March 2020

# Understanding and Optimizing Flash-based Key-value Systems in Data Centers

Yichen Jia
*Louisiana State University and Agricultural and Mechanical College*

# UNDERSTANDING AND OPTIMIZING FLASH-BASED KEY-VALUE SYSTEMS IN DATA CENTERS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science and Engineering

by
Yichen Jia
B.S., Jilin University, 2013
May 2020

## ACKNOWLEDGEMENTS

I would like express my gratitude and appreciation to the following people, without whom I would not have been able to complete this dissertation, and without whom I would not have made it through my Ph.D degree.

First of all, I am deeply indebted to my advisor, Dr. Feng Chen, for his consistent support and guidance during this research. On the academic level, he taught me fundamentals of conducting scientific research in computer systems, from selecting topics to defining research problems, from searching solutions to designing experiments, from writing papers to giving presentations. On the personal level, his hardworking and passionate attitude towards research have also deeply impressed and inspired me. Dr. Feng Chen has made my Ph.D study a wonderful journey and full of unforgettable memories.

Besides my advisor, I would like to thank other committee members (Dr. Konstantin Busch, Dr. Jian Zhang, and Dr. Kevin S. McCarter) for their constructive feedback and insightful comments to shape the final dissertation. I would also like to thank many other professors, such as Dr. Lu Peng, Dr. Evangelos Triantaphyllou, Dr. Seung-Jong Park, Dr. Jianhua Chen, Dr. Rahul Shah, Dr. Gerald Baumgartner, and Dr. Bijaya B. Karki, for their wonderful courses.

I am also grateful to my industrial collaborators at Arm, Inc. More specifically, I would like to thank Dr. Andrea Pellegrini, Dr. Eric Anger, Dr. Dongyuan Zhan, Jeremy Linton, Peter Droppa, and James Yang for their great mentorship and guidance during my internship. Thanks should also go to my academic collaborators: Dr. Zili Shao at The Chinese University of Hong Kong and Dr. Zhaoyan Shen at Shandong University for their valuable suggestions and efforts.

I also thank my labmates that include Dr. Binbing Hou, Jace Courville, Jian Liu, and Kefei Wang for the stimulating discussions and all the fun we have had. In particular, I am grateful to my undergraduate advisor, Dr. Tieru Wu, at Jilin University, Changchun for enlightening me the first glance of research and my manager, Bin Qi, at Appsoft, Beijing

for sharing me his valuable career experience.

Last but not least, I would like to extend my deepest gratitude to my family, especially my parents Baosheng Jia and Fan'e Qu. They have provided me continued encouragement and endless support throughout my life.

The works described in this dissertation were partially supported by U.S. National Science Foundation under Grants CCF-1453705, CCF-1629291, and CCF-1910958, Louisiana Board of Regents under Grant LEQSF(2014-17)-RD-A-01, Research Grants Council of the Hong Kong Special Administrative Region, China under Grants GRF 15222315, GRF 15273616, GRF 15206617, and GRF 15224918, and Direct Grant for Research, The Chinese University of Hong Kong (Project No. 4055096).

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# ABSTRACT

Flash-based key-value systems are widely deployed in today's data centers for providing high-speed data processing services. These systems deploy flash-friendly data structures, such as slab and Log Structured Merge(LSM) tree, on flash-based Solid State Drives(SSDs) and provide efficient solutions in caching and storage scenarios. With the rapid evolution of data centers, there appear plenty of challenges and opportunities for future optimizations.

In this dissertation, we focus on understanding and optimizing flash-based key-value systems from the perspective of workloads, software, and hardware as data centers evolve. We first propose an on-line compression scheme, called *SlimCache*, considering the unique characteristics of key-value workloads, to virtually enlarge the cache space, increase the hit ratio, and improve the cache performance. Furthermore, to appropriately configure increasingly complex modern key-value data systems, which can have more than 50 parameters with additional hardware and system settings, we quantitatively study and compare five multi-objective optimization methods for auto-tuning the performance of an LSM-tree based key-value store in terms of throughput, the 99th percentile tail latency, convergence time, real-time system throughput, and the iteration process, etc. Last but not least, we conduct an in-depth, comprehensive measurement work on flash-optimized key-value stores with recently emerging 3D XPoint SSDs. We reveal several unexpected bottlenecks in the current key-value store design and present three exemplary case studies to showcase the efficacy of removing these bottlenecks with simple methods on 3D XPoint SSDs. Our experimental results show that our proposed solutions significantly outperform traditional methods. Our study also contributes to providing system implications for auto-tuning the key-value system on flash-based SSDs and optimizing it on revolutionary 3D XPoint based SSDs.

# CHAPTER 1
# INTRODUCTION

The modern data centers are exciting places and are quickly evolving at an unprecedented pace. Contemporary society generates, uses, and retains huge amount of data, yet International Data Corporation(IDC) expects continuous data exposition in the following years [119]. According to the recent survey, the data created by the world is predicted to reach 163ZB by 2025, which is a tenfold increase from the amount of data generated in 2016 [119]. Data centers are playing an increasingly important role in storing and processing these data. In today's data centers, hard disk drives(HDDs) are still the main storage media to provide storage services [119]. However, as flash memory technologies advance, the capacities and endurance of flash-based solid state drives(SSDs) increase and their prices continue to fall. As a consequence, flash storage constitutes ever-growing percentage in data centers, and the flash array market worldwide soars past $11 billion in 2014 [42]. According to IDC [119], SSDs comprise about 10% of all the enterprise drive shipments in 2020 and are predicted to be nearly 20% by 2025. Flash SSDs are eroding HDD's long-dominant position in data center storage.

Flash-based SSDs show unique properties and great performance advantages compared with HDDs. By removing the traditional physical rotating parts, SSDs are based on semiconductor chips and provide significantly higher throughput, lower latency, and lower power consumption than mechanical disks [27, 28]. Besides, because of the rich internal parallelism resources, SSDs allow quick and efficient processing of intensive parallel I/Os [28, 75]. However, flash-based SSDs also have technical constraints, such as "no in-place overwrite" and "sequential-only writes" requirements [27, 28]. As a consequence, software designers have to be aware of these limitations and propose mechanisms to alleviate their impacts to fully exploit the performance potentials of flash-based SSDs.

The efficient and simple key-value model is suitable for flash-based SSDs. Flash-based key-value systems, including slab-based [40, 51, 151] and Log Structured Merge(LSM) tree

based ones [135, 138, 141, 154], are proposed to use simple key-value method to store, retrieve, and manage data. With this lightweight model, flash-based key-value systems gain great popularity either in caching scenario to eliminate expensive data retrieval from backend databases [40, 51, 71, 129], or in storage scenario to provide persistent data services [56, 135, 138, 141, 154].

Despite the great success of key-value systems in data centers, there still exist plenty of challenges and opportunities for system designers and entrepreneurs to leverage emerging trend and technologies to build high performance storage systems. The rapid evolution of data centers raises three critical issues that motivate this work.

**Critical Issue 1:** Workloads grow huge and incur scalability problem to the system. A recent analysis [12] has collected more than 284 billion requests over 58 days on five different Facebook's Memcached [140] servers, showing unique size distribution and access pattern. Although flash-based key-value caching systems [51, 151] can effectively alleviate the high cost and high power consumption problems of in-memory key-value caches [140, 146], it becomes more challenging to manage a huge amount of small-size(e.g., tens to hundreds of bytes) key-value pairs [12] efficiently on significantly larger(e.g., 10-100X) flash-based SSDs than DRAM. Therefore, further optimization for flash-based key-value systems are necessary to improve the caching effectiveness(e.g., hit ratio) by considering the unique characteristics of key-value workloads [12] and properties of flash memories [8, 27, 28].

**Critical Issue 2:** Software becomes complex and hard to tune. According to prior studies [9, 20, 45, 79, 89, 99], software, including flash-based key-value systems, becomes increasingly complex as new features and versions are released. For example, according to the official descriptions [137], facebook's RocksDB [138] has embedded over 50 parameters in the latest version. As a consequence, it is difficult for database administrators to fully understand the impact of these new features to the overall system and their dependencies. Thus, tuning continuously evolving and increasingly complex flash-based key-value systems becomes challenging and necessary, especially considering users' different Quality

of Service(QoS) requirements.

**Critical Issue 3:** Hardware evolves rapidly and causes many of the existing optimizations obsolete. As flash-based SSDs are being shipped to data centers worldwide, 3D XPoint based SSDs [59, 107], which are based on the Non-volatile Memory (NVM) technology, recently become mature and available in the market. Since 3D XPoint memory uses faster and more reliable materials for both selector and storage parts of memory cell [61], the long-existing read-write speed disparity, slow random write, and endurance problems in flash-based SSDs are significantly alleviated. According to Micron, compared to NAND flash, 3D XPoint provides up to 1,000 times lower latency and multiple orders of magnitude greater endurance [107]. As a consequence, as we transit from flash to 3D XPoint in the future, our prior understanding and optimization schemes for flash-based key-value systems may not be readily applicable to the newly emerging 3D XPoint based SSDs.

All the above mentioned issues stem from the rapid evolution of workloads, software, and hardware in modern data centers. This motivates us to understand the impact of these new trends and to optimize the flash-based key-value systems to provide better data processing services.

### 1.1 Dissertation Statement and Contributions

To design optimization schemes for the state-of-art flash-based key-value systems and provide a comprehensive study of them when the next-generation storage medium becomes available, we strive to understand and enhance flash-based key-value systems as workloads, software, and hardware evolve. The main contributions of this dissertation can be summarized as follows:

- We first present an adaptive on-line compression scheme for key-value caching in flash, called *SlimCache*, to improve the caching efficiency for significantly larger cache space and more intensive workloads. SlimCache identifies the key-value pairs that are suitable for compression, applies the compression/decompression algorithms at the proper granularity and expands the virtual flash caching space effectively. Our

proposed compression mechanism can effectively accommodate more key-value data in cache, which in turn significantly increases cache hit ratio and improves system performance for data-intensive applications in data centers. This work is also reported in our research papers [71].

- To appropriately configure a complicated LSM-tree based database, RocksDB [138], we study five multiple objective optimization(MOO) algorithms on the data store to achieve multiple Service Level Objectives(SLOs) simultaneously and quantitatively compare their performance. Through extensive experiments and quantitative analysis, we obtain several important findings related to the effectiveness and efficiency of auto-tuning algorithms. We find that although their efficacy differs, all the five MOO algorithms are able to converge to the (near-)optimal configuration, if given enough time. However, the best algorithm always differs according to the users' QoS requirements. We further present several system implications for system designers and practitioners for future optimizations for RocksDB.

- To understand the behaviors of RocksDB, which is the state-of-art LSM-tree based key-value store and particularly optimized for flash SSDs, on the newly released 3D XPoint SSDs, we further conduct the first, in-depth performance study on the impact of the next generation storage hardware to RocksDB. Through comprehensive evaluation, we reveal several unexpected system bottlenecks in the current RocksDB's design, which hinder us from fully exploiting the great performance potentials of 3D XPoint technology. Based on our observations, we also present three case studies to showcase the efficacy of overcoming these performance bottlenecks with simple approaches. We further discuss the system implications of our findings for system designers and practitioner to develop schemes in future optimizations. Our study shows that many of the current LSM-tree based key-value store designs need to be carefully revisited to effectively incorporate with the new generation hardware to

realize high-speed data processing.

## 1.2 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter 2 introduces the related background. Chapter 3 presents an online compression scheme, called *SlimCache*, to improve the cache hit ratio by virtually expanding the usable cache space through data compression. Chapter 4 presents our quantitative study and comparison of five multi-objective optimization methods to auto-tune RocksDB, which is a representative Log-Structured Merge (LSM) tree based data store. Chapter 5 presents our in-depth performance study of the state-of-art LSM tree based data store on the emerging 3D XPoint SSDs and reveals several unexpected performance bottlenecks and potentials of the current system designs. Chapter 6 discusses the limitations of our work presented in the dissertation and our future work, and Chapter 7 concludes the dissertation.

# CHAPTER 2
# BACKGROUND

In this chapter, we introduce the background of flash memory, key-value caching system and LSM-tree based databases.

## 2.1 Flash Memory

NAND flash is a type of EEPROM devices. Typically a flash memory chip is composed of several *planes*, and each plane has thousands of *blocks*. A block is further divided into multiple *pages*. NAND flash memory has three unique characteristics: (1) *Read/write speed disparity.* Typically, a flash page read is fast (e.g., 25-100 $\mu$s), but a write is slower (e.g., 200-900 $\mu$s). An erase must be conducted in blocks and is time-consuming (e.g., 1.5-3.5 ms). (2) *No in-place update.* A flash page cannot be overwritten once it is programmed. The entire block must be erased before writing any flash page. (3) *Sequential writes only.* The flash pages in a block must be written in a sequential manner. To address these issues, modern flash SSDs have the Flash Translation Layer (FTL) implemented in device firmware to manage the flash memory chips and to provide a generic Logical Block Address (LBA) interface as a disk drive. More details about flash memory and SSDs can be found in prior studies [8, 26, 27, 28].

## 2.2 Flash-based Key-value Caches

Similar to in-memory key-value caches, such as Memcached [140], flash-based key-value cache systems also adopt a slab-based space management scheme. Here we take Twitter's Fatcache [151] as an example. McDipper [51] has a similar design. In Fatcache, the flash space is divided into fixed-size *slabs*. Each slab is further divided into a group of *slots*, each

Parts of this chapter have been previously published as: Yichen Jia, Zili Shao, and Feng Chen, "SlimCache: Exploiting Data Compression Opportunities in Flash-based Key-Value Caching System", in Proceedings of 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'18), 209-222, 2018. DOI: 10.1109/MASCOTS.2018.00029.


Parts of this chapter have been previously published as: Yichen Jia, Eric Anger, and Feng Chen, "When NVMe over Fabrics Meets Arm: Performance and Implications", in Proceedings of 2019 IEEE 36th International Conference on Massive Storage Systems and Technology (MSST'19), 134-140, 2019. DOI: 10.1109/MSST.2019.000-9. 

of which stores a key-value item. The slots in a slab are of the same size. According to the slot size, slabs are classified into *slab classes*. For a given key-value pair, the smallest slot size that is able to accommodate the item and the related metadata is selected. A *hash table* is maintained in memory to index the key-value pairs stored in flash. A query operation (`GET`) searches the hash table to find the location of the corresponding key value item on flash and then loads that slot into memory. An update operation (`SET`) writes the data to a new location and updates the mapping in the hash table accordingly. A delete operation (`DELETE`) only removes the mapping entry from the hash table. A Garbage Collection (GC) process is responsible for reclaiming the deleted and obsolete items later.

Although in-memory key-value caches and in-flash key-value caches are similar in their structures, they show several remarkable distinctions. (1) *I/O granularity.* The flash SSD is treated as a log-structured storage. Fatcache maintains a small memory buffer for each slab class. This in-memory slab buffer is used to accumulate small slot writes, and when it is filled up, the entire slab is flushed to flash, converting small random writes to large sequential writes. (2) *Data management granularity.* Unlike Memcached, which keeps an object-level LRU list, the capacity-triggered eviction procedure in Fatcache reclaims slabs based on a slab-level FIFO order.

### 2.3 LSM-tree based Key-value Store

Most modern key-value data stores are built on *Log-structured Merge tree* (LSM-tree) structure [43, 44, 56, 138], which is optimized for handling intensive update operations. Typically, the key-value data are stored in both memory and storage devices. A set of *Memtables* are maintained in memory to collect incoming writes first and then flush to the storage device (e.g., a disk or an SSD) in *Sorted Sequence Table* (SST) data files. The related metadata information about SSTs is stored in *Manifest* files. SSTs are logically organized in multiple levels of increasing size, from Level 0 (L0) to Level N (LN) (see Figure 2.1). Except at Level 0, where the SSTs can have overlapping key ranges, the other SSTs at each level (L1 to LN) must have non-overlapping key ranges in a sorted manner.

A background merging process, called *compaction*, routinely runs to remove the deleted and obsolete key-value data. In particular, when the number of L0 files exceeds a predefined threshold, multiple L0 files merge with the L1 files that have overlapping key ranges, generating new L1 files and the input L0 and L1 files being discarded. The compaction processes at other levels are similar. Involving heavy I/O and computation overhead, compaction is considered as the main performance bottleneck in LSM-tree based key-value stores.

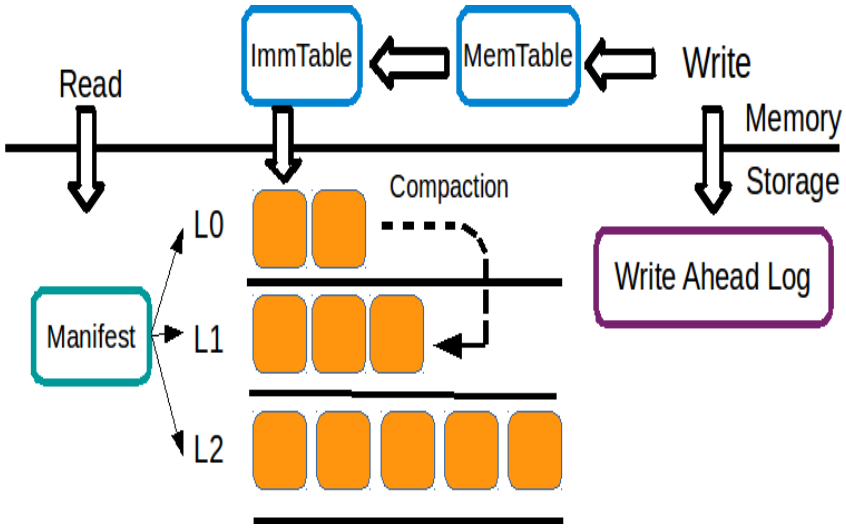



Figure 2.1. An illustration of LSM tree structure

Upon arrival of a write request, the update is firstly written into a Write-Ahead-Log(WAL) for crash recovery. Then the update is accommodated in an in-memory buffer, called *Memtable*. If the size of the Memtable reaches a predefined threshold, it is switched to a read-only *Immutable Memtable*. A new WAL and SST are allocated for holding the subsequent requests. A background thread periodically flushes the Immutable Memtables to persistent storage.

For handling a read request, we first check the Memtables, then the Immutable Memtables, and finally look up the key in the on-storage SSTs, starting from L0 to LN, until the requested item is found. To reduce the I/O cost, techniques, such as *Bloom Filters*, are used to speed up the queries. The block cache in RocksDB and the OS buffer cache can also eliminate unnecessary storage I/O operations.

## 2.4 RocksDB

Compared to LevelDB [56], RocksDB is a popular LSM-tree based key-value store particularly optimized for flash SSD. It employs multiple schemes to exploit the properties of flash SSDs for better performance, such as its rich internal parallelism resources [28]. For example, RocksDB adds multiple *column families* to logically partition the database and group associated keys together; multiple immutable Memtables are used to avoid write stalls; the compaction process is multi-threaded; separate thread pools with different priorities are used for flushing and compaction. After years of tuning, RocksDB has optimized its performance with flash storage, making it highly popular in the industry [43, 135, 139, 141, 143, 166].

## 2.5 3D-XPoint Memory

*3D XPoint* is a type of Non-volatile Memory (NVM) technology [59, 107]. Unlike Phase Change Memory (PCM), 3D XPoint memory uses faster and more reliable materials for both selector and storage parts of memory cell [61]. A 3D transistor-less, cross-point architecture is used by placing the selectors and storage parts of memory cells at the intersection of perpendicular wires, which increases the density and unit capacity [107]. Unlike NAND flash memory, 3D XPoint memory is byte addressable, meaning that it can be accessed in a fine granularity like DRAM. Since 3D XPoint memory does not need an erase operation before write, it can achieve a similar read and write performance and also be more reliable. According to Micron, compared to NAND flash, 3D XPoint provides up to 1,000 times lower latency and multiple orders of magnitude greater endurance [107]. Intel's Optane SSD, which is recently available in the market, is built on 3D XPoint memory. In this dissertation, we use Intel's Optane 900P SSD in our experiments.

# CHAPTER 3
# SLIMCACHE: EXPLOITING DATA COMPRESSION OPPORTUNITIES IN FLASH-BASED KEY-VALUE CACHING

In this chapter , we present our work that applies the compression technique to the flash-based key-value caching system by considering the compression/decompression overhead, characteristics of the workloads and the flash properties to improve the hit ratio and the performance of the overall system.

## 3.1 Introduction

Today's data centers still heavily rely on hard disk drives (HDDs) as their main storage devices. In order to reduce the traffic of requests to backend data stores, in-memory key-value cache systems, such as Memcached [140], become popular in data centers for serving various applications [51, 151]. Although memory-based key-value caches can eliminate a large amount of key-value data retrievals (e.g., "User ID" and "User Name") from the back-end data stores, they also raise concerns on high cost and power consumption issues in a large-scale deployment. As an alternative solution, flash-based key-value cache systems recently have attracted an increasingly high interest in industry. For example, Facebook has deployed a key-value cache system based on flash, called McDipper [51], as a replacement of the expensive Memcached servers. Twitter has a similar key-value cache solution, called Fatcache [151].

### 3.1.1 Motivations

The traditional focus on improving the caching efficiency is to develop sophisticated cache replacement algorithms [105, 72]. Unfortunately, it is highly challenging in the scenario of flash-based key-value caching. This is for two reasons.

First, compared to memory-based key-value cache, such as Memcached, flash-based

Parts of this chapter have been previously published as: Yichen Jia, Zili Shao and Feng Chen, "SlimCache: Exploiting Data Compression Opportunities in Flash-based Key-Value Caching System", in Proceedings of 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 209-222, 2018. DOI: 10.1109/MASCOTS.2018.00029 

key-value caches are usually 10-100 times larger. As key-value items are typically small (e.g., tens to hundreds of bytes), a flash-based key-value cache often needs to maintain billions of key-value items, or even more. Tracking such a huge number of small items in cache management would result in an unaffordable overhead. Also, many advanced cache replacement algorithms, such as ARC [105] and CLOCK-Pro [72], need to maintain a complex data structure and a deep access history (e.g., information about evicted data), making the overhead even more pronounced. Therefore, a complex caching scheme is practically infeasible for flash-based key-value caches.

Second, unlike DRAM, flash memories have several unique technical constraints, such as the well-known "no in-place overwrite" and "sequential-only writes" requirements [8, 27]. As such, flash devices generally favor large, sequential, log-like writes rather than small, random writes. Consequently, flash-based key-value caches do not directly "replace" small key-value items in place as Memcached does. Instead, key-value data are organized and replaced in large coarse-grained chunks, relying on Garbage Collection (GC) to recycle the space occupied by obsolete or deleted data. This unfortunately further reduces the usable cache space and affects the caching efficiency.

For the above two reasons, it is difficult to solely rely on developing a complicated, fine-grained cache replacement algorithm to improve the cache hit ratio for key-value caching in flash. In fact, real-world flash-based key-value cache systems often adopt a simple, coarse-grained caching scheme. For example, Twitter's Fatcache uses a First-In-First-Out (FIFO) policy to manage its cache in a large granularity of slabs (a group of key-value items) [151]. Such a design, we should note, is an unwillingly-made but necessary compromise to fit the needs for caching many small key-value items in flash. This work seeks an *alternative* solution to improve the cache hit ratio. This solution, interestingly, is often ignored in practice—increasing the *effective* cache size.

The key idea is that for a given cache capacity, the data could be compressed to save space, which would "virtually" enlarge the usable cache space and allow us to accommodate

more data in the flash cache, in turn increasing the hit ratio.

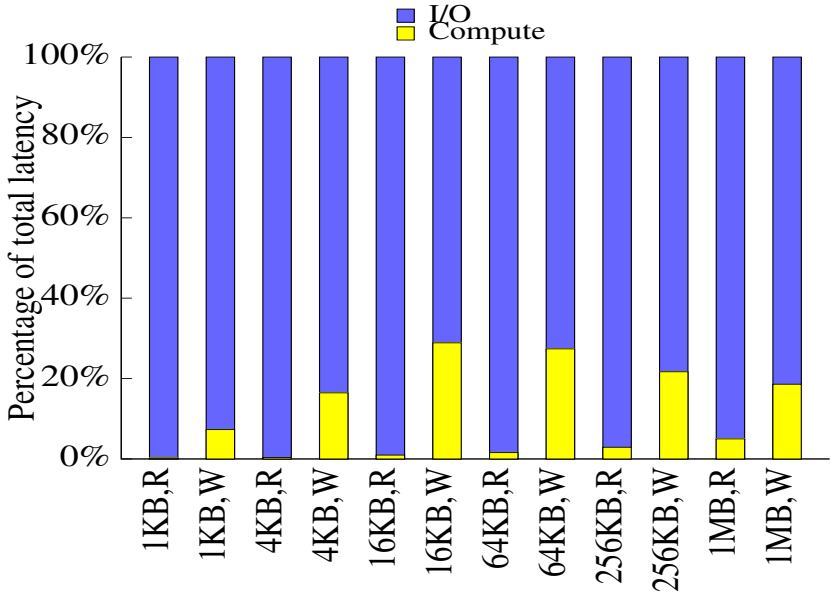



Figure 3.1. I/O time vs. computation time

In fact, on-line compression fits flash devices very well. Figure 3.1 shows the percentage of I/O and computation time for compressing and decompressing random data in different request sizes. The figure illustrates that for read requests, the decompression overhead only contributes a relatively small portion of the total time, less than 2% for requests smaller than 64KB. For write requests, the compression operations are more computationally expensive, contributing for about 10%-30% of the overall time, but it is still at the same order of magnitude compared to an I/O access to flash. Compared to schemes compressing data in memory, such as zExpander [164], the relative computing overhead accounts for an even smaller percentage, indicating that it would be feasible to apply on-line compression in flash-based caches.

### 3.1.2 Challenges and Critical Issues

Though promising, efficiently incorporating on-line compression in flash-based key-value cache systems is non-trivial. Several critical issues must be addressed.

First, various compression algorithms have significantly different compression efficiency and computational overhead [2, 3, 98]. Lightweight algorithms, such as `lz4` [98] and

`snappy` [2], are fast, but only provide moderate compression ratio (i.e., $\frac{uncompressed}{compressed}$); heavy-weight schemes, such as the deflate algorithm used in `gzip` [1] and `zlib` [3], can provide better compression efficacy, but are relatively slow and would incur higher overhead. We need to select a proper algorithm.

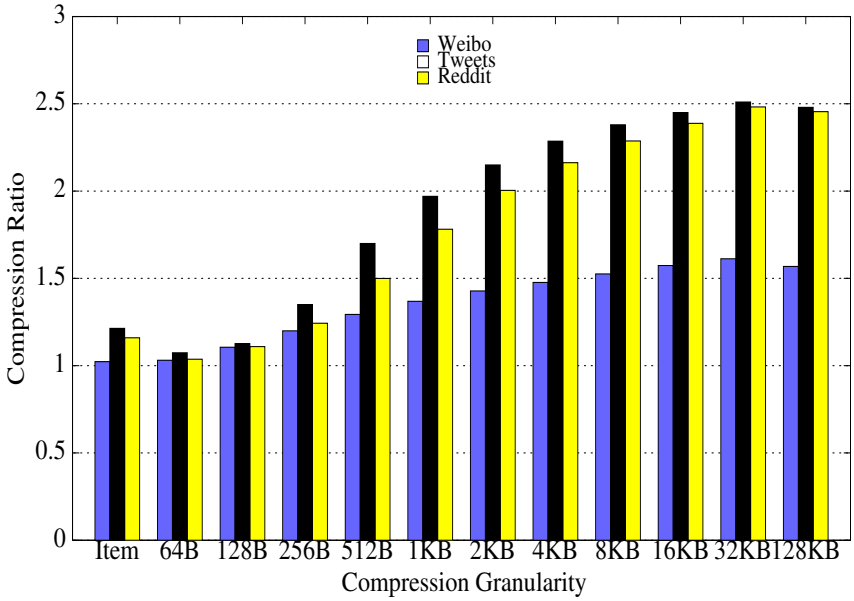



Figure 3.2. Compression ratio vs. granularity.

Second, compression efficiency is highly dependent on the compression unit size. A small unit size suffers from a *low compression ratio* problem, while aggressively using an oversized compression unit could incur a severe *read amplification* problem (i.e., read more than needed). Figure 3.2 shows the average compression ratio of three datasets (Weibo, Tweet, Reddit) with different container sizes. We can see that these three datasets are all compressible, as expected, and a larger compression granularity generally results in a higher compression ratio. In contrast, compressing each key-value item individually or using a small compression granularity (e.g., smaller than 4 KB) cannot reduce the data size effectively. In this chapter we will present an effective scheme, which considers the properties of flash devices, to pack small items into a proper-size container for bulk compression. This scheme allows us to achieve both high compression ratio and low amplification factor.

Third, certain data are unsuitable for compression, either because they are frequently

accessed or simply incompressible, e.g., JPEG images. We need to quickly estimate the data compressibility and conditionally apply on-line compression to minimize the overhead.

Last but not least, we also need to be fully aware of the unique properties of flash devices. For example, flash devices generally favor large and sequential writes. The traditional log-based solution, though being able to avoid generating small and random writes, relies on an asynchronous Garbage Collection (GC) process, which would leave a large amount of obsolete data occupying the precious cache space and negatively affect the cache hit ratio.

All these issues must be well considered for an effective adoption of compression in flash-based key-value caching.

### 3.1.3 Our Solution: SlimCache

In this chapter, we present an adaptive on-line compression scheme for key-value caching in flash, called *SlimCache*. SlimCache identifies key-value items that are suitable for compression, and applies a compression and decompression algorithm at a proper granularity, thus expanding the effectively usable flash space for caching more data.

In SlimCache, the flash cache space is dynamically divided into two separate regions, a hot area and a cold area, to store frequently and infrequently accessed key-value items, respectively. Based on the highly skewed access patterns in key-value systems [12], the majority, infrequently accessed key-value items are cached in flash in a compressed format for the purpose of space saving. A small set of frequently accessed key-value items is cached in their original, uncompressed format to avoid the read amplification and decompression penalty. The partitioning is automatically adjusted based on runtime workloads. In order to create the desired large sequential write pattern on flash, the cache eviction process and the hot/cold data separation mechanism are integrated to minimize the cache space waste caused by data movement between the two areas.

To our best knowledge, SlimCache is the first work introducing compression into flash-based key-value caches. Our compression mechanism achieves both high performance and

high hit ratio by restricting compressed unit within one flash page, dynamically identifying hot/cold data for caching without causing thrashing, and maintaining a large sequential access pattern on flash without wasting cache space. We have implemented a fully functional prototype based on Twitter's Fatcache [151]. Our experimental evaluations on an Intel 910 PCIe SSD have shown that SlimCache can accommodate more key-value items in the cache by up to 223.4%, effectively increasing throughput and reducing average latency by up to 380.1% and 80.7%, respectively. Such an improvement is essential for data-intensive applications in data centers.

The rest of this chapter is organized as follows. Section 3.2 introduces the design of SlimCache. Section 3.3 gives our experimental results. The related work is presented in Section 3.5. The final section concludes this chapter.

## 3.2 Design of SlimCache

In order to fully exploit compression opportunities for key-value caching in flash, we need to carefully consider three critical issues: the compression overhead, the data compressibility and the constraints of flash hardware.

**Compression Overhead**. Though simple, naïvely compressing all key-value data and decompressing them upon every access would incur high computation overhead. We particularly need to separate "hot" and "cold" data and selectively apply compression to them. So we have:

*Rule #1: Do not compress the hot data.*

**Compressibility**. Certain data types, such as multimedia data and encrypted strings, are already compressed or by nature incompressible. So we need a simple mechanism to estimate the compressibility of the target data beforehand and to determine a proper compression granularity to maximize the potential compression efficiency and avoid ineffective compression. So, we have:

*Rule #2: Do not compress the incompressible data.*

**Flash Constraints**. Since the underlying flash memory favors large sequential writes, the compression mechanism should not generate extra small random ones. Considering that all the invalid or duplicated values have to wait for the garbage collection process to asynchronously reclaim their occupied valuable cache space, we need to avoid generating much obsolete data. So, we have:

*Rule #3: Optimization should be flash-aware.*

### 3.2.1 Overview

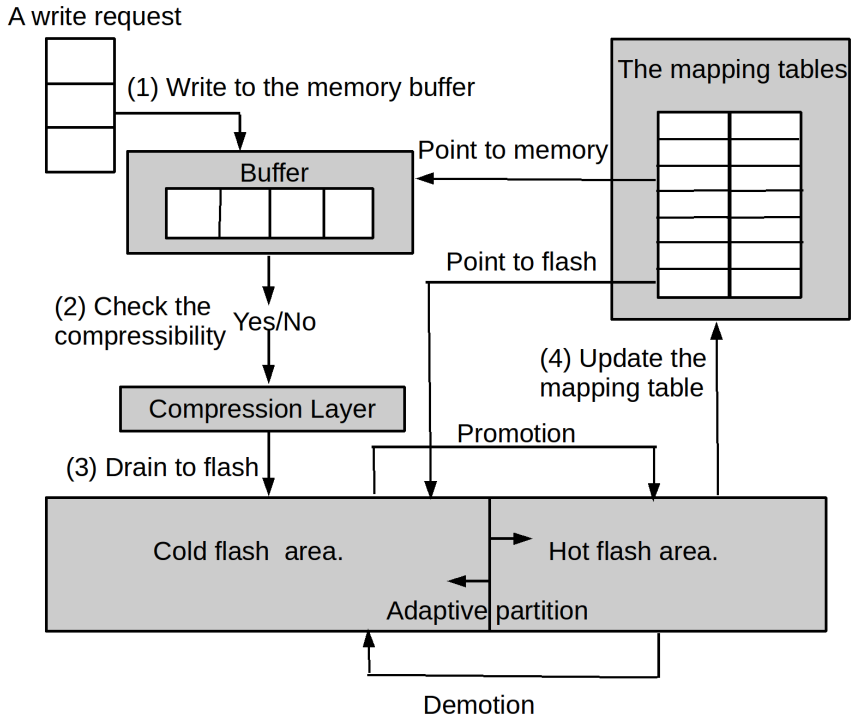



Figure 3.3. An illustration of the SlimCache architecture.

*SlimCache* is a comprehensive on-line compression scheme for flash-based key-value caching. As shown in Figure 3.3, SlimCache adopts a similar structure as Fatcache: A *hash table* is held in memory to manage the mapping from a hashed key to the corresponding value stored in flash, compressed or uncompressed; An *in-memory slab buffer* is maintained for each slab class, which batches up writes to flash and also serves as a temporary staging area for making the compression decision.

Unlike Fatcache, SlimCache has an adaptive *on-line compression layer*, which is re-

sponsible for selectively compressing, decompressing, and managing the flash space. In SlimCache, the flash space is segmented into two areas, a *hot area*, which stores the frequently accessed key-value data, and a *cold area*, which stores the relatively infrequently accessed data. Note that the key-value items in the hot area are stored in the original uncompressed format, which speeds up repeated accesses, while data in the cold area could be stored in either compressed and uncompressed format, depending on their compressibility. The division of the two regions is dynamically determined by the compression module at runtime. In the following, we will explain each of these components.

### 3.2.2 Slab Management

Similar to Fatcache, SlimCache adopts a slab-based space management: The flash space is sliced into *slabs*. A slab is further divided into equal-size *slots*, which is the basic storage unit. Slabs are virtually organized into multiple *slab classes*, according to their slot sizes. Differently, the slab slot in SlimCache can store compressed or uncompressed data. Thus, a slab could contain a mix of compressed slots and uncompressed slots. This design purposefully separates the slab management from the compression module and simplifies the management. A slab could be a *hot slab* or a *cold slab*, depending on its status. The hot slabs in aggregate virtually form the hot area, and similarly, the cold slabs together form the cold area. We will discuss the adaptive partitioning of the two areas later.

**Slab Buffer.** As flash devices favor large and sequential writes, a slab buffer is maintained to collect a full slab of key-value items in memory and write them to the flash in bulk. Upon an update (`PUT`), the item is first stored in the corresponding memory slab and completion is returned immediately. Once the in-memory slab becomes full, it is flushed to flash. Besides asynchronizing flash writes and organizing large sequential writes to flash, the buffer also serves as a staging area to collect compressible data.

**Compression Layer.** SlimCache has a thin compression layer to seamlessly integrate on-line compression into the I/O path. It works as follows. When the in-memory slab buffer is filled up, we iterate through the items in the slab buffer, and place the selected

compressible ones into a *Compression Container* until full. Then an on-line compression algorithm is applied to the container, producing one single *Compressed Key-value Unit*, which represents a group of key-value items in the compressed format. Note that the compressed key-value unit is treated the same as other key-value items and placed back to the in-memory slab buffer, according to its slab class, and waiting for being flushed. In this process, the only difference is that the slot stores data in the compressed format. It is unnecessary for the slab I/O management to be aware of such a difference.

