SUPPORTING EFFICIENT LARGE-SCALE KEY-VALUE SYSTEMS WITH AN OPTIMIZED STORAGE HIERARCHY

by

XINGBO WU

Presented to the Faculty of the Graduate School of The University of Texas at Arlington in Partial Fulfillment of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON
June 2018
SUPPORTING EFFICIENT LARGE-SCALE KEY-VALUE SYSTEMS WITH AN OPTIMIZED STORAGE HIERARCHY

The members of the Committee approve the doctoral dissertation of XINGBO WU

Song Jiang
Supervising Professor

Hong Jiang

Jia Rao

Jiang Ming

Dean of the Graduate School
To my family
2018. It’s been the 26th year of my student life and here at Arlington is the end of it. It’s the time for me to thank all the people who have companied, encouraged, guided, and helped me.

First I would like to thank Dr. Song Jiang, my PhD advisor. He guided me onto the road of doing high-quality research in the area of computer systems. He opened my eyes and really changed my life. When I first started my PhD study with Dr. Jiang, I didn’t have the confidence on finding a good idea to work on. Weeks of fruitless work made me really frustrated. He did encourage me to have patience and faith. Besides his efforts on helping me, what really made me dedicated was his own enthusiasm and passion. One day I was supposed to grade an exam. Unfortunately at the same day I happened to have a paper-submission deadline to catch up. When he got to know that issue he immediately took over the grading task which allowed me to focus on addressing the critical issues in the research. That day we both worked until 1AM and he didn’t even go home to have a good sleep. He just rested in his sleeping bag and continued his own work the next morning. What else can you expect from your advisor?

I would also like to thank the dissertation committee members, Dr. Hong Jiang, Dr. Jia Rao, and Dr. Jiang Ming. You offered your best help to me not only on my research as a PhD student, but also on getting me prepared for my future career as a professional researcher.

I’m fortunate to work with many researchers from academia and industry. I thank Zili Shao, Wenguang Wang, Li Zhang, Yufei Ren, Yandong Wang, Wei Zhang, and Michel Hack for their valuable input and kind help on many exciting research projects. These exciting research projects would not be possible without you.

I’m lucky to have many talented labmates who shared their best knowledge and experience with me. I thank Yuehai Xu, Jianqiang Ou, Kun Wang, Guoyao Xu, Yudi Wei, Fan Ni, Chunyi Liu, Xingsheng Zhao, Haitao Wang, Yong Zhao, Kun Suo, Mengxiao Wang, Zhichao Yan, and Xiaofeng Wu, for your company in my PhD life. I also thank my labmates Xianqin Chen, Chunnan Zhang, Hongguang Dong, Zaoyang Gong, Yingxia Li, Kai Huang, Jingwei Yang, Yuexiao Liu, Chengjian Wen, Yanyan Hu, Yang Yang, Siyao Zheng, Xinlei Zhou, Deping Dong, Fan Xia, Chun Zhao, Siyuan Wang, Jiangtao Wu, Jun He, Pingyang Guo, Hanqing Zhou, Long Chen, Wenzong Duan, Jiacheng Feng, Zhen Lin, Chao Song, Qian Cui, Jing Zhang, Junjie Ma, Jie Guo, Yan Zhai, Qiao Sun, Xiangnan Gu, Jianfei Wang, Han Wang, Tengfei Mu, Cong Wang, and Zhiyuan Cheng. You are my best mentors and friends, leading my way to many interesting topics since I was a undergraduate
student in Beihang. The enjoyable time with you well prepared me with a positive mind and professional skills to work on challenging tasks.

I’m very grateful to my friends. They are an integral part of my colorful student life. I thank Yuehua Wang, Pengfei Ren, Bo Peng, Yuanzhe Li, Chenguang Du, Jessica Diaz, Husheng Zhou, Ruohan Zhang, Xiaolong Xie, Jinqiu Yang, Changchang Liu, Wei-han Lee, Cheng Li, Mengting Yan, Yang Guo. Exchanging ideas with you made my PhD life a lot more interesting than I can ever imagine. I thank Yang Xu, Haixiao Xu, Huizhu Li, Hang Yin, Baoxin Zhou, Lu Liu, and Yi Shi. You’re always there where I need you, my best friends in my life.

I thank my high school math teacher Pu Lv, who offered enormous support on me in the darkest time of my life when I was struggling for the college entrance exam. He said “It is not unusual to be unfairly treated” (directly translated from Mandarin). With his words in my mind I stopped complaining (like the boy in “The Catcher in the Rye”) and became more confident to facing the challenges in the real world.

I thank my primary school math teacher Sufang Wan. She is the best teacher I have ever met. She is of great patience and inclusiveness for every student. Her classes fostered my great interests in mathematics and also everything about science. I was highly encouraged to discover the unknown throughout my student life. I was so lucky to have learned from her.

I thank my parents for their great love and support throughout all these years. Since I was a little kid, they try to save every penny they have earned in hope one day they can send me to college. They always work hard and their dedication influenced me. I learned by heart that one must earn his living by hard working. It’s a luxury to have great parents like them.

I thank my wife Chang. She has been always supporting me on everything. She sacrificed a lot for me and I’m so grateful to have her love. I dedicate this dissertation to my family: mother, father, Chang, and my daugther Yixuan.

June 4, 2018
ABSTRACT

SUPPORTING EFFICIENT LARGE-SCALE KEY-VALUE SYSTEMS WITH AN OPTIMIZED STORAGE HIERARCHY

XINGBO WU, Ph.D.
The University of Texas at Arlington, 2018

Supervising Professor: Song Jiang

Driven by the growing demands from big-data applications, the focus of their data management has been largely shifted from traditional SQL databases to NoSQL (Not-only-SQL) databases, such as key-value (KV) stores, which provides essential functionalities and much higher performance for storing and retrieving data. Correspondingly new hardware technologies have been developed to support the fast data accesses, such as NVMe SSDs and Infiniband network. However, existing designs of NoSQL databases usually see sub-optimal performance on fast hardware. Traditionally the computing overhead of a database system is overshadowed by the slow storage and network. With the adoption of the new hardware technologies the inefficiency at the software side is now the major source of bottlenecks in today’s systems.

In this dissertation we propose solutions to overcome barriers on the adoption of new technologies, such as large DRAMs, fast SSDs, and low-latency Infiniband network, into existing stacks of software systems. Accordingly introduce new designs, including Search Lookaside Buffer (SLB), zExpander, LSM-trie, and NVMcached, to improve the systems’ efficiency on accessing SSD, DRAM, non-volatile main memory, and CPU cache. We identify false temporal locality and false spatial locality in index search and propose SLB to effectively improve index search by removing the false localities. In zExpander we use compression to increase the effective capacity in KV caches without adding DRAM, which can substantially reduce misses in the KV cache. In LSM-trie we introduce a trie-based data structure to reduce the overhead of internal data reorganization by an order of magnitude for KV stores on SSDs. In NVMcached we remove expansive FLUSH operations for persistent and crash-consistent KV caches on byte-addressable NVM.
# TABLE OF CONTENTS

| ACKNOWLEDGEMENTS | .................................................. | v |
| ABSTRACT | .................................................. | vii |
| Chapter | Page |
| 1. INTRODUCTION | .................................................. | 1 |
| 1.1 Exploiting Access Locality in Index Data Structures | ........................................... | 2 |
| 1.2 Increasing Effective Size of In-memory KV Caches | ........................................... | 2 |
| 1.3 Improving Efficiency for Key-value Stores on SSDs | ........................................... | 3 |
| 1.4 Reducing Overhead for Crash-consistent KV Caches on NVMs | ........................................... | 4 |
| 1.5 Dissertation Organization | ........................................... | 5 |
| 2. SLB: EFFICIENT CACHING FOR INDEX DATA STRUCTURES | ........................................... | 6 |
| 2.1 Introduction | .................................................. | 6 |
| 2.2 Motivation | .................................................. | 9 |
| 2.2.1 False localities in B+-trees | ........................................... | 10 |
| 2.2.2 False localities in hash tables | ........................................... | 10 |
| 2.2.3 Search Lookaside Buffer: inspired by TLB | ........................................... | 10 |
| 2.3 Design of SLB | .................................................. | 11 |
| 2.3.1 API of SLB | .................................................. | 12 |
| 2.3.2 Data structure of the SLB cache | .................................................. | 14 |
| 2.3.3 Tracking access locality for cache replacement | .................................................. | 15 |
| 2.4 Evaluation | .................................................. | 17 |
| 2.4.1 Experimental setup | .................................................. | 18 |
| 2.4.2 Performance on index data structures | .................................................. | 19 |
| 2.4.3 Performance of KV applications | .................................................. | 23 |
| 2.4.4 Performance of networked KV applications | .................................................. | 26 |
| 2.4.5 Performance with real-world traces | .................................................. | 27 |
| 2.5 Related work | .................................................. | 29 |
| 2.5.1 Software approaches | .................................................. | 29 |
| 2.5.2 Hardware approaches | .................................................. | 30 |
| 2.5.3 Reducing index size | .................................................. | 31 |
| 2.6 Limitations | .................................................. | 31 |
| 2.7 Summary | .................................................. | 32 |
| 3. ZEXPANDER: INCREASING KV CACHE CAPACITY WITHOUT ADDING DRAM | .................................................. | 33 |
| 3.1 Introduction | .................................................. | 33 |
3.2 Motivation of Increasing Effective Cache Size and Using Batched Data Compression ........................................... 35
  3.2.1 Long-tail distribution and Impact of Larger Cache Size ....... 35
  3.2.2 Batched Data Compression ....................................... 38
3.3 Design of zExpander .................................................. 39
  3.3.1 Data Organization of the Z-zone ................................ 40
  3.3.2 Data Access and Replacement in the Z-zone ................... 42
  3.3.3 Limiting Accesses in the Z-zone ............................... 43
3.4 Evaluation ............................................................. 45
  3.4.1 Implementation of Two zExpander Prototypes ................. 46
  3.4.2 Experiment Setup .................................................. 46
  3.4.3 Results for memcached-based KV Caches ..................... 48
  3.4.4 Results for High-Performance KV Caches ..................... 50
  3.4.5 Effectiveness of Using Bloom Filter to Reduce Decompressions ...... 52
  3.4.6 Impact of Space Allocation between N-zone and Z-zone ...... 53
3.5 Related Work .......................................................... 55
  3.5.1 Compression in main memory ................................ 55
  3.5.2 Indexing data structure ........................................ 56
  3.5.3 Fragmentation in memory allocation .......................... 56
  3.5.4 KV store performance ........................................... 56
  3.5.5 Replacement strategy ............................................ 57
3.6 Summary .................................................................... 57
4. LSM-TRIE: AN ULTRA-LARGE KV STORE FOR SMALL DATA .... 58
  4.1 Introduction ................................................................ 58
  4.2 The design of LSM-trie .............................................. 60
    4.2.1 Write Amplification in LSM-tree ............................. 61
    4.2.2 Challenge on Reducing Write Amplification in the LSM-tree Compaction ......................................................... 62
    4.2.3 SSTable-trie: A Design for Minimizing Write Amplification ...... 63
    4.2.4 LSM-trie: a Large Store for Small Items ..................... 65
  4.3 Performance Evaluation .............................................. 72
    4.3.1 Experiment Setup .................................................. 72
    4.3.2 Experiment Results ................................................ 73
  4.4 Related Work .......................................................... 78
    4.4.1 Efforts on Supporting Efficient Writes ....................... 78
    4.4.2 Efforts on Supporting Efficient Reads ....................... 79
    4.4.3 Other Related Works ............................................. 79
  4.5 Summary ............................................................... 80
5. NVMcached: A KV Cache on Byte-addressable NVM

5.1 Introduction

5.2 Design of NVMcached

5.2.1 Storing Data Checksums to Avoid Ordered Writes

5.2.2 Managing Memory in Zones for its Efficient Allocation and Reclamation

5.2.3 Batched KV item replacement and space reclamation in Zones

5.2.4 Caching at DRAM

5.3 Evaluation

5.3.1 Experiment Setup

5.3.2 Evaluating the Impact of Crash-restart with Real Traces

5.3.3 Performance of NVMcached with Synthetic Workloads

5.3.4 Performance of NVMcached with Real-world Workloads

5.4 Related Work

5.4.1 Persistent Memory Allocator

5.4.2 Persistent Data Structures

5.4.3 Hardware Supports for NVM

5.5 Summary

6. Conclusions and Future Work

6.1 Contributions

6.2 Future Work

6.2.1 Redesigning stores and caches for big data

6.2.2 Supporting new applications with NoSQL systems

6.2.3 Leveraging Virtualization for high-performance big-data computing

References

Biographical Statement
CHAPTER 1
INTRODUCTION

We are in the era of big data. Large volumes of data are being generated, stored, and analyzed every second. This happens not only in data centers hosting thousands of servers, but also in everyone’s computers and smartphones. While the data volume is quickly growing, the tasks on organizing, retrieving, and processing the data become ever challenging. In the recent years the focus of data management has been largely shifted from traditional SQL databases to NoSQL (Not-only-SQL) databases, such as key-value (KV) stores and key-value caches, which provides essential functionalities and much higher performance for storing and retrieving data. Unfortunately, the design of these systems are often ineffective, sometimes incapable of tackling challenging tasks involving massive amount of data. Even running on servers with powerful CPUs, large DRAM, fast storages of large capacity, and low-latency network, the applications still have a hard time achieving their expected performance.

This dissertation investigates the root causes of the issues in KV systems and identifies opportunities to enable efficient data accessing for big-data applications. Here we use an example to demonstrate the overview of our efforts. In a typical setting in today’s data centers, databases or KV stores are usually deployed at the back-end to persistently store all the data in a well-organized way for future queries. In front of the persistent stores, KV caches are commonly used to accelerate the accesses to frequently used data by avoiding I/O and recomputation. In this scenario the access time of the result can be roughly determined by the following expression:

$$\text{AccessTime} = \text{HitTime} + \text{MissRatio} \times \text{MissPenalty}$$

To reduce access time, the effort can be made on reducing either hit time, miss ratio, or miss penalty. To achieve the goal of improving the efficiency of KV systems, we design and implement several new KV stores and caches, each improving one or multiple factors in the above expression. Our efforts include:

- Search Lookaside Buffer (SLB)—a index-lookup accelerator that maximizes the efficiency of CPU cache, which can reduce hit time [139].
- zExpander—a fast and space-efficient KV cache that reduces miss ratio [144].
- LSM-trie—a flash-based persistent KV store with much reduced I/O overhead, which primarily reduces the miss penalty [143].
NVMcached—a crash-consistent persistent KV cache for byte-addressable NVM that can reduce hit time [140].

In the following of this chapter we will introduce the motivation and the contributions of each work, then the outline the rest of this dissertation.

1.1 Exploiting Access Locality in Index Data Structures

More and more large-scale applications store their data sets in main memory to provide high-performance services. They rely on index data structures, such as B+-trees, to organize their data to facilitate quick search. Even if each data item requires only one memory access to read, the index traversal may add a number of memory accesses leading to significantly reduced performance. As those indexes are already structurally optimized, the conventional wisdom for improving performance is to keep them in the CPU cache and leverage access locality for efficient lookups. However, after taking a closer look at the indexes from the perspective of their actual use, we observed that a majority of their accesses can be considered of false locality. In an index each item is directly associated with a target entry, such as a leaf node in a B+-tree. The other entries in the index that facilitate the search are intermediate entries. Frequent accesses of hot target entries lead to frequent accesses to the intermediate entries, exhibiting false temporal locality which does not represent applications’ true access pattern about requested data. Further, memory data is accessed in the unit of cache lines while a search result can be (much) smaller than a cache line. When CPU fetches the target entries the unused data in the same cache line are read together into the cache, which leads to false spatial locality. Recognizing existence of the false localities we see a great potential for improvement of CPU cache efficiency.

To address this issue the most effective way is to identify data of true locality and organize them in a cache-friendly way. To this end we design Search Lookaside Buffer (SLB) [139], an index search accelerator for any user-defined index data structure. A key distinction of SLB from existing use of cache for the indexes is that SLB does not cache an index according to its memory access footprints. Instead, it identifies and caches search results embedded in the target entries. By keeping itself small and its contents truly hot, SLB can effectively improve cache utilization by eliminating both false temporal and spatial localities in an index search. Experiments with index-intensive applications show that SLB can significantly improve their performance. Even on a hash table which is already highly efficient, SLB can increase its lookup throughput by 73% with real-world workloads.

1.2 Increasing Effective Size of In-memory KV Caches

As an indispensable component of data center infrastructures, KV cache, such as memcached, provides fast access to data that can be slow to re-fetch or re-compute in
the back-end storage or database systems. However in a busy KV cache system even a small increase of miss ratio can lead to significant additional workloads on the slow database system. In other words, a small reduction of miss ratio can result in significant workload reduction on the database system, which is critical to KV cache systems. Because adding more and more DRAM to the server is not an economic option, a common solution is to use data compression to increase the effective cache size. However the common practice on using compression is very ineffective. On one hand, it is very expensive to apply compression or decompression on every KV access. On another hand, compression cannot effectively save space for small items that are mostly common in real-world workloads.

There seems to be a fundamental conflict between achieving high throughput and applying compression. However, the real motivation behind using compression is actually to save space for the entire KV cache, which does not necessarily mean to compress every single item. Accordingly we decoupled the efforts of providing high performance from that of reducing cache misses to achieve the goals of both high performance and low miss-ratio simultaneously. Observing that KV cache’s accesses are usually highly skewed, with a small fraction of KV items receiving a majority of accesses, We design zExpander, a KV cache that discriminately treats KV items according to their access frequency [144]. In zExpander the cache space is divided into two sub-caches. The first sub-cache hosts frequently accessed data which are not compressed. Because of highly skewed access pattern, this sub-cache can have a relatively small space to serve a majority of requests with high performance. The second sub-cache is expected to hold a much larger portion of the cache space but serve a smaller portion of requests for cold items. Therefore, with small impact on the KV cache’s performance the second sub-cache prioritizes its effort on memory efficiency to reduce misses. Experiments with real-world workloads show the effective cache size is sufficiently increased, which greatly reduces KV cache’s miss ratio by up to 46%.

1.3 Improving Efficiency for Key-value Stores on SSDs

KV stores play a critical role in the assurance of service quality and user experience for many highly-demanding data-intensive applications. KV store allows light-weight implementation and a much simplified interface. However, there are several trends on workload characteristics that are seriously challenging its design for high performance and scalability. First, very small KV items are widespread. For a store of a given capacity, smaller KV items demand more metadata for them to be located. Second, the demand on the capacity of the store on one server keeps increasing. However, this would significantly increase metadata size and make memory constrained. Third, many KV stores require high performance for both reads and writes. Google’s LevelDB, a state-of-the-art KV
store, is optimized for writes by organizing data in multiple levels. However, when some metadata cannot be held in memory, it requires multiple disk I/Os to serve a lookup request, degrading read performance. Even for writes we observed much degraded performance with a reasonably large store due to its inefficient internal data reorganization.

Our goal is to have a key-value storage system that can accommodate multi-billions of small items with a capacity of multiple terabytes at a single server with a limited memory demand. It should support a sustained high throughput even for workloads with little locality, a common scenario in a back-end storage system where front-end caching has been deployed. After studying many twists attempting to achieve a similar goal in the open-source community, We realized that there does not exist an incremental optimization as the solution. The state-of-the-art key-value stores rely on Log-structured merge-tree (LSM-tree) to organize their data. Internally LSM-tree uses background merge-sorts to reorganize its data. However, this internal operation causes high write-amplification which can be as high as 77, which means, for example, for every 1MB of data written by users, the store could generate additional 77MB of writes to the storage. To reduce the high write-amplification, we design a new key-value data structure from scratch, named LSM-trie [143]. LSM-trie uses the hashed keys, rather than the original keys to index the key-values. Leveraging a trie-like index, LSM-trie can reduce write amplification by an order of magnitude and leads to much reduced memory demand for index lookups.

