OPTIMIZING FLASH-BASED KEY-VALUE DATABASE ENGINE FOR BIG DATA AND MOBILE APPLICATIONS

ZHAOYAN SHEN

PhD

The Hong Kong Polytechnic University

2018

THE HONG KONG POLYTECHNIC UNIVERSITY

DEPARTMENT OF COMPUTING

Optimizing Flash-based Key-value Database Engine for Big Data and Mobile Applications

Zhaoyan SHEN

A Thesis Submitted in Partial Fulfillment of

the Requirements for the Degree of

Doctor of Philosophy

May 2018

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

_____(Signature)

Zhaoyan Shen (Name of Student)

ABSTRACT

The key-value database engine, which offers higher efficiency, scalability, availability, and usually works with simple NoSQL schema, is becoming more and more popular. It has been widely adopted as the caching system in today's low-latency Internet services, such as Memcached, Redis, McDipper, and Fatcache. However, these conventional key-value cache systems are either heavily reliant on expensive DRAM memory or utilize commercial solid state drives (SSDs) in an inefficient way. In addition, although the key-value database engine has simple interfaces and has been proven to be more profitable than the traditional relational SQL databases in cloud environments, it has seldom been adopted by mobile applications. The reason for this is that most applications running on mobile devices depend on the SQL interface to access databases, which the key-value database engine does not provide. In this thesis, we address these issues from several aspects including the integration of the emerging hardware open-channel SSD, the cross-layer hardware/software management, and the design of an SQLite-to-KV compiler for mobile applications.

First, we focus on optimizing the key-value caching performance through a deep integration of flash hardware devices and key-value software management. To lower the Total Cost of Ownership (TCO), the industry has recently been moving toward more cost-efficient flash-based solutions, such as Facebook's McDipper and Twitter's Fatcache. These cache systems typically take commercial SSDs and adopt a Memcached-like scheme to store and manage key-value cache data in flash. Such a practice, although simple, is inefficient because of the huge *semantic gap* between the key-value cache manager and the underlying flash devices. In this thesis, we advocate reconsidering the design of the cache system and directly opening device-level details of the underlying flash storage for key-value caching. We propose an enhanced flash-aware key-value cache manager, consisting of a novel unified address

mapping module, an integrated garbage collection policy, a dynamic over-provisioning space management, and a customized wear-leveling policy, to directly drive the flash management. A thin intermediate library layer provides a slab-based abstraction of low-level flash memory space and an API interface for directly and easily operating flash devices. A special flash memory SSD hardware that exposes flash physical details is adopted to store key-value items. This codesign approach bridges the semantic gap and well connects the two layers, allowing us to leverage both the domain knowledge of the key-value caches and the unique properties of the device. In this way, we can maximize the efficiency of key-value caching on flash devices while minimizing the weakness. We implemented a prototype, called DIDA-Cache, based on the open-channel SSD platform. Our experiments on real hardware show that we can significantly increase the throughput by 35.5%, reduce the latency by 23.6%, and decrease unnecessary erase operations by 28%.

Second, we propose a new programming storage interface for SSDs to provide flexible support for key-value caching. Solid-state drives (SSDs) are widely deployed in computer systems of numerous types and purposes, in two main usage modes. In the first mode, the SSD firmware hides the details of the hardware from the application and exports the standard, backward-compatible block I/O interface. This ease of use comes at the cost of low resource utilization, due to the semantic gap between application and hardware. In the second mode, the SSD directly exposes the low-level details of the hardware to developers, who leverage them for fine-grained application-specific optimizations. However, the improved performance significantly increases the complexity of the software and also the cost of developing it. Thus, application developers must choose between *easy development* and *optimal performance*, without a real possibility of being able to balance the two. To address this limitation, we propose *Prism-SSD*—a flexible storage interface for SSDs. Via a user-level library, Prism-SSD exports the SSD hardware in three levels of abstraction: as a raw flash medium with its low-level details, as a group of functions to manage flash capacity, and simply as a configurable block device. This multi-level abstraction allows developers to

choose the degree to which they want to control the flash hardware so that it best suits the semantics and performance objectives of their applications. To demonstrate the usability and performance of this new model and interface, we implemented a user-level library on the open-channel SSD platform to the prototype Prism-SSD. We implemented three versions of the key-value caching system by using each of the library's three levels of abstraction, and compared their performances and development overhead.

Third, we study the problem of making mobile applications benefit the efficient key-value database engine. SQLite has been deployed in millions of mobile devices from web to smartphone applications on various mobile operating systems. However, due to the uncoordinated nature of the IO interactions with the underlying file system (e.g., ext4), SQLite is not efficient, with a low number of transactions per second. In this thesis, we for the first time propose a new SQLite-like database engine, called SQLiteKV, which adopts the LSM-tree-based data structure but retains the SQLite operation interfaces. With its SQLite interface, SQLiteKV can be utilized by existing applications without any modifications, while providing high performance with its LSM-tree-based data structure. We separate SQLiteKV into front-end and back-end sections. In the front-end, we develop a light-weight SQLite-to-KV compiler to solve the semantic mismatch, so that SQL statements can be efficiently translated into key-value operations. We also design a novel coordination caching mechanism with memory fragmentation so that query results can be effectively cached inside SQLiteKV by alleviating the discrepancy in data management between front-end SQLite statements and back-end data organization. In the back-end, we adopt an LSM-tree-based key-value database engine, and propose a lightweight metadata management scheme to mitigate the memory requirement. We implemented and deployed SQLiteKV on a Google Nexus 6P smartphone. The results of experiments with various workloads show that SQLiteKV outperforms SQLite by up to 6 times.

Keywords: Key-value database, open-channel SSD, NAND flash memory, software/hardware codesign, Mobile device, SQL/NoSQL interface.

PUBLICATIONS

Journal Papers

1. **Zhaoyan Shen**, Feng Chen, Yichen Jia, Zili Shao, "DIDACache: An Integration of Device and Application for Flash-based Key-value Caching", Accepted in *ACM Transactions on Storage (TOS)*, 2018.

2. **Zhaoyan Shen**, Zhijian He, Shuai Li, QiXin Wang, Zili Shao, "A Multi-Quadcopter Cooperative Cyber-Physical System for Timely Air Pollution Localization", *IEEE Transactions on Embedded Computer System (TECS)*, 16:3:70, 2017.

3. Lei Han, **Zhaoyan Shen**, Duo Liu, Zili Shao, H. Howie Huang, Tao Li, "A Novel ReRAM-based Processing-in-Memory Architecture for Graph Traversal", Accepted in *IEEE Transactions on Storage (TOS)*, 2018.

4. Renhai Chen, **Zhaoyan Shen**, Chenlin Ma, Zili Shao, Yong Guan, "NVMRA: Utilizing NVM to Improve the Random Write Operations for NAND-Flash-Based Mobile Devices", Software: Practice and Experience (SPE), 2015.

5. **Zhaoyan Shen**, Yuanjing Shi, Zili Shao, Yong Guan, "An Efficient LSM-tree-based SQLite-like Database Engine for Mobile Devices", Minor revision in *ACM Transactions on Computer-Aided Design of Integrated Circuites and Systems (TCAD)*, 2018.

6. Chenlin Ma, **Zhaoyan Shen**, Yi Wang, Zili Shao, "Alleviating Hot Data Write Back Effect for Shingled Magnetic Recording Storage Systems", Major revision in *ACM Transactions on Computer-Aided Design of Integrated Circuites and Systems (TCAD)*, 2018.

Conference Papers

1. Lei Han, **Zhaoyan Shen**, Zili Shao, Tao Li, "Optimizing RAID/SSD Controllers with Lifetime Extension for Flash-based SSD Array", in *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (LCTES '18), Philadelphia, Pennsylvania, United States, June 18-22, 2018.

2. Yuanjing Shi, **Zhaoyan Shen**, and Zili Shao, "SQLiteKV: an efficient LSM-tree-based SQLite-like database engine for mobile devices," in *Proceedings of the 22nd Asia and South Pacific Design Automation Conference* (ASP-DAC '18), Jeju Island , Korea, Jan.22-25, 2018.

3. Lei Han, **Zhaoyan Shen**, Zili Shao, H. Howie Huang, and Tao Li, "A novel ReRAM-based processing-in-memory architecture for graph computing," in *2017 IEEE 6th Non-Volatile Memory Systems and Applications Symposium* (NVMSA '17), Hsinchu, Taiwan, Aug.16-18, 2017.

4. **Zhaoyan Shen**, Feng Chen, Yichen Jia, and Zili Shao, "DIDACache: a deep integration of device and application for flash based key-value caching," in *15th USENIX Conference on File and Storage Technologies* (FAST '17), SANTA Clara/CA, USA, Feb.27-Mar.2, 2017.

5. **Zhaoyan Shen**, Feng Chen, Yichen Jia, and Zili Shao, "Optimizing flash-based key-value cache systems," in *8th USENIX Workshop on Hot Topics in Storage and File Systems* (HotStorage '17), Denver/CO, USA, June.20-21, 2016.

6. Zhijian He, Shuai Li, **Zhaoyan Shen**, Muhammad Umer Khan, Qixing Wang, Zili Shao, "A quadcopter swarm for active monitoring of smog propagation," Poster in *Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems*, (ICCPS '15), Seattle, USA, April 14-16, 2015.

7. Zhijian He, Yanming Chen, **Zhaoyan Shen***, Enyan Huang, Shuai Li, Zili Shao, and Qixin Wang, “Ard-mu-Copter: A Simple Open Source Quadcopter Platform,” in *11th International Conference on Mobile Ad-hoc and Sensor Networks* (MSN ’15), Shenzhen, China, Dec 16-18, 2015.

ACKNOWLEDGEMENTS

First and foremost, I want to express my gratitude to my supervisor, Prof. Zili Shao, whose expertise, understanding, and patience, added considerably to my graduate experience. I appreciate his vast knowledge and skill in many areas and his professional supervision. It is my great pleasure to be a student of Prof. Shao, and I want to thank him for supporting me over the years, and for giving me so much freedom to explore and discover new areas of research. Without his help and support, this body of work would not have been possible.

I must acknowledge Dr. Feng Chen at Louisiana State University, Baton Rouge, LA, for his patient guidance, stimulating discussions, insightful comments, and encouragement. His true scientist's intuition and invaluable guidance contributed to my intellectual maturity, and I will benefit from, for a long time to come. I offer my regards and blessings to Dr. Chen for the support that he gave to me in different aspects of my research during my Ph.D. study.

I want to thank Dr. Gala Yadgar at Technion, Israel, for her comprehensive guidance during our discussions over my research and the revision of my paper. I have learned a great deal from her about the professional skill of writing academic papers. Her attitude of rigor towards scientific research will definitely benefit my research career.

I want to thank Dr. Shuai Li from the Hong Kong Polytechnic University, for his guidance, encouragement and advice. I also express my gratitude to the other members of Prof. Shao's research group - Dr. Yi Wang, Dr. Duo Liu, Dr. Zhiwei Qin, Dr. Renhai Chen, Chenlin Ma, Lei Han, Yuanjing Shi, Fang Wang, and Luguang Wang—for the assistance that they provided during my Ph.D. study. I also would like to thank all of my teachers from whom I learned so much during my long journey of acquiring a formal education.

I want to thank Prof. Lou Wei from the Hong Kong Polytechnic University for kindly

serving as the Chairman of the Board of Examiners (BoE). I also thank Prof. Yi Pan from Georgia State University, and Prof. Wei Zhang from the Hong Kong University of Science and Technology, for kindly taking the time from their busy schedules to serve as my external examiners.

I recognize that this thesis would not have been possible without the financial assistance that I received from the Hong Kong Polytechnic University. I thank Prof. Shao and the Department of Computing for offering me travel grants to attend several international conferences. I acknowledge the grant for the Research Student Attachment Program from the Hong Kong Polytechnic University.

Finally, I want to thank my family. They educated and guided me, and have watched over me every step of way. I want to thank them for their endless love, support, and encouragement that they gave me throughout my entire life, for letting me pursue my dream for so long and so far away from home, and for giving me the motivation to finish this thesis. Special thanks are due to my wife, Xiaofeng, who witnessed the joys and sorrows of my PhD study from miles away. This is the sixth year of our long-distance relationship, and we got married last year. I am truly grateful for her endless love, patience, understanding and support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

Various key-value data stores, such as Cassandra [15], Hbase [44], LevelDB [36], Memcached [86], McDipper [32], and Fatcache [123], have been widely deployed for data management to support internet services. Compared with the traditional relational database management systems (RDBMS), the key-value stores offer higher efficiency, scalability, and availability. The append-only log data structure, which transfers random writes to sequential writes, is a common data structure that has been adopted by the key-value data stores. The log-structured key-value data stores are initially optimized for hard disk drive (HDD) storage systems. In recent years, with the development of NAND flash technology, flash-based solid state drives (SSDs) are increasingly being adopted to replace the HDDs in the key-value data stores. However, due the device specificity of flash-based SSDs (e.g., out-of-place updates, limited programming cycles), it would not be efficient to simply replace HDDs with commercial SSDs in the key-value data stores. A huge semantic gap would be incurred and it would not be possible to exploit the full performance potentials of both the key-value software and the underlying flash hardware. In addition, although the key-value data stores have proven to be much more profitable than the RDBMS in big data environments, they cannot be directly employed by most mobile applications. Nowadays, most mobile applications are built based on the traditional SQL interface. However, the key-value data stores only support simple interfaces (such as `Set()`, `Get()`, and `Delete()`). Redesigning these mobile applications to support key-value interfaces would lead to too much development overhead. Thus, the interface mismatch has become the bottleneck keeping mobile applications from benefiting from the efficient key-value data stores.

In this thesis, we address the semantic gaps between the key-value store software and the underlying flash-based hardware and bridge the interface mismatch between mo-

bile applications and the key-value data stores. Specially, we employ an emerging hardware, open-channel SSD, to improve the performance of key-value data stores (especially key-value caching systems) in data centers. The open-channel SSD exposes its device-level details and raw flash operations directly to applications. The host is responsible for utilizing SSD resources with primitive functions through a simplified I/O stack. This means that the applications have the flexibility to schedule flash operations according to their own software semantics. Meanwhile, open-channel SSDs also brings much development overhead to developers of applications. Balancing the key-value data store performance and the development overhead becomes a big challenge. In addition, to address the interface mismatch issue, we propose a new SQLite-like database engine, which includes a light-weight SQLite-to-KV compiler to translate SQL statements into key-value operations.

We first propose a software/hardware codesign approach to bridge the huge semantic gap between the key-value cache manager and the underlying flash devices by employing open-channel SSDs. We advocate reconsidering the cache system design and directly opening device-level details of the underlying flash storage for key-value caching. By reconsidering the division between software and hardware, a variety of new optimization opportunities can become true: (1) A single, unified mapping structure can directly map the keys to physical flash pages storing the values, which completely removes the redundant mapping table and saves a large amount of on-device memory; (2) An integrated Garbage Collection (GC) procedure, which is directly driven by the cache system, can optimize the decision of when and how to recycle *semantically invalid* storage space at a fine granularity, which removes the high overhead caused by unnecessary and uncoordinated GCs at both layers; (3) An on-line scheme can determine an optimal size for Over-Provisioning Space (OPS) and dynamically adapt to the characteristics of the workload, which will maximize the usable flash space and greatly increase the cost efficiency of using expensive flash devices; (4) A wear-leveling policy that cooperates with GC to evenly wear out underlying flash blocks. We have implemented a fully functional prototype, called DIDACache, based on a PCI-E open-channel SSD hardware to demonstrate the effectiveness of this new design scheme.

Second, to balance the performance and development overhead, we propose a flexi-

ble storage interface for the underlying hardware SSDs to support the various needs of the key-value stores. We propose to redefine the programming model for open-channel SSDs, and develop an abstraction library between applications and the hardware to satisfy the different needs of applications. Based on the management granularity, the library abstracts flash memory operation interfaces into three levels: (1) A raw-level flash abstraction, which directly exposes all flash operations, such as physical page read/write, and block erase; (2) A function-level flash abstraction, which exposes flash management function interfaces, mainly includes address translator, garbage collector, and wear-lever, to applications; and (3) A user-level configurable FTL abstraction, which can be configured with different flash management policies and exposes block read/write and configure interfaces to applications. The proposed programming model allows applications to integrate the hardware management using different levels of abstraction by balancing their semantic redundancies with flash SSDs and the development overheads. We evaluated the powerfulness of this programming model with the use case key-value caching system implemented with its three abstractions.

Third, for mobile applications, we propose a new SQLite-like database engine, called SQLiteKV, which adopts the LSM-tree-based data structure but retains the SQLite operation interfaces. With its SQLite interface, SQLiteKV can be utilized by existing applications without any modifications, while providing high performance with its LSM-tree-based data structure. We separate SQLiteKV into a front-end and a back-end. For the front-end, we develop a light-weight SQLite-to-KV compiler to solve the semantic mismatch, so that SQL statements can be efficiently translated into key-value operations. We also design a novel coordination caching mechanism with memory defragmentation so query results can be effectively cached inside SQLiteKV by alleviating the discrepancy in data management between front-end SQLite statements and back-end data organization. In the back-end, we adopt an LSM-tree-based key-value database engine, and propose a lightweight metadata management scheme to mitigate the memory requirement. We implemented and deployed SQLiteKV on a Google Nexus 6P smartphone.

The rest of this chapter is organized as follows. Section 1.1 presents the related work. Section 1.2 discusses the unified research framework. Section 1.3 summarizes the

contributions of this thesis. Finally, Section 1.4 gives an outline of the thesis.

1.1 Related Work

In this section, we briefly discuss previous approaches to optimizing flash-based key-value stores. In previous studies, work has been done in three main domains: (I) Key-value databases, (II) Open-channel SSD integration, and (III) SQL-compatible key-value databases. We briefly describe these techniques, and present detailed comparisons with representative techniques in the respective chapters.

1.1.1 Key-Value Databases

Key-value stores [11, 14–16, 21, 26, 32, 36, 75, 123] are becoming widespread solutions for handling large-scale data in cloud center applications. Compared with traditional RDBMSs, key-value stores outperform in terms of simplicity, scalability, and high throughput. Key-value store systems are available for applications that are used as back-end databases to persistent data (e.g., LevelDB [36], RocksDB [4], and Cassandra [15]) and that are used as caching systems to buffer frequently accessed data (e.g., Memcached [86], Redis [103], and Fatcache [123]).

LevelDB [36], which is a typical key-value database, runs on a single node. By wrapping the client-server support around the LevelDB, LevelDB can be employed as the underlying single-node component in a distributed environment, such as Tair [5], Riak [3], and HyperDex [31]. In order to meet the high throughput and low latency demands of applications, several recent works explore the key-value store on the flash-based storage. FlashStore [25] presents a high throughput persistent key-value store, which uses flash memory as a non-volatile cache between RAM and the hard disk, to store the working set of key-value pairs on flash and using one flash read per key lookup. SkimpyStash [26] proposes a RAM space skimpy key-value store on flash-based storage for high throughput server applications. SkimpyStash requires an extremely low RAM footprint of about 1 byte per key-value

pair. SILT [76] further reduces the DRAM footprint to 0.7 bytes per entry, and retrieves key-value pairs using an average of 1.01 flash reads each. Key-value cache systems have recently shown their practical importance in Internet services [11,37,79,130]. A report from Facebook discusses the company's efforts to scale Memcached to handle the huge amount of Internet I/O traffic that they encounter [92]. McDipper [32] is their latest effort in the area of flash-based key-value caching. Several prior research studies specifically focus on the issue of optimizing key-value store/cache for flash. Ouyang et al. propose an SSD-assisted hybrid memory for Memcached in high performance networks [97]. This solution essentially takes flash as a swapping device. Flashield [29] is also a hybrid key-value cache that uses DRAM as a "filter" to minimize writes to flash. Hot slab pages are retained in memory, while cold slab pages are swapped out to flash SSDs. NVMKV [82] gives an optimized key-value store based on flash devices with several new designs, such as dynamic mapping, transactional support, and parallelization. Unlike NVMKV, our thesis proposes a key-value cache, which allows us to aggressively integrate the two layers together and exploit some unique opportunities. For example, we can invalidate all slots and erase an entire flash block, since we are dealing with a cache rather than with storage.

1.1.2 Open-Channel SSD Integration

Open-channel SSDs are a new class of SSDs that open up a large design space for SSD management. With open-channel SSDs, the internal channels and flash chips are exposed to the host. The host is responsible for utilizing SSD resources with primitive functions through a simplified I/O stack. From an abstract view of the software layer, the open-channel SSDs exhibit three key features. (1) Open-channel SSD exposes the internal parallelism of SSD to user applications. User applications can directly access individual flash channels, and can effectively organize their data and schedule their data accesses to fully utilize the raw flash performance. (2) The erase operation is exposed to software as a new interface. Erase is an expensive operation compared to read and write. Erase operations triggered by a GC process in conventional SSD can cause unpredictable service time fluctuation. With

the new erase interface, the software is responsible for conducting and scheduling erase operations before a block can be overwritten. (3) Open-channel SSD provides a simplified I/O stack. Applications can directly operate the device hardware through the Linux builds a complicated I/O stack which highly degrades the high-end SSD's performance. For the sake of efficiency, the open-channel SSD bypasses most of the I/O layers in the kernel and uses the *ioctl* interface to directly communicate with the hardware driver.

Recent research has proposed exposing internal flash layout details directly to the application. SDF [96] exposes the channels in commodity SSD hardware to software and causes the software to interact with the devices in a manner that is friendlier towards their performance characteristics to realize the raw bandwidth and storage capacity of the hardware. LOCS [125] integrates SDF with LSM-tree-based key-value stores to optimize scheduling and dispatching decisions according to data access patterns from LSM-tree-based key-value stores. ParaFS [131] exposes physical information about the device to the file system in order to exploit the internal parallelism, and coordinates GC processes in the FS and FTL levels to keep GC overhead low. KAML [49] presents a key-addressable, multi-log SSD with a key-value interface. KAML maps flash internal channels to its multiple logs and directly maps key-value items to physical flash pages, so as to improve system concurrency. FlashBlox [46] proposes utilizing flash parallelism to improve isolation between applications by running them on dedicated channels and dies, and balancing wear within and across different applications. AMF [71] moves the intelligence of the flash management from the device to applications by providing a new out-of-place block I/O interface to reduce flash management overhead and improve the performance of the applications. LightNVM [13] is an open-channel SSD subsystem in the Linux kernel, which introduces a new physical page address I/O interface that exposes SSD parallelism and storage media characteristics.

There are also some other works that propose leveraging the computing capability of SSDs. Kang [55] introduces a Smart SSD model that pairs in-device processing with a powerful host system capable of handling data-oriented tasks without modifying operating system codes. ActiveFlash [122] offloads data analysis tasks for HPC applications to SSD controller without degrading the performance of the simulation job. Willow [109] offers

programmers the ability to implement customized an SSD features to support particular applications. Programmers can augment and extend the semantics of an SSD with application-specific features without compromising file system protections.

Different from prior work, in our work we aim to provide a flexible programming model for open-channel SSD, so that applications can integrate their semantic logics with the flash management with different levels of abstraction. With our programming model, the application can benefit from the flash memory performance with minimum development overhead.

1.1.3 SQL-Compatible Key-Value Database

Key-value databases have simple interfaces (such as `Put()` and `Get()`) and are more efficient than the traditional relational SQL databases in cloud environments [11, 16, 27]. To utilize the advantages of a key-value database engine under SQL environments, Apache Phoenix [99] provides an open source relational database, in which an SQL statement is compiled into a series of key-value operations for HBase [44], a distributed, key-value database. Phoenix provides well-defined and industry standard APIs for OLTP and operational analytics for Hadoop [30, 132]. Nevertheless, without deep integration with the Hadoop framework, it would be difficult for mobile devices to adopt either HBase as their storage engine or Phoenix for SQL-to-KV transitions. Also, Phoenix, along with other Hadoop-related modules, has been designed for scalable and distributed computing environments with large datasets [34], which means they can hardly fit in mobile environments with limited resources [115]. However, the approach cannot be directly adopted by resource-limited mobile devices as it is targeted at scalable and distributed computing environments with large datasets [20, 78].

In this thesis, we propose an efficient LSM-tree-based lightweight database engine, SQLiteKV, which retains the SQLite interface for mobile devices, provides better performance than SQLite and adopts an efficient LSM-tree structure for its storage engine.

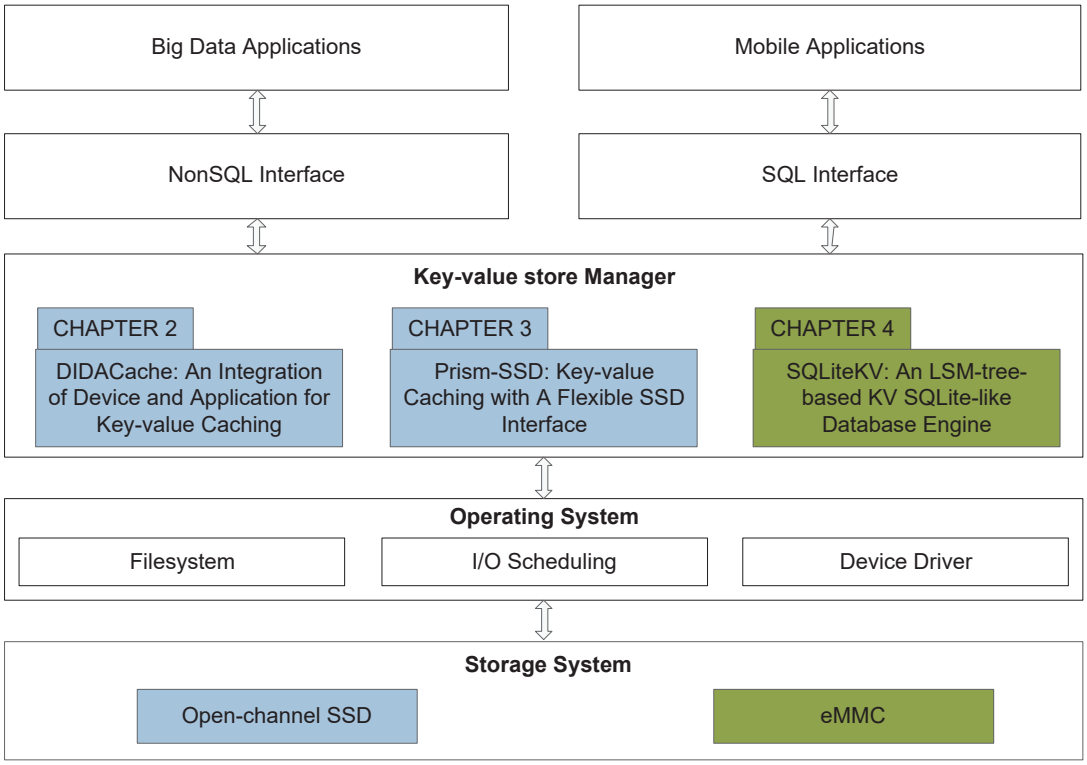


Figure 1.1: The Unified Research Framework.

1.2 The Unified Research Framework

In this section, we present the unified research framework for the proposed techniques. A sketch of our research framework is given in Figure 1.1.

In this thesis, the key-value store manager runs at the software layers, either as a caching system or as a persistent database. The flash-based hardware is used as the storage medium. As shown in Figure 1.1, for key-value caching systems adopted by big data applications, we propose to integrate the flash hardware management with the software development to improve the system performance and balance the development overhead. We further propose to make the mobile applications benefit from the efficient key-value database engine using an SQL-to-KV compiler.

For the first scheme, in Chapter 2, we advocate reconsidering the cache system design and directly opening device-level details of the underlying flash storage for key-value caching. Such a codesign effort not only enables us to remove the unnecessary intermediate

layers between the cache manager and the storage devices, but also allows us to leverage the precious domain knowledge of key-value cache systems. For the second scheme, in Chapter 3, we propose to export the SSD hardware in three levels of abstraction: as a raw flash medium with its low-level details, as a group of functions to manage flash capacity, and simply as a configurable block device. This multi-level abstraction allows developers to choose the degree to which they desire to control the flash hardware in a manner that best suits the semantics and performance objectives of their applications. For the third scheme, in Chapter 4, we propose a new SQLite-like database engine, called SQLiteKV, which adopts the LSM-tree-based data structure but retains the SQLite operation interfaces. With its SQLite interface, SQLiteKV can be utilized by existing applications without any modifications, while providing high performance with its LSM-tree-based data structure.

1.3 Contributions

The contributions of this thesis are summarized as follows.

- In order to bridge the semantic gap between the application (key-value cache) and the hardware (SSD), we propose to integrate the low-level hardware management with the software key-value caching system design. We have created a thin intermediate library layer, called `libssd`, which provides an easy-to-use programming interface to facilitate applications to access low-level device information and directly operate the underlying flash device, such as reading and writing a flash page, erasing a flash block, and so on. By using this library layer, the key-value cache manger integrates its software semantics and the characteristics of the hardware and significantly improves its performance.

- We propose a highly flexible system interface, designed as a user-level library, for developers to interact with flash-based SSDs in varying layers of abstraction. We present a fully functional prototype of Prism-SSD on the real open-channel hardware platform, which will be made available as an open-source project. We demonstrate

the efficacy of our approach in three use cases, with a range of development costs and performance benefits.

- We for the first time propose to improve the performance of SQLite by adopting the LSM-tree-based key-value database engine while retaining the SQLite interfaces for mobile devices. We design a slab-based coordination caching scheme to solve the semantic mismatch between the SQL interfaces and the key-value database engine, which also effectively improves the system performance. We have re-designed the index management policy for the LSM-tree-based key-value database engine.
- We implement prototypes with the proposed techniques. We conduct experiments and compare our proposed schemes with representative schemes. The experimental results prove the effectiveness of the proposed schemes.

1.4 Thesis Organization

The rest of this thesis is organized as follows.

