Chapter 5 we present the design, implementation, application and evaluation of Willow, a user-programmable SSD to overcome the limitation of the current "one-at-a-time" approach in SSD interface design. We explore the potential of making programmability a central feature of the SSD interface. Our prototype system, called Willow, allows programmers to augment and extend the semantics of an SSD with application-specific features without compromising file system protections. The SSD Apps running on Willow give applications low-latency, high-bandwidth access to the SSD's contents while reducing the load that I/O processing places on the host processors. The programming model for SSD Apps provides great flexibility, supports the concurrent execution of multiple SSD Apps in Willow, and suupports the execution of trusted code in Willow. We demonstrate the effectiveness and flexibility of Willow by implementing support for atomicity as an example and measuring its performance. We find that defining SSD semantics in software is easy and beneficial, and that Willow makes it feasible for database transaction processing workload to benefit from a customized SSD interface.

Acknowledgments

This chapter contains material from "KAML: A Flexible, High-Performance Key-Value SSD", by Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou and Steven Swanson, which appears in the 23rd IEEE Symposium on High Performance Computer Architecture. The dissertation author was the primary investigator and first author of this paper.

This chapter contains material from the paper, “Improving SSD Lifetime with Byte-Addressable Metadata”, by Yanqin Jin, Yannis Papakonstantinou and Steven Swanson, which has been submitted for publication. The dissertation author is the primary investigator and first author of this paper.

This chapter contains material from “Willow: A User-Programmable SSD”, by Sudharsan Seshadri, MarkGahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson, which appears in the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, (OSDI 2014). The dissertation author was the sixth investigator and author of this paper.

Chapter 2

Motivation and background

Advances in technology have enabled flash memory with decreased latency and increased density. Flash memory is electronic storage medium invented by Toshiba [13] in the 1980s. Due to its physical characteristics, flash memory has become a very attractive storage medium for modern storage devices. Most mobile devices e.g. smart phones, tablets, digital camerals, etc. use flash memory as storage due to its low power consumption, high density and insensitivity to vibration. In many data centers, flash-based SSDs are starting to replace conventional hard disks, signaling the arrival of all-flash cloud era.

Flash-based SSDs require additional care to utilize the underlying flash memory. Flash memory has its own complexities, e.g. lack of in-place page update and limited lifetime. SSD manufacturers rely on a layer of indirection to address these issues. With firmware providing functionality e.g. address translation, wear leveling, garbage collection, etc., flash-based SSDs manage to present a linear logical address space to the host applications and maintain backward compatibility with conventional hard disks.

Since flash memory is drastically different from magnetic medium, and SSDs have different architecture and features from spinning hard disks, they call for a scrutiny of

previous disk-centric software design and offer additional opportunities for optimization.

Database support for transaction has long been optimized for spinning hard disk. To achieve atomicity and durability, many algorithms e.g. ARIES [68] have been proposed. They usually take the classic write-ahead-logging (WAL) approach in which updates are logged first before applied to original data. Therefore, WAL consumes extra space, which can affect the SSD's lifetime. Furthermore, in systems that use ARIES, multiple transactions sequentially append their log entries to a log file, but only one of them can flush the log at a time upon commit. This restriction stems from the characteristics of hard disks with limited IO parallelism and poor random access due to the lengthy seek time. SSDs do not suffer from this limitation, and such disk-centric design decisions will under-utilize SSD's bandwidth and internal resources, and worse, lead to suboptimal performance.

SSDs use a layer of indirection for address mapping, and this indirection offers an opportunity to implement support for atomicity and durability with an alternative approach called copy-on-write (COW). COW is not new, but is considered to be inefficient due to its need for an extra layer of indirection. Since SSDs already have a layer of indirection anyway, thus COW is tempting for SSDs. However, this thesis shows that naive page-based COW used by SSDs can cause performance degradation due to coarse-grained isolation.

Log-structured file systems [84, 7] have been proposed to meet the characteristics of spinning hard disk, and the sequential write access pattern is friendly to flash memory [53]. However, porting a conventional log-structured file system to flash-based SSD is not trivial and can lead to suboptimal performance and efficiency. The background I/O activity, i.e. log cleaning consumes flash memory although no new data is written. Log-structured file systems suffer from the "wandering tree" problem [19] which also consumes flash memory.

The rest of this chapter presents a comprehensive introduction to the background of this thesis. Section 2.1 describes the physical characteristics of flash memory. Section 2.2 describes the architecture and design/implementation of flash translation layer. Section 2.3 presents the basics of transactional support inside flash-based SSDs. Section 2.4 briefly surveys the concepts and characteristics of log-structured storage.

2.1 Flash memory

A Flash memory cell is a CMOS transistor with a control gate (CG) and a floating gate (FG). The FG is insulated by an oxide layer around it. Flash memory can trap electrons on the FG to store data. The number of electrons on the FG determines the threshold voltage of the memory cell. Since the FG is electronically isolated by the oxide layer, high voltages are necessary to allow the electrons to pass the oxide layer and reach or leave the FG. Writing (programming) moves electrons to the FG, while erasing moves electrons away from the FG. To read the content of a cell, an intermediate voltage is applied and the conductivity of the cell is checked.

There are two types of flash memory. The first is single level cell (SLC), while the second one is called multiple level cell (MLC). An SLC cell can store only one bit of information while a MLC cell can store multiple bits of information. Intuitively, MLC technology can achieve higher density than SLC, but SLC has better performance than MLC. Flash device manufacturers make difference design decisions and trade-offs depending on their optimization goals.

Flash memory has several important characteristics that storage device designers need to accommodate. First, flash read, program, and erase operations work on different granularities and have very different latencies, as shown in Table 2.1. This table contains only typical values and actual latencies vary depending on underlying technology. Read

and program operate on 4–8 KB pages. Once written, the page becomes immutable unless the *block* that contains it is erased. Blocks typically contain between 64 and 512 pages, and pages within a block must be written sequentially. Reading a page generally takes less than 100 μs, while programming a page takes 100 μs to 2000 μs depending on the underlying flash devices [38]. Erase operations need several milliseconds to complete. Finally, each block can endure only a limited number of erase operations before becoming unreliable.

Table 2.1: **Flash memory operations** Flash memory supports three operations.

Operation	Unit of operation	Latency
Read	Page	under 100 μs
Program	Page	100-200 μs
Erase	Block	2000 μs

Flash pages include an OOB region in addition to the "in-band" 4 or 8 KB region. The OOB region has 64 to 256 bytes and the control program of flash devices stores per-page metadata in the OOB region. This can include ECC, erase count and other metadata.

2.2 Modern SSD architecture

Modern SSDs feature a heterogeneous architecture and require dedicated firmware to manage their internal resources.

2.2.1 Flash translation layer

Due to the asymmetric operations, using flash memory in SSDs involves non-trivial extra efforts. Flash-based SSDs have to use firmware to hide the complexities

of underlying flash. Such firmware is critical to the performance and reliability of flash-based SSDs.

The core component of the firmware is the flash translation layer (FTL). Since pages are immutable once written (until an erase), in-place page updates are not possible. Naive erase-before-program not only incurs intolerably long latency, but also negatively impacts the storage device's lifetime. The solution to this issue borrows its idea from address indirection. A famous quote of David Wheeler states that "all problems in computer science can be solved by another layer of indirection."

The FTL implements a layer of indirection between the logical address space and the physical address space. The former is visible and accessible to the host programs, while the latter is kept within the internals of SSDs. Host programs always use logical block address (LBA) to access the SSD, and the firmware uses the FTL to translate the LBAs to physical flash addresses.

The firmware uses a mapping table and some other metadata to maintain the mapping between logical and physical address spaces. A conventional SSD may keep the logical-to-physical address mapping as an array in the DRAM of the SSD. Some SSDs also use the per-flash-page OOB region to keep the physical-to-logical address mapping. Consequently, flash write operations are necessary to change the bi-directional mapping between two address spaces.

The FTL serves read and write differently. Upon the arrival of a read request, the firmware uses the LBA in the read request to perform a lookup in the logical-to-physical mapping table. If the LBA has been associated with a physical flash page, the firmware initiates flash read operation with the corresponding physical address. If the LBA is not mapped to any flash address at the moment, a predefined status code is returned to the host. For write requests, the FTL always writes the data to erased flash memory first, and then modifies the mapping table so that the LBA specified in the write request now

points to the flash page that contains the data just written. Changing the mapping table effectively invalidates the data contained by the flash page that was mapped to the LBA prior to the latest write request. The data is then referred to as "garbage" and the space it occupies needs to be reclaimed by the firmware.

2.2.2 SSD hardware architecture

Modern SSDs have heterogeneous hardware architecture, and Figure 2.1 shows an example. SSDs usually have multiple flash channels providing abundant internal I/O bandwidth. SSDs contain multiple embedded processors to execute firmware code and perform tasks including address translation, garbage collection and wear leveling. SSDs' internal data structures (e.g., the address mapping table, block metadata, and snapshots) reside in an on-board DRAM. Some enterprise SSDs [89] use battery or supercapacitor to protect the DRAM from abrupt power failures. SSDs can also emerging non-volatile main memories such as the 3D Xpoint [14] memory to replace DRAM.

2.2.3 Host interface and protocol

SSDs are attached to the host machine via the host interface. Currently there are several different interfaces, e.g. SATA [12], SAS [11], PCIe [9] etc.

The NVM Express (NVMe) [71] is a new protocol or logical interface [71] for host machine to access high-speed SSDs via PCIe. NVMe is block-oriented, but it allows for vendor-specific extensions that can process data with variable lengths.

2.3 SSDs and transactions

Many applications require some kind of transactional support [37]. The non-overwriting property of flash-based SSDs offer a possible opportunity to integrate some

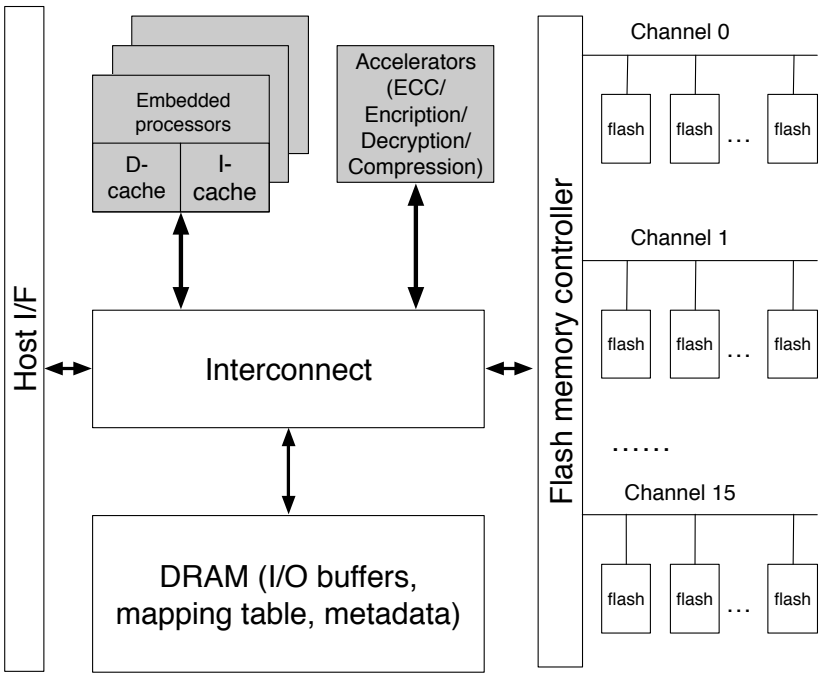


Figure 2.1: **Architecture of an example SSD** Modern SSDs have multiple channels to sustain huge internal bandwidth. Multiple embedded processors execute management firmware. System metadata reside in the internal DRAM that can be protected by battery or supercapacitor from power failures.

support for transactions in the FTL.

2.3.1 Concepts of transactions

In the terminology of computer science, a transaction is a group of operations on the data and/or metadata. Any transaction must have four properties, including *atomicity*, *consistency*, *isolation*, and *durability*.

- Atomicity, aka. “all or nothing”. If any part of the transaction fails, then the entire transaction fails. The failed transaction should not have any effect on the logical or physical view of the system, depending on the context.

- Consistency. A valid transaction must bring the system from one valid state to another valid state, in compliance with all the defined rules.

- Isolation. Even with multiple concurrent transactions, the system reaches a new state as if the transactions were executed sequentially.
- Durability. As long as a transaction successfully commits, its effect will be durable and able to survive system failures such as crash or power loss.

Databases use copy-on-write (COW) or write-ahead-logging (WAL) to support atomicity. If a system adopts COW, it never overwrites its data. Instead, the system writes the new version of the data to another location, and any future access to the data will be redirected to the new location. If a system uses WAL, all operations have to be recorded in logs residing in non-volatile storage. Original data cannot be updated until the log entry becomes persistent. Conventionally people prefer WAL since COW entails an extra layer of indirection causing performance and space overhead for the system. WAL has its disadvantages, too. First flushing log is often a bottleneck. Second, WAL consumes more space due to logs.

Databases implement concurrency control to offer support for isolation. Existing concurrency control methods broadly fall into two categories, optimistic concurrency control (OCC) or pesimistic concurrency control (PCC). In PCC, locking is a very popular technique. Data items, pages, records, etc. are protected by locks. To access a data item, a transaction must acquire a lock on the data item first. At a later point, the transaction can choose to release the lock.

Two-phase locking (2PL) is a locking protocol that enforces the order of lock acquisition and release. Each transaction must first acquire all the necessary locks before releasing any lock. Therefore, each transaction has a lock acquisition phase as well as a lock release phase, which gives its name.

Many databases use ARIES [68] to provide atomicity, isolation and durability. In ARIES, the database sequentially writes the log entries to a log file and applies the

logged changes to original data in the background. ARIES employs the "steal and no force" policy. When the host machine's main memory is sufficiently large, log flushing becomes the bottleneck [43]. Sequentially flushing log entries to a single log file matches the characteristics of conventional hard disks, but such a database system on a modern SSD with huge IO bandwidth may see very low bandwidth utilization.

2.3.2 Native support for transaction in SSDs

The existence of FTL as another layer of indirection inspires the adoption of COW in SSD's native support for transaction. Several research groups have proposed adding transactional support to SSDs [77, 76, 47, 91, 72]. KAML (and these proposals) provides native support for atomicity and durability, two key aspects of transactional ACID support, but leaves isolation and consistency to the application.

Leveraging conventional FTL designs to support atomicity is tempting, since the FTL already relies on atomic copy-on-write (COW). However most conventional FTLs are page-based, and many existing proposals [47, 91, 77] for atomic writes in the FTLs enable transactions to update data in the unit of pages. Another proposal [28] supports finer-grained locking, but relies on a more exotic byte-addressable non-volatile memory rather than flash.

Page-based atomic write interfaces are problematic when the application allows concurrent accesses to records on the same page [36]. If one transaction commits a page successfully, while another transaction with updates on the same page aborts, there is no copy of the page reflecting the correct state. Consequently transactions that use page-based atomic write have to acquire coarse-grained page locks to ensure correctness, which is detrimental to system performance. In contrast, most database storage engines e.g. InnoDB [42], Oracle Database [74], Shore-MT [93], etc. allow transactions to lock individual records. Therefore, they cannot benefit from page-based atomic write

interfaces.

Figure 2.2 illustrates what can go wrong if the system supports fine-grained record-level lock while allowing application to use page-level `atomic-write` to update data directly. T1 and T2 intend to update R1 and R2 respectively. Since the system enforces locking at record level, T1 and T2 do not block each other because they access different records on the same page. However, since the `atomic-write`perates on the page granularity, when T1 aborts and rolls the page back to its original state, the update on R2 performed by T2 will be lost, even if T2 has already committed. Consequently, to ensure correctness, the system has to enforce page-level locks, leading to sub-optimal performance.

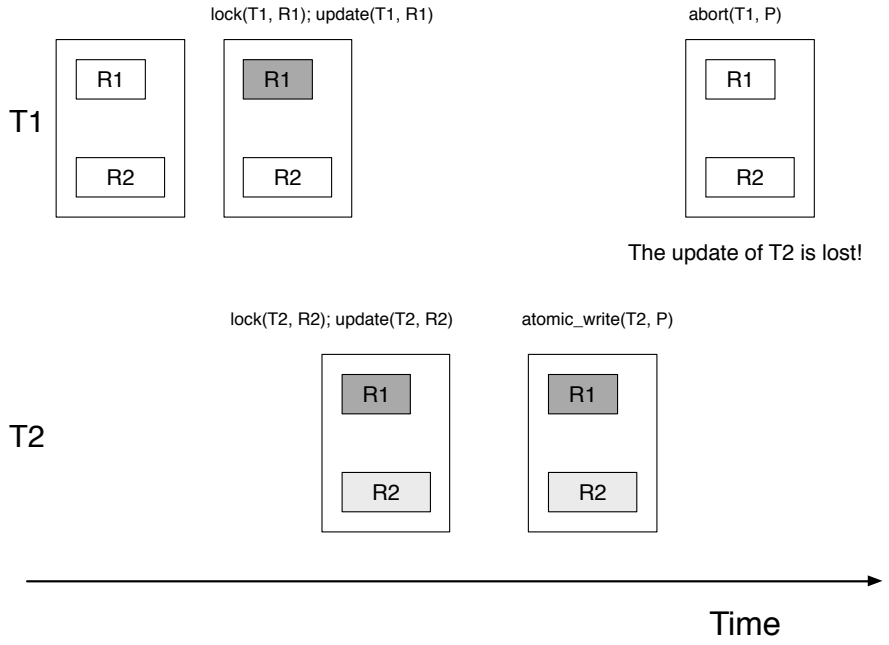


Figure 2.2: **Page-level `atomic-write` with record-level locks** When the system enforces record-level lock and allows transactions to use `atomic-write` to update data directly, concurrent transactions can cause one another to lose update.

KAML's variable-sized values allow for fine-grained, value-based locking, maxi-

mizing concurrency and making it a better match to real databases. Databases can rely on KAML's built-in support for atomicity rather than conventional write-ahead logging (WAL). ARIES [68], a classic logging algorithm (that adopts fine-grained record-based locking) poses a major bottleneck [43] in conventional databases due to contention on a global log, single-threaded log flushing and file system overhead. To make the matter worse, WAL schemes also induce redundant log write operations, which consume free space in flash memory rapidly.

2.4 Log-structured storage

The log-structured storage has been a classic idea to transform random writes to sequential writes. Conventional log-structured file systems [84, 7] aim to optimize for hard disks with much better sequential write performance than random access. A recent log-structured file system [53] targets flash memory devices from the beginning of its design. Sequential writes to multiple logs are flash-friendly and match the internal parallelism of SSDs. Log-structured merge tree [73] (LSM) organizes key-value pairs in logs. The FTL of a flash-based SSD also resembles a log-structured storage since it never overwrites old data but redirects updates to unused flash pages.

Log-structured file systems have to perform host log cleaning to create free *segments* in the logical address space for incoming write requests from user space applications. Log cleaning is similar to, but independent from the SSD's internal garbage collection[1]. Current SSDs confine the log-structured file system to block-based I/O interface. The semantic gap and lack of coordination between SSD garbage collection and file system log cleaning can lead to inefficiencies. Due to their independent data

[1] In some context, log cleaning is also called "garbage collection" of log-structured file systems. In this paper, for the purpose of simplicity and clarification, we use "log cleaning" in the context of log-structured file system, and "garbage collection" in the context of SSDs.

movement, valid data may be written multiple times by each of them, consuming erased flash pages rapidly.

Log-structured file systems suffer from the "wandering tree problem" [19]. This occurs because log-structured file systems must perform out-of-place updates. A write to one data block in a file can cause a cascade of writes as the file systems update the pointer to that data block, the pointer to the direct node block that contains the pointer, the pointer to the indirect block that points to the direct node block, and so on. In Figure 2.3, if the user program requests that the file system write block 6, the file system writes a new copy block $6'$ to a new location. Consequently block 2 that points to block 6 has to be written as well. This time, a new copy block $2'$ is written to a new location. This occurs repeatedly from the data block (leaf) all the way up to the inode block (root). In this process, simply writing a single data block causes the writes of multiple blocks, consuming flash memory more rapidly. This can negatively impact the lifetime and reliability of flash-based SSDs.

Acknowledgments

This chapter contains material from "KAML: A Flexible, High-Performance Key-Value SSD", by Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou and Steven Swanson, which appears in the 23rd IEEE Symposium on High Performance Computer Architecture. The dissertation author was the primary investigator and first author of this paper.

This chapter contains material from the paper, "Improving SSD Lifetime with Byte-Addressable Metadata", by Yanqin Jin, Yannis Papakonstantinou and Steven Swanson, which has been submitted for publication. The dissertation author is the primary investigator and first author of this paper.

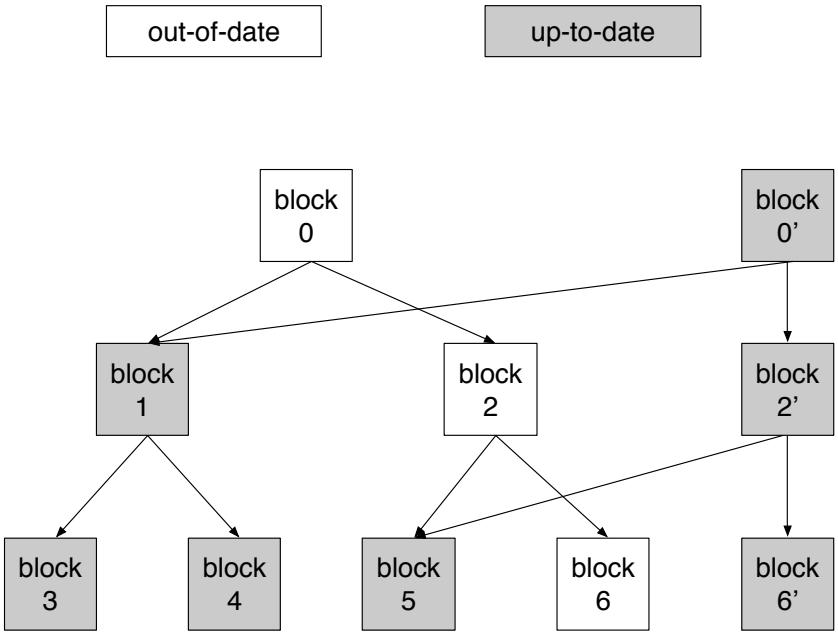


Figure 2.3: **Wandering tree problem** Log-structured file systems suffer from wandering tree problem caused by recursive writes of blocks.