1.4 Reducing Overhead for Crash-consistent KV Caches on NVMs

The emerging high-density byte-addressable NVMs enables an alternative to DRAM as main memory of better energy-efficiency, larger capacity, and most importantly, data persistency. KV caches have been playing a critical role in maintaining high service quality and improving user experience in many large-scale websites, such as Facebook and Amazon. By enabling data persistency, an NVM-based KV cache can retain its cached data and continue supplying them after a system restart without dramatically degrading service quality. To ensure KV items cached on NVM are still usable after a system crash, writing of the KV items must be crash-consistent. Existing solutions to maintain crash consistency include logging, copy-on-write, and checkpointing. However we found that for byte-addressable NVMs these solutions share a performance issue—they all rely on CPU’s expensive cache flush instructions to promptly persist data and enforce write order. Frequent use of flushes can be too expensive for high-performance KV caches on NVMs, which not only compromises NVM’s limited write endurance but also degrades the KV cache’s throughput.

NVMcached [140] leverages a unique property of KV caches to eliminate all flush operations. Unlike KV stores, it is acceptable for a KV cache to drop some of its cached
items, because anyway KV items can be replaced out of the cache without notifying the users. For correctness we only need to make sure that no wrong values are returned for subsequent lookups. To this end, NVMcached does not use any flushes for insertion requests. Instead, it stores checksum of data along with its corresponding metadata so that the data integrity can be verified after a crash. The NVM space is organized in a way that cache replacement and garbage collection (GC) are integrated to enable low-cost GC and batched replacement. NVMcached improves system performance by up to $2.85 \times$ for real-world workloads, while preventing miss-ratio spikes after a system crash and restart.

1.5 Dissertation Organization

The remainder of this dissertation is organized as follows. In Chapter 2 we introduce the concept of false temporal/spatial locality, and how can index lookups be improved by removing the false localities with Search Lookaside Buffer. In Chapter 3 we present zEx-pander, a KV cache of both high-performance and much reduced miss-ratio. In Chapter 4 we present how LSM-trie reduces KV store’s write-amplication by an order of magnitude. In Chapter 5 we design NVMcached to enable low-cost crash-consistency for KV caches on byte-addressable NVMs. In Chapter 6 we summarize our contributions and discuss the future work.
CHAPTER 2
SLB: EFFICIENT CACHING FOR INDEX DATA STRUCTURES

With the ever increasing DRAM capacity in commodity computers, applications tend to store large amount of data in main memory for fast access. Accordingly, efficient traversal of index structures to locate requested data becomes crucial to their performance. The index data structures grow so large that only a fraction of them can be cached in the CPU cache. The CPU cache can leverage access locality to keep the most frequently used part of an index in it for fast access. However, the traversal on the index to a target data during a search for a data item can result in significant false temporal and spatial localities, which make CPU cache space substantially underutilized. In this chapter we show that even for highly skewed accesses the index traversal incurs excessive cache misses leading to suboptimal data access performance. To address the issue, we introduce Search Lookaside Buffer (SLB) to selectively cache only the search results, instead of the index itself. SLB can be easily integrated with any index data structure to increase utilization of the limited CPU cache resource and improve throughput of search requests on a large data set. We integrate SLB with various index data structures and applications. Experiments show that SLB can improve throughput of the index data structures by up to an order of magnitude. Experiments with real-world key-value traces also show up to 73% throughput improvement on a hash table.

2.1 Introduction

In-memory computing has become popular and important due to applications’ demands on high performance and availability of increasingly large memory. More and more large-scale applications store their data sets in main memory to provide high-performance services, including in-memory databases (e.g., H-Store [81], MemSQL [102], and SQLite [124]), in-memory NoSQL stores and caches (e.g., Redis [118], MongoDB [105], and Memcached [57]), and large forwarding and routing tables used in software-defined and content-centric networks [154, 51, 22]. In the meantime, these applications rely on index data structures, such as hash table and B+-tree, to organize data items according to their keys and to facilitate search of requested items. Because the index always has to be traversed to locate a requested data item in a data set, the efficiency of the index traversal is critical. Even if the data item is small and requires only one memory access, the index traversal may add a number of memory accesses leading to significantly reduced performance. For example, a recent
study on modern in-memory databases shows that “hash index (i.e., hash table) accesses are the most significant single source of runtime overhead, constituting 14–94% of total query execution time.” [83]. A conventional wisdom to addressing the issue is to keep the index in the CPU cache to minimize index search time.

However, it is a challenge for the caching approach to be effective on reduction of index access time. The memory demand of an index (indices) can be very large. As reported, “running TPC-C on H-Store, a state-of-the-art in-memory DBMS, the index consumes around 55% of the total memory.” [151]. The study on Facebook’s use of Memcached with their five workloads finds that Memcached’s hash table, including the pointers on the linked lists for resolving hash collision and the pointers for tracking access locality for LRU replacement, accounts for about 20–40% of the memory space [20]. With a main memory of 128 GB or even larger holding a big data set, the applications’ index size can be tens of gigabytes. While a CPU cache is of only tens of megabytes, search for a data item with a particular key in the index would incur a number of cache misses and DRAM accesses, unless there is strong locality in the index access and the locality can be well exploited.

Indeed, requests for data items usually exhibit strong locality. For example, as reported in the Facebook’s Memcached workload study, “All workloads exhibit the expected long-tail distributions, with a small percentage of keys appearing in most of the requests...”. For one particular workload (ETC), 50% of the keys occur in only 1% of all requests [20]. Such locality is also found in the workloads of database [83] and network forwarding table [154]. Each of the requested data items is usually associated with an entry in the index data structure. The entry corresponds to the same key as the one in the request. Example index data structures include hash table and $B^+$-tree. The requested data item can be either directly included in the entry, such as a switch port number in a router’s forwarding table [154], or pointed to by a pointer in the entry, such as user-account status information indexed by the hash table in Facebook’s Memcached system [20]. In both cases, to access the requested data, one must search the index with a given key to reach the index entry, named target entry. The goal of the search is to obtain the search result in the target entry. The result can be the requested data item itself or a pointer pointing to the data item. Strong access locality of requested data is translated to strong locality in the access of corresponding target entries. However, this locality is compromised when it is exploited in the current practice of index caching for accelerating the search.

First, the temporal locality is compromised with index search. To reach a target index entry, one has to walk on the index and visit intermediate entries. For a hot (or frequently accessed) target entry, the intermediate entries on the path leading to it also become hot from the perspective of the CPU cache. This is illustrated in Figures 2.1 and 2.2 for the hash table and $B^+$-tree, respectively. However, access locality exhibited on the intermediate entries is artificial and does not represent applications’ true access pattern about requested
Figure 2.1: False temporal locality in a hash table. False temporal locality is generated on a path to a target entry in the hash table.

Figure 2.2: False temporal locality in a B\(^+\)-tree. False temporal locality is generated on a path to a target entry in a B\(^+\)-tree.

data. Accordingly, we name this locality \textit{false temporal locality}. Such locality can increase demand on cache space by many times leading to high cache miss ratio.

Second, CPU accesses memory and manages its cache space in the unit of cache lines (usually of 64 bytes). The search result in a target entry can be much smaller than a cache line (e.g., a 8-byte pointer vs. 64-byte cache line). In an index search spatial locality is often weak or even does not exist, especially when keys are hashed to determine their positions in the index. Because CPU cache space must be managed in the unit of cache line, (probably cold) index entries in the same cache line as those on the path to a target entry can be fetched into the cache as if there were spatial locality. We name the locality \textit{false spatial locality}, as illustrated in Figure 2.3 for the hash table. This false locality unnecessarily inflates cache demand, pollutes the cache, and reduces cache hit ratio.

To remove the aforementioned false localities and improve efficiency of limited CPU cache space, we introduce an index caching scheme, named \textit{Search Lookaside Buffer (SLB)}, to accelerate search on any user-defined in-memory index data structure. A key
The main contributions of this work are as follows:

- We identify the issue of false temporal and false spatial localities, in the use of major index data structures, responsible for degradation of index search performance for significant in-memory applications.
- We design and implement the SLB scheme, that can substantially increase cache hit ratio and improve search performance by removing the false localities.
- We conduct extensive experiments to evaluate SLB with popular index data structures, in-memory key-value applications, and a networked key-value store on a high-performance Infiniband network. We also show its performance impact using real-world key-value traces from Facebook.

### 2.2 Motivation

This work was motivated by observation of false temporal and spatial localities in major index data structures and their performance implication on important in-memory applications. In this section we will describe the localities and their performance impact in two representative data structures, $B^+$-trees and hash tables, followed with discussions on similar issues with process page table and on how the solution of SLB was inspired by an important invention in computer architecture—the TLB table.

Figure 2.3: False spatial locality in a hash table. False spatial locality is generated in the cache lines containing intermediate entries and target entry on a path in the hash table.

The distinction of SLB from existing use of cache for the indices is that SLB does not cache an index according to its memory access footprint. Instead, it identifies and caches search results embedded in the target entries. By keeping itself small and its contents truly hot, SLB can effectively improve cache utilization. SLB eliminates both false temporal and spatial localities in the index searches, and enables search at the cache speed.
2.2.1 False localities in B+-trees

B+-tree [21] and many of its variants have been widely used on managing large ordered indices in databases [72, 125] and file systems [120, 29].

In B+-tree each lookup needs to traverse the tree starting from the root to a leaf node with a key (see Figure 2.2). With a high fan-out, the selection of a child node leads to multiple cache misses in a single node. For example, a 4-KB node contains 64 cache lines, and requires roughly six (log₂ 64) cache-line accesses in the binary search. One lookup operation on a B+-tree of four levels could require 24 cache-line accesses. These cache lines are at least as frequently accessed as the target entry’s cache line. For a target entry in the working set, all these cache lines will also be included in the working set. However, if one can directly reach the target entry without accessing these cache lines, the search can be completed by only one cache line access with the false temporal locality removed.

2.2.2 False localities in hash tables

A commonly used hash table is to use chaining for resolving collision. A hash directory consists of an array of pointers, each representing a hash bucket pointing to a linked list to store items with the same hash value. With a target search entry on one of the lists, the aforementioned false temporal locality exists. A longer list is more likely to have substantial false temporal locality.

In addition to the false temporal locality, the hash table also exhibits false spatial locality. To reach a target entry in a bucket, a search has to walk over one or more nodes on the list. Each node, containing a pointer and possibly a key, is substantially smaller than a 64 B cache line. Alongside the nodes, the cache lines also hold cold data that is less likely to be frequently accessed. However, this false spatial locality issue cannot be addressed by increasing the directory size and shortening the list lengths. A larger directory would lead to even weaker spatial locality for access of pointers in it. For every 8 B pointer in a 64 B cache line, 87.5% of the cache space is wasted.

Some hash tables, such as Cuckoo hashing [110] and Hopscotch hashing [66], use open addressing, rather than linked lists, to resolve collision for a predictable worst-case performance. However, they share the issue of false spatial locality with the chaining-based hash tables. In addition, open-addressing hashing usually still needs to make multiple probes to locate a key, which leads to false temporal locality.

2.2.3 Search Lookaside Buffer: inspired by TLB

The issues challenging effective use of CPU cache for fast search on indices well resemble those found in the use of page table for virtual address translation. First, as each process in the system has its own page table, total size of the tables can be substantial and
it is unlikely to keep them all in the CPU cache. Second, the tables are frequently searched. For every memory-access instruction the table must be consulted to look up the physical address with a virtual address as the key. Third, the tree-structured table consists of multiple levels leading to serious false temporal locality. Fourth, though spatial locality often exists at the leaf level of the tables, such locality is less likely for intermediate entries. If the page tables were cached as regular in-memory data in the CPU cache, the demand on cache space would be significantly higher and the tables’ cache hit ratio would be much lower. The consequence would be a much slower system.

Our solution is inspired by the one used for addressing the issue of caching page tables, which is Translation Lookaside Buffer (TLB), a specialized hardware cache [130]. In TLB, only page-table search results—recently accessed Page Table Entries (PTEs) at the leaf level—are cached. With a TLB as large as only a few hundreds of entries, it can achieve a high hit ratio, such as a few misses per one million instructions [97] or less than 0.5% of execution time spent on handling TLB misses [23].

It is indisputable that use of TLB, rather than treating page tables as regular data structure and caching them in the regular CPU cache, is an indispensable technique. “Because of their tremendous performance impact, TLBs in a real sense make virtual memory possible” [19]. Index search shares most issues that had challenged use of page tables decades ago. Unfortunately, the success of TLB design has not influenced the design on general-purpose indices. An anecdotal evidence is that to allow hash indices associated with database tables to be cache-resident, nowadays one may have to take a table partitioning phase to manually reduce index size [98].

While SLB intends to accommodate arbitrary user-defined indices and search algorithms on them, which can be of high variation and irregularity, it is not a good choice to dedicate a hardware cache separate from regular CPU cache and to apply customized management with hardware support for SLB. Instead, SLB takes an approach different from TLB. It sets up a buffer in the memory holding only hot target entries. SLB intends to keep itself sufficiently small and its contents truly hot so that its contents can be all cached in the CPU cache. It aims to keep search requests from reaching the indices, so that the indices can be much less accessed and less likely to pollute the CPU cache.

2.3 Design of SLB

SLB is designed for applications where index search is a performance bottleneck. While numerous studies have addressed the issues with specific index data structures and search algorithms to ameliorate this bottleneck, the SLB solution is intended to serve any data structures and algorithms for accelerating the search. This objective makes the solution have the risk of being over-complicated and entangled with designs of various
data structures and algorithms. If this were the case, SLB would not have a clean interface
to the users’ programs. Fortunately, SLB is designed as a look-aside buffer and works in-
dependently of index data structures and their search algorithms. With limited interactions
through the SLB API, the programs are only required to emit search results to SLB and
delegate management of the search results to SLB.

To efficiently address the issue of false localities, the design of SLB will achieve the
following goals:

- SLB ensures the correctness of operations on the original index data structure, espe-
cially for sophisticated concurrent data structures.
- SLB is able to identify hot target entries in the index data structure and efficiently
  adapt to changing workload patterns with minimal cost.
- SLB is able to be easily integrated into programs using any index data structures by
  exposing a clean and general interface.

2.3.1 API of SLB

SLB’s API is shown in Figure 2.4. Its functions are used to support accessing of
the SLB cache, maintaining consistency for its cached data, and currency control for its
accesses.

2.3.1.1 Accessing the SLB cache

SLB is implemented as a library of a small set of functions that are called to accel-
erate key search in various index data structures. SLB is a cache for key-value (KV) items.
While conceptually the KV items in the cache are a subset of those in the index, SLB uses
its own key and value representations that are independent from those used in the index
data structure defined and maintained by user code. The format of user-defined keys and
values can be different in different user codes. For example, a key can either be a NULL-
terminated string or a byte array whose size is specified by an integer. A value can be either
stored next to its key in the target entry or linked to by a pointer next to the key.

Rather than duplicating the real key-value data in its cache, SLB stores a pointer to
the target entry for each cached key-value item. In addition, a fixed-size tag—the hash value
of the original key—is stored together with the pointer for quick lookup (see Section 2.3.2). In
this way the format of SLB cache is consistent across different indices and applications.
It is up to the user code to supply untyped pointers to the target entries in the user-defined
index, and to supply functions to extract or to compute hash tags from user-supplied keys
(keyhash()) and cached target entries (entryhash()) for SLB to use. While the
formats of keys and target entries are unknown to the SLB cache, SLB also needs a user-
supplied function (match()) to verify whether a key matches a target entry. All the
three functions are specified when an SLB cache is initialized with the `SLB_create()` function.

After an SLB’s initialization, the cache can be accessed with two functions. `SLB_emit()` emits a target entry successfully found in an index search to the SLB cache. Note that SLB will decide whether an emitted item will be inserted into the cache according to knowledge it maintains about the current cache use. The user simply calls `SLB_emit()` for every successful lookup on the index.

With SLB, a search in the index should be preceded by a lookup in the SLB cache through calling `SLB_get()` . If there is a hit, the search result is returned and a search on the actual index can be bypassed.

### 2.3.1.2 Maintaining consistency

To prevent SLB from returning stale data, user code needs to help maintain consistency between the index and the SLB cache. For this purpose, user code should call `SLB_invalidate()` when a user request removes an item from the index, or call `SLB_update()` when an item is modified. `SLB_update()` should also be called if a target
entry is relocated in the memory due to internal reorganization of the index, such as garbage collection.

As user code does not know whether an item is currently cached by SLB, it has to call `SLB_invalidate()` or `SLB_update()` functions for every item invalidation or updating, respectively. This is not a performance concern, as the invalidation or update operations on the index are expensive by themselves and execution of the function calls usually requires access only to one cache line. The performance impact is still relatively small even when the items are not in the SLB cache.

### 2.3.1.3 Managing concurrency

Applications usually distribute their workloads across multiple CPU cores for high performance. They often use concurrency control, such as locking, to allow a shared data structure to be concurrently accessed by multiple cores. Similarly, locking is used in SLB to manage concurrent accesses to its data structures. For this purpose, SLB provides two functions, `SLB_lock()` and `SLB_unlock()`, for user programs to inform SLB cache whether a lock on a particular key should be applied.

To prevent locking from being a performance bottleneck, SLB uses the lock striping technique to reduce lock contention [65, 63]. We divide the keys into a number of partitions and apply locking on each partition. By default there are 1024 partitions, each protected by a spinlock. SLB uses a 10-bit hash value of the key to select a partition.

A spinlock can be as small as only one byte. False sharing between locks could compromise the scalability of locking on multi-core systems. To address the issue, each spinlock is padded with unused bytes to exclusively occupy an entire cache line. Our use of stripped spinlocks can sustain a throughput of over 300 million lock-unlocks per second on a 16-core CPU, which is sufficient for SLB to deliver high throughput in a concurrent execution environment.

To avoid deadlocks between SLB and the index data structure, the user’s code should always acquire an SLB’s lock before acquiring any lock(s) for its own index. SLB’s lock should be released only after all modifications to the index have been finalized and the locks on the index are released.

### 2.3.2 Data structure of the SLB cache

The SLB cache is to facilitate fast reach to requested target entries with high time and space efficiency. For this reason, the cache has to be kept small to allow its content to stay in the CPU cache as much as possible, so that target entries can be reached with (almost) zero memory access. However, the target entries can be of different sizes in different indices.
and can be quite large. Therefore, we cannot store target entries directly in the SLB cache. Instead, we store pointers to them.

Specifically, search results emitted into the SLB cache are stored in a hash table named \textit{Cache Table}. To locate an item in a Cache Table, a 64-bit hash value is first obtained by calling the user-supplied functions (\texttt{keyhash}() or \texttt{entryhash}()) to select a hash bucket. As shown in Figure 2.5, each bucket occupies a cache line and the number of buckets is determined by the size of the SLB cache. Within each bucket there are seven pointers, each pointing to a target entry. As on most 64-bit CPU architectures no more than 48 bits are used for memory addressing, we use only 48 bits (6B) to store a pointer.

To minimize the cost for lookup of the requested target entry in a bucket, we use the higher 16 bits of the 64-bit hash value as a tag and store it with its corresponding pointer. On lookup, any target entry whose tag matches the requested key’s tag will be selected and then a full comparison between the keys is performed using the user-supplied \texttt{match}() function. If there is a match the value in the target entry is returned to complete the search.