- In Chapter 2, we handle the semantic gap between the software key-value cache manager and the underlying SSD hardware. We advocate opening the underlying details of flash SSDs for key-value cache systems so as to effectively exploit the great potential of flash storage while avoiding its weaknesses.
- In Chapter 3, to balance easy development and optimal performance, we propose a flexible storage interface for SSDs. We integrate the key-value caching system with the SSD hardware in three level details.
- In Chapter 4, to handle the interface semantic mismatch, we for the first time propose to improve the performance of SQLite by adopting the LSM-tree-based key-value database engine while retaining the SQLite interfaces for mobile devices.
- In Chapter 5, we present our conclusions and propose possible future directions of research arising from this work.

CHAPTER 2

DIDACACHE: A DEEP INTEGRATION OF DEVICE AND APPLICATION FOR FLASH-BASED KEY-VALUE CACHING

2.1 Introduction

High-speed key-value caches, such as Memcached [86] and Redis [103], are the "first line of defense" in today's low-latency Internet services. By caching the working set in memory, key-value cache systems can effectively remove time-consuming queries to the backend data store (e.g., MySQL or LevelDB). Though effective, the in-memory key-value caches heavily rely on large amount of expensive and power-hungry DRAM for high cache hit ratio [45]. As the workload size rapidly grows, an increasing concern with such memory-based cache systems is their cost and scalability [6]. A possible alternative is to directly replace DRAM with byte-addressable non-volatile memory (NVM), such as PCM [66, 77], however, these persistent memory devices are not yet available for large-scale deployment in commercial environment. Recently, a more cost-efficient alternative, *flash-based key-value caching*, has raised high interest in the industry [32, 123].

NAND flash memory provides a much larger capacity and lower cost than DRAM, which enables a low Total Cost of Ownership (TCO) for a large-scale deployment of key-value caches. Facebook, for example, deploys a Memcached-compatible key-value cache system based on flash memory, called McDipper [32]. It is reported that McDipper allows Facebook to reduce the number of deployed servers by as much as 90% while still delivering more than 90% "get responses" with sub-millisecond latencies [74]. Twitter also has a similar key-value cache system, called Fatcache [123].

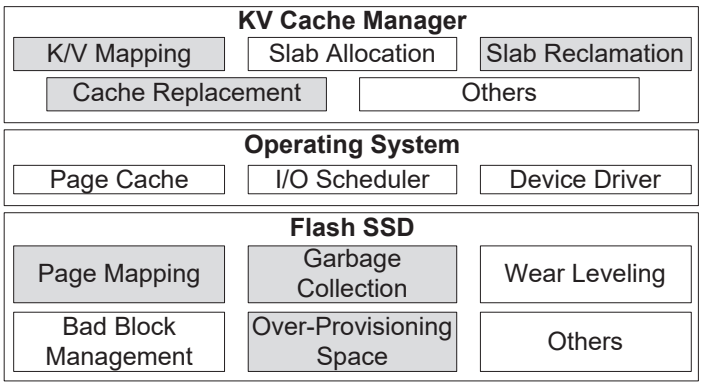


Figure 2.1: Architecture of flash-based key-value cache.

Typically, these flash-based key-value cache systems directly use commercial flash SSDs and adopt a Memcached-like scheme (NoSQL schema) to manage key-value cache data in flash. For example, key-values are organized into slabs of different size classes, and an in-memory hash table is used to maintain the key-to-value mapping. Such a design is simple and allows a quick deployment. However, it disregards an important fact – the key-value cache systems and the underlying flash devices both have very *unique properties*. Figure 2.1 shows a typical flash-based key-value cache architecture. The key-value cache manager that runs at the application level serves incoming requests and manages the cache space for allocation and replacement. The flash SSD at the device level manages flash chips and hides the unique characteristics of flash memory from applications. Simply treating flash SSDs as a faster storage and the key-value cache as a regular application not only fails to exploit various optimization opportunities but also raises several critical concerns: *Redundant mapping*, an application-level key-value-to-cache mapping and a device-level logical-to-physical flash space mapping; *Double garbage collection*, an application-level garbage collection process at the key-value item granularity to reclaim cache space and a device-level garbage collection process at the block granularity to reclaim flash space; and *Over-overprovisioning*, an application-level cache space reservation policy and a device-level over-provisioning space reservation. All these issues cause enormous inefficiencies in practice, which have motivated us to reconsider the software/hardware structure of the current flash-based key-value cache systems.

In this thesis, we will discuss the above-mentioned three key issues (Section 2.3) caused by the huge *semantic gap* between the key-value caches and the underlying flash devices, and further present a cohesive cross-layer design to fundamentally address these issues. Through our studies, we advocate to open the underlying details of flash SSDs for key-value cache systems. Such a co-design effort not only enables us to remove the unnecessary intermediate layers between the cache manager and the storage devices, but also allows us to leverage the precious domain knowledge of key-value cache systems, such as the unique access patterns and mapping structures, to effectively exploit the great potential of flash storage while avoiding its weakness.

By reconsidering the division between software and hardware, a variety of new optimization opportunities can be explored: (1) A single, unified mapping structure can directly map the "keys" to physical flash pages storing the "values", which completely removes the redundant mapping table and saves a large amount of on-device memory; (2) An integrated Garbage Collection (GC) procedure, which is directly driven by the cache system, can optimize the decision of when and how to recycle *semantically invalid* storage space at a fine granularity, which removes the high overhead caused by the unnecessary and uncoordinated GCs at both layers; (3) An on-line scheme can determine an optimal size of Over-Provisioning Space (OPS) and dynamically adapt to the workload characteristics, which will maximize the usable flash space and greatly increase the cost efficiency of using expensive flash devices; (4) A wear-leveling policy that cooperates with GC to evenly wear out underlying flash blocks.

We implement a fully functional prototype, called *DIDACache*, based on a PCI-E open-channel SSD hardware, and provide an performance analysis for both the conventional key-value cache system and our proposed DIDACache. A thin intermediate library layer, `libssd`, is created to provide a programming interface to facilitate applications to access low-level device information and directly operate the underlying flash device. Using the library layer, we developed a flash-aware key-value cache system based on Twitter's Fatcache [123], and carried out a series of experiments to demonstrate the effectiveness of our

new design scheme. Our experiments show that this approach can increase the throughput by 35.5%, reduce the latency by 23.6%, and remove erase operations by 28%.

The rest of chapter is organized as follows. Section 2.2 and Section 2.3 give background and motivation. Section 2.4 describes the design and implementation. Experimental results are presented in Section 2.5. Section 2.7 gives the related work. In Section 2.8 we summarize this chapter.

2.2 Background

This section briefly introduces three key technologies, flash memory, SSDs, and the current flash-based key-value cache systems.

• **Key-value Cache**. Key-value caching is the backbone of many systems in modern web-server architecture. A cache can be deployed anywhere in the infrastructure where there is congestion with data delivery. The two main cache models are *look-aside cache* and *inline cache*. The main difference of these two is that for inline cache, applications write new data or update the existing data in cache, which synchronously (write through) or asynchronously (write behind) write data to the backend data store. However, for look-aside cache, applications write new data to the backend data store, and then update the data in cache, if existing. In practice, key-value cache systems typically adopt the look-aside cache model, such as Memcached [86] and McDipper [32].

Figure 2.2 illustrates the basic workflow of a look-aside style key-value cache. In the example, the browser is the client, it sends requests to the application server, and the application server stores or accesses data from the key-value cache or the backend database. For writing a new data item, the application server directly stores the data to the backend database. For retrieving a data item, the application server first checks the key-value cache, if it is a cache hit, the data is returned from the cache without requesting the database; otherwise, the application server obtains data from the backend database and then writes it to the cache for future requests. For update operations, the application server updates

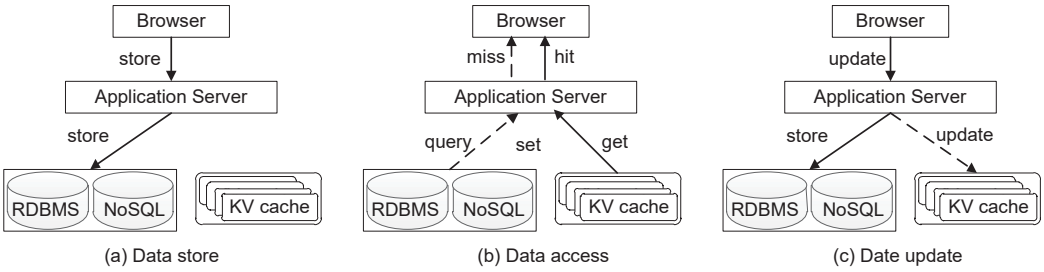


Figure 2.2: A look-aside key-value caching example.

existing data in both the key-value cache and the backend database. In this model, the data consistency is maintained by the application server.

• **Flash Memory**. NAND flash memory is a type of EEPROM device. A flash memory chip consists of two or more dies and each die has multiple *planes*. Each plane contains thousands of *blocks* (a.k.a. erase blocks). A block is further divided into hundreds of *pages*. Flash memory supports three main operations, namely `read`, `write`, and `erase`. Reads and writes are normally performed in units of pages. A read is typically fast (e.g., 50μs), while a write is relatively slow (e.g., 600μs). A constraint is that pages in a block must be written sequentially, and pages cannot be overwritten in place, meaning that once a page is programmed (written), it cannot be written again until the entire block is erased. An erase is typically slow (e.g., 5ms) and must be done in block granularity.

• **Flash SSDs**. A typical SSD includes four major components (Figure 2.3): A *host interface logic* connects the device to the host via an interface connection (e.g., SATA or PCI-E). An *SSD controller* is responsible for managing flash memory space, handling I/O requests, and issuing commands to flash memory chips via a *flash controller*. A dedicated *buffer* holds data or metadata, such as the mapping table. Most SSDs have multiple channels to connect the controller with flash memory chips, providing internal parallelism [19]. Multiple chips may share one channel. Actual implementations may vary in commercial products. More details about the SSD architecture can be found in prior work [9, 28]. A *Flash Translation Layer* (FTL) is implemented in SSD controller firmware to manage flash memory. and hide all the complexities behind a simple Logical Block Address (LBA) interface. which makes an SSD similar to to a disk drive. An FTL has three major roles: (1) *Logical block mapping*.

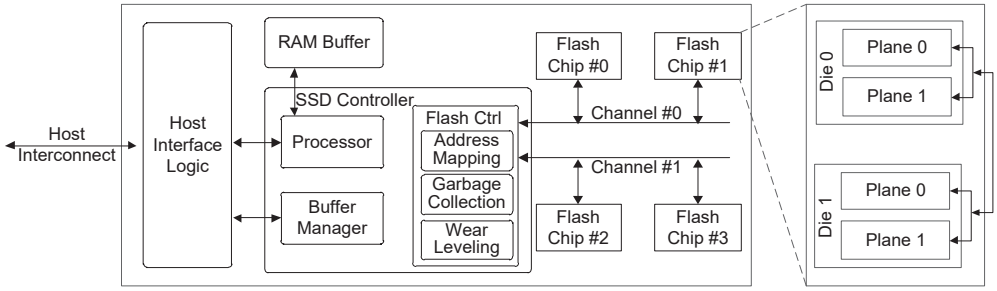


Figure 2.3: Illustration of SSD architecture [9].

An in-memory mapping table is maintained in the on-device buffer to map logical block addresses to physical flash pages dynamically. (2) *Garbage collection*. Due to the erase-before-write constraint, upon a write, the corresponding logical page is written to a new location, and the FTL simply marks the old page invalid. A GC procedure recycles obsolete pages later, which is similar to a Log-Structured File System [104]. (3) *Wear Leveling*. Since flash cells could wear out after a certain number of Program/Erase cycles, the FTL shuffles read-intensive blocks with write-intensive blocks to even out writes over flash memory. A previous work [35] provides a detailed survey of FTL algorithms.

• **Flash-based key-value caches**. In-memory key-value cache systems, such as Memcached, adopt a slab-based allocation scheme. Due to its efficiency, flash-based key-value cache systems, such as Fatcache [123], inherit a similar structure. Here we use Fatcache as an example; based on open documents [32], McDipper has a similar design. In Fatcache, the SSD space is first segmented into *slabs*. Each allocated slab is divided into *slots* (a.k.a. chunks) of equal size. Each slot stores a "value" item. According to the slot size, the slabs are categorized into different classes, from Class 1 to Class n, where the slot size increases exponentially. A newly incoming item is accepted into a class whose slot size is the best fit of the item size (i.e., the smallest slot that can accommodate the item). For quick access, a *hash mapping table* is maintained in memory to map the keys to the slabs containing the values. Querying a key-value pair (`GET`) is accomplished by searching the in-memory hash table and loading the corresponding slab block from flash into memory. Updating a key-value pair (`SET`) is realized by writing the updated value into a new location and updating the key-to-slab mapping in the hash table. Deleting a key-value pair (`DELETE`) simply removes the

mapping from the hash table. The deleted or obsolete value items are left for GC to reclaim later.

Despite the structural similarity to Memcached, flash-based key-value cache systems have several distinctions from their memory-based counterparts. First, the I/O granularity is much larger. For example, Memcached can update the value items individually. In contrast, Fatcache [123] has to maintain an in-memory slab to buffer small items in memory first and then flush to storage in bulk later, which causes a unique "large-I/O-only" pattern on the underlying flash SSDs. Second, unlike Memcached, which is byte addressable, flash-based key-value caches cannot update key-value items in place. In Fatcache, all key-value updates are written to new locations. Thus, a GC procedure is needed to clean/erase slab blocks. Third, the management granularity in flash-based key-value caches is much coarser. For example, Memcached maintains an object-level LRU list, while Fatcache uses a simple slab-level FIFO policy to evict the oldest slab when free space is needed.

2.3 Motivation

As shown in Figure 2.1, in a flash-based key-value cache, the *key-value cache manager* and the *flash SSD* run at the application and device levels, respectively. Both layers have complex internals, and the interaction between the two raises three critical issues, which have motivated the work presented in this thesis.

• **Problem 1: Redundant mapping**. Modern flash SSDs implement a complex FTL in firmware. Although a variety of mapping schemes, such as *block-level mapping* [40] and *page-level mapping* [41], exist, high-end SSDs often still adopt fine-grained *page-level mapping* for performance efficiency.As a result, for a 1TB SSD with a 4KB page size, a page-level mapping table could be as large as 1GB. Integrating such a large amount of DRAM on device not only raises production cost but also reliability concerns [41, 136, 137]. In the meantime, at the application level, the key-value cache system also manages another mapping structure, an in-memory hash table, which translates the keys to the corresponding slab blocks. The two mapping structures exist at two levels simultaneously, which unnecessarily

doubles the memory consumption.

A fundamental problem is that the page-level mapping is designed for general-purpose file systems, rather than key-value caching. In a typical key-value cache, the slab block size is rather large (in Megabytes), which is typically 100-1,000x larger than the flash page size. This means that the fine-grained page-level mapping scheme is an *expensive over-kill*. Moreover, a large mapping table also incurs other overheads, such as the need for a large capacitor or battery, increased design complexity, reliability risks, etc. If we could directly map the hashed keys to the physical flash pages, we can completely remove this redundant and highly inefficient mapping for lower cost, simpler design, and improved performance.

• **Problem 2: Double garbage collection**. GC is the main performance bottleneck of flash SSDs [9, 18]. In flash memory, the smallest read/write unit is a page (e.g., 4KB). A page cannot be overwritten in place until the entire erase block (e.g., 256 pages) is erased. Thus, upon a write, the FTL marks the obsolete page "invalid" and writes the data to another physical location. At a later time, a GC procedure is scheduled to recycle the invalidated space for maintaining a pool of clean erase blocks. Since valid pages in the to-be-cleaned erase block must be first copied out, cleaning an erase block often takes hundreds of milliseconds to complete. A key-value cache system has a similar GC procedure to recycle the slab space occupied by obsolete key-value pairs.

Running at different levels (application vs. device), these two GC processes not only are redundant but also could interfere with one another. For example, from the FTL's perspective, it is unaware of the semantic meaning of page content. Even if no key-value pair is valid (i.e., no key maps to any value item), the entire page is still considered as "valid" at the device level. During the FTL-level GC, this page has to be moved unnecessarily. Moreover, since the FTL-level GC has to assume all valid pages contain useful content, it cannot selectively recycle or even aggressively invalidate certain pages that contain semantically "unimportant" (e.g., LRU) key-value pairs. For example, even if a page contains only one valid key-value pair, the entire page still has to be considered valid and cannot be erased, although it is clearly of relatively low value. Note that `TRIM` command [119] cannot address this

issue as well. If we merge the two-level GCs and control the GC process based on semantic knowledge of the key-value caches, we could completely remove all the above-mentioned inefficient operations and create new optimization opportunities.

• **Problem 3: Over-overprovisioning**. In order to minimize the performance impact of GC on foreground I/Os, the FTL typically reserves a portion of flash memory, called Over-Provisioned Space (OPS), to maintain a pool of clean blocks ready for use. High-end SSDs often reserve 20-30% or even larger amount of flash space as OPS. From the user's perspective, the OPS space is nothing but an expensive unusable space. We should note that the factory setting for OPS is mostly based on a conservative estimation for worst-case scenarios, where the SSD needs to handle extremely intensive write traffic. In key-value cache systems, in contrast, the workloads are often read-intensive [11]. Reserving such a large portion of flash space is a significant waste of expensive resource. In the meantime, key-value cache systems possess rich knowledge about the I/O patterns and have the capability of accurately estimating the incoming write intensity. Based on such estimation, a suitable amount of OPS could be determined during runtime for maximizing the usable flash space for effective caching. Considering the importance of cache size for cache hit ratio, such a 20-30% extra space could significantly improve system performance. If we could leverage the domain knowledge of the key-value cache systems to determine the OPS management at the device level, we would be able to maximize the usable flash space for caching and greatly improve the overall cost efficiency as well as system performance.

In essence, all the above-mentioned issues stem from a fundamental problem in the current I/O stack design: the key-value cache manager runs at the application level and views the storage abstraction as a sequence of sectors; the flash memory manager (i.e., the FTL) runs at the device firmware layer and views incoming requests simply as a sequence of individual I/Os. This abstraction, unfortunately, creates a huge *semantic gap* between the key-value cache and the underlying flash storage. Since the only interface connecting the two layers is a strictly defined block-based interface, no semantic knowledge about the data could be passed over. This enforces the key-value cache manager and the flash memory manager to work individually and prevents any collaborative optimizations. This motivates us to study

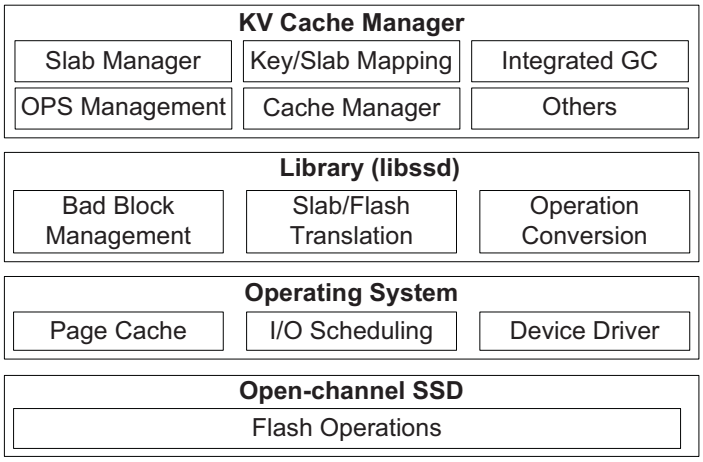


Figure 2.4: The architecture overview of DIDACache.

how to bridge this semantic gap and build a highly optimized flash-based key-value cache system.

2.4 Design

As an unconventional hardware/software architecture (see Figure 3.1), our key-value cache system is highly optimized for flash and eliminates all unnecessary intermediate layers. Its structure includes three layers.

- *An enhanced flash-aware key-value cache manager*, which is highly optimized for flash memory storage, runs at the application level, and directly drives the flash management;
- *A thin intermediate library layer*, which provides a slab-based abstraction of low-level flash memory space and an API interface for directly and easily operating flash devices (e.g., `read`, `write`, `erase`);
- *A specialized flash memory SSD hardware*, which exposes the physical details of flash memory medium and opens low-level *direct* access to the flash memory medium through the `ioctl` interface.

With such a holistic design, we strive to completely bypass multiple intermediate layers in the conventional structure, such as file system, generic block I/O, scheduler, and the FTL layer in SSD. Ultimately, we desire to let the application-level key-value cache manager leverage its domain knowledge and directly drive the underlying flash devices to operate only necessary functions while leaving out unnecessary ones. In this section, we will discuss each of the three layers.

2.4.1 Application Level: Key-value Cache

Our key-value cache manager has four major components: (1) a *slab management module*, which manages memory and flash space in slabs; (2) a *unified direct mapping module*, which records the mapping of key-value items to their physical locations; (3) an *integrated GC module*, which reclaims flash space occupied by obsolete key-values; and (4) an *OPS management module*, which dynamically adjusts the OPS size.

• **Slab Management**

Similar to Memcached, our key-value cache system adopts a slab-based space management scheme – the flash space is divided into equal-sized *slabs*; each slab is divided into an array of *slots* of equal size; each slot stores a key-value item; slabs are logically organized into different *slab classes* according to the slot size.

Despite these similarities to in-memory key-value caches, caching key-value pairs in flash has to deal with several unique properties of flash memory, such as the "out-of-place update" constraint. By directly controlling flash hardware, our slab management can be specifically optimized to handle these issues as follows.

• **Mapping slabs to blocks**: Our key-value cache directly maps (logical) slabs to physical flash blocks. We divide flash space into equal-sized slabs, and each slab is statically mapped to one or several flash blocks, as shown in Figure 2.5. There are two possible mapping schemes: (1) *Per-channel mapping*, which maps a slab to a sequence of contiguous physical flash blocks in one channel, and (2) *Cross-channel mapping*, which maps a slab across

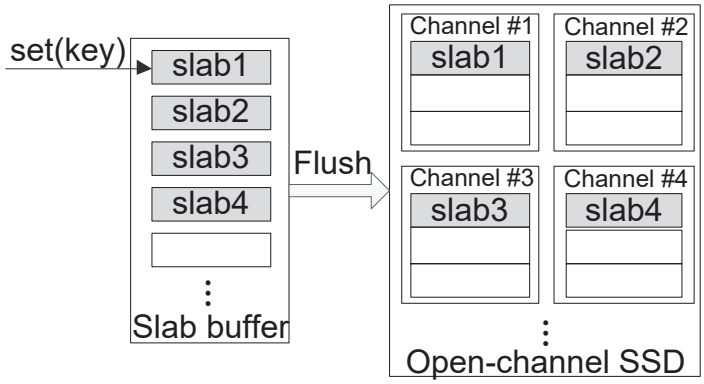


Figure 2.5: Mapping slabs to flash blocks.

multiple channels in a round-robin way. Both have pros and cons. The former is simple and allows to directly infer the logical-to-physical mapping, while the latter could yield a better bandwidth through channel-level parallelism.

We choose the simpler per-channel mapping for two reasons. First, key-value cache systems typically have sufficient slab-level parallelism. Second, per-channel allows us to directly translate "slabs" into "blocks" at the library layer with minimal calculation. For cross-channel mapping, a big slab whose size is of several flash blocks may lead to flash space waste and make the slab to block mapping more complicated. A small slab in cross-channel mapping may pollute several flash blocks upon operations of invalidating slabs, which contributes to device-level GC overhead. In fact, in our prototype, we directly map a flash slab to a physical flash block, since the block size (8MB) is appropriate as one slab. For flash devices with a smaller block size, we can group multiple contiguous blocks in one channel into one slab.

• **Slab buffer**: Unlike DRAM memory, flash does not support random in-place overwrite. As so, a key-value item cannot be directly updated in its original place in flash. For a `SET` operation, the key-value item has to be stored in a new location in flash (appended like a log), and the obsolete item will be recycled later. To enhance performance, we maintain some *in-memory slabs* as buffer for flash slabs. Upon receiving a `SET` operation, the key-value pair is first stored in the corresponding in-memory slab and completion is immediately returned. When the in-memory slab is full, it is flushed into an *in-flash slab* for persistent storage. (the

"Flush" process shown in Figure 2.5).

The slab buffer brings two benefits. First, the in-memory slab works as a write-back buffer. It not only speeds up accesses but also makes incoming requests asynchronous, which greatly improves the throughput. Second, and more importantly, the in-memory slab merges small key-value slot writes into large slab writes (in units of flash blocks), which completely removes the unwanted small flash writes. Thus, from the device's perspective, all I/Os seen at the device level are in large-size slabs, which renders the unnecessarity of the generic GC at the FTL level. For this reason, flash writes in our system are all large writes, in units of flash blocks. Our experiments show that a small slab buffer is sufficient for performance.

• **Channel selection and slab allocation**: For load balance considerations, when an in-memory slab is full, we first select the channel with the lowest load. The load of each channel is estimated by counting three key flash operations (`read`, `write`, and `erase`). Once a channel is selected, a free slab is allocated. For each channel, we maintain a *Free Slab Queue* and a *Full Slab Queue* to manage clean slabs and used slabs separately. The slabs in a free slab queue are sorted in the order of their erase counts, and we always select the slab with the lowest erase count first for wear-leveling purposes. The slabs in a full slab queue are sorted in the Least Recently Used (LRU) order. When running out of free slabs, the GC procedure is triggered to produce clean slabs, which we will discuss in more details later.

With the above optimizations, a fundamental effect is, all I/Os seen at the device level are shaped into large-size slab writes, which completely removes small page writes as well as the need for generic GC at the FTL level.

• **Unified Direct Mapping**

In order to address the double mapping problem, a key change is to remove all the intermediate mappings, and directly map the SHA-1 hash of the key to the corresponding physical location (i.e., the slab ID and the offset) in the in-memory hash table.

Figure 2.6 shows the structure of the in-memory hash table. Each hash table entry

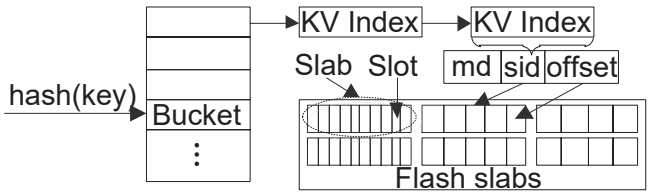


Figure 2.6: The unified direct mapping structure.

includes three fields: <md, sid, offset>. For a given key, md is the SHA-1 digest, sid is the ID of the slab that stores the key-value item, and offset is the slot number of the key-value item within the slab. Upon a request, we first calculate the hash value of the "key" to locate the bucket in the hash table, and then use the SHA-1 digest (md) to retrieve the hash table entry, in which we can find the slab (sid) containing the key-value pair and the corresponding slot (offset). The found slab could be in memory (i.e., in the slab buffer) or in flash. In the former case, the value is returned in a memory access; in the latter case, the item is read from the corresponding flash page(s).

Algorithm 2.4.1 shows the SET operation procedure in DIDACache with this unified mapping structure. When a SET request of one key-value item comes, DIDACache first checks wether it is an update operation or not. If it is an update operation, DIDACache removes the mapping record and updates the information associated with the operation of invalidating an obsolete key-value item (e.g., valid data ratio of its slab). Then, DIDACache allocates one slab whose slot size best fits this key-value pair, stores this key-value item in one slot, and updates the mapping with the slab and slot address. When there is not enough free memory slabs, the background "drain" process will be triggered to flush memory slabs to disk slabs. Similarly, an asynchronous integrated application-driven GC process will be called once there is not enough flash disk slabs inside SSD. Algorithm 2.4.2 presents the GET operations procedure, which is much simpler. When a GET request with one key comes, DIDACache searches the hash table, if the mapping record does not exist, an non-exist value is returned. Otherwise, DIDACache gets the ID of the slab ("sid") that stores the key-value item with the mapping structure. If the slab is in memory, the value is returned with one memory load operation. If the slab is in disk, DIDACache needs to read the flash page which contains the key-value item, and return the value.

The unified direct mapping brings two benefits. First, it removes the redundant lookup in the intermediate mapping structures, which speeds up the query processing. Second, and more importantly, it dramatically reduces the demand for a large and expensive on-device DRAM buffer. Since the mapping tables at different levels are collapsed into one single must-have in-memory hash table, the FTL-level mapping table becomes unnecessary and can be completely removed from the device. This saves hundreds of Megabytes to even Gigabytes of on-device DRAM space. We could either reduce production cost or make a better use of on-device DRAM, such as on-device caching/buffering.

• **Garbage Collection**

Garbage collection is a must-have in key-value cache systems, since operations (e.g., `SET` and `DELETE`) can create obsolete value items in slabs, which need to be recycled at a later time. When the system runs out of free flash slabs, we need to reclaim their space in flash.

With the semantic knowledge about the slabs, we can perform a fine-grained GC in one single procedure, running at the application level only. There are two possible strategies for identifying a victim slab: (1) *Space-based eviction*, which selects the slab containing the largest number of obsolete values, and (2) *Locality-based eviction*, which selects the coldest slab for cleaning based on the LRU order. Both policies are used depending on the runtime system condition.

• **Space-based eviction**: As a greedy approach, this scheme aims to maximize the freed flash space for each eviction. To this end, we first select a channel with the lowest load to limit the search scope, and then we search its *Full Slab Queue* to identify the slab that contains the least amount of valid data. As the slot sizes of different slab classes are different, we use the number of valid key-value items times their size to calculate the valid data ratio for a given flash slab. Once the slab is identified, we scan the slots of the slab, copy all valid slots into the current in-memory slab, update the hash table mapping accordingly, then erase the slab and place the cleaned slab back in the *Free Slab Queue* of the channel.