**Mapping Structure.** In SlimCache, each entry of the mapping table could represent two types of mappings. (1) *Key-to-uncompressed-value mapping*: An entry points to a slab slot that contains an original key-value item, which is identical to a regular flash-based key-value cache. (2) *Key-to-compressed-value mapping*: An entry points to the location of a slab slot that contains a compressed key-value unit, to which the key-value item belongs. That means, in SlimCache, multiple keys could map to the same physical location (i.e., a compressed slot in the slab). In the items stored on flash, we add a 1-bit attribute, called *compressed bit*, to differentiate the two situations. Upon a `GET` request, SlimCache first queries the mapping table, loads the corresponding slot from the flash, and depending on its status, returns the key-value item (if uncompressed) or decompresses the compressed key-value unit first and then returns the demanded key-value item.

The above design has two advantages. First, we maximize the reuse of the existing well-designed key-to-slab mapping structure. A compressed key-value unit is treated exactly the same as a regular key-value item—select the best-fit slab slot, append it to the slab, and update the mapping table. Second, it detaches the slab management from the on-line compression module, which is only responsible for deciding whether and how to compress a key-value item. This makes the management more flexible. For example, we can adaptively use different container sizes at runtime, while disregarding the details of storing and transferring data.

### 3.2.3 Compression Granularity

Deciding a proper compression container size is crucial, because the compression unit size directly impacts the compression ratio and the computational overhead. Two straightforward considerations are compressing data in slot granularity or compressing data in slab granularity. Here we discuss the two options and explain our decision.

**Option 1: Compressing Data in Slot Granularity.** A simple method is to directly compress each key-value item individually. However, such a small compression unit would result in a low compression ratio. As reported in prior work [12], in Facebook's Memcached workload, the size of most (about 90%) values is under 500 bytes, which is unfriendly to compression. As shown is Figure 3.4, around 80% of items in the three

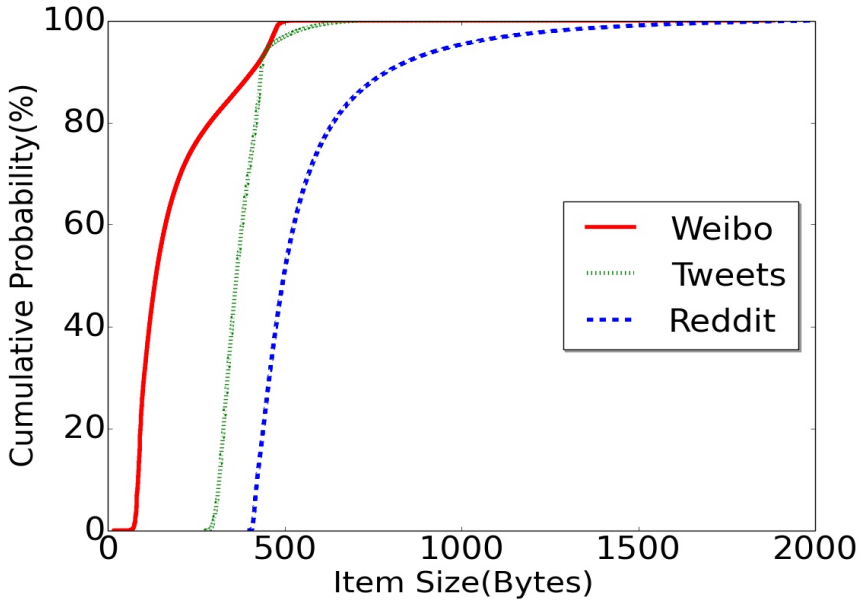



Figure 3.4. Distribution of item sizes.

datasets, Weibo [54, 55], Twitter [152] and Reddit [118], are under 288 bytes, 418 bytes and 637 bytes, respectively. Compressing such small-size values individually suffers from the low-compression-ratio problems (see Figure 3.2), and the space saving by compression would be limited.

**Option 2: Compressing Data in Slab Granularity.** Another natural consideration is to compress the in-memory slab, which is typically large (1 MB in Fatcache as default).

However, upon a request to a key-value item in a compressed slab, the entire compressed slab has to be loaded into memory, decompressed, and then the corresponding item is retrieved from the decompressed slab. This *read amplification* problem incurs two kinds of

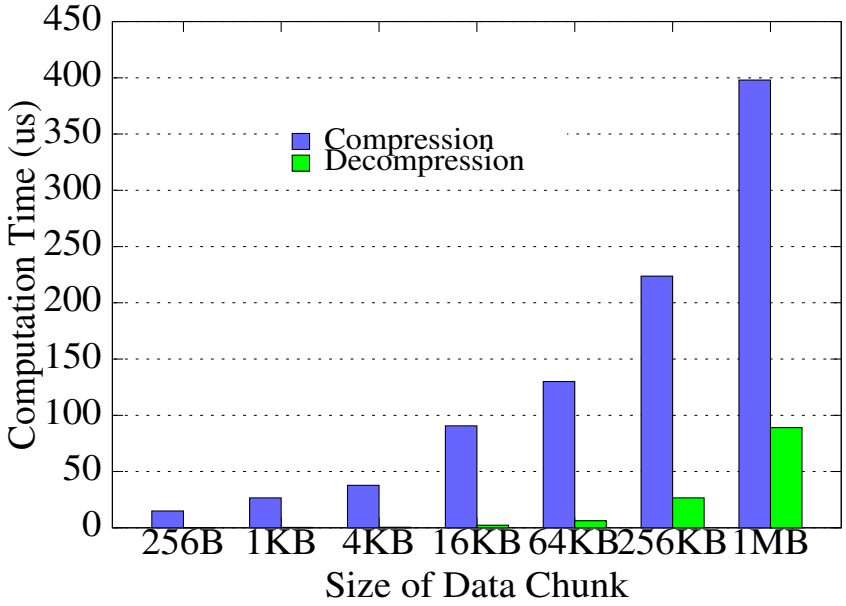



Figure 3.5. Compression time vs. unit size.

overhead. (1) *I/O overhead.* Irrelevant data have to be transferred over the I/O bus, no matter they are needed or not. (2) *Computational overhead.* We apply `lz4` [98], an efficient compression algorithm, on data chunks of different sizes, generated from `/dev/urandom`. As shown in Figure 3.5, the computational overhead becomes non-negligible when the compressed data chunk increases, considering that a flash page read is typically 25-100 $\mu$s. So, compressing data in slabs would cause concerns on the overhead issues.

The above analysis indicates that we must carefully balance between two design goals, achieving a high compression ratio and reducing the overhead. Directly applying compression in either slab or slot granularity would be unsatisfactory. SlimCache attempts to make a `GET` operation completed in no more than one flash page read. We keep track of the compression ratio after each compression operation at runtime, and calculate an average compression ratio, *avg_compression_ratio*, by calculating the arithmetic mean of the measured compression ratio. The estimated compression container size is calculated as

$k \times flash\_page\_size \times avg\_compression\_ratio$, where $flash\_page\_size$ is the known flash page size (typically 4-16 KB), and must be no smaller than a memory page size (4KB as default). The rationale behind this is that we desire to provide the compression algorithm a sufficient amount of data for compression (at least one memory page), and also minimize the extra I/Os of loading irrelevant data (at least one flash page has to be loaded anyway). It is worth noting that the purpose is not to guarantee that the amount of data after being compressed will surely fit into one flash page but to estimate a proper granularity to meet the goal as best efforts. Also, one can adjust the coefficient $k$ according to the properties of the target workloads to achieve the best performance. We set $k = 1$ in the prototype, which works well in our experiments. We will particularly study the effect of compression granularity on the performance in Section 3.3.3.

### 3.2.4 Hot/Cold Data Separation

In order to mitigate the computational overhead, it is important to selectively compress the infrequently accessed data, *cold data*, while leaving the frequently accessed data, *hot data*, in their original format to avoid the read amplification problem and unnecessary decompression overhead. For this purpose, we logically partition the flash space into two regions: The *hot area* contains frequently accessed key-value items in the uncompressed format; the *cold area* contains relatively infrequently accessed key-value items in the compressed format, if compressible (see Figure 3.6). We will present a model-based approach to automatically tune the sizes of the two areas in Section 3.2.5.

**Identifying Hot/Cold Data.** SlimCache labels the "hotness" at the fine-grained key-value item level rather than the slab level, considering that a slab could contain a random collection of key-value items that have completely different localities (hotness). Identifying the hot key-value items rather than hot slabs would provide more accuracy and efficiency. In order to identify the hot key-value items, we add an attribute, called *access_count*, in each entry of the mapping table. When updating a key-value item, its access_count is reset to 0. When the key-value item is accessed, its access_count is incremented by 1. During

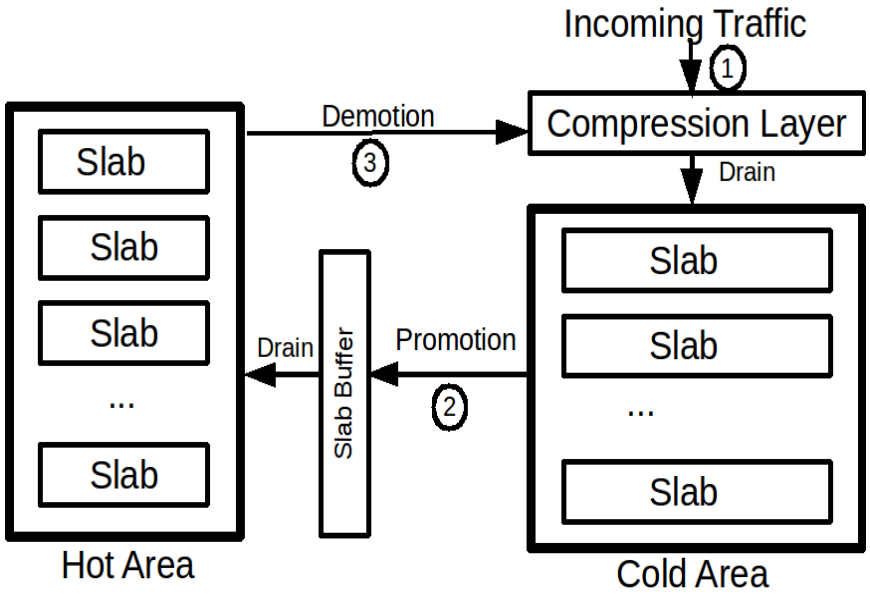



Figure 3.6. Hot and cold data separation

garbage collection, if a compressed key-value item's access_count is greater than zero, it means that this key-value item has been accessed at least once in a compressed format and could be a candidate for promotion to the hot area or continue to stay in the cold area. In Section 3.2.6, we will discuss these two polices. Another issue is how many bits should be reserved for an *access_count*. Intuitively, the more bits, the more precisely we can tell the hotness of a key-value item. We will study this effect in Section 3.3.3.

**Admitting Key-value Items in Cache.** Two options are possible for handling new key-value items. The first one is to insert the newly admitted key-value item into the hot area, and when the hot area runs out of space, we demote the cold items (access_count is 0) into the cold area, compress and "archive" them there. The second method is to first admit the key-value item into the cold area, and when the garbage collection process happens, we decompress and promote the hot items to the hot area. Both approaches have advantages and disadvantages. The former has to write most key-value data at least twice (one to the hot area and the other to the cold area), causing *write amplification*; the latter applies compression in the front, which could cause the decompression overhead if a promotion happens later. Considering the high locality in key-value caches, only a small set

of key-value items is hot and most are cold, the latter solution would remove unnecessary flash writes and thus be more efficient. We choose the second solution in SlimCache.

**Promotion and Demotion.** Key-value items can be promoted from the cold area to the hot area, and vice verse. Our initial implementation adopts a typical promotion approach, which immediately promotes a key-value item upon access, if its access_count is non-zero. However, we soon found a severe problem with this approach—in order to create a log-like access pattern on flash, when a key-value item is promoted into the hot area, its original copy in the cold area cannot be promptly released. Instead, it has to be simply marked as "obsolete" and waits for the garbage collection process to recycle at a later time. During this time window, the occupied space cannot be reused. In our experiments, we have observed a hit ratio loss of 5-10 percentage points (p.p.) caused by this space waste. If we enforce a direct reuse of the flash space occupied by the obsolete key-value items, random writes would be generated to flash.

SlimCache solves this challenging problem in a novel way. Upon a repeated access to a key-value item, we do not immediately promote it to the hot area; rather, we postpone the promotion until the garbage collector scans the slab. In the victim slab, if a key-value item has an access_count greater than the threshold (see Section 3.2.6), we promote it to the hot area and its original space is reclaimed then. In this way, we can ensure that hot data be promoted without causing any space loss, and in the meantime, we still can preserve the sequential write pattern.

In order to determine the coldest slab for demotion from the hot area, the slabs are organized in a linked list and we use the standard Least Recently Used (LRU) replacement algorithm in the slab granularity for eviction. Every time a slab is accessed, it is regarded as the Most Recently Used (MRU) one and moved to the head of the list. When the hot area is full, the Least Recently Used (LRU) hot slab is selected for demotion. Instead of directly dropping all the key-value items, SlimCache compresses the items with a non-zero access_count and demotes them into the cold area, which offers the items that have been

accessed a second chance to stay in cache. For the items that have never been accessed, SlimCache directly drops them since they are unlikely to be accessed again.

In both promotion and demotion, we simply place the compressed/uncompressed key-value items back to the slab buffer, and the slab buffer flushing process is responsible for writing them to flash later. Such a hot/cold data separation scheme is highly effective. In our experiments, the write amplification caused by data movement between the two areas is found rather low (see Section 3.3.3).

### 3.2.5 Adaptive Partitioning

As mentioned above, the partitioning of flash space effectively determines the portion of key-value items being stored in compressed or uncompressed format. The larger the cold region is, the more flash space could be saved, and the higher hit ratio would be; however, the more I/Os have to experience a time-consuming decompression. Thus, we need to first identify a reasonable initial partitioning plan and also provide a dynamic partitioning scheme to reflect the change of workload patterns. We use a simple model-based solution for such adaptive partitioning.

**Initializing Partitions.** If we assume the workload distribution follows the Zipf's law [172, 18, 125], a small portion of records will serve most of the requests. The Zipfian distribution has been extensively studied. In the following, we adopt the expressions defined in prior work [85] to explain how we determine the initial partition ratio. As defined in the work [85], Zipfian distribution has the random variable $X$ and parameters $\alpha$ and $N$, and the probability is

$$f(x) = \frac{1}{x^{\alpha} \sum_{i=1}^{N} (\frac{1}{i})^{\alpha}}, x = 1, 2, ..., N. \tag{3.1}$$

where $N$ is a positive integer and $\alpha \geq 0$. The true Zipf's law [172] has $\alpha = 1$, and a broader class of Zipf-like distributions [18] has $0 < \alpha < 1$ and close to 1. If we represent

the summation in the denominator as

$$H_{N,\alpha} = \sum_{i=1}^{N} (\frac{1}{i})^{\alpha} \tag{3.2}$$

the cumulative distribution function on the support of $X$ becomes

$$F(x) = P(X \leq x) = \frac{H_{x,\alpha}}{H_{N,\alpha}} \tag{3.3}$$

In the case that $\alpha = 1$, asymptotically $F(x) \approx \frac{\ln x}{\ln N}$. If we assume the distribution follows a true Zipf's law, where $\alpha = 1$, according to [85], the population mean of the true Zipf's law is

$$E[X] = \frac{H_{N,\alpha-1}}{H_{N,\alpha}} \tag{3.4}$$

When we set $x$ to $E[X]$, the cumulative distribution function becomes

$$F(E[X]) \approx \frac{\ln(\frac{N}{\ln N})}{\ln N} = 1 - \frac{\ln \ln N}{\ln N} \tag{3.5}$$

For more details, one may refer to prior studies [172, 85, 18, 125]. In this chapter, we use Equations 3.4 and 3.5 to determine the initial partition ratio. Two examples are shown as below.

(1) When the number of records in the system is 100 million (i.e., $N = 100M$), $E[X] = \frac{N}{\ln N} = 100M/18.42 = 5.43M$. This means that the update frequency of 5.43M records (i.e., approximately 5.4%) is above the average frequency. The hit ratio of all the 5.43M records is $F(E[X]) \approx 1 - \frac{\ln \ln N}{\ln N} = 1 - 2.91/18.42 = 84.2\%$.

(2) When the number of records in the system is 10 billion (i.e., $N = 10B$), $E[X] = \frac{N}{\ln N} = 10B/23.02 = 434M$. This means that the update frequency of 434M records (i.e., approximately 4.3%) is above the average frequency. The hit ratio of all the 434M records is $F(E[X]) \approx 1 - \frac{\ln \ln N}{\ln N} = 1 - 3.14/23.02 = 86.7\%$.

In our prototype, we set the hot area initially as 5% of the flash space. According to our analysis above, it is expected to satisfy about 85% of service requests to the cache server. Then we use a model-based on-line partitioning method to adaptively adjust the sizes of the two areas at runtime.

**Cost Model based Partitioning.** As mentioned above, there is a tradeoff between the decompression overhead and the cache hit ratio. We propose a simple cost model to estimate the effect of area partitioning.

$$\begin{aligned} Cost = H_{hot} \times C_{hot} + H_{cold} \times C_{cold} \\ +(1 - H_{hot} - H_{cold}) \times C_{miss} \end{aligned} \tag{3.6}$$

$H_{hot}$ and $H_{cold}$ are the ratios of hits contributed by the hot (uncompressed) key-value items and the cold (compressed) key-value items on the flash, respectively. $C_{hot}$ and $C_{cold}$ are the costs when the data is retrieved from the hot and cold areas, respectively. $C_{miss}$ is the cost of fetching data from the backend data store. These parameters can be obtained through runtime measurement. As shown in ALGORITHM 1, our model needs to consider two possible partitioning decisions, increasing or decreasing the hot area size:

Option #1: Increasing hot area size. If the size of the hot area is increased by $S$, more data could be cached in the uncompressed format. The hit ratio contributed by the head $S$ space of the cold area is denoted as $H_{c\_head}$. The hit ratio $H'_{hot}$ provided by the hot area after increasing by $S$ becomes $H_{hot} + H_{c\_head}/compression\_ratio$. The hit ratio $H'_{cold}$ provided by the cold area after decreasing by $S$ becomes $H_{cold} - H_{c\_head}$.

Option #2: Decreasing hot area size. If the size of the hot area is decreased by $S$, there will be less uncompressed data cached. The hit ratio contributed by the tail $S$ space of the hot area is denoted as $H_{h\_tail}$. The hit ratio $H'_{hot}$ provided by the hot area after decreasing by $S$ becomes $H_{hot} - H_{h\_tail}$. Correspondingly, the cold area will grow by $S$, so the hit ratio $H'_{cold}$ provided by the cold area will be increased to $H_{cold} + H_{h\_tail} \times compression\_ratio$.

We compare the current cost with the predicted cost after the possible adjustments. If

the current cost is lower, we keep the current partitioning unchanged. If the predicted cost after increasing or decreasing the hot area is lower, we enlarge or reduce the hot area size, accordingly.

The above-said model is simple yet effective. Other models, such as miss ratio curve [171], could achieve a more precise prediction but is more complex and costly. In our scenario, since multiple factors vary at runtime anyway and the step $S$ is relatively small, the cost estimation based on this simple model works well in our experiments.

**Algorithm 1** DYNAMIC PARTITIONING

Data: compression_ratio, hit_ratio
Result: The partition of flash space
// $H_{hot}$ and $H_{cold}$ mean hit ratio in hot and cold area respectively.
// $init\_hot\_area$, $curr\_hot\_area$ and $op\_hot\_area$ mean
// initialized, current and optimal hot area size, respectively.
$H_{hot}, H_{cold} \leftarrow Hit(init\_hot\_area, compression\_ratio)$;
$op\_hot\_area \leftarrow init\_hot\_area$;
$op\_cost \leftarrow Cost(H_{hot}, H_{cold})$;
$curr\_hot\_area \leftarrow init\_hot\_area$;
$step \leftarrow \{+S, -S\}$
$max\_hot\_area \leftarrow predefined\_threshold$
**procedure** $DYNAMIC\_PARTITIONING$
    **for** $i \leftarrow 0; i \leq 1; i \leftarrow i + 1$ **do**
        $new\_hot\_area \leftarrow curr\_hot\_area + step[i]$;
        $H_{hot}, H_{cold} \leftarrow Hit(new\_hot\_area, compression\_ratio)$;
        $new\_cost \leftarrow Cost(H_{hot}, H_{cold})$;
        **if** $new\_cost \leq op\_cost$ and $|new\_cost - op\_cost| > \epsilon$ **then**
            **if** $new\_hot\_area \leq max\_hot\_area$ **then**
                $op\_cost = new\_cost$;
                $op\_hot\_area \leftarrow new\_hot\_area$;
            **end if**
        **end if**
    **end for**
    $curr\_hot\_area \leftarrow op\_hot\_area$
    $ADJUST\_PARTITION(op\_hot\_area)$
**end procedure**

### 3.2.6 Garbage Collection

Garbage collection is a must-have process in flash-based key-value cache systems. Since flash memory favors large and sequential writes, when certain operations (e.g., SET and

DELETE) create obsolete value items in slabs, we need to write the updated content to a new slab and recycle the obsolete or deleted key-value items at a later time. When the system runs out of free slabs, we need to reclaim their space on flash.

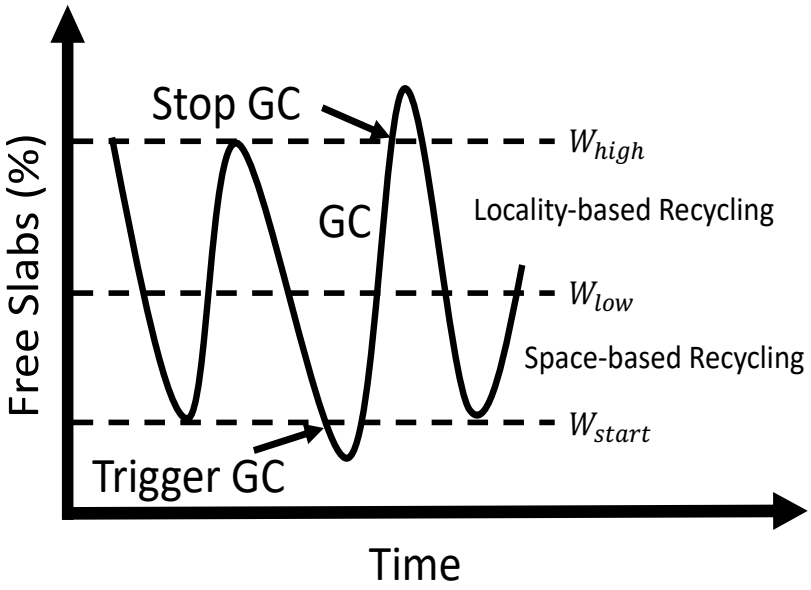



Figure 3.7. An illustration of the two-stage GC.

As Figure 3.7 shows, SlimCache deploys a *Two-stage Garbage Collection* similar to our prior work [129]. When the number of free slabs in cold area of SSD drops to the start watermark ($W_{start}$), *Space-based Eviction* is triggered and quickly cleans slabs. It switches to *Locality-based Recycling*, when the free slab number is brought back to the low watermark ($W_{low}$). The GC process continues until the number of free slab reaches the high watermark ($W_{high}$).

**Space-based Eviction.** When the number of the free slabs in the cold area drops to below the start watermark, $W_{start}$, the space-based eviction process is triggered to release the high space pressure. All the data in the Least Recently Used (LRU) slab, including the valid data, will be dropped directly to reclaim the free space quickly. This is safe, since the backend data store still contains the most recent version. After updating the hash table mapping, the whole slab is put into the free cold slab list. This GC policy aims to reclaim the free space as fast as possible. When the number of free slabs reaches the low

watermark, $W_{low}$, the GC process switches to locality-based recycling.

**Locality-based Recycling.** As Figure 3.8 shows, when the number of the free slabs in the cold area is between the low watermark, $W_{low}$, and the high watermark, $W_{high}$, the locality-based recycling is triggered. We search the slab queue of the cold area to

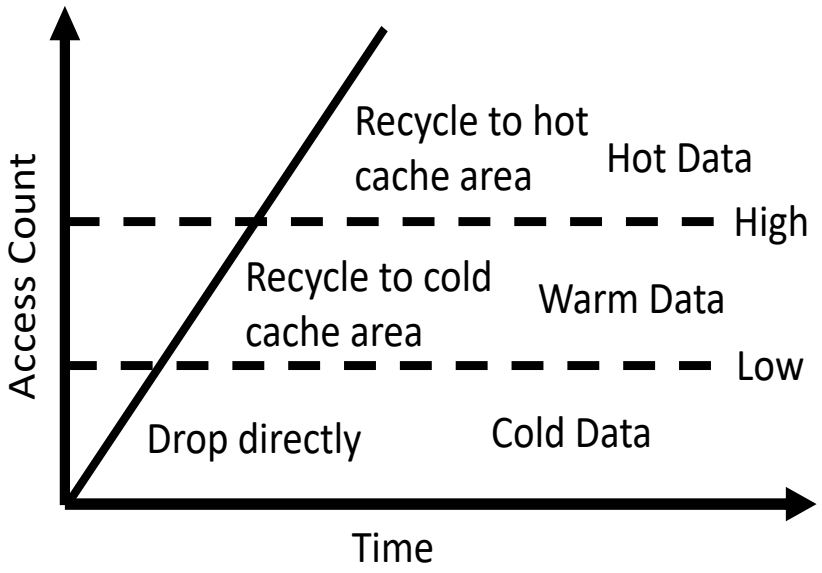



Figure 3.8. Data recycling in garbage collection.

identify the slab that is most frequently accessed. The whole slab is read and based on the access_count, the key-value items can be divided into three categories: *hot*, *warm* and *cold*. Accordingly, we apply different recycling policies for them— the cold or invalid (obsolete or deleted) key-value items are dropped directly; the warm items continue to stay in the cold area in the compressed format; the hot items are decompressed and promoted into the hot area. After updating the hash table mappings, the whole slab is cleaned and placed back to the free cold slab list. Unlike the space-based eviction, this garbage collection procedure takes more time, and collects and promotes valuable items for the purpose of retaining a high hit ratio. When the number of free slabs reaches the high watermark, $W_{high}$, the GC process stops.

These two GC policies are designed for different situations. The space-based eviction is responsible for evicting cold items and aims to reclaim the free space as quickly as possible.

So it is used when SlimCache runs out of free slabs to a severe degree. The locality-based recycling is mainly responsible for collecting and promoting the hot items to retain the hit ratio.

The demotion process in the hot area is similar. When the free space is below the low watermark, $W_{low}$, the LRU slab is selected and all the valid items are compressed and demoted into the cold area. The process repeats until the number of free slabs reaches to the high watermark, $W_{high}$.

### 3.2.7 Dynamic Compressibility Recognition

Some key-value data are incompressible by nature, such as encrypted or already-compressed data, e.g., JPEG images. Compressing them would not bring any benefit but incurs unnecessary overhead. We need to quickly estimate data compressibility and selectively apply compression.

A natural indicator of data compressibility is the *entropy* of the data [128], which is defined as $H = -\sum_{i=1}^{n} p_i \times \log_b p_i$. Entropy quantitatively measures the information density of a data stream based on the appearing probability ($p_i$) of the $n$ unique symbols. It provides a predictive method to estimate the amount of redundant information that could be removed by compression, such as the Huffman encoding [60, 82]. Entropy has been widely used for testing data compressibility in various scenarios, such as primary storage [60], memory cache [29], device firmware [124], image compression [104], and many others. We use *normalized entropy* [160], which is the entropy divided by the maximum entropy ($\log_b n$), to quickly filter out the incompressible data, which are indicated by a high entropy value.

We initialize the threshold to be the average entropy value of randomly generated strings. Since randomly generated strings are mostly incompressible [4], we can effectively skip compression operation for strings whose normalized entropy larger than that of random strings (i.e., incompressible) to remove the unnecessary computation overhead.

We have developed a Dynamic Compressibility Algorithm (DCR) to adaptively adjust

**Algorithm 2** DYNAMIC COMPRESSION RECOGNITION

```
1: Data: init_value, global_min, curr_min
2: Result: threshold for the entropy
3: init_value ← entropy(randomstrings);
4: threshold = init_value;
5: global_min ← predefined_value;
6: curr_min ← max_int;
7: while curr_blk do
8:     if global_min < compr_ratio(curr_blk) < curr_min then
9:         threshold = entropy(curr_blk);
10:        curr_min = max(compr_ratio(curr_blk), global_min);
11:    end if
12: end while
13: UPDATE_THRESHOLD(threshold)
```

the threshold during runtime based on the real-time compression ratio. It works as shown in ALGORITHM 2. The *global_min* is a predefined minimum compression ratio that is acceptable for SlimCache to apply compression operation to gain performance benefits. The *curr_min* is the minimum compression ratio found in the current workload. The entropy *threshold* is updated as the the entropy value of the current data block whose compression ratio is smaller than *curr_min* and larger than *global_min*. The *curr_min* is updated as the greater value of the compression ratios of the current data block and *global_min*. The rationale behind this algorithm is that we first ensure that using the entropy threshold would not result in a compression ratio lower than the acceptable ratio (defined by *global_min*), and we initially set a high entropy threshold to ensure a high compression ratio, and gradually tune down this entropy threshold if we observe an acceptable compression ratio during the run time. In this way, we can find the best cutoff thresholds for different workloads.

The items that are detected incompressible are directly written to the cold area in their original uncompressed format. Thus the cold area could hold a mix of compressed and uncompressed data. This entropy-based estimation fits well in our caching system, especially for its simplicity, low computation cost, and time efficiency. We will study the effect of dynamic compressibility recognition in Section 3.3.3.

### 3.2.8 Summary

SlimCache shares the basic architecture design with regular flash-based key-value caches, such as the slab/slot structure, the mapping table, the in-memory slab buffer, and the garbage collection. However, SlimCache also has several unique designs to realize efficient data compression.

First, we add a compression layer that applies compression algorithms on the suitable items at a proper granularity. The compressed unit is placed back to the slab-based cache structure as regular key-value items, so that the cache space can be consistently allocated and managed. Accordingly, the mapping structure is also modified to point to either compressed or uncompressed items. Second, SlimCache dynamically divides the flash cache space into two separate regions, a hot area and a cold area, to store data in different formats for minimizing the computational overhead caused by compression. Third, SlimCache also enhances the garbage collection process by integrating it with the hot/cold data separation mechanism to avoid the cache space waste caused by data movement between the two areas. Finally, we add compressibility recognition mechanism to identify the data suitable for compression. These differences between SlimCache and a regular flash-based key-value cache, such as Fatcache, contribute to the significant performance gain.

## 3.3 Evaluation

To evaluate the proposed schemes, we have implemented a prototype of SlimCache based on Twitter's Fatcache [151], which has been used in academic works [11, 52, 127, 81, 129] and commercial product benchmarking [122, 121]. Our implementation accounts for about 2,700 lines of code in C. In this section, we present our evaluation results for the SlimCache design on a real SSD hardware platform.

### 3.3.1 Experimental Setup

Our experiments are conducted on three Lenovo ThinkServers. All the three servers feature an Intel Xeon(R) 3.40GHz CPU and 16GB memory. In the key-value cache server, an 800GB Intel 910 PCIe SSD is used as the storage device for key-value caching. Note

that for a fair comparison, only a part of the SSD space (12-24 GB) is used for caching in our experiments, proportionally to the workload dataset size. All the experiments use `direct_io` to minimize the effect of page cache. Our backend data store is MongoDB v3.4.4 running on a separate server with 1TB Seagate 7200RPM hard drive. The clients run on another ThinkServer to generate traffic to drive the experiments. The three servers are connected via a 10Gbps Ethernet switch. For all the three servers, we use Ubuntu 14.04 with Linux kernel 4.4.0-31 and Ext4 file system.

We use Yahoo's YCSB benchmark suite [31] to generate workloads to access the key-value items, following three different distributions, Zipfian, Normal, and Hotspot[1], as described in prior work [21] [164] to simulate typical traffic in cloud services [12]. Since the YCSB workloads do not contain actual data, we use the datasets from Twitter [152], Flickr [63], and Reddit [118] to emulate three typical types of key-value data with different compressibility. The Twitter and Reddit datasets have a high compression ratio (about 2-4), while the Flickr dataset has a low compression ratio, near to 1 (incompressible). In order to generate fixed-size compressible values (Section 3.3.3), we use the text generator [53] based on Markov chain provided by Python to generate the pseudo-random fixed-size values. We use `lz4` [98] and the deflate method in `zlib` [3] for compression in comparison.

In the following, our first set of experiments evaluates the overall system performance with a complete setup, including both the cache server and the backend database. Then we focus on the cache server and study each design component individually. Finally we study the cache partitioning and further give the overhead analysis.

### 3.3.2 Overall Performance

In this section, our experimental system simulates a typical key-value caching environment, which consists of clients, key-value cache servers, and a database server in the

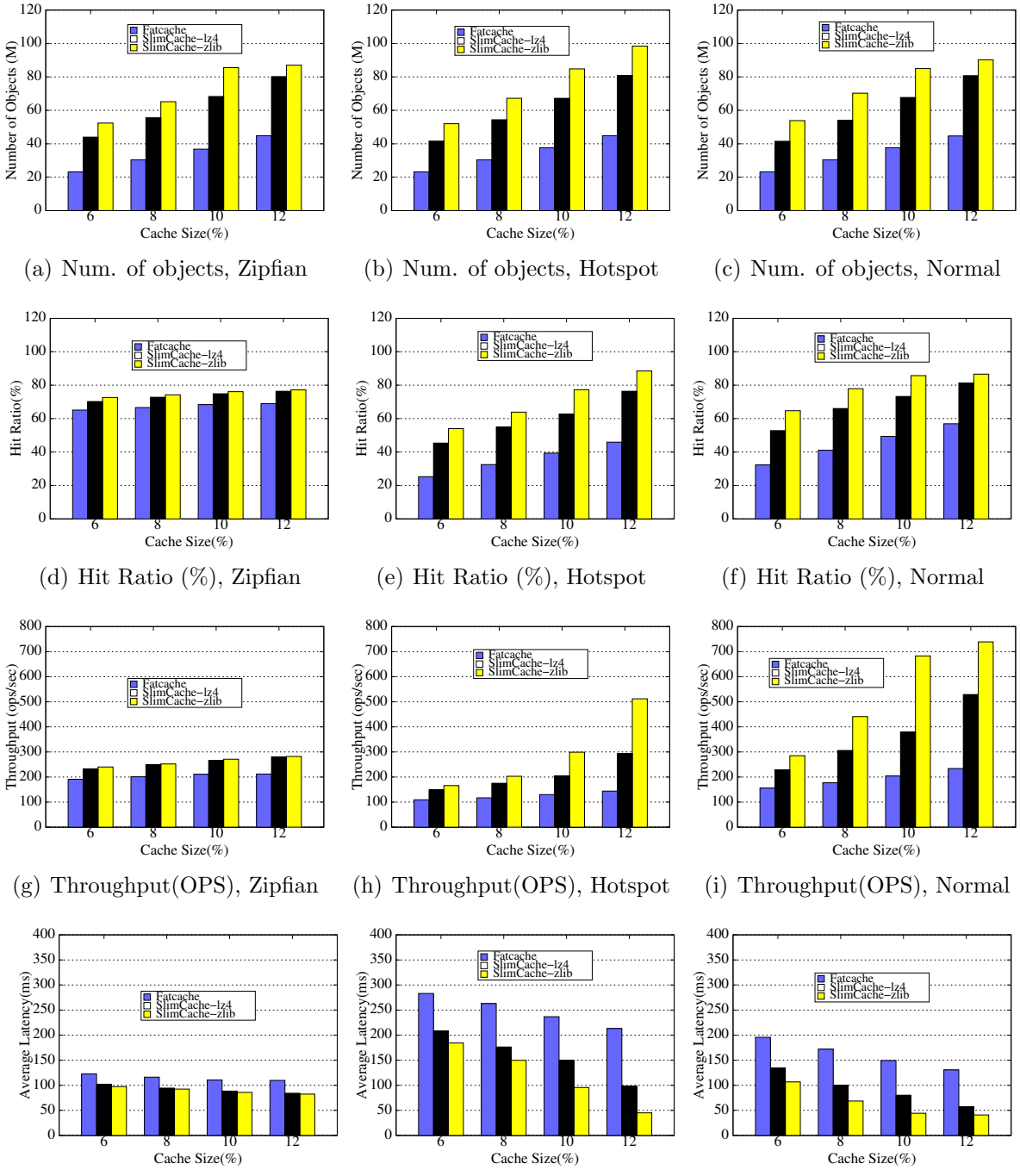
[1]Hotspot is a distribution in which 80% of the operations access 20% of the data items and the rest 20% of the operations access the rest 80% items. Elements for the hot set and cold set are chosen in an uniform manner.

backend. We test the system performance by varying the cache size from 6% to 12% of the dataset size, which is about 200 GB in total (480 million, 300 million, and 2 million records for Twitter, Reddit and Flicker, respectively), where only part of the 800GB SSD capacity is used as cache (12-24 GB). For each test, we first generate the dataset to populate the database, and then generate 300 million `GET` requests. We only collect the data for the last 30 minutes in the trace replaying to ensure that the cache server has been warmed up. All the experiments use 8 key-value cache servers and 32 clients.