### 2.3.3 Tracking access locality for cache replacement

As the SLB cache has limited space, a decision has to be made on what items can be admitted and what items can stay in the cache based on their recent access locality, or their \textit{temperatures}. Only comparatively hot items should be admitted or be kept in the cache. To this end, SLB needs to track temperatures for cached items and (uncached) target entries that can potentially be emitted to SLB. However, conventional approaches for tracking access locality are too expensive for SLB. For example, the list-based replacement schemes, such as LRU, require two pointers for each element, which would triple the size of Cache Table by storing three pointers for each item. Low cost replacement algorithm, such as CLOCK [45], uses only one bit per item. However it still requires global scanning to identify cold items. We develop a highly efficient locality tracking method that can effectively identify relatively hot items for caching in SLB.
2.3.3.1 Tracking access history of cached items

As shown in Figure 2.5, SLB’s Cache Table has a structure similar to that of hardware-based CPU cache, which partitions cache entries into sets and identifies them with their tags. Similarly, SLB’s replacement is localized within each hash bucket of a cache line size. A bucket contains seven 1-byte counters, each associated with a \{tag, pointer\} pair in the bucket (see Figure 2.5). Upon a hit on an item, its corresponding counter is incremented by one. However overflow can happen with such a small counter. To address this issue, when a counter to be incremented already reaches its maximum value (255) we randomly select another non-zero counter from the same bucket and decrement its value by one. In this way, relative temperatures of cached items in a bucket can be approximately maintained without any access outside of this bucket. To make room for a newly admitted item in a bucket, SLB selects an item of the smallest counter value for replacement.

2.3.3.2 Tracking access history of target entries

When a target entry is emitted to the SLB cache, SLB cannot simply admit it by evicting a currently cached item unless the new item is sufficiently hot. For this purpose, SLB also needs to keep tracking their accesses, or emissions made by the user code. However, this can be challenging. First, tracking the access history may require extra metadata attached to each item in the index. Example of such metadata include the two pointers in LRU and the extra bit in CLOCK. Unfortunately this option is undesirable for SLB as it requires intrusive modification to the user’s index data structure, making it error-prone. Second, tracking temperature of cold entries can introduce expensive writes to random memory locations. For example, each LRU update requires six pointer changes, which is too expensive with accesses of many cold entries.

To know whether a newly emitted item is hot, we use an approximate logging scheme to track its access history in a hash table, named Log Table and illustrated in Figure 2.6. In this hash table, each bucket is also of 64 byte, the size of a cache line. In each bucket
there can be up to 15 log entries, forming a circular log. When an item is emitted to SLB, SLB computes a 4-byte hash tag from the key and appends it to the circular log in the corresponding bucket, where the item at the log head is discarded if the log has been full. The newly admitted item is considered to be sufficiently hot and eligible for caching in the Cache Table if the number of a key’s hash tag in the log exceeds a threshold (three). In this history tracking scheme, different target entries may produce the same hash tag recorded in a log, which inflates the tag’s occurrence. However, with 4-byte tag and a large number of buckets this inflation is less likely to take place. Even if it does happen, the impact is negligible.

2.3.3.3 Reducing cost of accessing the Log Table

For a more accurate history tracking in the Log Table, we usually use a large table (by default four times the size of the Cache Table) and do not expect many of its buckets stay in the CPU cache. With expected heavy cache misses for the logging operations in the table, we need to significantly reduce the operations on it. To this end, SLB randomly samples emitted items and logs only a fraction of them (5% by default) into the Log Table. This throttled history tracking is efficient and its impact on tracking accuracy is small. If the SLB cache has a consistently high or low hit ratios, the replacement would have less potential to further improve or reduce the performance, respectively. As a result, history tracking is not performance-critical and can be throttled. When the workload changes its access pattern, the changes will still be reflected in the logs even with the use of throttling (though it will take a longer time). With a workload mostly running at its steady phases, this does not pose a problem. As throttling may cause new items to enter the Cache Table at a lower rate, SLB disables throttling when the table is not full yet to allow the SLB cache to be quickly warmed up.

2.4 Evaluation

We have implemented SLB as a C library and integrated it with a number of representative index data structures and memory-intensive applications. We conducted extensive experiments to evaluate it. In the evaluation, we attempt to answer a few questions:

- How does SLB improve search performance on various data structures?
- Does SLB have good scalability on a multi-core system?
- How much can SLB improve performance of network-based applications?
- How does SLB perform with real-world workloads?
Table 2.1: Hardware parameters

<table>
<thead>
<tr>
<th>Machine Model</th>
<th>Dell PowerEdge R730</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Version</td>
<td>Intel Xeon E5-2683 v4</td>
</tr>
<tr>
<td>Number of sockets</td>
<td>2</td>
</tr>
<tr>
<td>Cores per socket</td>
<td>16</td>
</tr>
<tr>
<td>L1 Cache (per core)</td>
<td>64 KB</td>
</tr>
<tr>
<td>L2 Cache (per core)</td>
<td>256 KB</td>
</tr>
<tr>
<td>L3 Cache (per socket)</td>
<td>40 MB</td>
</tr>
<tr>
<td>DRAM Capacity</td>
<td>256 GB (16 × 16 GB)</td>
</tr>
<tr>
<td>DRAM Model</td>
<td>DDR4-2133 ECC Registered</td>
</tr>
<tr>
<td>Infiniband Network</td>
<td>Mellanox ConnectX-4 (100 Gb/s)</td>
</tr>
</tbody>
</table>

Table 2.2: SLB parameters

<table>
<thead>
<tr>
<th>Cache Table size</th>
<th>16 MB</th>
<th>32 MB</th>
<th>64 MB</th>
</tr>
</thead>
<tbody>
<tr>
<td># target entries</td>
<td>1835008</td>
<td>3670016</td>
<td>7340032</td>
</tr>
<tr>
<td>Log Table size</td>
<td>64 MB</td>
<td>128 MB</td>
<td>256 MB</td>
</tr>
<tr>
<td># hash tags</td>
<td>15728640</td>
<td>31457280</td>
<td>62914560</td>
</tr>
<tr>
<td>Total Size</td>
<td>80 MB</td>
<td>160 MB</td>
<td>320 MB</td>
</tr>
</tbody>
</table>

2.4.1 Experimental setup

In the evaluation we use two servers. Hardware parameters of the servers are listed in Table 2.1. Hyper-threading feature in CPU is turned off in BIOS to obtain more consistent performance measurements. To minimize the interference of caching and locking between the CPU sockets, we use a single CPU socket (16 cores) to run the experiments unless otherwise noted.

The servers run a 64-bit Linux 4.8.13. To reduce the interference of TLB misses, we use Huge Pages [69] (2 MB or 1 GB pages) for large memory allocations. xxHash hash algorithm [146] is used in SLB.

We evaluate SLB with four commonly used index data structures (Skip List, B+-tree, chaining hash table, and Cuckoo hash table), and two high-performance key-value applications (LMDB [95] and MICA [93]). As it is very slow to fill up a large DRAM with small KV items, we use a data set of about 9 GB (including metadata and data) for all the experiments unless otherwise noted. We also evaluate SLB by replaying real-world key-value traces from Facebook [20], and by running SLB-enabled MICA on high-performance Infiniband network.
2.4.2 Performance on index data structures

In this experiment we first fill one of the data structures (Skip List, B+-tree, chaining hash table, and Cuckoo hash table) with 100 million key-value items, each with an 8 B key and a 64 B value. Then we issue GET requests to the index using 16 worker threads, each exclusively bound to a CPU core. The workload is pre-generated in memory following the Zipfian distribution with a skewness of 0.99. For each data structure, we vary size of SLB’s Cache Table from 16 MB, 32 MB, to 64 MB. We configure size of the Log Table to be 4 of the Cache Table’s size. SLB’s configurations are listed in Table 2.2. We vary the data set, or the key range used in the Zipfian generator, from 0.1 million to 100 million keys.

2.4.2.1 B+-tree and Skip List

Figure 2.7 shows GET throughput of the two ordered data structures: B+-tree and Skip List. As shown, SLB dramatically improves the throughput of searching on the two indices by as much as 22 times. Due to existence of significant false localities in the index search, even for a small data set of less than 10 MB, the actual working set observed by the CPU cache can be much larger than the CPU’s 40 MB cache, leading to intensive misses. In addition, search on the two indices requires frequent pointer dereferences and key comparisons, consuming many CPU cycles even for items that are already in the CPU cache. Consequently the two data structures exhibit consistently low throughput when SLB is not used.

When the data set grows larger, throughput with SLB reduces but remains at least more than 2× of the throughput with SLB disabled. A larger SLB cache helps to remove
false localities for more target entries. This explains the fact that the throughput of the 64 MB SLB is higher than that with a smaller SLB cache on a relatively large data set (≥ 100 MB). However, the performance trend reverses for a smaller data set, where a smaller SLB cache produces higher throughput. With a small data set on the index search and a relatively large SLB cache, the cache may store many cold items that fill the SLB’s cache space but produce a smaller number of hits. The relatively cold items in the SLB cache can still cause false spatial locality for a larger SLB cache. Though SLB’s performance advantage is not sensitive to the SLB cache size, it is ideal to match the cache size to the actual working set size to receive optimal performance.

2.4.2.2 Hash tables

Figure 2.8 shows the throughput improvement of SLB with two hash tables. Without using SLB, Cuckoo hash table has lower throughput than the chaining hash table with smaller data sets. On the Cuckoo hash table each lookup accesses about 1.5 buckets on average. In contrast, we configure the chaining hash table to aggressively expand its directory so that the chain on each hash bucket has only one entry on average. For this reason the Cuckoo hash table has more significant false localities that can be removed by the SLB cache.

For the chaining hash table, the improvement mostly comes from its elimination of false spatial locality. Figure 2.8b shows that the chaining hash table has very high throughput with small data sets that can be all held in the CPU cache. Once the data set grows larger, the throughput drops quickly because of false spatial locality. This is the time...
when SLB kicks in and improves its throughput by up to 28% for medium-size data sets of 20 MB to 1 GB. When the data set becomes very large, the improvement diminishes. This is because in the Zipfian workloads with large data sets, the access locality becomes weak, and hot entries in the tables are less distinct from cold ones. SLB becomes less effective as it relies on the locality to improve CPU cache utilization with a small Cache Table.

SLB only makes moderate improvements for chaining hash table because we choose the most favorable configuration for chaining hash table. Aggressively expanding the hash directory can maximize its performance but also consume excessive amount of memory. With a conservative configuration SLB can help to maintain a high throughput by removing more false localities.

To further evaluate SLB with even larger data sets, we increase the total number of KV items in the table to 1 billion, which consumes about 90 GB of memory. We rerun the experiments on the large tables. As shown in Figure 2.9, with a larger table the overall throughput of all test cases reduces. This is mainly because the random access over a larger index leads to increased TLB misses. Even so, the relative improvement made by the SLB cache mostly remains.

### 2.4.2.3 Scalability

To evaluate the scalability of SLB, we change the number of worker threads from 1 to 16 and rerun the experiments using the chaining hash table. As shown in Figure 2.10a, SLB exhibits strong scalability. Doubling the number of working threads leads to almost doubled throughput. With the increase of data set size the throughput ratio between 16 threads and
1 thread increases from 11.5 to 13.8, because a larger data set has more balanced accesses across the hash table, which reduces contention.

To evaluate SLB in a multi-socket system, we run the experiment by using equal number of cores from each of the two sockets. The two curves on the top of Figure 2.10b show the throughput of using 32 cores with SLB enabled/disabled. With both sockets being fully loaded, SLB can still improve the throughput by up to 34%.

We observe that the throughput with 32 cores is only 17% to 36% higher than that with 16 cores on one socket. When using 16 cores, the throughput with 8 cores on each of the two sockets is 30% lower than that with all 16 cores on a single socket. The impact of using two or more sockets in an index data structure is twofold. On one hand the increased cache size allows more metadata and data to be cached. On the other hand, maintaining cache coherence between different sockets is more expensive. Excessive locking and data sharing in a concurrent hash table can offset the benefit of increased cache size. As a result, localizing the accesses to a single socket is more cost-effective for a high-performance concurrent data structure.

### 2.4.2.4 Performance with mixed GET/SET

While SLB delivers impressive performance benefit with workloads of GET requests, SET requests can pose a challenge. Serving SET requests requires invalidation or update operations to maintain consistency between the SLB cache and the index. To reveal how SLB performs with mixed GET/SET operations, we change the workload to include a mix of GET/SET requests. On the hash table a SET operation is much more expensive.
than a GET. As shown in Figure 2.11, when SLB is not used, with a small percentage of SET requests (5%) the throughput is 31% lower than that of GET-only workload (see Figure 2.8b). It further decreases to less than 55 MOPS (million operations per second) with 50% SET in the workload, or another 41% decrease. When SLB is used, with 5% SET performance advantage of SLB remains (compare Figures 2.8b and 2.11a). However, with 50% SET, the benefit of SLB diminishes as expected.

### 2.4.3 Performance of KV applications

To understand SLB’s performance characteristics in real-world applications, we run the experiments with two high-performance key-value stores, LMDB [95] and MICA [93].

#### 2.4.3.1 LMDB

LMDB is a copy-on-write transactional persistent key-value store based on B\(^+\)-tree. LMDB uses mmap() system call to map data files onto the main memory for direct access. In a warmed-up LMDB all requests can be served from memory without any I/O operations. In total 124 lines of code are added to LMDB to enable SLB. We use the same workload consisting of GET requests described in Section 2.4.2.

Figure 2.12a shows the throughput of LMDB. With larger data sets LMDB’s throughput is similar to that of B\(^+\)-tree (See Figure 2.7a), because it uses B\(^+\)-tree as its core index structure. However, for small data sets, throughput with SLB-enabled LMDB is lower than that with B\(^+\)-tree. In addition to index search, LMDB has more overhead on version control
and transaction support. For a small data set whose working set can almost entirely be held in the CPU cache by using SLB, LMDB spent a substantial amount of CPU cycles on the extra operations. Its peak throughput is capped at 139 MOPS, about 27% reduction over the 190 MOPS peak throughput received for B\(^+\)-tree with SLB.

### 2.4.3.2 MICA in the CREW mode

MICA is a chaining-hash-table-based key-value store that uses bulk-chaining to reduce pointer chasing during its search [93]. In the hash table each bucket is a linked list, in which each node contains seven pointers that fills an entire cache line. It also leverages load-
balancing and offloading features provided by advanced NICs to achieve high throughput over high performance network [103]. In this experiment we first remove the networking component from MICA to evaluate SLB’s impact on MICA’s core index data structure.

MICA by default allows concurrent reads and exclusive writes (CREW) to the table. MICA uses a versioning mechanism to eliminate locking for concurrent read operations. In the meantime, writers still need to use locks to maintain consistency of the store. The implication of employing lockless concurrency model for reads is that MICA’s hash table cannot be resized when it grows. With a fixed hash table size, the average length of the chains at each bucket will increase linearly with the number of stored key-value items. Consequently the long chains can lead to significant false temporal locality. To shorten the long chains one might propose to allocate a very large number of buckets when the table is created. However, this may cause the items to be highly scattered in the memory, leading to false spatial locality even for a very small data set. This drawback makes MICA’s performance highly sensitive to the number of key-value items in the table. In the experiments we set up three MICA tables with different number of buckets ($2^{22}$, $2^{23}$, or $2^{24}$). Accordingly the average length of the chains in the three tables are 4, 2, and 1, respectively.

Figures 2.12b, 2.12c, and 2.12d show throughput of the three MICA configurations. MICA’s throughput is higher with more buckets and thus shorter chains that help to reduce false temporal locality. In the meantime, SLB still improves their throughput by up to 56% even for the table whose average chain length is one (see Figure 2.12d). The reason is that the versioning mechanism in MICA requires two synchronous memory reads of a bucket’s version number for each GET request. Synchronous reads can be much slower than regular memory reads even if the version number is already in the CPU cache.
2.4.3.3 MICA in the EREW mode

To further reduce the interference between CPU cores, MICA supports exclusive-read-exclusive-write (EREW) mode, in which the hash table is partitioned into a number of sub-tables, each exclusively runs on a core. As there is no concurrent access to each sub-table, all costly protections for concurrency can be safely removed. We experiment on this mode where the SLB cache is also partitioned and its locks are also removed.

Figure 2.13 shows the throughput of MICA in the EREW mode with 16 partitions. The peak throughput of MICA with SLB can reach 281 MOPS, a 40% increase over its non-partitioned counterpart. For MICA of $2^{24}$ buckets, which has no false temporal locality, SLB can still improve its throughput by up to 95% (see Figure 2.13b) by removing the false spatial locality. This improvement suggests removing locking in the management of the SLB cache can further its performance advantage.

2.4.4 Performance of networked KV applications

While today’s off-the-shelf networking devices can support very high bandwidth, SLB’s performance advantage on reducing CPU cache misses becomes relevant for networked applications. For example, using three 200Gb/s Infiniband links [71] (24 GB/s × 3) can reach a throughput equal to the bandwidth of CPU’s memory controller (76.8GB/s) [76]. With the ever increasing network performance, the performance of networked in-memory applications will become more sensitive to the caching efficiency. To reveal the implication of SLB on a real networked application, we port MICA of its CREW mode to Infiniband using IB_SEND/IB_RECV verbs API. We use a 100Gb/s (about 12GB/s) Infiniband link between two servers. We send GET requests in batches (2048 requests in each batch) to minimize the CPU cost on the networking operations.

Figures 2.14a and 2.14b show the throughput of MICA on the network. Compared to that without networking, the throughput of all configurations decreases and is capped at about 125 MOPS, as the network bandwidth becomes the bottleneck. For 64-byte values, each GET response contains 92 bytes including the value and associated metadata, and the 125 MOPS peak throughput of MICA with LSB is equivalent to 10.7 GB/s, about 90% of the network’s peak throughput.

Attempting to reach the highest possible performance of the networked application, we minimize the network traffic by replacing each key-value item in the responses with a 1-byte boolean value indicating whether a value is found for the GET request. This essentially turns the GET request into a PROBE request. Figures 2.14c and 2.14d show the throughput for the PROBE requests on MICA with two different numbers of buckets. As the network bottleneck has been further reduced, the peak throughput recovers back to about 200 MOPS, almost the same as that of MICA without networking (see Figure 2.12d).
In the meantime, most requests can be quickly served from cache and CPU is less involved in networking. However, the throughput drops quicker than that without networking. This is due to intensive DRAM accesses imposed by the Infiniband NIC which interfere with the DRAM accesses from the CPU.

2.4.5 Performance with real-world traces

To study SLB’s impact on real-world workloads, we replay five key-value traces that were collected on Facebook’s production Memcached system [20] on an SLB-enabled chaining hash table. The five traces are USR, APP, ETC, VAR, and SYS, whose characteristics have been extensively reported and studied [20]. As the concurrency information is
not available in the traces, we assign requests to each of the 16 worker threads in a round-
robin fashion to concurrently serve the requests. We use first 20% of each trace to warm
up the system and divide the remaining of the trace into seven segments to measure each
segment’s throughput. The hash table sizes after the warm-up phase are listed in Table 2.3.

Figure 2.15 shows throughput of the traces in each of their segments. The results
are quite different across the traces. USR is a GET-dominant workload (\text{GET} \geq 99\%). It
exhibits the least skewness compared with other traces—about 20% keys contribute to 85%
of accesses. Although this is still a skewed workload, its working set can be much larger
than CPU’s cache size. As a result, SLB is hard to reduce its cache miss ratio. Accordingly
SLB can hardly improve its throughput.

APP and ETC have much more skewed accesses than USR. In APP, 10% of the keys
contribute to over 95% of the accesses. In ETC, 5% of the keys contribute to over 98% of
the accesses. However, there two traces include about 10%–30% DELETE operations in their segments, which subsequently increases the miss ratio in the SLB cache. Misses in the SLB cache leads to slow index searches, which cannot be removed by SLB. For these two traces SLB increases the throughput by up to 20%.