• **Locality-based eviction**: This policy adopts an aggressive measure to achieve fast recla-

Algorithm 2.4.1 The Key-value `SET` Procedure

Input: key: Key for this key-value item

$value$:Value for this key-value item

CH_{num}: Channel number in SSD

function BOOL SET($key, value$)

    if hash(key) exists **then** //for update operation

        remove(hash(key));

        Update the invalid information;

    end if

    Select one memory slab whose slot size best fits the key-value size;

    Insert the key-value item to the slot, establish an index for hash(key);

    if number of free memory slab $< free_{threshold}$ **then**

        slab_drain_thread(); //trigger the background slab drain process

    end if

    return true;

end function

function VOID SLAB_DRAIN_THREAD()

    while full_memory_slab $> full_{threshold}$ **do**

        if $channel(CH_{num})$ does not have free disk slab **then**

            $CH_{num} \leftarrow CH_{num} + 1$;

        end if

        Drain one memory slab to disk slab;

        if number of free disk slab $< W_{high}$ **then**

            Integrated_GC_thread();

        end if

    end while

Algorithm 2.4.2 The Key-value `GET` Procedure

Input: key: Key for this key-value item

```
1:  function VALUE GET(key)
2:      if hash(key) does not exist then
3:          return -1; //key does not exist
4:      end if
5:      sid = hash(key)
6:      if sid is in memory then
7:          return value; //return value with one memory load
8:      else
9:          flash_read(dev, sid); //read the data from flash
10:         return value;
11:     end if
12: end function
```

mation of free slabs. Similar to *space-based eviction*, we first select the channel with the lowest load. We then select the LRU slab as the victim slab to minimize the impact to hit ratio. This can be done efficiently as the full flash slabs are maintained in their LRU order for each channel. A scheme, called *quick clean*, is then applied by simply dropping the entire victim slab, including all valid slots. It is safe to remove valid slots, since our application is a key-value cache (rather than a key-value store) – all clients are already required to write key-values to the backend data store first, so it is safe to aggressively drop any key-value pairs in the cache without any data loss.

Comparing these two approaches, *space-based eviction* needs to copy still-valid items in the victim slab, so it takes more time to recycle a slab but retains the hit ratio. In contrast, *locality-based eviction* allows to quickly clean a slab without moving data, but it aggressively erases valid key-value items, which may reduce the cache hit ratio. To reach a balance be-

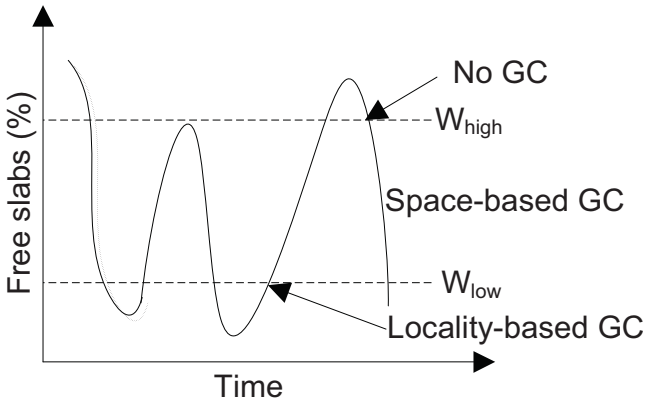


Figure 2.7: Low and high watermarks.

tween the hit ratio and GC overhead, we apply these two policies *dynamically* during runtime – when the system is under high pressure (e.g., about to run out of free slabs), we use the fast but imprecise *locality-based eviction* to quickly release free slabs for fast response; when the system pressure is low, we use *space-based eviction* and try to retain all valid key-values in the cache for hit ratio.

To realize the above-mentioned dynamic selection policies, we set two watermarks, low (W_{low}) and high (W_{high}). We will discuss how to determine the two watermarks in the next section. As shown in Algorithm 2.4.3, the GC procedure checks the number of free flash slabs, S_{free}, in the current system periodically. If S_{free} is between the high watermark, W_{high}, and the low watermark, W_{low}, it means that the pool of free slabs is running low but under moderate pressure. So we activate the less aggressive *space-based eviction* policy to clean slabs. This process repeats until the number of free slabs, S_{free}, reaches the high watermark. If S_{free} is below the low watermark, which means that the system is under high pressure, the aggressive *space-based eviction* policy kicks in and uses *quick clean* to erase the entire LRU slab and discard all items immediately. This fast-response process repeats until the number of free slabs in the system, S_{free}, is brought back to W_{low}. If the system is idle, the GC procedure switches to the *space-based eviction* policy and continues to clean slabs until reaching the high watermark. Figure 2.7 illustrates this process.

• Over-Provisioning Space Management

In conventional SSDs, a large portion of flash space is reserved as OPS, which is

Algorithm 2.4.3 The Integrated Application Driven Garbage Collection Procedure

Input: F_{dslab} : The number of free disk slab

W_{low} : Low watermark

W_{high} : High watermark

CH_{num} : Channel number in SSD

Output: Reclaim disk slabs.

if Timer **then**

Space-based eviction:

  if F_{dslab} is less than W_{high} **and** larger than W_{low}; **then**

    Choose a slab with maximum invalid data from the full slab queue of channel CH_{num};

    Scan the slab and do valid key-value pair copy;

    Erase the slab and insert it into the free slab queue CH_{num};

    $CH_{num} \leftarrow CH_{num}$ + 1;

    if CH_{num} equals to $Total\_CH$; **then**

      $CH_{num} \leftarrow 0$;

    end if

  end if

  if idle **and** F_{dslab} is less than W_{high}; **then**

    goto Space-based eviction

  end if

Locality-based eviction:

  while F_{dslab} is less than W_{low}; **do**

    Choose a victim disk slab which is recently least accessed from the

    LRU full disk slab queue CH_{num};

    Erase the slab and insert it into the free slab queue CH_{num};

    $CH_{num} \leftarrow CH_{num}$ + 1;

    if CH_{num} equals to $Total\_CH$; **then**

      $CH_{num} \leftarrow 0$;

    end if

  end while

end if

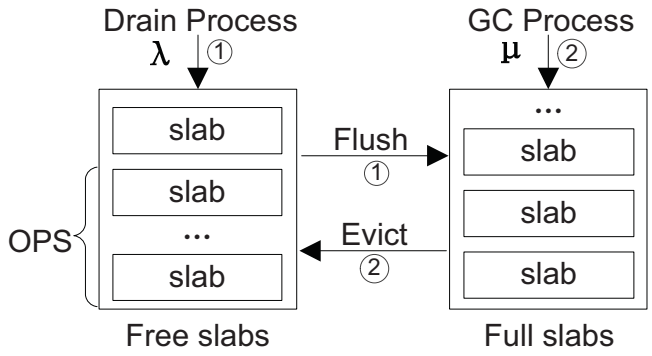


Figure 2.8: Throughput for key-value items of size 256 bytes with different `SET`/`Get` ratios.

invisible and unusable by applications. In our architecture, applications can access all the physical flash blocks. We aim to leverage the application's domain knowledge to dynamically adjust the reserved space and maximize the usable flash space for caching. In the following, we refer to this dynamically changeable reserved space as OPS, and build a model to adjust its size during the run time.

In our system, the two watermarks, W_{low} and W_{high}, drive the GC procedure. The two watermarks effectively determine the available OPS size – W_{low} is the dynamically adjusted OPS size, and W_{high} can be viewed as the upper bound of allowable OPS. We set the difference between the two watermarks, $W_{high} - W_{low}$, as a constant (15% of the flash space in our prototype). Ideally, we desire to have the number of free slabs, S_{free}, fluctuating in the window between the two watermarks.

Our goal is to keep just enough flash space for over-provisioning. However, it is challenging to appropriately position the two watermarks and make them adaptive to the workload. It is desirable to have an automatic, self-tuning scheme to dynamically determine the two watermarks based on runtime situation. In our prototype, we have designed two schemes, a *feedback-based heuristic model* and a *queuing theory based model*.

Our heuristic scheme is simple and works as follows: when the low watermark is hit, which means that the current system is under high pressure, we lift the low watermark by doubling W_{low} to quickly respond to increasing writes, and the high watermark is correspondingly updated. As a result, the system will activate the aggressive *quick clean* to produce more free slabs quickly. This also effectively reserves a large OPS space for use.

When the number of free slabs reaches the high watermark, which means the current system is under light pressure, we linearly drop the watermarks. This effectively returns free slabs back to the usable cache space (i.e., reduced OPS size). In this way, the OPS space automatically adapts to the incoming traffic.

The second scheme is based on the well-known queuing theory, which builds slab allocation and reclaim processes as a M/M/1 queue. As Figure 2.8 shows, in this system, we maintain queues for free flash slabs and full flash slabs for each channel, separately. The slab drain process consumes free slabs, and the GC process produces free slabs. Therefore we can view the drain process as the consumer process, the GC process as the producer process, and the free slabs as resources. The drain process consumes flash slabs at a rate λ, and the GC process generates free flash slabs at a rate μ. Prior study [11] shows that in real applications, the incoming of key-value pairs can be seen as a Markov process, so the drain process is also a Markov process. For the GC process, when S_{free} is less than W_{low}, the locality-based eviction policy is adopted. The time consumed for reclaiming one slab is equal to the flash erase time plus the schedule time. The flash block erase time is a constant, and the schedule time can be viewed as a random number. Thus the locality-based GC process is also a Markov process with a service rate μ. Based on the analysis, the process can be modeled as a M/M/1 queue with arrival rate λ, service rate μ, and one server.

According to Little's law, the expected number of slabs waiting for service is $\lambda/(\mu - \lambda)$. If we reserve at least this number of free slabs before the locality-based GC process is activated, we can always eliminate the synchronous waiting time. So, for the system performance benefit, we set

$$W_{low} = \lambda/(\mu - \lambda) \tag{2.1}$$

In the above equation, λ is the slab consumption rate of the drain process, and μ is the slab reclaim rate of GC, which equals $1/(t_{evict} + t_{other})$, where t_{evict} is the block erase time, and t_{other} is other system time needed for GC.

In Equation 2.2, the arrival rate is decided by the incoming rate of key-value pairs and their average size, which are both measurable. Assuming the arrival rate of key-values

is λ_{KV}, the average size is S_{KV}, and the slab size is S_{slab}, λ can be calculated as follows.

$$\lambda = \frac{\lambda_{KV} \times S_{KV}}{S_{slab}} \tag{2.2}$$

So, we have

$$W_{low} = \frac{\lambda_{KV} \times S_{KV} \times (t_{evict} + t_{other})}{S_{slab} - \lambda_{KV} \times S_{KV} \times (t_{evict} + t_{other})} \tag{2.3}$$

By using the above-mentioned equations, we can periodically update the settings of the low and high watermarks. In this way, we can adaptively tune the OPS size based on real-time workload demands.

• **Wear-leveling**

Flash memory wears out after a certain number of Program/Erase (P/E) cycles. In our prototype, key-value update operations are performed in an out-of-place way, meaning that the updated key-value items are stored within the newly allocated slabs, and the stale key-value items need to be reclaimed through the GC process. For wear leveling, when allocating slabs in the drain process and reclaiming slabs in the GC process, we take the erase count of each slab into consideration and always use the block with the smallest erase count. Our locality-based GC that selects the least recently used blocks also helps evict those cold key-value items from their occupied flash blocks. Furthermore, as our channel-slab selection and slab-allocation scheme can evenly distribute the workloads across all channels, wears can be approximately distributed across channels as well.

Despite of these optimization policies, uneven aging still exists. For example, flash blocks which are filled with read intensive key-value items may be rarely erased. To further ensure uniform aging of all flash blocks, we adopt a simple yet effective approach by periodically invoking the wear-leveling procedure. Nonetheless, instead of swapping flash blocks that have higher wear number with those lower ones, we propose to incorporate this periodical wear-leveling procedure within the GC process.

In DIDACache, we maintain the total erase count and erase number of each flash slab. The wear-leveling process is periodically triggered when the total erase count exceeds

m times of the total flash block number in the system. For example, we set $m = 2$ in our prototype. Suppose there are 1,000 flash blocks in the system, then the wear-leveling process will be triggered when the total erase count equals to 2,000. Once the wear-leveling process is triggered, we calculate the average wear number of flash blocks, and identify those flash blocks whose erase counts are far lower than the average number. These cold slabs are either seldom accessed or read-intensive. If a victim slab is seldom accessed, we can directly evict it out (just as quick-clean). If a victim slab is read-intensive, instead of simply swapping key-value items stored in the cold flash slab with a hot slab, DIDACache marks the cold block as victim block, and puts them into the GC queue. The GC process will reclaim these cold flash blocks and put them into the free slab queue to serve new incoming requests.

Traditional wear-leveling requires to shuffle frequently erased flash blocks with the less frequently erased ones, which involves a large amount of data copy, consuming I/O bandwidth and also increasing P/E cycles. In DIDACache, we are able to directly integrate wear-leveling within the GC procedure. This optimization policy reduces the amount of unnecessary data copy without defeating the purpose of GC and wear-leveling. In particular, since DIDACache does not support in-place update, if a slab has write-intensive key-value items, they must have already been copied out to other blocks, leaving obsolete slots ready for recycling. Thus, unlike traditional wear-leveling, we are able to skip copying these data. If a key-value items in the slab are not frequently read, as described in Section 2.4.1, DIDACache will devote the slab as "inactive" by checking its access count and use quick clean to directly erase this entire slab without moving data. Only if the key-value items are read-intensive, the GC process will find the slab active, and these hot items will be copied before erasing the slab. Thus, compared to traditional wear-leveling, this approach only needs to copy read-intensive data, achieving both effective wear-leveling and minimized data copy.

• Crash Recovery

Crash recovery is also a challenge. As a typical key-value cache, all the key-value items have their persistent copy in the back database store. Thus, when system crash happens, we may simply drop the entire cache upon crashes. However, due to the excessively

long warm-up time, it is preferred to retain the cached data through crashes [135]. In our system, all key-value items are stored in persistent flash but the hash table is maintained in volatile memory. There are two potential solutions to recover the hash table. One simple method is to scan all the valid key-value items in flash and rebuild the hash table, which is a time-consuming process. This approach demands more time for reconstructing the hash table. A more efficient solution is to periodically checkpoint the in-memory hash table into (a designated area of) the flash. Upon recovery, we only need to reload the latest hash table checkpoint into memory and then apply changes by scanning the slabs written after the checkpoint. Crash recovery is currently not implemented in our prototype. Applications use a persistent cache to improve repeated accesses. However, it is possible that the data in the backend data store are updated during the period of cache server downtime. Handling this situation is out of the scope of a look-aside cache, and applications or systems should implement certain methods to ensure that the data in the cache are still up-to-date after recovery. For example, when updating data, if the application finds the cache server is offline, it should not only update the data in the backend data store but also log the update operations locally or in another server, and when the cache server is recovered, the cache can be brought back to a consistent state by examining the log and replaying the update operations.

2.4.2 Library Level: `libssd`

As an intermediate layer, the library, `libssd`, connects the application and device layers. Unlike Liblightnvm [38], `libssd` is highly integrated with the key-value cache system. It has three main functions: (1) *Slab-to-block mapping*, which statically maps a slab to one (or multiple contiguous) flash memory block(s) in a channel. In our prototype, it is a range of blocks in a flash LUN (logic unit number). Such a mapping can be calculated through a mathematical conversion and does not require another mapping table. (2) *Operation transformation*, which converts key slab operations, namely `read`, `write`, and `erase`, to flash memory operations. This allows the key-value cache system to operate in units of slabs, rather than flash pages/blocks. (3) *Bad block management*, which maintains a list of flash

blocks that are detected as "bad" and ineligible for allocation, and hides them from the key-value cache.

2.4.3 Hardware Level: Open-Channel SSD

Recently, there is a new trend of SSD design, called open-channel SSD, which directly exposes the internal channels and its low-level flash details to the host. With open-channel SSD, the responsibility of flash management is shared between the host software and hardware device. Compared with conventional SSD design, open-channel SSD has three unique features: (1) SSD internal parallelism is exposed to user applications. Open-channel SSD exposes its internal geometry details (e.g., the layout of channels, LUNs, and flash blocks) to software applications. Applications have the flexibility of scheduling I/O tasks among different channels to fully utilize the raw flash performance. (2) Block erase command is available to applications. Open-channel SSD exposes its low-level details to applications, thus, the applications are capable of controlling the flash GC process. (3) Open-channel SSD enjoys a simplified I/O stack. Applications can directly operate the device hardware through the `ioctl` interface, which allows them to bypass many intermediate OS components, such as file system and the block I/O layer.

We use an open-channel SSD manufactured by Memblaze [85]. This hardware is similar to that used in SDF [96]. This PCIe based SSD contains 12 channels, each of which connects to two Toshiba 19nm MLC flash chips. Each chip contains two planes and has a capacity of 66GB. Unlike SDF [96], our SSD exposes several key device-level properties: first, the SSD exposes the entire flash memory space to the upper level. The SSD hardware abstracts the flash memory space in 192 LUNs, and an LUN is the smallest parallelizable unit. The LUNs are mapped to the 12 channels in a sequential manner, i.e., channel #0 contains LUNs 0-15, channel #1 contains LUNs 16-31, and so on. Therefore, we know the physical mapping of slabs on flash memory and channels. Second, unlike SDF, which presents the flash space as 44 block devices, our SSD provides direct access to raw flash memory through the `ioctl` interface. It allows us to directly operate the target flash memory

pages and blocks by specifying the LUN ID and page number to compose commands added to the device command queue. Third, all FTL-level functions, such as address mapping, wear-leveling, bad block management, are bypassed. This allows us to remove the device-level redundant operations and make them completely driven by the user-level applications.

2.5 Evaluation

In this section, we present evaluation results that demonstrate the benefits of the design choices of DIDACache. Specially, we seek to answer the following fundamental performance questions about DIDACache:

- Does this co-design approach result in higher SSD utilization, and how does it impact performance (throughput, latency), and device endurance?
- How does DIDACache perform with real workloads, compared to its peers?
- What is the effect of memory slab buffer on DIDACache's performance?
- What is DIDACache's garbage collection overhead with different policies?
- How does the dynamic over-provisioning space schemes perform?
- What is the CPU and memory overhead of DIDACache?

2.5.1 Prototype System

We have prototyped the proposed key-value cache on the open-channel SSD hardware platform manufactured by Memblaze [85]. Our implementation of the key-value cache manager is based on Twitter's Fatcache [123]. It includes 1,640 lines of code in the stock Fatcache and 620 lines of code in the library.

In Fatcache, when a `SET` request arrives, if running out of in-memory slabs, it selects and flushes a memory slab to flash. If there is no free flash slab, a victim flash slab is

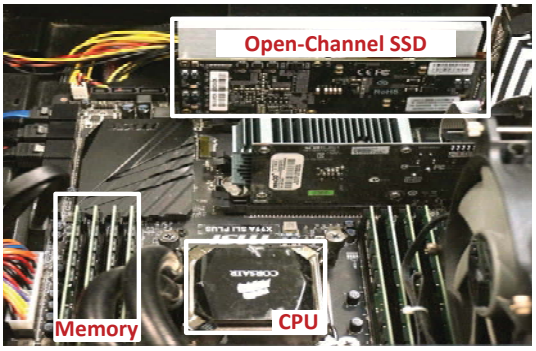


Figure 2.9: Hardware platform.

chosen to reclaim space. During this process, incoming requests have to wait synchronously. To fairly compare with a cache system with non-blocking flush and eviction, we have enhanced the stock Fatcache [**?**] by adding a drain thread and a slab eviction thread. The other part remains unchanged. We have open-sourced our asynchronous version of Fatcache for public downloading [1]. In our experiments, we denote the stock Fatcache working in the synchronous mode as "Fatcache-Sync", and the enhanced one working in the asynchronous mode as "Fatcache-Async" [1]. For each platform, we configure the slab size to 8 MB, the flash block size. The memory slab buffer is set to 128MB.

For performance comparison, we also run Fatcache-Sync and Fatcache-Async on a commercial PCI-E SSD manufactured by Memblaze. The SSD is built on the exact same hardware as our open-channel SSD but adopts a typical, conventional SSD architecture design. This SSD employs a page-level mapping and the page size is 16KB. Unlike the open-channel SSD, the commercial SSD has 2GB of DRAM on the device, which serves as a buffer for the mapping table and a write-back cache. The other typical FTL functions (e.g., wear-leveling, GC, etc.) are active on the device.

2.5.2 Experimental Setup

Our experiments are conducted on a workstation, which features an Intel i7-5820K 3.3GHZ processor and 16GB memory. An open-channel SSD introduced in Section 2.4.3 is used as DIDACache's underlying cache storage. (Figure 2.9). Since the SSD capacity is quite

large (1.5TB), it would take excessively long time to fill up the entire SSD. To complete our tests in a reasonable time frame, we only use part of the flash space, and we ensure the used space is evenly spread across all the channels and flash LUNs. Note that for the commercial SSD, since we cannot control its OPS space, Fatcache running on the commercial SSD is able to use more OPS space than it should, which favors the stock Fatcache configuration as a comparison to our DIDACache.For the software, we use Ubuntu 14.04 with Linux kernel 3.17.8. Our backend database server is MySQL 5.5 with InnoDB storage engine running on a separate workstation, which features an Intel Core 2 Duo processor (3.13GHZ), 8GB memory and a 500GB hard drive. The database server and the cache server are connected in a 1-Gbps local Ethernet network. Note that in our experimental environment, network is not the bottleneck.Fatcache-Sync and Fatcache-Async use the same system configurations, except that they run on the commercial SSD rather than the open-channel SSD.

2.5.3 Overall Performance

Our first set of experiments simulate a production data-center environment to show the overall performance. In this experiment, we have a complete system setup with a workload generator (client simulator), a key-value cache server, and a MySQL database server in the backend.

To generate key-value requests to the cache server, we adopt a workload model presented in prior work [14]. This model is built based on real Facebook workloads [11], and we use it to generate a key-value object data set and request sequences to exercise the cache server. The size distribution of key-value objects in the database follows a truncated Generalized Pareto distribution with location $\theta = 0$, scale $\psi = 214.4766$, and shape $k = 0.348238$. The object popularity, which determines the request sequence, follows a Normal distribution with mean μ_t and standard deviation σ, where μ_t is a function of time. We first generate 800 million key-value pairs (about 250GB data) to populate our database, and then use the object popularity model to generate 200 million requests. We have run experiments with various numbers of servers and clients with the above-mentioned workstation, but due to the

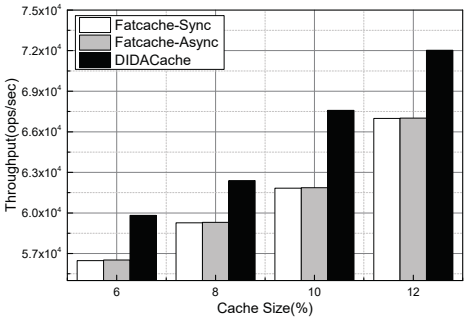


Figure 2.10: Throughput vs. cache size

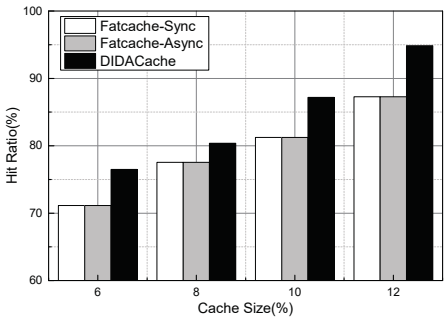


Figure 2.11: Hit ratio vs. cache size.

space constraint, we only present the representative experimental results with 32 clients and 8 key-value cache servers.

We test the system performance by varying the cache size (in percentage of the data set size). Figure 2.10 shows the throughput, i.e., the number of operations per second (ops/sec). We can see that as the cache size increases from 5% to 12%, the throughput of all the three schemes improves significantly, due to the improved cache hit ratio. Comparing the three schemes, DIDACache outperforms Fatcache-Sync and Fatcache-Async substantially. With a cache size of 10% of the data set (about 25GB), DIDACache outperforms Fatcache-Sync and Fatcache-Async by 9.7% and 9.2%, respectively. The main reason is that the dynamic OPS management in DIDACache adaptively adjusts the reserved OPS size according to the request arrival rate. In contrast, Fatcache-Sync and Fatcache-Async statically reserve 25% flash space as OPS, which affects the cache hit ratio (see Figure 2.11). Another reason is the reduced overhead due to the application-driven GC. The effect of GC policies will be examined in Section 2.5.4.

We also note that Fatcache-Async only outperforms Fatcache-Sync marginally in this workload. It is because for this workload, both Fatcache-Sync and Fatcache-Async use the commercial SSD as the underlying storage and use the static OPS policy; thus, they have the same cache hit ratio. Though Fatcache-Async adopts an asynchronous drain process and GC process, they only benefit the 'set' operations, and its 'get' performance is identical to Fatcache-Sync. When the cache size varies from 5% to 12% of the workload size, the cache hit ratio can range from 71% to 87%, which is already high; thus, we cannot see much further improvement between Fatcache-Async and Fatcache-Sync. Besides, when a cache

miss happens, a slow database query is needed, so the relative benefit from asynchronization is further diminished.

Figure 2.11 shows the hit ratios of these three cache systems. We can see that, as the cache size increases, DIDACache's hit ratio ranges from 76.5% to 94.8%, which is much higher than that of Fatcache-Sync, ranging from 71.1% to 87.3%.

2.5.4 Cache Server Performance

In this section we focus on studying the performance details of the cache servers. In this experiment, we directly generate `SET`/`GET` operations to the cache server. We create objects with sizes ranging from 64 bytes to 4KB and first populate the cache server up to 25GB in total. Then we generate `SET` and `GET` requests of various key-value sizes to measure the average latency and throughput. All experiments use 8 key-value cache servers and 32 clients.

• **Random `SET`/`GET` Performance**

Figure 2.12 shows the throughput of `SET` operations. Among the three schemes, our DIDACache achieves the highest throughput and Fatcache-Sync performs the worst. With the object size of 64 bytes, the throughput of DIDACache is 2.48×10^5 ops/sec, which is 1.3 times higher than that of Fatcache-Sync and 35.5% higher than that of Fatcache-Async. The throughput gain is mainly due to our unified slab management policy and the integrated application-driven GC policy. DIDACache also selects the least loaded channel when flushing slabs to flash. Thus, the SSD's internal parallelism can be fully utilized, and with software and hardware knowledge, the GC overhead is significantly reduced. Compared with Fatcache-Async, the relative performance gain of DIDACache is smaller and decreases as the key-value object size increases. As the object size increases, the relative GC efficiency improves and the valid data copy overhead is decreased. It is worth noting that the practical systems are typically dominated by small key-value objects, on which DIDACache performs particularly well.

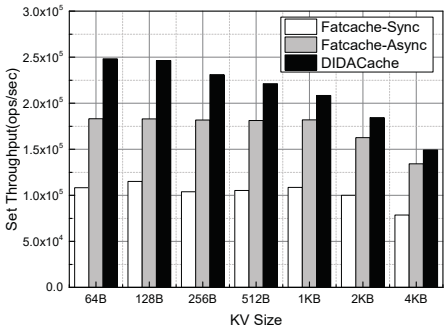


Figure 2.12: SET throughput vs. KV size

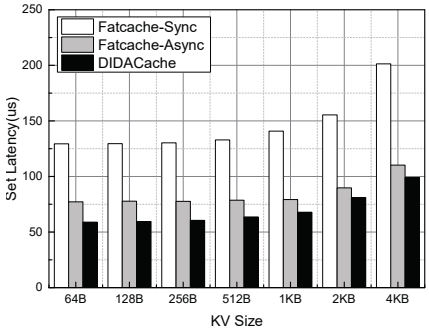


Figure 2.13: SET latency vs. KV size

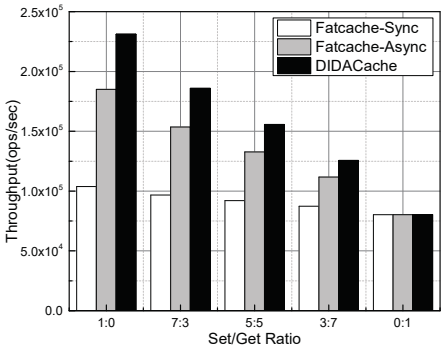


Figure 2.14: Throughput vs. SET/GET ratio.

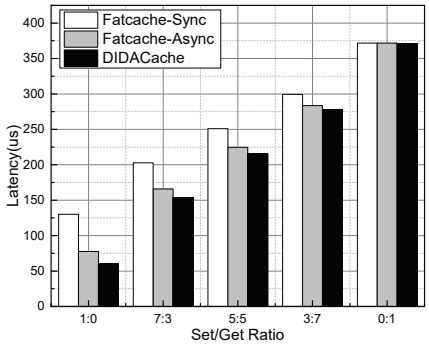
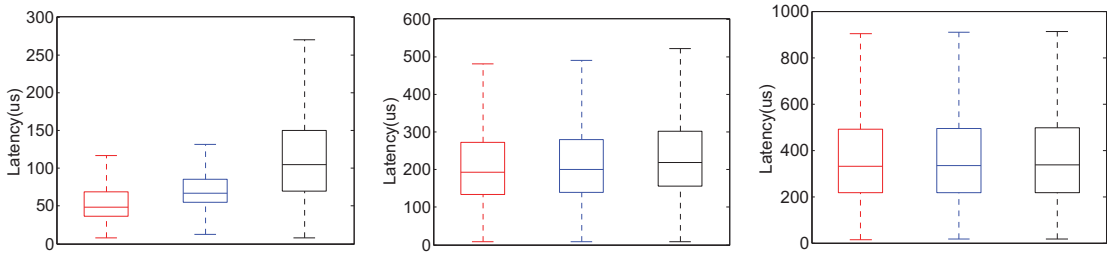


Figure 2.15: Latency vs. SET/GET ratio.



(a) SET/GET(1:0). (b) SET/GET(1:1). (c) SET/GET(0:1).

Figure 2.16: Latency (256-byte KV items) with different SET/GET ratios.