**Performance for Twitter Dataset.** Our on-line compression solution can "virtually" enlarge the size of the cache space. Figures 3.9(a), 3.9(c), and 3.9(b) show the number of items cached in SlimCache compared to the stock Fatcache with the same amount of flash space.

As shown in Figure 3.9(a), the number of items in cache increases substantially by up to 125.9%. Such an effect can also be observed in other distributions. Having more items cached in SlimCache means a higher hit ratio. Figures 3.9(d), 3.9(e), and 3.9(f) show the hit ratio difference between Fatcache and SlimCache. In particular, when the cache size is 6% of the dataset, the hit ratio (54%) of SlimCache-zlib for the hotspot distribution is 2.1 times of the hit ratio provided by Fatcache. For the Zipfian and normal distributions, the hit ratio of SlimCache-zlib reaches 72.6% and 64.7%, respectively. A higher hit ratio further results in a higher throughput. As the backend database server runs on a disk drive, the increase of hit ratio in the flash cache can significantly improve the overall system throughput and reduce the latencies. As we can see from Figures 3.9(g), 3.9(h), and 3.9(i), compared to Fatcache, the throughput improvement provided by SlimCache-zlib ranges from 25.7% to 255.6%, and the latency decrease ranges from 20.7% to 78.9%, as shown in Figures 3.9(j), 3.9(k), and 3.9(l).

To further understand the reason of the performance gains, we repeated the experiments with compression disabled. Table 3.1 shows the results with a cache size as 6% of the dataset. We can see that without data compression, solely relying on the two-area (hot




(a) Num. of objects, Zipfian (b) Num. of objects, Hotspot (c) Num. of objects, Normal

(d) Hit Ratio (%), Zipfian (e) Hit Ratio (%), Hotspot (f) Hit Ratio (%), Normal

(g) Throughput(OPS), Zipfian (h) Throughput(OPS), Hotspot (i) Throughput(OPS), Normal

(j) Latency (ms), Zipfian (k) Latency (ms), Hotspot (l) Latency (ms), Normal

Figure 3.9. Performance of Twitter dataset

and cold area) cache design in SlimCache only provides a slight hit ratio increase (1.1-1.5 p.p.) over the stock Fatcache. In contrast, SlimCache with compression provides a more significant hit ratio improvement (5.1-20.8 p.p.). It indicates that the performance gain is mainly a result of the virtually enlarged cache space by on-line compression rather

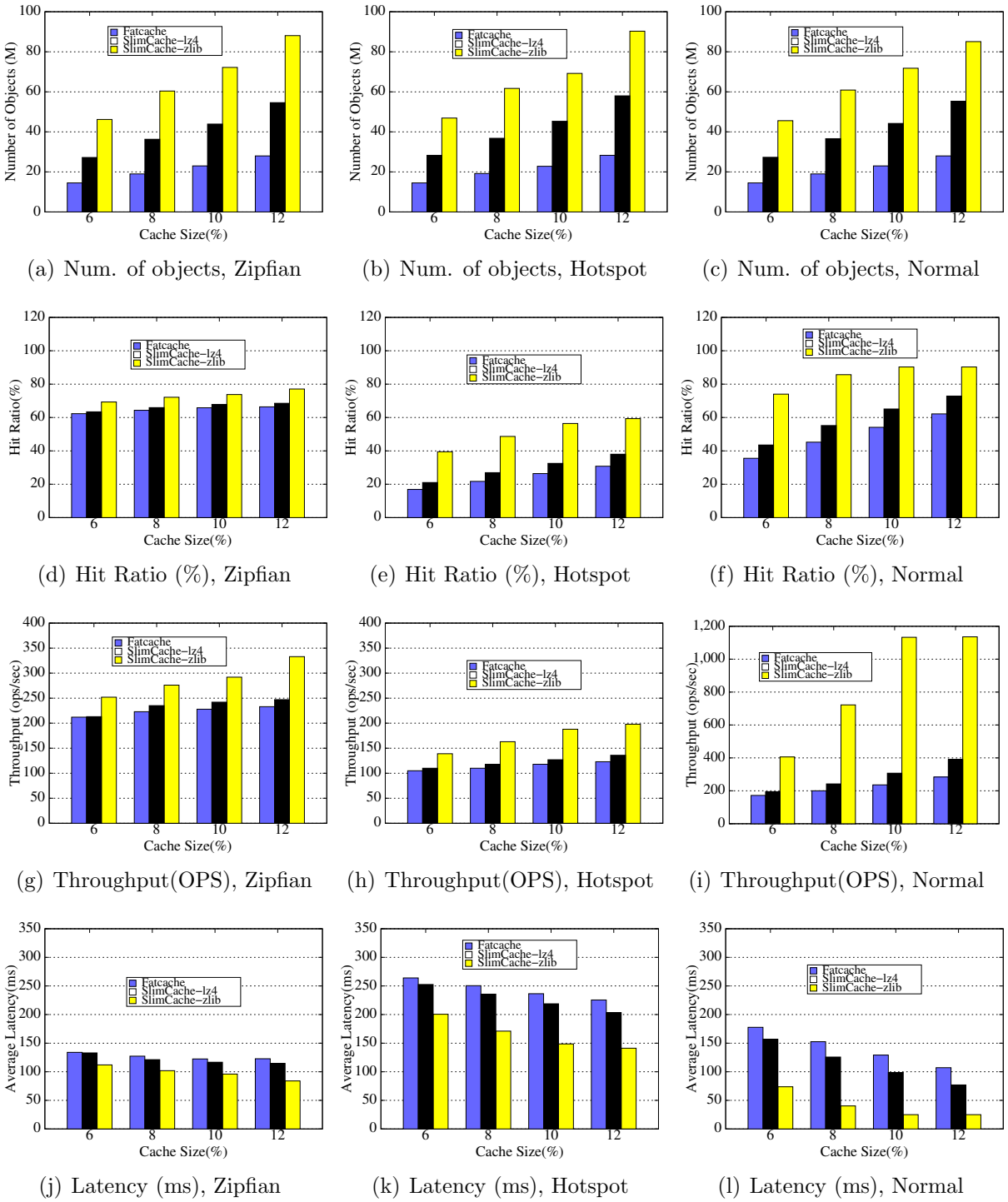
Table 3.1. Hit ratio gain of compression in SlimCache

| Scheme | Zipfian | Hotspot | Normal |
|---|---|---|---|
| Fatcache | 65.1% | 25.2% | 32% |
| SlimCache w/o Compression | 66.2 % | 26.4% | 33.5% |
| SlimCache with lz4 | 70.2 % | 45.4% | 52.8% |

than the two-area cache design.

**Performance for Reddit Dataset.** We further conduct experiments with Reddit on SlimCache to illustrate the effectiveness of our proposed approaches. Figures 3.10(a)-3.10(c) show the number of key-value items cached in SlimCache compared to Fatcache with the same amount of cache space. We can see from Figure 3.10(a) that the number of items cached in SlimCache-zlib increases significantly by to up to 223.4%. Such an increase can also be found with other distributions. More key-value items cached by SlimCache result in a higher hit ratio. We can observe the hit ratio difference between Fatcache and SlimCache from Figures 3.10(d)-3.10(f). For the Zipfian distribution, when the cache size is 6% of the working set, the hit ratio provided by SlimCache-zlib is about 69.3%, which is about 7 p.p. higher than Fatcache. A higher hit ratio further helps improve the throughput. As Figures 3.10(g)-3.10(i) show, the throughput improvement provided by SlimCache-zlib ranges from 18.9% to 380.1%. We can also observe in Figures 3.10(j)- 3.10(l) that, as the cache size increases, the latency decrease ranges from 16.4% to 80.7%. It well shows that SlimCache can gain significant performance improvement for both Twitter and Reddit datasets as shown in Section 3.3.2 and Section 3.3.2.

**Effect of the Compression Algorithms.** We compare the performance of applying three different compression algorithms, the lightweight `lz4`, `snappy`, and heavyweight deflate in `zlib`, when the cache size is 6% of the Twitter dataset. Figure 3.11 shows that `zlib` performs the best among the three, while `lz4` and `snappy` are almost identical. In particular, `zlib` provides a hit ratio gain of 2.4-11.9 p.p. over `lz4` and `snappy,` which results in a throughput increase of 3.4%-25%. Meanwhile, the CPU utilization ratio is up to 2.34% in all the cases as shown in Figure 3.11(c). This indicates that heavyweight compression




(a) Num. of objects, Zipfian (b) Num. of objects, Hotspot (c) Num. of objects, Normal

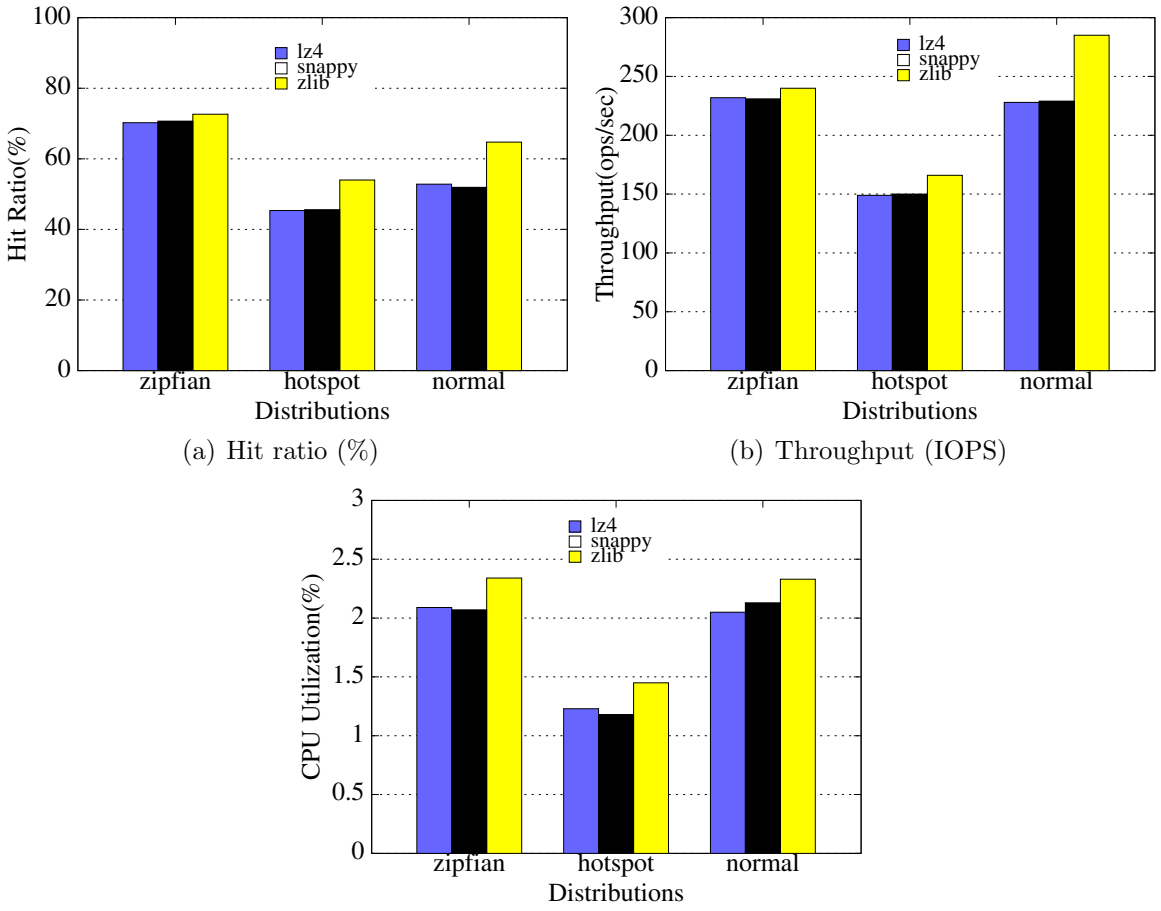
(d) Hit Ratio (%), Zipfian (e) Hit Ratio (%), Hotspot (f) Hit Ratio (%), Normal

(g) Throughput(OPS), Zipfian (h) Throughput(OPS), Hotspot (i) Throughput(OPS), Normal

(j) Latency (ms), Zipfian (k) Latency (ms), Hotspot (l) Latency (ms), Normal

Figure 3.10. Performance of Reddit dataset

algorithms, such as the deflate method in `zlib`, work fine with flash-based caches, since the benefit of increasing the hit ratio significantly outweighs the incurred computational overhead in most of our experiments.



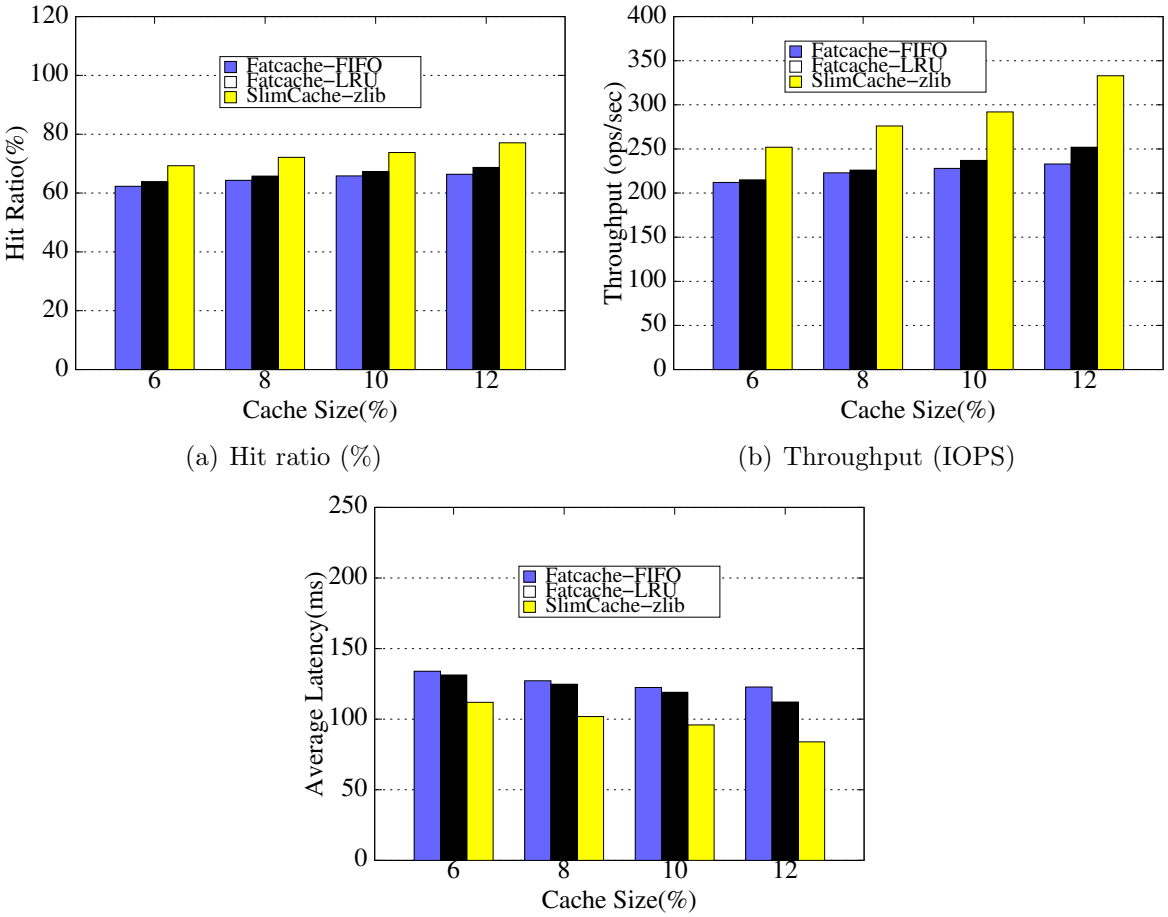

(a) Hit ratio (%)

(b) Throughput (IOPS)

(c) CPU Utilization

Figure 3.11. Effect of different compression algorithms.

**Effect of Replacement Algorithms.** Figure 3.12 shows the effect of different replacement algorithms on the performance of the system for the Reddit workload with Zipfian distribution. We compare Fatcache with the FIFO algorithm (default) and the LRU algorithm with our proposed SlimCache-zlib. Figure 3.12(a) shows that the hit ratio increases from 66.39% to 68.72%, if we change the replacement algorithm from FIFO to LRU for Fatcache when the cache size is 12% of the working set. Accordingly, Figure 3.12(b) shows that the throughput increases from 233 ops/sec to 252 ops/sec and Figure 3.12(c) shows that the average latency decreases from 122.78 ms to 112.17 ms. These experimental results illustrate that the LRU replacement algorithm only slightly improves performance over FIFO. In contrast, SlimCache-zlib outperforms Fatcache-LRU significantly. For example, the throughput of SlimCache-zlib is 32.1% higher than that of Fatcache-LRU when




(a) Hit ratio (%)

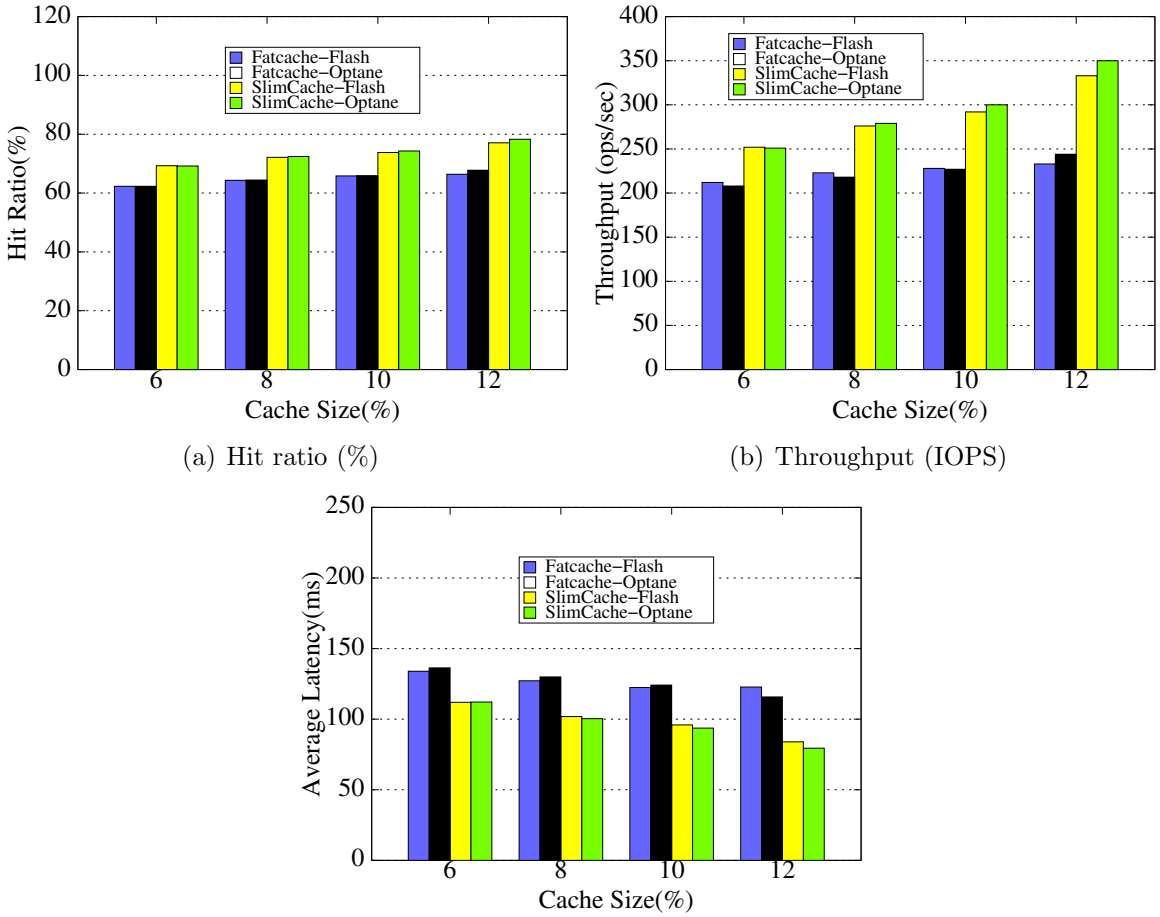
(b) Throughput (IOPS)

(c) Latency (ms)

Figure 3.12. Effect of cache replacement algorithms.

the cache size is 12%. It clearly shows that most of the performance gain of SlimCache-zlib is due to the efficient data compression mechanism, which significantly increases the cache hit ratio.

**Effect of Caching Devices.** Figure 3.13 shows the effect of caching devices on the performance of the system for the Reddit workloads with Zipfian distribution. In this experiment set, we replace the caching device with a 280GB Intel 900P Optane SSD [66], which is built on 3D XPoint non-volatile memory, while keeping the other configurations unchanged. Figure 3.13(a) shows that the conventional NAND flash based SSD and the new 3D XPoint based SSD provide nearly identical hit ratios. If we compare the two devices, as we can see in Figure 3.13(b), when the cache size is 12% of the workload,




(a) Hit ratio (%)

(b) Throughput (IOPS)

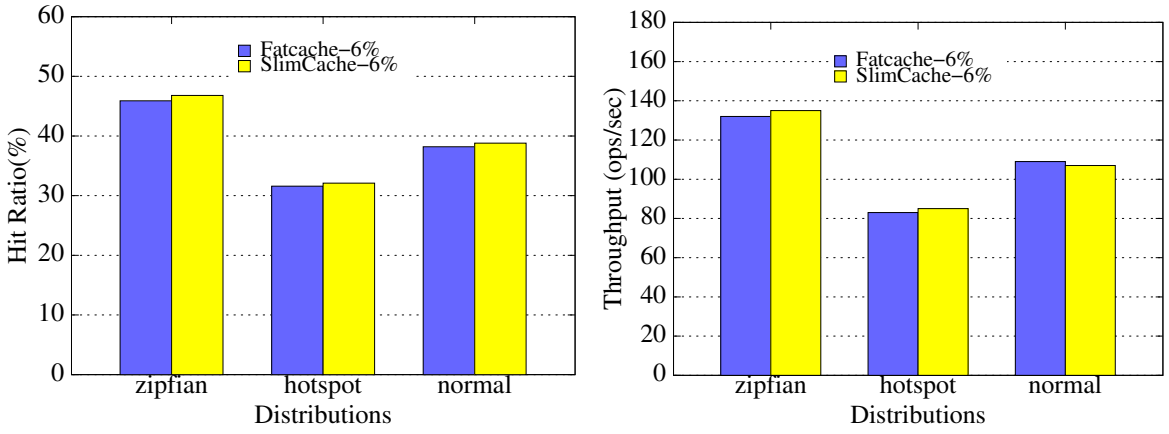
(c) Latency (ms)

Figure 3.13. Effect of caching devices.

Fatcache-Optane can provide 4.7% higher throughput than Fatcache-Flash, and SlimCache-Optane can provide 5.1% higher throughput than SlimCache-Flash. Correspondingly, as Figure 3.13(c) shows, Fatcache-Optane reduces 5.7% average latency than Fatcache-Flash, and SlimCache-Optane reduces 5.3% average latency than SlimCache-Flash. Together with the experimental results shown in Section 3.3.2, we can find that when the speeds of caching devices (SSDs) are on the same order of magnitude, it only incurs slight system performance difference, since the backend data store, which stores data in hard disk drive, is much slower than the caching device. As a consequence, increasing the cache hit ratio, which means fewer accesses being generated to the slow backend data store, would improve system performance more significantly than simply using a faster and more expensive caching

device. This observation further illustrates that SlimCache, which aims to improve hit ratio by caching more key-value items with compression techniques, is a more effective and practically more cost-efficient approach.

**Performance for Flickr Dataset.** We have also studied the performance of SlimCache when handling incompressible data. SlimCache can estimate the compressibility of the cache data, and skip the compression process for the items that are not suitable for compression, such as already-compressed images. We have tested SlimCache with the Flickr dataset and Figure 3.14 shows that for workloads with little compression opportunities, SlimCache can effectively identify and skip such incompressible data and avoid unnecessary overhead, showing nearly identical performance as the stock Fatcache.




(a) Hit ratio (%)

(b) Throughput (OPS)

Figure 3.14. Hit ratio and throughput with Flickr dataset.

### 3.3.3 Cache Server Performance

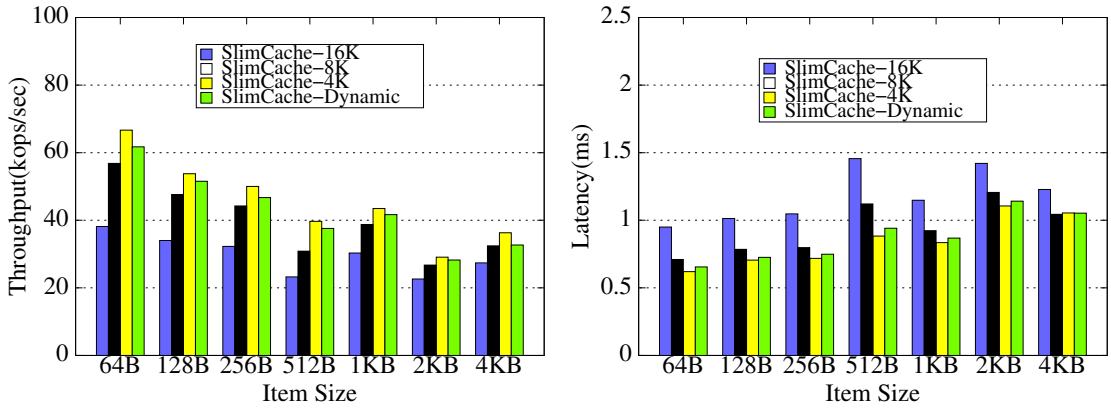
In this section, we study the performance details of the cache server by generating `GET`/`SET` requests directly to the cache server. Since we focus on testing the raw cache server capabilities, there is no backend database server in this set of experiments, if not otherwise specified, and we load about 30GB data using the Twitter dataset to populate the cache server, and generate 10 million `GET`/`SET` requests with Zipfian distribution for the test. All the experiments use 8 key-value cache servers and 32 clients.

**Effect of Compression Granularity.** We first study the effect of compression gran-

ularity. Table 3.2 shows the average compression ratio of fixed-size key-value pairs generated by Markov text generator [53] when compressed individually with lz4. In the following experiments, we compare our proposed dynamic compression granularity with static compression in three large granularities, 4KB, 8KB and 16KB, which achieve the highest compression ratios as shown in the table.

Table 3.2. Compression ratios of the key-value pairs of different sizes.

| **Item Size** | **64B** | **128B** | **256B** | **512B** | **1KB** | **2KB** | **4KB** | **8KB** | **16KB** |
|---|---|---|---|---|---|---|---|---|---|
| Compression Ratio | 0.98 | 1.00 | 1.03 | 1.07 | 1.12 | 1.13 | 1.19 | 1.37 | 1.37 |




(a) Throughput vs. granularity

(b) Latency vs. granularity

Figure 3.15. Effect of different granularity

Figure 3.15(a) and Figure 3.15(b) show the throughput and the average latency of the workload with a GET/SET ratio of 95:5. We vary the fixed-size compression granularity from 4 KB to 16 KB, as comparison to our dynamically adjusted approach (see Section 3.2). It shows that by limiting the size of the compressed items in one flash page, the throughput can be significantly higher than those spreading over multiple flash pages. For example, when the value size is 128 Bytes, if the compression granularity is 16 KB, the throughput is 34K ops/sec, and it increases to 51K ops/sec by using our dynamic method. The improvement is as high as 50%. Figure 3.15(a) also shows that the throughput of the dynamic mechanism is always among the top two and is close to the highest static setting. Figure 3.15(b) shows a similar trend. Using dynamic compression granularity, we can achieve both high compression ratio and high throughput simultaneously.

**Effect of Hot/Cold Data Separation.** Figure 3.16 compares the throughput with and without the hot area for the Twitter dataset with Zipfian distribution. As shown in the figure, the throughput of `GET` operations is 39K ops/sec and 65K ops/sec for SlimCache without and with hot/cold data separation, respectively. Thus, a 66.7% improvement can be achieved with hot/cold separation. Such an improvement can also be seen with other `SET`/`GET` ratios, but when all the requests are `SET` operations, the two mechanisms achieve almost the same throughput. That is because the SET path in SlimCache is identical, no matter the data separation is enabled or not—the items are all batched together and written to the cold area in the compressed format. However, the difference emerges when `GET` operations are involved, because the hot items are promoted to the hot area in uncompressed format, and the following `GET` requests to this item can avoid the unnecessary overhead. Although the hot area only accounts for a small percentage of the cache space, it improves the performance significantly compared to that without hot/cold separation.

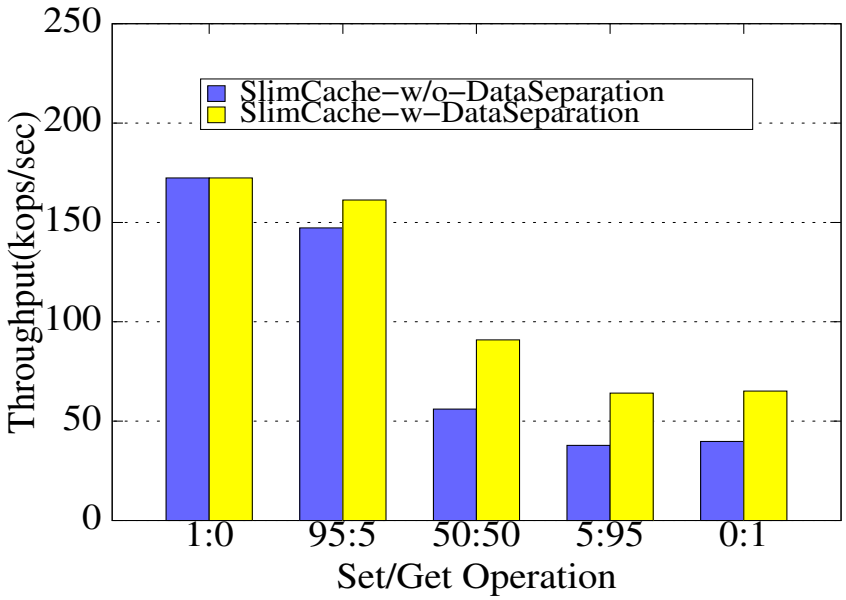
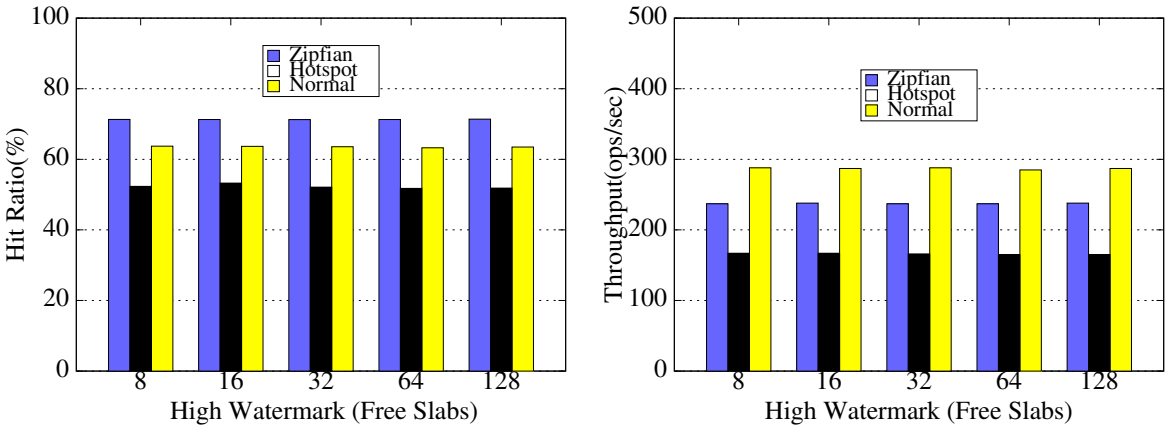



Figure 3.16. Hot/cold data separation

We note that such a great performance improvement is not for free. Frequent data movement between the hot and cold areas may cause a write amplification problem, which is harmful for the performance and also the lifetime of flash. In our experiments, we find that the write amplification factor (WAF) is up to 4.2% in SlimCache, meaning that only

4.2% of the write requests is caused by the switch between the two areas. Since the WAF is quite low and the hot/cold data switch is a background operation, the benefit introduced by hot/cold data separation clearly outweighs its overhead, as shown in Figure 3.16.

**Effect of Two-stage Garbage Collection.** We test the effect of the high watermark $W_{high}$ and low watermark $W_{low}$ to the performance by setting the high watermark from 8 to 128 free slabs, and the low watermark half of the high watermark. For Twitter dataset following different distributions, the performance is insensitive to the watermark settings. This is because the reserved free space only accounts for a very small portion of the entire cache space as shown in Figure 3.17(a) and Figure 3.17(b). In our experiments, we set $W_{high}$=16, $W_{low}$=8 and $W_{start}$=2, which is only about 1% of the overall cache space.




(a) Hit Ratio vs. Watermark

(b) Throughput vs. Watermark

Figure 3.17. Effect of GC watermark

**Effect of Data Recycling.** We investigate the effect of threshold setting for hot, warm and cold data identification during garbage collection, with 300 million requests following Zipfian distribution. The cache size is set 6% of the workload dataset size. Figure 3.18 shows the hit ratio change by setting different thresholds. When the high threshold and the low threshold are both 1 (denoted as H1L1 in the figure), which means that the items will be promoted to the hot area when they are reaccessed at least once and all the rest are dropped directly, the hit ratio reaches the highest, 70.4%, among all the settings. When we vary the threshold settings, the hit ratio drops to about 60%. It indicates that recycling hot data to the hot area is very effective to identify the most valuable data. However, recycling

warm data to the cold area incurs inefficient recollection, since many of the recollected warm data are not frequently reaccessed but occupy the cache space that could be used for other valuable items. Based on the experimental results, we simplify the garbage collection process without recycling warm data to the cold area. Instead, only hot items are promoted to the hot area.

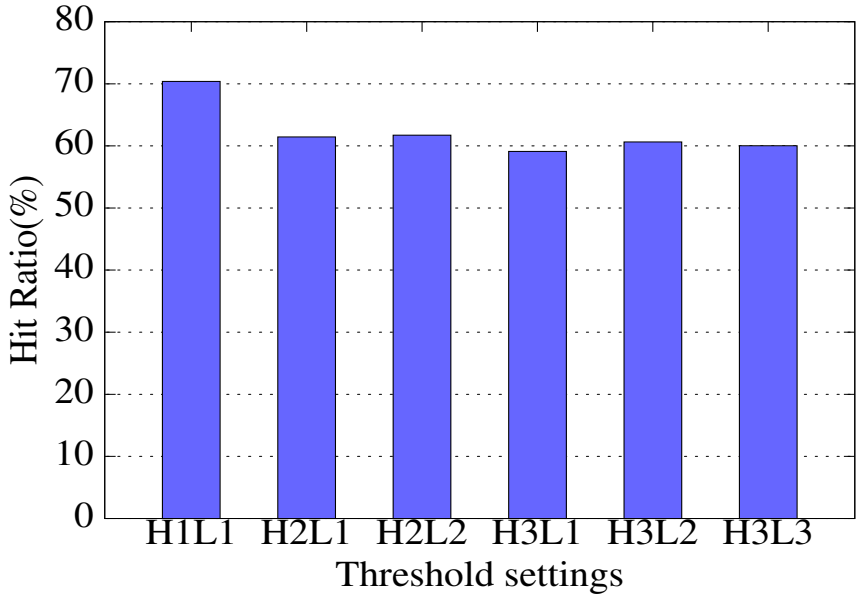



Figure 3.18. Threshold settings in GC

Table 3.3 shows the percentage of GET requests that are served from the hot area when the high threshold and the low threshold are both 1. With a SET:GET ratio of 5:95, 56.7% of the GET requests fall in the hot area, whose size is only 5% of the entire cache space. These results show that the hot/cold data separation can effectively alleviate the read amplification problem caused by on-line compression.