VAR and SYS mainly comprise GET and UPDATE operations. They have high skewness and relatively small working sets that can be identified by the SLB cache and kept in the CPU cache. As a result, SLB improves their peak throughput by up to 73% and 50%, respectively.

The experiments with Facebook traces show that the effectiveness of SLB mainly depends on the skewness of the workloads and the size of the hot data set, rather than the total size of the index.

2.5 Related work

With intensive use of indices in in-memory computing, studies on optimizing their data structures and operations are extensive, including improvements of index performance with software and hardware approaches, and reduction of index size for higher memory efficiency.

2.5.1 Software approaches

There are numerous data structures developed to organize indices, such as regular hash table using buckets (or linked lists) for collision resolution, Google’s sparse and dense hash maps [62], Cuckoo hashing [110], Hopscotch hashing [66], variants of B-tree [24], as well as Bitmap Index [35] and Columnar Index [86].

To speed up index search, one may reduce hops of pointer chasing in the index, such as reducing bucket size in hash tables or number of levels of trees. However, the approach usually comes with compromises. For example, Cuckoo hashing uses open addressing to guarantee that a lookup can be finished with at most two bucket accesses. However, Cuckoo hashing may significantly increase insertion cost by requiring possibly a large number of relocations or kickouts [129].

A tree-based index, such as $B^+$-tree, may reduce the depth of a tree and therefore the number of hops to reach a leaf node by employing a high fanout. However, wider nodes spanning a number of cache lines would induce additional cache misses. Masstree employs a prefix-tree to partition the key-values into multiple $B^+$-trees according to their key-prefixes [99]. This can reduce the cost of key comparisons with long keys. However, $B^+$-tree is still used in each partition to sort the key-values and the false localities in the index cannot be removed. Complementary to the techniques used by Masstree, SLB identifies the hot items in an index to further reduce the overhead on accessing them.
Specifically in the database domain efforts have been made on software optimizations for specific operations on indices, such as those on hash join algorithms to reduce cache miss rates [82, 98] and to reduce miss penalty by inserting prefetch instructions in the hash join operation [36]. These efforts demand extensive expertise on algorithms for executing corresponding database queries and their effectiveness is often limited on certain index organizations [64]. In contrast, SLB is a general-purpose solution that requires little understanding on the index structures and algorithms on them.

2.5.2 Hardware approaches

Other researches propose to accelerate index search with hardware-based supports. These can be either designing new specialized hardware components [83, 64, 100], or leveraging newly available hardware features [152, 149, 39, 61]. Finding that “hash index lookups to be the largest single contributor to the overall execution time” for data analytics workloads running contemporary in-memory databases, Kocberber et al. proposed Widx, an on-chip accelerator for database hash index lookups [83]. By building specialized units on the CPU chip, this approach incurs higher cost and longer design turn-around time than SLB. In addition, to use Widx programmers must disclose how keys are hashed into hash buckets and how to walk on the node list. This increases programmers’ burden and is in a sharp contrast with SLB, which does not require any knowledge on how the search is actually conducted.

To take advantage of capability of supporting high parallelism, researchers proposed to offload index-related operations to off-CPU processing units, such as moving hash-joins to network processors [61] or to FPGAs [39], or moving index search for in-memory key-value stores to GPUs [152]. Recognizing high cache miss ratio and high miss penalty in the operations, these works exploit high execution parallelism to reduce the impact of cache misses. As an example, in the Mega-KV work, the authors found that index operations take about 50% to 75% of total processing time in the key-value workloads [152]. With two CPUs and two GPUs, Mega-KV can process more than 160 million key-value requests per second. However, to achieve such a high throughput, it has to process the requests in large batches (10,000 requests per batch). Furthermore, the latency of each request is significantly compromised because of batching. Its minimal latency is 317 microseconds in Mega-KV, much higher than that in a CPU-based store—only 6-27 microseconds over an RDMA network [135]. For workloads with high access locality, SLB can make most requests serviced within the CPU cache. In this way, SLB is expected to achieve both high throughput and low latency without requiring specialized hardware support.
2.5.3 Reducing index size

Large-scale data management applications are often challenged with excessively large indices that consume too much memory. Major efforts have been made on reducing index sizes for database systems and key-value stores. Finding that indices consume about 55% of the main memory in a state-of-the-art in-memory database (H-Store), researchers have proposed dual-stage architectures to achieve both high performance and high memory efficiency [151]. It sets up a front store to absorb hot writes. However, it does not help with read performance. To improve Memcached’s hit ratio, zExpander maintains a faster front store and a compact and compressed backend store [144]. However, access of compressed data will use CPU cycles and may pollute the cache. In contrast, SLB reduces CPU cache miss ratio by improving caching efficiency with removed false localities.

A fundamental premise of these works is the access skew typically found in database and key-value workloads. In the workloads, there is a clear distinction of hot and cold data items and the corresponding locality is relatively stable [49, 127, 20]. This property has been extensively exploited to manage buffer for disks [54, 126, 77], to compress cold data in in-memory databases [58], and to construct and manage indices or data items in a multi-stage structures [151, 144, 79]. As use of any caches does, SLB relies on existence of temporal access locality in its workloads to be effective. Fortunately, existing studies on workload characterization and practices on leveraging the locality all suggest that such locality is widely and commonly available.

2.6 Limitations

Search Lookaside Buffer improves index lookup efficiency by removing the false temporal locality and false spatial locality in the process of index traversal and exploiting true access locality. For an application that uses index data structures, there are several factors that may impact the overall benefit of using the SLB cache. Here we list three possible scenarios where SLB produces only limited improvements on applications’ performance.

- For index data structures that have been highly optimized, such as some hash table implementations, there are not substantial false localities. As a result, there is limited space for SLB to improve the lookup efficiency.
- SLB’s effectiveness depends on skewness of workload access pattern. For workloads with weak locality, SLB has less opportunity to improve the cache miss ratio.
- When indices are used to access large data items, only a fraction of data access time is spent on index lookup. The program’s performance improvement due to the use of SLB can be limited even when the index lookup time is significantly reduced.
2.7 Summary

In this work we describe Search Lookaside Buffer (SLB), a software cache that can accelerate search on user-defined in-memory index data structures by effectively improving hardware cache utilization. SLB uses a cost-effective locality tracking scheme to identify hot items on the index and caches them in a small SLB cache to remove false temporal and false spatial localities from index searches. Extensive experiments show that SLB can significantly improve search efficiency on commonly used index data structures, in-memory key-value applications, and a high performance key-value store using 100 Gb/s Infiniband. Experiments with real-world Facebook key-value traces show up to 73% throughput increase with SLB on a hash table.
CHAPTER 3
ZEXPANDER: INCREASING KV CACHE CAPACITY WITHOUT ADDING DRAM

While key-value (KV) cache, such as memcached, dedicates a large volume of expensive memory to holding performance-critical data, it is important to improve memory efficiency, or to reduce cache miss ratio without adding more memory. As we find that optimizing replacement algorithms is of limited effect for this purpose, a promising approach is to use a compact data organization and data compression to increase effective cache size. However, this approach has the risk of degrading the cache’s performance due to additional computation cost. A common perception is that a high-performance KV cache is not compatible with use of data compacting techniques.

In this chapter, we show that, by leveraging highly skewed data access pattern common in real-world KV cache workloads, we can both reduce miss ratio through improved memory efficiency and maintain high performance for a KV cache. Specifically, we design and implement a KV cache system, named zExpander, which dynamically partitions the cache into two sub-caches. One serves frequently accessed data for high performance, and the other compacts data and metadata for high memory efficiency to reduce misses. Experiments show that zExpander can increase memcached’s effective cache size by up to 2× and reduce miss ratio by up to 46%. When integrated with a cache of a higher performance, its advantages remain. For example, with 24 threads on a YCSB workload zExpander can achieve throughput of 32 million RPS with 36% of its cache misses removed.

3.1 Introduction

As an indispensable component of data center infrastructures, key-value cache, such as memcached [57], provides fast access to data that can be slow to re-fetch or re-compute in the back-end storage or database systems. Presented in the form of key-value pairs, the data is cached in the memory of a cluster of servers, and is accessed with a simple interface, such as GET(key) for reading data, SET(key,value) for writing data, and DEL(key) for removing data. With its rapidly increasing importance on entire storage and/or database systems’ service quality, the cache’s performance and its improvement have received extensive studies. These optimization efforts include reducing network stack
cost, alleviating contention with concurrent accesses, reducing memory accesses, and optimizing memory management [91, 93, 145, 56, 121]. Some of the efforts leverage new hardware features, including direct cache access [70, 91] and RDMA [104, 53, 80, 135], or specialized hardware, such as GPU and FPGA [152, 67, 28], to speed up KV caches.

While a KV cache’s performance, either in request latency or in throughput, is important, the significance of reducing its misses cannot be downplayed. In many usage scenarios, each miss represents a request to a back-end database system. In a busy KV cache system these misses, which may represent a small miss ratio, can impose significant workload on the database system. As reported in a study on Facebook’s memcached workloads, eviction misses produced by one server can turn into over 120 million requests sent to the database per day, though they represent only a few percentage points of the server’s miss ratio [145]. In other words, seemingly minor reduction of miss ratio can lead to significant workload reduction on the database system. As the miss penalty can be as high as a few milliseconds or even seconds [109], miss-ratio reduction is also important to reducing effective request response time. Significant efforts are demanded to reduce it.

Without adding DRAM to the server, there are two approaches to reduce misses, and unfortunately both approaches could potentially compromise the cache’s performance. One approach is to apply advanced cache replacement algorithms. Over years a large number of replacement algorithms have been proposed, including LRU and advanced algorithms aiming to improve it, such as LIRS [78], ARC [101], and MQ [156]. However, in KV cache systems, only LRU or its approximations are adopted for their low cost. As examples, memcached uses LRU, and MemC3 chooses a lower-cost LRU-approximation scheme (similar to CLOCK) for higher space efficiency and better concurrency support. The advanced algorithms expected to produce lower miss ratios are often (much) more expensive, requiring more space to track access history for data that have been evicted. As we will show in Section 3.2, their limited improvements on miss ratio are less interesting when the cost is considered.

Another approach is to use data compression technique to improve memory efficiency. It increases effective cache size and reduces miss ratio. However, adoption of this technique would require compression for SET requests and decompression for GET requests. For a KV cache system designed for high performance this additional cost seems to be unbearable. However, in the next section, we will show that KV cache’s accesses are highly skewed and a majority of its GET requests are for a relatively small portion of the entire data set. This access pattern allows us to decouple the effort on providing high performance from that on reducing cache misses, and to achieve both goals (high performance and low miss ratio) simultaneously. The method is to partition the cache into two sub-caches, each managed by a different scheme and dedicated to achieving one of the goals. The first sub-cache is named N-zone, whose data are those frequently accessed
and will not be compressed. Because of highly skewed access pattern, N-zone can have a relatively small space to serve a majority of requests with high performance. This sub-cache can be managed by a state-of-the-art KV cache scheme designed for high performance. The second sub-cache is named Z-zone, which is expected to hold a (much) larger portion of the cache space but serve a smaller portion of requests. Therefore, with small impact on the KV cache’s performance Z-zone prioritizes its effort on memory efficiency to reduce misses.

We propose zExpander, a KV cache system applying this method in the management of a KV cache. In the following sections we will show that increasing cache size is an effective way to reduce misses. In the meantime, there are several challenges to address in the design of zExpander:

- First, in real-world KV workloads [145, 107, 4], most values in KV items are very small. Individually compressing them cannot produce substantial space saving.
- Second, metadata for indexing KV items can consume a substantial portion of memory when the items are small. A compact data organization is required to reduce the cost.
- Third, the space allocation between N-zone and Z-zone has to be dynamically adjusted to ensure the majority of accesses can be processed in N-zone to maintain performance and to ensure that Z-zone’s space is effectively utilized to reduce misses.

3.2 Motivation of Increasing Effective Cache Size and Using Batched Data Compression

In this section we use workloads collected at Facebook’s production memcached system and synthetic workload of representative access pattern to show that (1) accesses to KV cache are highly skewed, (2) increasing cache size is a necessary and effective means to reduce misses, and (3) batched data compression can achieve a much higher compression ratio than individually compressing KV items, facilitating the effort of increasing cache size.

3.2.1 Long-tail distribution and Impact of Larger Cache Size

To understand access pattern of representative KV cache workloads, we select three Facebook’s memcached traces (USR, APP, and ETC) [20]. We also use Yahoo’s YCSB benchmark suite [43] to generate a trace to access KV items. The trace covers a data set

---

1In the paper for characterizing Facebook’s memcached workload [20], there are five traces collected and analyzed. We choose three of them for this investigation. Trace VAR is not selected because it is write-dominated and reads only a few distinct keys. Trace SYS has a very small data set, and a cache of a few Gigabytes can produce almost a 100% hit ratio.
Figure 3.1: CDF curves for percentage of accesses associated with certain percentage of the most frequently accessed KV items.

of about 128 GB and the popularity of keys follows a Zipfian distribution with a skewness parameter of 0.99, which is YCSB’s default Zipfian parameter. The Zipfian distribution has been widely assumed on the KV cache’s workloads in prior works on KV caches [94, 91, 93, 56, 53].

Figure 3.1 plots the access CDF (Cumulative Distribution Function) curves of the four workloads, or the percentage of accesses associated with most frequently accessed KV items. As shown, all workloads exhibit distributions with long tails, and a relatively small cache can cover a majority of accesses. For ETC, APP, USR, and YCSB workloads, the 3.6%, 6.9%, 17.0%, and 5.9% most frequently accessed items receive 80% of total accesses, respectively. This observation is echoed by miss ratio curves with different replacement algorithms, including LRU, LIRS, and ARC, shown in Figure 3.2. This suggests that a relatively small cache holding these hot items can achieve low miss ratios.\footnote{A \texttt{SET} request is always considered as a hit on the corresponding KV item in the calculation of the miss ratio.} Let us define such a small cache that can accommodate the set of most frequently accessed KV items serving 80% of a workload’s total accesses (see Figure 3.1) as the workload’s base cache, and its size as the workload’s base cache size. As long as the base cache can
efficiently serve its requests, the cache’s performance is mostly warranted. However, this cannot lead to the conclusion that a small cache is sufficient.

Though miss ratios with a cache larger than a workload’s base cache are low, a busy KV cache can still produce a very large number of misses, or equivalently heavy load on the database system. A small improvement on miss ratio can make a substantial difference on the system’s performance. It is necessary to make major efforts to keep removing the misses, including using optimized replacement algorithms and/or increasing cache size. To estimate the contribution of locality-aware replacement algorithms on miss ratio improvement, we apply a hypothetical replacement algorithm, named LRU-X, in which the base cache uses LRU, and data out of the base cache but still in the memory are managed by the random replacement policy.

To clearly observe impacts of replacement algorithm and cache size on miss ratio, we list miss count with base cache size and the LRU-X replacement, which is considered as a reference value, as well as percentage of the misses that are removed with use of larger cache and/or optimized replacement algorithms in Table 3.1. We have several interesting observations. First, with a cache larger, or even several times larger, than the base cache, increasing cache size can still substantially reduce misses. For example, for every 50% increase of the base cache size, the miss count is reduced by 8% to 31% for ETC, and
Table 3.1: Reference miss counts with base cache size and LRU-X replacement (such as 968 million of misses for ETC at 9.5 GB (×1) and LRU-X), and percentages of the misses that are removed with use of larger cache (such as 1× or 2× of base cache size) and/or other replacement algorithms (LRU, LIRS, and ARC).

<table>
<thead>
<tr>
<th>Trace</th>
<th>Base Size</th>
<th>Algo.</th>
<th>Cache Size (×Base Size)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>×1.0</td>
</tr>
<tr>
<td>ETC</td>
<td>9.5 (GB)</td>
<td>LRU-X</td>
<td>968M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LRU</td>
<td>-0.00%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LIRS</td>
<td>-2.55%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ARC</td>
<td>-8.50%</td>
</tr>
<tr>
<td>APP</td>
<td>12.3 (GB)</td>
<td>LRU-X</td>
<td>2,273M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LRU</td>
<td>-0.00%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LIRS</td>
<td>-16.70%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ARC</td>
<td>-7.27%</td>
</tr>
<tr>
<td>USR</td>
<td>9.2 (GB)</td>
<td>LRU-X</td>
<td>17,417M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LRU</td>
<td>-0.00%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LIRS</td>
<td>-15.83%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ARC</td>
<td>-10.81%</td>
</tr>
<tr>
<td>YCSB</td>
<td>7.7 (GB)</td>
<td>LRU-X</td>
<td>4,905M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LRU</td>
<td>-0.00%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LIRS</td>
<td>-15.51%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ARC</td>
<td>-18.29%</td>
</tr>
</tbody>
</table>

by 30% to 38% for USR. Second, the locality-aware replacement algorithms moderately perform better than LRU-X, a scheme that simply selects random items in the long tail for replacement. This seems to suggest that for accesses in the long tail of very weak locality the room for further improvement by more carefully exploiting locality is limited. In addition, advanced algorithms, such as LIRS and ARC, need to spend substantial cache space to track accesses of KV items that have been evicted out of the cache, which essentially reduces effective cache size and offsets their advantages on miss ratio. Third, the benefit from increasing cache size is consistent across various replacement algorithms and workloads. In particular, this benefit does not disappear or even become smaller with advanced replacement algorithms producing lower miss ratio. While data compression can increase effective cache size, this observation suggests that it is a potentially effective technique for substantially reducing misses.

### 3.2.2 Batched Data Compression

A convenient approach to increasing KV-cache’s effective size is to individually compress KV items in the cache. This approach can be effective for items with large values.
<table>
<thead>
<tr>
<th>Container Size:</th>
<th>Individual</th>
<th>256 B</th>
<th>512 B</th>
<th>1 KB</th>
<th>2 KB</th>
<th>4 KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tweets</td>
<td>0.99</td>
<td>1.10</td>
<td>1.21</td>
<td>1.30</td>
<td>1.34</td>
<td>1.41</td>
</tr>
<tr>
<td>Places</td>
<td>1.28</td>
<td>1.28</td>
<td>1.45</td>
<td>1.60</td>
<td>1.70</td>
<td>1.77</td>
</tr>
</tbody>
</table>

Table 3.2: Average compression ratios with compression applied on containers of different sizes. “Individual” is for compression of individual values of KV items. “Tweets” and “Places” represent the data sets for Twitter’s tweets and location records, respectively. The ratio is between sizes of an uncompressed data object and its corresponding compressed object.

However, small values are common in KV cache workloads. As reported in the study on Facebook’s memcached workloads [20], except for one workload (ETC) 90% of cache space is occupied by values of under 500 B. In ETC, requests with values smaller than 16 bytes account for 40% of the total requests. Another workload USR virtually has only one value size, which is 2 bytes. In another example, a study on Twitter’s workloads finds that average value size of tweets is only 92 B [38].

To understand how compression ratio is affected by value size, we place values into containers of various sizes and apply LZ4 [7], a high-speed compression algorithm, on the containers. We test two value sets, one is a collection of about 10 million tweets that have been collected in Twitter’s service from September 2009 to January 2010 to study geolocation of twittering [38]. Average value size of the tweets is 92 B. Another value set contains records in a format named Places defined by Twitter to describe geographic locations with coordinates. We fill fields of the records with random data, and then encode the records using Google’s Protocol Buffers, which is a widely used method for serializing structured data [11]. The average record size is 100.9 B.

Table 3.2 shows average compression ratios of the two data sets when each data items are compressed individually or collectively in a container of various sizes. It can be seen that compression with a larger set of data is more effective on reducing data size. The larger the container is, the better compression ratio is. For tweets, individual compression does not result in any reduction of their sizes. With such an observation, users do not have the incentive to compress their data beforehand. Therefore, we increase effective cache size by compacting small items into containers for batched compression.