Figure 2.13 gives the average latency for SET operations with different key-value object sizes. Similarly, it can be observed that Fatcache-Sync performs the worst, and DIDACache outperforms the other two significantly. For example, for 64-byte objects, compared with Fatcache-Sync and Fatcache-Async, DIDACache reduces the average latency by 54.5% and 23.6%, respectively.

Figures 2.14 and 2.16 show the throughput and latency for workloads with mixed

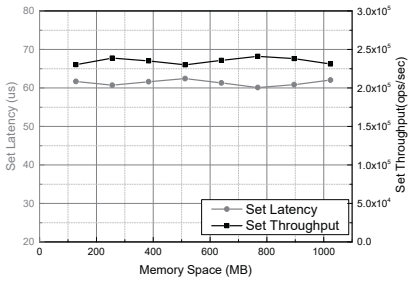


Figure 2.17: Latency and Throughput for Set Operation with Different Buffer Size.

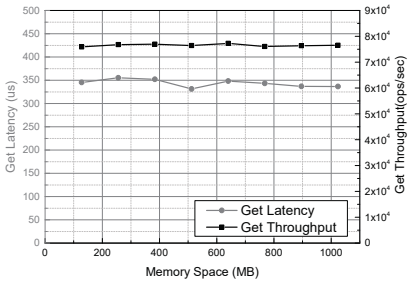


Figure 2.18: Latency and Throughput for Get Operation with Different Buffer Size.

SET/GET operations. We can observe that DIDACache outperforms Fatcache-Sync and Fatcache-Async across the board, but as the portion of GET operations increases, the related performance gain reduces. Although we also optimize the path of processing GET, such as removing intermediate mapping, the main performance bottleneck is the raw flash read. Thus, with the workload of 100% GET, the latency and throughput of the three schemes are nearly the same, which also indicates that the performance overhead (e.g., maintaining queues) introduced by our scheme is minimal. Figure 2.16 shows the latency distributions for key-value items of 256 bytes with different SET/GET ratios.

• **Memory Slab Buffer** Memory slab buffer enables the asynchronous operations of the drain and GC processes. To show the effect of slab buffer size, we vary the slab buffer size from 128MB to 1GB and test the average latency and throughput with the workloads generated with the truncated Generalized Pareto distribution. As shown in Figure 2.17 and Figure 2.18, for both SET and GET operations, the average latency and throughput are insensitive to the slab buffer size, indicating that a small in-memory slab buffer size (128M) is sufficient.

• **Garbage Collection**

Our cross-layer solution also effectively reduces the GC overhead, such as erase and valid page copy operations. In our cache-driven system, we can easily count erase and page copy operations in the library code. However, we cannot directly obtain these values on the commercial SSD as they are hidden at the device level. For effective comparison, we use the SSD simulator (extension to DiskSim [50]) from Microsoft Research and configure

Table 2.1: Garbage collection overhead.

GC Scheme	Key-values	Flash Page	Erase
DIDACache-Space	7.48GB	N/A	4,231
DIDACache-Locality	0	N/A	3,679
DIDACache	2.05GB	N/A	3,829
Fatcache-Greedy	7.48GB	5.73GB	5,024
Fatcache-Kick	0	3.86GB	4,122
Fatcache-FIFO	15.35GB	0	5,316

it with the same parameters of the commercial SSD. We first run the stock Fatcache on the commercial SSD and collect traces by using `blktrace` in Linux, and then replay the traces on the simulator. We compare our results with the simulator-generated results. In our experiments, we confine the available SSD size to 30GB, and preload it with 25GB data with workloads generated with the truncated Generalized Pareto distribution, and then do `SET` operations (80 million requests, about 30GB), following the Normal distribution.

Table 3.1 shows GC overhead in terms of valid data copies (key-values and flash pages) and block erases. We compare DIDACache using space-based eviction only ("DIDACache-Space"), locality-based eviction only ("DIDACache-Locality"), the adaptively selected eviction approach ("DIDACache") with the stock Fatcache using three schemes ("Fatcache-Greedy", "Fatcache-Kick", and "Fatcache-FIFO"). In Fatcache, the application-level GC has two options, copying valid key-value items from the victim slab for retaining hit ratio or directly dropping the entire slab for speed. This incurs different overheads of key-value copy operations, denoted as "Key-values". In this experiment, both Fatcache-Greedy and Fatcache-Kick use a greedy algorithm to find a victim slab, but the former performs key-value copy operations while the latter does not. Fatcache-FIFO uses a FIFO algorithm to find the victim slab and copies still-valid key-values. In the table, the flash page copy and block erase operations incurred by the device-level GC are denoted as "Flash Page" and "Erase", respectively.

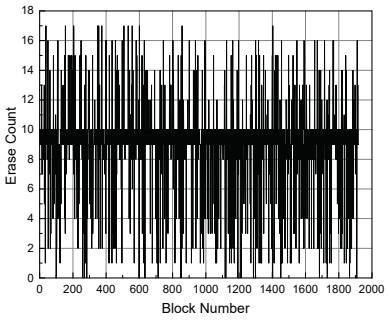


Figure 2.19: Wear distribution among blocks without wear-leveling.

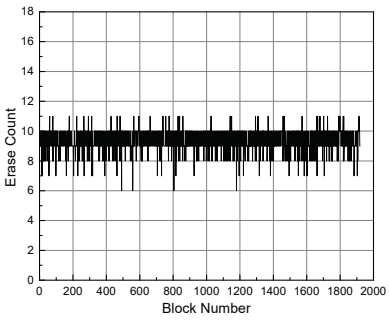


Figure 2.20: Wear distribution among blocks with wear-leveling.

Fatcache schemes show high GC overheads. For example, both Fatcache-Greedy and Fatcache-FIFO recycle valid key-value items at the application level, incurring a large volume of key-value copies. Fatcache-Kick, in contrast, aggressively drops victim slabs without any key-value copy. However, since it adopts a greedy policy (as Fatcache-Greedy) to evict the slabs with least valid key-value items, erase blocks are mixed with valid and invalid pages, which incurs flash page copies by the device-level GC. Fatcache-FIFO fills and erases all slabs in a sequential FIFO manner, thus, no device-level flash page copy is needed. All three Fatcache schemes show a large number of block erases.

The GC process in our scheme is directly driven by the key-value cache. It performs a fine-grained, single-level, key-value item-based reclamation, and no flash page copy is needed (denoted as "N/A" in Table 3.1). The locality-based eviction policy enjoys the minimum data copy overhead, since it aggressively evicts the LRU slab without copying any valid key-value items. The space-based eviction policy needs to copy 7.48 GB key-value items and incurs 4,231 erase operations. DIDACache dynamically chooses the most appropriate policy at runtime, so it incurs a GC overhead between the above two (2.05 GB data copy and 3,829 erases). Compared to Fatcache schemes, the overheads are much lower (e.g., 28% lower than Fatcache-FIFO).

• **Wear-leveling**

To investigate the block aging status in DIDACache, we carry out experiments by keeping issuing `SET` and `GET` operations to DIDACache and collect the distribution of block

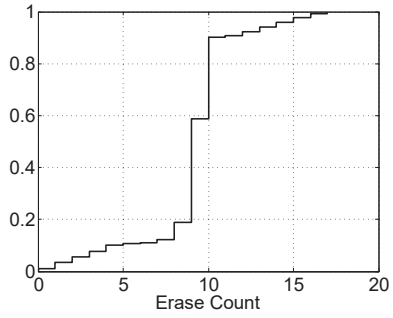


Figure 2.21: CDF of blocks' erase count without wear-leveling.

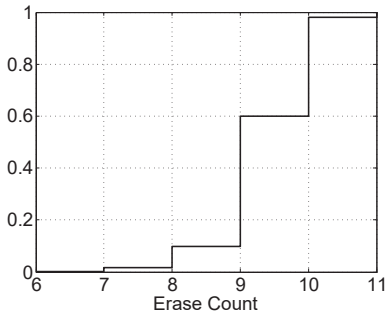


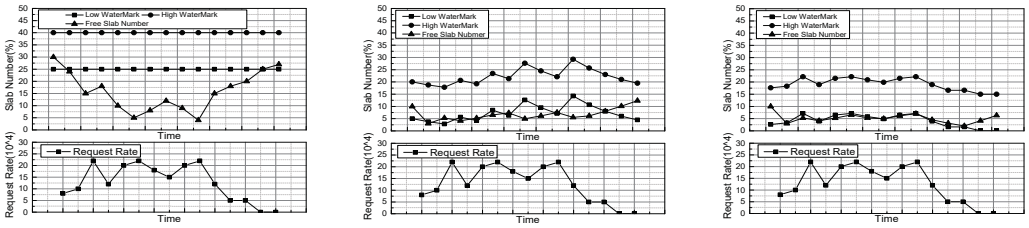
Figure 2.22: CDF of blocks' erase count with wear-leveling.

Table 2.2: Wear-leveling overhead.

	GC		Wear-leveling		**Flash Page**
	Data copy	Erase	Data copy	Erase	
Wear-leveling	13.48GB	16,542	6.34GB	1,323	N/A
No Wear-leveling	15.57GB	17,285	N/A	N/A	N/A

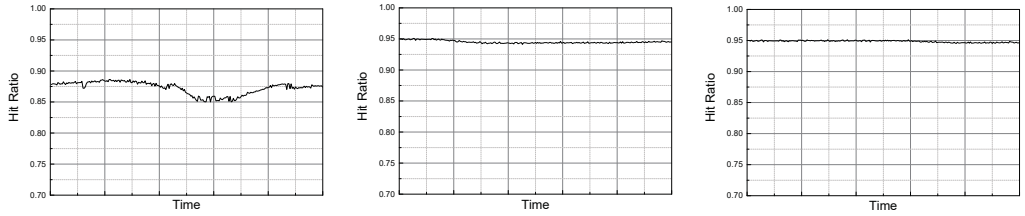
erase operations in our library layer. In this experiment, to control the experimental time, we further confine the available SSD size to 15GB, and preload it with 10GB data with workloads generated with the truncated Generalized Pareto distribution, and then do `SET` and `GET` operations with workloads (480 million requests, about 240GB, `SET`/`GET` ratio is 1:1) that follow the Normal distribution. During the experiment, we count the number of GC operations, and our wear-leveling policy is periodically triggered when the total GC time comes up to two times of the total number of flash blocks. When the wear-leveling is triggered, we mark those blocks whose erase count is less than half of the average block erase count as victim blocks, and then reclaim these flash blocks with the GC process.

Figure 2.19 and Figure 2.20 show the block wear out distribution before and after we apply our wear-leveling policy, respectively. It can be observed that after applying our wear-leveling policy, flash blocks in the system are worn out much more evenly. Without wear-leveling, the minimum block erase count is 0, and the maximum block erase count is 17. With our wear-leveling policy, the flash block erase counts vary between 6 and 11. The



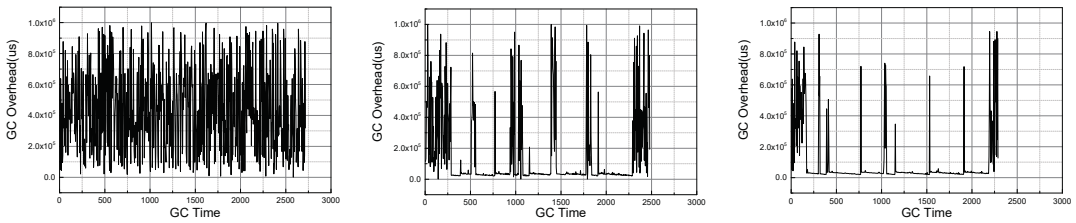
(a) Static policy. (b) Heuristic policy. (c) Queuing theory policy.

Figure 2.23: Over-provisioning space with different policies.



(a) Static policy. (b) Heuristic policy. (c) Queuing theory policy.

Figure 2.24: Hit ratio with different OPS policies.

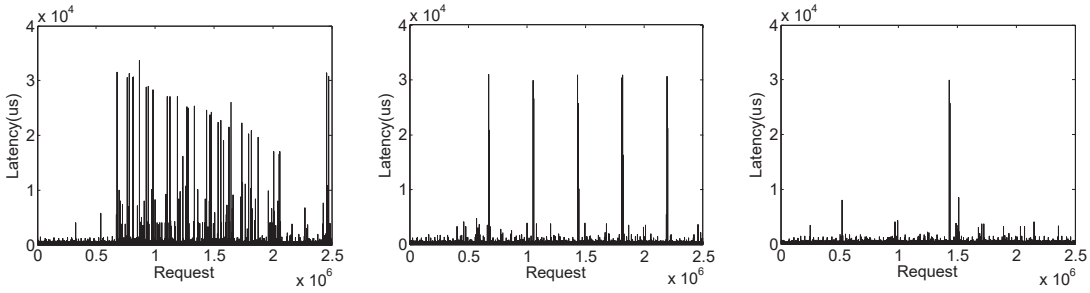


(a) Static policy. (b) Heuristic policy. (c) Queuing theory policy.

Figure 2.25: Garbage collection overhead with different OPS policies.

maximum gap is only 5, which is much smaller than 17 in the former case. Figure 2.21 and Figure 2.22 give the CDF graphs of block erase counts accordingly. From them, we can see that with our wear-leveling policy, more than 90% flash blocks are erased 9 or 10 times. For the scheme without wear-leveling, although the majority of flash blocks are also erased by 9 or 10 times, but the percentage is much smaller, and the variance range is also much larger.

The experimental results show that the our wear-leveling policy can effectively balance wears across flash blocks. However, since the wear-leveling policy incurs more GC



(a) Static policy. (b) Heuristic policy. (c) Queuing theory policy.

Figure 2.26: Request latency with different OPS policies.

operations, it also introduces some overhead. To illustrate the overhead of this mechanism, we compare the GC overheads of the systems with and without the wear-leveling policy (denoted as No wear-leveling and Wear-leveling) in Table 2.2. In this table, "Data copy" and "Erase" under column "GC" represent valid data copy and block erase operations caused by the GC process. Similarly, "Data copy" and "Erase" under column "Wear-Leveling" represent valid data copy and block erase operations caused by the wear-leveling process.

During the experiment, wear-leveling is triggered 4 times, and incurs 1323 block erase operations and 6.34GB data copy. Additionally, after applying our wear-leveling policy, the overhead of the GC procedure is less than that without our wear-leveling policy. The reason behind this is that we have integrated our wear-leveling procedure with the GC process. When wear-leveling happens, instead of swapping cold blocks with hot blocks, we mark those cold blocks as victim blocks. When reclaiming these victim blocks, we only copy those valid key-value items. If a victim block is not frequently accessed, we would directly erase the flash block without coping data. These measures, to some extent, can ease the pressure of the GC process. In all, with our wear-leveling, 580 more block erase and 4.25GB more data copy operations are introduced. We can further mitigate this overhead using a longer interval for wear-leveling, if needed.

• **Dynamic Over-Provisioning Space** To illustrate the effect of our dynamic OPS management, we run DIDACache on our testbed that simulates the data center environment in Section 2.5.3. We use the same data set containing 800 million key-value pairs (about 250G-

Table 2.3: Effect of different OPS policies.

GC Scheme	Hit Ratio	GC	Latency	Throughput
Static	87.7 %	2716	79.95	198,076
Heuristic	94.1 %	2480	64.24	223,146
Queuing	94.8 %	2288	62.41	229,956

B), and the request sequence generated with the Normal distribution model. We set the cache size as 12% (around 30GB) of the data set size. In the experiment, we first warm up the cache server with the generated data, and then change the request coming rates to test our dynamic OPS policies.

Figure 2.23 shows the dynamic OPS and the number of free slabs with the varying request incoming rates for three different policies. The static policy reserves 25% of flash space as OPS to simulate the conventional SSD. For the heuristic policy, we set the initial W_{low} with 5%. For the queuing theory policy, we use the model built in Equation 2.3 to determine the value of W_{low} at runtime. We set W_{high} 15% higher than W_{low}. The GC is triggered when the number of free slabs drops below W_{high}.

As shown in Figure 2.23(a), the static policy reserves a portion of flash space for over-provisioning. The number of free slabs fluctuates, responding to the incoming request rate. In Figure 2.23(b), our heuristic policy dynamically changes the two watermarks. When the arrival rate of requests increases, the low watermark, W_{low}, increases to aggressively generate free slabs by using *quick clean*. The number of free slabs approximately follows the trend of the low watermark, but we can also see a lag-behind effect. Our queuing policy in Figure 2.23(c) performs even better since it dynamically adjust the OPS according to the characteristics of the workloads , and it can be observed that the free slab curve almost overlaps with the low watermark curve. Compared with the static policy, both heuristic and queuing theory policies enable a much larger flash space for caching. Accordingly, we can see in Figure 2.24 that the two dynamic OPS policies are able to maintain a hit ratio close to 95%, which is 7% to 10% higher than the static policy. Figure 3.8 shows the GC cost,

and we can find that the two dynamic policies incur lower overhead than the static policy. In fact, compared with the static policy and the heuristic policy, the queuing theory policy erases 15.7% and 8% less flash blocks, respectively. Correspondingly, in Figure 2.26, it can be observed that the queuing policy can most effectively reduce the number of requests with high latencies.

To further study the difference of these three policies, we also compared their runtime throughput in Table 2.3. We can see that the static policy has the lowest throughput (198,076 ops/sec). The heuristic and queuing theory policies can deliver higher throughput, 223,146 and 229,956 ops/sec, respectively.

2.5.5 Overhead Analysis

DIDACache is highly optimized for key-value caching and moves certain device-level functions up to the application level. This could raise consumption of host-side resources, especially memory and CPU.

Memory Utilization: In DIDACache, memory is mainly used for three purposes. (1) In-memory hash table. DIDACache maintains a host-side hash table with 44-byte mapping entries (`<md, sid, offset>`), which is identical to the stock Fatcache. (2) Slab buffer. DIDACache performance is insensitive to the slab buffer size. We use a 128MB memory for slab buffer, which is also identical to the stock Fatcache. (3) Slab metadata. For slab allocation and GC, DIDACache introduces two additional queues (*Free Slab Queue* and *Full Slab Queue*) for each channel. Each queue entry is 8 bytes, corresponding to a slab. Each slab also maintains an erase count and a valid data ratio, each requiring 4 bytes. Thus, in total, DIDACache adds 16-byte metadata for each slab. For a 1TB SSD with a regular slab size of 8MB, it consumes at most 2MB memory. In our experiments, we found that the memory consumptions of DIDACache and Fatcache are almost identical during runtime. Also note that the device-side demand for memory is significantly decreased, such as the removed FTL-level mapping table.

Table 2.4: CPU utilization of different schemes.

Scheme	SET	GET	SET/GET (1:1)
DIDACache	47.7%	20.5 %	37.4 %
Fatcache-Async	42.3 %	20 %	33.8 %
Fatcache-Sync	40.1 %	20 %	31.3 %

CPU utilization: DIDACache is multi-threaded. In particular, we maintain 12 threads for monitoring the load of each channel, one global thread for garbage collection, and one load-monitoring thread for determining the OPS size. To show the related computational cost, we compare the CPU utilization of DIDACache, Fatcache-Async, and Fatcache-Sync in Table 2.4. It can be observed that DIDACache only incurs marginal increase of the host-side CPU utilization. In the worst case (100% `SET`), DIDACache only consumes extra 7.6% and 5.4% CPU resources over Fatcache-Sync (40.1%) and Fatcache-Async (42.3%), respectively. Finally it is worth noting that DIDACache removes much device-level processing, such as GC, which simplifies device hardware.

Cost implications: DIDACache is cost efficient. As an application-driven design, the device hardware can be greatly simplified for lower cost. For example, the DRAM required for the on-device mapping table can be removed and the reserved flash space for OPS can be saved. At the same time, our results also show that the host-side overhead, as well as the additional utilization of the host-side resources are minor.

2.6 Discussion on Extreme Conditions

Due to hardware constraint, some extreme cases are not triggered in our experiment. In this section, we will discuss the cache performance on some extreme conditions. We model the working procedure and analyze the performance of `SET` and `GET` operations, which are the two typical operations for key-value cache system. We breakdown and compare request latency for both the conventional key-value caching design and DIDACache.

• **SET Operation**: In both DIDACache and the conventional key-value caching, `SET` operations are served in an asynchronous way. When a `SET` operation comes, it will be firstly served by a memory slab. If the key-value item is stored in memory slab, the request can be returned, and the full memory slabs are flushed to flash in background as described in algorithm 2.4.1. So, in the best case, one key-value item `SET` operation only consists of one hash index build operation and one memory store operation. The request latency can be presented as:

$$t_{SET} = t_{hash} + t_{wmem}. \tag{2.4}$$

In here, t_{hash} and t_{wmem} stand for the hash index build time and memory store time, respectively.

However, in the worst case, the memory slab buffer and flash slabs are consumed very fast, which may cause incoming requests wait for the flash write and GC process synchronously. For DIDACache, in the worst case, the incoming `SET` request needs to wait for one flash block write operation and one integrated GC process. DIDACache adopts the *quick clean* scheme which directly erases the victim block without copying data; so when the system is starving for space, the integrated GC process only incurs one flash block erase operation. In the worst case, the request latency for `SET` operation can be denoted as:

$$t_{SET} = t_{hash} + t_{wmem} + t_{fwrite} + t_{erase}. \tag{2.5}$$

In here, t_{fwrite} is the time for one flash block write operation, t_{erase} is the time consumed by erasing one flash block.

In contrast, for conventional key-value caching, when the worst case happens, the serving process of one `SET` operation can be separated into software part and hardware part. From the software aspect, the request needs to wait for one software level GC process to reclaim cache space. From the hardware aspect, the request needs to wait for one slab flush operation and one hardware GC process to reclaim flash blocks. For the slab flush operation, the conventional SSD will slice one slab into stripes and flush the data to all its channels in parallel. Suppose the SSD contains N channels, and the time for one slab flush operation

is t_{fwrite}/N. In the worst case, when hardware GC happens, each flash block contains one invalid flash page. If each block has m flash pages, to reclaim one flash block, the SSD needs to copy $m(m-1)$ flash pages, and erase m flash blocks. So the latency for one hardware GC process can be $t_{fwrite} \times (m-1)/N + t_{erase} \times m/N$. Thus, in the worst case, the request latency for `SET` operation is:

$$t_{SET} = t_{hash} + t_{wmem} + t_{sgc} + t_{fwrite} \times (m-1)/N + t_{erase} \times m/N. \tag{2.6}$$

Here, t_{sgc} is the time consumed by software level GC process. In the worst case, the software level GC process needs to copy $S_{slab}/S_{KV} - 1$ key-value items.

• **GET Operation**: Basically, the working procedure for `GET` operations of DIDACache and the conventional key-value caching are the same. For a `GET` operation, the caching system will firstly look up its in-memory index. If the corresponding key-value item is in memory, the data can be returned by a memory load operation. Otherwise, the system needs to read the data from SSD flash. The difference is that in DIDACache, when reading data from SSD flash, it does not need to use address mapping model to translate the logical disk slab number to flash pages. The time consumption for one `GET` operation in the conventional key-value caching is:

$$t_{GET} = \begin{cases} t_{rhash} + t_{rmem} & \text{if } the\ KV\ item\ is\ in\ memory\ slab \\ t_{rhash} + t_{mapping} + t_{fread} & \text{if } the\ KV\ item\ is\ in\ disk\ slab \end{cases} \tag{2.7}$$

Here, t_{rhash} represents the time consumed by searching the in-memory hash table. $t_{mapping}$ denotes the time consumed by FTL address mapping model, and t_{fread} is the time for flash page read operation. When the key-value item is in memory slab, it can be returned by just one hash table search and one memory load operation. Otherwise, if the key-value item is in disk slab, the latency is composed of a hash table search operation, an SSD address mapping search operation, and a flash page read operation.

For DIDACache, the time consumption for one `GET` operation is:

Table 2.5: Key-value (256Bytes) request latency on extreme conditions.

Key-value Caching	Best Case		Worst Case	
	SET Latency	GET Latency	SET Latency	GET Latency
DIDACache	1 us	1 us	0.363 s	370 us
Conventional	1 us	1 us	15.492 s	370 us

$$t'_{GET} = \begin{cases} t_{rhash} + t_{rmem} & \text{if } the\ KV\ item\ is\ in\ memory\ slab \\ t_{rhash} + t_{fread} & \text{if } the\ KV\ item\ is\ in\ disk\ slab \end{cases} \tag{2.8}$$

Similar to the conventional key-value caching, if the key-value item is in memory slab buffer, the latency for GET request is also $t_{rhash} + t_{rmem}$. But if the key-value item is in disk slab, the latency just include one hash table search and a flash page read operation. To conclude, for both best case and worst case, the latency for GET operation of DIDACache and the conventional key-value caching are basically the same.

Table 2.5 shows an example of latencies for SET and GET request on two extreme cases with our experimental hardware configuration. Due to space constraint, we only show the results with key-value item size of 256 bytes. Key-value items of other sizes have the same trend. For a SET request, in the best case, its latency only includes one index build operation and one memory store operation. In our experiments, for both DIDACache and the conventional key-value caching, the shortest latency is around 1 us. In our experiment, the conventional SSD contains 12 channels and each block has 512 pages, and each slab is 8MB. The time for writing and erasing one block are 0.356s and 7 ms respectively. The time granularity for t_{hash} and t_{wmem} are in us, which can be ignored in comparison. With equations 2.5 and 2.6, we get that the worst latency for one SET request in DIDACache is 0.363 s, and the worst latency for one SET request of conventional key-value caching is 15.492 s. For a GET request, DIDACache and the conventional key-value caching have quite similar working procedure. In our experiment, the shortest latency for both DIDACache and the conventional key-value caching is 1 us. In the worst case, the main bottleneck for the GET request latency is the raw flash read performance, and it is about 370 us.

2.7 Other Related Work

Both flash memory [9, 17–19, 28, 39, 63, 73, 81, 83, 102, 110, 117, 127] and key-value systems [10, 11, 26, 37, 75, 79, 126, 130] are extensively researched. This section discusses prior studies most related to this thesis.

A recent research interest in flash memory is to investigate the interaction between applications and underlying flash storage devices. Yang et al. investigate the interactions between log-structured applications and the underlying flash devices [129]. Differentiated Storage Services [88] proposes to optimize storage management with semantic hints from applications. Nameless Writes [133] is a de-indirection scheme to allow writing only data into the device and let the device choose the physical location. Similarly, FSDV [134] removes the FTL level mapping by directly storing physical flash addresses in the file systems. Multi-stream SSD [53] maintains multiple write streams with different expected lifetime for SSD. Applications write to different streams according to data lifetime. This design aims to make the NAND capacity unfragmented and handle the GC without costly data movement. Although sharing a similar principle of leveraging application semantics for efficient device management, DIDACache aims to bridge the semantic gaps between application and the underlying hardware and is specific for key-value cache systems. For example, DIDACache leverages the properties of key-value cache for aggressive quick-clean without incurring a problem.Willow [109] exploits on-device programmability to move certain computation from the host to the device. FlashTier [107] uses a customized flash translation layer optimized for caching rather than storage. OP-FCL dynamically manages OPS on SSD to balance the space needs for GC and for caching [94]. Some commercial SSDs allow users to define their own OPS space, such as Samsung 840 Pro [106]. However, these SSDs only allow applications to adjust the OPS space statically, and the OPS space cannot be dynamically adjusted according to the applications' runtime patterns. Our DIDACache dynamically determines the minimum reserved space for OPS purpose and maximizes the usable cache space during the runtime according to the application workload pattern. RIPQ [121] optimizes the photo caching in Facebook particularly for flash by reshaping the small random writes to a flash-friendly workload. FlashBlox [46] proposes to utilize flash parallelism to improve

isolation between applications by running them on dedicated channels and dies, and balance wear within and across different applications. LightNVM [13] is an open-channel SSD subsystem in the Linux kernel, which introduces a new physical page address I/O interface that exposes SSD parallelism and storage media characteristics. Our solution shares a similar principle of removing unnecessary intermediate layers and collapsing multi-layer mapping into only one, but we particularly focus on tightly connecting key-value cache systems and the underlying flash SSD hardware.

Key-value cache systems recently show their practical importance in Internet services [11, 37, 79, 130]. A report from Facebook discusses their efforts of scaling Memcached to handle the huge amount of Internet I/O traffic [92]. McDipper [32] is their latest effort on flash-based key-value caching. Several prior research studies specifically optimize key-value store/cache for flash. Ouyang et al. propose an SSD-assisted hybrid memory for Memcached in high performance network [97]. This solution essentially takes flash as a swapping device. Flashield [29] is also a hybrid key-value cache which uses DRAM as a "filter" to minimize writes to flash. NVMKV [82] gives an optimized key-value store based on flash devices with several new designs, such as dynamic mapping, transactional support, and parallelization. Unlike NVMKV, our system is a key-value cache, which allows us to aggressively integrate the two layers together and exploit some unique opportunities. For example, we can invalidate all slots and erase an entire flash block, since we are dealing with a cache rather than storage.

Some prior work also leverages open-channel SSDs for domain optimizations. Our prior study [113] outlines the key issues and a preliminary design of flash-based key-value caching. Ouyang et al. present SDF [96] for web-scale storage. Wang et al. further present a design of LSM-tree based key-value store on the same platform, called LOCS [125]. KAML [49] presents a key-addressable multi-log SSD which exposes a key-value interface to enable applications to make use of internal parallelism of flash channels through using open-channel SSD. We share the common principle of bridging the semantic gap and aim to deeply integrate device and key-value cache management.

2.8 Summary

Key-value cache systems are crucial to low-latency high-throughput data processing. In this chapter, we present a co-design approach to deeply integrate the key-value cache system design with the flash hardware. Our solution enables three key benefits, namely a single-level direct mapping from keys to physical flash memory locations, a cache-driven fine-grained garbage collection, and an adaptive over-provisioning scheme. We implemented a prototype on real open-channel SSD hardware platform. Our experimental results show that we can significantly increase the throughput by 35.5%, reduce the latency by 23.6%, and remove unnecessary erase operations by 28%.