Table 3.3. Ratio of GET requests served in the hot area

| **SET:GET** | **95:5** | **50:50** | **5:95** | **0:100** |
|---|---|---|---|---|
| SlimCache | 79.1 % | 87.3% | 56.7% | 55% |

**Effect of Garbage-Collection-Merged Promotion.** We compare two different promotion approaches. The first one is on-line promotion, which moves the items to the hot area in the uncompressed format immediately after this item is re-accessed. The second one is called Garbage Collection Merged (GCM) promotion, which is used in GC in SlimCache

(see Section 3.2.6). In the GCM promotion, re-accessed items are promoted to the hot area during the GC period. Neither of the two approaches causes extra read overhead, since the on-demand read requests or the embedded GC process needs to read the items or the slab anyway. However, these two methods have both advantages and disadvantages. On-line promotion is prompt, but it wastes extra space, because the original copy of the promoted items would not be recycled until the slab is reclaimed, reducing the usable cache space and harming the hit ratio. On the contrary, the GCM promotion postpones the promotion until the GC process happens, but it does not cause space waste, which is crucial for caching.

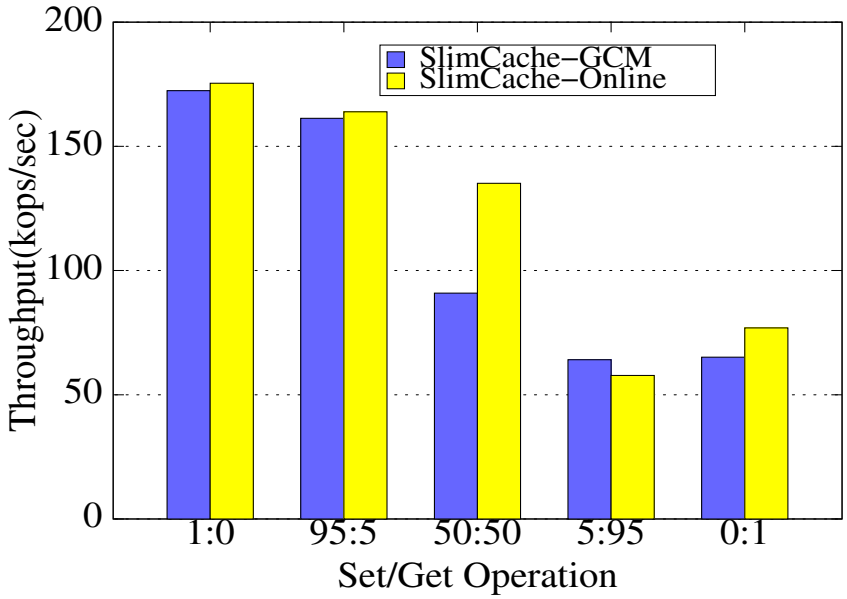



Figure 3.19. Online vs. GCM promotion.

As Figure 3.19 shows, when we test the server without considering the backend database server, the on-line promotion shows a relatively better performance than the GCM promotion, because the on-line compression can timely promote a frequently accessed item into the hot area, reducing the decompression overhead. However, the duplicate copies would incur a waste of cache space. Table 3.4 shows the effect of such a space waste on the hit ratio. We have repeated the Twitter experiments in Section 3.3.2 and set the cache size as 6% of the dataset size. It shows that SlimCache-GCM provides a hit ratio increase of 0.7-7.2 p.p. over SlimCache-Online, which would correspondingly translate into performance gains in cases when a backend database is involved. As space saving for hit ratio

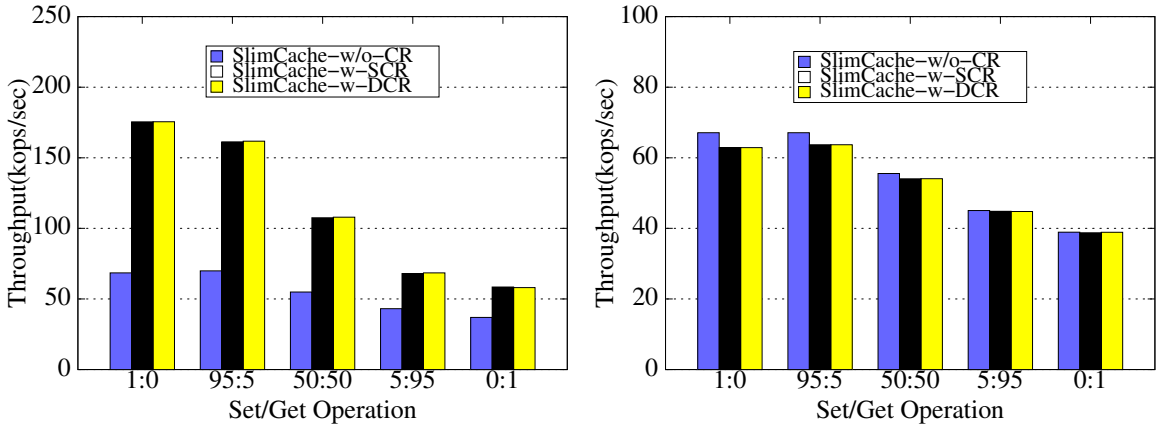
improvement is the main goal of SlimCache, we choose GCM in SlimCache. This highly integrated garbage collection and hot/cold data switch process is specifically customized for flash-based caching system, with significant performance improvement.

Table 3.4. Hit ratio of Online and GCM promotion

| Scheme | Zipfian | Hotspot | Normal |
|---|---|---|---|
| Fatcache | 65.1% | 25.2% | 32% |
| SlimCache-Online | 69.5% | 38.2% | 47% |
| SlimCache-GCM | 70.2% | 45.4% | 52.8% |

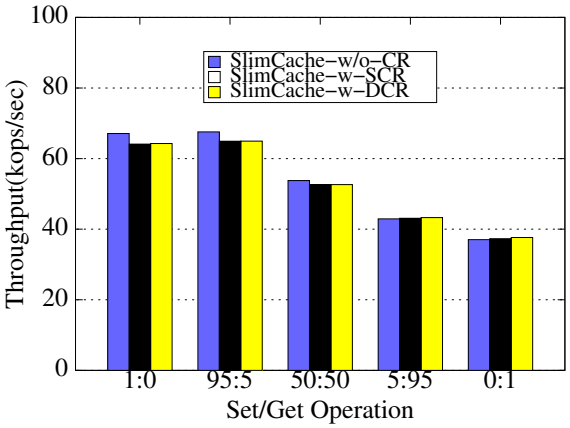
**Effect of Dynamic Compressibility Recognition (DCR).** The dynamic compressibility recognition (DCR) can bring both benefits and overhead. For incompressible data, it can reduce significant overhead by skipping the compression process. However, for compressible data, the compressibility check incurs additional overhead.

We have benchmarked the effect of DCR with Zipfian workloads. Figure 3.20(a) shows the benefit of applying compressibility recognition to the incompressible encrypted dataset. In particular, compressibility recognition improves the throughput by up to 156.1%. In contrast, the DCR mechanism adds overhead for the compressible Twitter dataset, as shown in Figure 3.20(b). We also can see that the overhead is mainly associated with `SET` operations. When the `GET` operations are dominant, which is typical in key-value cache systems, the overhead is minimal. Figure 3.20(c) shows the effect of DCR when the workload is a hybrid compressible (Twitter) and incompressible (encrypted) dataset at the ratio of 1:1. The overhead introduced by DCR is negligible when the dataset is hybrid. Compared to static compressibility recognition (SCR), DCR provides very close performance as shown in Figures 3.20(a), 3.20(b), and 3.20(c). However, DCR provides a more user-friendly interface than SCR since determining the proper compression ratio is more straightforward than determining the proper entropy. Our results show that the DCR mechanism is generally more efficient than compressing all the data indistinctively.




(a) DCR with encrypted data

(b) DCR with Twitter data




(c) DCR with hybrid data

Figure 3.20. Effect of Compressibility Recognition

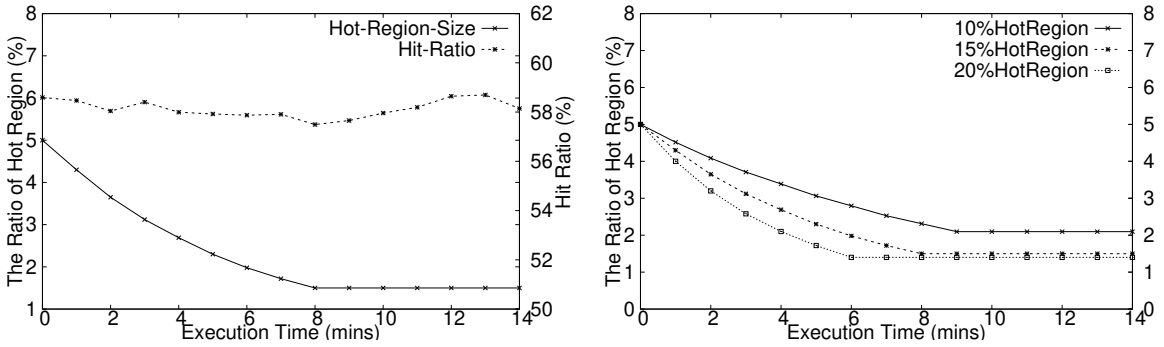
### 3.3.4 Adaptive Partitioning

**Effect of High Miss Cost.** To illustrate the adaptive partitioning, we collect the average read latency to configure our proposed cost model. The hot area cache read is measured 400 $\mu$s, the cold area cache read is 900 $\mu$s, and the backend fetch is 300 ms.

Table 3.5. Parameters used in Dynamic Partition Mechanism I

| Scheme | latency ($\mu$s) |
|---|---|
| Hot area read | 400 |
| Cold area read | 900 |
| Back-end fetch | 300,000 |

Figure 3.21(a) shows the runtime hot area size and the hit ratio when dynamic partitioning happens when the miss cost is high. As the speed of our backend database is slow, SlimCache tends to keep a larger cold area and attempts to reduce the number of




(a) Adaptive partitioning (b) Step $S$ for partitioning

Figure 3.21. Dynamic partition for high miss cost

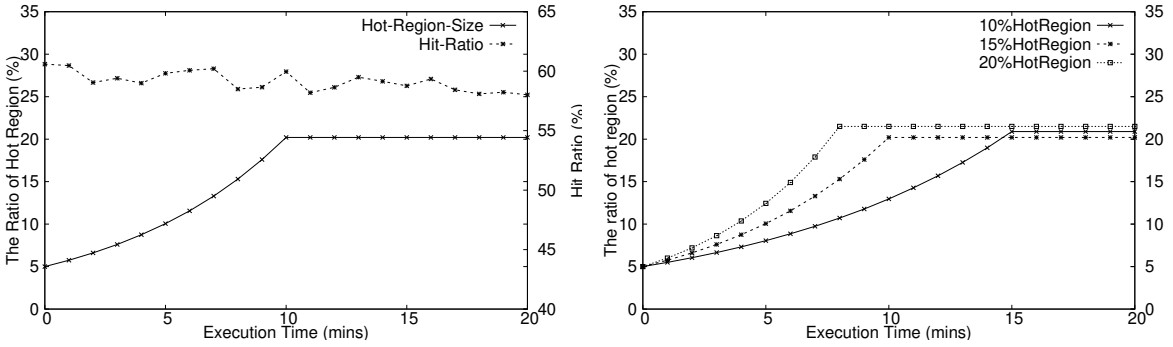
cache misses until the convergence condition is reached. Figure 3.21(a) shows that the hit ratio keeps stable when data migration happens in SlimCache. We have also studied the effect of step $S$ by setting it to 10%, 15%, and 20% of the hot area size. SlimCache can reach a stable cache partitioning within 9 minutes for all the step settings as Figure 3.21(b) shows. Considering that the up-time of a real server is often long (days to months), such a short time for reaching a stable cache partitioning means that our adaptive partitioning approach is reasonably responsive and effective.

**Effect of Low Miss Cost.** In this section, we configure the hot area cache read time to be 400 $\mu$s, the cold area cache read time to be 900 $\mu$s, and the backend access read time to be 1.2 ms to illustrate the adaptive partitioning when the backend access cost is low.

Table 3.6. Parameters used in Dynamic Partition Mechanism II

| Scheme | latency ($\mu$s) |
|---|---|
| Hot area read | 400 |
| Cold area read | 900 |
| Back-end fetch | 1,200 |

Figure 3.22(a) shows the runtime hot area size and the hit ratio, when dynamic partitioning happens in the situation of dealing with a fast backend data store. In this case, as the speed of our backend database is only 33% slower than cache server, SlimCache gradually expands the hot area (to about 20%) to reduce the incurred compression overhead until the convergence condition is reached. This is because with a fast backend data store, the relative benefit of maintaining a large amount of compressed data in the cold




(a) Adaptive partitioning (b) Step $S$ for partitioning

Figure 3.22. Dynamic partition for low miss cost

area decreases. Figure 3.22(a) also shows that the hit ratio keeps stable, when data migration happens in SlimCache. Similar to the high-miss cost case, we also study the effect of step $S$ by setting it to 10%, 15%, and 20% of the hot area size. It takes less than 15 minutes for SlimCache to reach a stable cache partitioning for all the step settings as Figure 3.22(b) shows. Together with results from Section 3.3.4, we can see that no matter the backend database is relatively fast or slow, our adaptive partitioning approach can reach a reasonably good partition quickly (within 15 minutes).

### 3.3.5 Overhead Analysis

SlimCache introduces on-line compression in flash-based key-value cache, which could increase the consumption of CPU, memory and flash resources on the sever side.

**Memory Utilization.** In SlimCache, memory is mainly used for three purposes. (1) In-memory hash table. SlimCache adds a 1-bit *access_count* attribute to record the access count of the item since stored in the system. (2) Slab buffer. SlimCache performance is not sensitive to the memory buffer. We maintain a 128 MB memory for slab buffer, which is identical to Fatcache. (3) Slab metadata. We add a 1-bit attribute for each slab, called *hotslab*. This bit indicates whether the slab belongs to the hot area or not. In total, for a 1TB SSD that stores 1 billion records, SlimCache consumes about 128 MB (128 MB for hash table entry metadata, 128 KB for slab metadata) more memory than Fatcache, which is about 0.3% of the overall memory consumption. In our experiments, we find that the actual memory consumption of SlimCache and Fatcache is similar at runtime.

**CPU Utilization.** SlimCache is multi-threaded. In particular, we maintain one thread for the drain operation, one thread for garbage collection, one thread for data movement between the hot and the cold areas, and one thread for dynamic partitioning. Compression and decompression operations also consume CPU cycles. As shown in Table 3.7, the CPU utilization of SlimCache is less than 3.5% in all our experiments. The main bottleneck is the backend database for the whole system. Computation resource is sufficient on the cache server to complete the demanded work.

Table 3.7. CPU utilization of SlimCache

| **Scheme** | **Zipfian** | | **Hotspot** | | **Normal** | |
|---|---|---|---|---|---|---|
| Cache | 6% | 12% | 6% | 12% | 6% | 12% |
| Fatcache | 1.93% | 2.08% | 1.07% | 1.19% | 1.84% | 2.25% |
| SlimCache | 2.09% | 2.14% | 1.23% | 2.21% | 2.05% | 3.37% |

**Flash Utilization.** We add a 1-bit *compressed* attribute to each key-value item to indicate whether the item is in compressed format or not. This attribute is used to determine if a decompression process should be applied when the slot is read upon a GET operation. Storing 1 billion records will consume 128 MB more flash space, which is a small storage overhead.

## 3.4 Limitations

Although SlimCache can achieve significantly better performance than Fatcache, there are still several limitations that are out of the scope of this work and worth studies in the future.

### 3.4.1 Data Persistence

As a replacement of Memcache, Twitter's Fatcache is not designed to guarantee data persistence in cache. In Fatcache, the mapping structure is completely stored in volatile memory rather than the flash SSD. This design choice removes the related performance overhead and also simplifies the system design, but when a system crash or power failure happens, the key-value data hosted in the flash cache would become invalid. It is worth noting that it is still safe, because the client can always find a copy of the most up-to-date

data in the backend data store. However, the cache system has to warm up again after restart, which could take a long duration. Similar to the stock Fatcache, SlimCache shares the same limitation in terms of cache data persistence. Our main goal in this work is to improve the caching performance and efficiency by adopting data compression techniques in cache management, as demonstrated in SlimCache. As an orthogonal challenge, the cache data persistence issue is worth further our studies in the future.

### 3.4.2 Flash Durability

Since SlimCache divides the logical space of the flash SSD into a hot area and a cold area, the data swapping between the two areas could potentially result in *write amplification*, causing additional amount of writes to the flash SSD and accelerating the wear-out process of the caching device. We find that because of the strict yet effective data swapping policy, the write amplification factor is observed to be about 4.2% in our experiments. Considering the significant performance gain of SlimCache and the continuously decreasing price of flash SSDs, such a relatively small write amplification is considered to be fairly acceptable to most users. Other measures, such as using a flash-based RAID, can also be adopted to reduce the concerns on the durability. We think it is worth future studies on further reducing the impact of flash durability in such scenarios.

## 3.5 Related Work

This chapter presents a highly efficient on-line data compression scheme for enhancing flash-based key-value caching. The two topics, data compression [16, 35, 109, 161, 164, 150, 6, 116] and key-value systems [12, 41, 92, 97, 163, 169, 129, 86], have been extensively researched in the community. This section discusses prior studies that are most related to these components.

Data compression is a popular technique. In prior works, extensive studies have been conducted on compressing memory and storage at both architecture and system levels, such as device firmware [67, 173], storage controller [64], and operating systems [7, 16, 109, 100, 150, 161]. Much prior works have also be done in database systems (e.g., [5, 6, 32,

83, 108, 116]). Our work focuses on applying data compression to improve the hit ratio by caching more key-value items in flash. To our best knowledge, Our SlimCache [71] is the first work introducing data compression into flash-based key-value caching.

Recent research on key-value cache focuses mostly on performance improvement [88, 93, 101], such as network request handling, OS kernel involvement, data structure design, and concurrency control, etc. Recently hardware-centric studies [94], such as FPGA-based design [17], Open-Channel SSD [129] and programmable NIC [86], began to explore the hardware features. In particular, DIDACache [129] provides a holistic flash-based key-value cache using Open-Channel SSD through a deep integration between hardware and software. KV-Direct [86] presents a high performance key-value system through remote direct key-value access to the host memory by extending the RDMA primitives based on programmable NIC. Similarly, NetCache [73] optimizes the queries to hot key-value items and attempts to balance the load across storage nodes by leveraging the flexibility of new programmable switches. Memshare [30] gives a DRAM-based key-value cache system with a dynamic memory management across applications. In order to reduce small random writes in photo caching, RIPQ [134] provides a framework to support advanced caching replacement algorithms with optimized writes on flash devices by collecting small writes, flushing updates lazily, and grouping similar data together. For a similar purpose, Flashield [47] gives a hybrid solution by using DRAM to filter and reduce writes to flash, which addresses the write amplification problem on flash SSDs. In comparison, SlimCache adopts a largely orthogonal approach, data compression, to improve the flash-based key-value cache performance and efficiency.

Besides the performance, some other studies deal with the scalability problem [51, 162, 111, 113], which results from hardware cost and power/thermal problems. For example, Nishtala et al. aim to scale Memcached to handle large amount of Internet traffic in Facebook [111]. Ouyang et al. design an SSD-assisted hybrid memory for Memcached to achieve high performance and low cost [113]. McDipper [51] is a flash-based key-value

cache solution to replace Memcached in Facebook. BlueCache [167] proposes to address the scalability challenges by implementing all the key-value operations including the flash controller operations directly in hardware. Anna [162] is a partitioned, multi-mastered key-value system that can effectively scale from a single core to multicore to the distributed system via wait-free execution and coordination-free consistency. As a scale-up solution, Tucana [115] presents an efficient and high-speed key-value store design for achieving both high performance and low CPU overhead. Cascade Mapping [156] provides a new index mapping structure in key-value caches to address the scalability challenge caused by limited memory resources.

With the popularity of persistent memory, a number of studies [62, 76, 25, 48, 156, 103] have been proposed to integrate non-volatile memory (NVM) within key-value systems. Huang et al. [62] propose to use cross-referencing logs to close the performance gap between the key-value stores in volatile DRAM and persistent NVM. NVMKV [103] optimizes flash-based key-value stores through techniques, such as alleviating dynamic mapping, providing transaction support, and leveraging parallelization. NoveLSM [76] proposes an LSM-tree based design of persistent key-value store by taking advantage of the byte addressability and persistence features of non-volatile memories, such as creating a byte-addressable skip list, directly manipulating persistent state, and exploiting opportunistic read parallelism, etc. HashKV [25] is designed to achieve high update performance based on KV separation and using hash-based data grouping. MyNVM [48] reduces the DRAM footprint of Facebook's key-value store by replacing DRAM with NVM. uDepot [81] presents a key-value design that fully utilizes the performance of fast NVM devices with a two-level indexing structure and a novel task-based IO run-time system. In contrast, the data compression scheme presented in this work is a general-purpose software-level solution without relying on any special hardware.

Among the prior works, zExpander [164], which applies compression in memory-based key-value caches, is the closest to our work. However, SlimCache is particularly designed

for key-value caching in flash, which brings several different and unique challenges. First, small random writes are particularly harmful for the lifetime and performance of flash devices, so storing and querying an item using a small-size (2KB) block on SSD as what zExpander does would be sub-optimal in our scenario. Second, as the amount of key-value items stored in flash-based key-value cache is much larger than that in a memory-based cache, the organization unit has to be much coarser and the metadata overhead brought by each item must be minimized. Third, choosing a proper compression granularity on flash needs to consider the flash page size to minimize the extra I/Os caused by loading irrelevant data. Finally, in order to guarantee that all the writes are sequential in flash, the space occupied by the obsolete values in one slab cannot be freed until the whole slab is dropped. A special mechanism is needed to handle such situations to avoid the loss of hit ratio caused by data promotion and demotion while preserving the sequential write pattern. All these distinctions and new challenges have motivated us to design an efficient, on-line data compression scheme, customized for caching key-value data in flash.

## 3.6 Conclusion

In this chapter, we present an on-line compression mechanism for flash-based key-value cache systems, called SlimCache, which expands the effectively usable cache space, increases the hit ratio, and improves the cache performance. For optimizations, SlimCache introduces a number of techniques, such as unified management for compressed and uncompressed data, dynamically determining compression granularity, efficient hot/cold data separation, optimized garbage collection, and adaptive cache partitioning. Our experiments show that SlimCache can effectively accommodate more key-value data in cache, which in turn significantly increases the cache hit ratio and improves the system performance.

# CHAPTER 4
# KILL TWO BIRDS WITH ONE STONE: AUTO-TUNING ROCKSDB FOR HIGH BANDWIDTH AND LOW LATENCY

In this chapter, we present our work that quantitatively studies and compares five multi-objective optimization(MOO) methods for auto-tuning the performance of increasingly complex LSM-tree based key-value stores, in terms of throughput, the 99th percentile tail latency, convergence time, real-time system throughput, and the iteration process, etc. The evaluation results demonstrate that multi-objective optimization methods can achieve a good balance among multiple targets, which satisfies the unique needs of key-value services.

## 4.1 Introduction

In today's data centers, Log-Structured Merge (LSM) tree based key-value data stores (e.g., LevelDB and RocksDB) are being widely deployed for high-speed data processing. Due to its complex internal structures, a modern key-value data store offers a number of configurable parameters (e.g., buffer size, thread number, table size, etc.), allowing users to tune system performance for various hardware and workloads. After years of optimizations, such a set of configurable parameters becomes indispensable for users to gain fine-grained customizability for performance tuning. For example, RocksDB, a highly popular key-value data store in industry, exposes over 50 tunable parameters to system administrators.

Appropriately configuring a key-value data store system is crucial to the runtime performance. Each configuration parameter controls a certain aspect of the system behavior, such as parallelism degree, I/O size, event-triggering frequency, etc. A selected configuration profile in effect determines the observed performance. Further considering the highly diverse workload and hardware properties in real-world deployment, a configuration that works optimally in one particular scenario may not work equally well in another. In other words, it is unlikely to have a universally optimal setting for all cases. For this reason, performance tuning in the deployment of key-value data stores is an important but notoriously tedious, time-consuming, and case-by-case work.

In the current practice, the industry still heavily relies on a traditional, experience-

based, *hand-tuning* approach, which significantly increases the administration cost and delays the time to deploy. *Auto-tuning*, as an automatic self-adaptive approach, is thus highly appealing. As a general system solution, auto-tuning was originally proposed to overcome the unscalable manual tuning. It has been studied in various scenarios, such as cloud storage, databases, parallel systems, and many others [99, 90, 79, 9, 45]. These prior works focus on optimizing for one objective function, such as throughput, power consumption, or monetary cost, etc. However, such a solution cannot readily suffice the need for quick deployment of a key-value store system.

A unique challenge for auto-tuning key-value data stores is that we must achieve *multiple* Service Level Objectives (SLOs) simultaneously. In a typical enterprise-class application scenario, the data store system needs to guarantee to achieve two important but sometimes conflicting optimization goals, *throughput* and *latency*[1] SLOs. Such a requirement makes auto-tuning in LSM-tree based key-value data stores even more challenging.

In this chapter, we have quantitatively studied and compared five multi-objective optimization algorithms for auto-tuning the performance of RocksDB, a highly popular LSM-tree based key-value store optimized for flash SSDs. To the best of our knowledge, this is the first work that presents quantitative analysis on the efficacy and efficiency of black-box multi-objective optimization methods on RocksDB and provides system-level explanations for the auto-tuning procedure. Our study has been conducted in two stages.

*Stage 1: Data collection*: In order to quantitatively demonstrate the ability of these multi-objective optimization algorithms, we have exhaustively evaluated RocksDB with over 2,000 software configurations on 4 hardware settings under 3 representative workloads. We have executed more than 12,000 experimental runs over 6 months. The performance metrics and related information are maintained in an MySQL [142] database for offline references. The information includes hardware and workload details, parameter settings, throughput, tail latency, etc. In this chapter, we focus on optimizing for two major SLOs,

[1]The main optimization targets for the latency SLOs are typically the $95^{th}$, $99^{th}$ or even $99.9^{th}$ percentile tail latency [36, 68, 87, 165].

*throughput* and *tail latency* (the $99th$ percentile). The methodology that we have used in this chapter can also be applied to achieving other optimization goals, such as power consumption and monetary cost, etc.

*Stage 2: Algorithm Analysis*: We select five representative multi-objective optimization (MOO) algorithms, namely Genetic Algorithms(GA) [14], Non-dominated Sorting Genetic Algorithm II (NSGA-II) [39], Speed-constrained Multi-objective Particle Swarm Optimizer (SMPSO) [110], $\epsilon$-constraint Method (ECM) [168] and Weighted Sum Method (WSM) [102], and apply them to the collected real experimental data to find the (near-)optimal configurations. In our study, the selected five MOO algorithms cover all the three common techniques (e.g. prior, posteriori and no-preference methods as discussed in Section 4.2) used to solve multi-objective optimization problems. Our experimental results show that, although their efficacy differs, all the five MOO algorithms are able to eventually converge to the (near-)optimal configuration, if given enough time. However, the best algorithm always differs according to the QoS requirements. We present 11 findings in this work, which cover our understanding of the auto-tuning algorithms as well as the optimization advice for RocksDB.

It is worth noting that our focus is not to simply find a good parameter setting for a specific hardware or system setup, which may vary on different platforms. Rather, our main objective is to compare and understand the practical efficacy of applying auto-tuning algorithms on RocksDB and to gain important insight on how they behave in real-world deployment. We hope that our findings and the associated system implications will pave the way for system designers and practitioners towards developing a highly efficient and effective auto-tuning solution for key-value data stores.

The rest of this chapter is organized as follows. Section 4.2 gives the background. Section 4.3 introduces the methodology. Section 4.4 presents the comparative analysis on MOO algorithms. Section 4.5 discusses the impact of hyper-parameters. Section 4.6 discusses the related system implications for system designers. Related work is presented

in Section 4.8. The final section concludes this chapter.

## 4.2 Background

### 4.2.1 Multi-objective Optimization Algorithms

Multi-Objective Optimization (MOO) methods deal with optimization problems that have two or more conflicting goals. As defined in prior work [37], the multi-objective optimization problem can be generalized as follows:

$$\underset{x}{Minimized} : F(x) = [F_1(x), F_2(x), ..., F_k(x)]^T \tag{4.1}$$

Subject to:

$$x_i^L \leq x_i \leq x_i^U; i = 1, 2, ..., n$$

$$g_j(x) \leq 0; j = 1, 2, ..., m$$

where $k$ is the number of targets, $m$ is the number of constraints, $x \in R^n$ is a collection of variables $x_i \in [x_i^L, x_i^U]$, $g_j(x)$ are the constraint functions, and $F(x) \in R^k$ is a collection of objective functions $F_i(x)$.

Typically, there is no single solution that can simultaneously optimize all the objectives. A solution is called Pareto optimal [149, 159], if no objective can be improved without degrading the other objectives. Without additional user-specific preference information, all the Pareto optimal solutions are considered equally good.

Based on the articulation of preferences, the multi-objective optimization algorithms can be classified into three main categories. This includes the following: (1) Algorithms with a *priori articulation* of preferences: These methods allow users to specify their preferences in terms of relative importance of all the objectives. (2) Algorithms with a *posteriori articulation* of preferences: When the users cannot provide an explicit preference function, it is effective to allow users to choose from a collection of possible solutions. (3) Algorithms with a *no articulation* of preferences: when the users cannot define their preference

explicitly, this group of methods assume all the objectives are equally important.

### 4.2.2 Applied Methods

**Single Objective Genetic Algorithm (SOGA)**. Genetic Algorithm (GA) [14] be-

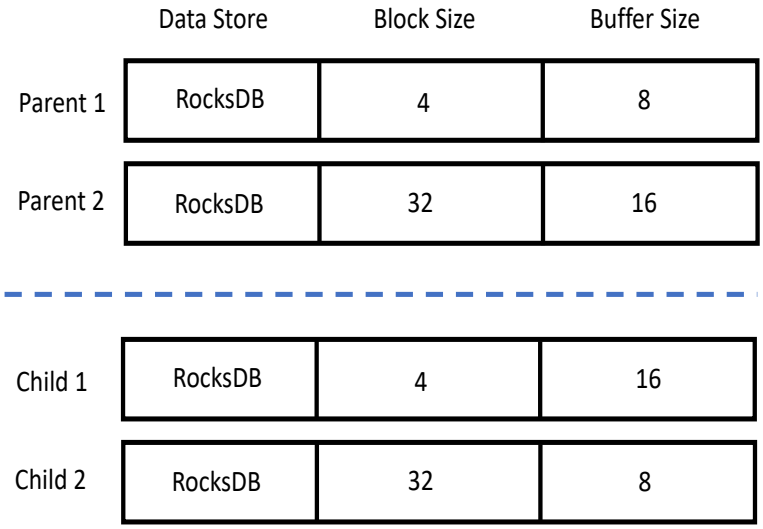



Figure 4.1. Crossover in GA

longs to evolutionary algorithms' family that is designed based on the natural selection process. The initial population is randomly generated. The population size is defined according to the nature of the problem. The fittest portions of the current population are selected based on the user-defined fitness function to create a new generation, during each successive generation. The next generation population is selected through a combination of genetic operators: crossover and mutation. This process guarantees that better genes are inherited with higher probability. The selection process is repeated until a termination condition is reached. Figure 4.1 illustrates the cross-over process in GA. A cross point is selected and the tails of the two parents (e.g., Buffer Size) are swapped to generate new offsprings.

**Non-dominated Sorting Genetic Algorithm (NSGA-II)**. To avoid converging into local optimal solution, NSGA-II [39] is developed to replace single objective genetic algorithms. The process of NSGA-II is illustrated in Figure 4.2. Firstly, the population is

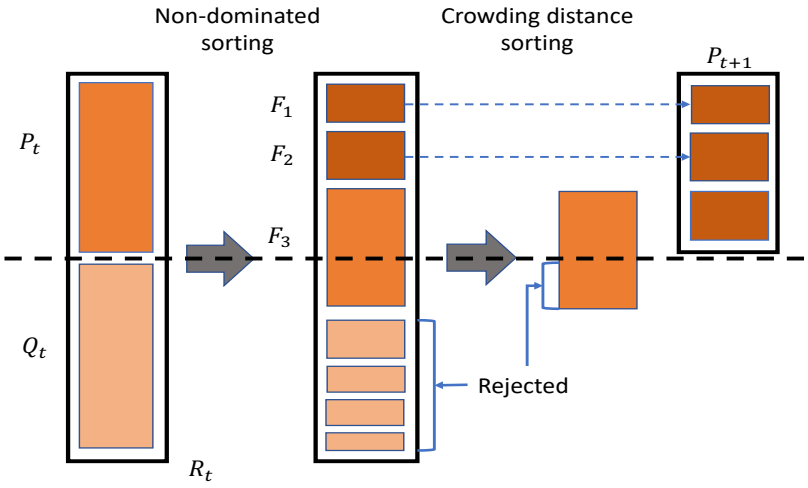



Figure 4.2. Selection in NSGA-II [39]

initialized randomly with predefined population size. All the chromosomes are sorted in the Pareto front based on Pareto Non-dominated sets. The chromosomes in the Pareto front are ranked based on euclidean distance or I-dist between solutions, which are terms defined in NSGA-II. Typically, solutions that are far from others will have a higher probability to be selected to improve diversity and to avoid a crowded solution set. Then the best ones in the current population is put into the mating pool. During the mating process, tournament selection, crossover and mating are conducted to generate offsprings. The offsprings and the current population are combined and sorted to pick the best N chromosomes into new population. The selection process continues until the maximum number of generations are reached. Finally, the highest ranked Pareto optimal solutions from the latest population are chosen as the final solutions.

**Speed-constrained Particle Swarm Optimization (SMPSO)**. Particle swarm optimization (PSO) [155] algorithm is developed by simulating the social behaviors of animals, such as insects, herds, fish, and birds. These swarms cooperate to find food by sharing information among members. Specifically, as Figure 4.3 shows, PSO updates its new position of the particle by combining the optimal position of the swarm and that of

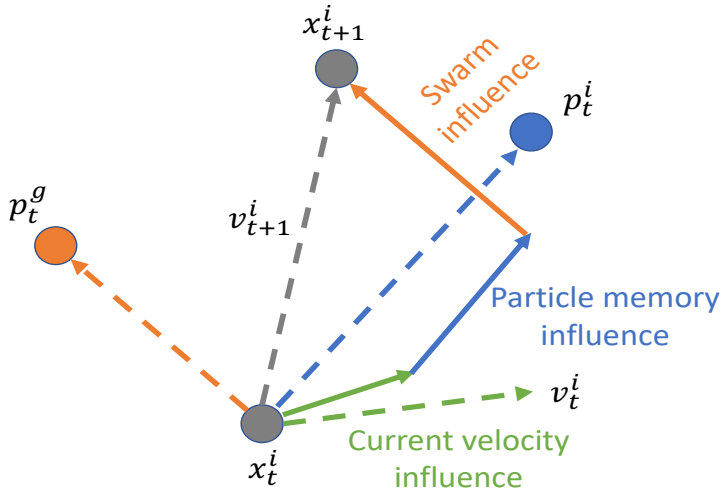



Figure 4.3. Position Update in PSO [155]

its own, as well as its velocity. Particles keep updating their states constantly until they reach the termination condition. As defined in prior work [155], a particle $i$ is defined by its position vector, $x_i$, and its velocity vector, $v_i$. Every iteration, each particle updates its position according to Equation 4.2 and Equation 4.3.

$$v^i_{t+1} = \omega v^i_t + c_1 r^t_1 (p^g_t - x^i_t) + c_2 r^t_2 (p^i_t - x^i_t) \tag{4.2}$$

$$x^i_{t+1} = x^i_t + v^i_{t+1} \tag{4.3}$$

$p^g_t$ and $p^i_t$ denote the best group position and the best particle position. The parameters $\omega$, $c_1$, $c_2$, $r_1$ and $r_2$ are inertia weight, two positive constants, and two random parameters within [0, 1], respectively. Speed-constrained Particle Swarm Optimization introduces a velocity constraint mechanism which restricts the value within the variable ranges and vanishes the erratic movements. SMPSO [110] is believed to deliver higher accuracy with less time over traditional PSO.