This chapter contains material from "Willow: A User-Programmable SSD", by Sudharsan Seshadri, MarkGahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson, which appears in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, (OSDI 2014). The dissertation author was the sixth investigator and author of this paper.

Chapter 3

KAML: a flexible, high-performance key-value SSD

Flash memory has risen to prominence and is replacing spinning disks as the storage medium of choice for many enterprise storage systems. Solid state drives (SSDs) offer decreased latency, increased bandwidth, and increased concurrency, all of which make them attractive for modern, data-intensive applications. However, the flash memory that SSDs use to store data is a poor match for the block-based, disk-centric interface that dominates storage systems. The key-value interface is an attractive alternative, since SSDs must already maintain a layer of indirection between logical storage addresses (or keys) and physical storage locations. This allows for simpler implementations and makes it possible to exploit the layer of indirection to provide additional services like snapshots or multi-part atomic operations.

However, current proposals for key-value interfaces for SSDs fall short in several respects. First, they provide a single, shared map for all the key-value pairs in the SSD, forcing the SSD to use uniform policies to manage the key-value mapping across all applications. Second, the proposals stop at the SSD and ignore the rest of the storage stack.

In particular, they do not provide generic caching facilities to improve the performance of key-value accesses.

Most proposals for multi-part atomic writes in flash-based SSDs also suffer from limitations. Most notably, they support only page-level atomic updates for transactions. While the page-level mapping that most SSDs employ makes this a convenient choice for the implementors, it is a poor match for what many applications require. Providing page-level atomic writes requires the application to perform page-level locking. For applications that deal with smaller units of data (e.g., database records), page-level locking has been shown to provide lower performance than finer-grained approaches.

To address these limitations, this paper presents the *key-addressable, multi-log SSD* (*KAML*). KAML extends existing proposals for key-value-based SSDs and provides a fine-grained multi-part atomic write interface and a host-side caching layer to accelerate accesses to data it contains. The caching layer builds on the SSD's transaction interface and implements a fine-grained locking protocol that minimizes transaction aborts and maximizes concurrency.

KAML allows applications to create multiple key-value *namespaces* that can, depending on application requirements, represent files, database tables, or arbitrary collections of objects.

This new interface leads to three benefits. First, KAML shoulders the responsibility of mapping keys directly to values' *physical* addresses, allowing applications to bypass unnecessary indirection. Without KAML, applications have to maintain their own indices to map keys to file offsets, and rely on the file system to translate file offsets to logical block addresses (LBAs).

Second, applications can create or update multiple key-value pairs atomically using KAML interface. Without this interface, applications must use write-ahead logging (WAL) or other application-level techniques to provide atomic updates, consuming more

space and incurring expensive file system operations such as `fsync`. Finally, the fine-grained key-value interface of KAML improves system concurrency in comparison with its coarse-grained counterparts, because it allows for fine-grained locking.

KAML's approach to managing its internal flash memory reflects the requirements of its interface. Modern SSDs have multiple, parallel, semi-independent flash channels and can perform flash operations in parallel across those channels. KAML maps those channels to the logs it uses to record updates to particular namespaces, and allows the application to tune the mapping between namespaces and flash channels to optimize performance and improve quality of service.

We implement the SSD functions of KAML by extending the firmware inside a commercially available flash-based SSD reference design. We have also implemented KAML caching layer that runs on the host machine. The result shows that the KAML caching layer can serve as both a database storage manager and a stand-alone key-value store: on online transaction processing (OLTP) workloads, it outperforms Shore-MT [93], an open-source storage engine, by $1.1\times - 4.0\times$. For NoSQL key-value store workloads KAML is $1.1\times - 3.0\times$ faster than Shore-MT.

The rest of the chapter is organized as follows. Section 3.1 presents an overview of the system, and Section 3.2 details the implementation. Section 3.3 reports experimental results. Section 3.4 places this work in the context of existing literature, while Section 3.5 summarizes this chapter.

3.1 System overview

The heart of KAML is an SSD that uses a novel FTL structure to manage flash memory. This FTL provides a transactional, key-value interface. The system comprises a customizable SSD and three different pieces of software. First, the new FTL runs on an

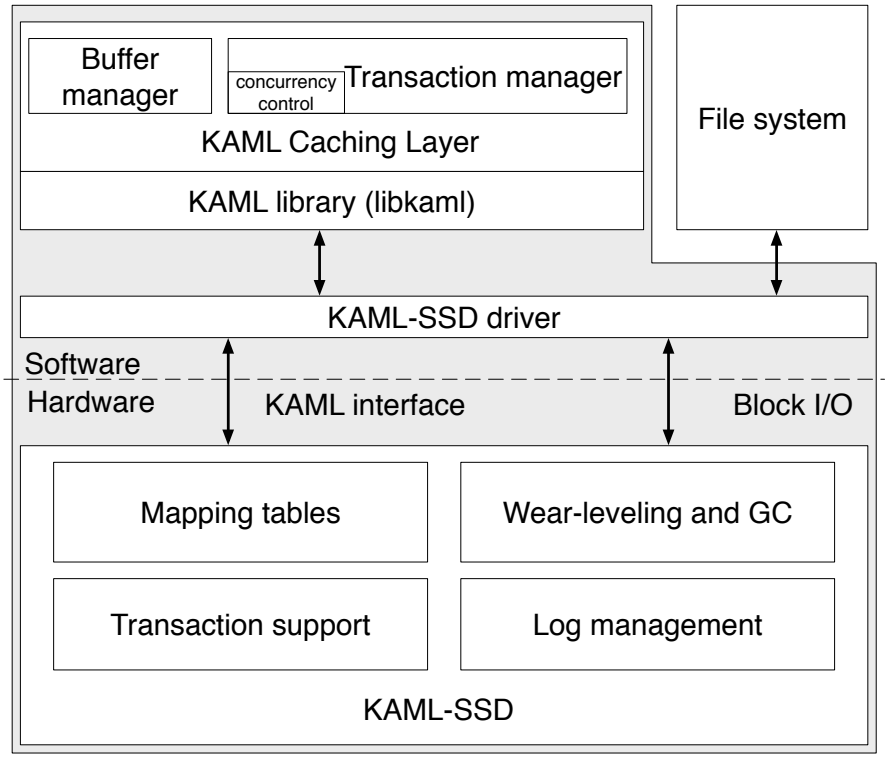


Figure 3.1: **KAML system architecture** The shaded part represents the KAML system with four components: KAML-SSD, a driver, a user library (libkaml) and KAML caching layer.

industrial SSD reference platform. Its interface allows host software (e.g., a database, file system, or other user space application) to create, manage, and access multiple key-value stores using fine-grained transactions. The firmware works in tandem with a kernel driver and a userspace library that run on the host machine. Finally, a host caching layer accelerates access using host DRAM and provides additional transactional facilities.

Figure 3.1 illustrates the relationship between these components. This section describes KAML's system components while next section describes its implementation in more detail.

KAML presents a key-value interface that supports fine-grained transactions. The interface allows the application to create and destroy key-value *namespaces* that represent logically related key-value pairs. Namespaces allow multiple, independent applications

Table 3.1: **KAML commands** These commands allow applications to access data on the SSD with transactional semantics and key-value addressing scheme.

Command	Description
`CreateNamespace`(attributes)	Create a namespace with given attributes, and return a namespaceID.
`DeleteNamespace`(namespaceId)	Delete a namespace with given namespaceID
`Get`(namespaceID, key)	Retrieve a value given its key and returns the value.
`Put`(namespaceIDs[], keys[], values[], lengths[])	Atomically update or insert a list of key-value pairs.

to share a KAML SSD. Table 3.1 summarizes the commands included in this interface.

3.1.1 Keys, values, and namespaces

Keys in KAML are 64-bits, but values can vary in size. Natively supporting variable-sized values lets application store a wide variety of objects in KAML. For instance, a conventional page-based file system could treat keys as block addresses and store 4 KB pages as values in a namespace. Alternately, a database could store individual tuples as values and use the tuple's key or record ID to map it into the SSD.

Exposing a key-value interface allows applications to eliminate redundant layers of indirection. In a conventional system, an application might map keys to locations in a file, the underlying file system would map that file location to a logical block address (LBA), and the FTL would map the LBA to a physical page number (PPN). KAML translates this into a single mapping from key to PPN and eliminates the application and file system overhead associated with the other mappings.

KAML also obviates the need to use log-structured techniques in the file systems, eliminating the performance problems that "log stacking" [103] can cause. The inter-

face allows KAML to consolidate garbage collection operations in the SSD, where the firmware has the most complete information about the flash's performance characteristics and the data layout.

3.1.2 Atomicity and durability

KAML's `Put` operation can atomically insert/update multiple key-value pairs, providing atomicity and durability (the "A" and "D" in ACID). The KAML caching layer can provide isolation. We leave consistency up to the application, because while atomicity and durability requirements are similar across applications, consistency requirements (e.g., database cascades, triggers, and constraints) vary widely. Likewise, if an application needs a custom locking protocol it can provide its own rather than rely on the caching layer's facility.

3.1.3 Fine-grained locking

Although the KAML SSD does not implement concurrency control, allowing for efficient concurrency control is a central goal of the KAML SSD interface, and the caching layers makes extensive use of it. Since KAML transactions operate on variable-sized key-value pairs [1] rather than fix-sized pages, applications can use fine-grained, record-based locking protocols rather than coarse-grained, page-based protocols.

Fine-grained locking is critical to performance. Without it, a transaction that modifies a record in a page must hold a lock for the entire page until the transaction commits, blocking any other transactions that need access to another record on the same page. We quantify the performance impact of coarse-grained locking in Section 3.3.

[1] In this paper, we use record and key-value pair interchangeably

3.1.4 KAML caching layer

KAML can improve application performance by caching data in DRAM, just as conventional systems cache page-based SSD or hard drive data. Systems with conventional SSDs cache data either in the operating system's page cache or application-specific caching layers (e.g., database buffer pool). KAML requires a different caching architecture because its interface is based on key-value pairs rather than pages or blocks.

KAML's caching layer provides both host DRAM-based caching and a richer transactional interface that provides isolation in addition to durability and atomicity that the KAML SSD provides.

Caching

The caching layer allows KAML to provide fast access to cached data and interact with the SSD via `Get` and `Put` commands. The caching layer differs from the conventional page cache in that it caches variable-length key-value pairs instead of fixed-sized pages (or blocks in file system terminology).

The caching layer uses the namespace ID and the key to form a compound key and probes a hash table. If the hash table has an entry that matches the key, the key-value pair is already in the cache, and the caching layer returns the key-value pair to the application.

If the hash table does not have a matching entry, the caching layer issues a `Get` request to the SSD, and inserts the resulting key-value pair into the hash table.

When a transaction commits (or when the caching layer runs out of space), the caching layer writes key-value pairs back to the SSD with a `Put`. Once the command completes, KAML can re-use the buffer space that the value occupied or keep the key-value pair in the cache and mark it as clean.

Table 3.2: **KAML caching layer API** The caching layer provides a transactional interface.

libkaml API	Description
`TransactionBegin`()	Start a new transaction and allocate resources.
`TransactionRead`()	Read a key-value pair in buffer pool or issue `Get` to bring it from SSD.
`TransactionUpdate`()	Update a key-value pair. The change stays in main memory until transaction commits.
`TransactionInsert`()	Insert a key-value pair. The record stays in main memory until transaction commits.
`TransactionCommit`()	Commit current transaction, persisting updates and releasing locks.
`TransactionAbort`()	Abort current transaction, abandoning updates and releasing locks.
`TransactionFree`()	Release the resources that the current transaction is using.

Transactions

The caching layer's transaction manager provides support for isolation using strong strict two-phase locking (SS2PL) [17] implemented on the host, and accesses key-value pairs stored on KAML via the caching layer.

The transaction manager maintains an array of transaction control blocks (XCB). When a transaction starts, the transaction manager allocates a XCB for the transaction. The XCB allows the transaction to be in one of the four states: IDLE (i.e., the transaction has not yet started), ACTIVE (i.e., the transaction is in progress), ABORTED, and COMMITTED. Figure 3.2 shows the possible transitions between these states, and Table 3.2 summarizes the actions that cause state transitions as well as transaction manager's API.

Active transactions acquire locks on key-value pairs before accessing them, and create private copies of the key-value pairs for use during the transaction. When the transaction commits, the transaction manager replaces the key-value pairs in the buffer

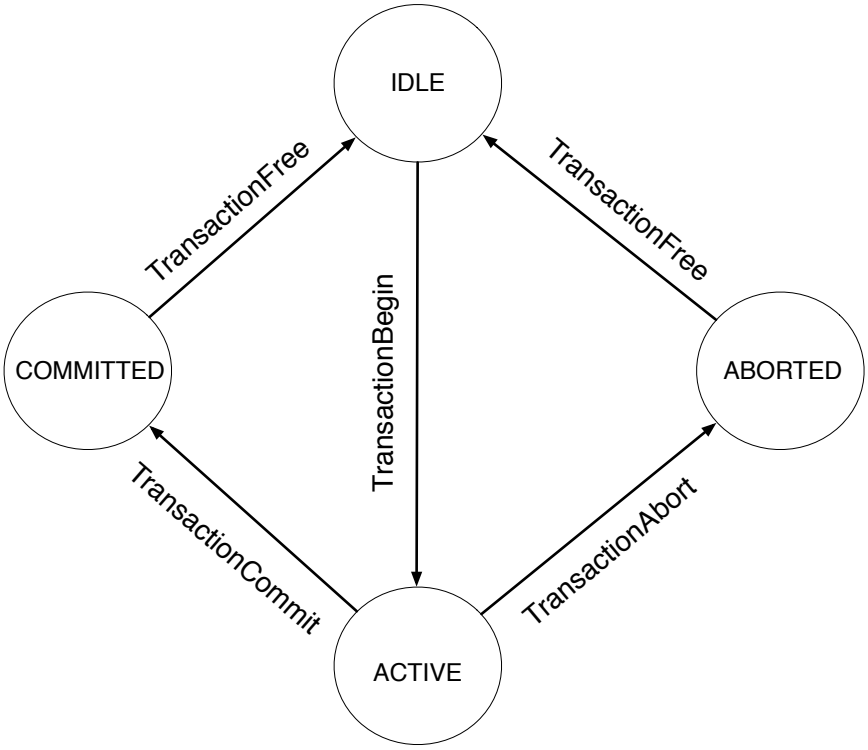


Figure 3.2: **State transition graph of a transaction** A transaction is in one of these states at any time, and transitions to another state by invoking KAML APIs.

pool with its private copies and then issues `Put` commands to flush the changes to KAML. On abort, the transaction simply discards its private copies and releases its locks.

3.2 KAML

KAML uses a customized SSD, custom driver, and lightweight userspace library to implement the KAML interface more efficiently than prior SSDs that provide a key-value interface. The firmware manages in-storage logs, maintains the mapping tables, performs wear leveling, and implements garbage collection (GC). Below, we describe the SSD prototyping hardware we use and key aspects of KAML's firmware and system software.

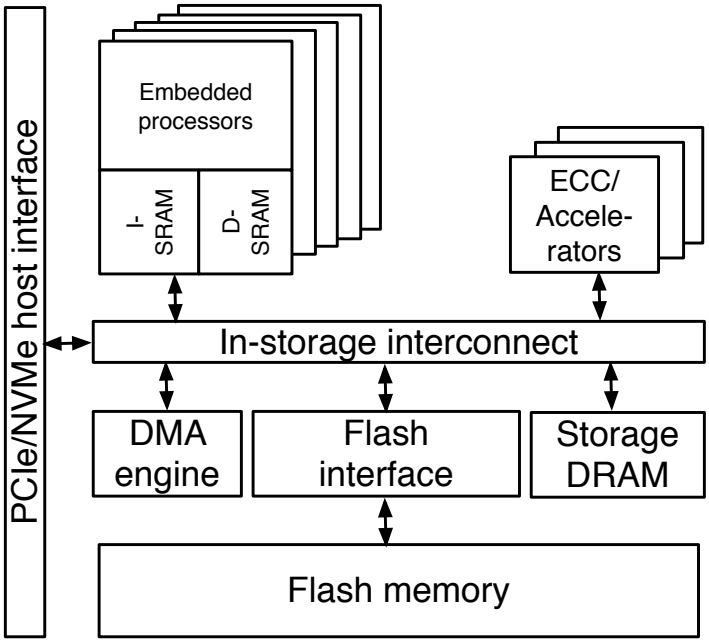


Figure 3.3: **KAML SSD architecture** KAML exposes its internal computation power to the host via PCIe/NVMe interface.

3.2.1 Hardware architecture

We have implemented KAML on a commercial NVMe [71] SSD development board. Like other modern SSDs, it comprises an array of flash chips (totalling 375 GB) arranged in 16 "channels" connected to a multi-core controller that communicates with a host system over PCIe. The firmware running on the controller defines the SSD's interface and implements management policies that aim to provide high performance and maximize flash life.

Figure 3.3 depicts the internal organization of the flash controller. The key features of the controller architecture are the cores themselves, the flash interface, and the on-board DRAM.

The cores have 64 KB private instruction and data memories. They can commu-

nicate with each other, the DRAM, and the flash interface using an on-chip network. To access flash, the cores issue a command to flash controller which reads or writes data from/to a buffer in DRAM. To access the contents of DRAM, the cores explicitly copy between the DRAM and their private data memories.

The flash channels each contain 4 flash chips that share control and data lines and attach to common flash memory controller on the SSD's controller. The flash chips can perform read, program, and erase operations in parallel, but only one chip can transfer data to or from the controller at a time.

For this work, we assume that the DRAM is persistent. In practice, it would either be battery or capacitor-backed or it could be replaced with a non-volatile technology like Intel's 3D-XPoint memory [14].

3.2.2 Namespace management

KAML manages the flash memory in each flash target as a log. When a `Put` operation arrives, KAML writes the key-value pair to the tail of the log, and updates an index structure that stores the location of the key-value pair corresponding to each key.

The architecture of the SSD determines the number of logs that the SSD maintains, KAML assigns each key-value namespace to multiple logs. This improves locality and leads to more efficient garbage collection (see below). However, the correspondence between namespaces and logs is not fixed: as workloads change the SSD can assign more or fewer logs to a single namespace to match workload the namespace is experiencing. If a namespace is particularly cold it might share a log with other namespaces. By default, all of the SSD's logs are available to all the namespaces. Multiple logs do not incur extra hardware overhead since modern SSDs already feature multi-channel architecture with internal parallelism.

Restricting the mapping between namespaces and logs serves as a locality opti-

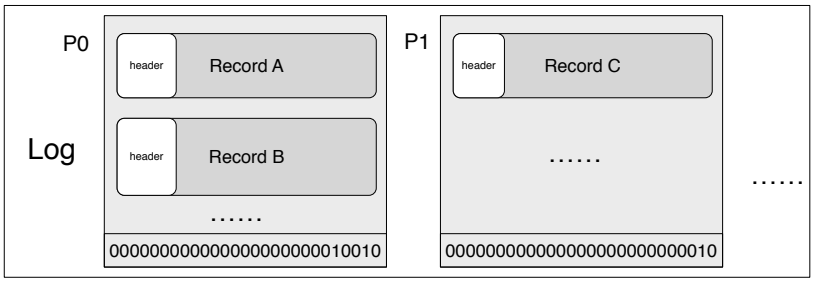


Figure 3.4: **In-storage log and records** Logs consist of flash pages, each of which stores a bitmap in the OOB region describing variable-sized records in the page. Record A occupies chunk 0 and 1 of page P0, record B occupies chunk 2, 3, and 4 of P0, record C occupies chunk 0 and 1 of P1.

mization allows for the SSD to control the allocation of resources – especially bandwidth – to namespaces. However, since the mapping is flexible, KAML's GC and flash management algorithms do not assume that data for a particular namespace resides in its assigned logs.

To bridge the gap between application requirements and flash memory characteristics, KAML stores variable-sized key-values in fixed-sized flash pages. Physical flash pages in our SSD are 8 KB + 256 bytes in size. Each page is divided into 64 fixed-sized chunks and each record can occupy variable number of chunks. The firmware uses 8-bytes of OOB data to store a bitmap that records the start and end chunks of each record on the page, as shown in Figure 3.4.

If a record's last chunk is the *i*-th chunk, the *i*-th bit of the bitmap is set. The first record always starts from chunk 0 of the page, and there are no unused chunks between two records on the same page. Other components of the firmware such as GC can use this bitmap to parse the content of the page (see Section 3.2.5).

Flash supports only whole-page program operations, but it must commit individual `Put` operations even if they do not fill a page. To overcome this problem, KAML uses a non-volatile, DRAM-like buffer to buffer multiple `Put` operations until a page's-worth

of data is ready to write or an internal timer times out. Existing SSDs use capacitor- or battery-backed memory for this purpose, but emerging non-volatile, byte-addressable memories like 3D XPoint would also be suitable.