### 3.3 Design of zExpander

zExpander is a KV cache designed for high performance and reduced misses, each objective is achieved by a dedicated cache partition (named as N-zone and Z-zone, respectively). While N-zone can be managed by any state-of-the-art high-performance KV cache scheme, we need to design a scheme for managing Z-zone for high memory-
Figure 3.3: KV items in a Z-zone are compacted into blocks, which are organized into a binary trie. Only a leaf block (shaded) stores KV items (in the left), and grows two child blocks when it is overloaded (in the right).

access-efficiency, and a policy to dynamically adjust cache space allocation between the two zones to ensure that the KV-cache’s performance is not (substantially) compromised with workloads of changing access patterns.

In zExpander, a new request is first processed at its N-zone. For a GET request, if it hits in the N-zone, the result can be immediately returned. Otherwise, the request is passed to the Z-zone. A SET request is always immediately admitted by the N-zone. Only when an item is evicted out of the N-zone, it is then admitted into the Z-zone. A DELETE is performed simultaneously at both zones. While an N-zone manager is almost a plug-in of any existing KV cache system, which is responsible for communicating with clients and serving frequent accesses, we focus on the design of Z-zone.

3.3.1 Data Organization of the Z-zone

As suggested in Section 3.2.2, KV items are placed into containers, which are named blocks, for effective compression. Accordingly, indices are built on these blocks, rather than on individual KV items, to reduce use of pointers for space efficiency. In the design there are two objectives. One is that a block should be always well loaded for space efficiency. The other is that time to reach a block for a requested item should be minimized for CPU efficiency.

To this end, we organize the blocks into a binary trie, or a binary prefix tree. As shown in Figure 3.3, each tree node is labeled with a binary prefix. Each KV item’s key is also a binary string. An item is stored in a leaf node whose label matches its key’s prefix. In other words, internal nodes do not contain data. When a block is full and a new item needs to be inserted into the block, it grows two child nodes, and each of its items is moved to one of the new nodes according to next bit of its key’s prefix. Accordingly the corresponding node becomes an internal tree node, the space held by the block is de-allocated. Without

---

3For KV items larger than half of a block’s capacity, each of the items is compressed and stored individually, and a pointer recording its address is stored in the block where it is supposed to stay. As zExpander is designed for workloads of small items, we expect such large items are rare in the cache.
Figure 3.4: A trie, where KV items are stored at its leaf nodes, is enhanced with ghost leaf nodes to form a complete binary tree (b). Nodes on the tree are accessed through two-level pointer arrays (a). Note that spaces for internal and ghost nodes in (b) and pointer segments, such as [0,1,2,3] and [16,17,18,19] in (a), are virtual, or not physically allocated, to save memory. Furthermore, the links between parent and children nodes in (b) do not have physical presence.

We choose binary tree, rather than regular tree allowing for more children from a node, so that new nodes (blocks) are more likely to be well loaded (at least half full). However, to this end we also need to make sure storage load on these two nodes is well balanced. In addition, the entire binary tree has to be well balanced to avoid excessively long paths leading to some leaf nodes, or to avoid long access time for walking from the root to a leaf node on the trie. To this end, we apply a hash function, such as MurmurHash [18], on keys, and then use the hashed keys to locate the corresponding KV items in the trie. In this way, every block has an equal probability to receive KV items, and the chance of unbalanced data storage in the tree structure is minimized.

A conventional approach to locate an item in a tree is to chase pointers starting from the root node along a deterministic path leading to a leaf node according to the item’s key. However, on a binary tree, the path can be substantially long. Due to processor cache misses during the pointer chasing, the access time can be unnecessarily high. For higher access efficiency, we add minimal number of ghost leaf nodes to the tree to make it a complete binary tree. In a complete binary tree, “every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.” [2]. Each node is pointed by a pointer. A ghost node does not have any physical presence, and is pointed by (or associated with) a NULL pointer. We linearize these pointers into an array ordered from the top level to the bottom level, and in each level from the left to the right.
complete binary tree is formed. From the last pointer’s position in the array, we can directly compute the index of the last level (leaf or ghost) node corresponding to a given prefix of a hashed key. From this last level node, we can trace up the tree to identify the leaf node (with a non-NULL pointer) that possibly stores the item. This approach avoids traversing a (long) list of pointers from the root. In practice, to identify the non-NULL pointer, we only need to inspect a few (usually fewer than three) consecutive pointers up along the path starting from the last level node.

As shown in Figure 3.4, to avoid recording many pointers to the ghost nodes, or NULL pointers, we evenly partition the array into segments of fixed size, each with 128 4-byte-pointers. This array is then considered as the second level of pointers, and we maintain a so-called first-level array of pointers, each pointing to a segment in the second level. If all pointers in a segment are NULLs, the segment is not allocated to save memory space, and a NULL pointer is set in the corresponding slot of the first-level array.

### 3.3.2 Data Access and Replacement in the Z-zone

As KV items are compacted and compressed as a whole in a block with a default capacity of 2 KB, accessing the KV items involves decompression and/or compression of the block. Writing a new item into a block always leads to its reconstruction, including decompressing items in the block, re-compressing the existing items and the new item, allocating new memory space(s) to store the new block(s), and freeing the space held by the original block. Reading an item from a block requires a decompression of the block and searching the decompressed data for the item. For a quick search in the block, items are arranged in a block according to a hashed key order [92], and a small index consisting of offsets of up to eight items evenly spaced in the block is recorded so that only a few KV items have to be checked for a look-up in the block. In addition, to avoid unnecessary decompression operations on a block when looking for non-existing items, each block is associated with a Bloom filter, named as Content Filter, to remember items in the block. Expecting a block contains roughly 20 (for block size of 2 KB and small item size of 100 B) or fewer KV items, the Bloom filter is of 16-byte long. For each GET or DELETE request the Content Filter must be first checked to determine if the block needs to be accessed.

Unlike memcached who uses slab class to manage its own space (de)allocation for KV items, zExpander relies on the general-purpose memory allocator (malloc), usually provided by Glibc, for the allocation/de-allocation in the Z-zone. In this way, there is no internal fragmentation in the zone. Meanwhile, because the allocation size (a block) is large, space efficiency is less of a concern [121].

When the Z-zone has reached its allocated size, we need to determine an item for replacement. For this purpose, memcached organizes KV items in linked lists to find the least-recently-used (LRU) item. However, for a cache storing small items, the space
overhead for so many pointers can be too high. In addition, zExpander cannot track access history of the blocks to make its replacement decision. As the block where a KV item is placed is determined by the hashed key, each key has an equal probability to be inserted into any block, and each block can usually contain a mix of popular and unpopular items. To address the issue we need to exploit access locality of items within individual blocks. This is much like what the replacement policy for a set-associative processor cache does, except that zExpander needs to determine an item in a block, rather than a cache line in a cache set, for replacement. To efficiently track access history of items in a block, we associate a Bloom filter, named Access Filter, to each block to record recently accessed items. Like a Content Filter, an Access Filter is also of 16-byte long. Whenever an item is accessed for GET, its key is recorded in the corresponding Access Filter.

In the replacement policy, all blocks, or the trie’s leaf nodes containing KV items, are linked into a circular list. zExpander sweeps around the list. At each block it stops at, it tries to select victim items for replacement before moving to the next block. The victim items are selected by randomly choosing half of the items that are not recorded in the Access Filter. If all items are recorded in the filter, it skips the block. The Access Filter is cleared before zExpander moves to the next block, so that recent accesses can be recorded in the filter before this block is examined again.

3.3.3 Limiting Accesses in the Z-zone

While accessing KV items in the Z-zone is more expensive than that in the N-zone, especially for serving write requests, zExpander needs to control (relative) number of accesses on the zone. There are two potential issues that may cause a Z-zone to receive too many accesses. One issue is that the corresponding N-zone is not sufficiently large and causes actively accessed KV items to be spilled into the Z-zone. The other issue is frequent movements of items into and out of the Z-zone due to use of two-zone caching in zExpander.

3.3.3.1 Adaptive Cache Space Allocation

To address the first issue, we adaptively adjust cache space allocation between the two zones so that most requests can be processed at the N-zone and, if possible, the Z-zone has a substantially large size to help with the system’s memory efficiency. An N-zone has its target size, and the gap between its actual size and the target size suggests an action to expand or shrink the N-zone. Accordingly, a Z-zone has an action status, which can be expand, shrink, or stay, which indicates size change is not necessary. zExpander periodically checks the fraction of requests serviced at the N-zone in the current time window (one minute by default). If it is non-trivially smaller than a target threshold (90%
by default) and the current action status is not expand, \texttt{zExpander} increases the zone’s target size by 3\% of the cache space. Otherwise, if it is non-trivially larger than the target threshold and the current action is not shrink, the zone’s target size is reduced by 3\% of the cache space. We assume that the N-zone is managed by a KV cache system that supports resizing memory to a given size. Details on its implementation in \texttt{zExpander}'s prototype is described in Section 3.4.

When new items are added into the N-zone, \texttt{zExpander} will usually evict not-actively-accessed items into the Z-zone. Accordingly, to expand N-zone to its target size, \texttt{zExpander} simply keeps these items from being evicted. To shrink the N-zone, we leave a thread in the background and activate it for moving not-actively-accessed items into the Z-zone when the processors are not fully utilized.

In the calculation of requests serviced at a zone, we do not consider requests that do not require block (de)compression. These include missed \texttt{GET} requests and \texttt{DELETE} requests on non-existing items. Both types of the requests can be identified by Content Filters and the requests can be efficiently serviced. On the other hand, we consider items that are evicted from the N-zone into the Z-zone as requests serviced at the Z-zone. In this way, only expensive operations are counted and their impact on the system’s performance can be effectively capped.

3.3.3.2 Minimizing Write Operations at Z-zone

To address the second issue, we need to identify unnecessary item movements and remove them. Due to existence of access locality, a KV item can be relatively either an actively accessed (hot) one or an inactively accessed (cold) one in a certain time period. A hot item’s home zone is N-zone and a cold one’s home zone is Z-zone. When access pattern changes and a cold (hot) item turns hot (cold), moving the item from Z-zone (N-zone) to N-zone (Z-zone) is necessary. In the meantime, there are two scenarios where the movements are not necessary.

One scenario is that when an item is read from the Z-zone when a \texttt{GET} request is serviced, we need to know if it should be removed from the zone and inserted into the N-zone. To do this effectively, we need to make sure the item has turned from cold to hot. Otherwise, it would quickly return from the N-zone and back to the Z-zone. An item’s status, hot or cold, depends on its relative locality strength compared to that of items in the other zone. Therefore, the key to the answer on whether the item currently in the Z-zone has turned hot when it is read is to quantitatively compare locality strength of this item to that of any item currently in the N-zone. To this end, we need to measure an N-zone’s locality strength, which is defined as the weakest strength of all of its cached items. It is desired to have an efficient approach that does not require modification of KV cache code managing the N-zone. To this end, we treat the zone as a black box and periodically issue
a special SET request, named as Marker request, into the N-zone. Each Marker request has a unique key\(^4\) and will never be re-accessed. We then observe how long it will take for the item written by the Marker request to be evicted out of the zone. This duration represents the Z-zone’s locality strength, and is considered as its locality benchmark. The shorter the benchmark, the stronger the Z-zone’s locality strength. To be more effective, the benchmark we adopt is a weighted average of three most recent benchmarks. When an item in the Z-zone is accessed for the first time, zExpander records its access time without moving it into the N-zone. When it is re-accessed, the time gap, or re-use time, from its last access is calculated. If the re-use time is smaller than the N-zone’s locality benchmark, the item is moved to the N-zone. Otherwise, it remains in the Z-zone. In this way, the item being moved into the N-zone is likely to be actively re-accessed and stay there, and the item remaining in the Z-zone is less likely to be re-accessed soon. As we only need to identify the items that are best qualified to be moved from Z-zone into N-zone, for each block we only maintain two records for its recent accesses, each containing a hashed key (4 bytes) and a access time (4 bytes). They are of only 16 bytes, less than 1% of each block’s size.

The other scenario is that a SET request is received and a new KV item is first written into the N-zone. If at this time the old version of the item (of the same key) is in the Z-zone (by checking the Content Filter of the corresponding block), zExpander needs to decide if the version should immediately be removed from the Z-zone. For memory efficiency, we should keep an item from being doubly cached. However, an immediate removal may be followed soon with an eviction of the item’s new version from the N-zone and insertion into the Z-zone if the item is a cold one. To avoid the unnecessary early removal, we postpone it for a time period at least equal to the N-zone’s benchmark. If the item is evicted before or around when the time period expires, removal and write operations are merged into one at the Z-zone. Otherwise, the removal will be combined with the space reclamation in the Z-zone. When Z-zone’s replacement algorithm runs, it will first execute the pending removal operations, if any, before looking for LRU items in the blocks for replacement.

### 3.4 Evaluation

zExpander has been implemented and extensively evaluated with different workloads and system configurations. In the evaluation, we will answer three questions. First, can zExpander substantially reduce misses with little or limited loss of performance? Second, if zExpander’s performance loss is minimal with using memcached to manage its N-zone, is this still true with a high-performance KV cache of very-low-cost networking, such as RDMA, and increasingly large number of threads? Third, can zExpander effectively respond to change of access pattern and opportunistically retain its advantage

\(^4\)It contains special characters so it cannot appear in real workloads.
on miss reduction with its adaptive space allocation? In addition, we will also demonstrate why individually compressing KV items in off-the-shelf KV caches is not sufficient to achieve desirable miss reduction.

### 3.4.1 Implementation of Two zExpander Prototypes

We have built two prototypes, a memcached-based zExpander and a high-performance KV-cache-based zExpander.

In the first one, the N-zone is managed by memcached (Ver. 1.4.24), which is also responsible for communication with clients. However, memcached does not support online change of cache size, a capability required by zExpander and we attempted to add into memcached. The challenge is that memcached maintains multiple slab classes, each for storing KV items at a certain size range. To add or remove cache space, we would have to decide which classes’ allocations need to be increased or decreased and by how much. A decision on this has implication on what KV items are cached or replaced as well as on the cache’s miss ratio [68]. To keep authenticity of memcached’s behaviors and performance in the evaluation, we choose not to include the mechanism for adaptively adjusting cache space allocation in this prototype. Instead, we manually determine the target sizes for N-zone and Z-zone and statically configure them. In this prototype, we add about 85 lines of code to integrate memcached into zExpander, mainly for catching events, such as item evictions, GET misses, and item writes, so that corresponding operations at the Z-zone can be triggered.

Because items in different classes are managed in different LRU queues in memcached according to their sizes, we maintain a locality strength benchmark for each class, and use them to decide item movements depending on item size.

It is known that memcached’s performance is seriously constrained by its networking cost [145, 107, 93]. Its use of linked list in the hash table and use of the LRU list can lead to substantial processor cache misses, and limits the KV-cache’s performance. While these overheads can potentially overshadow the cost of Z-zone operations, we build a high-performance zExpander prototype to fully expose the Z-zone operation cost. In this prototype, network processing is removed by issuing requests at the user level of the server where the prototype KV cache runs. It also adopts optimistic Cuckoo hashing and CLOCK-based replacement suggested in MemC3 [56] to further improve the efficiency of N-zone operations.

### 3.4.2 Experiment Setup

In this evaluation, we replay the four traces (three Facebook’s memcached traces and one Zipfian trace generated by Yahoo’s YCSB) used in Section 3.2 as the prototyped
Figure 3.5: Miss ratios of systems using memcached with different workloads. One thread is employed for serving requests.

Figure 3.6: Size of (uncompressed) KV items cached in the systems using memcached with different workloads at the time when a trace has been replayed.

system’s workloads. While the traces do not contain actual values, we use the data sets about Tweeter’s location records to emulate the values (see Section 3.2.2). The value size is distributed in the range from 2 B to 327 B with an average of 100.9 B. We use LZ4 compression algorithm [7]. In the YCSB trace the ratio of GET and SET requests are 95% vs. 5% if not otherwise specified. In the trace replaying, we use first 1/5 of a trace to warm up the cache before collecting measurements.

The server running the prototypes has two Xeon E5-2680 v3 CPU, each having twelve cores with hyper-threading disabled and 30 MB last-level cache. The server is equipped with 128 GB (8 \times 16 GB) DDR4 memory. For the memcached-based zExpander, we use another server of the same configuration to generate and send requests over 10Gb Ethernet.

Each of the Facebook traces is a sequence of serialized requests. Concurrency relationship among requests is lost in the trace collection. To keep authenticity of the trace we do not artificially break it into multiple concurrent segments for replaying. Accordingly for these workloads, only one thread is employed to serve requests at the server.

In the below for brevity we name memcached-based zExpander as M-zExpander, the high-performance-KV-cache-based zExpander as H-zExpander. We also have H-
Cache by removing the Z-zone from H-zExpander and running the high-performance KV-cache exclusively.

3.4.3 Results for memcached-based KV Caches

Figure 3.5 shows miss ratios for the four workloads (ETC, APP, USR, and YCSB) running on memcached and M-zExpander with various cache sizes. Cache sizes for different workloads are chosen according to their respective data set sizes. As shown, M-zExpander can substantially reduce miss ratio, by up to 46%. The actual reduction depends not only on the increase of effective size but also on the workload’s demand on cache space in the increased cache size range. Figure 3.6 shows the size of KV items cached in the systems (in their uncompressed form) corresponding to each of the experiments in Figure 3.5. As an example, Figure 3.5 shows that USR achieves the largest miss ratio reduction (from 37% to 46%) with zExpander. However, its increase of amount of cached KV items with zExpander is moderate (from 42% to 63%) compared to the increases with other workloads. As suggested in Figure 3.2 and in Table 3.1, USR’s miss ratio has the largest reduction in the range of cache size from 20 GB to 50 GB. Another observation in Figure 3.5 is that the miss ratio reduction is pretty consistent across the selected cache sizes, suggesting that zExpander can be effective in a considerably large range of cache capacity. With zExpander, cache can be sized economically without compromising miss ratio. For example, for APP, a 15-GB cache with zExpander can have a miss ratio lower than a 20-GB cache with memcached.

To obtain insights on how zExpander allows substantially more KV items to be cached without adding DRAM, we first analyze the memory usage of memcached. The left two bars of Figure 3.7 show the usage for the YCSB workload at a 60-GB cache (corresponding to the right two bars of Figure 3.6d). As shown, in a memcached cache of 60 GB, only 56% of it (34 GB) is actually used to store KV items, and about 32% is for metadata, including three pointers to each item for hash table and its LRU replacement policy, and other cache management metadata specific to each individual item. The remaining cache space is mainly consumed by internal fragmentation in its slab allocation. zExpander relies on its Z-zone to increase effective cache size, or store more KV items. So we assume a zExpander with only Z-zone to cache the YCSB KV items. As shown in the right two bars in Figure 3.7, in the zExpander-managed cache, 88% of its space is used to store (compressed) KV items. Compared to its uncompressed form, the size of cached KV items is increased by 126% (34 GB vs. 77 GB). Because of its use of compact data structure (organizing blocks in the binary trie), the metadata holds only 3.3% of the cache space.

Note that cache sizes indicated in Figure 3.2 and in Table 3.1 include only (uncompressed) KV items, or those shown in Figure 3.6 as “Size of KV Items”.

48
Figure 3.7: Comparison of memory usages of a 60-GB KV cache managed by memcached, memcached whose KV items are individually compressed, and zExpander that has only Z-zone. The usage includes spaces used for KV items, metadata, and others (e.g., allocation fragmentations). It also shows amount of KV items in a cache should they be not compressed.

Even the allocation fragmentation is reduced with the use of Glibc’s memory allocator. This is made possible by (de)allocating larger blocks, rather than individual KV items.