**Epsilon Constrained Method**. According to prior work [168], $\epsilon$-constrained Method selects and minimizes a primary objective by expressing the other objectives with inequality

constraints.

$$\underset{x}{Minimize} : F_p(x) \tag{4.4}$$

subject to $F_i(x) \leq \epsilon_i$, for $i = 1...k, i \neq p$. For example, in key-value data store, we may maximum the throughput by meeting the latency SLAs requirements. This approach is widely used to convert multi-objective optimization problem into single objective one.

**Weighted Sum Method**. According to prior work [102], the weighted sum method converts the multi-objective problem into a single-objective problem by constructing a weighted sum of all the objectives.

$$\underset{x}{Minimize} : F(x) = \sum_{i=1}^{k} (\omega_i * F_i(x)) \tag{4.5}$$

As Equation 4.5 shows, the single final target F(x) is the weighted sum of each individual target $F_i(x)$. The challenge of this approach is how to determine the weighting coefficients to each of the objectives. Besides, summing up two distinct objectives, such as throughput and tail latency, needs normalization to be semantically feasible.

The above summarizes the five methods briefly. More details can be found in their related papers [14, 39, 155, 168, 102].

### 4.2.3 Exploration and Exploitation

The *exploration* and *exploitation* tradeoff is well-known in the auto-tuning system to acquire new knowledge and maximize the uncertain payoffs. Exploitation means to probe a limited portion of the search space, expecting to improve the existing promising solution. This operation tries to leverage the vicinity of the current candidate to figure out a better solution. On the other hand, exploration means to probe a larger search space to avoid being trapped into a local optimum. The tradeoff between exploration and exploitation is among our interests in this chapter to illustrate the effectiveness and efficiency of the selected algorithms.

## 4.3 Methodology

In this section, we describe details of the hardware environments, workload characteristics, parameter spaces and our implementation of optimization algorithms.

Table 4.1. Experimental Machine Configurations

| **Machine** | Server ID | vCPU | Memory | Storage | Connection |
|---|---|---|---|---|---|
| **Baseline** | M1 | 4 | 8GB | Intel P3600 SSD | PCIe |
| **CPU-plus** | M2 | 32 | 8GB | Intel P3600 SSD | PCIe |
| **MEM-plus** | M3 | 32 | 16GB | Intel P3600 SSD | PCIe |
| **STOR-minus** | M4 | 32 | 16GB | Intel 530 SSD | SATA |

**Hardware.** In order to collect performance data with different hardware setups, we have defined four machine configurations as listed in Table 4.1. Based on their computing, memory, and storage capabilities, we categorize the four hardware setups as *Baseline*, *CPU-plus*, *MEM-plus* and *STOR-minus* to illustrate the effect of different computer components to the performance. The baseline machine (M1) is a Dell T620 Server with a 4-Core(TM) Intel Xeon E5-2690 2.9GHz CPU, 8-GB memory. We use an Intel P3600 PCIe SSD and an Intel 530 SATA SSD as storage devices in our experiments.

**Software and Workloads.** We use Ubuntu 14.04 with Linux Kernel 4.4.0, Ext4 file system and RocksDB 5.17.0 in our experiments. In order to cover a broad range of different workload patterns, we have enhanced RocksDB's default benchmarking tool, *db_bench*, to generate three workloads following typical key distributions. (1) *Zipfian*. A pattern of distribution where a small portion of items receives most of the requests, and most of items are requested rarely. (2) *Hotspot*. A majority (e.g., 80%) of its GET requests access a relatively small portion (e.g. 20%) of the entire data set. (3) *Random*. All the records in the database are access randomly with an equal probability. Among all the three distributions, Zipfian has the most skewed access pattern.

The working-set size has an obvious effect to the duration of experiments. Since we mainly focus on exploring a large parameter space within a practical time period, we choose to set the working set size to be 100GB. This large working set can guarantee that all experiments can be finished within a reasonably long time (six months in our case). We

have also repeated part of the experiments when the working set is 200GB to validate our findings. After warming up the system, we run the experiments for 180 seconds, which is long enough to collect stable evaluation results. Our experimental results show a wide range of performance numbers and are suitable for applying and evaluating auto-tuning algorithms.

**Parameter Space.** Compared with LevelDB [56], RocksDB has adopted several particularly optimized schemes to fully exploit the rich internal parallelism features of flash SSDs [27]. For example, multiple immutable Memtables are used to avoid write stalls. Flushes and compaction operations are multi-threaded with separated thread pools and execution priorities. Since RocksDB is optimized for parallel operations, system users are suggested to parallelize requests at the application level. Furthermore, RocksDB keeps multiple files in level 0 and triggers the compaction process when the number of level-0 files reaches a predefined threshold. RocksDB also keeps a read cache to accelerate READ operations.

Because of the aforesaid rich features, RocksDB maintains over 50 tunable parameters. Apparently we are unable to exhaustively study all combinations of the 50 parameters. Since our goal is to compare multi-objective optimization techniques, we desire to study a parameter space large enough to cover the important parameters. Based on the observations from prior work [99, 137], we select and focus on six most important parameters that have a significant impact to performance as discussed. Table 4.2 shows the parameters and the value range of each parameter.

Table 4.2. RocksDB Parameter Space

| Param. | Abbr. | Values | Notes |
|---|---|---|---|
| **Write Buffer(x32MB)** | WB | 1,2,4,8,16 | The number of write buffers. |
| **Concurrent Threads** | CT | 1,2,4,8 | Application-level requesting threads. |
| **Flush Writers** | FW | 1,2,4 | Background flush operations. |
| **Read Cache Size (GB)** | CS | 1,2,4 | The size of read cache |
| **Cleanup File Num** | CF | 1,2,4,8 | Compaction trigger threshold |
| **Concurrent Compactor** | CC | 1,2,4 | Number of compaction operations |

**Experiments and Implementations.** Our experimental studies have been per-

formed in two stages.

*Stage 1: Exhaustive experiments.* We first run experiments with an exhaustive combination of the six configurations as listed in Table 4.2, with three workloads on four machine setups. Over a period of 6 months, we have completed over 12,000 experimental runs. After that, we store all the data store configurations, workload and hardware information, and the benchmarking results in an MySQL [142] database for the emulation in Stage 2.

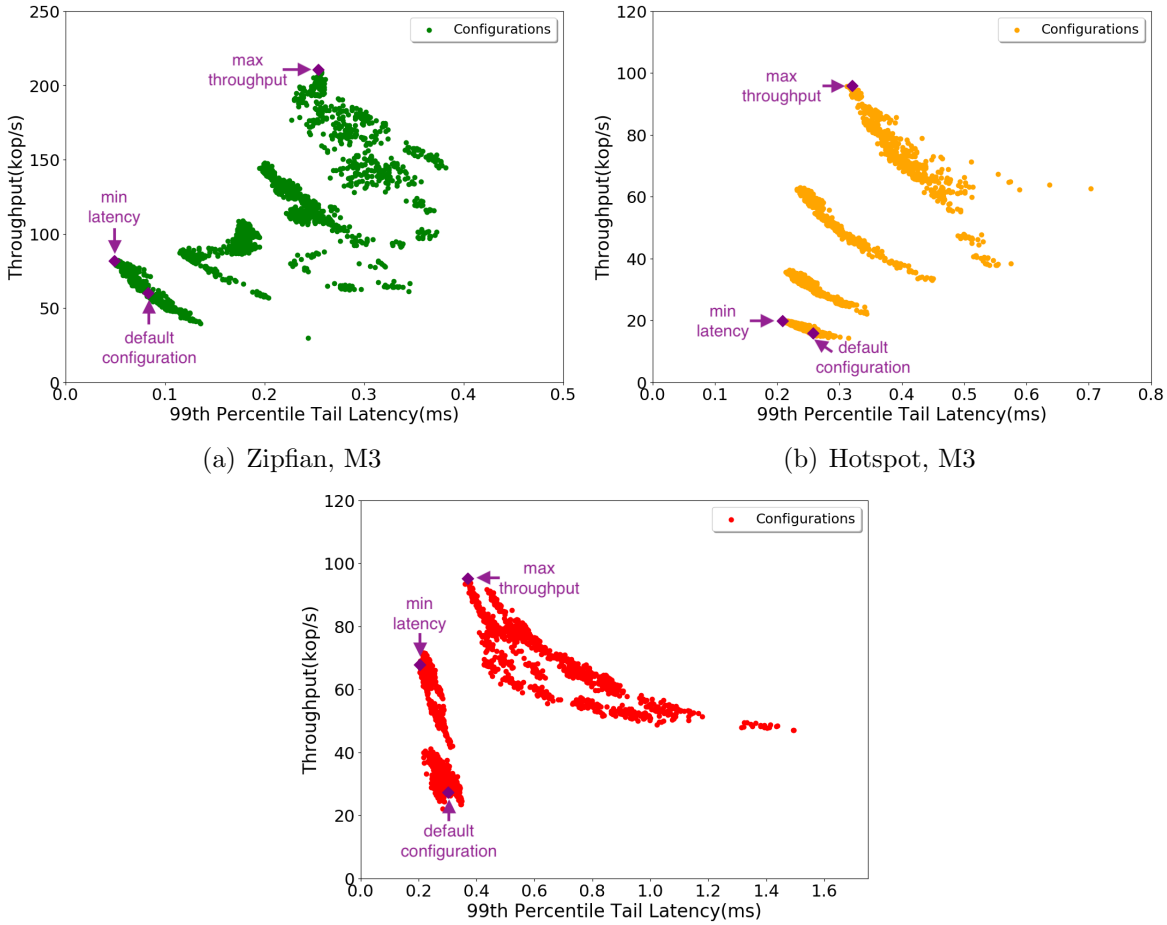
*Stage 2: Auto-tuning Emulation.* W emulate the process of auto-tuning key-value storage systems by running the MOO optimization method and querying MySQL for the evaluation results. In this way, we can avoid running the experiments against the RocksDB each time. Since this chapter focuses on multiple objective optimization approaches, we choose to optimize both *throughput* and *latency* (the 99th percentile tail latency) simultaneously in all our experiments. We believe that the same methodology can be used when more objectives, such as power consumption, are added.

We have implemented a client to make use of Platypus [145], which is an open source Python library for multi-objective optimization, for all of our experiments. We further convert the parameters in RocksDB into the algorithm-related ones. For example, we define write buffer number as *gene*, and each configuration as a *chromosome*. A sufficient number of configurations are measured as the evolution process continues.

The above-said experiment process provides two benefits. First, we only need to complete an experiment for each configuration once. In the algorithm evaluation (Stage 2), we can simply run the MOO methods and query the MySQL database to collect the corresponding data without actually running the experiments. This significantly saves the experimental time and allows us to repeat this evaluation process quickly. Second, since we have already completed all the experiments exhaustively, we can know the global optimal configurations, which allows us to quantitatively measure how close each MOO method can reach to the global optimum.

## 4.4 Experimental Results

### 4.4.1 Motivations




(a) Zipfian, M3

(b) Hotspot, M3

(c) Zipfian, M1

Figure 4.4. Motivations of Parameter Tuning

In this section, we first demonstrate the parameter space and its effect on the two key performance metrics, throughput and latency. Due to space constraint, we select three scenarios (Zipfian and Hotspot workloads running on two hardware setups, M1 and M3) for illustration, as shown in Figure 4.4(a), 4.4(b), and 4.4(c). Each point represents one RocksDB configuration. In the figures, we also mark the configurations that deliver the maximum throughput, the minimum latency, and the default configuration. We can obtain several important clues from the figures. (1) The RocksDB configurations have a significant performance impact. For example, as Figure 4.4(a) shows, the achievable

maximum throughput is 211 kop/s, while the tail latency is around 254 $\mu$s; the lowest achievable tail latency is 49 $\mu$s, while the throughput is only 82 kop/s. There is a clearly a tradeoff between the two. (2) The configurations are clustered in several groups, rather than uniformly distributed. This means that the parameters do not an equal effect on the performance, and some parameters could have a dominant effect on the performance. (3) The default configuration cannot achieve the optimum, in all three cases. Also, we can find in the figures that the default setting tends to optimize for tail latencies rather than high throughput. As shown in Figure 4.4(a), the default setting achieves an average tail latency of about 83 $\mu$s, while the throughput is only about 60 kop/s. (4) The effect of different RocksDB configuration varies significantly across different hardware and workloads, which is clearly illustrated in the figures by the distinct shapes of the clouds of configuration points.

These figures clearly show that although we may not find optimal solutions for both goals simultaneously, there is enough tradeoff and optimization space with specific users' preferences. In the following section, we will discuss the performance of the five MOO methods.

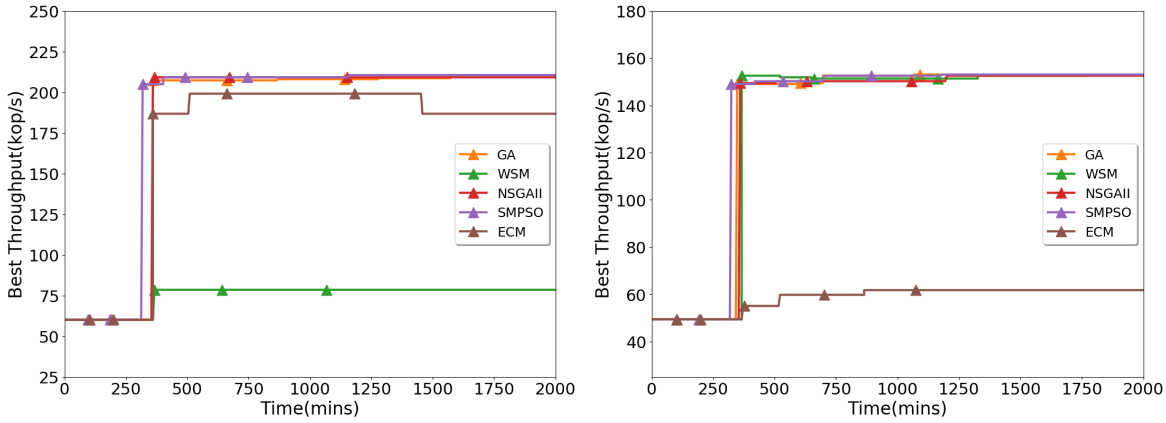
### 4.4.2 Comparative Analysis

In this section, we compare the five MOO methods whose configurations are shown in Table 4.3. We use term *optimal* and *near-optimal* to represent configurations that provide 100% and 95% of the maximum achievable throughput respectively and restrict the 99th percentile tail latency within 250$\mu$s as the QoS requirement in our experiments.

Table 4.3. Auto-tuning Algorithm Configurations

| **Algorithms** | Optimizations Targets | QoS Requirements |
|---|---|---|
| **GA** | Throughput | No consideration |
| **NSGA-II** | Throughput and latency are treated equally | No consideration |
| **SMPSO** | Throughput and latency are treated equally | No consideration |
| **ECM** | Throughput | 250$\mu$s |
| **WSM** | As shown in Equation 4.6 | A combined target |

As shown in Figure 4.5(a) and Figure 4.5(b), given enough time, all the five algorithms




(a) Throughput vs. Time on PCIe SSD (b) Throughput vs. Time on SATA SSD

Figure 4.5. Throughput on Multiple SSDs

can converge to relative stable performance. Specifically, as Figure 4.5(a) shows, on PCIe SSD, GA, NSGA-II, SMPSO and ECM can achieve comparable peak throughout, which is about 205 kop/s. Accordingly, the corresponding 99th percentile tail latency is about 250 $\mu$s. WSM shows a totally different behaviors. We have combined the throughput and the 99th percentile tail latency into one single target with a weighted sum method. The single final target is defined as

$$T = \frac{throughput}{curr\_max\_throughput} - \frac{latency}{curr\_max\_latency} \tag{4.6}$$

We use $curr\_max\_throughput$ and $curr\_max\_latency$ to denote the maximum throughput and latency respectively until the current step. Since the normalized throughput (first component in Equation 4.6) and latency (second component in Equation 4.6) make positive and negative contributions to the combined final target respectively, we try to increase the throughput and decrease the latency by maximizing the combined goal. We can see that the maximum throughput achieved by WSM is about 78 kop/s and the corresponding 99th percentile tail latency is around 64 $\mu$s. Thus, WSM achieves a relatively lower 99th percentile tail latency but also a lower throughput. Compared with the AutoTuning process on PCIe SSD, the behaviors of these algorithms on SATA SSD show different

trends. As Figure 4.5(b) shows, GA, WSM, NSGA-II and SMPSO show similar behaviors in finding the (near-)optimal configuration. They achieve maximum throughput which is about 150 kop/s and the related 99th percentile tail latency is around 542 $\mu$s. However, the maximum throughout achieved by ECM is only about 62 kop/s and the corresponding 99th percentile tail latency is about 240 $\mu$s. The reason is that in order to meet the QoS requirements (e.g., 250$\mu$s in our experiments), some parameters in RocksDB, such as concurrent thread number, has to be set significantly smaller than other algorithms. Thus, the maximum throughput achieved by ECM is remarkably reduced than other approaches. The performance differences of ECM as shown in Figure 4.5(a) and Figure 4.5(b) is because faster PCIe SSD provides a much larger search space than SATA SSD while constraining the 99th percentile tail latency within 250$\mu$s. As a consequence, ECM achieves higher throughput on PCIe SSD by comparing more possible genes(parameters) and chromosomes(configurations). It means that based on the quality of services requirements provided by the users, we should choose the proper AutoTuning approach to figure out the best configuration.

| Finding#1: Most of the algorithms can find the near-optimal configuration if given enough time. |
|---|

We have noticed that WSM shows total different behaviors on PCIe and SATA device as Figure 4.5(a) and Figure 4.5(b) show. To have a more straightforward explanation, we have illustrated the effect of the storage device to performance in Figure 4.6. Each dot in Figure 4.6 represents one configuration and the data with different devices are marked in different colors. We can see in Figure 4.6 that the tail latency range on SATA SSD(78$\mu$s-1.6ms) is significantly larger than that on PCIe SSD(49$\mu$s-382$\mu$s), while the throughput range is smaller but relatively close. As a consequence, the weight of latency component in the combined target, as Equation 4.6 shows, becomes significantly smaller on SATA SSD. Thus, WSM tends to tune the system to achieve near-maximum throughput on SATA SSD and to probe the system to produce moderate throughput on PCIe SSD.

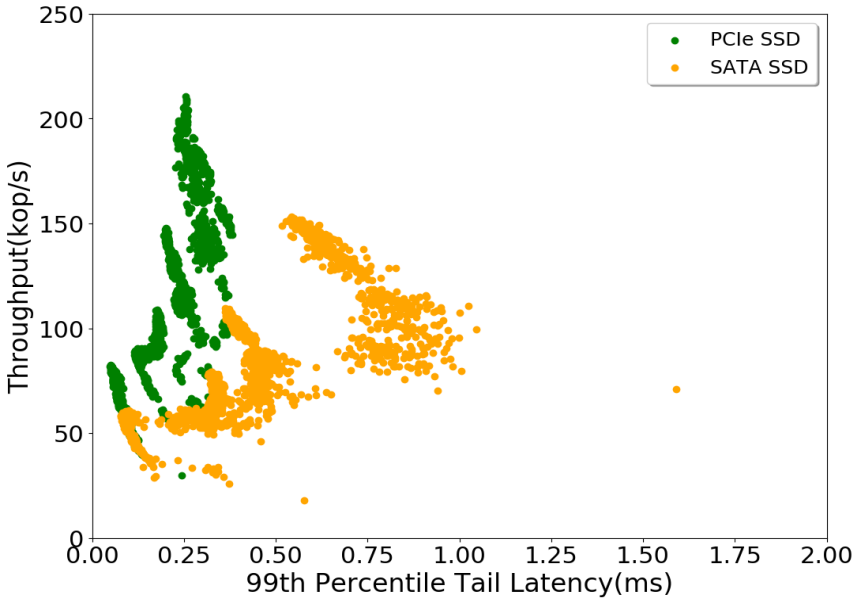



Figure 4.6. Performance on PCIe and SATA SSD

| Finding#2: ECM tends to achieve a relatively lower throughput than other algorithms on a slower storage hardware. On the contrary, Our defined WSM produces moderate throughput on a faster storage hardware. |
|---|

| Finding#3: ECM and WSM behave differently on PCIe and SATA SSDs, but the reasons are not distinct. ECM has to meet the QoS requirements while WSM optimizes the combined final target. |
|---|

To have a better understanding on the auto-tuning procedure, Figure 4.7 shows the intermediate and final tuning results of ECM on machine M3. We can see that the algorithm has gone through 23 intermediate configurations before finding the near-optimal configuration. By constraining the 99th percentile tail latency within 250 $\mu$s, the maximum throughput ECM achieves is about 199 kop/s. Compared with the peak throughput (about 210 kop/s) we have observed, ECM can achieve 95% of the maximum throughput while meeting users' latency requirements. Note that we have also observed experiments that produce no solutions when the QoS requirements are set extremely low (e.g. 20$\mu$s).

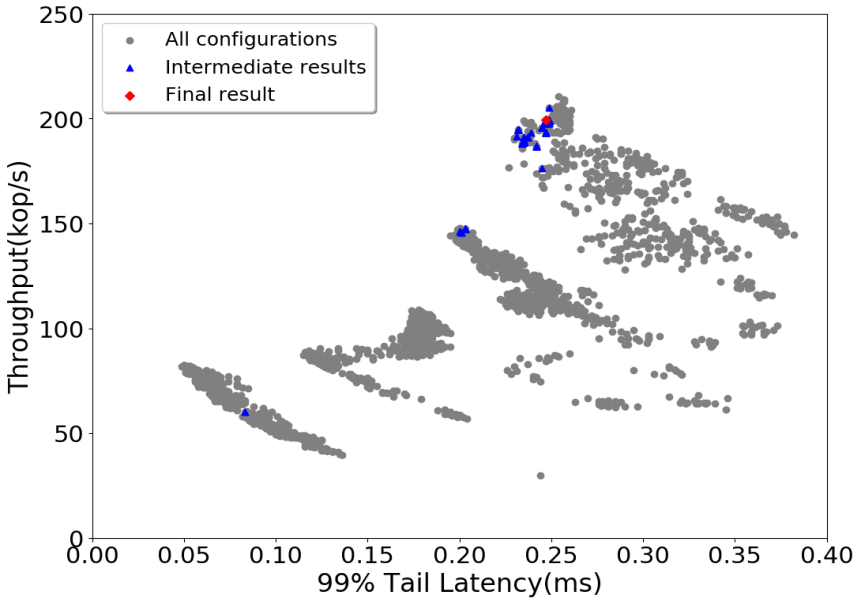
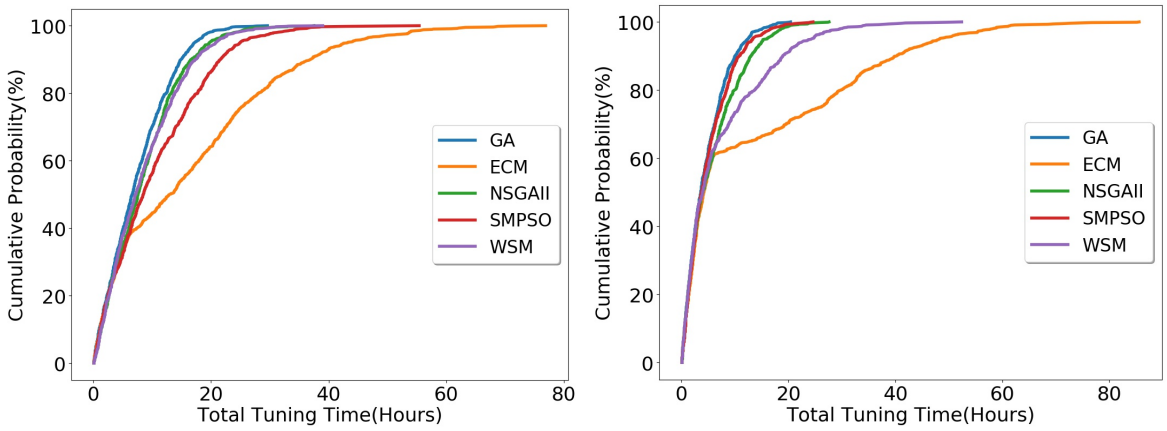



Figure 4.7. Intermediate and Final Results

Finding#4: ECM improves its solutions based on its intermediate results, but may produce no solutions when the users' QoS requirements are not properly set.




(a) CDF of Tuning Time on PCIe SSD

(b) CDF of Tuning Time on SATA SSD

Figure 4.8. AutoTuning time on multiple SSDs

We have also compared the efficacy of multi-objective auto-tuning methods in finding the near-optimal configurations on PCIe (see Figure 4.8(a)) and SATA (see Figure 4.8(b)) SSD. The Y axis shows the percentage of total runs (1,000) that can find near-optimal configurations within certain time (X axis). Apparently, the faster the better. Figure 4.8(a) shows that ECM takes longer than the rest algorithms to find the near-optimal configuration

on PCIe SSD. Specifically, 90% of the experiments take 37 hours for ECM to achieve near-optimal performance, compared to less than 21 hours for GA, NSGA, WSM and SMPSO. Similarly, on SATA SSD, as Figure 4.8(b) shows, ECM remains to be the most time-consuming approach among all the algorithms. 90% of the experiments take about 40 hours to find the near-optimal configuration while the other algorithms take less than 20 hours.

**Algorithm 3** Epsilon Constrained Method

```
1: population=generate(population_size)
2: procedure ϵConstrainedMethod
3:     i=0, nfe=10000
4:     while i ≤ nfe do
5:         parents=select(population)
6:         offspring=evolve(parents)
7:         evaluate(offspring)
8:         population=pick_qos(offspring)
9:         i=i+1
10:    end while
11: end procedure
```

Based on our observations, ECM is the most time consuming approach on both PCIe and SATA SSDs. As shown in Algorithm 3, the reason is that the population in ECM is updated with only the configurations that meet the QoS requirements. In our experiments, only offsprings that constraint the latency within 250$\mu$s are picked for the next-generation population. As a consequence, the gene diversity is reduced because of more strict selection conditions. Since gene diversity is one of the key factors that determine auto-tuning time, the total time taken by ECM is always the longest among all the algorithms. This unique operation in ECM, as a consequence, prolongs the tuning duration.

| Finding#5: ECM is the most time consuming algorithm among the selected five, although it meets QoS requirements to the best. |
|---|

### 4.4.3 Instantaneous Performance

Besides comparing the performance of near-optimal configurations, another aspect to consider is the instantaneous performance during the auto-tuning process. To find a good configuration, a reasonably large exploration space is necessary, but bad configurations (a configuration that takes a long time to complete) should be avoided as much as possible. A better algorithm is able to spend less time on bad configurations.

Figure 4.9 shows the instantaneous throughput (Y-axis) over time (X-axis) for one run for each method on PCIe SSD on machine M3. We can see that GA and SMPSO are the best two methods in terms of instantaneous throughput. They occasionally pick a worse configuration than the current one during the auto-tuning process. However, they both have the ability to discard these unpromising configuration and evolve based on the satisfactory ones. Specifically, only 16% and 26% of the overall auto-tuning time achieve throughput that is below 100 kop/s for GA and SMPSO respectively. In contrast, the throughput of ECM and NSGA-II drops frequently, because ECM and NSGA-II have tried noticeably more "bad" configurations. In specific, 45% and 32% of the auto-tuning time provide throughput less than 100 kop/s for ECM and NSGA-II, respectively. GA works by assigning the probability of surviving to the next generation based on the fitness value (i.e., throughput). Configurations with lower throughput values have a lower chance to be picked as parents, thus their genes (parameter values) have a lower chance to survive in the next generation. SMPSO can recognize and discard the unpromising configuration because it chooses new configurations by considering the previous one, groups best and particle best solutions (see Section 4.2). The combination of these three components help SMPSO keep focusing on the promising configurations.

The throughput degradation of ECM and NSGA-II is because they are typical multiple objective optimization methods, and they consider both throughput and latency simultaneously when selecting the next generation genes. All Pareto optimal configurations are treated equally good and selected. The difference between ECM and NSGA-II is that ECM

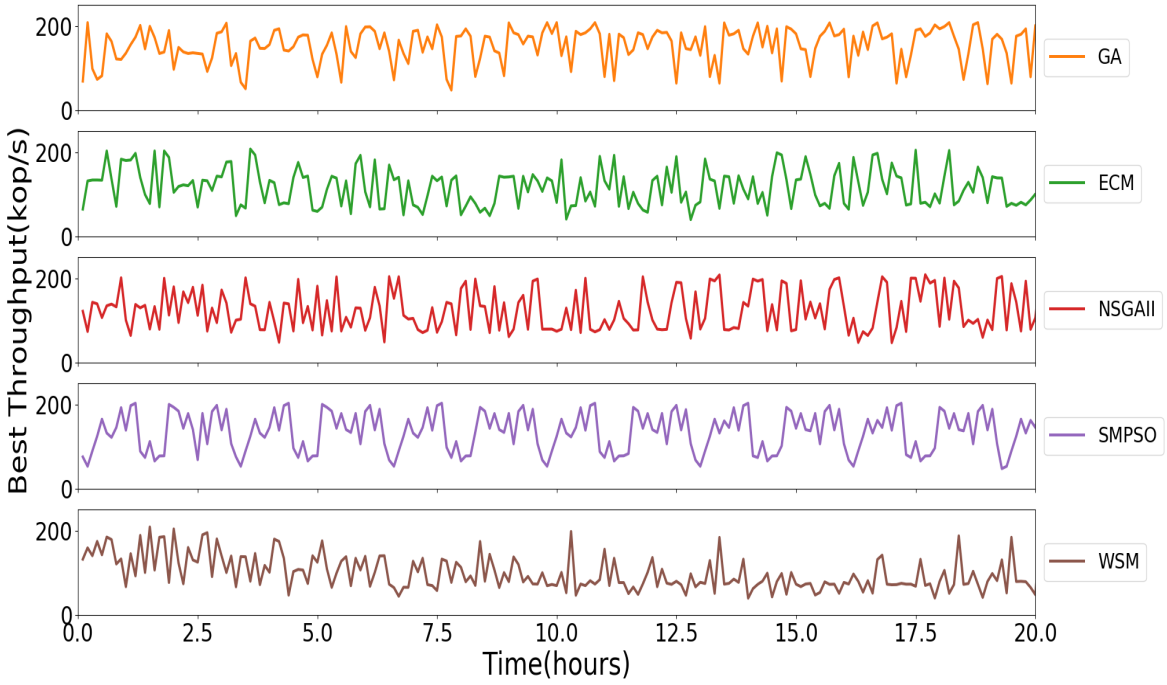


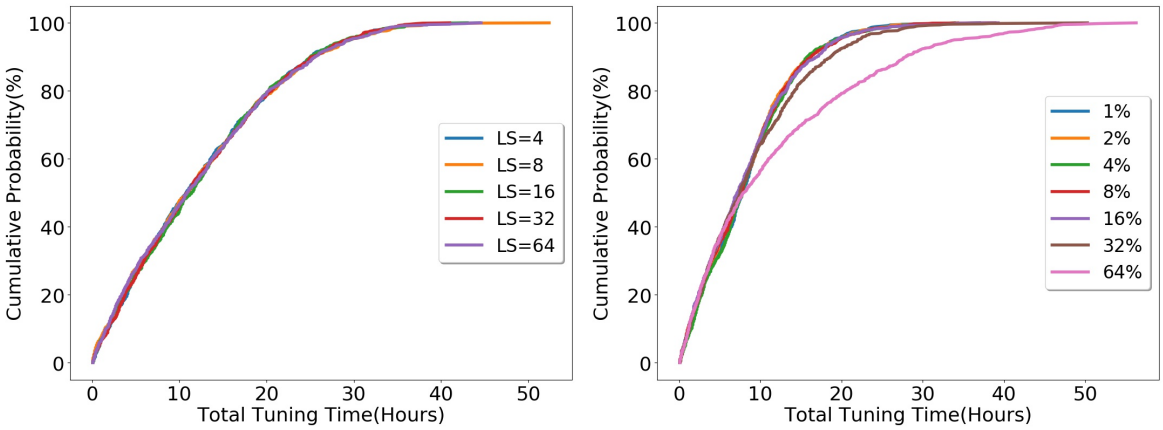

Figure 4.9. Instantaneous throughput

also considers the latency constraints, and only the configurations that can meet the latency requirements will be selected. As a consequence, they do not always try to maximize one single target, such as throughput in Figure 4.9. Different from all the other four algorithms, WSM shows a totally different behaviors. WSM achieves significantly lower instantaneous throughput than other approaches. Specifically, 77% of the tested configurations provide throughput less than 100 kop/s. It means that WSM gives configurations that provide moderate throughput more opportunities to survive in next generation. That's because in WSM, the optimization targets have been combined into a single target as defined in Equation 4.6. WSM tries to balance multiple optimization goals, including throughput and tail latency, instead a single target.

| Finding#6: Some optimization goals of multi-objective optimization methods may be compromised for both instantaneous and final performance. |
|---|

## 4.5 Impact of Hyper-Parameters

In the multiple-objective optimization methods, the hyper-parameters, i.e., the optimization algorithms' own parameters, play an important role in determining the total auto-tuning time. In this section, we discuss the impact of hyper-parameters from the perspective of exploitation and exploration(see Section 4.2.3).




(a) Impact of Leader Size

(b) Impact of Mutation Rate

Figure 4.10. Impact of Hyper-Parameters

### 4.5.1 Effect of Exploitation: Leader Size in SMPSO

In SMPSO, *leader size* means the number of the best particles at current stage, which determines the selection pressure for the next movement. Typically, a higher selection pressure, caused by a smaller leader size, pushes the search toward exploitation and expects a shorter tuning time. However, according to Figure 4.10(a), leader size does not have a noticeable influence on the overall tuning time. We have conducted the experiments by changing the leader size from 4 to 64 and the total tuning time remains almost the same. For example, 90% of the experiments can reach the near-optimal configuration within 25.6 hours. This is because some alleles (parameter values) play a dominant role in determining the performance. These alleles will be selected even though only fours leaders are selected during the selection process. These surviving alleles further produce offsprings which achieve good performance. We have a detailed discussion about alleles in Section 4.6.

### 4.5.2 Effect of Exploration: Mutation Rate in NSGA

Figure 4.10(b) shows that mutation rate in genetic algorithm plays a significant role for determining the total tuning time. For example, when mutation rate is 1%, the total time consumed to find the (near-)optimal configuration is 15.9 hours for 90% of the experiments. As we increase the mutation rate to 64%, the tuning time increases to 28 hours for 90% of the runs. Typically, a mutation operator modifies genes in each individual in a random manner with a given probably, thus the increasing structural diversity of the population pushes the search toward exploration. However, a high mutation rate (64%) causes the "good" gene disappear easily in the next generation, while a relative low mutation rate achieves a better balance for gene diversity and stability.

> Finding#7: Since only a few alleles play dominant role in determining the system performance, more strict exploitation improves performance slightly, while larger exploration space causes performance degradation when auto-tuning RocksDB.

## 4.6 Insight and System Implications

We have studied the auto-tuning behaviors of popular multiple objective optimization methods for RocksDB on SATA and PCIe SSDs. Despite successful deployment of these algorithms on auto-tuning complex key-value systems, little is known how and why some approaches work better than others for certain target. In this section, we attempt to open the "black-box" algorithms and gain insight into their internals based on the behaviors of these algorithms and our knowledge about the key-value systems. We further present several important system implications to effectively optimize RocksDB based on their Quality of Services requirements.

**Alleles.** It is expected that as the auto-tuning process moves forward, there will be some alleles (parameter values) dominant in the population (configurations). We present the alleles of genetic algorithm in Figure 4.11 as an example to demonstrate the evolution of each system parameter. The Y-axis shows 6 genes (parameters) as listed in Table 4.2, while each row represents one allele (parameter value). The X-axis shows the evolution

over the first 30 generations. Each cell is colored based on the number of alleles in each generation. More frequent alleles are colored with darker color.