3.2.3 Mapping tables

KAML uses per-namespace indices to locate the key-value pairs associated with each key. KAML indices are different from the maps that the FTLs in conventional SSDs in two ways. First, conventional FTLs map LBAs to PPNs, and the LBAs form a continuous block address space. This means that the meaning of an LBA is fixed (i.e., it corresponds to a particular logical address in the SSD's logical storage space), so it cannot contain useful application-level data (e.g., they cannot represent record IDs in a database table). It also means that LBAs correspond to fixed-sized chunks of data.

Second, in conventional FTLs, there is only a single mapping table, rather than many. This further limits the flexibility in how applications can use LBAs, since LBAs reside in a global, shared namespace. It also prevents the SSD from managing indices differently depending on access patterns or application needs.

For instance, KAML could limit the size of the mapping table for a namespace or even use different data structures (e.g., a tree instead of the hash tables [41] that KAML uses) to store the mapping tables.

We store the indices in the SSD's DRAM when the namespace is in use. Like other modern, high-end SSDs ours has several GBs of DRAM that we use to store the KAML indices. Since the KAML indices can be finer-grained they may be larger than the conventional LBA-to-PPN map. For example, a hash table for 100 million key-value pairs and a load factor of 75% requires approximately 2 GB in-storage DRAM.

To support larger data sizes, either the amount of DRAM the SSD provides may need to increase or the firmware may need to "swap" index information between flash and

DRAM. KAML currently loads the index for a namespace when an application accesses it, but it does not swap parts of the index. KAML employs a simple policy to swap unused mapping tables out to flash to make room for those in use before the application starts.

3.2.4 Transaction support

KAML's transaction mechanism supports fine-grained data access with high concurrency. This design decision separates KAML from most previous proposals for multi-part atomic write support in SSDs [47, 91]. Specifically, KAML-SSD provides native support for fine-grained, multi-record atomic writes, while the KAML caching layer provides support for data buffering and locking. KAML's native atomic write provides durability and atomicity, and the caching layer implements isolation.

The `Put` command provides the transactional interface. It accepts arrays of namespace IDs, keys, values, and value sizes that specify a set of updates/insertions to apply, and the SSD guarantees that all the updates/insertions occur atomically.

`Put` executes in three phases. In the first phase, the firmware receives a list of key-value pairs to update or create. The KAML firmware transfers the data to the SSD's non-volatile RAM. Then, the firmware checks if each key already exists in the namespace's index. If it does, it locks that entry. If the key does not exist, the firmware reserves and locks an empty entry in the index. After the firmware has acquired the locks, the operation has logically committed, and the firmware notifies the host.

In the second phase, the firmware initiates flash write operations to write the key-value pairs to flash. When a flash write completes, the firmware records its physical address. If a failure occurs before all flash write operations succeed, the firmware recovers using the data in the non-volatile buffers.

Once all the flash write operations are complete, the firmware has the information

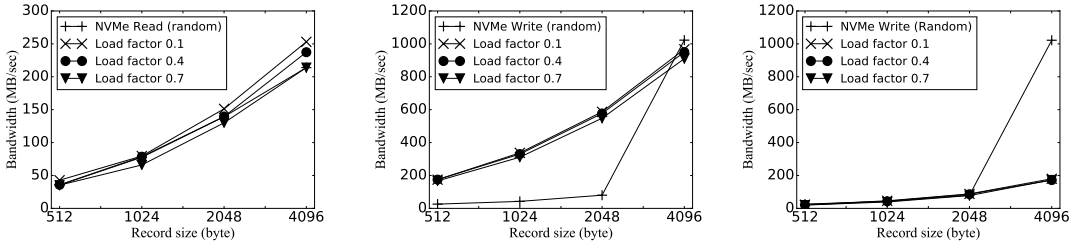
of all new flash addresses. The firmware updates the indices with the physical addresses of the newly-written key-values in flash memory. Once the firmware finishes applying the changes to the mapping tables, it releases locks, frees the buffers and marks the command as success. If a failure occurs, the firmware recovers using the flash addresses recorded in the second phase.

3.2.5 Wear-leveling and GC

KAML's GC algorithm operates on variable-sized key-value pairs rather than fixed-sized pages, but the basic operations are similar to other SSD GC schemes. The GC subsystem maintains a list of erased flash blocks for each log and selects a new one when the block at the end of the log is full. After writing all pages in a block, KAML firmware moves the block to a sorted list that orders blocks by their erase count and the amount of valid data (i.e., bytes that have not been replaced by a newer version of the same record).

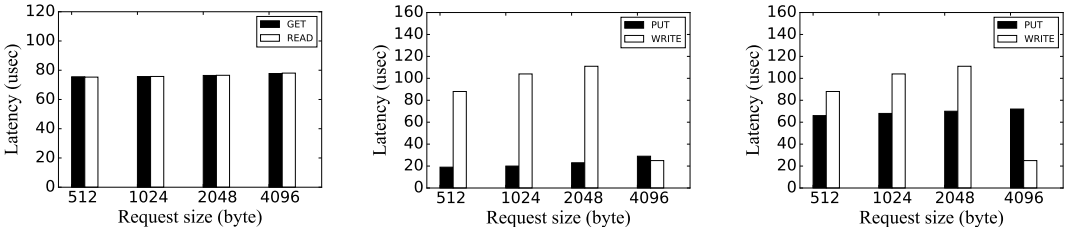
KAML's GC algorithm starts reclaiming space when the free block count falls below a threshold. It selects blocks to clean that have low erase counts and small amounts of valid data. This serves to spread erases evenly across the blocks and minimize the work needed to copy the valid data to a new block.

Once it has selected a flash block to clean, the firmware reads the pages from flash memory into the storage DRAM, and uses the per-page bitmap described in Section 3.2.2 to extract all the key-value pairs. For each key-value pair, the firmware searches for the key in the index. If the search result matches the physical address of the current key-value pair being scanned, the key-value pair is valid and the firmware writes it back to a new location. If the firmware cannot find an entry for the key-value pair or the search result points to a different physical location, the key-value pair is invalid and the firmware discards it. After examining all pages in a block, the firmware erases the block and adds it to the list of free blocks.



(a) `Fetch` (b) `Update` (c) `Insert`

Figure 3.5: **Bandwidth comparison between KAML and block I/O** `Get` outperforms `read` by up to 20% in `Fetch`. `Put` outperforms `write` by up to 7.9× for small requests in `Update` and 10% in `Insert`. `Put`'s overhead is greater than that of `write` when `Put` inserts *new* elements whose sizes are 4KB.



(a) `Fetch` (b) `Update` (c) `Insert`

Figure 3.6: **Latency comparison between KAML and block I/O** `Get` has almost the same latency as `read` for `Fetch`. `Put` latency is much lower than that of `write` (20%) for small requests in `Update` since writing small records incurs flash read-modify-write. In `Insert` the latency of `Put` is between 63% and 75% of that of `write` for small requests due to the same reason.

3.3 Evaluation

We use microbenchmarks to measure basic operation performance of KAML, discussing factors that affect KAML performance. Then we run more complex OLTP and NoSQL workloads to quantify how KAML design decisions impact application-level performance. Below we detail our experimental setup and the results of our testing.

3.3.1 Experimental setup

We built KAML using an industrial flash-based SSD reference platform that connects to the host machine via four-lane PCIe Gen 3. The controller of the SSD consists of multiple embedded processors running at 500 MHz. Data structures such as address mapping tables reside in the 2 GB on-board DRAM. We assume that the SSD uses a capacitor or battery to protect it from abrupt power failures. We implemented KAML by modifying a reference firmware provided by the board manufacturer. The original reference firmware supports only block-based I/O, while our implementation exposes the KAML commands as extensions to the standard NVMe command set. We make corresponding extensions to the Linux NVMe driver. Thus KAML is fully compatible with block devices. Before running the experiments, we preconditioned the device by filling the SSD with random data multiple times.

Our host system has two quad-core Intel Xeon E5520 CPUs on a dual-socket motherboard. This machine contains 64 GB of DRAM as the host main memory. The machine runs a Linux 3.16.3 kernel. We obtained all the results without using hyper-threading.

For microbenchmarks that measure bandwidth, we use eight threads running on the host machine to continuously issue I/O requests to the SSD. To eliminate the overhead of Linux virtual file system (VFS), the driver and the user-space library allow the baseline program to issue `read` and `write` commands directly to the SSD. The KAML version of microbenchmarks use `Get` and `Put` commands to manipulate a single record in each command. To measure KAML's latency, we issue commands using a single thread.

For OLTP and NoSQL key-value store applications, we compare KAML with Shore-MT [93], an open-source storage engine that offers ACID guarantees using a combination of ARIES-style [68] logging and two-phase locking (2PL). In our experiments, Shore-MT provides the same functionality as KAML with techniques used in many

popular databases, thus serves as an optimized baseline with low overhead. Shore-MT relies on a file system to store user data and logs. Unless otherwise noted, we configure Shore-MT to use record-level locks, since our measurements show that they provide better performance than page-level locks.

3.3.2 Microbenchmarks

We use `Fetch`, `Update` and `Insert` microbenchmarks to understand the performance characteristics of KAML. The baseline version of `Fetch` uses NVMe `read` commands to retrieve sectors from the SSD. The baseline version of `Insert` uses NVMe `write` commands to write sectors of data to previously unmapped LBAs, and `Update` uses NVMe `write` commands to write new versions of data to mapped LBAs. The KAML version of `Fetch` uses `Get` commands to retrieve records by their keys directly. For `Update` and `Insert`, we use `Put` commands to modify existing records and create new ones, respectively.

Figure 3.5 compares the throughput of `read`/`write` with that of `Get`/`Put` commands while varying the value size. The test uses one 1024 MB mapping table that can hold up to 64 million entries. Since the number of elements in the mapping table can affect KAML performance, Figure 3.5 reports the throughput of KAML requests when the mapping table has different number of elements, i.e. load factors.

The experimental results demonstrate that when the load factor of the mapping table is 0.1, the bandwidth of `Get` can be up to 1.2$\times$ of `read`. When the load factor is 0.4, `Get` and `read` achieve the same bandwidth. `read` starts to outperform `Get` after the load factor surpasses 0.7. `Get` achieves greater throughput than `read` because `Get` avoids the cost of LBA indirection. The SSD firmware has to obtain locks on LBA ranges to protect the data within the LBA range from changing or migrating during the `read` command. The performance gain of `Get` diminishes when the mapping table has more elements,

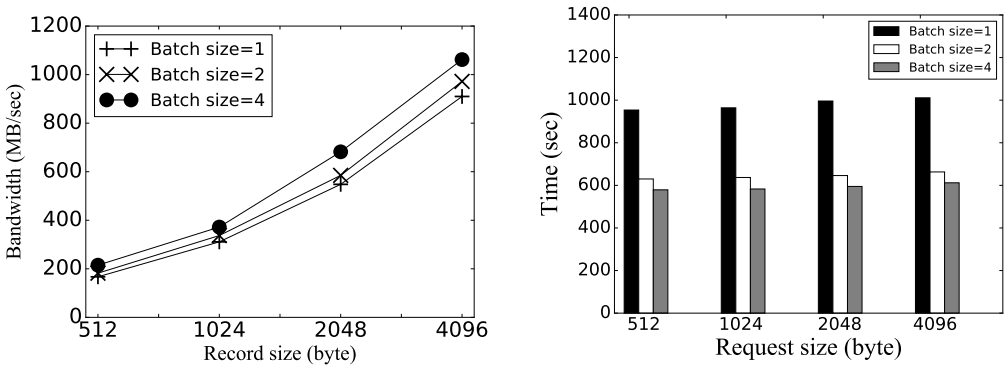
since the firmware has to scan more mapping table entries to search for an element.

In `Update`, `Put` outperforms `write` by $6.7\times - 7.9\times$ when the request size is smaller than 4 KB. `write` achieves marginal improvement over `Put` when the record size reaches 4 KB. The performance of `write` sees a huge leap when the record size reaches 4 KB because in the baseline, `write` requests smaller than 4 KB are treated as read-modify-write, indicating the command cannot return before the firmware reads 4 KB from flash into the RAM and modifies a portion of it. When the request size is 4 KB, the command can return after writing the new data into persistent DRAM in the SSD. KAML does not suffer from this limitation for small records since it adopts a log-structured approach and never overwrites previous data.

In `Insert`, the throughput of `Put` is close to that of `write` for requests smaller than 4 KB. `write` outperforms `Put` for 4 KB requests. On the one hand, 4 KB `write` does not have to perform long-latency read-modify-write operation. On the other hand, `Put` needs to insert new address mapping entries into the hash-based mapping table while `write` updates an element in an array.

Figure 3.6 compares the latency of `read`/`write` with that of `Get`/`Put` commands with different value size when the hash table load factor is 0.4.

`Get` and `read` have the same latency. For `Update`, `Put` latency is only 20% of that of `write` for small requests. The latency of `write` for small requests is high due to the read-modify-write described before. In contrast, `Put` does not suffer from the latency increase when the record size is smaller than 4 KB. For `Insert`, the latency of `Put` is between 63% and 75% of `write` when request size is smaller than 4 KB. For 4 KB requests, `Put` latency is 2.9 $\times$ of `write`. Our experiments also show that for `Get`, 98% of the latency comes from hardware including the PCIe link and SSD internal latency, while the remaining 2% comes from software including user space and OS kernel. In `Update`, hardware contributes 92% of the latency. In `Insert`, hardware latency is 97%



(a) `Update` (b) `Insert`

Figure 3.7: **Effect of batch size on KAML bandwidth** Increasing the number of key-value pairs to update or insert by a `Put` command improves throughput of `Update` by 1.2× - 1.3×, and reduces the amount of time by 40% to populate an empty namespace to 70% full (mapping table load factor reaches 0.7).

of the total.

In addition to updating/inserting a single record with each command, the `Put` command supports updating/inserting a *batch* of records atomically. The number of records in a batch is the *batch size*. Figure 3.7 measures the effect of batch sizes on performance and shows that increasing batch size leads to larger bandwidth. Specifically, increasing the batch size from 1 to 4 leads to 1.2× – 1.3× increase in throughput for `Update`. It also reduces the amount of time required to add data to the namespace by 40%.

3.3.3 Effect of number of logs

KAML-SSD allows for the configurable allocation of internal write bandwidth to user applications in terms of the number of logs associated with each namespace to append. Figure 3.8 shows the effect of the number of logs on bandwidth of `Put` in the `Update` benchmark. When the number of logs increases from 16 to 64, the maximum bandwidth that the namespace can achieve increases by 5.8× since more logs can support

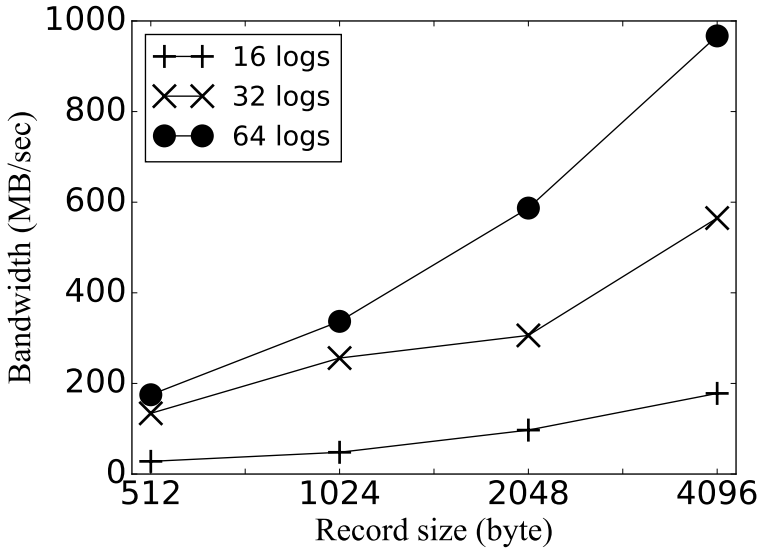


Figure 3.8: **Effect of multiple KAML logs** Increasing the number of logs from 16 to 64 leads to 5.8× increase in throughput.

more concurrent commands.

3.3.4 OLTP

To quantify the benefits of KAML for OLTP workloads, we implemented both TPC-B [96] and a subset of TPC-C [97] using the API that the KAML caching layer provides. The caching layer effectively serves as a database storage engine in these implementations. Shore-kits [92], an open-source benchmark suite using Shore-MT, provides a reference implementation for TPC-B and TPC-C. For the sake of fair comparison, our implementation of TPC-B and TPC-C uses the same lock manager as Shore-MT, and leverages Shore-kits' code to serialize/deserialize database records. Due to current hardware limitation and to comply with the TPC specifications, all values are 512 bytes except TPCC CUSTOMER table, whose values are 1024 bytes. Shore-MT's default page size is 8 KB. The scaling factor for both benchmarks is 100, yielding 10 million values for TPCB and 60 million values for TPCC so that the mapping tables of each benchmark

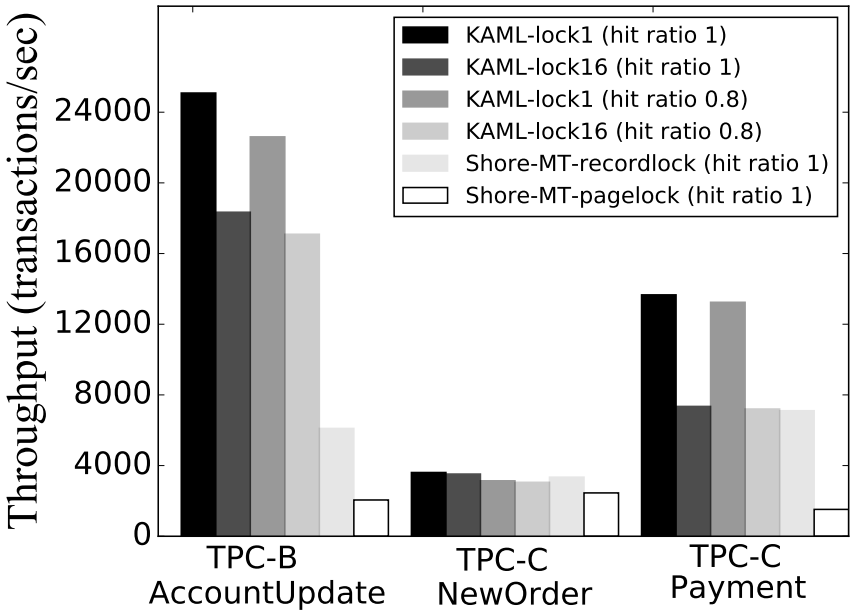


Figure 3.9: **Throughput of OLTP workloads on KAML** KAML outperforms Shore-MT with record-level lock by 4.0× for TPC-B, 1.1× for TPC-C `NewOrder` and 2.0× for `Payment`. Coarse-grained locks negatively impact throughput.

can fit in the in-SSD DRAM. We configure the amount of memory allocated to KAML caching layer to compare the performance when hit ratios are 0.8 and 1.0 respectively. Shore-MT allocates sufficient memory to the buffer pool so that the entire working set can fit in host main memory. This is ideal for conventional SQL databases and storage engines. To guarantee atomicity and durability, both KAML and Shore-MT must flush data to the SSD when transactions commit. For KAML, we vary the number of records protected by a lock to be 1 and 16. For Shore-MT, we run the benchmarks with both record-level and page-level locks.

Figure 3.9 shows the throughput of running TPC-B's `AccountUpdate` transaction, TPC-C's `NewOrder` transaction and `Payment` transaction. These three transactions are the most important (i.e., frequently executed) queries in the benchmarks. The results show two things: First, KAML outperforms Shore-MT with record-level lock by up to

4.0× for TPC-B's `AccountUpdate`, 1.1× for TPC-C's `NewOrder`, and 2.0× for TPC-C's `Payment`. Second, coarse-grained locks negatively impact transaction throughput. For example, KAML's throughput drops by up to 47% when the number of records protected by a lock increases from 1 to 16. Likewise, Shore-MT's performance drops by up to 80% when it switches from record-level locks to page-level locks.

Performance advantage over Shore-MT

Several factors account for KAML performance advantage over Shore-MT with record-level locks. First, centralized, synchronous logging is the major bottleneck in most conventional storage engines [43] with ARIES-style logging, and only a single transaction can acquire the global lock and flush the log at the same time. This transaction will block other transactions even if there is no data conflict among these transactions, and this transaction cannot commit before data becomes persistent in the SSDs via an `fsync`. Consequently, the system under-utilizes the bandwidth of modern SSDs. In contrast, the KAML's `Put` command allows multiple transactions to commit in parallel if they do not contend for the same records, making full use of SSD's I/O bandwidth.

Second, conventional storage engines have to perform checkpointing and copy dirty data out of the log to limit log size. Although this happens in the background, it can interfere with foreground activity. The resulting degradation in performance is in addition to the performance impact of SSD garbage collection. In KAML, the log cleaning happens in the SSD, so the application only suffers the effect of one layer of garbage collection rather than two.

Finally, KAML avoids the extra layers of indirection that the file system adds and that Shore-MT (and other conventional storage engines) must pay the price for. In this regard, KAML's transactions are lighter-weight and allow for faster commit.

Impact of locking granularity

Support for fine-grained locking is one of the key advantages of KAML over existing key-value SSDs. Intuitively, the cost of coarse-grained locking results from the contention between multiple transactions for exclusive access to data – especially "hot" data. We analyze the performance impact of locking granularity.

Assume there are K keys divided into pages that hold l keys each. A lock protects each page. If N updates that each target key i with probability p_i arrive at about the same time, we can reduce the problem to the classic "balls into bins" problem [80].