To understand how a memcached that simply compresses items individually would help with memory efficiency, we write compressed KV items into the cache. The result is shown in the middle two bars of Figure 3.7. Unfortunately, with this compression only 13.5% more KV items are cached, and metadata cannot be reduced at all without use of batched compression and compact data structure.

Figure 3.8 shows throughput of the KV caches with various workloads and cache sizes corresponding to each of the experiments in Figure 3.5. As we explained, only one thread is employed to obtain the results. For the YCSB workload, we increase the number of threads (up to 24), each exclusively on one core, and show the throughput with different cache sizes in Figure 3.9. As seen, in the experiments M–zExpander’s throughput is
Figure 3.9: Throughput of systems using memcached with the YCSB workload and different number of threads. Caches of three different sizes are tested.

Figure 3.10: Throughput of systems using the high-performance cache with the YCSB workload and different number of threads. The cache size is 60 GB. Different mixes of GET and SET requests are tested.

within 4% of that of memcached, though it serves about 10% of requests at its Z-zone. A major reason is that memcached has a serious bottleneck on its networking processing [145, 107, 93]. Its throughput is less than 100 K RPS (requests per second) with one thread, and less than 700 K RPS with 24 threads. The throughput is even much lower than that of zExpander that serves all requests at its Z-zone, which is around 1.3 M RPS with one thread and around 18.1 M RPS with 24 threads, if networking is excluded. To fully expose the potential performance impact we compare the high-performance cache without networking involved (H-Cache), and zExpander with H-Cache to manage its N-zone (H-zExpander).

### 3.4.4 Results for High-Performance KV Caches

Figure 3.10 shows throughput of H-Cache and H-zExpander using YCSB workload with different mixes of GET and SET requests. As seen, the peak throughput can reach as high as 33 M RPS with the all-GET workload. For each workload, H-Cache’s throughput keeps increasing with number of threads until the number arrives at about 15.
Beyond this, it reaches a plateau and even slightly falls because lock contention intensifies. Before the number of threads becomes high (around 20), H-zExpander’s throughput is about 10%–15% lower than that of H-Cache. This is understandable as around 10% of requests are served at the Z-zone with a higher cost. However, when more threads are added, H-Cache’s throughput increase becomes slower or even stops. H-zExpander’s throughput keeps its increase longer and eventually (almost) catches up with H-Cache’s throughput. With the same number of threads, the lock contention is less severe in H-zExpander than that in H-Cache as some of the threads are diverted to perform more expensive operations at Z-zone. In other words, H-zExpander leverages some of the CPU cycles waiting for locks to meet the demand from Z-zone operations. In this way, for H-zExpander operations at its Z-zone does not compromise its N-zone’s performance. Another scenario where H-zExpander almost does not suffer any performance loss is when the system does not run at its peak capacity and has spare CPU cycles. Because most KV cache systems are over provisioned, this scenario is common.
With more SETs, both systems’ throughput reduces, as SETs intensifies H-Cache’s lock contention and they are (much) more expensive in H-zExpander’s Z-zone by involving compression. However, the relative throughput trend between the two systems stays.

Figure 3.11 compares the systems’ performance for the YCSB workload with 24 threads using a different metric—request processing time, which is the turnaround time of a request in the system and will be part of the latency observed by clients in a networked setting. With a smaller percentile, H-Cache has smaller processing time. However, at high percentiles, the H-zExpander’s latency becomes smaller. For example, at the 99% percentile, the times for H-zExpander and H-Cache are 4.0\mu s and 4.6\mu s, respectively (see Figure 3.11b). Intensive lock contention has been notoriously known to cause long execution delay [31, 32]. H-zExpander accidentally ameliorates the contention at the N-zone by re-directing some requests to the Z-zone.

Figure 3.12 shows the miss rate, or number of misses per second in each of the experiments shown in Figure 3.10. The reductions of misses are significant, often by 30% to 40% and by up to 1.48 million requests per second. Though H-zExpander has a lower throughput (by 10% to 15%), its reduction on throughput does not overshadow its improvement on miss rate. Altogether H-zExpander’s improvement on miss reduction is still impressive.

3.4.5 Effectiveness of Using Bloom Filter to Reduce Decompressions

To access a key in zExpander the binary trie for the Z-zone can only locate a block that possibly contains the key. To know if the key actually appears in the block, one has to perform expensive block decompression operation before the key can be searched. For reading non-existing keys, or GET misses, zExpander employs a Bloom filter (Content Filter) for each block to avoid unnecessary decompression operations. To quantitatively assess efficacy of the filter, we run a GET-only workload with different percentages of non-
existing keys, or different miss ratios. Figure 3.13 shows the throughput with the YCSB workloads at different thread count when the filters are used or not.

As shown in the figure, using the filters can substantially increase the cache’s throughput. Our measurements show that the filters’ false positive ratio remains at around 5%, or around 95% of misses do not come with block depressions. The throughput increase correlates with the miss ratio when the thread count is small. For example, with 50%, 75%, and 100% miss ratios, the increases are 39%, 53%, and 64% with five threads, respectively. However, when more threads are used, a higher miss ratio does not lead to a higher throughput increase. For example, with 20 threads the increases are 27%, 26%, and 20% at 50%, 75%, and 100% miss ratios, respectively. With a large number of threads and correspondingly high throughput, the impact of decompression on performance becomes less significant and other costs, such as lock contention, take a higher weight. In this case the benefit of using the filters does not increase with miss ratio.

Another observation on Figure 3.13 is that higher miss ratio leads to lower throughput, even when the Bloom filters are used. With a highly skewed access pattern, most request hits are served at the N-zone, thus are much more efficient than misses, which are always served at the Z-zone. Even though most decompressions can be avoided for misses, a higher miss ratio still degrades the throughput.

### 3.4.6 Impact of Space Allocation between N-zone and Z-zone

How to allocate cache space between N-zone and Z-zone is an important issue. With a too-large allocation to the N-zone, effort on miss reduction would be compromised. With a too-large Z-zone allocation, the cache’s performance could be unduly affected. While the allocation has to change in respond to changes of access pattern, a parameter
of zExpander about this is target percentage threshold, or the percentage of requests that should be processed at the N-zone. Figure 3.14 shows throughput and miss ratio of a 60 GB cache with the YCSB workload. As expected, the larger the threshold, the higher the throughput and the higher the miss ratio. As long as this threshold is sufficiently large (but not too close to 100%), its impact on throughput and miss rate is moderate. In the H-zExpander prototype, we choose 90% as the threshold, which provides high throughput and decent miss ratio reduction.

With a selected target threshold, zExpander automatically adapts to its space allocation to access pattern change. To observe the adaptation and its performance implication, we write about 24 GB KV items to the N-zone and the rest to fill the Z-zone of a 60 GB cache. We then send requests with a mix of 95% GET and 5% SET to the cache, initially with a uniform access key distribution, then (at around 480th second) change access pattern to Zipfian, or the access distribution assumed in our YCSB workload. Figure 3.15 shows the amount of data cached in the N-zone and that in the Z-zone. Figure 3.16 shows the miss ratio and throughput corresponding to allocations in Figure 3.15. With the uniform access pattern, N-zone has its maximum allocation, and almost all items are uncompressed. The cache stores only about 40 GB items. In the meantime, it has high throughput (29 M RPS) and high miss ratio (about 37%). After changing to the Zipfian access pattern, it takes about 900 seconds for the space re-allocation to be completed. After the re-allocation, N-zone has about 25 GB and most of the space goes to the Z-zone. With compression at Z-zone, the size of cached items increases to about 65 GB. The throughput is only moderately reduced (from 29 M RPS to 24 M RPS), and miss ratio is reduced to only 5.2%. 
3.5 Related Work

zExpand\text{er} is a KV cache system that leverages highly-screwed access pattern to simultaneously achieve high performance and low miss ratio. There have been many efforts reported in the literature related to various components of this work, including those on memory compression, memory allocation, effective indexing of KV items, and high performance KV caches.

3.5.1 Compression in main memory

As memory is fast but expensive, Compression Cache [52], ZSwap [16] and z\text{ram} [15] use some memory space as swapping area, in addition to that on the slow disk, and compress data in the area. Because memory pages in the space are compressed, the relatively small in-memory virtual area can be presented as a (much) larger swap space. With in-memory data compression, zExpand\text{er} follows the same idea. However, it has to address a difficult challenge. As data for compression on swap area is in the page unit, which is usually 4 KB and large enough to support an effective compression, KV items are of sizes distributed in a large range and often small (as small as a few bytes). Even though some KV in-memory stores, such as Redis [5], recommend users to individually compress KV items, the benefit is highly dependent on item sizes. zExpand\text{er} is immune to this constraint by aggregating items into larger blocks before applying compression. Another issue with individual compression is that metadata cannot be ‘compressed’. With a large number of small items in a cache, the metadata can be significant. By aggregating items, zExpand\text{er} can also substantially reduce metadata. While small KV items are common in KV-cache workloads, zExpand\text{er}’s contribution is substantial.
3.5.2 Indexing data structure

When metadata, or data for indexing KV items for their locations, can be a significant space overhead with small items, KV stores usually use compact data structure with sparse indices, such as LSM-tree [6, 143, 92]. As KV items of a KV store are mostly on the disks, the design goal is to minimize I/O operations, instead of processor cache misses, in searching for an item. In contrast, zExpander takes effort to reduce the metadata’s memory size and the cache misses by using batched item storage, balanced binary trie, and address calculation.

3.5.3 Fragmentation in memory allocation

Another source of memory overhead is memory allocation. Dynamic memory allocators, such as malloc/free in Glibc and its alternatives, are convenient choices and widely used [8, 55, 60]. However, for frequent allocation and deallocation of a large number of small items, space overhead due to fragmentation can be very high. While small items are popular in KV caches, this issue has to be addressed. There are two possible approaches. One is that adopted in memcached, which obtains large fixed size memory chunks (2-MB slabs) from the system and performs memory allocation by itself. Slab allocation has been a very successful strategy for managing data items of fixed sizes, such as inode and dentry of file systems. However, KV items are of all different sizes, which makes substantial internal fragmentation in the slabs, as revealed in our experiments. Another approach is to use recently proposed log-structured memory allocation [121]. However, it requires constantly moving data objects, imposing high CPU overhead, especially when the memory space is fully occupied, which is almost always the case with a KV cache. By aggregating KV items into blocks and requesting memory in blocks, zExpander can use the Glibc allocator without concern of its space efficiency.

3.5.4 KV store performance

Recent research on KV cache is mostly on its performance, or on how to increase its peak throughput [99, 93, 91]. Their efforts include optimizing data structure to reduce processor cache misses, leveraging advanced hardware features, such as RDMA and Direct Cache Access, for fast networking, and efficient concurrency control. zExpander is complementary to the optimizations, as N-zone can be managed by any high-performance KV cache management to take advantage of the improvements.
3.5.5 Replacement strategy

KV cache systems usually employ light-weight replacement algorithms to identify and keep an active set of KV items in the cache, so that requests can be quickly processed. To this end, even the efficient LRU algorithm is replaced with a CLOCK algorithm in MemC3 [56] to remove two pointers for each KV items and associated operations on them in LRU. MICA [93] even uses a replacement policy similar to that for the set associative cache to minimize the cache footprint in each lookup. With this technical trend on replacement algorithms in KV caches, there is little room to accommodate more intelligent but more expensive replacement algorithms to reduce miss ratio. In contrast, zExpander takes a different approach to reduce miss ratio, which is to increase effective cache size. The replacement policy used in zExpander’s Z-zone is also of very low cost by only identifying victim items for replacement within individual blocks.

3.6 Summary

We propose zExpander, a KV cache with both high performance and substantially reduced misses at the same time. This is made possible by uniquely leveraging an observation common in KV-cache’s workloads – accesses of the cache are highly skewed with a long tail. To enable an efficient system, we introduce a number of techniques, including batched compression, efficient indexing and data location on a complete binary trie, adaptive space allocation, and minimized data migration. More interestingly, zExpander can integrate any KV designs for high performance into its cache management with small code instrumentation. As an example, in one of the two prototypes, we add fewer than 100 lines of code into memcached to build the M-zExpander system. Porting more existing KV cache systems to zExpander is in our future work plan. We have extensively evaluated zExpander and demonstrated impressive results on both performance and miss reduction.
CHAPTER 4
LSM-TRIE: AN ULTRA-LARGE KV STORE FOR SMALL DATA

Key-value (KV) stores have become a backbone of large-scale applications in today’s data centers. The data set of the store on a single server can grow to billions of KV items or many terabytes, while individual data items are often small (with their values as small as a couple of bytes). It is a daunting task to efficiently organize such an ultra-large KV store to support fast access. Current KV storage systems have one or more of the following inadequacies: (1) very high data write amplifications, (2) large index set, and (3) dramatic degradation of read performance with overspill index out of memory.

In this chapter, we present LSM-trie as a solution to these issues. LSM-trie is a KV storage system that substantially reduces metadata for locating KV items. It reduces write amplification by an order of magnitude, and it needs only two disk accesses with each KV read even when only less than 10% of metadata (Bloom filters) can be held in memory. To this end, LSM-trie constructs a trie, or a prefix tree, that stores data in a hierarchical structure and keeps re-organizing them using a compaction method much more efficient than that adopted for LSM-tree. Our experiments show that LSM-trie can improve write and read throughput of LevelDB, a state-of-the-art KV system, by up to 20 times and up to 10 times, respectively.

4.1 Introduction

Key-value (KV) stores play a critical role in the assurance of service quality and user experience in many websites, including Dynamo [50] at Amazon, Voldemort [10] at LinkedIn, Cassandra [1] at Apache, LevelDB [6] at Google, and RocksDB [14] at Facebook. Many highly-demanding data-intensive internet applications, such as social networking, e-commerce, and online gaming, rely on quick access of data in the stores for quality service.

A KV store has its unique advantage on efficient implementation with a flat data organization and a much simplified interface using commands such as Put(key,value) for writing data, Get(key) for reading data, and Delete(key). However, there are several trends on workload characteristics that are seriously challenging today’s state-of-the-art KV store implementations for high performance and high scalability.

First, very small KV items are widespread. As an example, Facebook had reported that 90% of its Memcached KV pools store KV items whose values are smaller than 500 bytes [20]. In one KV pool (USR) dedicated for storing user-account statuses all values
are of 2 bytes. In its nonspecific, general-purpose pool (ETC) 2-, 3-, or 11-byte values add up to 40% of the total requests to the store. In a replicated pool for frequently accessed data, 99% of KV items are smaller than 68 bytes [107]. In the wildcard (the default pool) and a pool devoted for a specific application, 75% of items are smaller than 363 bytes. In Twitter’s KV workloads, after compression each tweet has only 362 bytes, which contains only 46 bytes of text [4]. In one of Instagram’s KV workloads the key is the media ID and the value is the user ID. Each KV item is just as large as a couple of bytes [13]. For a store of a given capacity, smaller KV items demand more metadata to locate them. The metadata may include index for locating a data block (e.g., a 4 KB disk block) and Bloom filters for determining data existence in the block.

Second, demand on a KV store’s capacity at individual KV servers keeps increasing. The rising demand is not only due to data-intensive applications, but also because of the cost benefit of using fewer servers to host a distributed KV store. Today it is an economical choice to host a multi-terabyte KV store on one server using either hard disks or SSDs. However, this would significantly increase metadata size and make memory constrained, which is especially the case when significant applications, such as MapReduce jobs, are scheduled to the cluster hosting the store, competing the memory resource with the storage service [46, 150].

Third, many KV stores require high performance for both reads and writes. It has been reported that ratio of read and write counts in typical low-latency workloads at Yahoo had shifted from anywhere between 2 and 9 to around 1 in recent years [122]. Among the five core workloads in Yahoo’s YCSB benchmark suite two of them have equal share of read and write requests [43]. There are KV stores, such as LevelDB, that are optimized for writes by organizing data in multiple levels. However, when not all metadata can be held in memory, multiple disk reads, each for metadata of a level, are needed to serve a read request, degrading read performance. In the meantime, for some KV stores, such as SILT [92], major efforts are made to optimize reads by minimizing metadata size, while write performance can be compromised without conducting multi-level incremental compactions.

We propose LSM-trie, a KV storage system that can accommodate multi-billions of small items with a capacity of multi-terabytes at one server with limited memory demand. It supports a sustained throughput of over 500 K writes per second, and a sustained throughput of over 50 K reads per second even for workloads without any locality and thus with little help from caching. To achieve this, LSM-trie uses three novel techniques. First, it integrates exponential growth pattern in the LSM tree (Log-Structured Merge-tree)—a commonly adopted KV-store organization—with a linear growth pattern. This enables a

---

1The throughput of read is significantly lower than that of write because one read needs access of at least one 4 KB block, while multiple small KV items in write requests can be compacted into one block.
compaction design that can reduce write amplification by an order of magnitude and leads to much improved write throughput. A high write throughput is desired as data modifications and deletions are also processed as writes in the store implementation. Second, using a trie, or a prefix tree, to organize data in the store, LSM-trie almost eliminates index. This allows more and stronger Bloom filters to be held in memory, making service of read requests faster. Third, when Bloom filters become too large to be entirely held in the memory, LSM-trie ensures that on-disk Bloom filters are clustered so that in most cases only one 4 KB-block read is required to locate the data.

Experiments show that LSM-trie significantly improves write throughput over schemes in comparison, including LevelDB, RocksDB, and SILT, by up to 20 times regardless of system configurations such as memory size, store size, storage devices (SSD or HDD), and access pattern (uniform or Zipfian key distributions). LSM-trie can also substantially improve read throughput, especially when memory available for running the KV store is limited, by up to 10 times.

Note that LSM-trie uses hash functions to organize its data and accordingly does not support range search. This is a choice similarly made in the design of many important KV stores, including Amazon’s Dynamo [50], LinkedIn’s Voldermort [10], and SILT [92], as this command is not always required by their users. Furthermore, there are techniques available to support the command by maintaining an index above these hash-based stores with B-link tree [33] or dPi-tree [96], and experimental studies indicate that “there is no absolute winner” in terms of range-search performance between stores natively supporting it and those relying on external support [113].

4.2 The design of LSM-trie

The design of LSM-trie was motivated by the excessively large write amplification of LSM-tree due to its data organization and compaction scheme [108]. In this section we will describe the issue in the context of LevelDB, a popular implementation of LSM-tree from Google. Then we will describe a trie-based LSM-tree implementation that can dramatically reduce write amplification in Section 4.2.3. However, this optimized LSM-tree still retains an index, which grows with the store size and eventually becomes a barrier to the system’s scalability. In addition, it may require multiple reads of Bloom filters on the disk with a large store. In Section 4.2.4, we describe LSM-trie, where KV items are hashed into individual buckets, indices are accordingly removed, and Bloom filters are grouped together to support efficient access.
Figure 4.1: Using multi-level structure to grow an LSM-tree store. Each solid rectangle represents an SSTable.

4.2.1 Write Amplification in LSM-tree

A KV store design based on LSM-tree has two goals: (1) new data must be quickly admitted into the store to support high-throughput write; and (2) KV items in the store are sorted to support fast data location. We use a representative design, LevelDB, as an example to explain the challenges on simultaneously achieving both of the goals.

To meet the first goal LevelDB writes to the disk in a large unit (a couple of megabytes) to generate an on-disk data structure called SSTable. Specifically, LevelDB first uses an in-memory buffer, called MemTable, to receive incoming KV items. When a MemTable is full, it is written to the disk to become an immutable SSTable. KV items in an SSTable are sorted according to their keys. An SSTable is stored as a file, and KV items are placed in 4 KB blocks of the file. To locate a KV item in the SSTable, LevelDB places an index in the file recording the key of the first KV item in each block. Conducting binary search on the index, LevelDB knows in which block a KV item can possibly be located. Because 4 KB block is a disk access unit, it is not necessary to maintain a larger index to determine byte offset of each item in a block. However, the index does not tell whether an item is actually in the block. If not, accessing the block is unnecessary and can substantially increase read latency. To this end, LevelDB maintains a Bloom filter for each block to indicate whether an item is in it [27]. To minimize its false positive rate, the filter must be sized proportionally to the number of items in a block, usually 10–16 bits per item.