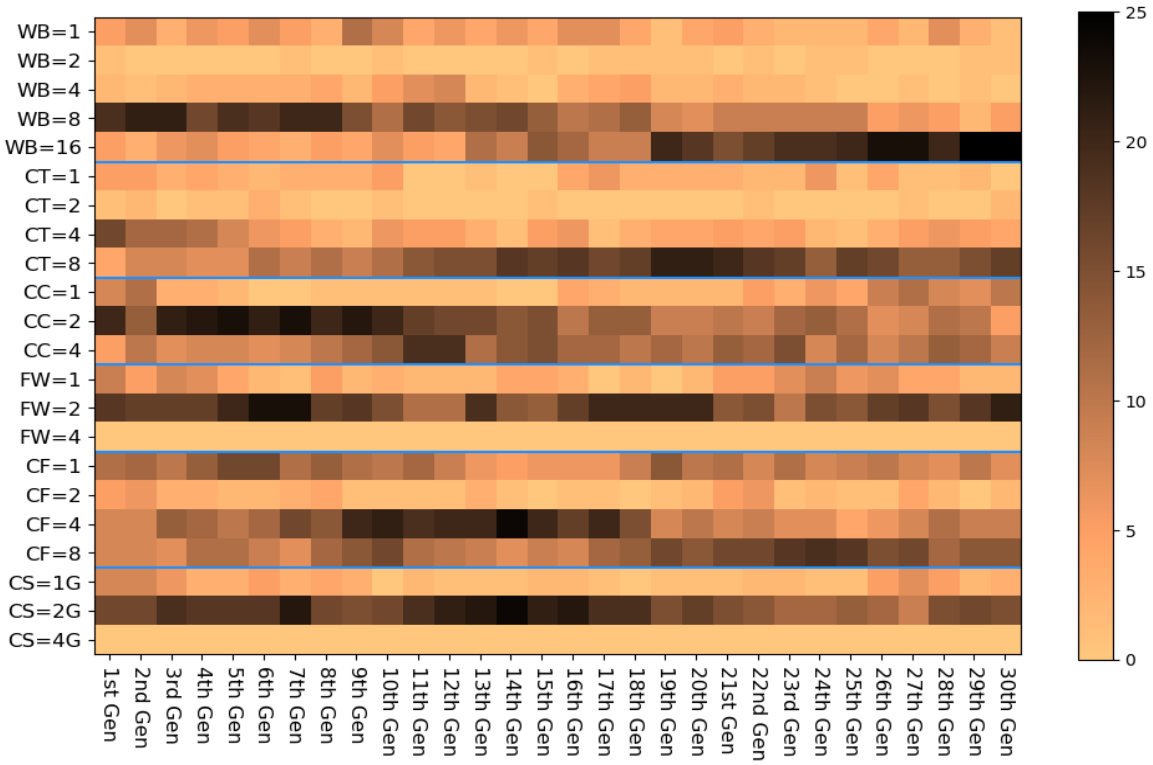



Figure 4.11. Alleles of Genetic Algorithm

In the first generation, the number of write buffers being 8 and concurrent thread number being 4 are dominant. However, as the evolution process continues, write buffer number being 16 and concurrent thread number being 8 becomes more dominant than other alleles. Since GA simulates the natural selection process where alleles with better fitness are more likely to survive, this indicates that GA prefers more write buffers and more concurrent threads when optimizing the throughput. For some other parameters, such as concurrent compactors and cleanup file number, their alleles become more diverse as the evolution continues, which means that they do not have significant impact to the overall throughput, compared to write buffer size and concurrent thread number.

Finding#8: As auto-tuning continues, the most dominant alleles become clear for most parameters, such as write buffer(WB), concurrent threads(CT) and flush writers, etc.

**Importance of Parameters.** Based on the prior observations, an interesting question is what is the impact of each parameter to the overall performance of RocksDB. To answer this question, we have quantitatively studied the correlations between each parameter and two performance metrics (throughput and the 99th percentile tail latency).

As the parameters we have studied in this work are all discrete numbers, whereas the throughput is continuous, we have taken a widely used approach to calculate the correlation between discrete and continuous values [22]. We illustrate with the Concurrent Threads(CT) as an example. We set concurrent threads with 4 values (1,2,4,8) in our experiments. We convert this parameters into 4 binary variables: $x_1$, $x_2$, $x_3$ and $x_4$. If the thread number is set to be 1, we assign $x_1 = 1$ and $x_2$, $x_3$ and $x_4$ are set to be 0. Let Y represent the corresponding throughput values. We then do a linear regression with ordinary least squares(OLS) on Y and $x_1$, $x_2$, $x_3$ and $x_4$. $R^2$ is a commonly used metric to measure how data fits a regression line [158, 20]. In our approach, $R^2$ measures the the correlations between the selected parameter and the received performance(throughput and tail latency). Typically, $R^2 > 0.6$ is a indicator that the parameter has significant impact on the performance. Parameters with highest $R^2$ are colored green in Table 4.4. To find the second important parameter, the same process is applied to the remaining parameters, but with the first important parameter being fixed. For example, we calculate $R_2$ respectively by setting CT to be 1, 2, 4 and 8. We take the highest value as the $R^2$ value for this parameter. We color the second important parameter with the blue color.

As Table 4.4 shows, we have conducted extensive experiments with multiple hardware configurations and workloads, and the correlated parameters remain the same across different platforms. The number of concurrent threads and write buffers play the most important role in determining throughput and tail latency. However, we also notice the number of

Table 4.4. Importance of Parameters (measured by $R^2$)

| Metric | SSDs | vCPU | MEM | WL | WB | CT | CC | FW | CF | CS |
|---|---|---|---|---|---|---|---|---|---|---|
| **Throughput** | PCIe | 4 | 8 GB | Zipfian | 0.91 | 0.79 | - | - | - | - |
| **Throughput** | PCIe | 32 | 16 GB | Zipfian | 0.91 | 0.74 | - | - | - | - |
| **Throughput** | PCIe | 32 | 8 GB | Zipfian | 0.91 | 0.87 | - | - | - | - |
| **Throughput** | PCIe | 32 | 16 GB | Hotspot | 0.92 | 0.89 | - | - | - | - |
| **Throughput** | SATA | 32 | 16 GB | Zipfian | 0.93 | 0.68 | - | - | - | - |
| **Tail Latency** | PCIe | 4 | 8 GB | Zipfian | 0.92 | 0.65 | - | - | - | - |
| **Tail Latency** | PCIe | 32 | 16 GB | Zipfian | 0.66 | 0.89 | - | - | - | - |
| **Tail Latency** | PCIe | 32 | 8 GB | Zipfian | 0.81 | 0.82 | - | - | - | - |
| **Tail Latency** | PCIe | 32 | 16 GB | Hotspot | 0.84 | 0.70 | - | - | - | - |
| **Tail Latency** | SATA | 32 | 16 GB | Zipfian | 0.54 | 0.85 | - | - | - | - |

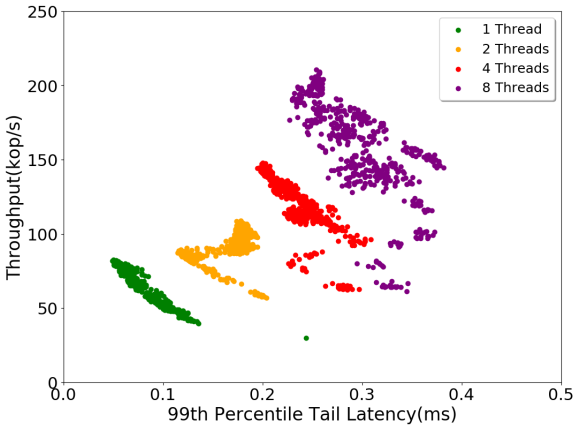
concurrent threads affects the tail latency more significantly than the throughput, while the number of write buffers has a stronger influence on the throughput than on the tail latency.

> Finding#9: Write buffer and concurrent thread number play dominant roles in determining system's performance across hardware and workloads.

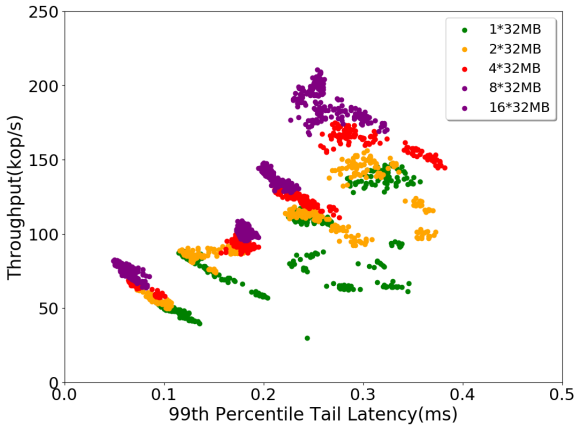
**Implications to RocksDB Design.** To further look into RocksDB's system design and verify our findings, we have plotted the 99th percentile tail latency and throughput to illustrate the effect of concurrent threads, write buffers and compaction threads on machine M3 in Figure 4.12(a) to Figure 4.12(c). Each dot on the figures represents one configuration.

We can see in Figure 4.12(a) that configurations with more threads tend to produce a higher maximum throughput at a cost of a higher tail latency. However, the performance ranges have intersections when thread number differs. Furthermore, we find that the corresponding set of dots (purple) for 8 threads has a higher variance than the other sets. That means when more threads are used, other parameters tend to play an increasingly more important role in determining the performance.

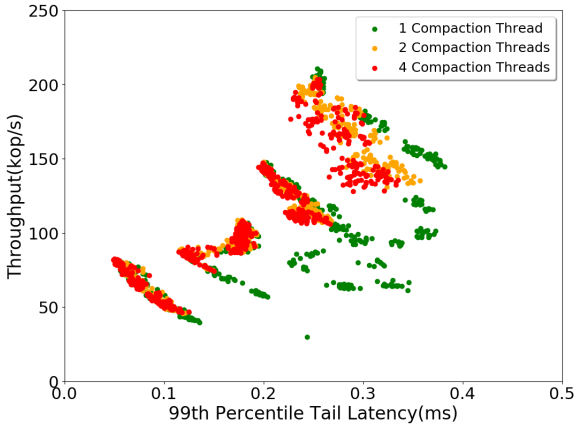
Figure 4.12(b) shows the performance with different numbers of write buffers. We can see that more write buffers provide relatively higher throughput, while not necessarily increasing the tail latency. Also, the changes of throughput and tail latency when changing




(a) Effect of Concurrent Threads




(b) Effect of Write Buffer




(c) Effect of Compaction Threads

Figure 4.12. Effect of different parameters in RocksDB

write buffer number is not as large as the performance changes with different concurrent threads.

Finally, Figure 4.12(c) shows the performance change when changing the number of background compaction operations. We can observe an obvious throughput improvement when increasing compaction operation number from 1 to 2. However, the throughput does not have a obvious change when the parameter value further increases from 2 to 4. That is because compaction is an expensive background operation in RocksDB and the improvement of the background operations becomes negligible for the foreground processes. That explains why this parameter is not chosen as the most influential ones in Table 4.4. Our observations about the performance trend in this Section (Sec 4.6) are consistent with

our analysis about the multi-objective algorithms in Section 4.6 and Section 4.6. These findings are helpful for system designer to optimize and tune the system.

> Finding#10: Increasing the number of write buffer and setting a proper user-level parallelism degree will be top choices for system designers and practitioners to balance multiple objectives.

> Finding#11: Other parameters, such as the number of flush and compaction processes, have noticeable impact to the performance. But the tuning space can be restricted to a reasonably small number (e.g. 4 in our experiments).

### 4.7 Limitations and Future Work

In this chapter, we have presented the first comprehensive study on the multi-objective optimization algorithms on RocksDB. However, since auto-tuning the increasingly complicated key-value system is a challenging task, there are several work that remain to be worth exploring in the future. **(1) Optimization targets.** In this work, we mainly focus on the trade-off between throughput and tail latency, which is a common consideration in real world deployment. However, other metrics, such as power consumption and capital cost, are also of practical interests. We plan to expand the work to study the balance between performance and power consumption in the future work. **(2) Workloads and Parameter Space.** In this chapter, we collect the experimental data with synthesized workloads following representative zipfian, hotspot and random distributions. We plan to further explore the possibility of repeating real enterprise workloads on our platform. What's more, we currently focus on 6 most influential parameters for performance. When the target is power consumption, the parameter selection may need to change. We plan to extend the parameter space consideration, according to the optimization target and our evaluation results in the future. **(3) Algorithm Improvement.** In this chapter, we have discussed the impact of hyper-parameters in the algorithms. Based on our findings, we plan to improve the traditional multi-objective optimization algorithms to make them more robust

for different workloads, hardware, and optimization targets, by integrating them with new techniques, such as penalty function or re-initialization, etc.

## 4.8 Related Work

Auto-tuning the computer systems has been extensively studied. Prior work can be roughly divided into two categories: Single and multiple objective auto-tuning.

**Single Objective Auto-tuning**. In recent years, auto-tuning technologies have been widely studied to maximize one single objective, such as throughput, in complex computer systems. For example, Behzad *et al.* [15] propose to apply GA to HDF5 applications to improve I/O performance. More recently, Li *et al.* [89] aim to optimize Lustre with neural network-based deep reinforcement learning. Aken *et al.* [9] use supervised and unsupervised machine learning methods to identify (near-)optimal configurations for database management systems. Alipourfard *et al.* [10] try to figure out the best configuration for big data analytics in cloud. Besides, GA is also used in other purposes, such as storage system provisioning [132] and recovery [77]. Rafiki [99] tires to tune the parameters of NoSQL database, such as Cassandra and ScyllaDB, for HPC and dynamic metagenomics workloads. Cao *et al* [20] compare multiple black-box single-objective auto-tuning approaches for storage systems. However, all these prior work try to optimize one single goal, such as throughput, in the computer systems. Our work is to present the first study of multi-objective auto-tuning approaches on RocksDB on different hardware.

**Multiple Objective Auto-tuning.** Multiple objective optimization(MOO) approaches are widely studied in the field of engineering [69, 38, 126, 149, 112, 78, 19]. Prior studies also have applied multi-objective auto-tuning techniques on computer systems [46, 58, 74, 80]. Durillo *et al.* have discussed the advantages and drawbacks of existing single-objective and multiple-objective auto-tuning algorithms [46]. Gschwandtner *et al.* have applied multi-objective auto-tuning to parallel applications to optimize execution time, energy and resource usage simultaneously [58]. Jordan *et al.* introduce framework to auto-tune compiler and run-time components to optimize run-time and efficiency [74]. Kofler *et al.* try to

enhance traditional multi-objective algorithm with region division [80]. In this chapter, we mainly focus on auto-tuning RocksDB on multiple machines and gain insight and system implications for future system designers and practitioners.

## 4.9 Conclusion

Tuning key-value store system configurations can bring significant performance improvement, such as higher throughput or lower tail latency. Increasingly complex key-value systems make auto-tuning become necessary and challenging. Instead of focusing on one single optimization target in traditional approaches, in this chapter, we have conducted the first comprehensive evaluation to understand the behaviors of five multi-objective optimization algorithms for auto-tuning RocksDB on multiple SSD devices. We have further discussed the instantaneous performance and the impact of hyper-parameters of the auto-tuning approaches from the perspective of exploitation and exploration. Based on our observations, we have also presented the associated system implications for system designers and practitioners in future optimizations. We believe the methodology used in this work can also be applied to other optimization targets and other systems.

# CHAPTER 5
# FROM FLASH TO 3D XPOINT: PERFORMANCE BOTTLENECKS AND POTENTIALS IN ROCKSDB WITH STORAGE EVOLUTION

In this chapter , we present the first, in-depth performance study on the impact of the rapid storage hardware evolution to RocksDB to reveal several unexpected bottlenecks in the current key-value store design, which hinder us from fully exploiting the great performance potential of the new storage hardware. Our study shows that many of the current LSM-tree based key-value store designs need to be carefully revisited to effectively incorporate the new generation of hardware for realizing high-speed data processing.

## 5.1 Introduction

With the rapid growth of Internet services, data center systems need to handle a huge volume of data at a very high processing rate [120]. This grand trend demands a non-stopping, fast-pace evolution of the storage subsystems, incorporating cutting-edge hardware and software technologies and optimizing the entire system in a cohesive manner.

On the hardware side, NAND flash based SSDs have already been widely adopted in today's data centers [123, 49]. Although compared to conventional disk drives, flash SSDs can deliver higher performance and better power efficiency, the well-known slow random write and limited lifetime issues still remain a non-negligible concern in large-scale systems.

More recently, Intel's Optane SSD [66], which is built on 3D XPoint [59], a type of Non-volatile Memory (NVM) technology, has received increasingly high interests in the industry. Unlike flash SSD, 3D XPoint SSD uses resistance-based recording material to store bits, enabling it to provide much lower latency and higher throughput. Most importantly, 3D XPoint SSD significantly alleviates many long-existing concerns on flash SSDs, such as the read-write speed disparity, slow random write, and endurance problems [107, 170]. Thus 3D XPoint SSD is widely regarded as a pivotal technology for building the next-generation storage system in the future.

On the software side, in order to accelerate I/O-intensive services in today's data

centers, RocksDB [138], which is the state-of-art Log Structured Merge tree (LSM-tree) based key-value store, has been widely used in various major data storage and processing systems, such as graph databases [33, 136, 34], stream processing engine [13], event tracking systems [144] and object data store [23, 24]. They all rely on RocksDB as the storage engine to provide high-speed queries for key-value workloads.

However, since RocksDB is particularly optimized for flash-based SSDs, new challenges would naturally emerge as we transit to 3D XPoint SSD in the future. Considering the architectural differences between flash SSD and 3D XPoint SSD, it is a highly interesting and practical question —*Is the current design of LSM-tree based key-value store readily applicable to the new 3D XPoint based SSD?*

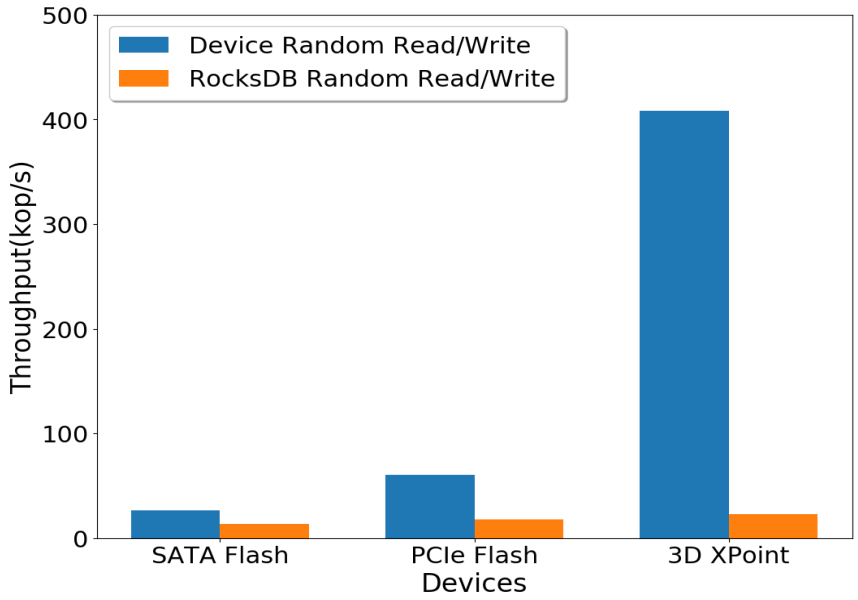



Figure 5.1. A motivating example—performance improvement of RocksDB on Intel Optane SSD

To have a glimpse of the potential problem, we show a motivating example in Figure 5.1. We use Intel Open Storage Toolkit [106] to generate 4KB random requests with 8 threads and read/write ratio being 1:1 to access the first 10GB storage space on a 280GB Intel Optane 900P SSD. The raw I/O throughput increases from 26 kop/s on an Intel 530 SATA SSD to 408 kop/s on the 3D XPoint SSD, which is a speedup of about 15.7 times. Then we benchmark RocksDB with 4KB requests following the *randomreadrandomwrite* distribution and read/write ratio being 1:1. The key-value I/O throughput increases from 13 kop/s to

23 kop/s, which is an increase of only about 76.9%. It indicates that impediments exist in the current RocksDB design, hindering us from effectively exploiting the high processing capacity of the new hardware.

In this chapter, we present a set of comprehensive experimental studies to quantitatively measure the performance impact of 3D XPoint SSD on RocksDB. In the experiments, we have identified several unfolded performance bottlenecks and also gained important insight for future deployment of RocksDB in large-scale systems. To our best knowledge, this work is the first study on the performance behavior of RocksDB on 3D XPoint SSD.

For our experiments, we use *db_bench* to generate various types of key-value workloads to conduct extensive experiments and characterize RocksDB. Our purpose is not to compare the absolute performance difference of RocksDB running on different storage devices. Rather, we desire to obtain insightful understanding on the effect of the unique properties of the new-generation storage hardware on RocksDB performance, and to gain system implications for system designers to effectively integrate RocksDB into large-scale systems. We have made the following contributions in this chapter:

(1)This is the first work studying the performance behavior of RocksDB on 3D XPoint SSD. We have identified several important bottlenecks in RocksDB through experiments and analysis, such as the throttling mechanism, the level-0 file query overhead, the read/write interference, etc.

(2)Leveraging our findings, we have also designed and implemented three exemplary case studies to showcase the efficacy of removing the bottlenecks with simple methods on RocksDB, such as avoiding the near-stop situation for workloads with periodic write bursts, dynamic Level-0 file management that improves the throughput by up to 13% and NVM logging approach that reduces the 90th percentile write tail latency by up to 18.8%.

(3)Based on our observations, we have also discussed the related system implications as future guidance to optimize RocksDB on 3D XPoint based SSD and to integrate into large-scale systems and data centers.

The rest of this chapter is organized as follows. Section 5.2 and Section 5.3 present the experimental setup and results. Section 5.4 gives three case studies. In Section 5.5, we summarize our key observations and discuss several implications to users and system designers. Related work is presented in Section 5.6. The final section concludes the chapter.

## 5.2 Methodology

Our experiment system is a two-socket Intel W2600CR server. It is equipped with 16 cores on two Intel(R) Xeon(R) E5-2690 2.90GHz processors and 128 GB memory. An Intel 530 Series SATA Flash SSD, an Intel Data Center P3600 PCIe Flash SSD, and an Intel Optane PCIe 3D XPoint SSD are used as three representative storage devices. We use Ubuntu 14.04 with Linux Kernel 4.4.0, Ext4 file system and RocksDB 5.17.0 in our experiments.

The data set used in our experiments is around 100GB. Accordingly, we purposefully set the physical memory space to be 8GB when system boots, which is about 8% of the data set size. Based on prior study about I/O characterization in large-scale data centers [31], we configure the workload with the value size being 1 KB and write intensive (write operation over 25%). We use *db_bench*, RocksDB's default benchmarking tool, to generate workloads following the *randomreadrandomwrite* distribution in each experiment. Each experiment runs for 300 seconds, which is reasonably long enough to show the performance trend, if not otherwise specified.

## 5.3 Bottleneck Identification

As reported by Yahoo! [120], the percentage of write operations in emerging workloads, such as cloud computing, mobile devices, and social networks, has significantly increased at an unprecedented pace. In 2010, the workloads contained about 10-20% writes, which increased to nearly 50% in 2012. The high insertion ratio contributes to the quickly increasing popularity of RocksDB [153, 23], since the multi-level, append-only structure is highly suitable for handling an intensive traffic of updates on Flash device. In this work, we conduct a comprehensive measurement to study the RocksDB performance with 3D XPoint SSD,

particularly to identify the corresponding bottlenecks on the new-generation hardware.

### 5.3.1 Throttling Mechanism

The memory component of LSM-tree is for two purposes. First, an entry can be inserted (SET) into the memory-resident memtable without involving any I/O cost. Only large, batched I/Os can be seen at the storage level. Second, GET requests to the entries in memory can be served quickly without incurring an I/O to the storage device.

Although the in-memory memtable structure can buffer I/O requests and eliminate data retrievals from the slow on-storage component, it raises cost and power consumption concerns for a large-scale deployment. Thus, in practice, users and system administrators often impose a limit on the number of Level-0 files (36 by default) in RocksDB. When the number of Level-0 files hits a threshold, a *write throttling* mechanism is triggered to purposefully slow down the incoming request traffic to save the memory used for buffering the I/O, until the background process makes enough space in Level-0 by merging and deleting Level-0 files. As the insertion ratio of data center workloads quickly increases, the throttling mechanism would be more frequently triggered, posing huge overhead on the system performance. In this section, we analyze the throttling mechanism and its impact by increasing the insertion ratio from 0% to 100% with 4 parallel processes.

**Finding #1**. On both SATA and PCIe flash SSD, as Figure 5.2 shows, the throughput of RocksDB increases as the insertion ratio increases, because of the reduced number of expensive READ requests. For example, on the PCIe flash SSD, the throughput increases from 32 kop/s to 41.3 kop/s, when the insertion ratio increases from zero to 100%. As Figure 5.4(a) and Figure 5.4(b) show, READ latency is significantly longer on flash SSDs than 3D XPoint SSD. In particular, with a high insertion ratio (90% write), the 90th percentile read tail latency on 3D XPoint SSD is only 251 $\mu$s, in contrast to that on the SATA flash SSD (839 $\mu$s); the 90th percentile write tail latency is 26 $\mu$s, which is close to that on the SATA flash SSD (28 $\mu$s). Thus, the reduced expensive READ operations bring more benefits to the throughput on the flash SSDs.

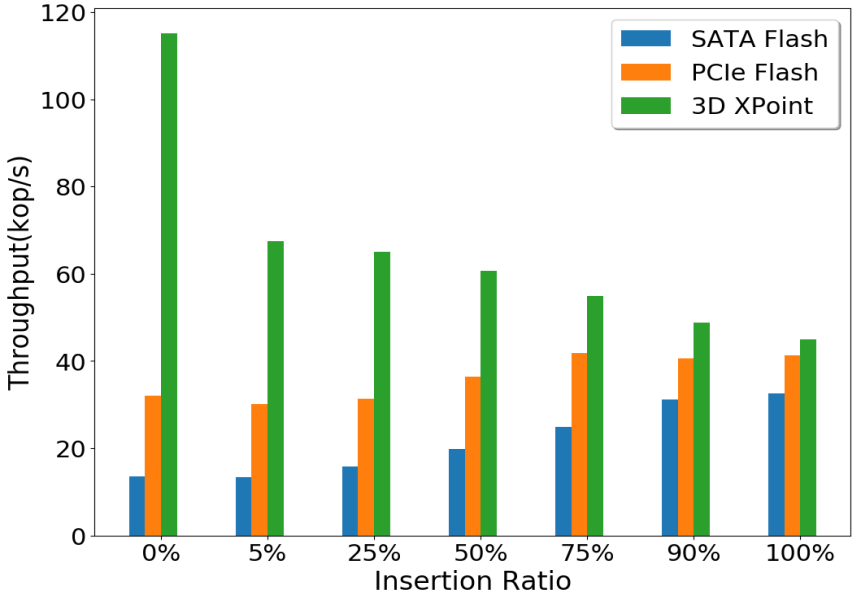
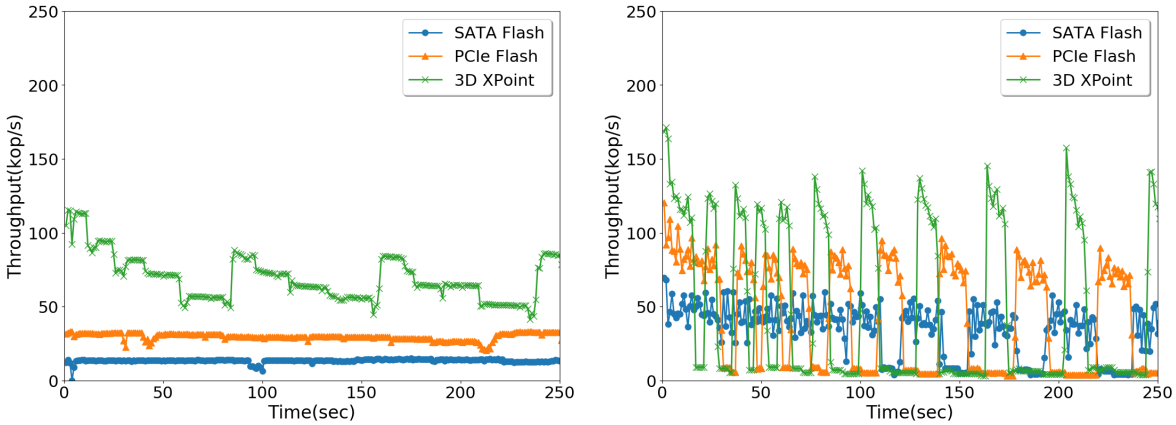



Figure 5.2. Throughput for different insertion ratios

On 3D XPoint SSD, interestingly, Figure 5.2 shows an opposite trend—the throughput decreases from 115 kop/s to 45 kop/s, which is close to the PCIe flash SSD. When insertion ratio continues increasing, the throughput difference becomes less significant, despite the hardware difference. To explain the results, we show more details in Figure 5.3(a) and 5.3(b). When WRITE operations are dominant, the triggered write throttling mechanism would introduce delays for insertion operations, causing performance fluctuation.



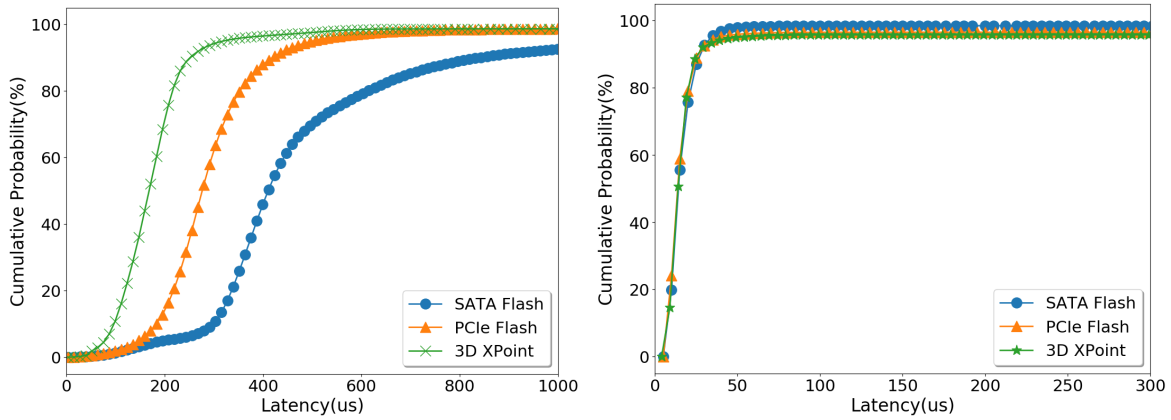

(a) Throughput (5% Write)

(b) Throughput (90% Write)

Figure 5.3. Instantaneous throughput for different insertion ratios

As shown in Figure 5.3(a) and 5.3(b), the throughput variation on 3D XPoint SSD is

clearly evident. The throttling periodically happens, pulling down the throughput. For example, when insertion ratio is 90%, the throughput drops from 169 kop/s to as low as 3 kop/s when throttling happens on 3D XPoint SSD.




(a) Read (90% Write) (b) Write (90% Write)

Figure 5.4. Latency distributions when insertion ratio is 90%

Thus, opposite to our common expectation that 3D XPoint SSD can always provide significantly better performance than flash SSD, we find that such a performance benefit is workload dependent (more significant with a lower insertion ratio). For a write-intensive workload, the current throttling mechanism in RocksDB, which is heavily optimized for flash memory, drags down the performance of RocksDB on 3D XPoint SSD to the level of running on a regular flash SSD.

**Analysis #1**. Algorithm 4 briefly describes the write control process in RocksDB. The write process is put to sleep when a delay is needed (e.g., Level-0 slowdown threshold is hit). The *delayed_write_rate* is adjusted to make sure the amount of data being processed at each interval during compaction roughly constant and stable (i.e., ideally, $Prev\_Bytes$ equals $Esti\_Bytes$).

We use a simple model to explain and estimate the decrease of throughput. Let $\lambda_a$ and $\lambda_s$, respectively, denote the application level throughput, which is the performance perceived by applications, and the system level throughput, which is the processing capacity of the key-value system; $refill\_interval$ denotes the minimum of an injected delay period.

From Algorithm 4, we can see $\lambda_a$ converges to *delayed_write_rate* to guarantee the

request arrival rate roughly equals the processing rate. For a time period $t$ during which one write finishes (i.e., the median latency), we have:

$$\lambda_a \times (refill\_interval + t) = \lambda s \times t \tag{5.1}$$

**Algorithm 4** WRITE CONTROL PROCESS

```
1: Dec=0.8, Inc=1.25
2: delayed_write_rate = usr_defined_value
3: refill_interval =1024 μs
4: num_bytes = last_batch_group_size
5: procedure WRITE(num_bytes)
6:     if need_dalay then
7:         if Prev_Bytes <= Esti_Bytes then
8:             delayed_write_rate* = Dec
9:         else
10:            delayed_write_rate* = Inc
11:        end if
12:        delay = DELAYWRITE(num_bytes)
13:        sleep(delay)
14:    end if
15:    Process(WRITE)
16: end procedure
17: function DELAYWRITE(num_bytes)
18:    time_slice = time_now − last_refill_time
19:    bytes_refilled = time_slice * delayed_write_rate
20:    if bytes_refilled > num_bytes then
21:        if time_slice > refill_interval then
22:            return 0
23:        end if
24:    end if
25:    single_ref = refill_interval * delayed_write_rate
26:    if bytes_refilled + single_ref > num_bytes then
27:        return refill_interval
28:    else
29:        return num_bytes/delayed_write_rate
30:    end if
31: end function
```

Based on Equation 5.1, we can have:

$$\lambda_a = \frac{t}{refill\_interval + t} \times \lambda_s \tag{5.2}$$

According to our measurement, for 3D XPoint SSD, $\lambda s$, which means the background processing capacity of RocksDB when compaction happens, is 190 kop/s and $t$, which means median write latency, is 15 $\mu$s, so we have:

$$\lambda_a = \frac{15}{1024 + 15} \times 190 kop/s = 2.74 kop/s$$

For SATA SSD, $\lambda s$ and $t$ are measured to be 130 kop/s and 15 $\mu$s, respectively, so we have:

$$\lambda_a = \frac{15}{1024 + 15} \times 130 kop/s = 1.88 kop/s$$

The above calculated results are close to our measurement data in Figure 5.3(b). We can see that when the throttling process is triggered, the application level throughput would degrade to a similarly low level, disregarding the hardware differences.

**Discussion #1**. Due to the throttling mechanism, the achievable application-level throughput is much lower than expected, even though the underlying storage hardware is highly capable. The throttling process is such a case that exactly showcases the potential problem—if we blindly operated the same throttling strategy on 3D XPoint SSD in the same way as that on the slow devices, the hardware performance benefit enabled by the new technology would become negligible. As we evolve to the new-generation storage technology, many such optimizations should be carefully reconsidered.

### 5.3.2 Level-0 File Query Overhead

In the LSM-tree structure, the keys in a Level-0 SST files are not sorted. Thus the key ranges could be overlapped. Typically, it requires to search multiple Level-0 files and/or other-level files to find the key-value item. This results in a significant *read amplification* problem on flash based devices. As the storage becomes dramatically faster with the 3D XPoint technology, it is worth studying if this read amplification problem could become less significant.

We desire to study the performance impact with different number of Level-0 files. In

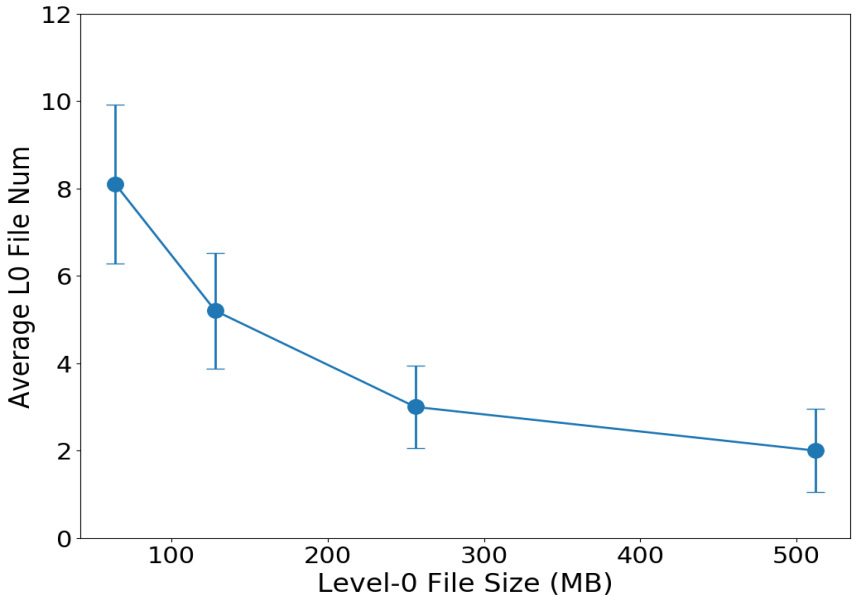



Figure 5.5. Number of Level-0 Files vs. File Size

the current RocksDB, a challenge is that the number of Level-0 files cannot be statically set. In fact, it dynamically changes during runtime. We find that the number of Level-0 files is largely determined by the file size. To show such a relationship between file size and the number of Level-0 files, we vary the Level-0 file size from 32MB to 512MB with 4 concurrent processes and 1:1 READ/WRITE ratio and show the results in Figure 5.5. By setting different Level-0 file sizes, we can roughly control the desired numbers of files generated at Level 0.