We denote the N updates as req_1, req_2, ..., req_N that arrive at t_1, t_2, ..., t_N. The update requests arrive in chronological order, i.e. the arrival times satisfy the following condition,

$$0 < t_1 < t_2 < ... < t_N \tag{3.1}$$

Consider the ith update request. When the ith request arrives, requests req_1,req_2, ..., req_{i-1} have already arrived. The probability of req_i not having to contend with these previous requests is the following,

$$P_i = p_1(1-p_1)^{i-1} + p_2(1-p_2)^{i-1} + ... + p_{\frac{K}{l}}(1-p_{\frac{K}{l}})^{i-1} \tag{3.2}$$

$$= \sum_{j=1}^{\frac{K}{l}} p_j(1-p_j)^{i-1} \tag{3.3}$$

Therefore, the expected number of conflicts for all the requests is

$$\mathbb{E}[conflicting\ requests] = \sum_{i=1}^{N}(1 - \sum_{j=1}^{\frac{K}{l}} p_j(1-p_j)^{i-1}) \tag{3.4}$$

$$= N - \sum_{i=1}^{N}\sum_{j=1}^{\frac{K}{l}} p_j(1-p_j)^{i-1} \tag{3.5}$$

$$= N - \sum_{j=1}^{\frac{K}{l}}\sum_{i=1}^{N} p_j(1-p_j)^{i-1} \tag{3.6}$$

$$= N - \sum_{j=1}^{\frac{K}{l}} p_j \sum_{i=1}^{N}(1-p_j)^{i-1} \tag{3.7}$$

$$= N - \sum_{j=1}^{\frac{K}{l}} p_j \frac{1-(1-p_j)^N}{1-(1-p_j)} \tag{3.8}$$

$$= N - \sum_{j=1}^{\frac{K}{l}} (1-(1-p_j)^N) \tag{3.9}$$

$$= N - \frac{K}{l} + \sum_{j=1}^{\frac{K}{l}} (1-p_j)^N \tag{3.10}$$

which gives the expected number of conflicts (i.e., the number of requests that will contend for a page lock) as:

$$\mathbb{E}[conflicting\ requests] = N - \frac{K}{l} + \sum_{i=1}^{\frac{K}{l}} (1-p_i)^N \tag{3.11}$$

If the choice of keys satisfy uniform distribution, i.e. $p_i = \frac{1}{K}$ for $0 \leq i \leq K-1$, the formula becomes

$$\mathbb{E}[conflicting\ requests] = N - \frac{K}{l}[1-(1-\frac{1}{K})^N] \tag{3.12}$$

Table 3.3: **YCSB workloads summary** The ratio of different operations in each workload.

workload	read	update	insert	read-modify-write
a	0.5	0.5	0	0
b	0.95	0.05	0	0
c	1	0	0	0
d	0.95	0	0.05	0
f	0.5	0	0	0.5

So, as l increases, the number conflicts increases as well. As a result, KAML and many databases adopt fine-grained record-level locking.

3.3.5 NoSQL key-value store

The KAML caching layer can serve as a NoSQL key-value store. This section compares the performance of YCSB benchmark [104] on KAML and Shore-MT. In both cases, we populate the key-value store with 20 million 1024-bytes records and allocate 8 GB main memory to the buffer pool. We choose not to cache the entire data set in memory since we want to test the performance of `Get`.

Table 3.3 summarizes YCSB workloads, and Figure 3.10 shows the throughput of those workloads running on KAML and Shore-MT. KAML outperforms Shore-MT by between 1.1× and 3.0×. KAML achieves more performance improvement over Shore-MT for write-intensive workloads than for read-intensive ones.

3.4 Related work

Many systems have explored SSD architecture and key-value store optimization. Below we describe these efforts and place KAML in context with them.

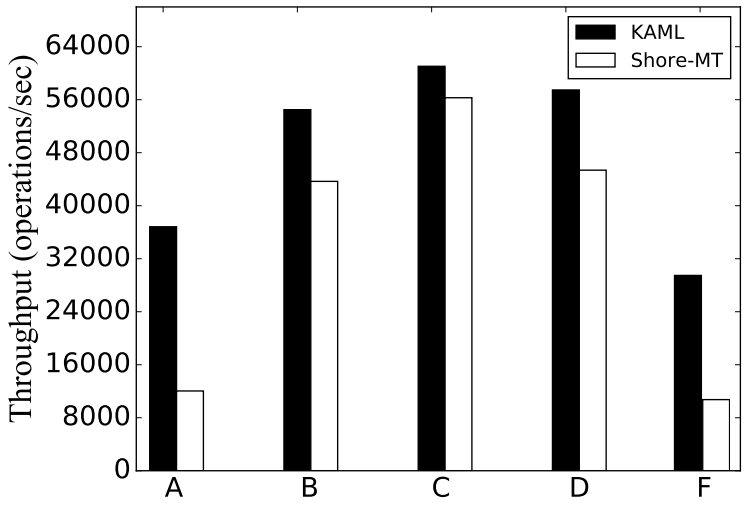


Figure 3.10: **YCSB throughput on KAML** KAML achieves up to 3.0× throughput gain relative to Shore-MT, while the average improvement is 2.3×.

3.4.1 Innovations in SSD architecture

The architecture of SSDs is an active field of study, and researchers have addressed several opportunities and challenges that SSDs present [45, 106, 99, 33, 67, 91, 22, 61]. These include devising new interfaces to take advantage of flash memory's characteristics and refining the design of the FTL. Below we discuss key developments in each of these areas and how KAML differs from previous work.

Novel interfaces

The flash memory that SSDs use to store data suggests a wide range of alternatives to the conventional block-based interface for storage.

The multi-streamed SSD [45] allows host applications to categorize accesses into "streams" that the SSD can manage differently depending on their characteristics. KAML can manage different namespaces according to different policies, providing a wider range of policy options than multi-stream SSDs, and it makes those parameters explicit in the

SSD's interface. Nameless Writes [106] remove the indirection from LBA to PPN by allowing the file system to access flash memory via physical address and exposing the data movement that occurs during GC to the host. KAML goes a step further, mapping user-specified keys (not just LBAs) to PPNs, so that applications can more easily exploit the mapping inside the SSD.

SDF [75] targets datacenter workloads. SDF's minimum write size is a flash block, and it exposes individual channels to applications which must explicitly manage erasure and GC. LOCS [99] is an LSM-tree-based key-value store [73] exploiting the internal bandwidth of SDF. Data analytics can also benefit from new SSD interfaces that provide in-storage data filtering [33]. Besides exposing new interfaces, people have proposed adding programmability to SSDs, making it more convenient to change the interaction between the host and the SSD [90].

Seagate Kinetic Open Storage Platform [88] supports access to remote storage via key-value interface over Ethernet. Seagate currently offers only hard drives with the interface but SSDs are an obvious extension. While Kinetic targets data transfer to/from remote disks, current KAML prototype focuses on local SSDs with lower latency and higher throughput. Moreover, KAML provides a key-value interface with transactional semantics supporting fine-grained isolation.

Transactions on SSDs

Several projects add transaction support to SSDs by leveraging the copy-on-write facilities that FTLs already employ. TxFlash [77] enables applications to write multiple pages atomically using a novel commit protocol. Atomic-write [76] extends a log-based FTL and uses one bit per block to track if the block is part of an atomic-write. Atomic-write can eliminate double-write that induces considerable overhead in InnoDB [42]. X-FTL [47] and Möbius [91] atomically update multiple pages. However, they perform

operations on entire pages, and, therefore require page-level lock. MARS [28] supports fine-grained atomic-write, but assumes the SSD uses byte-addressable non-volatile memory which is not commercially available yet. In contrast, KAML uses flash which will remain the dominant solid state storage technology in the foreseeable future.

FTL Design

There have been a number of efforts focusing on the design of efficient FTL [35, 26, 56, 39, 64, 100, 22, 49]. DFTL [39] selectively caches page-level address mappings to improve the performance for random writes. CAFTL [22] uses the hash value of the data to detect duplicates to improve the lifetime and performance of SSD. CA-SSD [40] exploits the value locality of data access pattern to reduce SSD response time with small additional hardware support. Compared with these efforts, KAML generalizes its FTL to a set of mapping tables from user-specified keys to physical addresses. Instead of a single array, KAML supports variable number of mapping tables.

Virtual Storage Layer (VSL) [98] is a software FTL between the SSD and applications tailored specifically to FusionIO's SSDs. It supports sparse addressing, dynamic mapping and transactional persistence. VSL keeps FTL data structures in the host memory. KAML adopts an alternative approach that offloads the FTL to the SSD. Therefore KAML consumes less host resources but requires careful budgeting of SSD's internal resources. Manufacturers have put more powerful processors and larger RAMs into SSDs and this trend will likely continue in the foreseeable future.

3.4.2 Key-Value stores in SSDs

Researchers have built key-value stores optimized for SSDs. FlashStore [31] and SkimpyStash [32] store key-value pairs on an SSD and apply various optimizations to improve performance and memory consumption, but they rely on conventional SSDs

resulting in stacked logs and the inefficiencies that those entail. Similarly, using Bw-Tree [58] as a key-value database leads to similar issues. In contrast, KAML implements a key-value interface in the SSD, eliminating stacked logs and reducing complexity while providing fast access via its caching layer.

SILT [62] is a high-performance, memory-efficient storage system on a single node. It has three key-value stores with different optimization goals for memory-efficiency and write performance. KAML combines application index with original FTL by mapping keys to physical addresses directly. The new mapping tables reside in battery-backed DRAM inside the SSD, thus reducing the amount of host memory usage.

NVMKV [67] builds upon VSL [98] to implement key-value stores. Its mapping table resides in host memory and keys in different stores are evenly distributed across the entire LBA space. Furthermore, NVMKV relies on LevelDB's [59] read cache. In contrast, KAML provides separate namespaces for different key-value stores and a generic caching layer.

3.5 Summary

Existing proposals to modernize the SSD interface by providing key-value semantics present a single inflexible key space, while proposals for transactional SSDs require inefficient coarse-grained locking. KAML solves these problems by supporting multiple, independent namespaces for key-value pairs and allow applications to tune the performance of each namespace to meet application requirements. It also enables fine-grained locking by treating key-value pairs as the basic unit of transactional operations. Finally, KAML provides a caching layer analogous to a conventional page cache to improve performance. These changes allow KAML to outperform conventional designs in a wide range of workloads.

Acknowledgments

This chapter contains material from “KAML: A Flexible, High-Performance Key-Value SSD”, by Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson, which appears in *The 23rd IEEE Symposium on High Performance Computer Architecture*, (HPCA 2017). The dissertation author was the primary investigator and author of this paper.

Chapter 4

Improving SSD lifetime with byte-addressable metadata

Due to the idiosyncrasies of flash, solid state drives (SSDs) implement complex flash translation layers (FTLs) to hide the details of flash, including its limited lifetime. To support this, NAND flash devices provide out-of-band (OOB) regions on each flash page that the FTLs use to store metadata. Storing the metadata in flash limits its utility since the OOB region is subject to the same idiosyncrasies as the primary "in-band" data.

Emerging non-volatile byte-addressable memories avoid the idiosyncrasies of flash memory and will eventually enable very fast SSDs. However, in the near future, these memories will be too expensive to replace flash-based SSDs completely. A more economical alternative is to use non-volatile, byte-addressable memories to store FTL metadata (i.e. the OOB data). Such a change would come at a modest price but could dramatically increase the flexibility of FTLs and enable a wide range of useful features.

Based on these observations, we have developed a prototype SSD named *PebbleSSD* that allows us to explore the implications and applications of *byte-addressable metadata (BAM)*. We describe the architecture of PebbleSSD and the new FTL features

it enables, and use those features to improve the efficiency of two log-structured file systems.

The first new feature of PebbleSSD is the use of BAM to store the logical address to which each physical page has been mapped by the FTL. This BAM-based mapping is the inverse of the normal logical-to-physical address mapping that most FTLs already store in the conventional DRAM. The PebbleSSD supports a new command `remap` that allows the FTL to dynamically change the bi-directional mapping between logical and physical addresses, achieving flexible and fast relocation of data in the logical address space without having to write data to flash.

`remap` makes log cleaning more efficient. Original log-structured file systems have to read valid data from their previous locations and write them to new destinations. In contrast, our custom log-structured file systems running on PebbleSSD can "move" data without writing to flash, thus effectively reducing the amount of data written during log cleaning.

The second new feature of PebbleSSD is using BAM to temporarily store file system metadata. PebbleSSD provides a new command `fs_write` that persists not only file data blocks[1] in flash memory, but also their offset in the file and file inode number in BAM.

`fs_write` allows for more efficient flushing of file data blocks. This allows log-structured file systems to avoid the so-called "wandering tree problem" in which the file systems write data and index blocks to newly allocated space, thus writing a data block can cause recursive flushing of its ancestor index blocks in the tree-based block mapping [19]. With `fs_write`, log-structured file systems write only file data blocks during `fsync`. The file systems do not have to write the index blocks during `fsync`

[1] Throughout this chapter, we refer to the granularity of flash erase operation as "flash erase block", "flash block" or "erase block", while "block", "data block", "node block", "file block" etc. refer to the basic data unit in file systems.

since the information in BAM written by `fs_write` is already sufficient to guarantee the recoverability of file. Therefore BAM allows log-structured file system to write less data to flash, thus improving the efficiency and lifetime of PebbleSSD.

We implement PebbleSSD on a commercial, flash-based SSD reference design. We modify NILFS2 [7] and F2FS [53] to demonstrate the benefits of PebbleSSD. The experimental results show that PebbleSSD can reduce the amount of data written by log-structured file systems during log cleaning by up to 99% with the `remap` command. PebbleSSD's write-optimized file block mapping and `fs_write` reduce the flash write by up to 33% for a wide range of workloads.

The rest of the paper is organized as follows. Section 4.1 presents an overview of the system. Section 4.2 and Section 4.3 describes the components of the system in detail. Section 4.4 presents our experimental results. Section 4.5 places this work in the context of existing research, while Section 4.6 summarizes this chapter.

4.1 System overview

PebbleSSD has a heterogeneous architecture using flash memory as primary data storage and NVRAM to store BAM in addition to metadata in the flash's OOB.

Figure 4.1 illustrates the relationship between PebbleSSD and other components of the system. This section describes system components while Section 4.2 and Section 4.3 present their implementation in more detail.

4.1.1 PebbleSSD interface

PebbleSSD uses the BAM to enable a new interface that allows applications to manage their data more efficiently. Table 4.1 summarizes the commands included in this interface.

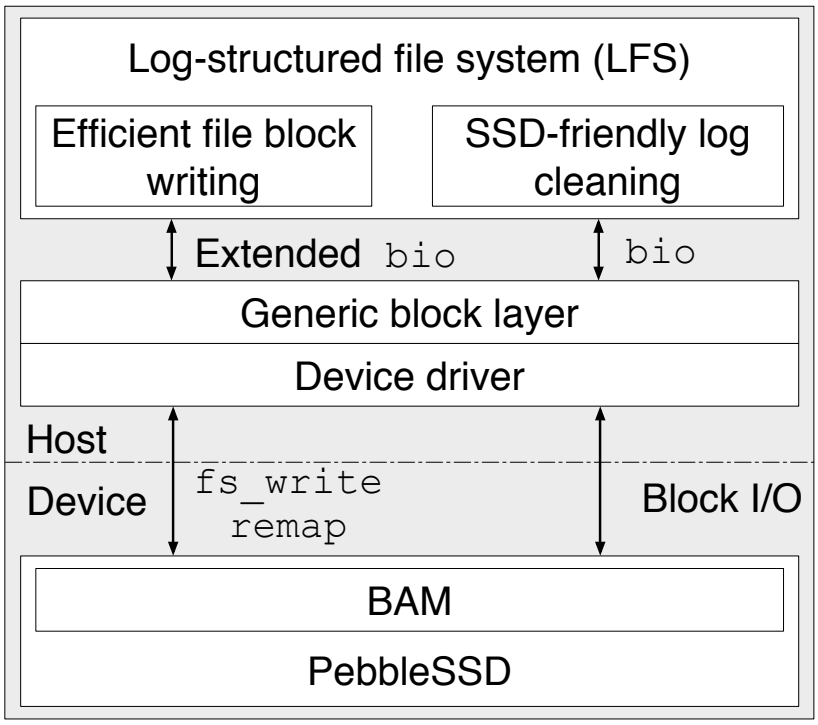
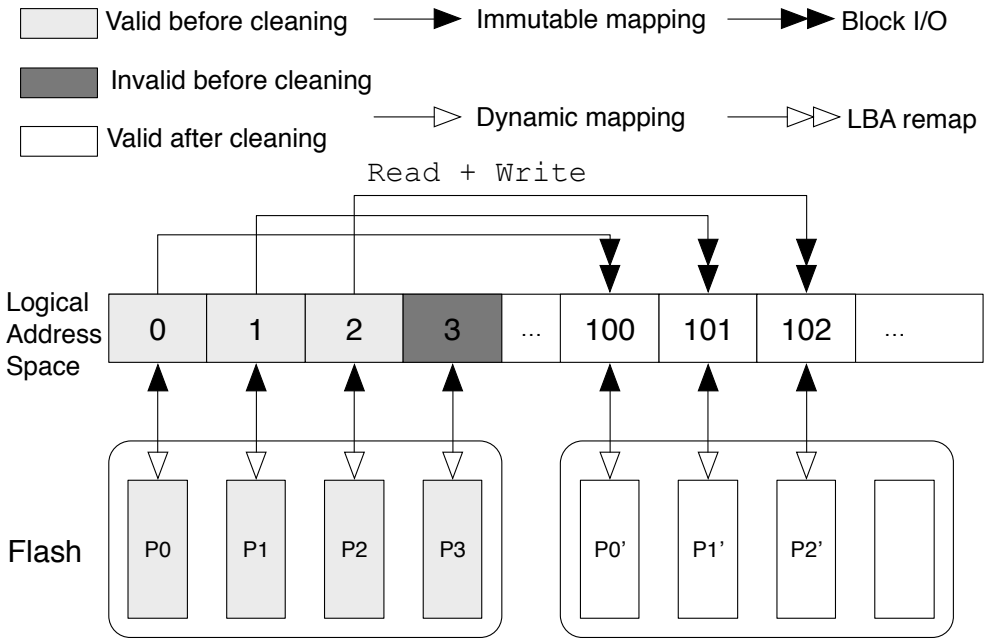


Figure 4.1: **Pebble system architecture** The system consists of the following components: PebbleSSD, log-structured file system, and block layer and device driver.

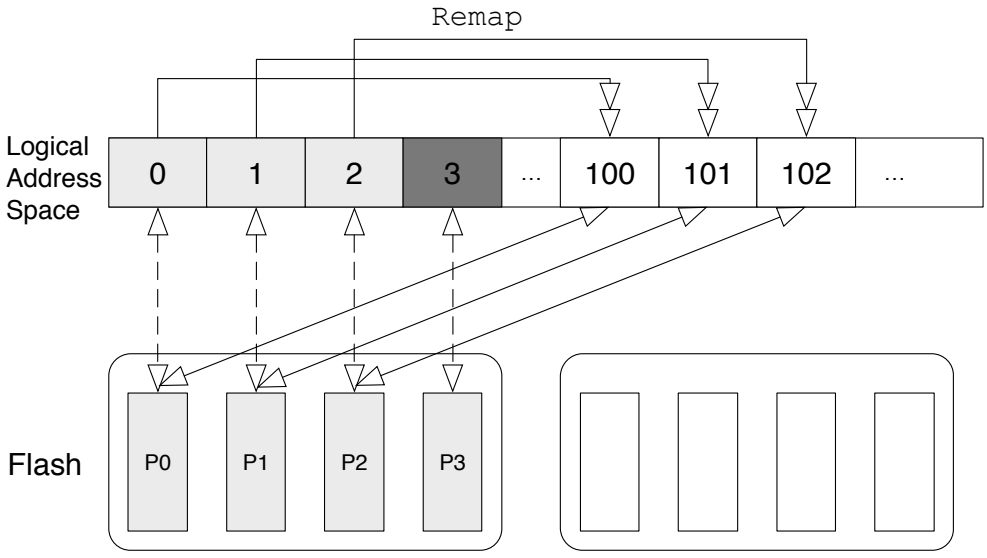
First, PebbleSSD stores the physical-to-logical address mapping in BAM. This is different from conventional designs that store such mapping in flash-based OOB region. The `remap` command can dynamically change both the normal LBA-to-PPN mapping and the BAM-based PPN-to-LBA mapping.

Second, the `fs_write` command not only writes data to flash memory, but also updates their corresponding metadata in BAM. This can be useful to file systems because they can avoid having to issue multiple `write` commands to flash memory. For example, file systems can issue a single `fs_write` command to write file data blocks as well as update the pointers pointing to them.

Third, PebbleSSD supports conventional `write`, `read` and `trim` commands to maintain compatibility.



(a) Log cleaning with conventional SSD



(b) Log cleaning with PebbleSSD

Figure 4.2: **Log cleaning** The `remap` command allows log-structured file systems to remap physical pages to new LBAs without writing flash memory.

Table 4.1: **PebbleSSD commands** The new `fs_write` and `remap` commands allow applications e.g. file systems to reduce the amount of data written to flash.

Command	Description
`read`(startLBA, num)	Read num sectors from startLBA
`write`(startLBA, num)	Write num sectors from startLBA
`trim`(startLBA, num)	Mark num sectors starting from startLBA as invalid
`fs_write`(startLBA, num, file, offstInFile)	Write num sectors starting from startLBA for a file.
`remap`(srcLBA, dstLBA, num)	Move num sectors starting from srcLBA to dstLBA.

4.1.2 PebbleSSD applications

System can use PebbleSSD's new commands in a variety of ways. We focus on two optimizations especially suited to log-structured file systems: using `remap` to reduce the cost of log cleaning and using `fs_write` to improve the efficiency of writing data blocks.

Log-structured file systems have to move clean, valid data from their original logical segments to newly allocated ones during log cleaning. To reduce data movement overhead, log-structured file systems prefer to select old segments for cleaning. Therefore, there is a high chance that the pages in this segment are already clean and persistent in flash memory.