Although this chapter focuses on key-value caching, such an integrated approach can be generalized and applied to other semantic-rich applications. For example, for file systems and databases, which have complex mapping structures in different levels, our unified direct mapping scheme can also be applied. For read-intensive applications with varying patterns, our dynamic OPS approach would be highly beneficial. Various applications may benefit from different policies or different degrees of integration with our schemes. As our future work, we plan to further generalize some functionality to provide fine-grained control on flash operations and allow applications to flexibly select suitable schemes and reduce development overheads.

CHAPTER 3

ONE SIZE NEVER FITS ALL: A FLEXIBLE STORAGE INTERFACE FOR SSDS

3.1 Introduction

Solid State Drives (SSDs) are becoming the mainstream secondary storage in various computing systems. Applications typically access SSDs through the standard block I/O interface. To be general and versatile, the block interface simply exports the storage as a sequence of logical blocks. Unfortunately, the resulting semantic gap between applications and devices causes numerous significant, well-known issues, such as 'log-on-log' [80, 129] and high tail latency [24, 43, 127].

A recent industry trend directly opens the SSD hardware details to the upper-level applications [13,96,109,114,125]. Open-channel SSD is one such representative and popular example [13, 96, 114, 125].

With open-channel SSD, the physical layout details (e.g., channels, chips, and blocks) are directly exposed to applications, which manage them via direct access to the core flash operations—page-read, page-write, and block-erase. This low-level control allows applications to optimize their performance through a tight integration of software and hardware. At the same time, it introduces significant challenges into software development. For example, a strong expertise in SSD hardware is required from application developers; the development process becomes more complex, involving both software and hardware in debugging and testing; application optimizations become ad-hoc and hardware dependent. These issues greatly limit the portability of software code, as one deployment case may not be immediately applicable to other hardware platforms.

Currently, developers must choose between these two extreme usage modes, neither of which is ideal. They can adopt the easy-to-use block interface, but suffer the long-term consequences on their application's performance, or directly control and optimize every aspect of their SSDs by taking on excessive development burden. In reality, many application types and development scenarios call for a finer-grained compromise. For example, a developer may wish to parallelize I/Os but not be interested in explicit control of garbage collection (GC). Existing interfaces do not allow developers to make a balance between development cost and performance.

To support the versatile needs of such applications, we present *Prism-SSD*–a flexible storage interface. Prism-SSD exports the SSD via a user-level library in three levels of abstraction, allowing software developers to interact with the SSD hardware in different degrees of detail and complexity: (1) A *raw-flash level abstraction*, which directly exposes the low-level flash details, including physical structures and core flash operations to applications; (2) A *flash-function level abstraction*, which presents the SSD as a group of flash management functions that can be scheduled and custom-defined by applications, such as GC, wear-leveling, etc.; (3) A *user-policy level abstraction*, which presents flash hardware as a block device that is configurable by selecting predefined high-level policies. At the bottom of Prism-SSD, the user-level flash monitor, which is a user-level module, runs as a core daemon and connects the library to the OS kernel-level device driver. This flash monitor is also responsible for allocating and managing physical flash capacity for applications that share the same SSD hardware.

Unlike some host-side FTL schemes [51, 87], Prism-SSD provides the storage abstraction at the user library level. This holds several advantages. First, it provides application developers a familiar programming interface, to easily interact with the flash storage. Second, the user-level library resides between the application layer and the OS kernel, which allows it to bypass the intermediate OS kernel components, such as file systems, and directly communicate with the device via `ioctl`. Third, it does not require any kernel-level changes, thus ensuring portability across different hardware platforms.

To demonstrate the versatility and performance of this new storage model, we implemented a Prism-SSD prototype on the open-channel SSD platform. We enhanced the I/O module of three representative applications, using each of the three abstraction levels provided by the Prism-SSD library. We modified a key-value cache based on Twitter's Fatcache [123], a user-level log-structured file system based on Linux FUSE [89], and a graph computing engine based on GraphChi [65]. Together, these three use cases well demonstrate the flexibility and efficiency of our model. Our results show that Prism-SSD allows developers to flexibly choose the most suitable storage abstraction for optimizing their applications, at different tradeoff points between performance and development cost.

Our main contributions are as follows. (1) We propose a highly flexible system interface, designed as a user-level library, for interacting with flash-based SSDs in varying layers of abstraction. (2) We present a fully functional prototype of Prism-SSD on the real open-channel hardware platform, which will be made available as an open-source project. (3) We demonstrate the efficacy of our approach in three use cases, with a range of development costs and performance benefits.

The rest of this chapter is organized as follows. Section 3.2 gives the background for this work. Section 3.3 describes our goals, with the design details in Section 3.4 and a discussion in Section 3.5. We present our prototype in Section 3.6, and the evaluation use cases in Section 3.7. We discuss related work in Section 3.8, and conclude in Section 3.9.

3.2 Background

This section briefly introduces the background of traditional flash SSDs and open-channel SSDs.

Generic flash SSDs. Conventional flash SSDs typically encapsulate an array of flash memory chips, providing a generic block I/O interface to the host. An SSD controller, as a major component of SSDs, is used to process I/O requests, and manage flash memory by issuing commands to the *flash memory controller*. A *Flash Translation Layer* (FTL) is usu-

ally implemented in the device firmware to manage flash memory and hide its complexities (e.g., *sequential write*, *out-of-place overwrite* constraints) behind the *Logical Block Address (LBA)* interface. An FTL mainly consists of three components: an *address mapping table* translating logical addresses to flash physical pages, a *garbage collector* reclaiming invalid flash blocks, and a *wear-leveler* spreading the wear of flash blocks evenly across the chips. The details of FTL algorithms can be found in prior work [9, 22, 35].

Open-Channel SSDs. Open-channel SSDs expose their device-level details and raw flash operations directly to applications. The host is responsible for utilizing SSD resources with primitive functions through a simplified I/O stack. The following design principles of open-channel SSDs open up new prospects for SSD management. (1) Internal geometry details, such as the layout of channels, LUNs, and chips, are exposed to user-level applications. With this knowledge, applications can effectively organize their data and schedule accesses to fully exploit the raw flash performance. (2) Applications can directly operate the device hardware through the `ioctl` interface, allowing them to bypass the intermediate OS components, such as file system. (3) FTL-level functions, such as address mapping, GC, and wear-leveling, are removed from the device firmware. Instead, applications are responsible for dealing with flash physical constraints. For example, applications are responsible for allocating physical flash pages, ensuring a block being erased before it is overwritten. Thus, it can avoid issues, such as the 'log-on-log' problem [129], by directly issuing commands to erase physical blocks [114].

3.3 Design Goals

Open-channel SSDs have been deployed with various kinds of applications, such as file systems [131], key-value stores and caches [114, 125], and virtualization environments [46], where they help achieve significant performance improvements. However, the prohibitive development overhead associated with open-channel SSDs hinders them from a much wider adoption, especially by applications that require special but only minor deviations from the standard block I/O interface (e.g., erase a block). This unrealized potential motivates our

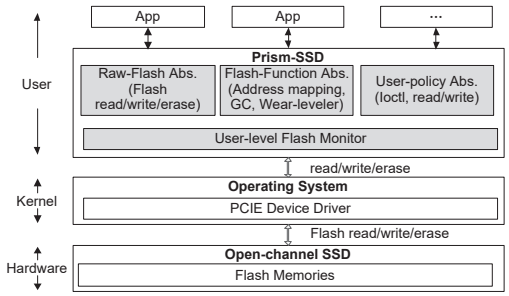


Figure 3.1: Overview of Prism-SSD architecture.

new model of a flexible storage interface for SSDs. We build our model, Prism-SSD, with the following design goals.

• *Flexibility*: Applications should be able to flexibly choose the degree of control they require over the SSD operations. We achieve this goal by providing multiple levels of abstraction for programmers to choose from.

• *Generality*: The application design, and most of the library implementation, should be general and portable to different hardware and OS platforms. We achieve this goal by encapsulating the low-level flash accesses within the user-level flash monitor, and decoupling it from the user-perceived storage abstraction.

• *Efficiency*: Applications should experience minimal overhead as a result of using our library. We achieve this goal by implementing it in user-space, thus bypassing most of the kernel's I/O stack and the latencies it entails.

By following these goals, our proposed flexible storage interface is designed to provide users a fine-grained control on the tradeoff between performance and development cost, while incurring minimal overhead.

3.4 The Design of Prism-SSD

Figure 3.1 depicts the architecture of Prism-SSD, which consists of three main components: (1) *A specialized flash memory SSD hardware*, which exposes the physical details of flash memory and opens low-level direct access to the flash memory media via the user-level library. (2) *A user-level abstraction library*, which provides a comprehensive set of storage

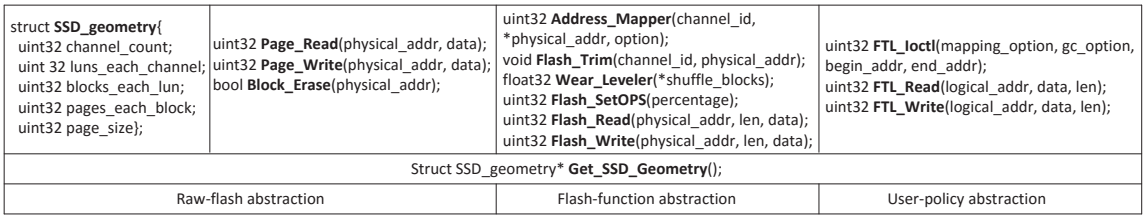


Figure 3.2: APIs of Prism-SSD.

I/O stack abstractions to allow application developers to choose the most suitable way of interacting with the underlying flash hardware through different APIs. (3) *Applications*, which customize their software design with flash memory management at different degrees of integration through the library's abstraction interface, so as to exploit the properties of the underlying devices.

Prism-SSD bypasses multiple intermediate layers in the conventional storage I/O stack, such as the file system, generic block I/O, the scheduler, and the FTL in the SSD firmware. We allow applications to leverage their domain knowledge, while controlling only those aspects of the underlying flash device that are absolutely necessary for exploiting this knowledge. The other aspects should be transparently handled by the library.

The flexibility in software and hardware co-design is provided in Prism-SSD by a user-level library, shown in Figure 3.1. The library includes three sets of abstraction APIs with different degrees of hardware exposure to applications: a raw-flash level, a flash-function level, and a user-policy level. The library also includes a flash monitor running as a user-level module, which is responsible for allocating flash capacity to different applications sharing the same SSD hardware, and for isolating them from one another [46]. Figure 3.2 presents the APIs of the three abstraction levels.

The full documentation of the each abstrtaction layer and its APIs will be made available as an open-source project along with our prototype. Below, we focus on their details required for demonstrating our flexible I/O interface approach. We note, however, that the specific three-layer design presented below is only one of many possible realizations of our approach. Other designs may define different or additional abstraction layers to allow for finer-grained tradeoff points between developement cost and performance.

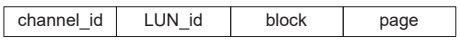


Figure 3.3: The physical address format.

3.4.1 The User-level Flash Monitor

At the bottom layer of the Prism-SSD library is a user-level flash monitor. Its main role is two-fold. First, as a storage capacity manager, it allocates the required flash capacity to applications and ensures space isolation. Second, it is responsible for sharable services, such as OPS allocation, global wear-leveling, bad block management, etc.

Applications request storage space through the user-level flash monitor. The monitor uses LUNs as the basic allocation unit[1] for satisfying applications' capacity requirements. Prism-SSD allocates LUNs in a round-robin fashion across channels. Consider an open-channel SSD with 12 channels, each providing access to 16 LUNs of 1GB. If an application requests a capacity of 24GB, the device manager will allocate two LUNs from each channel. The monitor also allocates an amount of over-provisioning space (OPS) as specified by the application. The developer can determine the size of OPS based on the application's properties. For write-intensive applications, the OPS can be set larger (e.g., 25%, similar to a typical high-end SSD); for read-intensive applications, the percentage can be smaller. The over-provisioning space is also allocated in units of LUNs. In the above example, for an OPS of 25%, six extra flash LUNs will be allocated to the application. As the flash monitor tracks the channels and LUNs allocated to each application, the flash capacity of different applications is completely isolated. Applications access the flash space allocated to them using the address format as shown in Figure 3.3.

Workload patterns of different applications may vary considerably, causing the erase counts of flash blocks in different channels to vary as well. To handle uneven wear of the flash device, the design of Prism-SSD includes a global wear leveling module, which is based on FlashBlox [46]: Global wear leveling is applied in LUN granularity—it calculates the average erase count of each LUN to distinguish between 'hot' and 'cold' LUNs. If the

[1] An open-channel SSD usually consists of several *channels*, each providing access to multiple *LUNs*, which are the smallest parallelization unit [19]. Each LUN includes multiple flash blocks.

difference in average erase counts exceeds a threshold, a hot LUN will be shuffled with a cold LUN, and their allocation to applications will be updated. This module is not implemented in our current prototype.

For bad block management, the flash monitor maintains a list of blocks that are detected faulty and marked ineligible, hiding them from applications.

3.4.2 Abstraction 1: Raw-Flash Level

The raw-flash level abstraction of Prism-SSD exposes the device geometry and allows applications to control the low-level flash hardware. Applications can directly operate on flash pages or blocks through page read/write and block erase commands. Meanwhile, application developers should be fully aware of the unique characteristics of flash memories, such as the out-of-place update constraint, to operate the device correctly.

With this abstraction level, typical FTL functions, such as address mapping, GC and wear-leveling, are not provided by the library. Whether to implement them or not depends on the application's requirements. The application should also be responsible for its own flash space allocation and management, and for integrating them with its software semantic. The library simply delivers function calls from applications to the device driver through the `ioctl` interface.

Figure 3.2 shows the APIs provided by the raw-flash level abstraction. `Get_SSD_Geometry` returns the SSD layout information to the application. The SSD layout is described by the number of channels, LUNs in each channel, blocks in each LUN, pages in each block, and the page size. This layout information is exposed to all abstraction layers via the same interface. Applications use `Page_Read` and `Page_Write` to directly read and write flash physical pages, and `Block_Erase` to erase a specified block.

The raw-flash abstraction gives applications full knowledge and direct control of the low-level flash device, at the cost of considerable development effort. The applications that will likely benefit most from this abstraction are those with special, regular, and well-defined

access patterns, such as large-chunk writes. The low-level details and control exposed by this layer are similar to those exposed by existing low-level interfaces. However, it differs from these interfaces in two main aspects. Its APIs are decoupled from any specific SSD hardware, providing an additional degree of portability for developers. In addition, unlike existing stand-alone interfaces, the raw-flash layer is provided within a multi-layer flexible framework. This allows developers to build their applications with the same hardware but with higher layers of abstractions, while being aware of the layout details exposed by the lowest layer.

3.4.3 Abstraction 2: Flash-Function Level

Our flash-function level abstraction models the flash storage as a collection of *core functions* for flash management, such as GC, wear-leveling, etc. Application developers can compose them and implement more sophisticated and complex management tasks. Thus, they can maintain a certain low-level control, while avoiding the need to handle other irrelevant details of the SSD hardware. Figure 3.2 shows the core APIs of the flash-function level. These APIs are used to divide the four main components of flash management between the application and the library, as follows.

Space allocation. At this level, applications directly read and write flash physical addresses via `Flash_Read` and `Flash_Write`, while the library is responsible for erasing blocks and for allocating them to applications. The application requests physical blocks via `Address_Mapper`, specifying the channel in which the block should be allocated, and the mapping scheme (i.e., page-level or block-level) for that block. It then maps the physical address returned by the library to an application-managed logical address. This function call returns the amount of free space available for the application, allowing the application to invoke garbage collection according to its needs.

Garbage collection (GC). At this level, the application is responsible for selecting the victim blocks for GC, and for identifying the valid data on this block. The granularity of the valid data is determined by the application and can be as small as a tuple of several hundred bytes.

Thus, the application is also responsible for copying the valid data to a new location. By calling the `Flash_Trim` command, the application notifies the library that a block is ready to be erased, in the background, and reallocated.

Wear-leveling. At this level, the application manages the logical-to-physical block mapping while the library maintains the blocks' erase counts. Thus, wear leveling is triggered by the application and executed by the library, as follows. The application invokes the `Wear_Leveler` in a suitable time. The library identifies the hottest blocks and the coldest ones, and swaps the data written on them. It returns these block addresses via the "shuffle_block" parameter, as well as the maximum variance between erase counts of the application's allocated blocks. The application then updates its mapping of the two blocks, and potentially invokes another wear leveling operation according to its target variance.

OPS management. The application can dynamically determine the over-provisioning space it requires according to its current workload via `Flash_SetOPS`. The library reserves the specified OPS for this application. The library cannot provide the requested OPS if too many blocks are currently mapped by the application. In this case, the application must first release sufficient flash space.

Algorithm 3.4.1 shows an example of block allocation and garbage collection implemented with the flash-function level. In this simple example, the application requests 10 flash blocks in an idle channel by repeatedly calling the address mapping function with block-level address mapping ("Block"). This function call returns the number of free blocks currently available in this channel. If the available free space is below a predefined threshold, the application triggers an application-controlled background GC process in this channel. The GC process selects a victim block, copies its valid data elsewhere, and releases this block for erasure by the library.

The flash-function level abstraction exports basic flash functions that application developers can use to configure different flash-management policies and to invoke them at the most suitable timing according to their current workload. At the same time, it hides the low-level device details, such as LUNs and erase counts, from the application level. This

abstraction level is suitable for applications that can leverage their software semantics for specific optimizations but are not willing to handle the low-level management details. To the best of our knowledge, it is the first general-purpose implementation to provide this fine-grained tradeoff between application management and ease-of development.

3.4.4 Abstraction 3: User-policy Level

The user-policy level abstraction hides all the flash related management operations from users, allowing them to manage the SSD as a simple block device. To some extent, it can be regarded as a user-level FTL that handles address mapping, GC, wear-leveling, etc. This abstraction level is designed to provide the highest generality for SSDs. However, unlike conventional device-level FTLs, this "FTL" runs as part of the user-level library and is configurable, allowing applications to select their preferred policies for managing and allocating flash capacity.

The applications use their semantic knowledge about the data usage patterns to choose the best policies for optimizing their specific objectives. Thus, these configuration parameters serve as application 'hints' to the FTL. At the same time, the full device layout information is exposed to applications, allowing them to optimize the size of their data structures or level of I/O parallelism for the underlying device.

Figure 3.2 lists the APIs provided in the user-policy level abstraction. Logical addresses are read and written via the `FTL_Read` and `FTL_Write` block I/O interface commands. Applications configure the key flash management policies, address mapping and garbage collection, via the `FTL_Ioctl` function. The same policies implemented in the flash-function level (see Section 3.4.3) are available for selection.

The user-policy level abstraction is similar to existing host-level FTLs. However, it is managed as part of a general-purpose user-level library which also exposes additional abstraction layers. Furthermore, it allows application developers to leverage their semantic knowledge to configure the FTL policies. This abstraction level requires the lowest integration overhead, which makes it suitable for applications that only demand certain hard-

Algorithm 3.4.1 Example of block allocation and GC implemented with the flash-function level.

```
1: FBN; //free block space
2: PBN; //physical address
3: LBN; //logical address
4: len ← 10 × Blocksize; //length
5: while len > 0 do
6:     CH_id ← choose a channel with the least workload;
7:     FBN ← Address_Mapper(CH_id, &PBN, "Block");
8:     LBN ← PBN; // map logical to physical
9:     //allocate physical block in channel CH_id
10:    if FBN < GC_Threshold then
11:        //Free space is under a GC threshold
12:        APP_GC(CH_id);
13:    end if
14:    len ← len − 1;
15: end while
16: function VOID APP_GC(CH_id)
17:    while FBN < GC_Threshold do
18:        PBN_victim ← victim block in CH_id;
19:        //select block by "Greedy", "LRU", etc.
20:        //copy valid data from the victim block elsewhere
21:        Flash_Trim(CH_id, PBN_victim);
22:    end while
23: end function
```

ware/software cooperation through a configurable interface, but are sensitive to their development cost.

3.5 Discussion

The introduction of a new, flexible storage interface holds great potential for application development and optimization, as well as for quick adoption of new hardware. By realizing this new model within a user-level library—unlike the traditional approaches that implement the complex flash management policies in the device firmware [9] or the OS kernel driver [51, 56]—Prism-SSD delivers this potential to general developers and hardware vendors.

We discussed the perspective of the application developers in detail above and demonstrate it in our use-cases below. Prism-SSD presents them with the first opportunity to choose from a range of abstraction layers for interacting with their hardware. They may further leverage its APIs to develop and stack additional libraries for different languages and applications.

From the perspective of hardware vendors, a flexible storage interface in the form of a user-level library is a powerful tool for reducing development costs and the time-to-market. Moving complex internal firmware into the library layer will allow hardware vendors to quickly roll out new features in the form of library updates, and to accelerate the development cycle thanks to reduced coding, debugging, and testing requirements at the user level. These advantages are particularly appealing with the increasing complexity of hardware storage devices.

Finally, hardware vendors can easily offer custom-built hardware/software solutions addressing various applications' requirements. Such solutions are currently prohibitive in terms of development time and costs. Combined, these advantages offer hardware vendors the means to build and own a complete vertical stack to closely connect with applications, creating more business opportunities.

3.6 Implementation and Prototype System

We have built a prototype of Prism-SSD on the open-channel SSD hardware platform manufactured by Memblaze [85]. This PCI-E based SSD contains 12 channels, each of which connects to two Toshiba 19nm MLC flash chips. Each chip consists of two planes and has a capacity of 66GB. The SSD exports its physical space to the upper level as one volume, with access to 192 LUNs. The 192 LUNs are evenly mapped to the 12 channels in a channel-by-channel manner. That is, channel #0 contains LUNs 0-15, channel #1 contains LUNs 16-31, and so on. Thus, the physical mapping of flash memory LUNs to channels is known.

This interface is different from that of open-channel SSDs used in other studies, which exports flash space as 44 individual volumes [96]. The hardware used in our prototype allows the upper level to directly access raw flash memory via the `ioctl` interface, by specifying the LUN ID, block ID, or page ID in the commands sent to the device command queue. Standard FTL-level functions, such as address mapping and GC, are not provided. In Prism-SSD, they are implemented in the library. The user-level flash monitor module is responsible for conveying the I/O operations to the device driver via `ioctl`.

Our prototype implements the user-level flash monitor and the three abstraction levels, accounting for 4,460 lines of C code. Specifically, the user-level flash monitor module accounts for 560 lines, the raw-flash abstraction for 380 lines, and the flash-function abstraction and the user-policy abstraction for 2,580 and 940 lines, respectively. We deployed our prototype on a Linux workstation, which features an Intel i7-5820K 3.3GHZ processor and 16GB memory. We use Ubuntu 14.04 with Linux kernel 3.17.8 as our operating system.

3.7 Case Studies

The Prism-SSD model offers a powerful tool for developers to optimize their performance with SSDs. Choosing the abstraction level that best suits their needs, application developers can integrate their software design with the hardware management at the right balance between performance and development cost.

In this section, we demonstrate the versatility of this approach in typical software development scenarios, mainly from the perspective of developers. We carefully selected three major applications as our case studies: a key-value cache system based on Twitter's Fatcache (Section 3.7.1), a user-level log-structured file system based on Linux FUSE (Section 3.7.2), and a graph computing engine based on GraphChi (Section 3.7.3). Due to space constraint, we use key-value caching as our main case study and the other two as examples demonstrating the general applicability of Prism-SSD.

3.7.1 Case 1: In-flash Key-value Caching

Recently, flash-based key-value cache systems have raised high interest in industry, such as Facebook's McDipper [32] and Twitter's Fatcache [123]. The flash-based key-value cache systems typically run on commercial flash SSDs and adopt a slab-based allocation scheme, as Memcached [86], to manage key-value pairs in flash. Taking Fatcache as an example, Fatcache divide the SSD space into large slabs (e.g., 1MB), slabs are further separated into slots. Each slot is used to store one key-value item. An in-memory hash table is used to record the mapping between key-value items and slabs.

Flash-based key-value cache systems are designed with the awareness of the underlying SSD devices, and has several flash-friendly properties. First, SSDs are treated as a log-structured object store, and I/O operations are done with large `slab` unit. For example, to accommodate small key-value items , Fatcache maintains an in-memory slab to buffer small items in memory first and then flush to storage in bulk later, which causes a unique "large-I/O-only" pattern on the underlying flash SSDs. Second, flash-based key-value caches cannot update key-value items in place. In Fatcache, all key-value updates are written to new locations. Thus, a GC procedure is needed to clean/erase slab blocks. Third, the management granularity in flash-based key-value caches is much coarser. For example, Fatcache uses a simple slab-level FIFO policy to evict the oldest slab when free space is needed.

Flash-based key-value caching represents a category of applications that are highly suitable for deep software/hardware integration. For example, Fatcache segments the SSD

space into large slabs, and all I/O operations are no smaller than 1MB, which makes low-level page-based mapping unnecessary. Fatcache never updates key-value items in place. Instead, it implements a periodic application-level GC for space recycling, which makes it a perfect candidate to merge with the device-level GC. Furthermore, Fatcache is fully aware of the slab liveness, which can serve as ideal 'hints' for scheduling erase operations.

Optimizing Fatcache with Prism-SSD. In this use case, we illustrate how to optimize Fatcache with Prism-SSD. We will demonstrate three different approaches of using the Prism-SSD library for this purpose.

• *Deep Integration*: Using the raw-flash level abstraction provided by Prism-SSD, we allow the key-value cache manager to fully exploit the semantic knowledge of Fatcache and directly drive the SSD hardware. To that end, we have augmented the key-value cache manager as described in DIDACache [114], with four major components, as follows.

(1) a slab/block management module, which directly translates slabs into one or more blocks with minimal calculation; (2) a unified hash-key-to-block mapping module, which records the mapping of key-value items to their physical flash locations; (3) an integrated garbage collection module, which reclaims flash space occupied by obsolete key-value items; and (4) a dynamic over-provisioning space (OPS) management module, which dynamically adjusts the OPS size based on a queuing-theory based model. Unlike DIDACache, our implementation uses the library's APIs, and accounts for 1,450 lines of code. Due to the space constraint, we omit the details about the DIDACache policies, which can be found in the paper [114].

• *Function-level Integration*: The second implementation, based on the flash-function level abstraction, allows the key-value cache manager to design cache-friendly policies without managing all low-level details.

In contrast to the raw-level approach, the four major components in this implementation are as follows: (1) A slab to block mapping module. At the function level, the application can still see and manage the physical flash blocks via the library APIs. Thus, it is responsible for the mapping between slabs and flash physical blocks; (2) A hash-key-to-

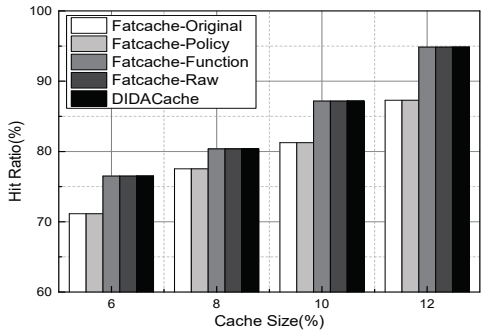


Figure 3.4: Hit ratio vs. cache size.

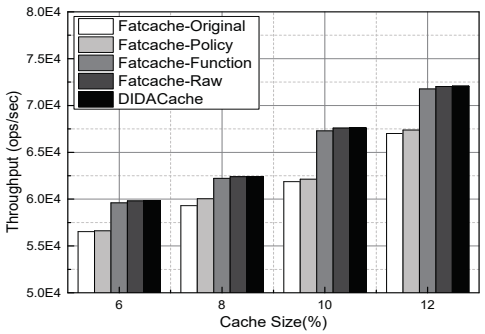


Figure 3.5: Throughput vs. cache size.

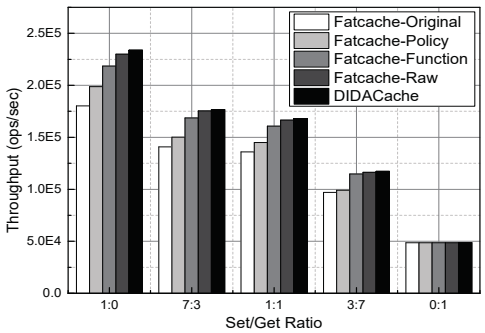


Figure 3.6: Throughput vs. `Set/Get` ratio.

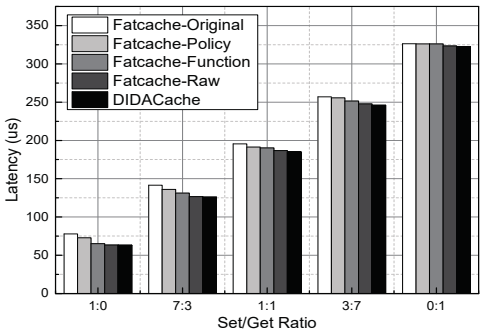


Figure 3.7: Latency vs. `Set/Get` ratio.

slab mapping module. The key-value cache manager also records the mapping of key-value items to their slab locations, which is identical to the stock Fatcache implementation. (3) A garbage collection module. The key-value cache reclaims slab space occupied by obsolete (deleted or updated) key-value items. The flash physical blocks are invalidated and recycled via the library API; (4) A dynamic OPS management module, which estimates the preferred OPS based on a queuing theory based model. Note that at this level, the slab-to-flash-block mapping is still maintained by the application, but the block allocation, reclamation, and status are maintained by the library. This implementation consists of 860 lines of code.

• *Light Integration*: For comparison, we also implemented a light-weight optimization for Fatcache by using the user-policy level abstraction. In this implementation, the key-value cache manager is nearly identical to the stock Fatcache. Our implementation only replaces the device initialization process with the library APIs. The change accounts for 210 lines of code.