To meet the second goal LevelDB builds a multi-level tree-like structure to progressively sort KV items. As shown in Figure 4.1a, new SSTables, which are just converted from MemTables, are placed in Level0. To quickly admit incoming items, items in new SSTables are not immediately sorted with those in existing SSTables at Level0. Instead,
each of the SSTables becomes a sub-level \((L_{0,0}, L_{0,1}, L_{0,2}, \ldots)\) of Level 0 (See Figure 4.1a).

In the background, LevelDB merge-sorts a number of \(L_0\) SSTables to produce a list of non-overlapping SSTables at Level 1 \((L_1)\), an operation called \textbf{compaction}. To quickly have more data sorted into one list, starting from Level 1 there are no sub-levels and the ratio of two adjacent levels’ sizes is large \((\text{Size}(L_{k+1})/\text{Size}(L_k), \text{where} \ k = 0, 1, \ldots)\). We name the ratio amplification factor, or \(AF\) in short, which is 10 in LevelDB by default. As every level \((L_{k+1})\) can be 10 times as large as its immediate upper level \((L_k)\), the store keeps producing exponentially larger sorted list at each level and becomes very large with only a few levels.

However, this exponential growth pattern leads to an excessively large write amplification ratio, a ratio between actual write amount to the disk and the amount of data requested for writing by users. Because the range of keys covered by each level is roughly the same, to push one SSTable at a level down to its next lower level LevelDB needs to read this SSTable and ten SSTables in the lower level (in the worst case) whose entire key range matches the SSTable’s key range. It then merge-sorts them and writes the 11 resulting SSTables to the lower level. That is, the write amplification ratio is 11, or \(AF + 1\). For a new KV item to reach Level \(k\) \((k = 0, 1, 2, \ldots)\), the write amplification ratio can go up to \(k \times (AF + 1)\). When the \(k\) value reaches 5 or larger, the amplification ratio can become unacceptably large (55 or larger). Such an expensive compaction operation can consume most of the I/O bandwidth and leave little for servicing frontend user requests.

For a store of given capacity, efforts on reducing the write amplification by limiting number of levels would have counter effect. One example is the SILT KV store [92], which essentially has two levels (HashStore and SortedStore). When the store grows large, its SortedStore has to be much larger than HashStore (even when multiple HashStores are employed). This causes its very high write amplification (see Section 4.3 for measurements), which justifies the use of multiple levels for progressive compaction in the LSM-tree-based stores.

### 4.2.2 Challenge on Reducing Write Amplification in the LSM-tree Compaction

A compaction entails reading sorted lists (one SSTable from \(L_k\) and a number of SSTables matching its key range from \(L_{k+1}\)), merging-sorting them into one sorted list, and writing it back to \(L_{k+1}\). While any data involved in the operation contribute to the write amplification, it is the larger data set from the lower level \((L_{k+1})\) that makes the amplification ratio excessively large. Because the purpose of the compaction is to push data to the lower level, the contribution to the amplification from accessing data at the upper level is necessary. If we manage to allow only data at the upper level to be involved in a compaction, the write amplification can be minimized.
To this end, we introduce the linear growth pattern. As shown in Figure 4.1b, in addition to Level 0 other levels also consist of a number of its sub-levels. Sub-levels belonging to the same level are of the same (maximum) size. When a new sub-level is produced at a level, the store linearly grows at this level. However, when a new level is produced, the store exponentially grows (by \( AF \) times). During growth of the store, new (sub)-levels are produced alternatively using the linear and exponential growth patterns. In other words, each LevelDB’s level is replaced by multiple sub-levels. To minimize write amplification, we can merge-sort data in the sub-levels of a level \( L_k \) to produce a new sub-level of its next lower level \( L_{k+1} \). As similar amount of data in each sub-level, but no data in the next lower level, are involved in a compaction, write amplification can be minimized.

A key consideration in LevelDB’s implementation is to bound each compaction’s maximum cost in terms of number of SSTables involved, or \( AF + 1 \), to keep service of user requests from being disruptively slowed down by the background operation. For the same purpose, in the use of linear growth pattern in a compaction we select one SSTable at each sub-level of a level \( L_k \), and merge-sort these SSTables into a sequence of non-overlapping SSTables at Level \( L_{k+1} \). The range of keys involved in a compaction represents the compaction’s key range. Among all compactions moving data from \( L_k \) to \( L_{k+1} \), we must make sure their key ranges are not overlapped to keep any two SSTables at Level \( L_{k+1} \) from having overlapped key ranges. However, this cannot be achieved with the LevelDB data organization because the sorted KV-items at each sub-level are placed into the SSTables according to the tables’ fixed capacity (e.g., 32 MB). The key range size of an SSTable can be highly variable and the ranges’ distribution can be different in different sub-levels. Therefore, ranges of the aforementioned compactions are unlikely to be un-overlapped.

### 4.2.3 SSTable-trie: A Design for Minimizing Write Amplification

To enable distinct key range in a compaction, we do not use a KV-item’s ranking (or its position) in a sorted list to determine the SSTable it belongs to in a level. Instead, we first apply a cryptographic hash function, such as SHA-1, on the key, and then use the hashed key, or hashkey in short, to make the determination. This essentially converts the LevelDB’s multi-level structure into a trie, as illustrated in Figure 4.2. Accordingly we name this optimized LevelDB SSTable-trie.

An SSTable-trie is a prefix tree whose nodes are table containers, each containing a number of SSTables. Each node has a fixed number of child nodes and the number is equivalent to the \( AF \) (amplification factor) in LevelDB. If the number is assumed to be 8, a node’s children can be distinguished by a three-bit binary (000, 001, …, or 111). A node in the trie can also be identified by a binary, usually of more bits. Starting from the root node, we can segment the binary into consecutive three-bit groups with the first group indicating a root’s child. As each bit group identifies a corresponding node’s child, we can
follow the bit groups to find a path to the node corresponding to the binary. All nodes of the same depth in a trie constitute a level in the trie structure, which is equivalent to a level in LevelDB. Each container has a pile of SSTables (see Figure 4.2). A trie level consists of a number of SSTable piles. All SSTables at the same position of the piles at a trie level constitute a sub-level of the trie, which corresponds to a sub-level in LevelDB.

As each KV item is also identified by a binary (the hashkey), its location in a level is determined by matching the hashkey’s prefix to the identity of a node in the level (see Figure 4.2). In contrast to the KV-item placement in a level of LevelDB, a KV-item’s location in a trie level is independent of other keys in the same level. A compaction operation involves a pile of SSTables in only one container. After a compaction KV items in a pile are moved into the container’s children according to their respective hashkeys, rather than their rankings in the sorted list as LevelDB does. By using hashkeys each compaction’s key range is unique and SSTables produced by a compaction are non-overlapping. Such a compaction
incurs minimal write amplification. Figure 4.3 illustrates a compaction operation in a trie. Note that use of SHA-1 as the hash function to generate hashkey guarantees a uniform distribution of KV items at each (sub)-level regardless of distribution of original keys.

4.2.4 **LSM-trie: a Large Store for Small Items**

Our goal is to enable very large KV stores in terms of both capacity and KV-item count in a server. A big challenge on designing such a store is the management of its metadata that often have to be out of core (the DRAM).

4.2.4.1 **Out-of-Core Metadata**

For a given KV item, there is at most one SSTable at each (sub)-level that may store the item in LevelDB because every (sub)-level is sorted and its SSTables’ key ranges are not overlapped. The store maintains a very small in-memory search tree to identify the SSTable at each level. At the end of each SSTable file an index and Bloom filters are stored to facilitate search in the table. The index is employed to identify a 4 KB block and a Bloom filter is maintained for each block to tell whether a KV item is possibly in the block. The indices and Bloom filters in a KV store can grow very large. Specifically, the size of the indices is proportional to the store’s capacity (or number of 4 KB blocks), and the size of the Bloom filters is proportional to total item count. For a large store the metadata can hardly be accommodated in memory. For example, a 10 TB store holding 100 B-KV-items would require about 125 GB space for 10-bit-per-key Bloom-filters and 30 GB for indices. While it is well affordable now and even so in the near future to have an HDD array or even an SSD array as large as 10 TB in a server, it is not cost-effective to dedicate such a large DRAM only for the metadata. Therefore, we have to assume that significant portion of the metadata is only on the disk when the store grows large. Because locality is usually not assumed in KV-store workloads [25, 132], the fact can be that most reads require retrieval of metadata from the disk before data can be read. The critical issue is how to minimize number of metadata reads in serving a read request for a KV item. These metadata are possibly stored in multiple SSTables, each at a different level. As the metadata are associated with individual SSTables and are distributed over them, having multiple reads seems to be unavoidable in the current LSM-tree’s structure.

SSTable-trie introduces the linear growth pattern, which leads to the design of **LSM-trie** that removes almost all indices and enables one metadata disk access per read request. Before describing the design, let us first address a concern with SSTable-trie. Using the linear growth pattern one can substantially increase number of levels. As a multi-level KV-item organization requires continuous search of levels, starting from Level 0, for a requested item until it is found, it relies on Bloom filters in each level to skip as many
levels without the item as possible. However, as each Bloom filter has a false positive rate (about 0.82% for a setting of 10 bits per item), the probability of searching levels without the item increases with the increase of level count (e.g., from 5.7% for a 7-level structure to 46% for a 56-level one). Therefore, the Bloom filter must be beefed up by using more bits. For example, using a setting of 16 bits per item would ensure less than 5% false positive rate for an entire 120-level structure. Compared with the disk capacity, the additional on-disk space for the larger Bloom filters is minimal. As we will show, LSM-trie removes indices and uses only one disk access to read Bloom filters.

### 4.2.4.2 Removing Indices by Using HTables

LSM-trie represents an improvement over SSTable-trie by incorporating an efficient metadata management. A major change is to replace the SSTable in SSTable-trie with HTable, a hash-based KV-item organization (see Figure 4.4). In an SSTable, items are sorted and index is needed for locating a block. In HTable, each block is considered as a bucket for receiving KV items whose keys are hashed into it. While each KV item has a SHA-1-generated 160-bit hashkey and its prefix has been used to identify an SSTable in SSTable-trie, or an HTable in LSM-trie, we use its suffix to determine a bucket within an HTable for the KV item. Specifically, if there are \( m \) buckets in an HTable, a KV item with Hashkey \( h \) would be placed in Bucket \( (h \mod m) \).

To eliminate the index in an HTable, LSM-trie must use buckets of fixed size. Further, as Bloom filter is applied on individual buckets, an entire bucket would be read should its filter indicate a possible existence of a lookup item in the bucket. Therefore, for access efficiency buckets should be of the same size as disk blocks (4 KB). However, a challenging issue is whether the buckets can be load balanced in terms of aggregate size of KV items hashed into them. It is known that using a cryptographic hash function allows each bucket to have statistically equal chance to receive a new item, and item count in each bucket
Figure 4.5: Distribution of bucket load across buckets of an HTable with a uniform distribution of KV-item size and an average size of 100 B (a), 200 B (b), and 300 B (c). The keys follow the Zipfian distribution. For each plot, the buckets are sorted according to their loads in terms of aggregate size of KV items in a bucket.

follows a normal distribution. In addition to key’s distribution, item size\(^2\) and variation of item size also add to variation of the bucket load.

Figure 4.5 shows the distribution of bucket load across the buckets in an HTable after we store KV items, whose keys are of the Zipfian distribution, into a 32 MB HTable of 8192 4 KB-buckets. For each plot, the item size is of the uniform distribution with different average sizes, and the size is in the range from 1 B to a size about doubling their respective averages. In each experiment we keep writing KV items to the store until it is 95% full. By using the highly-skewed Zipfian distribution, the results represent a conservative estimation of non-uniformity of bucket load distribution.\(^3\) As shown, there are increasingly more over-loaded buckets and more under-loaded buckets with the increase of average item size.

Obviously LSM-trie must move excessive items out of over-loaded buckets to make sure every bucket has 4 KB or less data. Like SSTable, HTable is also immutable. During the construction of an HTable, we use a greedy algorithm to migrate some items that were originally hashed to an overloaded bucket to an under-loaded bucket for storage. As illustrated in Figure 4.6, the buckets are first sorted into a list according to their initial loads. We then conduct a paired migration operation within the list, in which a minimal number of KV items are moved out of the most overloaded bucket (the source) to the most under-loaded bucket (the destination) until the remaining items in the source can fit in the bucket. The source bucket is removed from the list and we keep the list sorted. We then repeat the migration operation on the shorter list. The operation continues until either a list’s source bucket is not overloaded or the list’s destination bucket is also overloaded. To minimize the

\(^2\)With larger KV items it is harder to balance the load across the buckets in an HTable.

\(^3\)Interestingly the results are little affected by the key distribution. Even the uniform key distribution produces similar results.
(a) KV items are assigned to the buckets by the hash function, causing unbalanced load distribution.

(b) Buckets are sorted according to their loads and balanced by using a greedy algorithm.

Figure 4.6: Balancing the load across buckets in an HTable.

chance of having the second scenario, we set a limit on the aggregate size of KV items that can be stored in an HTable, which is 95% of the fixed HTable capacity (32 MB by default). This approach is effective. For example, with such a small reduction on usable capacity we have not observed a single item that is moved out of an over-loaded bucket but cannot be accommodated in an under-loaded bucket for HTables whose item sizes are 400 B on average and are uniformly distributed between 1 B and 800 B. Figure 4.7 shows the bucket load distribution after the load is balanced.

To handle the case of overflown items that cannot be accepted into any regular buckets, mostly due to excessively large KV items, during creation of a new HTable, LSM-tree sets up a special bucket to receive them. Items in the special bucket are fully indexed. The index is saved in the HTable file and is also cached in memory for efficiently locating
the items. As the bucket is designed only for a few large KV items, its index should be of minimal size. Generally, workloads for accessing consistently large items (a few KBs or larger) should use SSTable-trie. In fact, such workloads do not pose a challenge on their metadata management in most KV stores.

There are several issues to address on the load balancing strategy. One is how to efficiently identify KV items overflowed out of a bucket. To minimize the bookkeeping cost for the purpose, we use a hash function on the keys to rank KV items in a bucket and logically place them into the bucket according to their rankings. We then use the bucket capacity (4 KB) as the watermark. Any items that are across or above the watermark are considered as overflowed items for migration. We only need to record the hash value for the item at the watermark, named HashMark, for future lookups to know whether an item has been migrated. For the hash function, we simply select a 32-bit infix in the 160-bit hashkey (e.g., from 64th bit to 95th bit), as illustrated in Figure 4.8. We also record where the items are migrated (the destination bucket ID). A migrated item can be further migrated and searching for the item would need to walk over multiple buckets. To minimize the chance for an item to be repeatedly migrated, we tune the hash function by rotating the 32-bit infix by a particular number of bits, where the number is a function of bucket ID. In this way,
different functions can be applied on different buckets, and an item is less likely to keep staying above buckets’ watermarks for repeated migrations.

The metadata for each bucket about its overflown items comprise a source bucket ID (2 B), a migration destination ID (2 B), and a HashMark (4 B). They are stored in the bucket on the disk. A design issue is whether to cache the metadata in memory. If we cache every bucket’s metadata, the cost would be comparable to the indices in SSTable, which records one key for each block (bucket). Actually it is not necessary to record all buckets’ metadata if we do not require exactly one bucket read in an HTable lookup. As shown in Figure 4.5, distribution of overflown items over the buckets is highly skewed. So we only need to cache metadata for the most over-loaded buckets (20% by default) and make lookup of these items be re-directed to their respective destination buckets without a disk read. In this way, with slightly increased disk reads LSM-trie can significantly reduce its cached metadata. For example, when KV items are of 100 B in average and their sizes are uniformly distributed between 1 B and 200 B, only 1.01 bucket reads per lookup are needed with only 14 KB (1792 × 8 B) of the metadata cached, about 1/10 of the size of an SSTable’s indices.

Similar to LevelDB, LSM-trie maintains a Bloom filter for each bucket to quickly determine whether a KV item could be there. The migration of KV items out of a bucket does not require updating the bucket’s Bloom filter, as these KV items still logically remain in the bucket and are only physically stored in other bucket(s). Their physical locations are later revealed through the bucket’s migration-related metadata.

4.2.4.3 Clustering Bloom Filters for Efficient Access

LSM-trie does not assume that all Bloom filters can always be cached in memory. A Bloom filter at each (sub)-level needs to be inspected until a requested item is found. LSM-trie makes sure that all Bloom filters that are required to service a read request in a level but are not cached can be retrieved into memory with only one disk read. To this end LSM-trie gathers all Bloom filters associated with a column of buckets at different sub-levels of an HTable container into a single disk block named BloomCluster, as illustrated in Figure 4.9. Because the same hash function is applied across the sub-levels, a KV item can appear only in one particular column of buckets if it is in the container. In this way, only one disk read of Bloom filters is needed for a level.

While LSM-trie is designed to support up to a 10 TB store, its data is organized so that at most one read of metadata (Bloom filters) is required to access any item in the store. The prototyped LSM-trie system uses 32 MB HTables and an amplification factor (AF) of

---

As shown in Figure 4.9, the column of buckets refers to all buckets at the same position of respective HTables in a container.
8. The store has five levels. In the first four levels, LSM-trie uses both linear and exponential growth pattern. That is, each level consists of eight sub-levels. All the Bloom filters for the first 32 sub-levels are of 4.5 GB, assuming a 64 B average item size and 16 bit Bloom filter per key. Adding metadata about item migration within individual HTables (up to 0.5 GB), LSM-trie needs up to only 5 GB memory to hold all necessary metadata. At the fifth level, which is the last level, LSM-trie uses only linear growth pattern. As one sub-level of this level has a capacity of 128 G, it needs 8 such sub-levels for the store to reach 1 TB, and 80 such sub-levels to reach 10 TB. All the sub-levels’ Bloom filters are well clustered into a BloomCluster so that only one disk read of Bloom filter is required for a read request. Though the false positive rate increases with level count, it can be well capped by using additional bits per KV item, as shown in Table 4.1. When LSM-trie uses 16-bit-per-item Bloom filters, the false positive rate is only about 5% even for a 112-sub-level 10 TB KV store. In the worse case there are only 2.05 disk reads, one for a BloomCluster and 1.05 on average for data.

In the LSM-trie structure, multiple KV items of the same key, including special items for Delete operations, can simultaneously stay in different sub-levels of the last

---

5 Actual number of sub-levels in a level can change during compaction operations. It varies between 0 and 16 with an average of 8.
level without being merged as there are no merge-sort operations at this level. Among the items of the same key, only the item at the highest sub-level is alive and the others are considered as garbage. This may lead to under-utilized disk space, especially when the level contains substantial amount of garbage. To ameliorate the effect, we periodically sample a few random HTable containers and assess their average garbage ratio. When the ratio is larger than a threshold, we schedule garbage-collection operations in a container-by-container manner either periodically or when the system is not loaded.

### 4.3 Performance Evaluation

To evaluate LSM-trie’s performance, we implement a prototype and extensively conduct experiments to reveal insights of its performance behaviors.