Log structured file systems can use PebbleSSD's `remap` command to reduce the cost of moving clean data to new log segments. Rather than copying the data using `read` and `write` commands, the file system can `remap` the clean data into the new segment. Figure 4.2 shows this operation in action.

Log structured file system can use `fs_write` to avoid the wandering tree problem described in Section 2.4. The file system can use `fs_write` to store file inode numbers

and data blocks' offsets within their files in the BAM, allowing the file system to defer the flushing of node blocks in the conventional tree-based block mapping index untile next checkpoint.

4.2 PebbleSSD

We have built PebbleSSD on a commercial SSD development board. Our implementation includes a customized firmware, a custom driver and modified Linux block layer to support BAM. The PebbleSSD firmware manages the flash memory and BAM area. The modified PebbleSSD block driver supports an extended interface that includes `remap` and `fs_write`. The block layer invokes the device driver to issue low-level commands to PebbleSSD. In this section we describe in detail the core features of PebbleSSD focusing on `remap` and `fs_write` commands.

4.2.1 Hardware architecture

Our commercial NVMe [71] SSD development platform connects to the host machine via four-lane PCIe 3.0. It comprises an array of SLC flash chips (375 GB in total) organized in 16 channels. The channels are connected to a multi-core controller. PebbleSSD firmware runs on the controller to manage flash memory and provide support for its commands.

Figure 4.3 shows the hardware architecture of PebbleSSD. The most important components are the embedded cores, the flash interface and the in-SSD DRAM.

The SSD controller comprises multiple embedded processors running firmware at 500 MHz. Each of the cores in the controller has 64 KB private instruction and data memories. An on-chip network connects the cores, the flash interface and the DRAM so that they can communicate with each other. The embedded cores issue commands to the

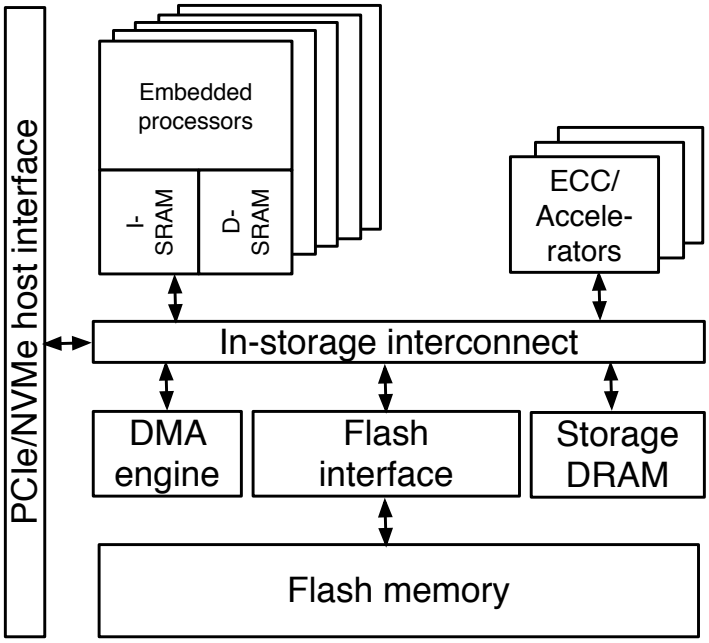


Figure 4.3: **PebbleSSD architecture** PebbleSSD exposes its internal DRAM and processing power to host programs via PCIe/NVMe interface.

flash interface which reads or writes data from/to a buffer in the PebbleSSD's internal DRAM. The cores have to initiate explicit data movements between the DRAM and their private data memories to operate on the data stored in the DRAM.

The in-device DRAM (totalling 2 GB) holds PebbleSSD's BAM as well as FTL's metadata. The DRAM also stores statistics and state information for each flash erase block so that the firmware can perform garbage collection and wear leveling efficiently.

In this paper, we assume that the DRAM is persistent. In practice, it would either be battery or capacitor-backed. New memory technologies e.g. Intel 3D-XPoint memory [14] could also replace the original DRAM.

4.2.2 BAM

The BAM is the heart of PebbleSSD's new features, and the firmware stores mapping information in the BAM to support `remap` and `fs_write`. Below we describe the implementation of `remap` and `fs_write` in more detail.

Physical-to-logical mapping and `remap`

PebbleSSD uses the BAM to track the bidirectional mapping between logical and physical address spaces, and the `remap` command exposes the mapping to software. The PPN-to-LBA table is an inverse of the normal LBA-to-PPN table that conventional FTLs use to emulate a block I/O interface.

PebbleSSD stores the PPN-to-LBA mapping table in the BAM. PebbleSSD uses a 64-bit integer to represent this information. The total size of the PPN-to-LBA table is no larger than that of the LBA-to-PPN table since the physical address space of flash memory available in an SSD is usually smaller than the 64-bit logical address space.

The current PebbleSSD firmware implements this PPN-to-LBA mapping as an array. To keep the bidirectional mapping between logical and physical address space consistent and up-to-date, the `write` and `remap` commands always update the LBA-to-PPN and PPN-to-LBA tables together.

The `remap` command operates on both LBA-to-PPN and PPN-to-LBA mapping table. The `remap` command takes three parameters, `srcLBA`, `dstLBA` and `num`. The firmware copies the content of the `num` consecutive mapping table entries starting from `srcLBA` to the entries starting from `dstLBA`. Therefore the actual data stored on those physical pages can be accessed from a new logical address `dstLBA` by future `read` operations. In this way, `remap` achieves the movement of data from `srcLBA` to `dstLBA` at low cost without incurring any flash write.

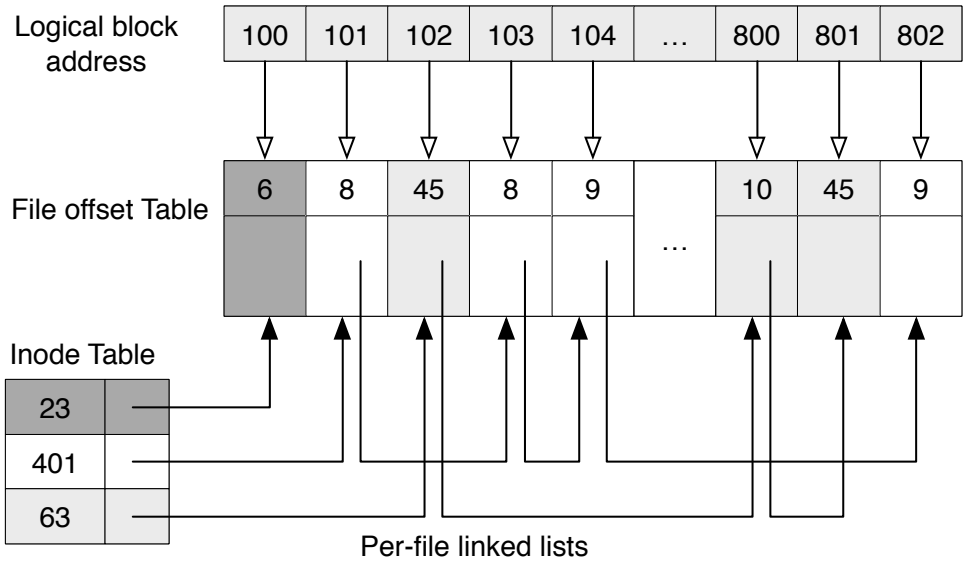


Figure 4.4: **PebbleSSD write-optimized file block mapping** PebbleSSD stores in the BAM an auxiliary file block mapping for files. This file block mapping comprises an inode table and a file offset table whose entries form per-file linked lists. The linked lists keep track of data blocks that have been written for each file since the latest checkpoint.

Efficient file block mapping and `fs_write`

PebbleSSD can also store the mapping from data blocks to corresponding file-system-specific information in BAM. Figure 4.4 illustrates the mapping tables used for this purpose. Each entry in the table has two components. The first component stores the data block's offset within the file it belongs to, and the second component is a pointer that forms a linked list with other data blocks in the file.

The space required for each entry depends on file system data block size, the maximum size of an individual file and the capacity of the BAM. For example, if the file system's data block size is 4 KB, a file can reach 16 TB at most, and the BAM has a total capacity of 4 GB, then the firmware can use two 32-bit integers to store each entry.

For each opened file that supports `write` operation, PebbleSSD firmware maintains a linked list of the aforementioned mapping table entries. By traversing the per-file

linked list, the firmware retrieves the metadata of all the data blocks that belong to the file and that have been written back by the file system since the most recent file system checkpoint.

The linked lists are append-only. PebbleSSD is able to traverse the linked list and retrieve entries, but can only add new entries to its tail. The `fs_write` command causes the firmware to add mapping table entries to the linked list in addition to flushing a number of data blocks to flash memory. Once the firmware finishes the insertion into the linked list, the data blocks logically become part of the file so that the firmware can later locate them by traversing the linked list in BAM. Thus the file system no longer has to flush the index blocks of the file to flash memory frequently, effectively breaking the recursive updates in the wandering tree problem.

A separate hash table residing in BAM maps a file's inode number to the head of its linked list. PebbleSSD uses this hash table to locate the linked list that belongs to a particular file.

4.2.3 Block layer and device driver

The PebbleSSD requires a custom Linux block layer and device driver so that software can use the `fs_write` and `remap` commands. The block layer accepts extended `bio` requests from the log-structured file system and passes the `bio` structure to the SSD driver. In the extended `bio` struct, the field `bi_rw` encodes the type of the operation, and two new fields contain additional arguments to supply to the device driver, e.g. `offsetInfile`, `inodeNum` or `srcLBA`. Then the device driver translates the `bio` to appropriate NVMe commands and issues them to PebbleSSD.

4.3 File systems customization

Log-structured file systems provide two opportunities to apply PebbleSSD's new capabilities. Leveraging the new commands allows these file systems to reduce the amount of data written to flash improving the SSD's lifetime. The `remap` command of PebbleSSD allows log-structured file systems to perform fast and efficient log cleaning and reduce the amount of data written to the SSD. The `fs_write` command avoids the recursive updates on other interior node blocks in the tree-based block mapping index.

4.3.1 Log cleaning

Log-structured file systems perform log cleaning to create free, contiguous segments in the logical address space to service future write requests. During the cleaning process, log-structured file systems scan the logs, copy valid data to new locations, discard invalid data and reclaim the space originally occupied.

The `remap` command supports efficient movement of data inside the SSD without incurring flash writes and host-device data transfers. Log structured file systems can use `remap` to avoid copying data to a new segment as long as the data in the segment is clean (i.e., there is not a more recent update waiting in the operating system page cache).

Determining whether a block is clean is not always simple. The obvious approach is to use the Linux page descriptor's dirty bit, but this not always sufficient. For instance, F2FS [53] employs a "lazy migration" policy that marks valid data blocks in the page cache as dirty so the writeback thread will write them back later. To leverage `remap` in our version of F2FS, we used a new status bit to indicate whether this block is truly dirty or just marked dirty by the log cleaning thread.

The file system also has to track the old LBAs of all blocks before issuing the `remap` command. The original NILFS2 [7] fails to do so. After reading a file block into

page cache, NILFS2 overwrites the original LBA. Consequently when the log cleaning thread later migrates this file block, it has no information about its source location. To address this issue, for each clean and up-to-date file block that the log cleaning thread plans to migrate, the `buffer_head` struct has an extra 64-bit field to store the old LBA.

4.3.2 Data block writing

Log-structured file systems can use PebbleSSD's new interface to improve the efficiency of writing file data blocks. The file system can issue one `fs_write` command to persist data blocks in flash memory and their metadata in the BAM area. The metadata include the blocks' offset within their file and the file inode number. This allows the file system to avoid the recursive updates of index blocks containing pointers pointing to the data blocks.

When writing back data blocks to a file with `fs_write`, the log-structured file systems send the following information to PebbleSSD via the block layer and device driver: `startLBA`, `length`, `inodeNum` and `offsetInFile`. The `fs_write` stores data blocks in flash memory and their associated metadata in the BAM area. Since PebbleSSD already allocates sufficient space for both inode table and file offset table, `fs_write` does not dynamically increase the space consumption of BAM.

The file system periodically flushes the original tree-based block mapping index to flash memory during checkpoint to limit the number of dirty blocks in the index. This helps reduce file system metadata loss and recovery overhead in the event of failures.

The file system maintains the conventional block mapping in page cache and continue to use it for normal file system operations unless there is a failure. The file system performs periodic checkpointing and write back the node blocks of the conventional block mapping index. After the checkpointing task finishes, the file systems request that PebbleSSD reset the BAM-based block mapping via a vendor-specific NVMe command

supported by our prototype. The function of this command is to clear a region in the NVRAM inside PebbleSSD.

The file systems use the BAM-based block mapping only for the purpose of recovery. Should an error occur, a kernel thread performs recovery on the data blocks of this file. The thread retrieves the <`inodeNum`, `offsetInFile`> metadata from the BAM by reading its content into host main memory and reconstruct the normal tree-based block mapping.

4.4 Evaluation

We run microbenchmarks to measure the performance of PebbleSSD with a focus on the new commands it provides. Then we measure the performance and efficiency of log cleaning of both F2FS and NILFS2 with `remap` and compare with their baseline implementations. We refer to the customized versions as `F2FS-opt` and `NILFS2-opt`, while we refer to the baseline as `F2FS-baseline` and `NILFS2-baseline`. Finally we run workloads to quantify the benefit of `fs_write` on F2FS and NILFS2. Again, the customized versions are `F2FS-opt` and `NILFS2-opt`, while the baseline versions are `F2FS-baseline` and `NILFS2-baseline`. In this section, we describe our experimental setup and results in detail.

4.4.1 Experimental setup

We implemented PebbleSSD by modifying its original reference firmware provided by the SSD manufacturer. The original firmware supports only conventional block-based I/O, while PebbleSSD extends the NVMe command set to provide additional interfaces, e.g. `fs_write` and `remap`. To allow host programs to use PebbleSSD, we also modified Linux NVMe driver and block layer. Thus PebbleSSD supports not only

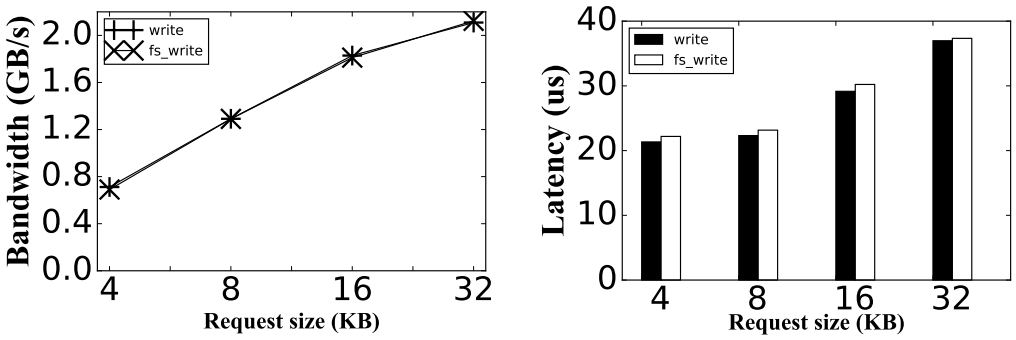


Figure 4.5: **Performance comparison of `fs_write` and `write`** (a) shows that `fs_write` achieves the same throughput as that of `write` since the overhead of linked list operations is minimal. (b) shows that `fs_write` and `write` have similar latency.

block-based I/O, but also `fs_write` and `remap`. Before running the experiments, we preconditioned the SSD by filling it with random data multiple times.

The host machine that we use has a quad-core Intel Xeon E3-1230 CPUs on a single-socket motherboard. This machine contains 32 GB of DRAM as main memory and runs a customized Linux 3.16.3 kernel. We collected all results without enabling hyper-threading.

Microbenchmarks measure the bandwidth and latency of the new commands `fs_write` and `remap` provided by PebbleSSD. To measure PebbleSSD's bandwidth, four threads running on the host machine issue commands to PebbleSSD continuously. To measure latency, only a single thread issues commands to the SSD.

To measure the performance and efficiency of `remap` on log-structured file systems, we first run a subset of Filebench [3] workloads. We run the workloads for equal amount of time on each baseline file system and its customized version. Then we manually trigger file-system-specific log cleaning subroutines which keeps executing until there is no data to clean.

We run a subset of Filebench, an open-source implementation [10] of TPC-C [97]

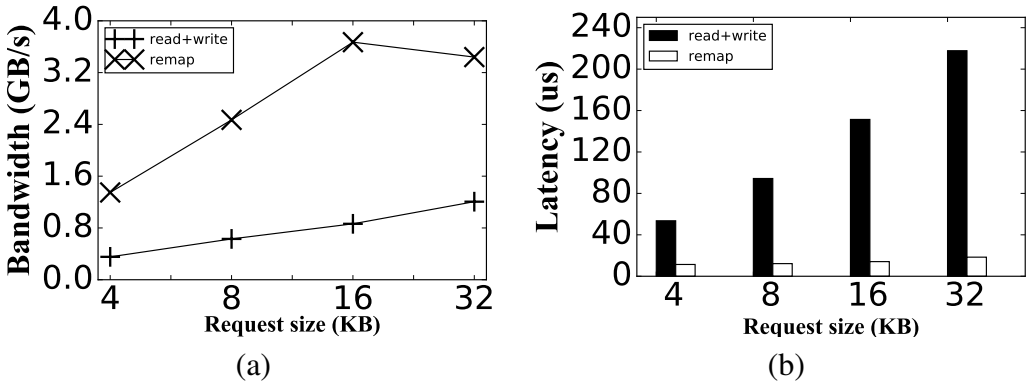


Figure 4.6: **Performance of `MOVE` with conventional SSD and PebbleSSD** (a) shows that the `remap`-based implementation of `MOVE` achieves 3.7× improvement in throughput and (b) shows that it achieves 87% reduction in latency, compared with conventional implementation using `read` and `write`.

and LinkBench [4] to quantify the effect of `fs_write` on F2FS and NILFS2, comparing with their baseline implementations.

4.4.2 Microbenchmarks

In comparison with original `write` command, the `fs_write` command not only writes the block to SSD, but also updates its BAM. Since we are concerned about the single command performance here, we eliminated the overhead of Linux virtual file system (VFS) by allowing applications to send `fs_write` commands to PebbleSSD directly via `ioctl`. Figure 4.5 presents the throughput and latency of `fs_write` and compares it with the original `write` command. The extra overhead caused by `fs_write`'s linked list operations is minimal and `fs_write` achieves almost the same performance as `write`. Although as a single command, `fs_write` does not provide better performance than `write`, but as we will demonstrate in Section 4.4.4, log-structured file systems can utilize `fs_write` to improve the performance and lifetime of SSDs.

Another microbenchmark named `MOVE` measures the performance of `remap` command. `MOVE` requires host program to move data from source LBAs to destination LBAs.

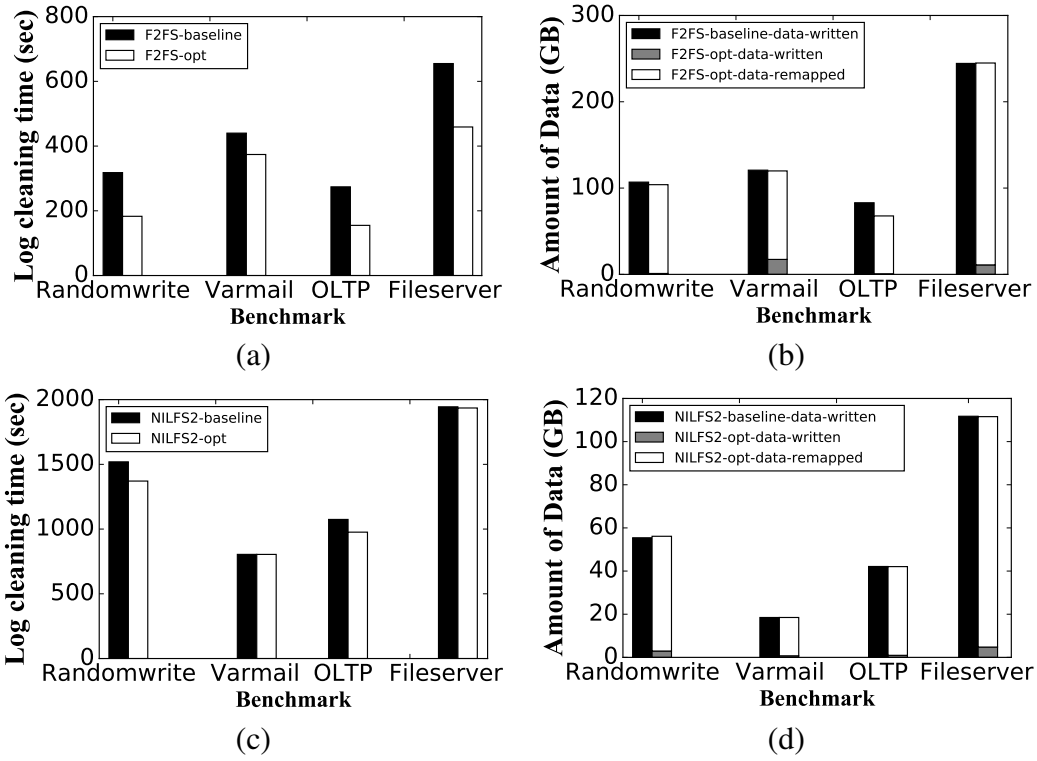


Figure 4.7: **Improvement on log cleaning efficiency due to `remap`** In (a) and (b), `F2FS-opt` with `remap` requires 29% less time to clean the segments than `F2FS-baseline`, and writes 96% less data in average (99% in the best case) during log cleaning. In (c) and (d), `NILFS2-opt` also requires less time to clean the old segments than `NILFS2-baseline`, and saves write traffic by 97% in average (99% in the best case) during log cleaning.