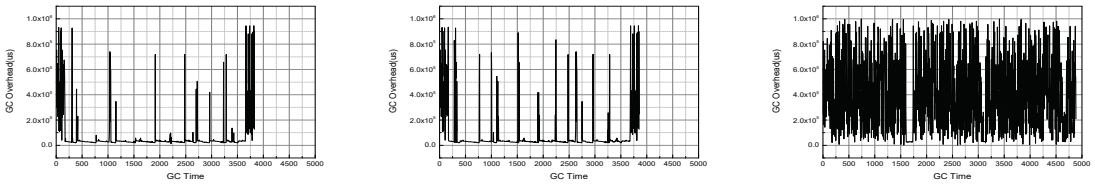
Implementation and Evaluation. Our implementation is based on Twitter's Fatcache [123].

For fair comparison, we added non-blocking slab allocation and eviction to the stock Fatcache. We use this version as our baseline and denote it as "Fatcache-Original" [1]. We refer to our implementation with the raw-flash level, flash-function level, and user-policy level abstractions as "Fatcache-Raw", "Fatcache-Function", and "Fatcache-Policy", respectively. For performance comparison, we run Fatcache-Original on a commercial PCI-E SSD, which has the same hardware as the open-channel SSD. We also show the results of DIDACache [114], denoted as "DIDACache", which directly integrates the hardware management into the Fatcache application, representing the ideal case.

• *Overall performance*. We first evaluate the key-value cache system performance in a simulated production data-center environment. This setup includes a front-end client, a key-value cache server, and an MySQL database in the backend. The key-value workload is generated using a model based on real Facebook workloads [11, 14], which is also used in prior work [114].

Figure 3.4 shows the hit ratios of the four cache systems with cache sizes of 6%–12% of the data set size. As the cache size increases, the hit ratio of all schemes improves significantly. Fatcache-Original and Fatcache-Policy have the same hit ratio because they both reserve 25% flash capacity as static OPS. In contrast, DIDACache, Fatcache-Raw, and Fatcache-Function have a higher hit ratio thanks to their adaptive OPS policy, which adaptively tunes the reserved space according to the workload, saving more space for caching. This extends the available cache space to accommodate more cache data. As a result, they outperform Fatcache-Original and Fatcache-Policy substantially: their hit ratios range between 76.5% and 94.8%, while those of Fatcache-Original and Fatcache-Policy are between 71.1% and 87.3%.

Figure 3.5 shows the throughput, i.e., the number of operations per second (ops/sec). We can see that as the cache size increases from 6% to 12%, the throughput of all the four schemes improves significantly, due to the improved cache hit ratio. Fatcache-Raw has the highest throughput, and Fatcache-Function is slightly lower. With a cache size of 10% of the data set (about 25GB), Fatcache-Raw outperforms Fatcache-Original, Fatcache-Function,



((a) Raw-Flash level. (b) Flash-Function level. (c) User-Policy level.

Figure 3.8: Garbage collection overhead with different OPS policies of three abstractions.

and Fatcache-Policy by 9.2%, 0.4%, and 8.8%, respectively.

• *Cache server performance*. In our next set of experiments, we study the performance details of the cache server. We first populate the cache server with 25GB key-value items, and then directly issue `Set` and `Get` operations to the cache server. Figure 3.6 shows the throughput of the five cache systems with different `Set`/`Get` ratios. Fatcache-Raw achieves the highest throughput across the board. Fatcache-Original achieves the lowest throughput. With 100% `Set` operations, the throughput of Fatcache-Raw is 27.6% higher than that of Fatcache-Original, 5.2% higher than that of Fatcache-Function, and 15.5% higher than that of Fatcache-Policy. The performance gain of Fatcache-Raw is mainly due to its unified slab management policy and the integrated application-driven GC policy, and the better use of the SSD's internal parallelism. Fatcache-Raw achieves almost the same performance as DIDACache. The throughput of Fatcache-Raw is only 1.7% lower than that of DIDACache in the worst case, which also demonstrates that the overhead of the Prism-SSD library is negligible compared to its benefits.

As we expected, the performances of Fatcache-Function and Fatcache-Policy are lower than that of Fatcache-Raw, which is more thoroughly optimized. The performance of Fatcache-Function is slightly lower than that of Fatcache-Raw. This is because although Fatcache-Function cannot operate with full low-level controls, it can still integrate the cache semantics within flash management, such as the GC process. Fatcache-Policy outperforms Fatcache-Original by 10.2%, due to its simplified I/O stack and block-level mapping, which reduces the overhead. As the portion of `Get` operations increases, the raw flash read latency becomes the main bottleneck, and this performance gain decreases.

Figure 3.7 shows the average latency of the five cache systems with different `Set`/`Get` ratios. Fatcache-Original suffers the highest latency, while Fatcache-Raw, implemented with Prism-SSD, has the lowest latency. For example, with 100% `Set` operations, Fatcache-Raw reduces the average latency of Fatcache-Original, Fatcache-Function, and Fatcache-Policy by 22.9%, 2.8% and 12.1%, respectively.

• *Effect of garbage collection*. In our next set of experiments, we evaluate the effect of optimized GC on the flash erase counts, which directly affect the device lifetime. We configure the available SSD size to 30GB, and preload it with 25GB data. We then issue 140M `Set` operations following the Normal distribution, writing approximately 50GB of logical data. To retrieve the erase counts of Fatcache-Original, which runs on a commercial SSD, we collect its I/O trace and replay it with the widely used SSD simulator from Microsoft Research [50]. Table 3.1 shows the GC overhead in terms of valid data copies (key-values and flash pages) and block erases of the four schemes.

Fatcache-Original suffers from the highest GC overhead. It uses the greedy GC policy and the SSD hardware uses page-level mapping. As a result, blocks selected for erasure store a mix of valid and invalid pages, incurring flash page copies by the device-level GC. In contrast, the block-level mapping used by Fatcache-Function and Fatcache-Policy maps each slab directly to one flash block, thus eliminating all page copies caused by the device-level GC. By aggressively evicting valid clean items as part of the cache management policy, Fatcache-Function, Fatcache-Raw, and DIDACache further leverage application semantics to reduce the key-value copies to only 3.63 GB, 3.49 GB and 3.45 GB, respectively.

Figure 3.8 shows the GC cost, and we can find that the GC overheads of Fatcache-Raw and Fatcache-Function are basically the same. In fact, compared with Fatcache-Policy, Fatcache-Raw and Fatcache-Function erase 15.5% and 15.2% less flash blocks, respectively. Fatcache-Policy is more affected by the GC overhead due to the lack of deep optimization.

This case study demonstrates the effectiveness of Prism-SSD by comparing four implementations of an optimized in-flash key-value caching. With the raw-flash abstraction, the developer can tightly control the low-level flash operations and optimize flash physical

Table 3.1: Garbage collection overhead.

GC Scheme	Key-values	Flash Page	Erase Count
Fatcache-Original	13.27 GB	7.15 GB	8,540
Fatcache-Policy	13.27 GB	0	7,620
Fatcache-Function	3.63 GB	0	6,017
Fatcache-Raw	3.49 GB	N/A	5,994
DIDACache	3.45 GB	N/A	5,985

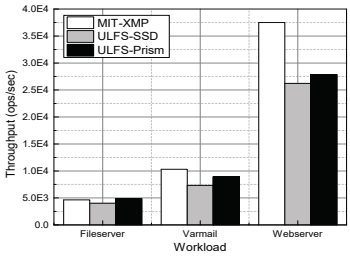


Figure 3.9: Performance evaluation.

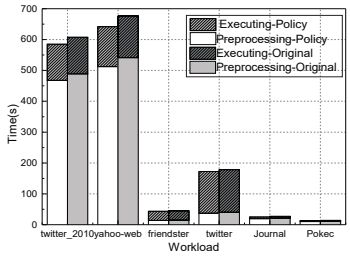


Figure 3.10: Pagerank performance.

layout. With the flash-function abstraction, the software integrates its software semantics into hardware management without handling low-level details, and the performance can be close to the raw-flash implementation. With the user-policy abstraction, the application achieves noteworthy performance gains with minimal development overhead (210 lines of code). This successfully demonstrates the flexibility of our proposed storage interface.

3.7.2 Case 2: Log-structured File System

Flash-based Log-structured file systems suffer from the well-known "log-on-log" problem [129, 131], which has detrimental effects on GC costs and performance. In this use case, we have implemented a user-level log-structured file system, called ULFS-Prism, on open-channel SSD with flash-function abstraction provided by Prism-SSD to avoid redundant logging.

ULFS-Prism directly allocates flash physical blocks to files. It maintains only the block-to-file mapping. Similarly, ULFS-Prism triggers greedy GC policy when the number of free blocks drops below a threshold, using the library's implementation of the greedy

Table 3.2: Filesystem GC overhead.

Filesystem	File copy	Flash copy	Erase
ULFS-SSD	9.82GB	7.24GB	6,594
ULFS-Prism	9.82GB	N/A	5,280
MIT-XMP	N/A	9.37GB	5,429

scheme. ULFS-Prism implements the channel-level parallelism and load balancing explicitly, by utilizing the channel information provided by the function-level abstraction. It maintains a workload queue for each channel, and counts the read/write/erase operations in each queue. A similar scheme was implemented in ParaFS [131] as a kernel-level file system on top of a specialized device-level FTL.

We also implemented a user-level log-structured file system, *ULFS-SSD*, and ran on a commercial PCI-E SSD, which has the same hardware as our open-channel SSD. For performance reference, we compare both log-structured file systems to MIT-XMP—a user level file system implemented as a FUSE wrapper for the host Ext4 file system [7] that ran on the commercial SSD.

In our first experiment, we use Filebench [2] to compare the results of the three file systems with three workloads, namely fileserver, webserver, and varmail. Figure 3.9 shows the throughput (operations per second, ops/sec) of the three file system implementations. The throughput of the two log-structured file systems is of a similar order of magnitude with MIT-XMP. ULFS-Prism outperforms ULFS-SSD in all three workloads, thanks to the cooperation between the hardware and software. Its throughput is 21.5% higher than that of ULFS-SSD on the varmail workload.

Table 3.2 shows the erase counts and valid data copied—file copies in the filesystem level and flash page copies in the device level—of each file system. ULFS-Prism and ULFS-SSD incur the same amount of file copies, but ULFS-Prism does not incur any flash page copies, because it does not require any additional device-level GC. However, with ULFS-SSD, the device-level GC may choose segments that still include valid data as victims blocks.

Table 3.3: Graphs computing workloads.

Graph Name	Nodes	Edges	Size
Twitter2010 [64]	41.7 million	1.4 billion	26.2GB
Yahooweb [8]	1.4 billion	6.6 billion	50GB
Friendster [128]	6.6 million	1.8 billion	211MB
Twitter [84]	81,306	1.8 million	44MB
LiveJournal [128]	4.0 million	34.7 million	1.1GB
Soc-Pokec [120]	1.6 million	30.6 million	404MB

MIT-XMP performs in-place updates at the file-system level, but incurs high GC cost at the device level.

This use case demonstrates that the flash-function level abstraction allows developers to quickly optimize their software design with relatively lower development cost. ULFS-SSD was implemented from scratch with 2,880 lines of code, and ULFS-Prism required only 660 more lines for integrating its flash management optimizations. This medium-level development effort is well paid off for a 21.5% speedup.

3.7.3 Case 3: Graph Computing Engine

Graph computing platforms are important for analyzing massive graph data and extracting valuable information from social, biological, health-care, information and cyber-physical systems. External memory graph processing systems, such as GraphChi [65], X-Stream [105] and GridGraph [138], are especially important, due to their scalability and low cost. In this use case, we have enhanced a popular graph computing platform, GraphChi, with the Prism-SSD library.

We modify GraphChi with Prism-SSD using the user-policy level abstraction, as a showcase of quick, light-weight integration. In the initialization process, we divide the allocated logical space into two parts, and use one to store the shard data, and the other to store the results. We divide the data of each shard into block-sized segments (instead of files),

Table 3.4: Use case summary.

Application	Abstraction Level	Code Lines	Library Services	Application Responsibilities
Key-value caching	Raw-flash	1,450	Transfer to flash operations	Slab-to-block mapping, block allocation, garbage collection, OPS management
	Flash-function	860	Block allocation, Wear leveling, Asynchronous block erase	Slab-to-block mapping, Dynamic OPS, Valid data copies
	User-Policy	210	Wear leveling, garbage collection, block allocation, block mapping	Slab allocation, item-to-slab mapping
User-level LFS	Flash-function	(2,880+) 660	Wear leveling, block erasure, block allocation, block mapping	File-to-segment mapping, segment-based garbage collection, load balancing
Graph computing	User-Policy	490	Wear leveling, garbage collection, block allocation, block mapping	Shard-to-segment mapping, logical space partitioning

and record the mapping between shards, intervals, and segments. We configure the logical space for shards with block-level mapping. The GC policy is irrelevant because this data is never updated. Similarly, we divide the result data into segments and record their mapping information in the application. We configure the logical space for result data with block-level mapping and greedy GC.

We compare the original GraphChi platform to our optimized implementation by running the "pagerank" algorithm on the graphs shown in Table 3.3. Figure 3.10 shows the total run time of both GraphChi versions on each graph, divided into preprocessing time and execution time. In general, our optimized implementation outperforms the stock GraphChi in both preprocessing and execution steps across the board. For example, on the *Soc-Journal* graph, the optimized GraphChi reduces the preprocessing and execution times of the original platform by 5.2% and 7.6%, respectively, resulting in an overall 5.7% reduction.

This performance improvement is limited compared to our previous use cases. This is mainly due to the highly optimized nature of the original GraphChi: its I/O stack is simplified and its access pattern has already been carefully optimized for SSDs. Also, I/O is not a major bottleneck in this platform, and optimizing it does not have a major impact on overall runtime. Nevertheless, we still were able to noticeably reduce this run time with a small development effort with only 490 lines of code.

3.7.4 Summary and Discussion

Table 3.4 summarizes the development cost and main characteristics of our three use cases. These case studies clearly demonstrate the flexibility and versatility provided by Prism-SSD—applications built with the raw-flash level abstraction require the most development effort, while the user-policy level abstraction requires the least code adaptation. Developers can choose how to integrate the application software with the low-level hardware management, according to their design goals.

Our case studies also show that the potential benefit from the flexible storage interface strongly depends on the application. Some applications have only little to gain from it. For example, some applications already generate flash friendly I/O traffic, such as read-only, highly parallel, or large I/Os. For these applications, applying the light-weight user-policy level abstraction will make little difference. The graph computing use case serves as a representative example for this class of applications. Another category of applications that cannot gain much benefit from this model consists of computation-intensive applications, such as typical machine learning, HPC, image processing applications. Our proposed storage model and its flexible intercase are particularly suitable for data-intensive applications with rich semantic information, such as key-value stores, user-level file servers, object stores, and virtual machines.

3.8 Related Work

Flash memory based SSDs have been extensively studied and optimized in the last decades. We focus here on the work related to their storage interface and their integration with file systems and applications.

SSDs with block I/O interface. File system and applications implemented on top of SSDs with the traditional block I/O interface often suffer from resource underutilization and low performance. Several approaches have been proposed for addressing this problem at the file-system level. For example, SFS [90] is a log-structured file system that transforms random application writes to sequential writes at the SSD level.

At the FTL level, BAST [60] combines page and block mapping granularities to allow efficient handling of both sequential and random writes, while reducing storage overhead of page-level address translation. FAST [70] further enhances random write performance with flexible mapping in log area, thus improving its utilization. CAFTL [**?**] reduces write traffic to the flash memory by eliminating duplicate writes and redundant data.

An alternative approach is to implement the FTL at the host level, allowing it to control data placement and I/O scheduling. Host-side FTLs have been implemented by both FusionIO DFS [51] and Violin Memory [87]. FSDV [**?**] relies on mapping at the device level, but assumes that the file system can learn and replace some of the mappings in the device by directly querying and storing physical flash address in the file systems.

Open-Channel SSDs. Several recent works proposed to expose some or all of the internal flash layout details directly to the application. The Open-Channel SSD used in our implementation is one such example. Another example is SDF [96], which exposes the channels in commodity SSD hardware to the software, allowing it to fully utilize the device's raw bandwidth and storage capacity. FlashBlox [46], based on Open-Channel SSD, utilizes flash parallelism to improve isolation between applications. It runs them on dedicated channels and dies, and balances wear within and across different applications.

Other designs, implemented on customized SSDs or FPGAs, followed a similar approach. ParaFS [131] exposes device physical information to the file system, which in turn exploits its internal parallelism and coordinates the garbage collection processes minimize its overhead. AMF [71] provides a new out-of-place block I/O interface, reducing flash management overhead and pushing management responsibilities to the applications.

While the semantics of the interfaces exported by these systems vary, they all export a fixed interface of the device to the application level. However, as we have demonstrated in our use cases and discussion, many applications can benefit from a flexible interface that will allow developers to balance their performance and development cost. At the same time, we believe that the designs in these works can be implemented and made portable with PrismSSD.

Smart storage. Another emerging trend is to leverage the computing capability of SSDs to offload some of the application's tasks. Kang [55] introduces a Smart SSD model, which pairs in-device processing with a powerful host system capable of handling data-oriented tasks without modifying operating system code. ActiveFlash [122] proposes offloading data analysis tasks for HPC applications to SSD controller without degrading the performance of the simulation job. Willow [109] offers programmers the ability to implement customized SSD features to support particular applications.

We view this line of research as another indication that the interface between the devices and the applications must be made more flexible, to allow users to enjoy the full power of the hardware that they own.

3.9 Summary

In this chapter, we presented a flexible storage interface, Prism-SSD, which exports SSDs to applications in three abstraction levels. This interface allows developers to choose how tightly they want to integrate flash management into their application, providing more than just the two extreme options available to developers today. We demonstrated the usability of the proposed model by comparing application performance improvement and development cost of three representative use cases. Our evaluation results reveal potential optimization opportunities that are facilitated by our model in a wide range of applications.

CHAPTER 4

AN EFFICIENT LSM-TREE-BASED SQLITE-LIKE DATABASE ENGINE FOR MOBILE DEVICES

4.1 Introduction

Smart mobile devices, e.g., smartphones, phablets, tablets, and smartTVs, are becoming prevalent. SQLite [118], which is a server-less, transactional SQL database engine, is of vital importance and has been widely deployed in these mobile devices [33, 58, 59]. Popular mobile applications such as messenger, email and social network services, rely on SQLite for data management. However, due to the inefficient date organization and the poor coordination between its database engine and the underlying storage system, SQLite suffers from poor transactional performance [47, 48, 62, 67, 69, 112].

Many efforts have been done to optimize the performance of SQLite. These optimization approaches mainly fall into two categories. (1) Investigating I/O characteristics of different workloads of SQLite and mitigating its journaling over journal problem [48, 54, 62, 69, 112]. Lee et al. [69] points out that the excessive IO activities caused by uncoordinated interactions between EXT4 journaling and SQLite journaling are one of the main sources that incur inefficiency to mobile devices. Jeong et al. [48] integrate external journaling to eliminate unnecessary file system metadata journaling in the SQLite environment. Shen et al. [112] optimize SQLite transactions by adaptively allowing database files to have their custom file system journaling mode. (2) Utilizing emerging non-volatile memory technology, such as phase change memory (PCM), to eliminate small, random updates to storage devices [57, 61, 93, 98]. Oh et al. [93] optimize SQLite by persisting small insertions or updates in SQLite with the non-volatile, byte-addressable PCM. Kim et al. [61] develop N-

VWAL (NVRAM Write-Ahead Logging) for SQLite to exploit byte addressable NVRAM to maintain the write-ahead log and to guarantee the failure atomicity and the durability of a database transaction. Though various mechanisms have been proposed, they all culminate with limited performance improvement. In this work, we for the first time propose to leverage the LSM-tree-based key-value data structure to improve SQLite performance.

Key-value database engine, which offers higher efficiency, scalability, availability, and usually works with simple NoSQL schema, is becoming more and more popular [15, 21, 25, 36]. Key-value databases have simple interfaces (such as `Put()` and `Get()`) and are more efficient than the traditional relational SQL databases in cloud environments [11, 16, 27]. To utilize the advantages of key-value database under SQL environments, Apache Phoenix [99] provides a SQL-like interface which translates SQL statements into a series of key-value operations in a NoSQL database HBase. Phoenix demonstrates outstanding performance in data cluster environment. However, the approach cannot be directly adopted by resource limited mobile devices as targeting at scalable and distributed computing environments with large datasets [20, 78].

There exist key-value databases on mobile device, such as SnappyDB [116], UnQlite [124], and Leveldown-Mobile [72]. However, they are not widely used in mobile devices for two major reasons. Firstly, nowadays, most mobile applications are built with SQL statements. Lacking of the SQL interface causes semantic mismatch between the SQL-based mobile applications and key-value database engine. Thus, mobile applications need to be redesigned to be compatible with the key-value databases, which incurs too much overhead. Secondly, current key-value databases require large memory footprints to maintain in-memory metadata [26, 45, 75]. Such an in-memory metadata management approach introduces the overhead of notable memory occupation. In most of cloud computing environments, this is not a critical issue. However, for mobile devices with constrained memory space, this is nontrivial [68].

To make mobile applications benefit from the efficient key-value database engine, in this chapter, we propose a novel SQLite-like database engine, called SQLiteKV, which

adopts the LSM-tree-based key-value data structure but retains the SQLite interfaces. SQLiteKV consists of two layers: (1) A front-end layer that includes an SQLite-to-KV compiler and a novel coordination caching mechanism. (2) A back-end layer that includes an LSM-tree-based key-value database engine with an effective metadata management scheme.

In the front-end, the SQLite-to-KV compiler receives SQL statements and translates them into the corresponding key-value operations. A caching mechanism is designed to alleviate the discrepancy of data organization between SQLite and the key-value database. Considering the memory constraints issue in mobile devices, we manage the caching with a slab-based approach to eliminate memory fragmentation. Cache space is firstly segmented into slabs while each slab is further striped into an array of slots with equal size. One query result is buffered into the slab whose slot size is of the best fit with its own size [92,108,114].

For the back-end, we deploy an LSM-tree-based key-value database engine which transforms random writes to sequential writes by aggregating multiple updates in memory and dumping them to storage in a "batch" manner. To deal with the limited memory and energy resources for mobile devices [111], we propose to store exclusively the metadata for the top levels of the LSM tree in memory and leave others on storage to mitigate the memory requirement.

We have implemented and deployed the proposed SQLiteKV on a Google Nexus Android platform based on a key-value database-SnappyDB [116]. The experimental results with various workloads show that, our SQLiteKV presents up to 6 times performance improvement compared with SQLite. Our contributions are concluded as follows:

- We for the first time propose to improve the performance of SQLite by adopting the LSM-tree-based key-value database engine while remaining the SQLite interfaces for mobile devices.
- We design a slab-based coordination caching scheme to solve the semantic mismatch between the SQL interfaces and the key-value database engine, which also effectively improves the system performance.

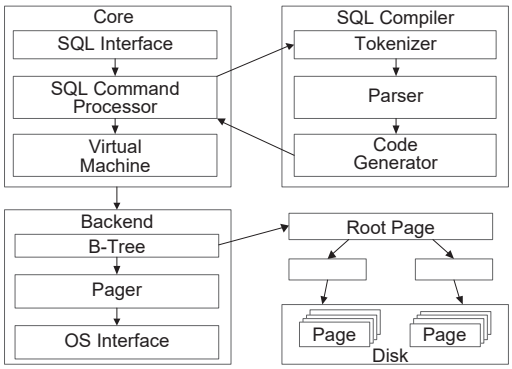


Figure 4.1: Architecture of SQLite.

- To mitigate the memory requirement for mobile devices, we have re-designed the index management policy for the LSM-tree-based key-value database engine.
- We have implemented and deployed a prototype of SQLiteKV with a real Google Android platform, and the evaluation results show the effective of our proposed design.

The rest of chapter is organized as follows. Section 4.2 presents some background information. Section 4.3 gives the motivation. Section 4.4 describes the design and implementation. Experimental results are presented in Section 4.5. Section 4.6 concludes this chapter.

4.2 Background

This section introduces some background information about SQLite, the LSM-tree-based key-value database, and other SQL-compatible key-value databases.

4.2.1 SQLite

SQLite is an in-process library, as well as an embedded SQL database widely used in mobile devices [52, 118]. Figure 4.1 gives the architecture of SQLite. SQLite exposes SQL interfaces to applications, and works by compiling SQL statements to bytecode, which then will be executed by a virtual machine. During the compiling of one SQL statement, the SQL

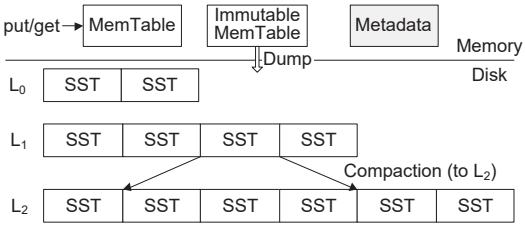


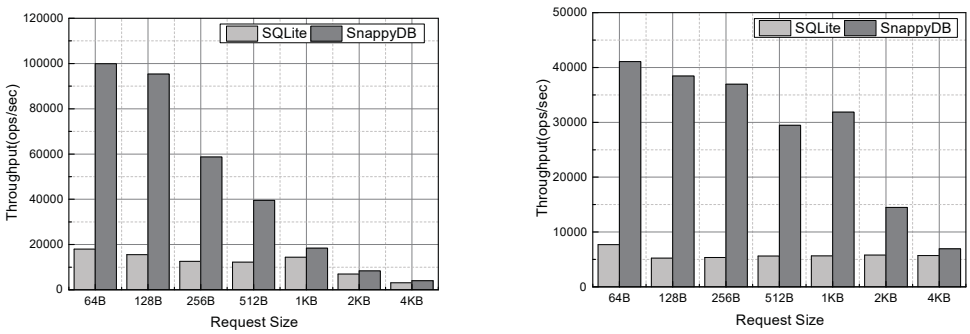
Figure 4.2: Architecture of the LSM-tree-based database.

command processor first sends it to the tokenizer. The tokenizer breaks the SQL statement into tokens and passes those tokens to the parser. The parser assigns meaning to each token based on its context, and assembles the tokens into a parse tree. Thereafter, the code generator analyzes the parser tree and generates virtual machine code that performs the work of the SQL statement. The virtual machine will run the generated virtual machine code with different operations files.

The data organization of SQLite is based on B-tree. One separate B-tree is used for each table in the database. The B-tree indexes data from the disk in fix-sized pages. The pages can be either data page, index page, free page or overflow page. All pages are of the same size and are comprised of multi-byte fields. The pager is responsible for reading, writing, and caching these pages. SQLite communicates with the underlying file system by system calls like `open`, `write` and `fsync`. Moreover, SQLite uses a journal mechanism for crush recovery, which makes the database file and journal file [12, 101] synchronized frequently with the disk and leads to a performance degradation consequently.

4.2.2 LSM-tree-based Key-Value Database

The LSM-tree-based data structure has been widely adopted by key-value databases which map a set of keys to associated values [15,36,95]. Applications access their data through simple `Put()` and `Get()` interfaces, that are the most generally used in NoSQL database [42, 91,100]. Figure 4.2 presents the architecture of an LSM-tree-based key-value storage engine, which consists of two MemTables in main memory and a set of sorted SSTables (shown as SST) in the disk. To assist database query operations, metadata, including indexes, bloom



(a) Throughput w. Insert operation.

(b) Throughput w. Query operation.

Figure 4.3: Performance comparison of SQLite vs SnappyDB

filters, key-value ranges and sizes of these on-disk SSTables, are maintained in memory [16, 108].

The LSM-tree-based key-value design is based on two optimizations: (1) New data must be quickly admitted into the store to support high-throughput writes. The database first uses an in-memory buffer, called MemTable, to receive incoming key-value items. Once a MemTable is full, it is first transferred into a sorted immutable MemTable, and dumped to disk as an SSTable. Key-value items in one SSTable are sorted according to their keys. Key ranges and a bloom filter of each SSTable are maintained as metadata cached in memory space to assist key-value query operations. (2) Key-value items in the store are sorted to support fast data localization. A multilevel tree-like structure is build to progressively sort key-value items in this architecture as shown in Figure 4.2.

The youngest level, $Level0$, is generated by writing the immutable MemTable from memory to disk. Each level has a limitation on the maximum number of SSTables. In order to keep the stored data in an optimized layout, a compaction process will be conducted to merge overlapping key-value items to the next level when the total size of $Levell$ exceeds its limitation.

4.2.3 Other SQL-Compatible Key-Value Databases

Apache Phoenix [99] is an open source relational database, in which a SQL statement is compiled into a series of key-value operations for HBase [44], a distributed, key-value database. Phoenix provides well-defined and industry standard APIs for OLTP and operational analytics for Hadoop [30, 132]. Nevertheless, without a deep integration with the Hadoop framework, it is difficult for mobile devices to adopt either HBase as its storage engine or Phoenix for SQL-to-KV transitions. Besides, Phoenix, along with other Hadoop related modules, is designed for scalable and distributed computing environments with large datasets [34], which means they can hardly fit in mobile environments with limited resources [115].

In this thesis, we propose an efficient LSM-tree-based lightweight database engine, SQLiteKV, which retains the SQLite interface for mobile devices, provides better performance compared with SQLite and adopts an efficient LSM-tree structure on its storage engine.

4.3 Motivation

To compare the performance of SQLite and the key-value based database engine, we choose one lightweight LSM-tree-based key-value database, called SnappyDB [116], and measure the throughput (operation per second, ops/sec) by running them with a Google Nexus smartphone. We use the Zipfian distribution [23] to generate the request popularity and request sizes are varied from 64 bytes to 4096 bytes. Figure 4.3 presents the throughput for both insert and query operations of these two databases, respectively.