**Finding #2**. Figure 5.6 shows the throughput with different number of Level-0 files. Opposite to our expectation, we find that the number of Level-0 files has a significant impact on the performance with 3D XPoint SSD, which is even more pronounced compared to flash SSDs. In specific, the throughput on PCIe flash SSD decreases by only about 12.3%, from 41.5 kop/s to 36.4 kop/s by increasing the number of files from 2 to 8. In contrast, the throughput on 3D-XPoint SSD decreases even more, from 86.4 kop/s to 69.2 kop/s, which is about 19.9%, meaning that the negative impact of the read amplification problem is more significant on faster storage.

To investigate this result, a closer look at READ requests further reveals where the

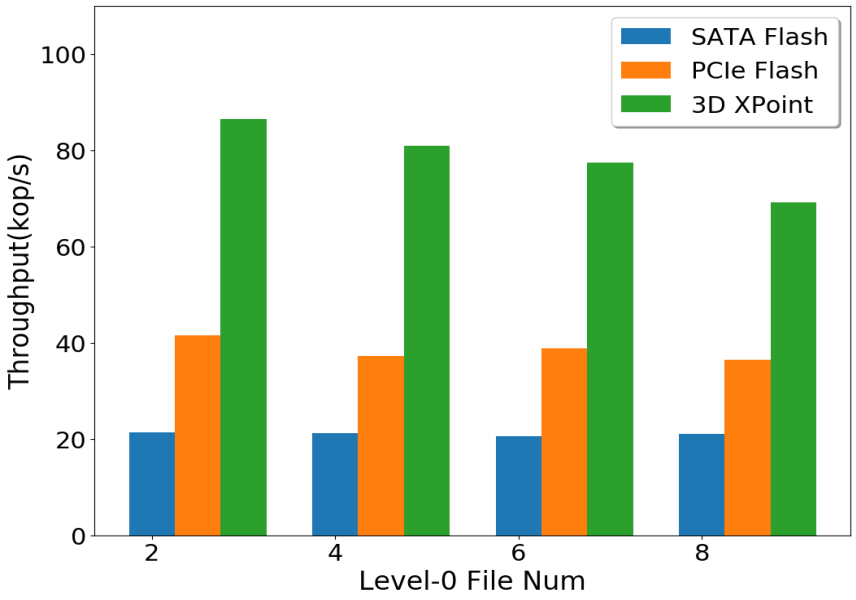



Figure 5.6. Throughput vs. Number of Level-0 Files

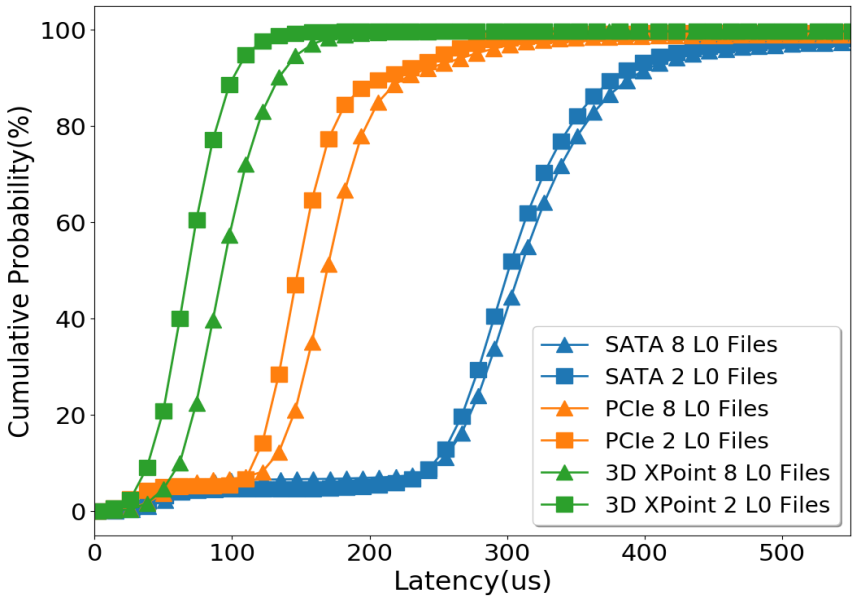



Figure 5.7. Read Latency vs. Number of Level-0 Files

overhead originates. For READ requests, fewer Level-0 files are generally beneficial, on all the three devices. For example, on 3D XPoint storage, the 90th percentile tail latency of READ requests is 134 $\mu$s when the number of Level-0 files is 8, and it drops to 101 $\mu$s when the number of Level-0 files decreases to 2, as Figure 5.7 shows.

This overhead difference can be explained by the following lookup process. Figure 5.8

illustrates the querying processes with large and small Level-0 files, both containing the same number of items. Note that all the small files whose key ranges cover the desired key have to be searched until the key is found.

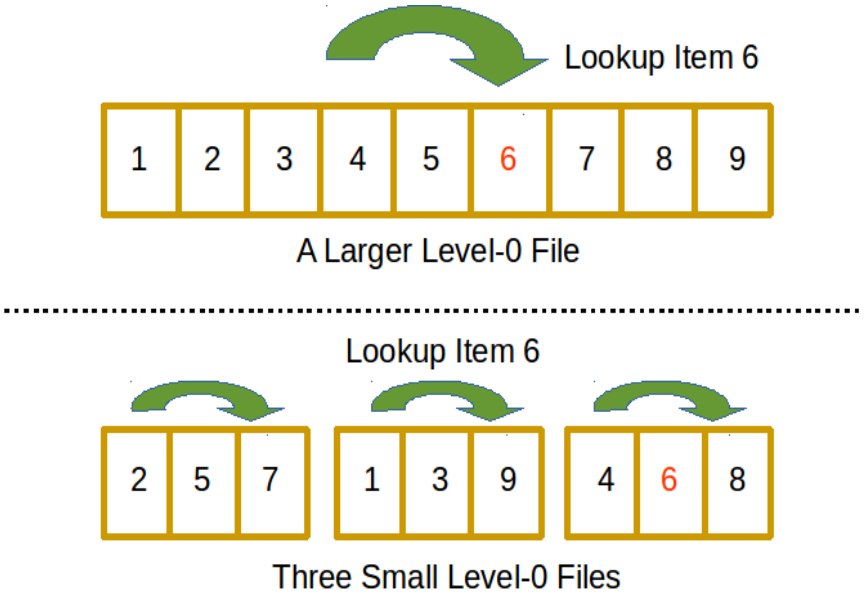



Figure 5.8. Large File vs. Small Files

Assume the small file case has $k$ small files, each with size $N$, while the large file case has a large file with size $k \times N$. Because the files in Level-0 are organized by Skiplist [114], if we split the large file into $k$ small files, the lookup complexity would increase from $O(log(k \times N))$ to $O(k \times log(N))$, which is equal to $O(log(N^k))$. Since a lookup operation in a large Level-0 file on 3D XPoint is not significantly longer (e.g., 9.7 $\mu$s for 256MB vs. 8.5 $\mu$s for 32MB), the number of Level-0 files that need to be searched becomes the dominant factor. As so, maintaining more Level-0 files would lead to a longer READ latency.

**Analysis #2**. Our results show that it is beneficial to maintain a small number of files with a relatively large size. However, it does not mean that we should maintain one single huge Level-0 file. In the current system design, the mutable memtable becomes immutable when it is full. Then another new mutable memtable is allocated to continue serving incoming requests. The immutable table is further flushed to Level 0. Thus a larger Level-0 file, meaning a larger mutable memtable, will cause a longer latency for WRITE operations.

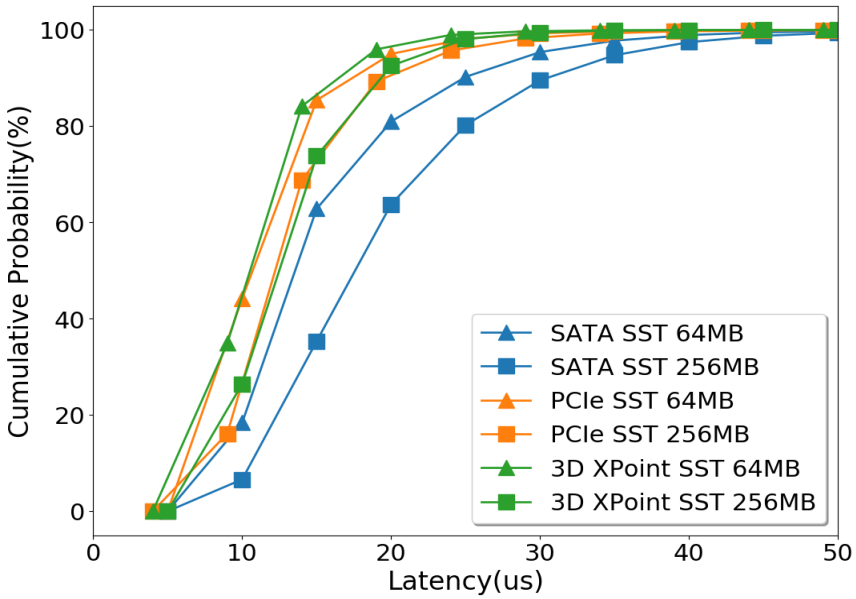



Figure 5.9. Write Latency vs. SST File Size

As shown in Figure 5.9, the 90th percentile tail latency of WRITE operations increases from 25 $\mu$s to 31 $\mu$s, when the SST file size increases from 64MB to 256MB on the SATA flash SSD. It is because WRITE operations are first accumulated in the mutable memtable and then it switches to immutable when full and is further flushed to disk. As the complexity of insertion to a skiplist is $O(log(N))$, a larger memtable would cause a longer latency for WRITE operations.

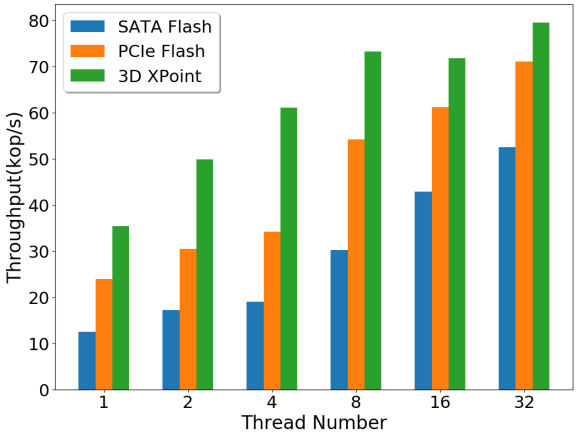
According to the analysis above, fewer but larger Level-0 files would result in smaller READ latencies due to a smaller number of files for search, but a very large Level-0 file size would cause a longer WRITE latency. Such an effect is more evident on 3D XPoint SSD than that on flash based ones, since 3D XPoint SSD is much faster and the overhead involved in Level-0 file queries contributes a more significant portion to the overall request processing time.

**Discussion #2**. Unlike our expectation, as the underlying storage media becomes faster, the role of the memory components and Level-0 files of RocksDB would become even more important. The memory component management, as well as the Level-0 file number setting and the querying mechanism, would have a greater impact on the overall performance.

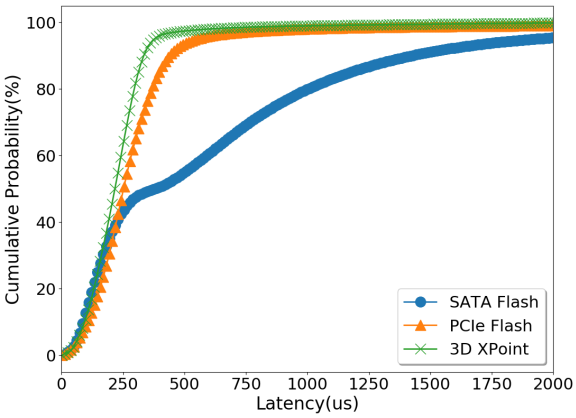
### 5.3.3 Parallelism and Interference

Due to internal parallelism, flash SSDs are particularly suitable for parallel I/O processing. RocksDB supports concurrent writers and write batching to provide high throughput performance. However, the parallel read and write operations may interfere each other [28]. We benchmark RocksDB by varying the number of processes from 1 to 32 with READ/WRITE ratio being 1:1.

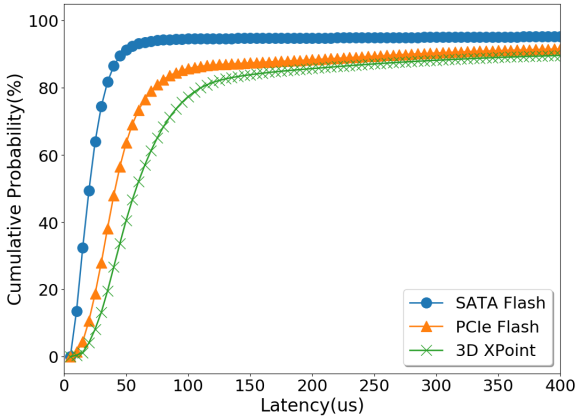
**Finding #3**. Figure 5.10(a) shows that the throughput increases significantly as the




(a) Throughput vs. Parallelism




(b) Read-32Threads




(c) Write-32Threads

Figure 5.10. Performance for different parallelism degree

number of threads increases on all three devices. For example, the throughput increases from 35.4 kop/s to 79.5 kop/s on 3D XPoint SSD by changing the parallelism degree from 1 to 32.

However, READ and WRITE requests behave differently as storage device evolves. As

for READ requests, as Figure 5.10(b) shows, read latency on 3D XPoint SSD is significantly smaller than flash SSDs. With 32 threads, the 90th percentile tail latency on the 3D XPoint SSD is 335 $\mu$s, which is 76% smaller than that on the SATA flash SSD (1.4 ms).

The WRITE requests show opposite results. The 90th percentile tail latency on the 3D XPoint SSD (440 $\mu$s) is significantly *longer* than that on the SATA flash SSD (47 $\mu$s) when the thread number is 32, as shown in Figure 5.10(c).

**Algorithm 5** PIPELINED WRITE PROCESS

```
1: procedure PIPELINEDWRITEIMPL
2:     if writer.state==Memtable_Leader then
3:         for writers in Group do
4:             writer.state=Memtable_Writer
5:         end for
6:     else if writer.state==Memtable_Writer then
7:         Write()
8:     else
9:         Await()
10:    end if
11:    PickNewLeader()
12: end procedure
```

The pipelined write process, which is used in RocksDB to improve concurrent write throughput, can explain this phenomena. By default, RocksDB keeps one single write thread queue for concurrent writers. A thread becomes the leader of the write batch group, when it reaches the head of the queue, and is responsible for flushing the Memtable and WAL for the batch group. The whole process is described in Algorithm 5.

3D XPoint SSD provides a high processing speed for READ requests, which unfortunately places a high pressure on the concurrent WRITE requests. A result is that more awaiting writes are accumulated during the same period of time. As shown in Figure 5.11, the average number of waiting threads on 3D XPoint SSD is evidently larger than that on flash devices with 32 threads. More waiting threads would lead to a longer waiting time for WRITE requests on 3D XPoint SSD than on flash SSD, as Figure 5.10(c) shows.

**Discussion #3**. Opposite to the traditional understanding that 3D XPoint SSD always provides low latency service than flash SSD, WRITE requests could take longer

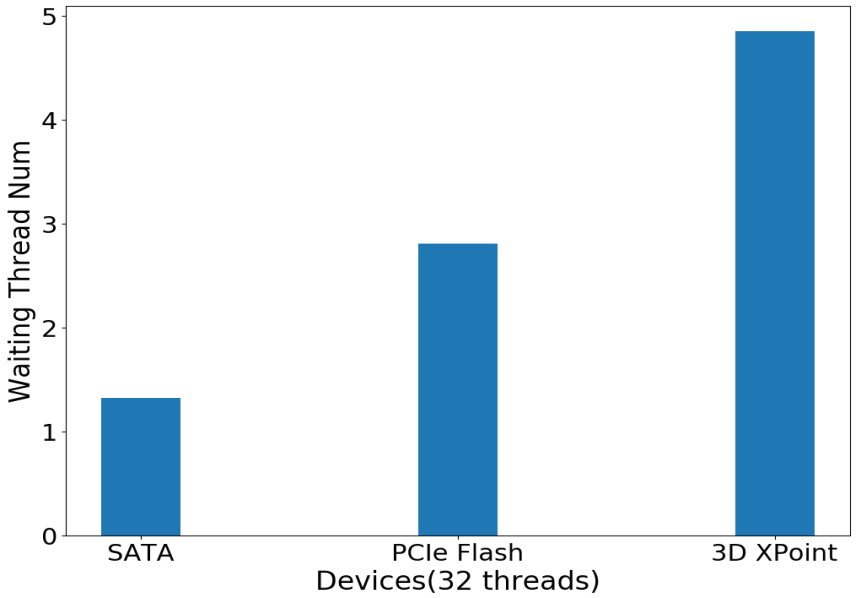



Figure 5.11. Waiting Threads vs. Devices

to complete in mixed workloads. With a high arrival rate of requests and intensive READ/WRITE interference, software designs in RocksDB, such as pipelined write process, would play a significant role for request latency. As a result, more accumulated WRITE requests would result in a longer processing time.

### 5.3.4 Logging

LSM-tree based key-value store usually manages a Write Ahead Log (WAL) for data recovery and system consistency. Specially, an update in RocksDB is logged in the write ahead log (WAL) first and then in the in-memory data structure, memtable, which is flushed to SST files on disk later.

Though being crucial, logging mechanisms can incur high performance overhead. With a high-speed 3D XPoint SSD, it is worth studying if the involved overhead could be weakened. We study their impact by enabling and disabling logging mechanisms, using a workload with READ/WRITE being 1:9.

**Finding #4**. According to Figure 5.12, WAL still has a significant impact on the write performance, even with the much faster 3D XPoint SSD. The 90% tail latency of WRITE operations was reduced from about 54 $\mu$s to about 22 $\mu$s, if disabling the WAL mechanism

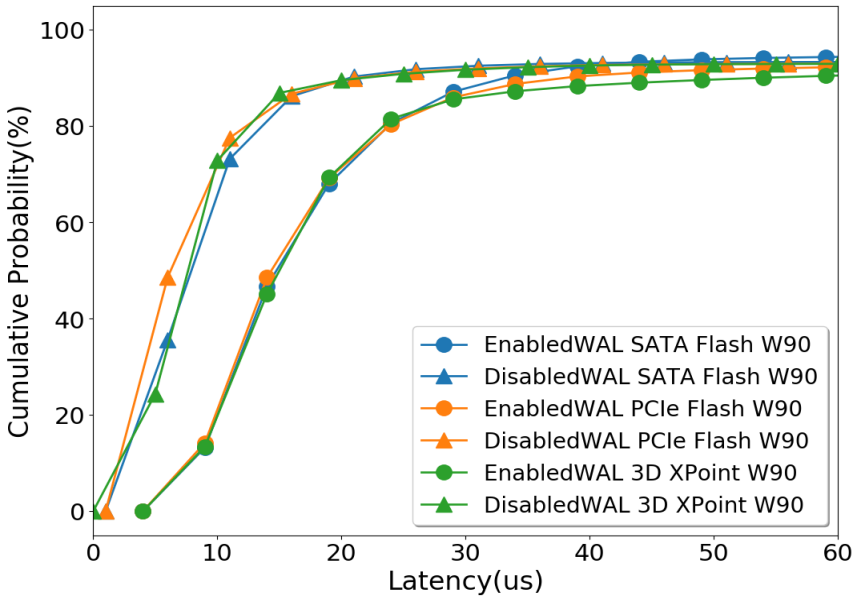



Figure 5.12. Write Latency vs. WAL

on 3D XPoint SSD. It is because when a write request is issued, the WAL update is first written to the write buffer. Then the real write operation is executed in the in-memory data structure, memtable. The WAL and memtable are flushed to disk asynchronously and separately later. Thus the device differences are diminished. Our results show that logging is still an important factor hindering the performance of RocksDB on all devices.

**Discussion #4**. WAL shows great performance penalty, even on 3D XPoint SSD. We need to consider solutions to speed up this process. Further optimization for this costly process remains an important demand to achieve stable performance, even on high-speed storage devices.

## 5.4 Case Studies

In this section, we present three case studies to illustrate how to optimize the performance of RocksDB by overcoming the bottlenecks discussed above. It is worth noting that these pilot solutions are not intended to be fully optimized designs but to serve as examples to showcase the potential optimizations that could be enabled based on our findings.

### 5.4.1 Removing Near-Stop Status

The original Level-0 file management mechanism works as follows. When the number of Level-0 files reaches a compaction trigger threshold, the background compaction process is launched to merge the files that have overlapping keys in Level-0 and the following levels. When the number of Level-0 files reaches a slowdown threshold, a throttling process as described in Algorithm 4 will be triggered. In extreme cases, when the number of Level-0 files reaches *stop_threshold*, a stopping process will be launched to constrain the Level-0 size according to the user's requirements.

Such a simple throttling mechanism is particularly detrimental when the system undergoes a periodically appearing "flash of crowd" situation, in which write bursts periodically appear and pull the system into a near-stop (under 10 kop/s) situation. The real-time throughput, as we show in Section 5.3.1, can be as low as 3 kop/s, rendering the system almost unusable. In order to remove this near-stop situation, we propose a two-stage throttling mechanism to avoid a sharp performance drop. This is particularly beneficial to workloads with periodic write bursts, which is a common case in large scale systems [148, 96, 95].

*Stage 1: Slight Throttling.* When the number of Level-0 files reaches the user-defined *slowdown_threshold*, the maximum acceptable *delayed_write_rate* is used to set the write rate for throttling.

*Stage 2: Aggressive Throttling.* When the number of Level-0 files continues to grow and reaches the second-level throttling threshold, which is defined as (*slowdown_threshold* + *stop_threshold*)/2, the more aggressive throttling, Algorithm 4, will be applied to slow down the incoming traffic.

To evaluate the effect of our proposed two-stage throttling mechanism, we benchmark RocksDB using a workload with READ/WRITE ratio being 1:1. This workload has a periodic write burst (READ/WRITE ratio being 1:9) lasting for 25 seconds per minute. As Figure 5.13 shows, the throttling throughput of the original design is about 9 kop/s

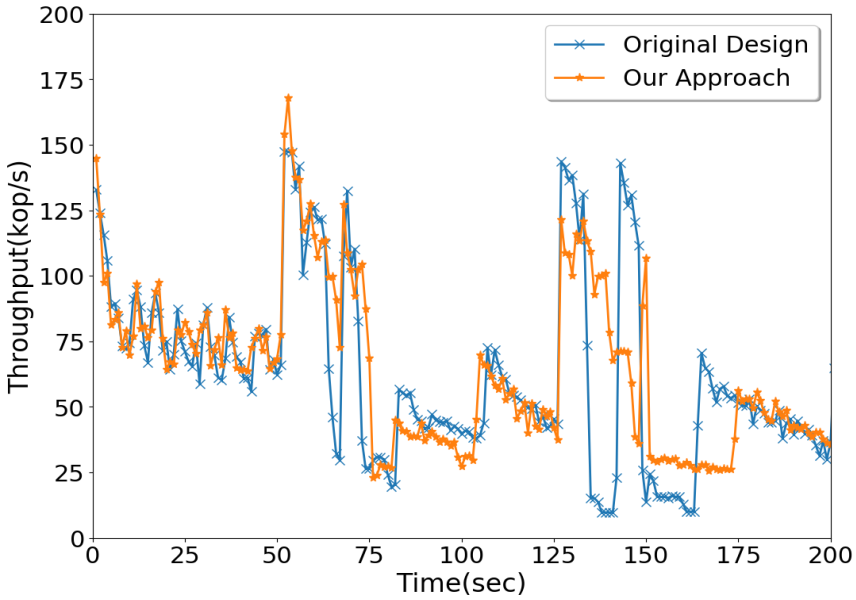



Figure 5.13. Throughput vs. Time

from 135s to 142s, 12 kop/s from 151s to 163s. On the contrary, we can hardly see near-stop periods with our proposed approach, meaning that even such a simple approach can effectively remove the near-stop status for workloads with periodic write bursts.

### 5.4.2 Dynamic Level-0 Management

As Section 5.3.2 shows, the number and size of Level-0 files have a great impact on the performance of RocksDB. In specific, having fewer level-0 files would reduce the READ latency due to a smaller number of files for searching, while a smaller level-0 file size would reduce the WRITE latency due to the reduced insertion overhead to a smaller skiplist. Therefore, there exists a tradeoff point between the two factors, depending on the workloads (READ or WRITE intensive).

Assuming the aggregate volume of level-0 files is constant, we present a simple dynamic Level-0 management mechanism that optimizes the number of Level-0 files and the file size based the READ/WRITE ratio measured online.

The RocksDB is initialized to throttle writes when the number of files at Level-0 reaches 24. Then we measure the READ/WRITE ratio during runtime. When the workload is observed to be WRITE intensive, we configure the Level-0 to have smaller, yet more (e.g.,

24 in our example) files; When the workload is READ intensive, we configure the Level-0 to have larger, yet fewer (e.g., 6 in our example) files.

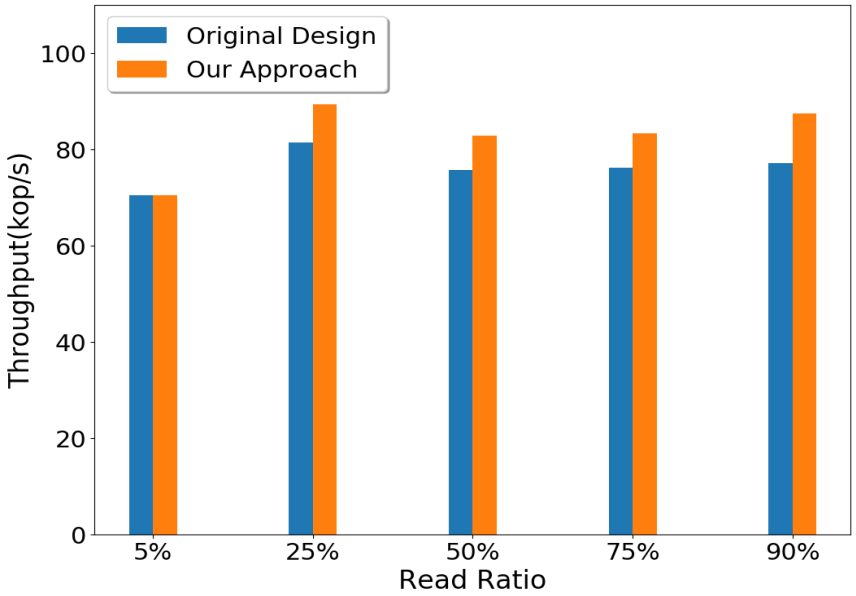



Figure 5.14. Throughput vs. Read Ratio

To evaluate the effect of our proposed dynamic Level-0 management mechanism, we benchmark RocksDB with the default configuration by varying the READ ratio from 5% to 90%. In our example, we empirically tag the workloads to be WRITE intensive, if write operations account for more than 25%. According to Figure 5.14, our proposed approach can improve the throughput of RocksDB in most cases. For example, when the READ ratio is 90%, our approach can improve the throughput from 77 kop/s to 87 kop/s, which is an improvement of 13%. When the read ratio is 5%, these two approaches achieves similar throughput.

### 5.4.3 Reducing Logging Overhead

As we show in Section 5.3.4, WAL incurs noticeable overhead to RocksDB. In order to reduce the logging overhead, a potential solution is to separate WAL out and relocate it to a faster device, such as byte-addressable non-volatile memory (NVM). Since the size of WAL is quite small, it is reasonable to deploy a rather small NVM device to accumulate WAL updates very quickly.

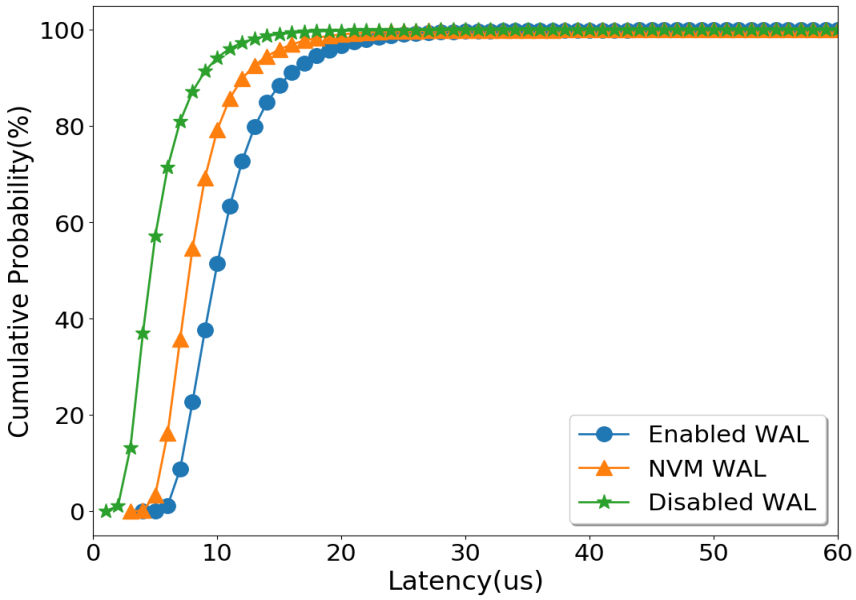



Figure 5.15. Latency vs. Log Approaches

To evaluate the effect of our proposed NVM logging mechanism, we emulate NVM by using Linux `tmpfs` in DRAM. We benchmark RocksDB with the default configuration by setting the insertion ratio to be 50%. Figure 5.15 shows that logging in NVM can effectively reduce the overhead. For example, the 90th percentile write tail latency, decreases from about 16 $\mu$s to 13 $\mu$s, which is an improvement of 18.8%. However, compared with disabling WAL, logging in NVM still incurs noticeable performance penalty. It indicates that the WAL overhead can not be totally removed using such a simple approach and a more sophisticated solution is needed to fully address it.

## 5.5 System Implications and Discussions

We have studied the performance of RocksDB on three generations of SSDs through extensive experiments and analysis. Here we present important implications for system designers and practitioners to effectively deploy RocksDB and optimize the software management on 3D XPoint SSD.

**Reads**. For READ operations, the new 3D XPoint SSD provides much better performance than flash SSD in most cases, which is as expected. We also find that other factors, such as Level-0 file management, can significantly affect READ performance. In fact, fewer

Level-0 files means less search overhead, which is proven to be noticeably beneficial for the READ performance. In addition, since the number of level-0 files is greatly affected by in-memory component (e.g., memtable size), the memory management could have non-negligible, indirect performance impact than expected. Thus optimizing data management in memory also becomes important as the storage performance continues to improve. Finally, since the relative cost of I/O operations becomes lower, reducing the total number of I/O accesses becomes important. For example, combining multiple small I/Os into big ones would result in noticeably lower overhead.

**Writes**. For WRITE operations, 3D XPoint SSD can provide better performance than flash SSDs, for workloads with light to moderate amount of writes. With a heavy load of write requests, however, many optimizations in the current key-value store design, such as the throttling process and write pipelining, which are customized and heavily tuned for flash SSDs, can create unexpected negative effect on 3D XPoint SSD. It unfortunately cancels the great performance advantage of 3D XPoint SSD, despite the much faster hardware. As real-world applications are becoming increasingly WRITE intensive, we must carefully reconsider such "obsolete" optimizations and develop new schemes to fit the properties of the new-generation hardware.

**Memtable**. The memtable in RocksDB has an opposite effect on READ and WRITE operations. On a faster 3D XPoint SSD, a larger memtable in LSM-tree based data store still can bring benefits for READ-intensive workloads. It is because a larger memtable would result in fewer Level-0 files, which reduces the search time at Level 0. Due to the smaller performance gap between memory and underlying storage, such a benefit is more significant on 3D XPoint SSD.

For WRITE-intensive workloads, a large memtable increases the overhead of insertion operations, causing unanticipated performance degradation. The current system design largely ignores this issue, simply assuming a large in-memory memtable would be generally beneficial for both READ and WRITE operations. More sophisticated mechanisms are

needed for performance improvement. For example, since memtable size has opposite effects on READ and WRITE requests, system designers may consider dynamically merging multiple small mutable memtables into a large immutable memtable according to workload characteristics.

**Read/Write Interference**. We have observed significant performance improvement with a high READ/WRITE parallelism in all three SSDs. Although RocksDB on 3D XPoint SSD outperforms flash based SSDs in terms of throughput, Read/Write interference on 3D XPoint SSD has a more pronounced impact on the WRITE request latency than on flash SSD. It is because 3D XPoint SSD processes READ requests significantly faster, which in turn results in more WRITE threads being queued up and waiting until the previous batch writing is done.

Many optimizations can be done to alleviate this problem. For example, we may use multiple short write thread queues rather than one single long queue, creating more parallelism. We can also associate write requests with different priority-based performance policies. For example, latency sensitive requests can be processed with a high priority. For practitioners, selecting a proper parallelism setting to balance throughput and latency is a wise choice.

**Logging**. Logging is a necessary mechanism to ensure the reliability of the data store. However, it is also an important factor affecting performance, which is particularly visible on the high-speed 3D XPoint SSD. In fact, we see a non-trivial performance improvement in both throughput and latency by speeding up the WAL accesses. In practice, system designers should consider optimizing the current logging mechanism or proposing novel designs to minimize the overhead. For example, compressing and condensing the data written to the log could help reduce the I/O traffic and correspondingly reduce the time overhead of logging.

The essential difference between flash SSD and 3D XPoint SSD with RocksDB is not only the significant speed improvement of the physical storage media, but also the contin-

uously diminishing performance gap between memory and storage. As a result, although several original designs, such as write throttling mechanism, Level-0 file management mechanism, and the pipelined writing mechanism, work well on flash SSD, the involved overhead becomes non-negligible on 3D XPoint SSD. Our work has provided quantitative results, explanations and guidance for integrating RocksDB on 3D XPoint SSD into current enterprise data centers.

## 5.6 Related Work

LSM-tree based key-value stores have been extensively studied. Prior work can be roughly divided into two categories: improving and modeling the key-value store performance.

**Improving Key-value Store Performance.** Given the importance of write performance, HyperLevelDB [50] aims to increase the write throughput by introducing a fine-grained locking and a new compaction algorithm. bLSM [120] proposes a new merge scheduler to effectively optimize both write latency and throughput. VT-Tree [130] introduces an additional level of indirection for sorting data and also eliminates unnecessary data copy operations. WiscKey [97] takes a different approach to improve performance. It separates keys and values and moves the values out of the LSM structure. LOCS [157] exploits the internal parallelism of open-channel SSDs by leveraging the exposed low-level hardware details to improve the compaction performance. Similarly, cLSM [57] also aims to increase the concurrency in LSM-tree based stores. LSbM-tree [147] develops an on-disk buffer to mitigate the effect of invalidated system buffer cache to enhance the compaction efficiency. LSM-trie [163] develops a data structure, called *trie*, to organize keys, therefore mitigating the problem of write amplification. PebblesDB [117] proposes a new structure, called Fragmented Log-Structured Merge Trees (FLSM), to achieve high write performance. In this chapter, we particularly focus on RocksDB, a popular key-value store, and study the potential performance impact of the new storage hardware.

**Modeling Key-value Store Performance.** Many prior studies have evaluated the

performance of the LSM-tree based key-value store designs. Some focus on analytic metrics, such as read/write amplification factor, analyzing the experiment results and studying the design rationales [138, 50, 92, 103, 120, 91]. Some other works use the concept of per-operation costs for analysis [157, 154, 130, 131]. Besides, other factors, such as operating system [84] and hardware [133, 28], also have significant performance impact on the key-value stores. These prior works more focus on the key-value store itself or other specific layers and lack sufficient studies on the interaction between the software design and the underlying hardware. Our work focuses on this aspect, in particular how the current software design of RocksDB interact with the underlying storage devices, which are quickly evolving. Thus this paper is largely orthogonal to these prior works.