The baseline version of `MOVE` implementation first reads data from PebbleSSD into host main memory and then writes them to their destination locations in the SSD. In contrast, `remap`-based implementation uses the `remap` command to remap data from source LBAs to destinations. Apparently the baseline incurs multiple data transfer between the host and PebbleSSD while `remap`-based version does not suffer from this overhead. As a result, `remap`-based version can achieve 3.7× improvement in throughput and reduces the latency by 87% on average, as shown in Figure 4.6. Both implementation sends commands from the host to PebbleSSD via `ioctl` directly.

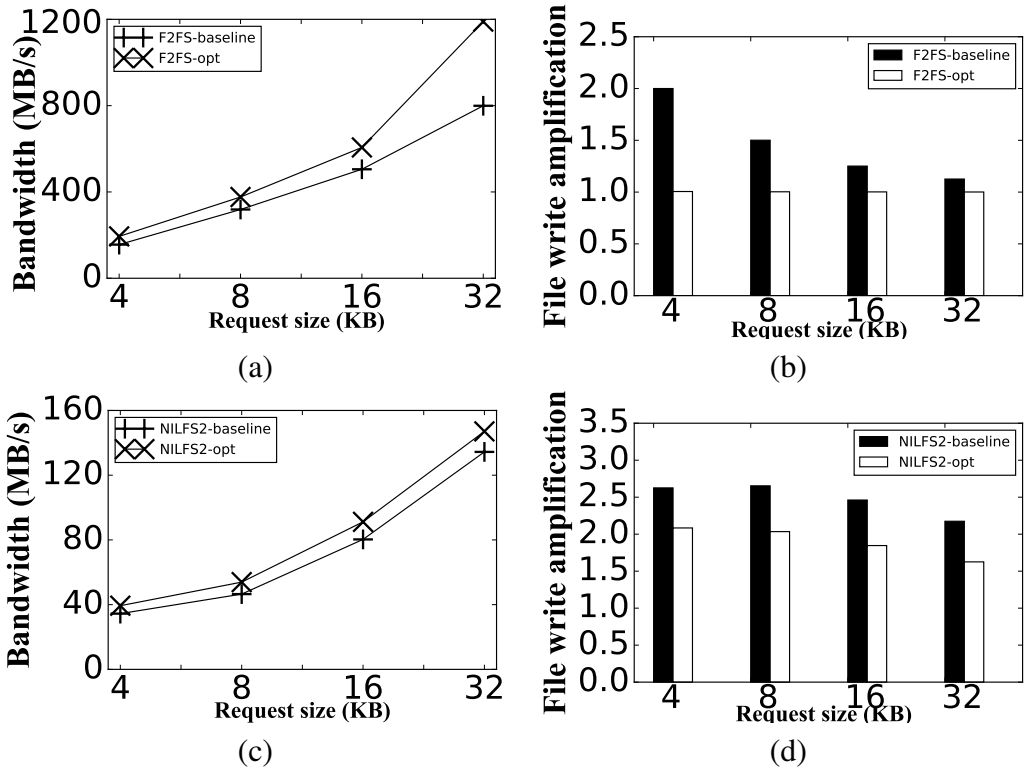


Figure 4.8: **Improvement on writing to files with `fs_write`** (a) shows that with `fs_write`, `F2FS-opt` achieves 1.28× improvement in throughput while (b) shows that `F2FS-opt` reduces file system write amplification by 28%. (c) and (d) show similary results. With `fs_write`, `NILFS2-opt` improves throughput by 1.13× and reduces file write amplification by 24%, compared with `NILFS2-baseline`.

4.4.3 Efficient log cleaning

Figure 4.7 depicts the effect of using `remap` in the log cleaning of F2FS and NILFS2. In most cases, a kernel thread executes the log cleaning subroutine for F2FS and the kernel thread wakes up periodically, typically between every 30 and 60 seconds. In order to measure how much time it takes to clean all used segments in F2FS, we add an `ioctl` to F2FS to trigger F2FS log cleaning mannually and let the log cleaning subroutine run continuously until there is nothing left to clean. For NILFS2, we use its own `nilfs-clean` utility program [6] that performs log cleaning and space reclamation after a certain period of time specified by the user. In our experiments, we set the period

to 600 seconds.

Before starting log cleaning subroutines, we first run the corresponding workload to emulate a used SSD. Figure 4.7 (a) shows the time required to finish log cleaning of F2FS, and Figure 4.7 (b) shows the amount of data written by F2FS log cleaning. `F2FS-opt` spends 33% less time than `F2FS-baseline` to finish cleaning the segments in average. Figure 4.7 (b) can account for the reason of this improvement. `F2FS-baseline` always causes flash write operations during log cleaning, while `F2FS-opt` can use `remap` to clean the majority of data. In the case of `F2FS-opt`, the log cleaning subroutine writes up to 99% less data than `F2FS-baseline` in average. The rest of data are cleaned using the `remap` command which does not incurr flash write at all.

Figure 4.7 (c) and (d) depict the result for NILFS2 log cleaning. Similarly, `nilfs-clean` writes up to 97% less data in `NILFS2-opt` than `NILFS2-baseline` because the former can use `remap` to move data. Since NILFS2 is not optimized for parallel storage devices e.g. SSDs, `nilfs-clean` does not fully utilize the I/O bandwidth. Consequently, the saving in log cleaning time is not as big as in F2FS. `NILFS2-opt` improves the SSD lifetime because it incurs less flash write than `NILFS2-baseline`.

4.4.4 Write-optimized file block index

We measure the impact of `fs_write` on F2FS and NILFS2. `F2FS-baseline` already writes only direct node blocks to the SSD during `fsync` [53] to improve the performance of `fsync`. `F2FS-opt` uses `fs_write` to further avoid writing direct node blocks. To compare the effectiveness in reducing the amount of data written to the SSD, we use a metric called *file write amplification* similar to the concept in [64] which is defined as the quotient of the total amount data written (data and index blocks) over the amount of file data blocks.

Figure 4.8 illustrates the advantage of `F2FS-opt` over `F2FS-baseline` and

NILFS2-opt over NILFS2-baseline in a simple benchmark. The benchmark executes four threads each of which writes data to its dedicated file synchronously. We assign each thread to its own file to avoid the contention for the lock on the file's inode. Figure 4.8 (a) shows that with F2FS-opt, the aggregate throughput of the threads achieves 1.28× improvement comparing with F2FS-baseline. Figure 4.8 (b) shows that the file write amplification of F2FS-opt is 28% lower than that of F2FS-baseline. Figure 4.8 (c) similarly shows that the benchmark throughput on NILFS2-opt is 1.13× the throughput on NILFS2-baseline. Figure 4.8 (d) shows that file write amplification of NILFS2-opt is 24% less than that of NILFS2-baseline. The performance improvement and reduction in file write amplification result from fs_write that writes only data blocks for both F2FS-opt and NILFS2-opt.

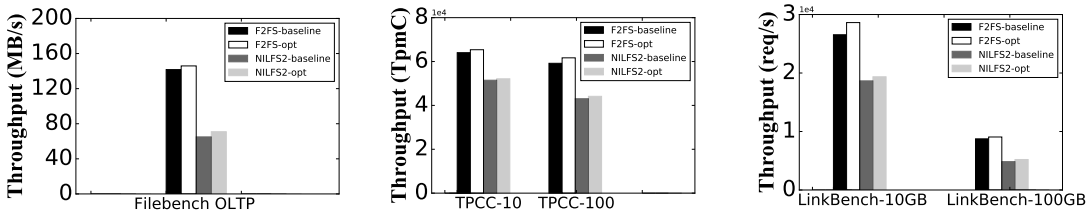


Figure 4.9: **Performance comparison between F2FS-opt, NILFS2-opt and their baseline implementations** In (a), Filebench OLTP running on F2FS-opt and NILFS2-opt achieve 1.02× and 1.09× improvement in throughput. In (b), TPCC running on F2FS-opt and NILFS2-opt can achieve slightly better performance than on F2FS-baseline and NILFS2-baseline respectively. In (c) LinkBench can also achieve slightly larger throughput on F2FS-opt NILFS2-opt than on F2FS-baseline and NILFS2-baseline.

Figure 4.9 and Figure 4.10 further compare the effect of running more complex benchmarks on F2FS-baseline, F2FS-opt, NILFS2-baseline and NILFS2-opt. Figure 4.9 (a) presents the throughput of Filebench OLTP benchmark, and F2FS-opt outperforms F2FS-baseline by 3% while NILFS2-opt achieves 9% improvement. Figure 4.9 (b) shows the throughput of TPC-C varying the size of data set. The total

data set size of TPCC-10 and TPCC-100 are approximately 800 MB and 8 GB respectively. Both benchmarks run on top of MySQL [5] 5.5 with InnoDB [42] whose buffer pool size is 16 GB. In both cases, `F2FS-opt` outperforms `F2FS-baseline` by a small margin and the same holds for `NILFS2-opt` and `NILFS2-baseline`. Figure 4.9 (c) shows the result for LinkBench with different sized data sets. Similarly, `F2FS-opt` and `NILFS2-opt` lead to larger throughput than `F2FS-baseline` and `NILFS2-baseline` respectively. The throughput improvement is not high because even `F2FS-baseline` and `NILFS2-baseline` cannot fully saturate the IO bandwidth of PebbleSSD, thus making them write less data does not lead to larger throughput.

Figure 4.10 shows that for Filebench OLTP, TPCC-10 and LinkBench-10GB, `F2FS-opt` can use `fs_write` to reduce the file write amplification from close to 1.5 to nearly 1.0. Similarly, `NILFS2-opt` reduces file write amplification from 2.7 to 2.0. When the data set size increases, especially in the case of LinkBench-100GB, the reduction in file write amplification diminishes since MySQL will have to move data between main memory buffer pool and the SSD, leading to lower write throughput. Consequently the workloads issue fewer `fsync` to the underlying log-structured file systems.

4.4.5 BAM space utilization

With `remap` and `fs_write`, PebbleSSD can make more efficient use of its internal NVRAM than conventional SSDs.

As described in Section 4.2.1, PebbleSSD exposes 375 GB flash to the host and has 2 GB NVRAM to store BAM. Assume the total amount of user-visible flash is 512 GB for simpler calculation.

According to the implementation of original firmware, a conventional SSD based on this platform uses 4 bytes for each address mapping table entry, and each entry maps 4 KB in logical address space to the physical address space. This configuration leads to

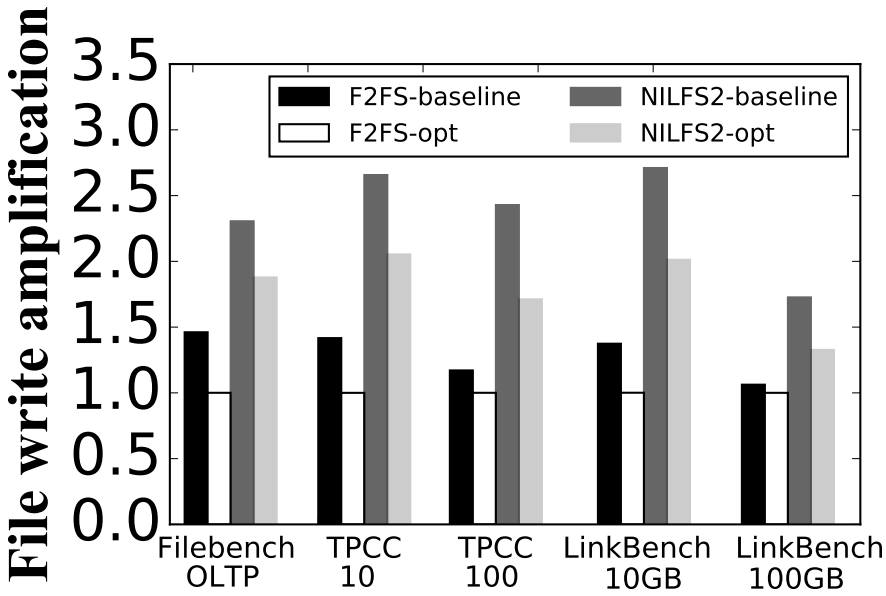


Figure 4.10: **Improvement on file write amplification for F2FS and NILFS2** `F2FS-opt` can reduce file write amplification from 1.5 to nearly 1.0 for synchronous write-intensive workloads e.g. database applications. `NILFS2-opt` can reduce file write amplification from 2.7 to 2.0 for the same workloads.

2^{27} entries, consuming 512 MB of the BAM. Since the original SSD keeps the PPN-to-LBA mapping in flash memory, the space utilization of BAM is about 35% due to the logical-to-physical mapping and some other system data structures.

To support `remap`, PebbleSSD maintains both logical-to-physical and physical-to-logical mappings in BAM. The entries in both tables are 4 bytes in size and describes the mapping for 4 KB regions. Therefore, the total space consumption will be 1 GB, leading to a 60% utilization of BAM.

For `fs_write`, the original address mapping table also occupies 512 MB of BAM. Both the file offset table and inode table use 8-byte entries. The file offset table has 2^{27} entries, consuming 1 GB BAM. The inode table keeps an entry for each file that is currently open with read-write access. In our current system, we support 1 million files that can be open for write at the same time. Considering the load factor of the hash-based inode table, this requires approximately 16 MB space in BAM. The space utilization of

BAM can reach 80%.

4.5 Related work

Many prior research efforts have explored various techniques aiming to reduce write amplification in flash memory and SSDs. In this section, we present a review of these efforts and place PebbleSSD in the context with them.

4.5.1 Backward pointer

The inverse mapping is similar to the backward pointers in some previous work. Many file systems and FTLs have adopted the backward pointer mechanism to reduce the overhead of persistent data migration and synchronization. File systems, e.g. Pilot [81] file system, Pangaea [85], NoFS [23] use backward pointers to ensure system consistency. BtrFS [1] and Backlog [66] maintain back references to support dynamic relocation of data blocks without flushing index updates to persistent storage. An object-based FTL, i.e. OFTL [64] stores for each page the information of the object to which the page belongs. Such backward reference resides in the flash-based OOB region of each page, thus does not support in-place update. In contrast, PebbleSSD stores the inverse mapping in dedicated NVRAM-based OOB region to support efficient modification. In addition, PebbleSSD allows for easy and efficient retrieval of each file's inverse mapping by maintaining per-file linked list of mapping table entries.

4.5.2 Address map manipulation

Most flash-based SSDs introduce a layer of indirection due to the FTL's address map [27]. One of the most important design goals of FTL is to improve the lifetime of the device. Researchers have proposed a number of FTLs [35, 56, 39, 22, 100, 49, 98].

CAFTL [22] exploits the hashed signature of data chunks to detect duplicates and reduce flash writes, leading to improvement in SSD's lifetime. FTL2 [100] implements atomicty at the level of FTL so as to reduce the overhead of writing database logs to flash. DFTL [39] selectively caches page-level mapping. These approaches still focus on the normal logical-to-physical address mapping table, while PebbleSSD goes a step further to use BAM to enhance the flexibility of FTLs.

Several systems have also researched the possibility of achieving efficient flash-write-free data movement by manipulating this map directly. JFTL [24] remaps addresses of journal pages to their original locations in the logical address space without writing the same data to flash memory. ANViL [101] allows host system to access the address map and supports snapshot, deduplication and single-write journaling. Researchers evaluated JFTL on a simulation platform, while ANVil's address map resides in the main memory of host machine. Furthermore, neither of them discussed how to extend the idea of address map manipulation to an SSD with flash-based OOB. If the FTL stores the LBA in the spare space of each flash page, merely modifying the LBA-to-PPN address map does not fulfil the goal of moving data in the logical address space. Instead flash write is inevitable to modify the metadata in the OOB region. In contrast, PebbleSSD goes a step further to address this issue and presents an evaluation of the effectiveness of its solution.

SHARE [72] is able to remap addresses inside the SSD and achieve write atomicity. SHARE caches a subset of the PPN-to-LBA mapping entries in the SDRAM inside the OpenSSD [8], therefore, the firmware determines which entries to keep and which to evict to flash. This approach is able to handle large amount of flash with relatively small SDRAM, but requires extra logic at the firmware level to implement cache eviction policies. Furthermore, system performance can also be affected by the size of the cache. In contrast, PebbleSSD keeps the entire PPN-to-LBA mapping in BAM for the functionality of `remap`. PebbleSSD needs larger BAM size than SHARE, but does not require a cache

management policy. SHARE and PebbleSSD represents different points in the design space with different trade-offs and considerations. Depending on the optimization goal, either can be more preferable than the other.

4.5.3 Coordinated garbage collection

A storage system with a log-structured file system running on a flash-based SSD has multiple layers of logs, and the gaps between them lead to inefficiencies such as unnecessary data migration [103].

Researchers have proposed coordinating file system log cleaning with SSD's garbage collection to improve flash device lifetime. Application-Managed Flash (AMF) [55] and ParaFS [105] both employ a coordinated garbage collection approach with supports from both the FTL and file system. During garbage collection, the host file system migrates data to its proper new locations by consulting domain-specific knowledge about data placement, while the FTL shoulders the responsibility of erasing flash blocks for future use. Both file systems need to accommodate the physical characteristics of flash memory. For example, they need to ensure that a flash block is erased first before writing to its flash pages. This potentially requires a non-trivial re-engineering of the I/O path because general file systems do not have this limitation. Furthermore, host CPUs are involved in the GC of flash memory. In comparison, PebbleSSD does not require the file system to be aware of the physical layout and characteristics of underlying flash. Thus modification of legacy file system is moderate, and host CPUs can also be freed from GC.

Coordinated GC has also been discussed in other conventional file systems running on SSDs. EXT3 [2] with nameless write [106] allows the FTL to move valid flash pages and inform the host file system of the new physical addresses of data pages via *migration callbacks*. PebbleSSD, in contrast, lets the file system determine the new

logical addresses for its data.

4.5.4 Metadata caching

Some efforts use NVRAM-based storage as a durable cache for database or file system metadata to reduce synchronous writes to flash memory. NVMFS [78] stores file system metadata in NVM dimms attached to the memory bus. DuraSSD [46] uses non-volatile in-device cache to support atomicity and durability. Cooperative Data Management [54] considers NVM as a cache and allows the NVM-resident copy to invalidate flash-resident copy. In contrast, PebbleSSD does not cache identical copies of data in the NVRAM. Instead, PebbleSSD stores metadata i.e. inverse mapping in the NVRAM, saving host interface bandwidth and NVRAM space.

4.6 Summary

By providing BAM in the byte-addressable OOB region, PebbleSSD can support a range of useful features to improve device lifetime, including write-optimized file block mapping and fast, efficient log-cleaning. PebbleSSD exposes two new commands, `fs_write` and `remap` allowing file systems to access BAM. Log-structured file systems can use `fs_write` command to avoid recursive updates on file index blocks, and use `remap` command to perform fast and efficient movement of valid data during log cleaning. In both cases, log-structured file systems achieve better performance while reducing flash writes. Therefore BAM is effective in improving SSDs' lifetime.

Acknowledgments

This chapter contains material from the paper, “Improving SSD Lifetime with Byte-Addressable Metadata”, by Yanqin Jin, Yannis Papakonstantinou and Steven Swanson, which has been submitted for publication. The dissertation author is the first investigator and author of the paper.

Chapter 5

Willow: a user-programmable SSD

The scope of possible new interfaces is enormously broad and includes both general-purpose and application-specific approaches. Recent work has illustrated some of the possibilities and their potential benefits. For instance, an SSD can support complex atomic operations [28, 76, 77], native caching operations [18, 86], a large, sparse storage address space [44], delegating storage allocation decisions to the SSD [106], and offloading file system permission checks to hardware [21]. These new interfaces allow applications to leverage SSDs' low latency, ample internal bandwidth, and on-board computational resources, and they can lead to huge improvements in performance.

Although these features are useful, the current one-at-a-time approach to implementing them suffers from several limitations. First, adding features is complex and requires access to SSD internals, so only the SSD manufacturer can add them. Second, the code must be trusted, since it can access or destroy any of the data in the SSD. Third, to be cost-effective for manufacturers to develop, market, and maintain, the new features must be useful to many users and/or across many applications. Selecting widely applicable interfaces for complex use cases is very difficult. For example, editable atomic writes [28] were designed to support ARIES-style write-ahead logging, but not

all databases take that approach.

To overcome these limitations, we propose to make programmability a central feature of the SSD interface, so ordinary programmers can safely extend their SSDs' functionality. The resulting system, called *Willow*, will allow application, file system, and operating system programmers to install customized (and potentially untrusted) *SSD Apps* that can modify and extend the SSD's behavior.

Applications will be able to exploit this kind of programmability in (at least) four different ways.

- **Data-dependent logic:** Many storage applications perform data-dependent read and write operations to manipulate on-disk data structures. Each data-dependent operation requires a round-trip between a conventional SSD and the host across the system bus (i.e., PCIe, SATA, or SAS) and through the operating system, adding latency and increasing host-side software costs.

- **Semantic extensions:** Storage features like caching and logging require changes to the *semantics* of storage accesses. For instance, a write to a caching device could include setting a dirty bit for the affected blocks.

- **Privileged execution:** Executing privileged code in the SSD will allow it to take over operating and file system functions. Recent work [21] shows that issuing a request to an SSD via an OS-bypass interface is faster than a system call, so running some trusted code in the SSD would improve performance.

- **Data intensive computations:** Moving data-intensive computations to the storage system has many applications, and previous work has explored this direction in disks [82, 83, 51] and SSDs [48, 20, 95] with promising results.