Figure 4.3(a) shows the throughput for insert operations with SQLite and SnappyDB over vary-sized requests. It is obvious that SnappyDB outperforms SQLite significantly across the board. For instance, with the request size of 64 bytes, SnappyDB outperforms SQLite 7.3 times. The reason is mainly two folds. First, SQLite is a relational database which has strict data organization schema. All insert requests have to strictly follow the data organization schema and the slow transaction process of SQLite. The second reason is the

journaling of journal problem. In SQLite, an insert transaction firstly logs the insertion at an individual SQLite journal, and then inserts the record to the actual database table. At the end of each phase, SQLite calls fsync() to persist the results. Each fsync() call makes the underlying file system (e.g., ext4) update the database file and write the new metadata to the file system journal. Hence, a single insert operation may result in up to 9 I/Os to the storage device, and each I/O is done with the 4KB unit. This uncoordinated IO interaction brings much write amplification and sacrifices the performance of SQLite a lot. On the other hand, SnappyDB maintains a shared log, and an insert operation is firstly logged in the log file, and then served by its memory table as shown in Figure 4.2. This process is much simple and incurs less IOs to the storage device compared with SQLite.

It can also be observed from Figure 4.3(a) that with the request size increasing, the performance improvement of SnappyDB over SQLite decreases. This is because when the function fsync() is called, the underlying file system will do write operation with the 4KB unit. With the request size increasing close to 4KB, the write amplification overhead decreases. When the request size is of 4KB, the throughput of SQLite and SnappyDB are almost the same. Figure 4.3(b) gives the throughput for query operations of SQLite and SnappyDB, and it shows the same trend with insert operations.

We can conclude that the efficient LSM-tree-based key-value database engine outperforms the traditional relational database SQLite significantly. However, since the key-value database engine does not support SQL statements, most mobile applications cannot be directly moved to it and benefit from its high performance. To address this issue, in this thesis, we propose a new database engine, called SQLiteKV, which retains the SQLite interface for mobile devices and adopts an efficient LSM-tree-based key-value data structure on its storage engine.

4.4 SQLiteKV: An SQLite-like Key Value Database

To make mobile applications benefit from the efficient key-value database engine, we propose SQLiteKV, which not only inherits the high performance of key-value database en-

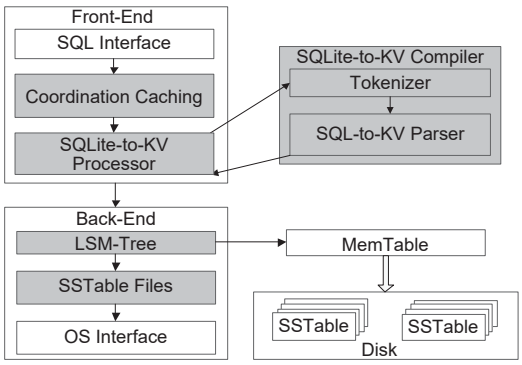


Figure 4.4: Architecture of SQLiteKV.

gines, but also provides the application compatible SQLite interfaces. In this section, we first present an overview of the SQLiteKV design, and then give the detailed descriptions for each of its modules.

4.4.1 Design Overview

Figure 4.4 presents the architecture of SQLiteKV. Similar to SQLite, SQLiteKV is composed of a front-end statement parser layer and a back-end data storage management layer. SQLiteKV's front-end layer mainly consists of two function modules: an SQLite-to-KV processor and a coordination read cache. Instead of translating SQL statements into virtual machine code in SQLite, the front-end of SQLiteKV parses the SQL statements into the corresponding key-value operations (e.g., `Put`, `Get`). The coordination read cache is used to buffer and quickly serve hot query requests. To reduce memory fragmentation, we adopt a slab-based way to manage the cache space. The SQLiteKV back-end layer is used to maintain the key-value pairs on disk with the LSM-tree-based data structure, and serve the parsed key-value requests. It also includes two function modules: a redesigned in-memory index management module which is used to save memory space for mobile devices, and an LSM-tree-based storage engine.

With SQLiteKV, when a SQL query statement comes, it will first search the coordination read cache, if the request data states in cache, the query will be directly returned. Otherwise, the SQL query statement will be translated into its corresponding key-value `Get`

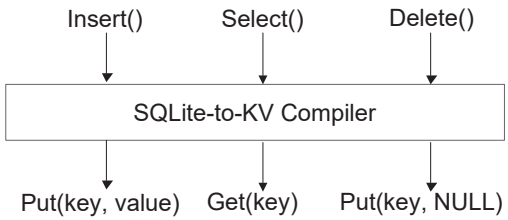


Figure 4.5: The SQLite to KV compiler.

operations through the SQLite-to-KV compiler. At last, the `Get` operation will be served by the back-end key-value database engine. In the following sections, we will introduce these four function modules in detail.

4.4.2 Front-End Layer

The SQLiteKV front-end layer includes two major components: an SQLite-to-KV compiler (Figure 4.5) which provides the compatible SQLite interface, and a coordination read cache (Figure 4.6) which accelerates the query process.

• SQLite-to-KV Compiler

As shown in Figure 4.5, the function of the SQLite-to-KV compiler is to translate a SQL statement to the corresponding key-value operations. It provides users SQLite-compatible interface while storing/retrieving key-value pairs with `Put` and `Set` operations in the back-end database. When a SQL statement comes, the SQLite-to-KV compiler firstly breaks down the statement into several tokens. Then it will assign each token a meaning based on the context and assemble it into a parser tree. The parsing process of the compiler is similar to SQLite. The noteworthy difference is that the SQLite-to-KV compiler generates key-value operations based on the result of the parse tree instead of SQL bytecode. The generated key-value operations do not depends on any virtual machine like SQLite, which make them more effective.

Basically, the SQLite-to-KV compiler translates the most commonly used SQLite interfaces `Insert()` and `Select()` to the corresponding key-value `Put()` and `Set()`

operations. Since we adopt an LSM-tree-based key-value database engine that does not support in-place updates, the SQLite interface `delete()` is transferred into one key-value `Put()` operations with an invalid value (e.g., "NULL" in Figure 4.5).

Algorithm 4.4.1 presents the working process for one insert operation in SQLiteKV. When SQLiteKV receives one insert SQL statement, it firstly begins a new transaction for this operation. Then SQLiteKV analyzes the tokens (Primary_key, column1, and column2) contained in this SQL statement and put them into the parse tree (container). SQLiteKV organizes the key and value pair for this SQL request based on this parse tree. In this example, SQLiteKV directly makes the primary_key as the key, and the column1 as the value. At last, the corresponding `Put()` operation with the newly identified key-value pair is issued to the back-end key-value database engine. As described above, the LSM-tree-based key-value engine does not support in-place updates, so the `delete()` statement is performed by the `put()` operation with an invalid value part (NULL) in SQLiteKV as shown in Algorithm 4.4.3.

Algorithm 4.4.2 describes the working process for a query operation in SQLiteKV. Similar to insert operation, SQLiteKV firstly parse the SQL statement into tokens (lines 3-4). Then it calculates a hash value based on these tokens, and search the cache with the hashed value. With cache hits, the data is directly returned from the cache. Otherwise, a corresponding key-value `Get()` operation is issued to the key-value database engine to retrieve the value to users and store it in the cache.

The SQLite-to-KV compiler makes it possible that existing applications run smoothly with original SQL statements and leverage the potentials of key value storage.

• Coordination Caching

Caching mechanism is of vital importance for improving the query efficiency of databases. Through buffering part of hot data in memory, query operations can be served fast without accessing the low-latency disk. In SQLiteKV, there are two cache configuration choices. As shown in Figure 4.6, the first choice is to maintain the cache in the back-end

Algorithm 4.4.1 Insert operation in SQLiteKV.

Input:

1: $insert\, values(Primary\_key, column1)$

Output: Perform key-value put operation

2: SQLiteKV.getWriteableDatabase();

3: //open the database for write

4: SQLiteKV.beginTransaction();

5: Container.put(Primary_key);

6: //construct the key for key-value pair

7: container.put(column1);

8: //construct the value for key-value pair

9: SQLiteKV.put(container.key, container.value);

10: //put the key-value item in the container to the key-value database

11: SQLiteKV.endTransaction();

12: return;

key-value database engine, and the second one is to put the cache in the front-end and before the SQLite-to-KV compiler module.

In SQLiteKV, we propose to adopt the second configuration as shown in Figure 4.6(b). The reason is that in Figure 4.6(a), the KV cache module stays in the back-end key-value database engine. Hot data buffered by this cache are maintained in the format of key-value pairs. When a SQL statement comes, if the data hit in cache, this configuration firstly needs to re-organize the data in key-value pairs to the SQL column format and then return to the users. Besides, in this configuration, whether cache hits or not, an incoming SQL statement always needs to go through the SQLite-to-KV compiler, which incurs reasonable overhead. In the second configuration, the cache stays in the front-end layer, and the hot data are main-

Algorithm 4.4.2 Query operation in SQLiteKV.

Input:

1: $select\ from\ test\ where\ column = values$

Output: Perform key-value get operation

```
2: SQLiteKV.getRriteableDatabase();
3: select_token = column+"=?";
4: arg_token = values;
5: hash_sql = hash(select_token, arg_token);
6: result = SQLiteKV.Cache.get(hash_sql)
7: // first access cache
8: if result == NULL then // not in cache
9:     result = SQLiteKV.get(arg_token);
10:    //get key-value pairs from KV database
11:    SQLiteKV.Cache.put(hash_sql, result);
12:    //buffer the result in cache
13: end if
14: return result;
```

tained in a SQL statement-oriented approach. When a SQL statement comes, if cache hits, the results can be directly returned without performing the SQLite-to-KV translation.

To further utilize the memory space efficiently, we adopt a slab-based memory allocation scheme as shown in Figure 4.7. We first segment the cache memory space into slabs. Each slab is further divided into an array of slots of equal size. Each slot stores one request data. Slabs are logically organized into different slab classes based on the slot sizes (e.g., 32B, 64B, 128B, ...). The data for one SQL query is stored into a class whose slot size is the best fit of its size. A hash mapping table is used to record the position of each SQL state-

Algorithm 4.4.3 Delete operation in SQLiteKV.

Input:

1: $delete\ from\ test\ where\ column = values$

Output: Perform key-value delete operation

```
2: SQLiteKV.getWriteableDatabase();
3: delete_token = column+"=?";
4: arg_token = values;
5: hash_sql = hash(delete_token, arg_token);
6: result = SQLiteKV.Cache.get(hash_sql)
7: // first access cache
8: if result != NULL then // clean cache
9:       SQLiteKV.Cache.delete(hash_sql);
10: end if
11: SQLiteKV.put(arg_token, NULL);
12: return;
```

ment. The key for the hash table is calculated by hashing tokens of each SQL query request as shown in Algorithm 4.4.2. When one query comes, SQLiteKV firstly analyzes its tokens, gets the hash value, and then read the data by combining the slab "sid" with offset "offset". Such a design can effectively address the issue of memory fragmentation, and utilizes the limited embedded memory resource more properly.

4.4.3 Back-End Layer

In SQLiteKV, we adopt an LSM-tree-based key-value database engine, like Google's LevelDB [36], and Facebook's Cassandra [15]. In this section, we will introduce our proposed new metadata management scheme, and the data storage in this database engine.

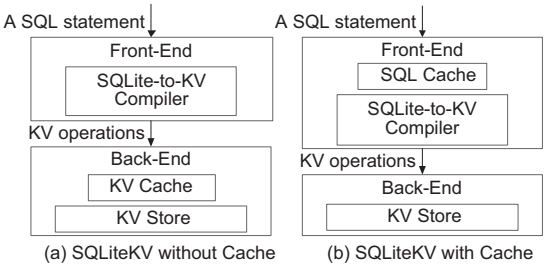


Figure 4.6: SQLiteKV Coordination Caching Mechanism.

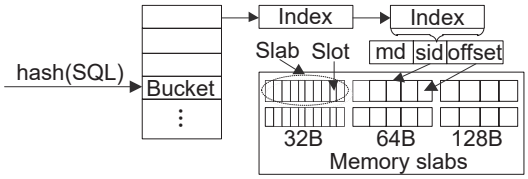


Figure 4.7: Slab-based cache management.

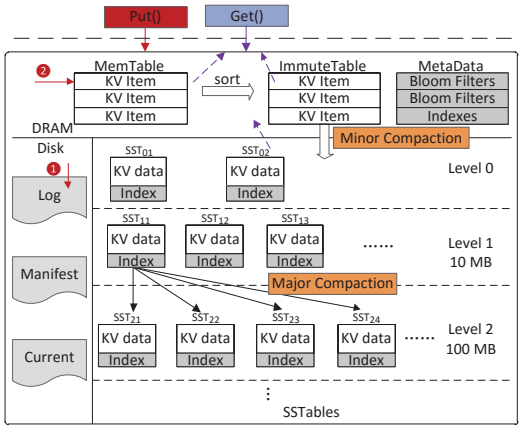


Figure 4.8: Back-End in-memory index management.

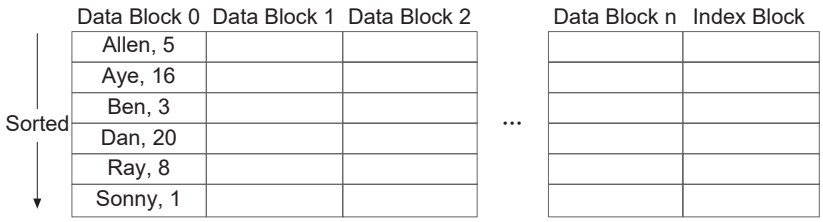


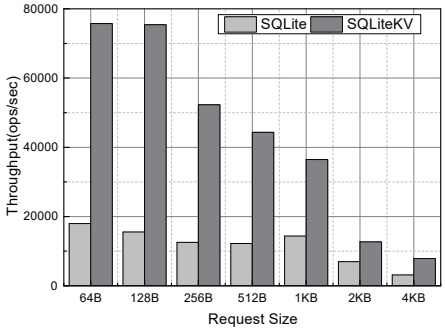
Figure 4.9: Data management in SSTable.

• Data Storage Management

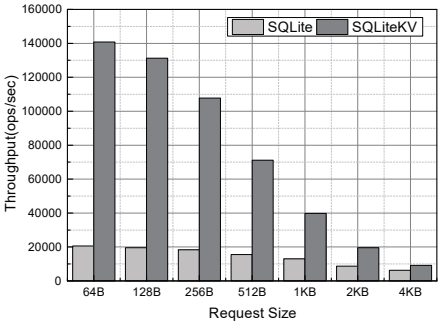
Figure 4.8 shows the data storage engine for the back-end LSM-tree-based key-value store. As described in Section 4.2, LSM-tree-based data store aggregates key-value items into equal-sized tables. There are three kinds of tables in the key-value database engine: memory table (`MemTable`), immutable table (`ImmuteTable`), and on-disk SSTable (`SST`). `Memtable` and `ImmuteTable` are maintained in memory, and SSTables are stored on disk. The SSTables are maintained in several levels (e.g., $Level0$, $Level1$, $Level2$). Each level contains different numbers of SSTables, and its capacity grows exponentially. `Log`, `Manifest`, and `Current`, are three configuration files used to assist the working process of the database engine.

When a key-value write request comes, it will firstly be written to the disk `Log` file in order to guarantee the overall consistency, and then buffered into the `MemTable` in memory. Once the `MemTable` becomes full, the key-value items in this table will be sorted and stored in the `ImmuteTable`. A minor compaction process will flush the key-value items in `ImmutaTable` to one disk SSTable in $Level0$. Key-value pairs stored in SSTables are sorted with their keys as shown in Figure 4.9. Each SSTable consists of several data blocks and index blocks. The index blocks maintain the mapping of the key range to the data blocks. With more SSTables flushed, if one level runs out of its space, a major compaction will be triggered to select one of its SSTable and do the merge sort operation with several SSTables in the next level (As shown in Figure 4.8, one SSTable in $Level1$ is compacted with 4 SSTables in $Level2$).

• Index Management

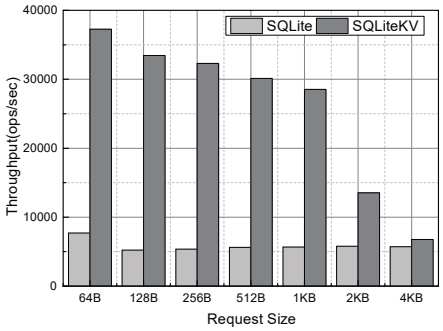


(a) Random insertions.

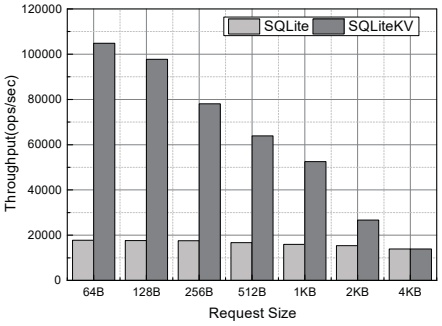


(b) Sequential insertions.

Figure 4.10: Insertion throughput vs. Request size



(a) Random queries.



(b) Sequential queries.

Figure 4.11: Basic performance of SQLiteKV and SQLite.

As described, each SSTable maintains some indexes to assist the quick search of key-value items. Usually, the indexes are stored at the end of each SSTable. To accelerate the query process, LSM-tree-based storage engines commonly scan over the entire disk and maintain a copy of all indexes in memory [126]. Hence when a query operation is to be executed, the in-memory meta data is accessed quickly with the target key to locate the data block on disk. Thus, the data block is visited to get the key-value item. Generally, one disk seek is required for a single query on LSM-tree-based key-value database engines.

However, this approach is not practical nor efficient for mobile devices. Since most mobile devices are memory constraint and cannot accommodate all the indexes in memory. Considering this limitation, we redesign the indexing management approach, which exclusively stores indexes of data blocks from higher levels, like $Level0$ and $Level1$, of the entire LSM tree. The reason is that as the level goes further down, data at lower levels are less likely to be visited. In other words, the data on top levels are fresher and more likely to be visited, since nearly 90 percent request are served by $Level0$ and $Level1$ [27]. This approach helps reduce the memory requirement in our key-value database with minimum overhead.

4.5 Evaluation

We have prototyped the proposed efficient LSM-tree-based SQLite-like database engine - SQLiteKV, on a Google mobile platform. Our implementation of the database engine is based on SnappyDB [116], which is a representative key-value database for mobile devices. Our SQLiteKV totally includes 2,506 lines in Java. In this section, we will first introduce the basic experimental setup, and then provide the experimental results with real-world benchmarks [23] and synthetic workloads.

4.5.1 Experiment Setup

The prototype of our proposed SQLiteKV is implemented on a Google mobile platform - Google Nexus 6p, which is equipped with a 2.0GHz oct-core 64 bit Qualcomm Snapdragon

810 processor, 3GB LPDDR4 RAM, and 32GB Samsung eMMC NAND Flash device. We use the Android 8.0 operating system with Linux Kernel 3.10. In the evaluation, SQLite 3.18 is utilized in the experiments as it is the current version in Android 8.0 Oreo. The page size of SQLite is set as 1024 bytes, which is the default value. SnappyDB 0.5.2, which is the latest version of a Java implementation of Google's LevelDB, is adopted.

Since in most real-world SQLite workloads, one SQLite query always carry more than one records. So, in our experiments, we make each SQL statement in SQLite contains up to 999 records, which is the maximum value allowed. With this performance tuning method, we can make a fair comparison between SQLiteKV and SQLite. Moreover, trivial calls, like moving cursors after queries in SQLite, are omitted for the sake of efficiency.

4.5.2 Basic Performance

We first evaluate the basic performance of SQLiteKV and SQLite, mainly in terms of throughput (operations per second, ops/sec). In this experiment, we set the data size vary from 64 bytes to 4096 bytes, and investigate the throughputs of both random and sequential accesses. We test and compare the throughput of SQLiteKV and SQLite with the commonly used insert, query, and delete operations.

• Insertion Performance

Figure 4.10a and Figure 4.10b show the performances of SQLiteKV and SQLite with random and sequential insertion operations, respectively. The request sizes vary from 64 bytes to 4096 bytes. For the sequential access workload, the requests are in ascending order, while the requests are randomly traversed for the random case.

It can be observed that the performances of SQLiteKV outperform SQLite significantly for request of all sizes. For sequential operations, with the request size of 64 bytes, the throughput of SQLiteKV is 1.41×10^5, which is 6.1 times higher than that of SQLite. For random operations, the SQLiteKV outperforms SQLite 3.8 times in maximum with request size of 64 bytes. As the request size increases, the throughputs decrease across the board

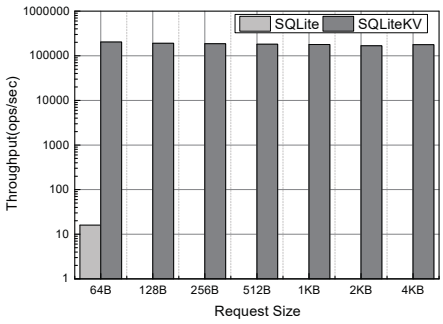


Figure 4.12: Delete throughput vs. Delete operations.

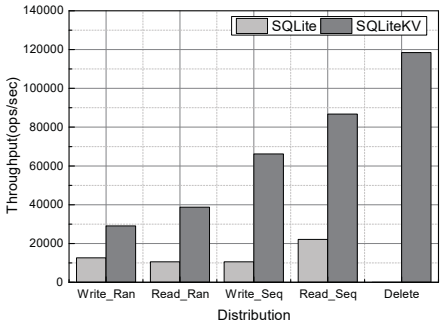


Figure 4.13: Throughput vs. Request size with Zipfan model.

for both SQLite and SQLiteKV. The performance improvement of SQLiteKV over SQLite also decreases with the request size increasing. This is because the write amplification effect caused by SQLite's journal of journal problem declines as the request size increases. Besides, it can be observed that for SQLitKV, the insertion performance with sequential access workloads are better than that with random access ones (the average improvement is 40%). On the contrary, for SQLite, there are basically no differences between the random and sequential cases.

• Query Performance

Figure 4.11b shows the performance of SQLiteKV and SQLite with random and sequential query operations, respectively. Basically, query operations show the same trend with insert operations. For sequential queries, with the request size of 64 bytes, the throughput of SQLiteKV is 1.01×10^5, which is 4.91 times higher than that of SQLite. For random queries, the performance improvement of SQLiteKV over SQLite is up to 5.4 times with the request size of 128 bytes. Furthermore, the sequential query throughput of SQLiteKV is much higher than the random query one, which is about 1.3 times.

• Delete Performance

Figure 4.12 presents the throughput of delete operations for SQLiteKV and SQLite, respectively. It is obvious that the throughput for delete operations of SQLiteKV is much higher than that of SQLite. For instance, with the request size of 4KB, it even takes more than

several minutes to delete a record for SQLite. However, in SQLiteKV, the delete operation is implemented by the key-value `Put` operation with invalid data area. Thus, the delete operations of SQLiteKV perform the same as insert operations.

We further test the random insert, random query, sequential insert, and sequential query performance of SQLiteKV and SQLite with request sizes that follow the Zipfian [23, 114] distribution. The results are given in Figure 4.13. We can conclude that SQLiteKV outperforms SQLite for all cases. Especially, for sequential write operations, the improvement is up to 5.3 times.

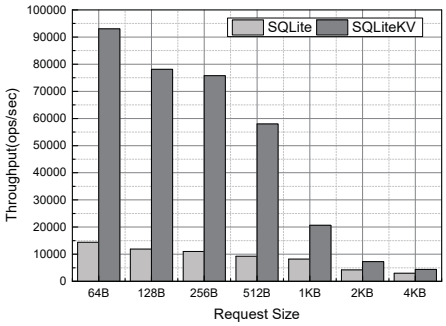
4.5.3 Overall Performance

Table 4.1: Workload characteristics.

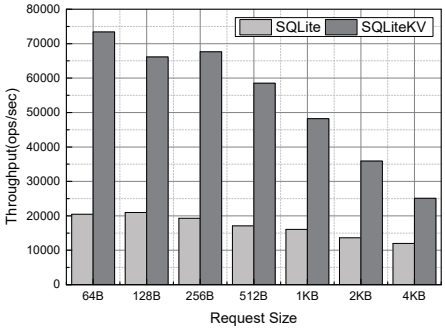
Workload(s)	Query	Insert
Update Heavy	0.5	0.5
Read Most	0.95	0.05
Read Heavy	1	0
Read Latest	0.95	0.05

To evaluate the overall performance, we further test the proposed SQLiteKV with a set of YCSB [23] core workloads that define a basic benchmark as shown in Table 4.1. Here, the update heavy workload has a mix of 50/50 reads and writes, the read most workload has a 95/5 reads/write mix, the read heavy workload is 100% read, and in the read latest workload, new records are inserted, and the most recently inserted records are the most popular.

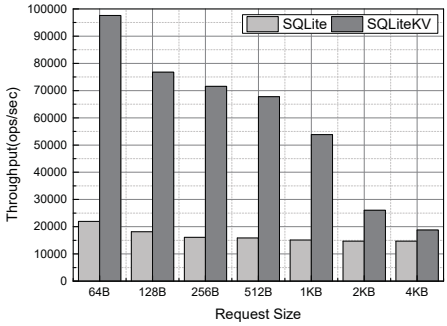
To conduct the experiments, we first generate 100 thousand key value pairs to populate SQLiteKV and SQLite, and then use the object popularity model to generate 100 thousand requests sequence. The object popularity, which determines the requests sequence, follows the Zipfian distribution, by which records in the head will be extremely popular while



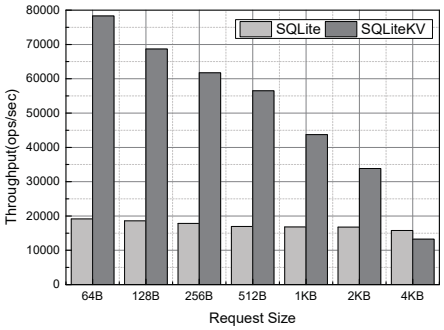
(a) Throughput w. Update heavy.



(b) Throughput w. Read most.



(c) Throughput w. Read heavy.



(d) Throughput w. Read latest.

Figure 4.14: Overall Performance

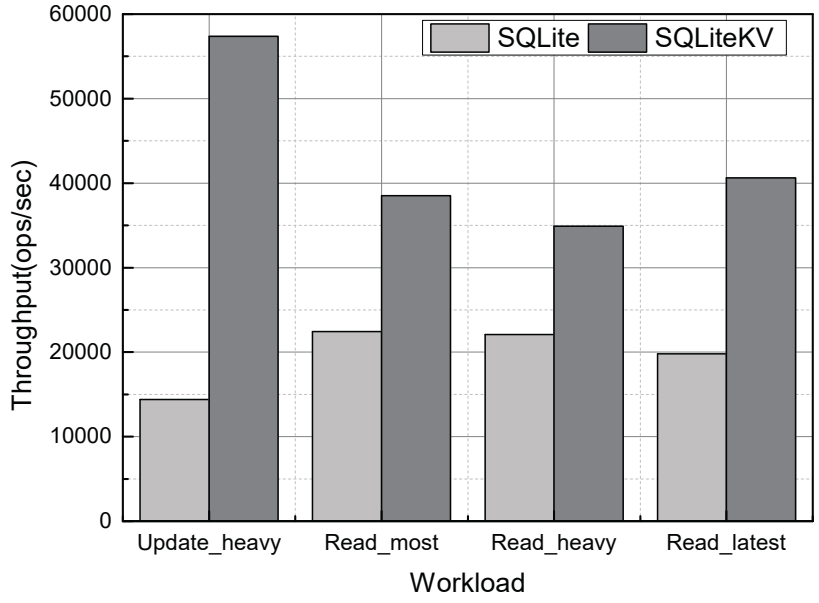


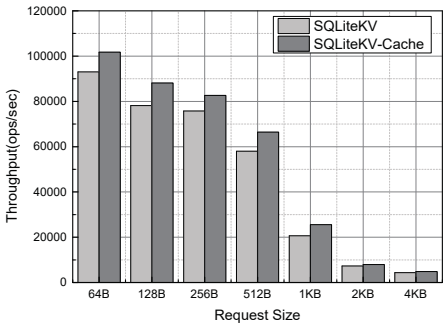
Figure 4.15: Performance evaluation.

those in the tail are not. For the read latest workload, we make most recently inserted records in the head that will be accessed more frequently. For the request size, similarly, we use both the fixed request sizes from 64 bytes to 4KB and the request sizes which follows the Zipfian distribution.

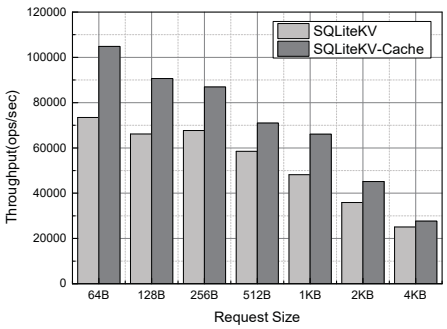
Figure 4.14 shows the experimental results by running SQLiteKV and SQLite with the four workloads in Table 4.1. For each workload, the request sizes vary from 64 bytes to 4096 bytes. It can be observed that, compared with SQLite, SQLiteKV significantly increases the throughput across the board with varied request sizes. For the update heavy workload, the throughput of SQLiteKV is 3.9 times higher than that of SQLite on average. With the request size of 256 bytes, the performance improvement achieves the highest point, which is about 5.9 times. On average, the performance improvement of SQLiteKV over SQLite for the other three workloads: read most, read heavy and read latest, are 2 times, 2.4 times and 1.9 times, respectively.