Among the related work, NoveLSM [76], which uses a byte-addressable skip list, directly manipulates the persistent state, and enhances read I/O parallelism on NVM, is the closest to this work. However, our work focuses on identifying the bottlenecks of the start-of-art key-value store design on the emerging 3D XPoint SSD, which is fundamentally different from byte-addressable NVM.

### 5.7 Conclusion

In this chapter, we have conducted comprehensive experiments to understand the performance impact of the dramatically faster underlying devices on RocksDB. Besides confirming the great performance potential of 3D XPoint SSD. Our experimental results have also observed many unexpected bottlenecks of RocksDB with this new storage technology. Many software designs that are originally customized for flash devices, such as the throttling mechanism and the Level-0 file management mechanism, need a reconsideration to fully exploit the performance potentiality of the new generation hardware. We have also developed three case studies as examples to showcase possible optimizations based on our findings, which can remove the near-stop status and reduce write latency by up to 18.8%. Finally, we have discussed the system implications for system designers and practitioners. Our work shows that the rapidly evolving storage technology, such as 3D XPoint SSD, not

only presents many challenges but also opens numerous opportunities.

# CHAPTER 6
# LIMITATIONS AND FUTURE WORK

Our work focuses on understanding and optimizing flash-based key-value systems in data centers from the perspective of workloads, software, and hardware. We have proposed an on-line compression scheme, called *SlimCache*, to improve caching efficiency and studied five representative auto-tuning approaches on RocksDB for high throughput and low latency. We further investigate the performance impact of the emerging 3D XPoint technology to the popular RocksDB. The experimental results have demonstrated the efficiency and effectiveness of our findings and proposed solutions.

Our work has several limitations. First, as a replacement of in-memory cache and an enhancement of Twitter's Fatcache, our proposed compression scheme, *SlimCache*, does not guarantee data persistence when system crashes. In our current design, the mapping structure between keys and values is stored in volatile memory, rather than flash SSDs to simplify the system design. However, the key-value pairs hosted in *SlimCache* would become invalid after sudden system crash and the system needs to warm up again after restart. Second, our auto-tuning work mainly focuses on the trade-off between throughput and tail latency, which is a common consideration in real world deployment. Other optimization metrics, such as power consumption and capital cost, are also worth exploring. Third, although our performance study of RocksDB on 3D XPoint SSDs reveals several unexpected performance bottlenecks and potentials of RocksDB on the newly emerging storage medium and presents three exemplary case studies to showcase the efficacy of removing these bottlenecks with simple methods, a more holistic redesign of RocksDB is needed to further exploit the performance advantages of the new storage hardware.

We can further improve our work from the following perspectives. To guarantee the data persistence in *SlimCache* and accelerate the warm-up process when system restarts, we plan to explore the feasibility of storing the hash table to non-volatile memory. As for the auto-tuning work, based on our findings, we plan to study multi-objective optimization

algorithms for other optimization targets, such as the balance between performance and power consumption, etc. Besides, we have collected the experimental data with synthesized workloads following representative zipfian, hotspot, and random distributions. We plan to further study these algorithms by repeating real enterprise workloads on our platform. Finally, to further optimize RocksDB on 3D XPoint SSDs, we plan to propose a holistic system design based on our findings, which expects to remove the periodical performance drop and alleviate the read/write inferences, etc. In order to take advantage of sequential operations on flash-based SSDs, some current designs, such as compaction process, attempt to avoid random access, but incur expensive overhead to the system. We may reconsider the sorting data structure on disks to provide a more suitable key-value system design for 3D XPoint SSDs, by considering the new features of 3D XPoint technology.

# CHAPTER 7
# CONCLUSIONS

Our work on understanding and optimizing flash-based key-value systems from the perspective of workloads, software, and hardware with the evolution of data centers include both scheme design and performance measurement.

Considering the unique characteristics of key-value systems, such as small-sized items, highly skewed access pattern and continuously increasing scale, we have designed an on-line compression scheme, *SlimCache*, to expand the effectively usable cache space, increase the hit ratio, and improve the cache performance. *SlimCache* presents a number of techniques, such as unified management of compressed and uncompressed data, dynamically determining compression granularity, efficient hot/cold data separation, optimized garbage collection, and adaptive cache partitioning. Furthermore, in order to appropriately configure the increasingly complex key-value data store, which has more than 50 parameters with additional hardware and system settings, we have quantitatively studied and compared five multi-objective optimization methods for auto-tuning the performance of RocksDB in terms of throughput, the 99th percentile tail latency, convergence time, real-time system throughput, and the iteration process, etc. Last but not least, we have conducted an in-depth, comprehensive measurement work on flash-optimized RocksDB with recently emerging 3D XPoint SSDs, which provide much lower latency and higher throughput and remove many long-existing concerns on flash SSDs, such as the read-write speed disparity, slow random write, and endurance problems, to provide system implications for future optimizations for RocksDB on this revolutionary storage hardware.

In the future work, we plan to optimize *SlimCache* for data persistence, extend our auto-tuning work for other optimization goals and provide a holistic key-value design for 3D XPoint SSDs. We hope our work can pave the way for system designers and practitioners to optimize key-value systems on flash-based SSDs and emerging 3D XPoint based SSDs.

# APPENDIX A
# THE IEEE PUBLICATION AGREEMENT

This dissertation reuses part of the materials of three of the author's papers [70, 71] that were published by the Institute of Electrical and Electronics Engineers (IEEE). Based on the publication agreement of the IEEE, authors can reuse any portion of their published articles in their dissertation as long as they cite the work in a proper manner. The permission policy of the IEEE for reusing the published materials [65] is as follows:

## Can I reuse my published article in my thesis?

You may reuse your published article in your thesis or dissertation without requesting permission, provided that you fulfill the following requirements depending on which aspects of the article you wish to reuse.

- **Text excerpts:** Provide the full citation of the original published article followed by the IEEE copyright line: © 20XX IEEE. If you are reusing a substantial portion of your article and you are not the senior author, obtain the senior author's approval before reusing the text.
- **Graphics and tables:** The IEEE copyright line (© 20XX IEEE) should appear with each reprinted graphic and table.
- **Full text article:** Include the following copyright notice in the references: "© 20XX IEEE. Reprinted, with permission, from [full citation of original published article]."

When posting your thesis on your university website, include the following message:

> "In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [name of university or educational entity]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink. If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation."

Only the accepted version of your article, ***not the final published version***, may be posted online in your thesis.

## REFERENCES


[1] Gzip. http://www.gzip.org/.

[2] Snappy. https://github.com/google/snappy.

[3] Zlib. https://zlib.net/.

[4] Kolcomplexity. http://theory.stanford.edu/~trevisan/cs154-12/kolcomplexity-rev.pdf, 2012.

[5] WiredTiger Storage Engine. https://docs.mongodb.com/manual/core/wiredtiger/, 2019.

[6] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*, Chicago, IL, USA, June 26-29 2006.

[7] Bulent Abali, Mohammad Banikazemi, Xiawei Shen, Hubertus Franke, Dan E. Poff, and T. Basil Smith. Hardware Compressed Main Memory: Operating System Support and Performance Evaluation. *IEEE Transactions on Computers*, 50(11):1219–1233, November 2001.

[8] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of 2008 USENIX Annual Technical Conference (USENIX ATC '08)*, Boston, MA, June 22-27 2008.

[9] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*, pages 1009–1024, New York, NY, USA, 2017. ACM.

[10] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *Proceedings of 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, pages 469–482, Boston, MA, March 2017. USENIX Association.

[11] Zhongqi An, Zhengyu Zhang, Qiang Li, Jing Xing, Hao Du, Zhan Wang, Zhigang Huo, and Jie Ma. Optimizing the Datapath for Key-value Middleware with NVMe SSDs over RDMA Interconnects. In *Proceedings of 2017 IEEE International Conference on Cluster Computing (CLUSTER '17)*, pages 582–586, 09 2017.

[12] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of 2012 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*, London, UK, June 11-15 2012.

[13] Marton Balassi, Paris Carbone, and etc. Apache Flink. `https://flink.apache.org/`, 2014.

[14] David Beasley, David R. Bull, and Ralph R. Martinz. An Overview of Genetic Algorithms : Part 1, Fundamentals. *University Computing*, 15(2):58–69, 1993.

[15] Babak Behzad, Huong Vu Thanh Luu, Joseph Huchette, Surendra Byna, Prabhat, Ruth Aydt, Quincey Koziol, and Marc Snir. Taming Parallel I/O Complexity with Auto-tuning. In *Proceedings of 2013 International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*, pages 1–12, Nov 2013.

[16] Vicenc Beltran, Jordi Torres, and Eduard Ayguade. Improving Web Server Performance Through Main Memory Compression. In *Proceedings of the 14th IEEE International Conference on Parallel and Distributed Systems (ICPADS '08)*, Melbourne, VIC, Australia, December 8-10 2008.

[17] Michaela Blott, Kimon Karras, Ling Liu, Kees Vissers, Jeremia Baer, and Zsolt Istvan. Achieving 10Gbps Line-rate Key-value Stores with FPGAs. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '13)*, San Jose, CA, June 25-26 2013.

[18] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOMM '99)*, New York, NY, March 21-25 1999.

[19] Lam Thu Bui, Sameer Alam, Lam Thu Bui, and Sameer Alam. *Multi-Objective Optimization in Computational Intelligence: Theory and Practice (Premier Reference Source)*. IGI Global, Hershey, PA, USA, 1 edition, 2008.

[20] Zhen Cao, Vasily Tarasov, Sachin Tiwari, and Erez Zadok. Towards Better Understanding of Black-box Auto-tuning: A Comparative Analysis for Storage Systems. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18)*, pages 893–907, Berkeley, CA, USA, 2018. USENIX Association.

[21] Damiano Carra and Pietro Michiard. Memory Partitioning in Memcached: An Experimental Performance Analysis. In *Proceedings of 2014 IEEE International Conference on Communications (ICC '14)*, Sydney, Austrilia, June 2014.

[22] George Casella and Roger L Berger. *Statistical Inference*, volume 2. Duxbury Pacific Grove, CA, 2002.

[23] Ceph. Ceph. `https://ceph.io/`.

[24] Ceph. Ceph Internal Storage. `https://docs.ceph.com/docs/master/rados/configuration/storage-devices/`.

[25] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *Proceedings of 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, pages 1007–1019, Boston, MA, 2018. USENIX Association.

[26] Feng Chen, Binbing Hou, and Rubao Lee. Internal Parallelism of Flash Memory-Based Solid-State Drives. *ACM Transactions on Storage*, 12(3):13:1–13:39, May 2016.

[27] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '09)*, pages 181–192, New York, NY, USA, 2009. ACM.

[28] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential Roles of Exploiting Internal Parallelism of Flash Memory based Solid State Drives in High-speed Data Processing. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture (HPCA '11)*, San Antonio, Texas, February 12-16 2011.

[29] Seonghyeog Choi and Euiseong Seo. A Selective Compression Scheme for In-Memory Cache of Large-Scale File systems. In *Proceedings of 2017 International Conference on Electronics, Information, and Communication (ICEIC '17)*, Phuket, Thailand, January 11 2017.

[30] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. Memshare: A Dynamic Multi-tenant Key-value Cache. In *Proceedings of 2017 USENIX Annual Technical Conference (USENIX ATC '17)*, pages 321–334, Santa Clara, CA, 2017. USENIX Association.

[31] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of 2010 ACM Symposium on Cloud Computing (SoCC '10)*, Indianapolis,Indiana, June 2010.

[32] Gordon V. Cormack. Data Compression on a Database System. *Communications of the ACM*, 28(12):1336–1342, December 1985.

[33] Dong Dai, Philip Carns, Robert B. Ross, John Jenkins, Kyle Blauer, and Yong Chen. GraphTrek: Asynchronous Graph Traversal for Property Graph-Based Metadata Management. In *Proceedings of 2015 IEEE International Conference on Cluster Computing (CLUSTER '15)*, pages 284–293, Sep. 2015.

[34] Dong Dai, Yong Chen, Philip Carns, John Jenkins, and Robert Ross. Lightweight Provenance Service for High-Performance Computing. In *Proceedings of 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT '17)*, pages 117–129, Sep. 2017.

[35] Rodrigo S. de Castro, Alair Pereira do Lago, and Dilma Da Silva. Adaptive Compressed Caching: Design and Implementation. In *Proceedings of the 15th Symposium*

*on Computer Architecture and High Performance Computing (SBAC-PAD '03)*, Nov 2003.

[36] Jeffrey Dean and Luiz Andre Barroso. The Tail at Scale. *Communications of the ACM*, 56:74–80, 2013.

[37] Kalyanmoy Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley Sons, Inc., USA, 2001.

[38] Kalyanmoy Deb and Rituparna Datta. Hybrid Evolutionary Multi-objective Optimization and Analysis of Machining Operations. *Engineering Optimization*, 44(6):685–706, 2012.

[39] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002.

[40] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: High Throughput Persistent Key-Value Store. *Proceedings of the VLDB Endowment*, 3(1–2):1414–1425, September 2010.

[41] Biplob Debnath, Sudipta Sengupta, and Jin Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage. In *Proceedings of the 2011 International Conference on Management of Data (SIGMOD '11)*, Athens, Greece, Jun 2011.

[42] Dell. Flash Storage Hits Its Stride in the Data Center. `https://i.dell.com/sites/csdocuments/Shared-Content_data-Sheets_Documents/en/us/Dell_SD_Flash_Storage_Hits_Its_Stride_in_the_Data_Center.pdf`.

[43] Cassandra Developers. Apache Cassandra. `http://cassandra.apache.org/`, 2016.

[44] HBase Developers. Apache HBase. `https://hbase.apache.org/`.

[45] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning Database Configuration Parameters with iTuned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, August 2009.

[46] Juan Durillo and Thomas Fahringer. From Single-to Multi-objective Auto-tuning of Programs: Advantages and Implications. *Sci. Program.*, 22(4):285–297, October 2014.

[47] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: A Hybrid Key-value Cache that Controls Flash Write Amplification. In *Proceedings of 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, pages 65–78, Boston, MA, February 2019. USENIX Association.

[48] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM Footprint with NVM in Facebook. In *Proceedings of the 13th EuroSys Conference (EuroSys '18)*, pages 42:1–42:13, New York, NY, USA, 2018. ACM.

[49] EMC. The Rise of the Modern Data Center. `https://www.emc.com/collateral/white-papers/emc-modern-data-center-whitepaper.pdf`.

[50] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex. `http://hyperdex.org/performance/leveldb/`, 2012.

[51] Facebook. McDipper: A Key-value Cache for Flash Storage. https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920/.

[52] Annie Foong and Frank Hady. Storage as Fast as Rest of the System. In *Proceedings of 2016 IEEE 8th International Memory Workshop (IMW '16)*, 05 2016.

[53] Python Software Foundation. Text Generator based on Markov Chain. `https://pypi.python.org/pypi/markovgen/0.5`, 2019.

[54] Kingwa Fu. Weiboscope Open Data. https://hub.hku.hk/cris/dataset/dataset107483, 2017.

[55] Kingwa Fu, CH Chan, and Michael Chau. Assessing Censorship on Microblogs in China: Discriminatory Keyword Analysis and Impact Evaluation of the Real Name Registration Policy. *IEEE Internet Computing*, 17(3):42–50, 2013.

[56] Sanjay Ghemawat and Jeff Dean. LevelDB. `http://leveldb.org/`, 2014.

[57] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling Concurrent Log-structured Data Stores. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys '15)*, pages 32:1–32:14, New York, NY, USA, 2015. ACM.

[58] Philipp Gschwandtner, Juan J. Durillo, and Thomas Fahringer. Multi-Objective Auto-Tuning with Insieme: Optimization and Trade-Off Analysis for Time, Energy and Resource Usage. In *Euro-Par 2014: Parallel Processing*, pages 87–98, Cham, 2014. Springer International Publishing.

[59] Frank T. Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform Storage Performance with 3D XPoint Technology. *Proceedings of the IEEE*, 105(9):1822–1833, Sep. 2017.

[60] Danny Harnik, Ronen Kat, Oded Margalit, Dmitry Sotnikov, and Avishay Traeger. To Zip or Not to Zip: Effective Resource Usage for Real-Time Compression. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST '13)*, San Jose, February 12-15 2013.

[61] Gary Hilson. Partnership Puts ReRAM in SSDs. https://www.eetimes.com/document.asp?doc_id=1332348, 2017.

[62] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. Closing the Performance Gap between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs. In *Proceedings of 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, pages 967–979, Boston, MA, 2018. USENIX Association.

[63] Mark J. Huiskes and Michael S. Lew. The MIR Flickr Retrieval Evaluation. In *Proceedings of the 2008 ACM International Conference on Multimedia Information Retrieval (MIR '08)*, Vancouver, Canada, October 30-31 2008.

[64] IBM. IBM Real-time Compression in IBM SAN Volume Controller and IBM Storwize V7000. http://www.redbooks.ibm.com/redpapers/pdfs/redp4859.pdf, 2015.

[65] IEEE. Avoid Infringement upon IEEE Copyright. https://journals.ieeeauthorcenter.ieee.org/choose-a-publishing-agreement/avoid-infringement-upon-ieee-copyright/.

[66] Intel. Intel Optane SSD. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/gaming-enthusiast-ssds/optane-900p-series.html, 2018.

[67] Intel. Intel SSDs. https://www.intel.com/content/www/us/en/support/articles/000006354/memory-and-storage.html, 2018.

[68] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding Up Distributed Request-response Workflows. In *Proceedings of the ACM Conference on SIGCOMM (SIGCOMM '13)*, pages 219–230, New York, NY, USA, 2013. ACM.

[69] Sampreeti Jena, Pandaba Patro, and Siddhartha Shankar Behera. Multi-Objective Optimization of Design Parameters of a Shell Tube Type Heat Exchanger Using Genetic Algorithm. *International Journal of Current Engineering and Technology*, 3(4):1379–1386, 2013.

[70] Yichen Jia, Eric Anger, and Feng Chen. When NVMe over Fabrics Meets Arm: Performance and Implications. In *Proceedings of 2019 IEEE 35th Symposium on Mass Storage Systems and Technologies (MSST '19)*, pages 134–140, May 2019. 10.1109/MSST.2019.000-9. © IEEE 2019. Reprinted with Permission.

[71] Yichen Jia, Zili Shao, and Feng Chen. SlimCache: Exploiting Data Compression Opportunities in Flash-Based Key-Value Caching. In *Proceedings of 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '18)*, pages 209–222, Sep. 2018. DOI:10.1109/MASCOTS.2018.00029. © IEEE 2018. Reprinted with Permission.

[72] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *Proceedings of 2005 USENIX Annual Technical Conference (USENIX ATC '05)*, Anaheim, CA, April 10-15 2005.

[73] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, pages 121–136, New York, NY, USA, 2017. ACM.

[74] Herbert Jordan, Peter Thoman, Juan J. Durillo, Simone Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. A Multi-objective Auto-tuning Framework for Parallel Codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*, pages 1–12, Nov 2012.

[75] Myoungsoo Jung and Mahmut Kandemir. An Evaluation of Different Page Allocation Strategies on High-Speed SSDs. In *Proceedings of the 4th USENIX conference on Hot Topics in Storage and File Systems (HotStorage '12)*, Boston, MA, June 2012.

[76] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, pages 993–1005, Boston, MA, 2018. USENIX Association.

[77] Kimberly Keeton, Dirk Beyer, Ernesto Brau, Arif Merchant, Cipriano Santos, and Alex Zhang. On the Road to Recovery: Restoring Data After Disasters. *SIGOPS Operating System Review*, 40(4):235–248, April 2006.

[78] I. Y. Kim and O. L. de Weck. Adaptive Weighted-sum Method for Bi-objective Optimization: Pareto Front Generation. *Structural and Multidisciplinary Optimization*, 29:149–158, 2005.

[79] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Selecta: Heterogeneous Cloud Storage Configuration for Data Analytics. In *Proceedings of 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, pages 759–773, Boston, MA, 2018. USENIX Association.

[80] Klaus Kofler, Juan J. Durillo, Philipp Gschwandtner, and Thomas Fahringer. A Region-Aware Multi-Objective Auto-Tuner for Parallel Programs. In *Proceedings of 2017 46th International Conference on Parallel Processing Workshops (ICPPW '17)*, pages 190–199, Aug 2017.

[81] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the Performance of Fast NVM Storage with uDepot. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19)*, pages 1–15, Boston, MA, February 2019. USENIX Association.

[82] Sanjeev R. Kulkarni. *Information, Entropy, and Coding*. 2002. Lecture Notes for ELE201 Introduction to Electrical Signals and Systems, Princeton University, 2002.

[83] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter Boncz, Thomas Neumann, and Alfons Kemper. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*, San Francisco, CA, June 26-July 01 2016.

[84] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs. In *Proceedings of 2019 USENIX Annual Technical Conference (USENIX ATC '19)*, pages 603–616, Renton, WA, July 2019. USENIX Association.

[85] Larry Leemis. Zipf. http://www.math.wm.edu/ leemis/chart/UDR/PDFs/Zipf.pdf, 2019.

[86] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, pages 137–152, New York, NY, USA, 2017. ACM.

[87] Ning Li, Hong Jiang, Dan Feng, and Zhan Shi. PSLO: Enforcing the Xth Percentile Latency and Throughput SLOs for Consolidated VM Storage. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*, pages 28:1–28:14, New York, NY, USA, 2016. ACM.

[88] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, Seongil O, Sukhan Lee, and Pradeep Dubey. Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-Value Store Server Platform. In *Proceedings of the 42nd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA '15)*, Portland, OR, June 13-17 2015.

[89] Yan Li, Kenneth Chang, Oceane Bel, Ethan L. Miller, and Darrell D. E. Long. CAPES: Unsupervised Storage Performance Tuning Using Neural Network-based Deep Reinforcement Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*, pages 42:1–42:14, New York, NY, USA, 2017. ACM.

[90] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang, and Guangzhong Sun. Metis: Robustly Tuning Tail Latencies of Cloud Systems. In *Proceedings of 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, pages 981–992, Boston, MA, 2018. USENIX Association.

[91] Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Towards Accurate and Fast Evaluation of Multi-Stage Log-structured Designs. In *Proceedings of 14th*

*USENIX Conference on File and Storage Technologies (FAST '16)*, pages 149–166, Santa Clara, CA, 2016. USENIX Association.

[92] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proceedings of the 23th Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, Oct 2011.

[93] Hyeontaek Lim, Dongsu Han, David G.Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI '14)*, Seattle, WA, April 2-4 2014.

[94] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *Proceeding of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*, Tel Aviv, Israel, June 23-27 2013.

[95] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the Role of Burst Buffers in Leadership-class Storage Systems. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST '12)*, pages 1–11, April 2012.

[96] Yang Liu, Raghul Gunasekaran, Xiaosong Ma, and Sudharshan S. Vazhkudai. Server-Side Log Data Analytics for I/O Workload Characterization and Coordination on Large Shared Storage Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*, pages 819–829, Nov 2016.

[97] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of 14th USENIX Conference on File and Storage Technologies (FAST '16)*, Santa Clara, Feb 2016.

[98] lz4. Extremely Fast Compression. http://lz4.github.io/lz4/.

[99] Ashraf Mahgoub, Paul Wood, Sachandhan Ganesh, Subrata Mitra, Wolfgang Gerlach, Travis Harrison, Folker Meyer, Ananth Grama, Saurabh Bagchi, and Somali Chaterji. Rafiki: A Middleware for Parameter Tuning of NoSQL Datastores for Dynamic Metagenomics Workloads. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware '17)*, pages 28–40, New York, NY, USA, 2017. ACM.

[100] Thanos Makatos, Yannis Klonatos, Manolis Marazakis, Michail D. Flouris, and Angelos Bilas. Using Transparent Compression to Improve SSD-based I/O Caches. In *Proceedings of the 5th European conference on Computer systems (EuroSys '10)*, Paris, France, April 13-16 2010.

[101] Yandong Mao, Eddie Kohler, and Robert Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*, Bern, Switzerland, April 10-13 2012.

[102] R. Timothy Marler and Jasbir S. Arora. The Weighted Sum Method for Multi-objective Optimization: New Insights. *Structural and Multidisciplinary Optimization*, 41:853–862, 2010.

[103] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store. In *Proceedings of 2015 USENIX Annual Technical Conference (USENIX ATC '15)*, pages 207–219, Santa Clara, CA, 2015. USENIX Association.

[104] Venkata Lakshmi Marripudi and P. Yakaiah. Image Compression based on Multilevel Thresholding Image Using Shannon Entropy for Enhanced Image. *Global Journal of Advanced Engineering Technologies*, 4:271–274, 2015.

[105] Nimrod Megiddo and Dharmendra Modha. ARC: A Self-tuning, Low Overhead Replacement Cache. In *Proceedings of the 2nd USENIX Conference on File and Storage (FAST '03)*, San Francisco, CA, February 12-15 2003.

[106] Michael Mesnier. Intel Open Storage Toolkit. `https://sourceforge.net/projects/intel-iscsi/`, 2016.

[107] Micron. Micron 3D XPoint. `https://www.micron.com/products/advanced-solutions/3d-xpoint-technology`.

[108] Ingo Muller, Cornelius Ratsch, and Franz Faerber. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *Proceedings of the 17th International Conference on Extending Database Technology (EDBT '14)*, Athens, Greece, March 24-28 2014.

[109] Dorom Nakar and Shlomo Weiss. Selective Main Memory Compression by Identifying Program Phase Changes. In *Proceedings of the 23rd IEEE Convention of Electrical and Electronics Engineers in Israel*, Tel-Aviv, Isreal, September 6-7 2004.

[110] A. J. Nebro, J. J. Durillo, J. Garcia-Nieto, C. A. Coello Coello, F. Luna, and E. Alba. SMPSO: A New PSO-based Metaheuristic for Multi-objective Optimization. In *2009 IEEE Symposium on Computational Intelligence in Multi-Criteria Decision-Making (MCDM '09)*, pages 66–73, March 2009.

[111] 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 *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, IL, April 2-5 2013.

[112] G. O. Odu and O. E. Charles-Owaba. Review of Multi-criteria Optimization Methods – Theory and Applications. *IOSR Journal of Engineering (IOSRJEN)*, 3:01–14, 2013.

[113] 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 *Proceedings of the 41st International Conference on Parallel Processing (ICPP '12)*, Pittsburgh, PA, September 10-13 2012.

[114] Thomas Papadakis. SkipList. `https://en.wikipedia.org/wiki/Skip_list`, 1993.

[115] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store. In *Proceedings of 2016 USENIX Annual Technical Conference (USENIX ATC '16)*, pages 537–550, Denver, CO, 2016. USENIX Association.

[116] Meikel Poess and Dmitry Potapov. Data Compression in Oracle. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, page 937–947. VLDB Endowment, 2003.

[117] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, pages 497–514, Oct 28-32 2017.

[118] Reddit. Reddit Comments. `https://www.reddit.com/r/datasets/comments/3bxlg7/i_have_every_publicly_available_reddit_comment/`, 2015.

[119] David Reinsel, John Gantz, and John Rydning. Data Age 2025. `https://assets.ey.com/content/dam/ey-sites/ey-com/en_gl/topics/workforce/Seagate-WP-DataAge2025-March-2017.pdf`.

[120] Sears Russell and Ramakrishnan Raghu. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*, pages 217–228, New York, NY, USA, 2012. ACM.

[121] Samsung. Ultra-Low Latency with Samsung Z-NAND SSD - Breakthrough Storage for a New Generation of Enterprise and Data Center Infrastructure. `https://www.samsung.com/semiconductor/global.semi.static/Brochure\_Samsung\_S-ZZD\_SZ985\_1804.pdf`, 2017.

[122] Samsung. Samsung Z-SSD SZ985 - Ultra-low Latency SSD for Enterprise and Data Centers - Brochure. `https://www.samsung.com/semiconductor/global.semi.static/Brochure_Samsung_S-ZZD\_SZ985\_1804.pdf`, 2018.

[123] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *Proceedings of 14th USENIX Conference on File and Storage Technologies (FAST '16)*, pages 67–80, Santa Clara, CA, 2016. USENIX Association.

[124] Bon-Keun Seo, Seungryoul Maeng, Joonwon Lee, and Euiseong Seo. DRACO: A Deduplicating FTL for Tangible Extra Capacity. *IEEE Computer Architecture Letters*, 14:123–126, July-December 2015.

[125] Dimitrios N. Serpanos and Wayne H. Wolf. Caching Web Objects Using Zipf's Law. *Proceedings of the SPIE*, 3527:320–326, October 1998.

[126] Matias Sessarego, K.R. Dixon, David E. Rival, and D.H. Wood. A Hybrid Multi-objective Evolutionary Algorithm for Wind-turbine Blade Optimization. *Engineering Optimization*, 47(8):1043–1062, 2014.

[127] Dipti Shankar, Xiaoyi Lu, Md Rahman, Nusrat Islam, and D.K. Panda. Benchmarking Key-value Stores on High-performance Storage and Interconnects for Web-scale Workloads. In *Proceedings of 2015 IEEE International Conference on Big Data (Big Data '15)*, pages 539–544, 10 2015.

[128] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27:379–423, 1948.

[129] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. DIDACache: A Deep Integration of Device and Application for Flash-based Key-value Caching. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, Santa Clara, CA, February 27-March 2 2017.

[130] Pradeep J. Shetty, Richard P. Spillane, Ravikant R. Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building Workload-Independent Storage with VT-Trees. In *Proceedings of 11th USENIX Conference on File and Storage Technologies (FAST '13)*, pages 17–30, San Jose, CA, 2013. USENIX.

[131] Richard P. Spillane, Pradeep J. Shetty, Erez Zadok, Sagar Dixit, and Shrikar Archak. An Efficient Multi-tier Tablet Server Storage Architecture. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC '11)*, pages 1:1–1:14, New York, NY, USA, 2011. ACM.

[132] John D. Strunk, Eno Thereska, Christos Faloutsos, and Gregory R. Ganger. Using Utility to Provision Storage Systems. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, pages 21:1–21:16, Berkeley, CA, USA, 2008. USENIX Association.

[133] Steve Swanson. Early Measurements of Intel's 3DXPoint Persistent Memory DIMMs. `https://www.sigarch.org/early-measurements-of-intels-3dxpoint-persistent-memory-dimms/`.

[134] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: Advanced Photo Caching on Flash for Facebook. In *Proceedings of 13th USENIX Conference on File and Storage Technologies (FAST '15)*, pages 373–386, Santa Clara, CA, February 2015. USENIX Association.

[135] ArangoDB Team. ArangoDB. https://www.arangodb.com/, 2014.

[136] Dgraph Team. DGraph. https://dgraph.io/, 2017.

[137] Facebook RocksDB Team. DBBench. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools, 2014.

[138] Facebook RocksDB Team. RocksDB. https://rocksdb.org/, 2014.

[139] Faktory Team. Faktory. https://contribsys.com/faktory/, 2012.

[140] Memcached Team. Memcached: A Distributed Memory Object Caching System. https://memcached.org/.

[141] MongoDB Team. MongoDB. https://www.mongodb.com/, 2007.

[142] MySQL Team. MySQL. https://www.mysql.com/, 2019.

[143] PingCap Team. TiDB. https://github.com/pingcap/tidb, 2018.

[144] Pinterest Team. Pinterest. https://www.pinterest.com/, 2010.

[145] Platypus Team. Platypus. https://github.com/Project-Platypus/Platypus, 2019.

[146] Redis Team. Redis. https://redis.io/.

[147] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Siyuan Ma, Yanfeng Zhang, and Xiaodong Zhang. LSbM-tree: Re-Enabling Buffer Caching in Data Management for Mixed Reads and Writes. In *Proceedings of 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS '17)*, 2017.

[148] Eno Thereska, Austin Donnelly, and Dushyanth Narayanan. Sierra: Practical Power-proportionality for Data Center Storage. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys '11)*, pages 169–182, New York, NY, USA, 2011. ACM.

[149] Evangelos Triantaphyllou, Bo Shu, Salvador Nieto Sánchez, and Thomas G. Ray. Multi-Criteria Decision Making: An Operations Research Approach. *Encyclopedia of Electrical and Electronics Engineering*, 15:175–186, 1998.

[150] Irina Chihaia Tuduce and Thomas Gross. Adaptive Main Memory Compression. In *Proceedings of 2005 USENIX Annual Technical Conference (USENIX ATC '05)*, pages 237–250, Anaheim, CA, April 10-15 2005.

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

[152] Twitter. Tweets2011. http://trec.nist.gov/data/tweets/, 2011.

[153] Marc-Andre Vef, Nafiseh Moti, Tim Suß, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and Andre Brinkmann. GekkoFS - A Temporary Distributed File System for HPC Applications. In *Proceedings of 2018 IEEE International Conference on Cluster Computing (CLUSTER '18)*, pages 319–324, Sep. 2018.

[154] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. LogBase: A Scalable Log-structured Database System in the Cloud. *Proceedings of the VLDB Endowment*, 5(10):1004–1015, June 2012.

[155] Dongshu Wang, Dapei Tan, and Lei Liu. Particle Swarm Optimization Algorithm: An Overview. *Soft Computing*, 22:387–408, January 2018.

[156] Kefei Wang and Feng Chen. Cascade Mapping: Optimizing Memory Efficiency for Flash-based Key-value Caching. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*, pages 464–476, New York, NY, USA, 2018. ACM.

[157] 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 *Proceedings of the 9th European Conference on Computer Systems (EuroSys '14)*, pages 16:1–16:14, New York, NY, USA, 2014. ACM.

[158] WIKI. Coefficient of Determination. `https://en.wikipedia.org/wiki/Coefficient_of_determination`.

[159] WIKI. Multi-objective Optimization. `https://en.wikipedia.org/wiki/Multi-objective_optimization`.

[160] Wikipedia. Entropy (Information Theory). `https://en.wikipedia.org/wiki/Entropy_(information_theory)`, 2019.

[161] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The Case for Compressed Caching in Virtual Memory Systems. In *Proceedings of the 1999 USENIX Annual Technical Conference (USENIX ATC '99)*, Monterey, CA, June 6-11 1999.

[162] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. Anna: A KVS for Any Scale. In *Proceedings of 2018 IEEE 34th International Conference on Data Engineering (ICDE '18)*, pages 401–412, April 2018.

[163] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *Proceedings of 2015 USENIX Annual Technical Conference (USENIX ATC '15)*, July 2015.

[164] Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel HHack, and Song Jiang. zExpander: a Key-Value Cache with Both High Performance and Fewer Misses. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys '16)*, London, UK, April 18-21 2016.

[165] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. CosTLO: Cost-effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI '15)*, pages 543–557, Berkeley, CA, USA, 2015. USENIX Association.

[166] Zuyang Wu. SSDB. `http://ssdb.io/`, 2013.

[167] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. Bluecache: A Scalable Distributed Flash-based Key-value Store. *Proceedings of the VLDB Endowment*, 10(4):301–312, November 2016.

[168] Zhixiang Yang, Xinye Cai, and Zhun Fan. Epsilon Constrained Method for Constrained Multiobjective Optimization Problems: Some Preliminary Results. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation (GECCO Comp '14)*, pages 1181–1186, New York, NY, USA, 2014. ACM.

[169] Heng Zhang, Mingkai Dong, and Haibo Chen. Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, Santa Clara, Feb 2016.

[170] Jiachen Zhang, Peng Li, Bo Liu, Trent G. Marbach, Xiaoguang Liu, and Gang Wang. Performance Analysis of 3D XPoint SSDs in Virtualized and Non-Virtualized Environments. In *Proceedings of 2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS '18)*, pages 1–10, Dec 2018.

[171] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamically Tracking Miss-Ratio-Curve for Memory Management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '04)*, Boston, MA, October 7-13 2004.

[172] George Kingsley Zipf. Relative Frequency as a Determinant of Phonetic Change. *Reprinted from the Harvard Studies in Classical Philology*, XL, 1929.

[173] Aviad Zuck, Sivan Toledo, Dmitry Sotnikov, and Danny Harnik. Compression and SSD: Where and How? In *Proceedings of the 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW '14)*, Broomfield, CO, Oct 2104.

# VITA

Yichen Jia was born in 1991, at Zoucheng City, Shandong Province, China. He received his bachelor of science degree in Mathematics from Jilin University, Changchun, China in June 2013. He then worked as a software engineer at Appsoft, Beijing, China for one year. In August 2014, he joined the Ph.D program in Division of Computer Science and Engineering in Louisiana State University, Baton Rouge, Louisiana. Over the summer of 2018 and 2019, he interned with Arm at Austin, Texas. He is currently a candidate for the degree of Doctor of Philosophy in Computer Science.