Willow focuses on the first three of these use cases and demonstrates that adding generic programmability to the SSD interface can significantly reduce the cost and

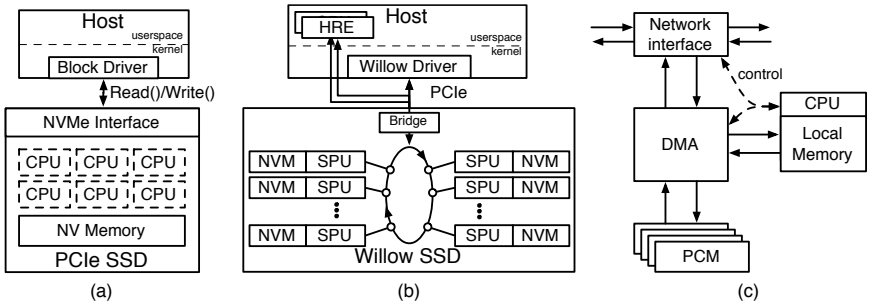


Figure 5.1: **A conventional SSD vs. Willow** Although both a conventional SSD (a) and Willow (b) contain programmable components, Willow's computation resources (c) are visible to the programmer and provide a flexible programming model.

complexity of adding new features. We describe a prototype implementation of Willow based on emulated PCM memory that supports a wide range of applications. Then, we describe the motivation behind the design decisions we made in building the prototype. We report on our experience implementing a suite of example SSD Apps. The results show that Willow allows programmers to quickly add new features to an SSD and that applications can realize significant gains by offloading functionality to Willow.

This chapter provides an overview of Willow, its programming model, and our prototype in Sections 5.1 and 5.2. Section 5.3 presents and evaluates an example SSD App, Section 5.4 places our work in the context of other approaches to integrating programmability into storage devices, and Section 5.5 summarizes this chapter.

5.1 System design

Willow revisits the interface that the storage device exposes to the rest of the system, and provides the hardware necessary to support that interface efficiently. This section describes the system from the programmer's perspective, paying particular attention to the programming model and hardware/software interface. Section 5.2 describes the prototype hardware in more detail.

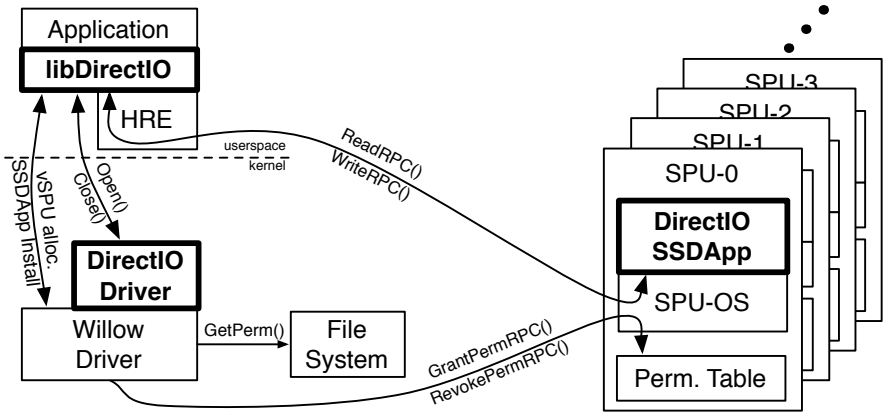


Figure 5.2: **The anatomy of an SSD App** The boldface elements depict three components of an SSD App: a userspace library, the SPU code, and an optional kernel driver. In the typical use case, a conventional file system manages the contents of Willow, and the Willow driver grants access to file extents based on file system permissions.

5.1.1 Willow system components

Figure 5.1(a) depicts a conventional storage system with a high-end, PCIe-attached SSD. A host system connects to the SSD via NVM Express (NVMe) [71] over PCIe, and the operating system sends commands and receives responses over that communication channel. The commands are all storage-specific (e.g., read or write a block) and there is a point-to-point connection between the host operating system and the storage device. Modern, high-end SSDs contain several (often many) embedded, programmable processors, but that programmability is not visible to the host system or to applications.

Figure 5.1(b) shows the corresponding picture of the Willow SSD. Willow's components resemble those in a conventional SSD: it contains several *storage processor units (SPUs)*, each of which includes a microprocessor, an interface to the inter-SPU interconnect, and access to an array of non-volatile memory. Each SPU runs a very small operating system called SPU-OS that manages and enforces security (see Section 5.1.6 below).

The interface that Willow provides is very different from the interface of a

conventional SSD. On the host side, the Willow driver creates and manages a set of objects called *Host RPC Endpoints (HREs)* that allow the OS and applications to communicate with SPUs. The HRE is a data structure that the kernel creates and allocates to a process. It provides a unique identifier called the *HRE ID* for sending and receiving RPC requests and lets the process send and receive those requests via DMA transfers between userspace memory and the Willow SSD. The SPUs and HREs communicate over a flexible network using a simple, flexible RPC-based mechanism. The RPC mechanism is generic and does not provide any storage-specific functionality. SPUs can send RPCs to HREs and vice versa.

The final component of Willow is programmable functionality in the form of SSD Apps. Each SSD App consists of three elements: a set of RPC handlers that the Willow kernel driver installs at each SPU on behalf of the application, a library that an application uses to access the SSD App, and a kernel module, if the SSD App requires kernel support. Multiple SSD Apps can be active at the same time.

Below, we describe the high-level system model, the programming model, and the security model for both SPUs and HREs.

5.1.2 The Willow usage model

Willow's design can support many different usage models (e.g., a system could use it as a tightly-coupled network of "wimpy" compute nodes with associated storage). Here, however, we focus on using Willow as a conventional storage device that also provides programmability features. This model is particularly useful because it allows for incremental adoption of Willow's features and ensures that legacy applications can use Willow without modification.

In this model, Willow runs an SSD App called `Base-IO` that provides basic block device functionality (i.e., reading and writing data from and to storage locations).

Base-IO stripes data across the SPUs (and their associated banks of non-volatile memory) in 8 kB segments. Base-IO (and all the other SSD Apps we present in this paper) runs identical code at each SPU. We have found it useful to organize data and computation in this way, but Willow does not require it.

A conventional file system manages the space on Willow and sets permissions that govern access to the data it holds. The file system uses the Base-IO block device interface to maintain metadata and provide data access to applications that do not use Willow's programmability.

To exploit Willow's programmability, an application needs to install and use an additional SSD App. Figure 5.2 illustrates this process for an SSD App called Direct-IO that provides an OS-bypass interface that avoids system call and file system overheads for common-case reads and writes (similar to [21]). The figure shows the software components that comprise Direct-IO in bold. To use Direct-IO, the application uses the Direct-IO's userspace library, libDirectIO. The library asks the operating system to install Direct-IO in Willow and requests an HRE from the Willow driver to allow it to communicate with the Willow SSD.

Direct-IO also includes a kernel module that libDirectIO invokes when it needs to open a file on behalf of the application. The Direct-IO kernel module asks the Willow driver to grant the application permission to access the file. The driver requests the necessary permission information from the file system and issues trusted RPCs to SPU-OS to install the permission for the file extents the application needs to access in the SPU-OS permission table. Modern file systems already include the ability to query permissions from inside the kernel, so no changes to the file system are necessary.

Base-IO and Direct-IO are "standard equipment" on Willow, since they provide functions that are useful for many other SSD Apps. In particular, other SSD Apps can leverage Direct-IO's functionality to implement arbitrary, untrusted operations on file

data.

5.1.3 Building an SSD App

SSD Apps comprise interacting components running in multiple locations: in the client application (e.g., `libDirectIO`), in the host-side kernel (e.g., the `Direct-IO` kernel module), and in the Willow SSD. To minimize complexity, code in all three locations uses a common set of interfaces to implement SSD App functionality. In the host application and the kernel, the HRE library implements these interfaces, while in Willow, SPU-OS implements them. The interfaces provide the following capabilities:

1. **Send an RPC request:** SPUs and HREs can issue RPC requests to SPUs, and SPUs can issue RPCs to HREs. RPC delivery is non-reliable (due to limited buffering at the receiver), and all-or-nothing (i.e., the recipient will not receive a partial message). The sender is notified upon successful (or failed) delivery of the message. Willow supports both synchronous and asynchronous RPCs.
2. **Receive an RPC request:** RPC requests carry an *RPC ID* that specifies which SSD App they target and which handler they should invoke. When an RPC request arrives at an SPU or HRE, the runtime (i.e., the HRE library or SPU-OS) invokes the correct handler for the request.
3. **Send an RPC response:** RPC responses are short, fixed-length messages that include a result code and information about the request it responds to. RPC response delivery is reliable.
4. **Initiate a data transfer:** An RPC handler can asynchronously transfer data between the network interface, local memory, and the local non-volatile memory (for SPUs only).
5. **Allocate local memory:** SSD Apps can declare static variables to allocate space in the SPU's local data memory, but they cannot allocate SPU memory dynamically.

Code on the host can allocate data statically or on the heap.

6. **General purpose computation:** SSD Apps are written in C, although the standard libraries are not available on the SPUs.

In addition to these interfaces, the host-side HRE library also provides facilities to request HREs from the Willow driver and install SSD Apps.

This set of interfaces has proved sufficient to implement a wide range of different applications (see Section 5.3), and we have found them flexible and easy to use. However, as we gain more experience building SSD Apps, we expect that opportunities for optimization, new capabilities, and bug-preventing restrictions on SSD Apps will become apparent.

5.1.4 The SPU architecture

In modern SSDs (and in our prototype), the embedded processor that runs the SSD's firmware offers only modest performance and limited local memory capacity compared to the bandwidth that non-volatile memory and the SSD's internal interconnect can deliver.

In addition, concerns about power consumption (which argue for lower clock speeds) and cost (which argue for simple processors) suggest this situation will persist, especially as memory bandwidths continue to grow. These constraints shape both the Willow hardware we propose and the details of the RPC mechanism we provide.

The SPU has four hardware components we use to implement the SSD App toolkit (Figure 5.1(c)):

1. **SPU processor:** The processor provides modest performance (perhaps 100s of MIPS) and kilobytes of per-SPU instruction and data memory.
2. **Local non-volatile memory:** The array of non-volatile memory can read or write data at over 1 GB/s.

```
void Read_Handler (RPCHdr_t *request_hdr) { // RPCHdr_t part of the
    RPC interface
    // Parse the incoming RPC
    BaseIOCmd_t cmd;
    RPCReceiveBytes(&cmd, sizeof(BaseIOCmd_t)); // DMA the IO
        command header
    RPCResp_t response_hdr;                     // Allocate response
    RPCCreateResponse(request_hdr,              // populate the
        response
              &response_hdr,
              RPC_SUCCESS);
    RPCSendResponse(response_hdr);    // Send the response

    // Send the read data back via a second RPC
    CPUID_t dst = request_hdr->src;
    RPCStartRequest(dst,              // Destination PU
              sizeof(IOCmd_t) + cmd.length, // Request body length
        READ_COMPLETE_HANDLER);       // Read completion RPC ID
    RPCAppendRequest(LOCAL_MEMORY_PORT, // Source DMA port
        sizeof(BaseIOCmd_t),         // IO command header size
        &cmd);                       // IO command header address
    RPCAppendRequest(NV_MEMORY_PORT, // Source DMA Port
              cmd.length,                 // Bytes to read
              cmd.addr);                  // Read address
    RPCFinishRequest();              // Complete the request
}
```

Figure 5.3: **READ() implementation for `Base-IO`** Handling a READ() requires parsing the header on the RPC request and then sending requested data from non-volatile memory back to host via another RPC.

3. **Network interface:** The network provides gigabytes-per-second of bandwidth to match the bandwidth of the local non-volatile memory array and the link bandwidth to the host system.
4. **Programmable DMA controller:** The DMA controller routes data between non-volatile memory, the network port, and the processor's local data memory. It can handle the full bandwidth of the network and local non-volatile memory.

The DMA controller is central to the design of both the SPU and the RPC mechanism, since it allows the modestly powerful processor to handle high-bandwidth streams of data. We describe the RPC interface in the following section.

The SPU runs a simple operating system (SPU-OS) that provides simple multi-threading, works with the Willow host-side driver to manage SPU memory resources, implements protection mechanisms that allow multiple SSD Apps to be active at once, and enforces the file system's protection policy for non-volatile storage. Section 5.1.6 describes the protection facilities in more detail.

5.1.5 The RPC interface

The RPC mechanism's design reflects the constraints of the hardware described above. Given the modest performance of the SPU processor and its limited local memory, buffering entire RPC messages at the SPU processor is not practical. Instead, the RPC library parses and assembles RPC requests in stages. The code in Figure 5.3 illustrates how this works for a simplified version of the READ() RPC from `Base-IO`.

When an RPC arrives, SPU-OS copies the RPC header into a local buffer using DMA and passes the buffer to the appropriate handler (`Read_Handler`). That handler uses the DMA controller to transfer the RPC parameters into the SPU processor's local memory (`RPCReceiveBytes`). The header contains generic information (e.g., the source of the RPC request and its size), while the parameters include command-specific values (e.g., the read or write address). The handler uses one or more DMA requests to process the remainder of the request. This can include moving part of the request to the processor's local memory for examination or performing bulk transfers between the network port and the non-volatile memory bank (e.g., to implement a write). In the example, no additional DMA transfers are needed.

The handler sends a fixed-sized response to the RPC request (`RPCCreateResponse` and `RPCSendResponse`). Willow guarantees the reliable delivery of fixed-size responses (acks or nacks) by guaranteeing space to receive them when the RPC is sent. If the SSD App needs to send a response that is longer than 32 bits (e.g., to return the data for a read),

it must issue an RPC to the sender. If there is insufficient buffer space at the receiver, the inter-SPU communication network can drop packets. In practice, however, dropped packets are exceedingly rare.

The process of issuing an RPC to return the data follows a similar sequence of steps. The SPU gives the network port the destination and length of the message (`RPCStartRequest`). Then it prepares any headers in local memory and uses the DMA controller to transfer them to the network interface (`RPCAppendRequest`). Further DMA requests can transfer data from non-volatile memory or processor memory to the network interface to complete the request. In this case, the SSD App transfers the read data from the non-volatile memory. Finally, it makes a call to signal the end of the message (`RPCFinishRequest`).

5.1.6 Protection and sharing in Willow

Willow has several features that make it easy for users to build and deploy useful SSD Apps: Willow supports untrusted SSD Apps, protects against malicious SSD Apps (assuming the host-side kernel is not compromised), allows multiple SSD Apps to be active simultaneously, and allows one SSD App to leverage functionality that another provides. Together these four features allow a user to build and use an SSD App without the permission of a system administrator and to focus on the functionality specific to his or her particular application.

Providing these features requires a suite of four protection mechanisms. First, it must be clear which host-side process is responsible for the execution of code at the SPU, so SPU-OS can enforce the correct set of protection policies. Second, the SPU must allow an SSD App to access data stored in Willow only if the process that initiated the current RPC has access rights to that data. Third, the SPU must restrict an SSD App to accessing only its own memory and executing only its own code. Finally, it must

allow some control transfers between SSD Apps so the user can compose SSD Apps. We address each of these below.

Tracking responsibility: The host system is responsible for setting protection policy for Willow, and it does so by associating permissions with operating system processes. To correctly enforce the operating system's policies, SPU-OS must be able to determine which process is responsible for the RPC handler that is currently running.

To facilitate this, Willow tracks the *originating HRE* for each RPC. An HRE is the originating HRE for any RPCs it makes and for any RPCs that an SPU makes as a result of that RPC and any subsequent RPCs. The PCIe interface hardware in the Willow SSD sets the originating HRE for the initial RPC, and SPU hardware and SPU-OS propagate it within the SSD. As a result, the originating HRE ID is unforgeable and serves as a capability [60].

To reduce cache coherence traffic, it is useful to give each thread in a process its own HRE. The Willow driver allocates HREs so that the high-order bits of the HRE ID are the same for every HRE belonging to a single process.

Non-volatile storage protection: To limit access to data in the non-volatile memory banks, SPU-OS maintains a set of permissions for each process at each SPU. Every time the SSD App uses the DMA controller to move data to or from non-volatile memory, SPU-OS checks that the permissions for the originating HRE (and therefore the originating process) allow it. The worst-case permission check latency is 2 μs.

The host-side kernel driver installs extent-based permission entries on behalf of a process by issuing privileged RPCs to SPU-OS. The SPU stores the permissions for each process as a splay tree to minimize permission check time. Since the SPU-OS permission table is fixed size, it may evict permissions if space runs short. If a request needs an evicted permission entry, a "permission miss" occurs, and the DMA transfer will fail. In response, SPU-OS issues an RPC to the kernel. The kernel forwards the request to

the SSD App's kernel module (if it has one), and that kernel module is responsible for resolving the miss. Most of our SSD Apps use the `Direct-IO` kernel module to manage permissions, and it will re-install the permission entry as needed.

Code and Data Protection: To limit access to the code and data in the SPU processor's local memory, the SPU processor provides segment registers and disallows access outside the current segment. Each SSD App has its own data and instruction segments that define the base address and length of the instruction and data memory regions it may access. Accesses outside the SSD App's segment raise an exception and cause SPU-OS to notify the kernel via an RPC, and the kernel, in turn, notifies the applications that the SSD App is no longer available. SPU-OS provides a trusted RPC dispatch mechanism for incoming messages. This mechanism sets the segment registers according to the SSD App that the RPC targets.

The host-side kernel is in charge of managing and statically allocating SPU instruction and data memory to the active SSD Apps. Overlays could extend the effective instruction and data memory size (and are common in commercial SSD controller firmware), but we have not implemented them in our prototype.

Limiting access to RPCs: A combination of hardware and software restricts access to some RPCs. This allows safe composition of SSD Apps and allows SSD Apps to create RPCs that can be issued only from the host-side kernel.

To support composition, SPU-OS provides a mechanism for changing segments as part of a function call from one SSD App to another. An *SSD App-intercall table* in each SPU controls which SSD Apps are allowed to invoke one another and which function calls are allowed. A similar mechanism restricts which RPCs one SSD App can issue to another.

To implement kernel-only RPCs, we use the convention that a zero in the high-order bit of the HRE ID means the HRE belongs to the kernel. RPC implementations can

check the ID and return failure when a non-kernel HRE invokes a protected RPC.

SSD Apps can use this mechanism to bootstrap more complex protection schemes as needed. For example, they could require the SSD App's kernel module to grant access to userspace HREs via a kernel-only RPC.

5.2 The Willow prototype

We have constructed a prototype Willow SSD that implements all of the functionality described in the previous section. This section provides details about the design.

The prototype has eight SPUs and a total storage capacity of 64 GB. It is implemented using a BEE3 FPGA-based prototyping system [16]. The BEE3 connects to a host system over a PCIe 1.1x8. The link provides 2 GB/s of full-duplex bandwidth.

Each of the four FPGAs that make up a BEE3 hosts two SPUs, each attached to an 8 GB bank of DDR2 DRAM. We use the DRAM combined with a customized memory controller to emulate phase change memory with a read latency of 48 ns and a write latency of 150 ns. The memory controller implements start-gap wear-leveling [79].

The SPU processor is a 125 MHz RISC processor with a MIPS-like instruction set. It executes nearly one instruction per cycle, on average. We use the MIPS version of gcc to generate executable code for it. For debugging, it provides a virtual serial port and a rich set of performance counters and status registers to the host. The processor has 32 kB of local data memory and 32 kB of local instruction memory.

The kernel driver statically allocates space in the SPU memory to SSD Apps, which constrains the number and size of SSD Apps that can run at once. SPU-OS maintains a permission table in the local data memory that can hold 768 entries and occupies 20 kB of data memory.

The ring in Willow uses round-robin, token-based arbitration, so only one SPU

may be sending a message at any time. To send a message, the SPU's network interface waits for the token to arrive, takes possession of it, and transmits its data. To receive a message, the interface watches the header of messages on the ring to identify messages it should remove from the ring. The ring is 128 bits wide and runs at 250 MHz for a total of 3.7 GB/s of bisection bandwidth.

For communication with the HREs on the host, a bridge connects the ring to the PCIe link. The bridge serves as a hardware proxy for the HREs. For each of the HREs, the bridge maintains an upstream (host-bound) and downstream (Willow-bound) queue. This queue-based interface is similar to the scheme that NVMExpress [71] uses to issue and complete IO requests. The bridge in our prototype Willow supports up to 1024 queue pairs, so it can support 1024 HREs on the host.

The bridge also helps enforce security in Willow. Messages from HREs to SPUs travel over the bridge, and the bridge sets the originating HRE fields on those messages depending on which HRE queue they came in on. Since processes can send messages only via the queues for the HREs they control, processes cannot send forged RPC requests.

5.3 Case study: AtomicWrites

Willow makes it easy for storage system engineers to improve performance by incorporating new capabilities into a storage device. In this thesis, we use Willow to implement support for transactions and and compare its performance to implementation that uses a conventional storage interface.

Many storage applications (e.g., file systems and databases) use write-ahead logging (WAL) to enforce strict consistency guarantees on persistent data structures. WAL schemes range from relatively simple journaling mechanisms for file system metadata to the complex ARIES scheme for implementing scalable transactions in databases [68].

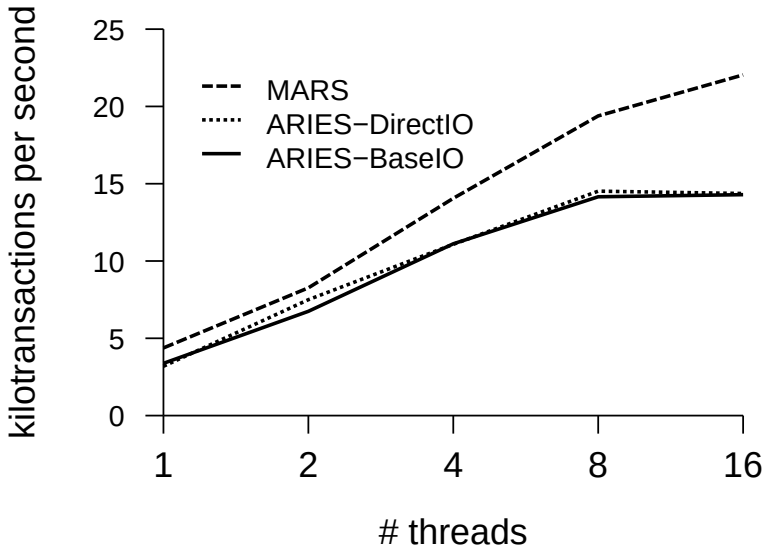


Figure 5.4: **TPC-B throughput** MARS using `Atomic-Writes` yields up to 1.5× throughput gain compared to ARIES using `Base-IO` and `Direct-IO`

Recently, researchers and industry have developed several SSDs with built-in support for multi-part atomic writes [76, 77], including a scheme called MARS [28] that aims to replace ARIES in databases.