We also notice that when the key-value sizes are over 2048 bytes, SQLiteKV only outperform SQLite slightly. The reason is that bigger request size can reduce the write amplification effect in SQLite. Besides, for LSM-tree-based databases, keys and values are written at least twice: the first time for the transactional log and the second time for storing data to storage devices. Thus, the per-operation performance of SQLiteKV is degraded by extra write operations. Regardless of this degradation, as most data sets in mobile applications only contain very few large requests, SQLiteKV can still significantly outperform SQLite in practice.

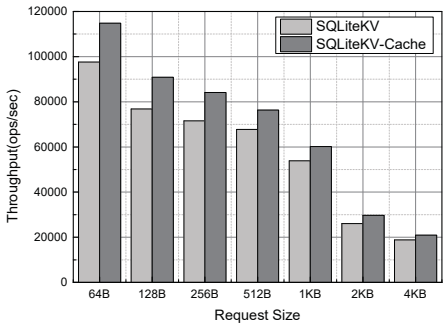
We also test the database performance with request sizes following the Zipfian distribution. Figure 4.15 presents the results with the four workloads. It can be observed that with the update heavy workload, SQLiteKV achieves the highest performance improvement over SQLite, which is 2.7 times. On average, the SQLiteKV outperforms SQLite 1.7 times for all these four workloads.



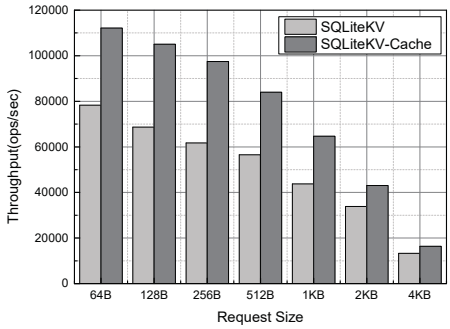
(a) Throughput w. Update heavy.



(b) Throughput w. read most.



(c) Throughput w. Read heavy.



(d) Throughput w. Read latest.

Figure 4.16: Performance of SQLiteKV with and without cache.

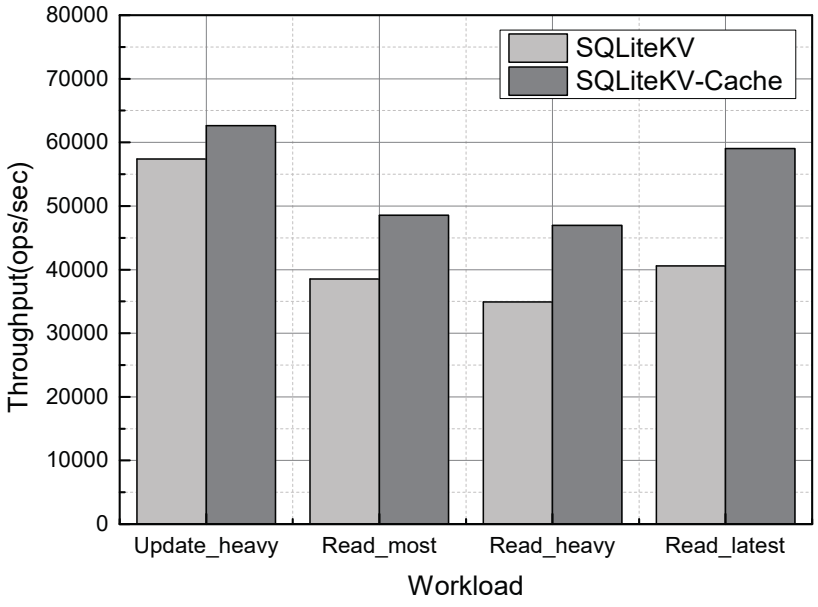


Figure 4.17: Cache effect with Zipfian distributed request sizes.

4.5.4 Coordination Cache Effect

We further evaluate the efficiency of our proposed coordination read cache with different workloads . Figure 4.16 presents the performance comparisons of SQLiteKV with and without cache. Similarly, the request sizes vary from 64 bytes to 4KB, and for all these experiments, we configured the cache size fixed with 1MB. Through the figures, we can clearly see that the coordination cache can effectively improve the database performance. The average performance improvement are 12.7% for the update heavy workload, 28.9% for the read most workload, 14.7% for the read heavy workload, and 43% for the read latest workload. The highest performance improvement with the coordination cache achieves 57.9% for the read latest workload with the request size of 256 bytes. We also test the cache effect with the request sizes that follow the Zipfian distribution. As shown in Figure 4.17 , similarly the coordination cache effectively improves the throughput across all the workloads.

4.5.5 CPU and Memory Consumption

In this section, we will investigate the efficiency of our re-designed index management policy, and then compare the memory and CPU consumption of SQLite, SnappyDB, and SQLiteKV. For SQLiteKV, we have enabled the coordination cache. In this experiment, we also generated 100 thousand request data to pollute the database, and then issue insert and query requests to investigate their effects on the CPU and memory.

Table 4.2: CPU and memory consumption.

Databases	CPU/(%)		Memory/(MB)	
	Insert	Query	Insert	Query
SQLite	41	38	165.06	148.89
SnappyDB	26	50	188.28	192.29
SQLiteKV	32	61	82.47	81.9

Memory Footprint. Table 4.2 presents the memory utilization status for SQLite, SnappyDB, and SQLiteKV. SnappyDB consumes the most memory space for both insert and query operations. The reason is that in SnappyDB, the LSM-tree-based data structure needs to maintain the indexes of all the SSTables from all the levels in memory, which is non-trivial. As we have described in Section 4.4.3, nearly 90 percent query requests goes to $Level0$ and $Level1$ in real key-value store applications [27]. Thus, in SQLiteKV, we only maintain the indexes of $Level0$ and $Level1$ in memory, and leaves the others on disk. The experimental results in Table 4.2 show that our index management policy significantly reduces the memory requirement. Comparing with SnappyDB, SQLiteKV saves 56.2% and 57.4% memory space for insert and query operations, respectively.

CPU Utilization. We can observe that for insert operations, SQLite requires the most CPU resource. This is because SQLite needs to maintain its in-memory B-tree index structure, which may include many split and compaction processes. On the contrary, the indexes management for SnappyDB and SQLiteKV maintain the bloom filters and key ranges information, whose operation are relatively simple. However, we also notice that our SQLiteKV consumes nearly 20% more CPU resource compared with SnappyDB. The reason is that in SQLiteKV, the SQLite-to-KV compiler requires the participation of CPU to keep translating the incoming SQL statements into the corresponding key-value operations. For query operations, SnappyDB and SQLiteKV require more CPU resources compared with SQLite, and SQLiteKV consumes the most. Since in SnappyDB and SQLiteKV, to locate a request, they may need to check several bloom filters and the indexes of SSTables in more than one levels, which requires extra CPU resource. For SQLiteKV, except the SQLite-to-KV compiler consumption, it may need to do more search to located one key-value item compared with SnappyDB, since it only stores the metadata of $Level0$ and $Level1$ in memory. Thus, SQLiteKV requires the most CPU resource.

4.6 Summary

In this chapter, we propose a new database engine for mobile devices, called SQLiteKV, which is an SQLite-like key-value database engine. SQLiteKV adopts the LSM-tree-based data structure but retains the SQLite operation interfaces. SQLiteKV consists of two parts: a front end that contains a light-weight SQLite-to-key-value compiler and a coordination caching mechanism; a back end that adopts a LSM-tree-based key-value database engine. We have implemented and deployed our SQLiteKV on a Google Nexus 6P Android platform based on a key-value database SnappyDB. Experimental results with various workloads show that the proposed SQLiteKV outperforms SQLite significantly.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

The key-value database engine, which provides higher efficiency, scalability, availability, and works with a simple NoSQL interface, has been widely adopted as the caching system in todays low-latency Internet services. However, when using SSDs as the storage devices, several redundant functions exist between the key-value caching systems and the underlying hardware, which hurt the system performance. In addition, although the key-value database engine has been proven to be more profitable than traditional relational SQL databases in cloud environments, it has seldom been adopted by mobile applications, in this thesis, we optimize the key-value data stores with three schemes, namely the integration of the emerging hardware open-channel SSD, the cross-layer hardware/software management, and the design of an SQLite-to-KV compiler for mobile applications.

For the first scheme, we present a codesign approach to deeply integrate the key-value cache system design with the flash hardware. Our solution enables three key benefits, namely a single-level direct mapping from keys to physical flash memory locations, a cache-driven fine-grained garbage collection, and an adaptive over-provisioning scheme. We implemented a prototype on a real open-channel SSD hardware platform. Our experimental results show that we can significantly increase the throughput by 35.5%, reduce the latency by 23.6%, and remove unnecessary erase operations by 28%. Although this thesis focuses on key-value caching, such an integrated approach can be generalized and applied to other semantically rich applications, such as file systems, databases, and virtualization, which will be the focus of our work in the future.

For the second scheme, current SSDs either adopt the traditional block I/O interface or directly expose its low-level details to applications with open-channel SSDs. Both have their advantages and disadvantages, but neither is the optimal approach. We propose a new programming model that includes three layers of abstraction for open-channel SSDs: a raw flash abstraction, a flash function abstraction, and a user-FTL abstraction. The programming model provides applications with the flexibility to integrate their software semantics with the flash management with different granularities. We have implemented applications based on the programming model, and the results of the evaluation show the effectiveness of our proposed programming model.

For the third scheme, we propose a new database engine for mobile devices, called SQLiteKV, which is an SQLite-like key-value database engine. SQLiteKV adopts the LSM-tree-based data structure but retains the SQLite operation interfaces. SQLiteKV consists of two parts: a front end that contains a light-weight SQLite-to-key-value compiler and a coordination caching mechanism; and a backend that adopts an LSM-tree-based key-value database engine. We have implemented and deployed our SQLiteKV on a Google Nexus 6P Android platform based on a key-value database SnappyDB. The results of experiments with various workloads show that the proposed SQLiteKV significantly outperforms SQLite.

5.2 Future Work

The work presented in this thesis can be extended to different directions in the future.

First, crash recovery is an important character of key-value caching system. How to combine our approach to effectively do crash recovery can be a future direction for us to explore. Second, when deploying DIDACache in a distributed environment, we will encounter new challenges and problems. We will continue to work on DIDACache to make it run in a distributed environment and try to solve the new challenges under a distributed storage cluster. Third, our prototype Prism-SSD is implemented with three user level abstractions. However, the implementation is not strictly constrained to these three levels. We will further extend the library with some application-specific interfaces, such as key-value

set/get interfaces for key-value stores. Forth, our approach is based on flash-based hardware. We will extend our approach to other emerging non-volatile-memories (NVMs). We will explore how to integrate NVMs into the key-value stores to further improve their performances. Fifth, wear-leveling is critical to flash lifetime. We will further explore how to integrate the key-value store semantics with flash management to customize new effective wear-leveling policies. Finally, as the first step in the exploration of SQLite-like key-value database engines, the translation of SQL operations to key-value operations in SQLiteKV is a straightforward process. In the future, we will extend the compiler design to make this process of translation to support more complicated SQL interfaces.

REFERENCES

[1] Fatcache-Async. https://github.com/polyu-szy/Fatcache-Async-2017.

[2] Filebench benchmark. http://sourceforge.net/apps/mediawiki/filebench.

[3] Riak. http://basho.com/leveldb-in-riak-1-2/.

[4] Rocksdb, a persistent key-value store for fast storage environments. https://rocksdb.org/.

[5] Tair. http://code.taobao.org/p/tair/src/.

[6] Whitepaper: memcached total cost of ownership (TCO). https://goo.gl/SD2rZe.

[7] Xmp. https://github.com/libfuse/libfuse/releases.

[8] yahoo-web. http://webscope.sandbox.yahoo.com/catalog.php?datatype=g.

[9] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conference (ATC 08)*, 2008.

[10] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. Cheap and large CAMs for high performance data-intensive networked systems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*, 2010.

[11] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review (SIGMETRICS 12)*, 2012.

[12] Steve Best. Jfs log: how the journaled file system performs logging. In *Proceedings of the 4th annual Linux Showcase*, page 9, 2000.

[13] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. Lightnvm: The linux open-channel ssd subsystem. In *USENIX Conference on File and Storage Technologies (FAST 17)*, 2017.

[14] Damiano Carra and Pietro Michiardi. Memory partitioning in Memcached: an experimental performance analysis. In *International Conference on Communications (ICC 14)*, 2014.

[15] Apache Cassandra. http://cassandra.apache.org/.

[16] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: a distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26:4, 2008.

[17] 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 *USENIX Conference on File and Storage Technologies (FAST'11)*, 2011.

[18] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 09)*, 2009.

[19] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA 11)*, 2011.

[20] Renhai Chen, Yi Wang, Jingtong Hu, Duo Liu, Zili Shao, and Yong Guan. vflash: virtualized flash for optimizing the i/o performance in mobile devices. *IEEE Transac-*

tions on Computer-Aided Design of Integrated Circuits and Systems, 36:1203–1214, 2017.

[21] Yen-Ting Chen, Ming-Chang Yang, Yuan-Hao Chang, Tseng-Yi Chen, Hsin-Wen Wei, and Wei-Kuan Shih. Kvftl: optimization of storage space utilization for key-value-specific flash storage devices. In *IEEE 22nd Asia and South Pacific Design Automation Conference (ASP-DAC 2017)*, pages 584–590, 2017.

[22] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. A survey of flash translation layer. *Journal of Systems Architecture*, 55(5):332–343, 2009.

[23] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *ACM Symposium on Cloud Computing (SOCC 2010)*, pages 143–154, 2010.

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

[25] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: high throughput persistent key-value store. *Proceedings of the VLDB Endowment (VLDB 10)*, 2010.

[26] Biplob Debnath, Sudipta Sengupta, and Jin Li. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 11)*, 2011.

[27] Wei Deng, Ryan Svihla, and DataStax. The missing manual for leveled compaction strategy. goo.gl/En73gW, 2016.

[28] C. Dirik and B. Jacob. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device, architecture, and system organization. In *International Symposium on Computer Architecture (ISCA 09)*, 2009.

[29] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a key-value cache that minimizes writes to flash. *arXiv preprint arXiv:1702.02588*, 2017.

[30] Sara B Elagib, and Aisha H Hashim Atahur Rahman Najeeb, and Rashidah F Olanrewaju. Big data analysis solutions using mapreduce framework. In *IEEE International Conference on Computer and Communication Engineering (ICCCE 2014)*, pages 127–130, 2014.

[31] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: a distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM 12)*, 2012.

[32] Facebook. McDipper: a key-value cache for flash storage. https://goo.gl/ZaavWa.

[33] Hossein Falaki, Ratul Mahajan, Srikanth Kandula, Dimitrios Lymberopoulos, Ramesh Govindan, and Deborah Estrin. Diversity in smartphone usage. In *The 8th International Conference on Mobile Systems, Applications, and Services (MobiSys'10)*, pages 179–194, 2010.

[34] George H Forman and John Zahorjan. The challenges of mobile computing. *Computer*, pages 38–47, 1994.

[35] E. Gal and S. Toledo. Algorithms and data structures for flash memories. In *ACM Computing Survey (CSUR)*, volume 37:2, 2005.

[36] Sanjay Ghemawat and Jeffery Dean. Leveldb, a fast and lightweight key/value database library by google., 2014.

[37] Salil Gokhale, Nitin Agrawal, Sean Noonan, and Cristian Ungureanu. KVZone and the search for a write-optimized key-value store. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 10)*, 2010.

[38] Javier González, Matias Bjørling, Seongno Lee, Charlie Dong, and Yiren Ronnie Huang. Application-driven flash translation layers on Open-Channel SSDs. 2016.

[39] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. Characterizing flash memory: anomalies, observations, and applications. In *International Symposium on Microarchitecture (Micro 09)*, 2009.

[40] Yong Guan, Guohui Wang, Yi Wang, Renhai Chen, and Zili Shao. Blog: block-level log-block management for nand flash memorystorage systems. In *Annual ACM SIGPLAN / SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 17)*, 2013.

[41] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 09)*, 2009.

[42] Jing Han, Haihong E, Guan Le, and Jian Du. Survey on nosql database. In *IEEE International Conference on Pervasive computing and applications (ICPCA 2011)*, pages 363–366, 2011.

[43] Mingzhe Hao, Gokul Soundararajan, Deepak R Kenchammana-Hosekote, Andrew A Chien, and Haryadi S Gunawi. The tail at store: a revelation from millions of hours of disk and SSD deployments. In *USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.

[44] Apache Hbase. https://hbase.apache.org/.

[45] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. Lama: optimized locality-aware memory allocation for key-value cache. In *USENIX Annual Technical Conference (ATC 2015)*, pages 57–69, 2015.

[46] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K Qureshi. Flashblox: achieving both performance isolation and uniform lifetime for virtualized ssds. In *USENIX Conference on File and Storage Technologies (FAST 17)*, 2017.

[47] Daeho Jeong, Youngjae Lee, and Jin-Soo Kim. Boosting quasi-asynchronous IO for better responsiveness in mobile devices. In *USENIX Conference on File and Storage Technologies (FAST 2015)*, pages 191–202, 2015.

[48] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O stack optimization for smartphones. In *USENIX Conference on Usenix Annual Technical Conference (ATC 2013)*, pages 309–320, 2013.

[49] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. Kaml: a flexible, high-performance key-value ssd. In *IEEE International Symposium on High Performance Computer Architecture (HPCA 17)*, 2017.

[50] Steve Schlosser Greg Ganger John Bucy, Jiri Schindler. DiskSim 4.0. http://www.pdl.cmu.edu/DiskSim/.

[51] William K Josephson, Lars A Bongo, Kai Li, and David Flynn. DFS: a file system for virtualized flash storage. *ACM Transactions on Storage (TOS)*, 6(3):14, 2010.

[52] Lv Junyan, Xu Shiguo, and Li Yijie. Application research of embedded database sqlite. In *International Forum on Information Technology and Applications (IFITA 2009)*, volume 2, pages 539–543, 2009.

[53] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed solid-state drive. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, 2014.

[54] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-ftl: transactional FTL for SQLite databases. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 2013)*, pages 97–108, 2013.

[55] Yangwook Kang, Yang-suk Kee, Ethan L Miller, and Chanik Park. Enabling cost-effective data processing with smart SSD. In *IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST 13)*, 2013.

[56] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *USENIX Technical Conference*, 1995.

[57] Dohee Kim, Eunji Lee, Sungyong Ahn, and Hyokyung Bahn. Improving the storage performance of smartphones through journaling in non-volatile memory. *IEEE Transactions on Consumer Electronics*, 59:556–561, 2013.

[58] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting storage for smartphones. *ACM Transactions on Storage (TOS)*, 8:14, 2012.

[59] Je-Min Kim and Jin-Soo Kim. Androbench: benchmarking the storage performance of android-based mobile devices. *Springer Frontiers in Computer Education*, pages 667–674, 2012.

[60] Jesung Kim, Jong Min Kim, Sam H Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.

[61] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: exploiting NVRAM in write-ahead logging. pages 385–398, 2016.

[62] Wook-Hee Kim, Beomseok Nam, Dongil Park, and Youjip Won. Resolving journaling of journal anomaly in android i/o: multi-version b-tree with lazy split. In *USENIX Conference on File and Storage Technologies (FAST 2014)*, pages 273–285, 2014.

[63] Ana Klimovic, Christos Kozyrakis, Eno Thereksa, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *The Eleventh European Conference on Computer Systems (EuroSys 16)*, 2016.

[64] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *19th international conference on World Wide Web (WWW)*, 2010.

[65] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. GraphChi: large-scale graph computation on just a PC. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[66] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *ACM Association for Computing Machinery Special Interest Group on Computer Architecture (SIGARCH 09)*, 2009.

[67] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. F2FS: a new file system for flash storage. In *USENIX Conference on File and Storage Technologies (FAST 15)*, 2015.

[68] Hyung Gyu Lee and Naehyuck Chang. Energy-aware memory allocation in heterogeneous non-volatile memory systems. In *International Symposium on Low Power Electronics and Design (ISLPED 2003)*, pages 420–423, 2003.

[69] Kisung Lee and Youjip Won. Smart layers and dumb result: IO characterization of an android-based smartphone. In *Proceedings of the tenth ACM international conference on Embedded software (EMSOFT 2012)*, pages 23–32, 2012.

[70] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3):18, 2007.

[71] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, et al. Application-managed flash. In *USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.

[72] Leveldown-Mobile. https://github.com/No9/leveldown-mobile.

[73] Adam Leventhal. Flash storage memory. In *Communications of the ACM*, volume 51(7), pages 47–51, 2008.

[74] Paul Lilly. Facebook ditches DRAM, flaunts flash-based McDipper. http://www.maximumpc.com/facebook-ditches-dram-flaunts-flash-based-mcdipper., 2013.

[75] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. SILT: a memory-efficient, high-performance key-value store. In *ACM Symposium on Operating Systems Principles (SOSP 11)*, 2011.

[76] Seung-Ho Lim and Kyu-Ho Park. An efficient NAND flash file system for flash memory storage. *IEEE Transactions on Computers*, 55(7):906–912, 2006.

[77] Duo Liu, Tianzheng Wang, Yi Wang, Zhiwei Qin, and Zili Shao. Pcm-ftl: A write-activity-aware nand flash memory management scheme for pcm-based embedded systems. In *IEEE Real-Time Systems Symposium (RTSS 11)*, 2011.

[78] Linbo Long, Liu Duo, Liang Liang, Xiao Zhu, Kan Zhong, Zili Shao, and Edwin Hsing-Mean Sha. Morphable resistive memory optimization for mobile virtualization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35.

[79] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. WiscKey: separating keys from values in SSD-conscious storage. In *USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.

[80] Youyou Lu, Jiwu Shu, Weimin Zheng, et al. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *USENIX Conference on File and Storage Technologies (FAST 13)*, 2013.

[81] Fabio Margaglia, Gala Yadgar, Eitan Yaakobi, Yue Li, Assaf Schuster, and André Brinkmann. The devil is in the details: implementing flash page reuse with WOM codes. In *USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.

[82] Leonardo Mármol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. NVMKV: a scalable and lightweight, FTL-aware key-value store. In *USENIX Annual Technical Conference (ATC 15)*, 2015.

[83] Brian Marsh, Fred Douglis, and P Krishnan. Flash memory file caching for mobile computers. In *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, 1994.

[84] Julian McAuley and Jure Leskovec. Learning to discover social circles in ego networks. In *25th International Conference on Neural Information Processing Systems (NIPS)*, 2012.

[85] Memblaze. Memblaze. http://www.memblaze.com/en/.

[86] Memcached. Memcached: a distributed memory object caching system. http://www.memcached.org.

[87] Violin Memory. All flash array architecture. 2012.

[88] Michael P. Mesnier, Jason Akers, Feng Chen, and Tian Luo. Differentiated storage services. In *ACM Symposium on Operating System Principles (SOSP 11)*, 2011.

[89] Nikolaus Rath Miklos Szeredi. Filesystem in userspace. http://fuse.sourceforge.net.

[90] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: random write considered harmful in solid state drives. In *USENIX Conference on File and Storage Technologies (FAST 12)*, 2012.

[91] ABM Moniruzzaman and Syed Akhter Hossain. Nosql database: new era of databases for big data analytics-classification, characteristics and comparison. *arXiv preprint arXiv:1307.0191*, 2013.

[92] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In

USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), 2013.

[93] Gihwan Oh, Sangchul Kim, Sang-Won Lee, and Bongki Moon. SQLite optimization with phase change memory for mobile applications. *International Conference on Very Large Databases (VLDB 2015)*, pages 1454–1465, 2015.

[94] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H Noh. Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems. In *USENIX Conference on File and Storage Technologies (FAST 12)*, 2012.

[95] Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. The log-structured merge-tree (lsm-tree). *Springer Acta Informatica*, 33:351–385, 1996.

[96] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: software-defined flash for web-scale internet storage systems. In *ACM International Conference Architecture Support for Programming Languages and Operating Systems (ASPLOS 14)*, 2014.

[97] Xiangyong Ouyang, Nusrat S Islam, Raghunath Rajachandrasekar, Jithin Jose, Miao Luo, Hao Wang, and Dhabaleswar K Panda. SSD-assisted hybrid memory to accelerate memcached over high performance networks. In *International Conference on Parallel Processing (ICPP 12)*, 2012.

[98] Chen Pan, Mimi Xie, Chengmo Yang, Yiran Chen, and Jingtong Hu. Exploiting multiple write modes of nonvolatile main memory in embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(4), 2017.

[99] Apache Phoenix. https://phoenix.apache.org/.

[100] Jaroslav Pokorny. Nosql databases: a step to database scalability in web environment. *Emerald Group Publishing Limited International Journal of Web Information Systems*, 9.

[101] Vijayan Prabhakaran, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *USENIX Annual Technical Conference (ATC 2005)*, pages 196–215.

[102] Zhiwei Qin, Yi Wang, Duo Liu, Zili Shao, and Yong Guan. Mnftl: An efficient flash translation layer for mlc nand flash memory storage systems. In *Proceedings of the 48th Design Automation Conference (DAC 11)*, 2011.

[103] Redis. http://redis.io/.

[104] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. volume 10, pages 26–52, 1992.

[105] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: edge-centric graph processing using streaming partitions. In *ACM Symposium on Operating Systems Principles (SOSP 13)*, 2013.

[106] SamSung. *Samsung 840 Pro.*

[107] Mohit Saxena, Michael M Swift, and Yiying Zhang. Flashtier: a lightweight, consistent and durable storage cache. In *The European Conference on Computer Systems (EuroSys 12)*, 2012.

[108] Russell Sears and Raghu Ramakrishnan. bLSM: a general purpose log structured merge tree. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 2012)*, pages 217–228, 2012.

[109] Sudharsan Seshadri, Mark Gahagan, Meenakshi Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: a user-programmable SSD. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[110] Mansour Shafaei, Peter Desnoyers, and Jim Fitzpatrick. Write amplification reduction in flash-based SSDs through extent-based temperature identification. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[111] Zili Shao, Yongpan Liu, Yiran Chen, and Tao Li. Utilizing PCM for energy optimization in embedded systems. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2012)*, pages 398–403, 2012.

[112] Kai Shen, Stan Park, and Meng Zhu. Journaling of journal is (almost) free. In *USENIX Conference on File and Storage Technologies (FAST 2014)*, pages 287–293, 2014.

[113] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. Optimizing flash-based key-value cache systems. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[114] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. Didacache: a deep integration of device and application for flash based key-value caching. In *USENIX Conference on File and Storage Technologies (FAST 17)*, 2017.

[115] Sharad Sinha and Wei Zhang. Low-power FPGA design using memorization-based approximate computing. *IEEE Transactions on Very Large Scale Integration Systems (VLSI 2016)*, pages 2665–2678, 2016.

[116] SnappyDB. SnappyDB: a key-value database for Android. http://www.snappydb.com/.

[117] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending SSD lifetimes with disk-based write caches. In *USENIX Conference on File and Storage Technologies (FAST 10)*, 2010.

[118] Apache SQLite. https://www.sqlite.org/.

[119] T13. T13 documents referring to TRIM. https://goo.gl/5oYarv.

[120] L. Takac and M. Zabovsky. Data analysis in public social networks. In *International Scientific Conference and International Workshop on Present Day Trends of Innovations*, 2012.

[121] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: advanced photo caching on flash for facebook. In *USENIX Conference on File and Storage Technologies (FAST 15)*, 2015.

[122] Devesh Tiwari, Sudharshan S Vazhkudai, Youngjae Kim, Xiaosong Ma, Simona Boboila, and Peter Desnoyers. Reducing data movement costs using energy-efficient, active computation on SSD. In *USENIX Conference on Power-Aware Computing and Systems (HotPower 12)*, 2012.

[123] Twitter. Fatcache. https://github.com/twitter/fatcache.

[124] UnQlite. UnQlite: an embeddable NoSQL database engine. https://unqlite.org/.

[125] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *ACM Proceedings of the Ninth European Conference on Computer Systems (EuroSys 14)*, 2014.

[126] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. Lsm-trie: an LSM-tree-based ultra-large key-value store for small data. In *USENIX Conference on Usenix Annual Technical Conference (ATC 2015)*, pages 71–82, 2015.

[127] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Hao Tong, Swaminatahan Sundararaman, Andrew A Chien, and Haryadi S Gunawi. Tiny-tail flash: near-perfect elimination of garbage collection tail latencies in nand ssds. In *USENIX Conference on File and Storage Technologies (FAST 17)*, 2017.

[128] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *IEEE 12th International Conference on Data Mining (ICDN)*, Dec 2012.

[129] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't stack your log on my log. In *Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*, 2014.

[130] Heng Zhang, Mingkai Dong, and Haibo Chen. Efficient and available in-memory KV-store with hybrid erasure coding and replication. In *USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.

[131] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. ParaFS: a log-structured file system to exploit the internal parallelism of flash devices. In *USENIX Annual Technical Conference (ATC 16)*, 2016.

[132] Junbo Zhang, Jian-Syuan Wong, Tianrui Li, and Yi Pan. A comparison of parallel large-scale knowledge acquisition using rough set theory on different mapreduce runtime systems. *Elsevier International Journal of Approximate Reasoning*, 55:896–907, 2014.

[133] Yiying Zhang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for flash-based SSDs with nameless writes. In *USENIX Conference on File and Storage Technologies (FAST 12)*, 2012.

[134] Yiying Zhang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Removing the costs and retaining the benefits of flash-based SSD virtualization with FSDV. In *International Conference on Massive Storage Systems and Technology (MSST 15)*, 2015.

[135] Yiying Zhang, Gokul Soundararajan, Mark W Storer, Lakshmi N Bairavasundaram, Sethuraman Subbiah, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Warming up storage-level caches with bonfire. In *USENIX Conference on File and Storage Technologies (FAST 13)*, 2013.

[136] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S Yang, Bill W Zhao, and Shashank Singh. Torturing databases for fun and profit. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[137] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. Understanding the robustness of SSDs under power fault. In *USENIX Conference on File and Storage Technologies (FAST 13)*, 2013.

[138] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX Annual Technical Conference (ATC 2015)*, 2015.