MARS relies on a WAL primitive called editable atomic writes (EAW). EAW provides the application with detailed control over where logging information resides inside the SSD and allows it to edit log records prior to committing the atomic operations.

We have implemented EAWs as an SSD App called `Atomic-Writes` that implements four RPCs—LOGWRITE(), COMMIT(), LOGWRITECOMMIT(), and ABORT(), as summarized in Table 5.1. `Atomic-Writes` makes use of the `Direct-IO` functionality as well.

The implementations of LOGWRITE() and COMMIT() illustrate the flexible programmability of Willow's RPC interface. Each SPU maintains the redo-log as a complex persistent data structure for each active transaction. An array of log metadata entries resides in a reserved area of non-volatile memory with each entry pointing to a log record, the data to be written, and the location where it should be written. LOGWRITE() appends

Table 5.1: **RPCs for `Atomic-Writes`** The `Atomic-Write` SSD App allows applications to combine multiple writes into a single atomic operation and commit or abort them.

RPC	Description
LOGWRITE()	Start a new atomic operation and/or add a write to an existing atomic operation.
COMMIT()	Commit an atomic operation.
LOGWRITECOMMIT()	Create and commit an atomic operation comprised of single write.
ABORT()	Abort an atomic operation.

an entry to this array and initializes it to add the new entry to the log.

COMMIT() uses a two-phase commit protocol among the SPUs to achieve atomicity. The host library tracks which SPUs are participating in the transaction and selects one of them as the coordinator. In Phase 1, the coordinator broadcasts a "prepare" request to all the SPUs participating in this transaction (including itself). Each participant decides whether to commit or abort and reports back to the coordinator. In Phase 2, if any participant decides to abort, the coordinator instructs all participants to abort. Otherwise the coordinator broadcasts a "commit" request so that each participant plays its local portion of the log and notifies the coordinator when it finishes.

We have modified the Shore-MT [93] storage manager to use MARS and EAW to implement transaction processing. We also fine-tuned EAWs to match how Shore-MT manages transactions, something that would not be possible in the "black box," one-size-fits-all implementation of EAWs that a non-programmable SSD might include. Figure 5.4 shows the performance difference between MARS and ARIES for TPC-B [96]. MARS scales better than ARIES when increasing thread count and outperforms ARIES by up to 1.5×. These gains are ultimately due to the rich semantics that `Atomic-Writes` provides.

5.4 Related work

Many projects (and some commercial products) have integrated compute capabilities into storage devices, but most of them focus on offloading bulk computation to an active hard drive or (more recently) an SSD.

In the 1970s and 1980s, many advocates of specialized database machines pressed for custom hardware, including processor-per-track or processor-per-head hard disks to achieve processing at storage device. None of these approaches turned out to be successful due to high design complexity and manufacturing cost.

Several systems, including CASSM [94], RAP [87], and RARES [63] provided a processor for each disk track. However, the extra logic required to enable processing ability on each track limited storage density, drove up costs and prevented processor-per-track from finding wide use.

Processor-per-head techniques followed, with the goal of reducing costs by associating processing logic with each read/write head of a moving head hard disk. The Ohio State Data Base Computer (DBC)[50] and SURE [57] each took this approach. These systems demonstrated good performance for simple search tasks, but could not handle more complex computation such as joins or aggregation.

Two different projects, each named Active Disks, continued the trend toward fewer processors, providing just one CPU per disk. The first [82] focused on multimedia, database, and other scan-based operations, and their analysis mainly addressed performance considerations. The second [15] provided a more complete system architecture but supported only stream-based computations called disklets.

Several systems [87, 29] targeted (or have been applied to) databases with programmable in-storage processing resources and some integrated FPGAs [69, 70]. IDisk [51] focused on decision support databases and considered several different soft-

ware organizations, ranging from running a full-fledged database on each disk to just executing data-intensive kernels (e.g., scans and joins). Willow resembles the more general-purpose programming models for IDisks.

Recently researchers have extended these ideas to SSDs [33, 52], and several groups have proposed offloading bulk computation to SSDs. The work in [48] implements Map-Reduce [30]-style computations in an SSD, and two groups [20, 95] have proposed offloading data analysis for HPC applications to the SSD's processor. Samsung is shipping an SSD with a key-value interface.

Projects that place general computation power into other hardware components, such as programmable NICs, have also been proposed [34, 102, 65]. These devices allow for application-specific code to be placed within the NIC in order to offload network-related computation. This in turn reduces the load of the host OS and CPU in a similar manner to Willow.

Most of these projects focus on bulk computation, and we see that as a reasonable use case for Willow as well, although it would require a faster processor. However, Willow goes beyond bulk processing to include modifying the semantics of the device and allowing programmers to implement complex, control-intensive operations in the SSD itself. Some programmable NICs have taken this approach. Many projects [28, 76, 77, 18, 86, 44, 106, 21, 25] have shown that moving these operations to the SSD is valuable, and making the SSD programmable will open up many new opportunities for performance improvement for both application and operating system code.

5.5 Summary

Solid state storage technologies offer dramatic increases in flexibility compared to conventional disk-based storage, and the interface that we use to communicate with

storage needs to be equally flexible. Willow offers programmers the ability to implement customized SSD features to support particular applications. The programming interface is simple and general enough to enable a wide range of SSD Apps that can improve performance on a wide range of applications.

Acknowledgments

This chapter contains material from "Willow: A User-Programmable SSD", by Sudharsan Seshadri, MarkGahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson, which appears in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, (OSDI 2014). The dissertation author was the sixth investigator and author of this paper.

Chapter 6

Conclusion

Flash-based SSDs have risen to prominence over the past decade, but their adherence to conventional block-based I/O interface poses several challenges in performance, lifetime and reliability. Existing proposals to modernize the SSD interface by providing key-value semantics present a single inflexible key space, while proposals for transactional SSDs require inefficient coarse-grained locking. KAML solves these problems by supporting multiple, independent namespaces for key-value pairs and allow applications to tune the performance of each namespace to meet application requirements. It also enables fine-grained locking by treating key-value pairs as the basic unit of transactional operations. Finally, KAML provides a caching layer analogous to a conventional page cache to improve performance. These changes allow KAML to outperform conventional designs in a wide range of workloads.

By providing BAM in the internal NVRAM, PebbleSSD can support a range of useful features to improve device lifetime, including write-optimized file block mapping and fast, efficient data movement. PebbleSSD exposes two new commands, `fs_write` and `remap` allowing file systems to access BAM. Log-structured file systems can use `fs_write` command to avoid recursive updates on file index blocks, and use `remap`

command to perform fast and efficient movement of valid data during log cleaning. In both cases, log-structured file systems achieve better performance while reducing flash writes. Therefore BAM is effective in improving SSDs' lifetime.

Besides the one-at-a-time approach adopted by KAML and PebbleSSD, Willow takes advantage of in-storage resources to offer dramatic increases in flexibility thanks to its programmability as a central feature. Willow provides programmers with the ability to implement customized SSD features to support particular applications. The programming interface is simple and general enough to enable a wide range of SSD Apps that can improve performance on a wide range of applications.

Bibliography

[1] BtrFS official site. https://btrfs.wiki.kernel.org/index.php/Main_Page.

[2] Ext3 file system. http://lxr.free-electrons.com/source/fs/ext3/ext3.h?v=3.7.

[3] Filebench. https://github.com/filebench/filebench/wiki.

[4] LinkBench. https://github.com/facebookarchive/linkbench.

[5] MySQL. https://www.mysql.com/.

[6] NILFS2 clean utility. http://nilfs.sourceforge.net/en/man8/nilfs-clean.8.html.

[7] NILFS2 official site. http://nilfs.sourceforge.net/en/.

[8] *OpenSSD*. http://www.openssd-project.org/wiki/The_OpenSSD_Project.

[9] PCI Express. http://pcisig.com/specifications/pciexpress/.

[10] Percona-Lab implementation of TPC-C. https://github.com/Percona-Lab/tpcc-mysql.

[11] SAS, Serial-Attached SCSI. https://www.ibm.com/support/knowledgecenter/POWER6/arebj/sasoverview.htm#.

[12] SATA. http://serialata.org/.

[13] Toshiba. http://www.toshiba.com/tai/.

[14] 3D XPoint. https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/.

[15] A. Acharya, M. Uysal, and J. Saltz. Active Disks: Programming Model, Algorithms And Evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 81–91, New York, NY, USA, 1998. ACM.

[16] BEE3. http://www.beecube.com/platform.html.

[17] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[18] M. S. Bhaskaran, J. Xu, and S. Swanson. BankShot: Caching Slow Storage in Fast Non-Volatile Memory. In *First Workshop on Interactions of NVM/Flash with Operating Systems and Workloads*, INFLOW '13, 2013.

[19] A. Bityutskiy. JFFS3 design issues. www.linux-mtd.infradead.org/tech/JFFS3design.pdf, 2005.

[20] S. Boboila, Y. Kim, S. Vazhkudai, P. Desnoyers, and G. Shipman. Active Flash: Out-of-core Data Analytics on Flash Storage. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–12, 2012.

[21] 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 *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, March 2012. ACM.

[22] F. Chen, T. Luo, and X. Zhang. CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory Based Solid State Drives. In *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011*, pages 77–90, 2011.

[23] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency without Ordering. In *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012*, page 9, 2012.

[24] H. J. Choi, S.-H. Lim, and K. H. Park. JFTL: A Flash Translation Layer Based on a Journal Remapping for Flash Memory. *Trans. Storage*, 4(4):14:1–14:22, Feb. 2009.

[25] S. Chu. Memcachedb. http://memcachedb.org/.

[26] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. System Software for Flash Memory: A Survey. In *Embedded and Ubiquitous Computing, International Conference, EUC 2006, Seoul, Korea, August 1-4, 2006, Proceedings*, pages 394–404. 2006.

[27] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. A survey of flash translation layer. *Journal of Systems Architecture*, 55(5):332–343, 2009.

[28] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson. From ARIES to MARS: Transaction Support for Next-generation, Solid-state Drives. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 197–212, 2013.

[29] G. P. Copeland, Jr., G. J. Lipovski, and S. Y. Su. The architecture of CASSM: A cellular system for non-numeric processing. In *Proceedings of the First Annual Symposium on Computer Architecture*, ISCA '73, pages 121–128, New York, NY, USA, 1973. ACM.

[30] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI '04: Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[31] B. Debnath, S. Sengupta, and J. Li. FlashStore: High Throughput Persistent Key-value Store. *PVLDB*, 3(2):1414–1425, 2010.

[32] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM Space Skimpy Key-value Store on Flash-based Storage. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 25–36, 2011.

[33] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1221–1230, New York, NY, USA, 2013. ACM.

[34] M. E. Fiuczynski, R. P. Martin, T. Owa, and B. N. Bershad. SPINE: A Safe Programmable and Integrated Network Environment. In *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, EW 8, pages 7–12, New York, NY, USA, 1998. ACM.

[35] E. Gal and S. Toledo. Algorithms And Data Structures for Flash Memories. *ACM Comput. Surv.*, 37(2):138–163, 2005.

[36] G. Graefe. Write-optimized b-trees. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 672–683. VLDB Endowment, 2004.

[37] J. Gray. The Transaction Concept: Virtues and Limitations (Invited Paper). In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, pages 144–154. VLDB Endowment, 1981.

[38] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing Flash Memory: Anomalies, Observations, And Applications. In *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, 2009, New York, New York, USA*, pages 24–33, 2009.

[39] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, pages 229–240, 2009.

[40] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam. Leveraging Value Locality in Optimizing NAND Flash-based SSDs. In *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011*, pages 91–103, 2011.

[41] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch Hashing. In *International Symposium on Distributed Computing*, pages 350–364. Springer, 2008.

[42] InnoDB. http://dev.mysql.com/doc/refman/5.7/en/innodb-storage-engine.html.

[43] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: A Scalable Approach to Logging. *Proceedings of the VLDB Endowment*, 3(1-2):681–692, 2010.

[44] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn. DFS: A File System for Virtualized Flash Storage. *Transactions on Storage*, 6(3):14:1–14:25, Sept. 2010.

[45] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho. The Multi-Streamed Solid-State Drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage '14, Philadelphia, PA, USA, June 17-18, 2014.*, 2014.

[46] W.-H. Kang, S.-W. Lee, B. Moon, Y.-S. Kee, and M. Oh. Durable Write Cache in Flash Memory SSD for Relational and NoSQL Databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 529–540, New York, NY, USA, 2014. ACM.

[47] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, and C. Min. X-FTL: Transactional FTL for SQLite Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 97–108, 2013.

[48] Y. Kang, Y. Kee, E. Miller, and C. Park. Enabling Cost-effective Data Processing with Smart SSD. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–12, 2013.

[49] Y. Kang, R. Pitchumani, T. Marlette, and E. L. Miller. Muninn: A Versioning Flash Key-Value Store Using an Object-based Storage Model. In *International Conference on Systems and Storage, SYSTOR 2014, Haifa, Israel, June 30 - July 02, 2014*, pages 13:1–13:11, 2014.

[50] K. Kannan. The Design of A Mass Memory for A Database Computer. In *Proceedings of the 5th Annual Symposium on Computer Architecture*, ISCA '78, pages 44–51, New York, NY, USA, 1978. ACM.

[51] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A Case for Intelligent Disks (IDISKs). *SIGMOD Record*, 27(3):42–52, Sept. 1998.

[52] S. Kim, H. Oh, C. Park, S. Cho, and S. Lee. Fast, Energy Efficient Scan inside Flash Memory. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2011, Seattle, WA, USA, September 2, 2011.*, pages 36–43, 2011.

[53] C. Lee, D. Sim, J. Hwang, and S. Cho. F2FS: A New File System for Flash Storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, Santa Clara, CA, Feb. 2015. USENIX Association.

[54] E. Lee, J. Kim, H. Bahn, and S. H. Noh. Reducing Write Amplification of Flash Storage through Cooperative Data Management with NVM. In *32nd Symposium on Mass Storage Systems and Technologies, MSST 2016, Santa Clara, CA, USA, May 2-6, 2016*, pages 1–6, 2016.

[55] S. Lee, M. Liu, S. Jun, S. Xu, J. Kim, and Arvind. Application-Managed Flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 339–353, Santa Clara, CA, Feb. 2016. USENIX Association.

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

[57] H.-O. Leilich, G. Stiege, and H. C. Zeidler. A Search Processor for Data Base Management Systems. In *Proceedings of the Fourth International Conference on Very Large Data Bases - Volume 4*, VLDB '78, pages 280–287. VLDB Endowment, 1978.

[58] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-Tree for New Hardware Platforms. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 302–313, 2013.

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

[60] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[61] J. Li, H.-W. Tseng, C. Lin, Y. Papakonstantinou, and S. Swanson. Hippogriffdb: Balancing I/O And GPU Bandwidth in Big Data Analytics. *PVLDB*, 9(14):1647–1658, 2016.

[62] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 1–13, 2011.

[63] C. S. Lin, D. C. P. Smith, and J. M. Smith. The Design of A Rotating Associative Memory for Relational Database Applications. *ACM Transactions on Database Systems*, 1(1):53–65, Mar. 1976.

[64] Y. Lu, J. Shu, and W. Zheng. Extending the Lifetime of Flash-based Storage Through Reducing Write Amplification from File Systems. In *Proceedings of the 11th USENIX conference on File and Storage Technologies, FAST 2013, San Jose, CA, USA, February 12-15, 2013*, pages 257–270, 2013.

[65] A. Maccabe, W. Zhu, J. Otto, and R. Riesen. Experience in Offloading Protocol Processing to A Programmable NIC. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 67–74, 2002.

[66] P. Macko, M. I. Seltzer, and K. A. Smith. Tracking Back References in a Write-Anywhere File System. In *8th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 23-26, 2010*, pages 15–28, 2010.

[67] L. Mármol, S. Sundararaman, N. Talagala, and R. Rangaswami. NVMKV: A Scalable, Lightweight, FTL-aware Key-value Store. In *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 207–219, 2015.

[68] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Transactions Database Systems*, 17(1):94–162, Mar. 1992.

[69] R. Mueller and J. Teubner. FPGA: What's in It for A Database? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 999–1004, New York, NY, USA, 2009. ACM.

[70] R. Mueller, J. Teubner, and G. Alonso. Data Processing on FPGAs. *Proc. VLDB Endow.*, 2(1):910–921, Aug. 2009.

[71] NVM Express. http://www.nvmexpress.org/.

[72] G. Oh, C. Seo, R. Mayuram, Y.-S. Kee, and S.-W. Lee. SHARE Interface in Flash Storage for Relational and NoSQL Databases. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 343–354. ACM, 2016.

[73] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-Structured Merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[74] Oracle Database. https://www.oracle.com/database/index.html.

[75] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. SDF: Software-defined Flash for Web-scale Internet Storage Systems. In *ACM SIGPLAN Notices*, volume 49, pages 471–484, 2014.

[76] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. Panda. Beyond Block I/O: Rethinking Traditional Storage Primitives. In *17th International Conference on High-Performance Computer Architecture (HPCA-17 2011), February 12-16 2011, San Antonio, Texas, USA*, pages 301–311, 2011.

[77] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional Flash. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 147–160, 2008.

[78] S. Qiu and A. L. N. Reddy. NVMFS: A Hybrid File System for Improving Random Write in Nand-flash SSD. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5, May 2013.

[79] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing Lifetime And Security of PCM-based Main Memory with Start-gap Wear Leveling. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–23, New York, NY, USA, 2009. ACM.

[80] M. Raab and A. Steger. Balls into Bins – A Simple and Tight Analysis. In *Randomization and Approximation Techniques in Computer Science*, pages 159–170. Springer, 1998.

[81] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell. Pilot: An Operating System for a Personal Computer. *Commun. ACM*, 23(2):81–92, Feb. 1980.

[82] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active Disks for Large-Scale Data Processing. *Computer*, 34(6):68–74, June 2001.

[83] E. Riedel, G. A. Gibson, and C. Faloutsos. Active Storage for Large-Scale Data Mining and Multimedia. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 62–73, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

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

[85] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming Aggressive Replication in the Pangaea Wide-area File System. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*, 2002.

[86] M. Saxena, M. M. Swift, and Y. Zhang. FlashTier: A Lightweight, Consistent And Durable Storage Cache. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 267–280, New York, NY, USA, 2012. ACM.

[87] S. Schuster, H. B. Nguyen, E. Ozkarahan, and K. Smith. RAP.2: An Associative Processor for Databases And Its Applications. *Computers, IEEE Transactions on*, C-28(6):446–458, 1979.

[88] Seagate Kinetic. http://www.seagate.com/solutions/cloud/data-center-cloud/platforms/.

[89] Seagate Nytro XP6500. http://www.tomsitpro.com/articles/seagate-nytro-xp6500-enterprise-ssd,2-1017.html.

[90] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. Willow: A User-programmable SSD. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 67–80, 2014.

[91] W. Shi, D. Wang, Z. Wang, and D. Ju. Möbius: A High Performance Transactional SSD with Rich Primitives. In *IEEE 30th Symposium on Mass Storage Systems and Technologies, MSST 2014, Santa Clara, CA, USA, June 2-6, 2014*, pages 1–11, 2014.

[92] Shore-kits. https://bitbucket.org/shoremt/shore-kits/src.

[93] Shore-MT. http://research.cs.wisc.edu/shore-mt/.

[94] S. Y. W. Su and G. J. Lipovski. CASSM: A Cellular System for Very Large Data Bases. In *Proceedings of the 1st International Conference on Very Large Data Bases*, VLDB '75, pages 456–472, New York, NY, USA, 1975. ACM.

[95] D. Tiwari, S. S. Vazhkudai, Y. Kim, X. Ma, S. Boboila, and P. J. Desnoyers. Reducing Data Movement Costs Using Energy Efficient, Active Computation on SSD. In *Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems*, HotPower '12, pages 4–4, Berkeley, CA, USA, 2012. USENIX Association.

[96] TPC-B. http://www.tpc.org/tpcb/.

[97] TPC-C. http://www.tpc.org/tpcc/.

[98] VSL. http://www.fusionio.com/products/vsl.

[99] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong. An Efficient Design and Implementation of LSM-tree Based Key-value Store on Open-channel SSD. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, pages 16:1–16:14, 2014.

[100] T. Wang, D. Liu, Y. Wang, and Z. Shao. FTL2: A Hybrid Flash Translation Layer with Logging for Write Reduction in Flash Memory. In *SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2013, LCTES '13, Seattle, WA, USA, June 20-21, 2013*, pages 91–100, 2013.

[101] Z. Weiss, S. Subramanian, S. Sundararaman, N. Talagala, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. ANViL: Advanced Virtualization for Modern Non-Volatile Memory Devices. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 111–118, Santa Clara, CA, Feb. 2015. USENIX Association.

[102] P. Willmann, H. Kim, S. Rixner, and V. Pai. An Efficient Programmable 10 Gigabit Ethernet Network Interface Card. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 96–107, Feb 2005.

[103] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman. Don't Stack Your Log On My Log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*, Broomfield, CO, Oct. 2014. USENIX Association.

[104] Yahoo Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB.

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

[106] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. De-indirection for Flash-based SSDs with Nameless Writes. In *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012*, page 1, 2012.