#### 4.3.1 Experiment Setup

The experiments are run on a Dell CS23-SH server with two Intel Xeon L5410 4-core processors, 64 GB FB-DIMM memory, and 64-bit Linux 3.14. The SSD (Samsung 840 EVO, MZ-7TE1T0BW) has 1 TB capacity. Because of its limited storage capacity (1 TB), we install DRAM of moderate size on the computer (64 GB), a configuration equivalent to 256 GB memory with a 4 TB store. We also build a KV store on a hard disk, which is 3 TB Seagate Barracuda (ST3000DM001) with 64 MB cache and 7200 RPM. Table 4.2 lists the disks’ performance measurements. As we can see, the hard disk’s random read throughput is too small and it’s not competitive considering SSD’s rapidly dropping price. Therefore, we do not run read benchmarks on the hard disk. All experiments are run on the SSD(s) unless stated otherwise. In LSM-trie immediately after a table is written to the disk, we issue `fsync()` to persist its data.

In the evaluation, we compare LSM-trie with LevelDB [6], RocksDB (an optimized LevelDB from Facebook) [14], and SILT [92]. LSM-trie uses 32 MB HTables, LevelDB and RocksDB use 32 MB SSTables, and SILT uses 32 MB HashStore. We run SILT using its source code provided by its authors with its default setup [12]. We do not include experiments for SSTable-trie as its write performance is the same as LSM-trie, but its read

<table>
<thead>
<tr>
<th></th>
<th>SSD</th>
<th>HDD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Read 4 KB (IOPS)</td>
<td>52,400</td>
<td>70</td>
</tr>
<tr>
<td>Sequential Write (MB/s)</td>
<td>230</td>
<td>144</td>
</tr>
<tr>
<td>Sequential Read (MB/s)</td>
<td>298</td>
<td>138</td>
</tr>
</tbody>
</table>

Table 4.2: Basic disk performance measurements.
4.3.2 Experiment Results

In this section we present and analyze experiment results for write and read requests.

4.3.2.1 Write Throughput

Figure 4.10 plots the write throughput, in terms of number of PUT queries served per second (QPS), for LSM-trie, LevelDB, RocksDB, and SILT with different store sizes, or numbers of KV items in the store. We have a number of interesting observations on the plots.

---

6We do have a test for the Zipfian distribution in Section 3.2.
Figure 4.11: Write amplification ratios of different stores. For each store, the execution stops when either the store reaches 1TB or the run time reaches 24 hours, whichever occurs earlier.

The LSM-trie store has throughput way higher than other stores. Even the throughput for LSM-trie on the hard disk (see the “LSM-trie-HDD” curve) more than doubles those of other stores on the SSD. It takes about 24 hours for LSM-trie to build a 1TB store containing nearly 8 billions of small items on an HDD. As it is too slow for the other stores to reach the size of 1TB within a reasonable time period, we stop their executions after they run for 24 hours. By estimation it would take RocksDB and LevelDB about 4–6 days and even longer time for SILT to build such a large store on the SSD. Admittedly SILT is designed mainly to service read requests \[^92\]. However, taking so long to build a large store is less desirable in the first place. To understand their big performance gaps, we draw the write amplification ratio (WAR) plots for the stores in Figure 4.11.

It’s not a surprise to see SLIT’s WAR increases almost linearly with the store size, as SILT does not adopt a multi-level organization. By maintaining a large SortedStore and merge-sorting much smaller HashStores into it, most of its compaction I/O is to access data in the SortedStore, and contributes to the WAR. While both LevelDB and RocksDB adopt LSM-tree’s multi-level organization, its exponential growth pattern significantly compromises its WAR. The WAR curve of RocksDB is obtained by running its performance monitoring tool (db_bench). The curve exhibits large variations, mainly because of its choice of sampling points for performance measurements. While RocksDB generally has a higher WAR, its write throughput is higher than that of LevelDB because of its use of multiple threads to better utilize parallelism available in SSD and CPU. The WAR curves
for LSM-trie ("LSM-trie-*" curves in Figure 4.11) have small jumps at about 0.12 and 1.0 billion items in the KV store, corresponding to the timings when the store grows into Levels 3 and 4, respectively (Figure 4.11). Once the store reaches its last level (Level 4), the WAR curves become flat at around 5 while the store increases up to 10TB.

The write throughput curve for the hard disk ("LSM-trie-HDD") in Figure 4.10 has two step-downs, well matching the two jumps in its corresponding WAR curve. After the store reaches 1 billion items, its throughput does not reduce with the increase of the store. For LSM-trie on the SSD, we do see the first and second step-downs on the curve ("LSM-trie-1SSD" in Figure 4.10) corresponding to the two WAR jumps. However, we had been confused by the third step-down, as marked in Figure 4.10, when the store size reaches about 1.7 billion items or 210GB. One might attribute this throughput loss to the garbage collection. However, we had made efforts to use large HTables (32MB) and aligned them to the erase block boundaries. After investigation, it turns to be due to SSD’s internal static wear-leveling.

As we know, frequency of data re-writing at different levels dramatically varies. The ratio of the frequencies between two adjacent levels (lower level vs. upper level) can be as high as 8. For data at Level 4 and at Level 0, the ratio of their re-write frequencies could be 4096 (8^4)! With such a large gap between the frequencies, dynamical wear-leveling is insufficient and SSD’s FTL (Flash Translation Layer) has to proactively move data at the lower level(s) around to even out flash wear across the disk. The impact of the wear-leveling becomes increasingly serious when more and more SSD’s space is occupied. To confirm our speculation, we introduce a second SSD and move data at the two upper level (about only 2.5 GB) to it, and run LSM-trie on the two SSDs (see “LSM-trie-2SSD” in Figure 4.10). The third step-down is postponed to a significantly later time (from about 1.7 billion items to about 5.2 billion items). The new third step-down is caused by re-write frequency gaps among data at Levels 2, 3, and 4 in the first SSD. Using more SSDs and separating them onto different SSDs would eliminate the step-down. In practice, it is a viable solution to have a few small but wear-resistant SSDs (e.g., SLC SSD) to separate the first several levels of data.

We also issue write requests with the Zipfian key distribution to LSM-trie on two SSDs. It has a smaller WAR than those with the uniform key distribution (see “LSM-trie-2-zipf” in Figure 4.11), and higher throughput (see “LSM-trie-2-zipf” in Figure 4.10). Strong locality of the workload produces substantial overwrites, which are merged during the compactions. As a result, about one third of items are removed before they reach the last level, reducing write amplification and increasing throughput. The Zipfian distribution also allows LevelDB to significantly reduce its WAR (compare “LevelDB” and “LevelDB-zipf” in Figure 4.11) and to increase its write throughput (compare “LevelDB” and “LevelDB-zipf” in Figure 4.10).
In almost all scenarios, LSM-trie dramatically improves WAR, leading to significantly increased write throughput. The major reason of the improvements is the introduction of the linear growth pattern into the LSM tree and the adoption of the trie structure to enable it.

### 4.3.2.2 Performance of Read

Figures 4.12 and 4.13 plot the read throughput for various stores on one SSD with 64Gb and 4GB memory, respectively, except SILT. Keys of read requests are uniformly distributed. As explained, we cannot build a sufficiently large SILT store to measure its read performance. Instead, we will use the results reported in its paper for comparison [92]. To
Table 4.3: Read Latency under mixed read/write workload.

<table>
<thead>
<tr>
<th>Latency Percentile</th>
<th>5% read</th>
<th>50% read</th>
<th>95% read</th>
</tr>
</thead>
<tbody>
<tr>
<td>95%</td>
<td>95%</td>
<td>700 µs</td>
<td>700 µs</td>
</tr>
<tr>
<td>99%</td>
<td>99%</td>
<td>830 µs</td>
<td>830 µs</td>
</tr>
</tbody>
</table>

accelerate the building of the LevelDB and RocksDB stores, we use YCSB to generate a trace of write requests whose keys are sorted. The stores can then be quickly built without any compactions.

As shown in Figure 4.12, when the store size is relatively small (with fewer than about 1 billion KV items or 128 GB data), almost half of accessed data can be cached in memory and the throughput is very high (much higher than 80K QPS). This throughput is not explicitly shown in the figure, as it is less I/O related. LSM-trie has higher throughputs than LevelDB and RocksDB for both small and large store sizes. With a small store size, LSM-trie uses less memory to cache metadata and leaves more for caching data than other stores, producing higher hit ratios and read throughputs. When the store becomes larger, the working set becomes larger due to uniform key distribution and the memory size becomes less relevant to the throughput. LSM-trie’s higher throughputs with larger store are due to the alignment of its block to the SSD pages in its implementation. Without the alignment, one access of an SSTable-file’s block may result in access of an additional page. For the following experiment we augment LevelDB and RocksDB by aligning their blocks to the SSD pages. LSM-trie’s throughput with a large store (over 6 billions KV items) is around 96% of one SSD’s raw read throughput in terms of number of 4 KB-blocks read per second. This is the same percentage reported in the SILT paper [92].

Considering the scenario where a server running a KV store may simultaneously run other application(s) demanding substantial memory resource, or where a KV store runs within a disk drive with small memory, we evaluate LSM-trie’s performance with a constrained memory size. Figure 4.13 shows read throughput when the memory is only 4 GB. Current LSM-trie’s implementation always keeps metadata for the first four levels in the memory. More and more requests require one read of out-of-core metadata in addition to one read of data after the store grows beyond the first four levels. This is why the curve for LSM-trie starts to drop beyond 1.2-billion-item store size. The throughput curves of LevelDB and RocksDB also drop with the increase of store size. They drop much more than that of LSM-trie. RocksDB’s throughput is higher than that of LevelDB initially, as it caches more metadata by giving metadata a caching priority higher than data.

Our measurements show that all requests can be completed in 1 ms, and its 99% percentile latency is 0.92 ms. To know how read latency is affected by concurrent write

Note that write performance is not affected by the small memory.
requests, we list the 95% and 99% percentile latencies for different percentages of read requests among all the read/write requests in Table 4.3. The read latencies are not sensitive to write intensity. The KV store store many small items in write requests into one block while each read request has to retrieve an entire block. Thanks to the much reduced write compaction in LSM-trie, intensity of write requests has a small impact on read latency.

4.4 Related Work

Key-value stores have become an increasingly popular data management system with its sustained high performance with workloads challenging other systems, such as those generating a huge number of small data items. Most related works aim for efficient writes and reads.

4.4.1 Efforts on Supporting Efficient Writes

Most KV stores support fast writes/updates by using log-based write, such as FAWN [17], FlashStore [47], SkimpyStash [48], SILT [92], LevelDB [6], and bLSM [122]. Though log-appending is efficient for admitting new data, it is not sufficient for high write efficiency. There can be significant writes caused by internal data re-organization and their efficiency can be critical to the write throughput observed by users. A primary objective of the re-organization is to remove garbage from the log. Some systems, such as FAWN, FlashStore, and SkimpyStash, focus mostly on this objective and incurs a relatively small number of additional writes. Though these systems are efficient for serving writes, they leave the data not well organized, and produce a large metadata set leading to slow reads with relatively small memory.

Another group of systems, such as LevelDB, SILT, and bLSM, aim to build a fully organized data structure—one (almost) sorted list of KV items. This is apparently ideal for reducing metadata size and facilitating fast reads. It is also essential for a scalable system. However, it can generate a very large write amplification. The issue quickly deteriorates with the growth of the store. To address the issue, RocksDB compacts more than two contiguous levels at once intending to sort and push data faster to the lower level [14]. However, the improvement is limited as the amplification is fundamentally due to the difference of the data set sizes at different levels. To mitigate the compaction cost, TokuDB uses a Fractal Tree, in which data is pushed to its next level by simply being appended into log files at corresponding tree nodes [85, 26]. Without well sorting its data, TokuDB has to maintain a much larger index, leading to larger memory demand and/or additional disk access for metadata. In contrast, with the support of the trie structure and use of linear growth pattern, LSM-trie minimizes write amplification.
4.4.2 Efforts on Supporting Efficient Reads

Read efficiency is mostly determined by two factors. One is metadata size and the other is the efficiency of retrieving metadata from the disk. Both determine how many disk reads are needed to locate a requested KV item.

As SILT has a fully sorted list of KV items and uses a highly compact index representation, it produces very small metadata [92]. In contrast, LevelDB’s metadata can be much larger as they include both indices and Bloom filters. It may take multiple reads for LevelDB to load its out-of-memory metadata. FAWN [17] and FlashStore [47] have very large metadata as they directly store pointers to the on-disk items, especially when the items are small and the store is large. SkimpyStash stores hash table buckets on the disk, essentially leaving most metadata on the disk and may require many disk reads of metadata to locate the data [48]. In contrast, LSM-trie substantially reduces metadata by removing almost all indices. It requires at most one metadata read for each read request with its well clustered metadata.

4.4.3 Other Related Works

Sharding (or partitioning), as a technique to distribute heavy system load such as large working sets and intensive I/O requests across nodes in a cluster, has been widely used in database systems and KV stores [9, 105, 3]. It has been proposed as a potential method for reducing merge (or compaction) overhead by maintaining multiple smaller store instances (shards) at a node [92]. However, if the number of shards is moderate (fewer than one hundred) at a node, each shard has to grow into four or larger number of levels when the store becomes large. Accordingly write amplification cannot be substantially reduced. Meanwhile, because memory demand, including MemTables and metadata, is about proportional to the number of shards, using many shards increase pressure on memory. In contrast, LSM-trie fundamentally addresses the issue by improving store growth pattern to minimize compaction cost without concerns of sharding.

Being aware of large compaction cost in LevelDB, VT-Tree opportunistically looks for any block at a level whose key range does not overlap with that of blocks at another level during merge-sorting of the two levels’ KV items [123]. Effectiveness of this method relies on probability of having non-overlapping blocks. For workloads with small items, there are a large number of keys in a block, reducing the probability. Though it had been reported that this method can reduce write amplification by about $\frac{1}{3}$ to $\frac{2}{3}$, it is far from enough. In contrast, LSM-trie reduces the amplification by up to an order of magnitude.

While LSM-trie trades some disk space (around 5%) for much improved performance, Yu et al. proposed a method to improve performance of the disk array by trading capacity for performance [148]. They trade 50% of the disk space for a throughput improvement of 160%.
4.5 Summary

We describe LSM-trie, a key-value store designed to manage a very large data set in terms of both its data volume and KV item count. By introducing linear growth pattern, LSM-trie minimizes compaction cost for LSM-tree-based KV systems. As our extensive experiments demonstrate, LSM-trie can manage billions of KV items with a write amplification of only five. By design it can manage a store of up to 10 TB. LSM-trie can service a read request with only two SSD reads even when over 90% of the bloom-filters is not in the memory. Furthermore, with a second small SSD (only 20 GB) to store the bloom-filters, the overall throughput can reach the peak throughput of the raw device (50 K QPS vs. 52 K IOPS), and 99% of its read latency is below 1 ms.
CHAPTER 5
NVMCACHED: A KV CACHE ON BYTE-ADDRESSABLE NVM

As byte-addressable, high-density, and non-volatile memory (NVM) is around the corner to be equipped alongside the main memory, issues on enabling indispensable key-value cache services, such as memcached, on the new storage medium must be addressed now. While theoretically NVM allows data in a KV cache to survive power outage and system crash, their integrity and accessibility depend on data consistency enforced during writes to NVM. While techniques for enforcing the consistency are available (journaling, COW, or checkpointing), they all heavily rely on the expensive FLUSH operation, which causes extra writes and compromising NVM's limited write endurance.

In this chapter we design and evaluate NVMcached, a KV cache for non-volatile memory that can significantly reduce flushes while minimizing data loss by leveraging batched space allocation and reclamation. Experiments show that NVMcached can improve the system throughput by up to $2.8 \times$ for write-intensive real-world workloads, comparing with a non-volatile memcached.

5.1 Introduction

With the recent announcement of 3D XPoint technology [74], non-volatile memory (NVM) becomes reality and will change how major system infrastructures are designed and built. In particular, emerging byte-addressable, high-density NVMs, such as PCM [137], STT-RAM [117], and RRAM [134], enable an alternative to DRAM as main memory to be more energy-efficient and of larger capacity. When computer servers configured with NVM become commonly available, porting popular key-value (KV) caches, whose designs assume DRAM memory, onto NVM-equipped servers would allow their data to survive power outage and system crash. This transition brings significant benefit to the critical services in today’s data centers. KV caches leverage large memory in a cluster of servers to temporarily store data and provide quick access to them, while it can be expensive to regenerate a KV item (e.g., from a long execution of an SQL statement in a backend database system) with a miss in the cache [89]. The cache is accessed via a simple interface, such as GET(key) for reading data, SET(key, value) for writing data,¹ and DELETE(key) for removing data. KV caches have been playing a critical role in maintaining high service

¹SET command could perform either INSERT or UPDATE depending on the presence of its corresponding key.
Figure 5.1: Ratio of memory write throughput over that with a flush attached to each write. The 1-KB writes are issued uniformly in a continuous space (the working set). The working set size increases from 1 MB to 64 GB.

quality and improving user experience in many large-scale websites, such as Facebook and Twitter [107]. By enabling data persistency, an NVM-based KV cache can retain its cached data and continue supplying them after a system restart without dramatically degrading service quality [153].

However, to ensure the KV items cached on NVM are still usable after a system crash, KV items written onto the NVM must be crash consistent [112]. That is, the caches must maintain data integrity in the presence of a system crash that potentially leads to partial/reordered writes. There have been many studies on providing crash consistency. Proposed solutions can be categorized into logging [114, 41, 133], copy-on-write (COW) [42, 131], and checkpointing [119, 59]. However, existing solutions are too expensive for high-performance KV caches. They can incur substantial overhead on writes, which not only compromises NVM’s limited write endurance but also degrades the cache’s throughput.

Recent research on KV cache has pushed its peak throughput to many millions and theoretically 1 billion requests per second, by using efficient data structures to reduce processor cache misses and enable efficient concurrency control [91, 93, 145, 56]. The aforementioned solutions, such as logging and COW, mostly rely on cache flush to promptly persist data and enforce write order. However, it has been reported that frequently flushing cache lines can slow memory writes by more than $4 \times$ [106]. To illustrate how prompt data persisting affects the efficiency of memory access, we run a micro-benchmark that does random writes of 1-KB values in a given memory space. We gradually increase the size of the memory space (or the working set size) from 1 MB to 64 GB. The benchmark runs on a Dell PowerEdge R630 server with two Xeon E5-2680 v3 12-core processors of 30 MB L3 cache, and 256 GB (16 × 16 GB) DDR4 DRAM. As shown in Figure 5.1, memory write throughput, in terms of number of 1-KB data writes per second, can be increased by over $18 \times$ in a small working set of a few megabytes if all flushes are removed. Even with a
large working set of tens of gigabytes, the throughput can still be improved by $6 \times$ if flushes are removed.

Frequent cache flush neutralizes cache’s write-back capability that exploits access locality. This is why using flush imposes a larger penalty on accesses with a smaller working set. Cache flush is especially undesirable for real-world KV cache workloads, which are usually highly skewed and have small writes (keys and even values of a few bytes) [20, 107]. Furthermore, heavily using flushes can also disrupt the cache line replacement logic, which may have been well designed for optimal memory access performance.

In this work, we leverage a unique property of KV cache to remove all flushes except those associated with DELETE and UPDATE requests. Unlike KV stores, it is acceptable for a KV cache to lose some of its cached KV items because anyway the KV items can be replaced out of the cache without notifying the users. Note that KV cache is a look-aside cache, and any lost KV items can be re-computed and re-inserted into the cache by users. Take memcached as an example, it is described as “clients must treat memcached as a transitory cache; they cannot assume that data stored in memcached is still there when they need it.” [136]. For correctness we only need to make sure that no wrong value is returned for subsequent GET requests. To this end, we do not use any flush for INSERT requests to promptly persist metadata and data or to enforce their write order. Instead, we store checksum of data along with its corresponding metadata so that the data integrity can be verified when necessary. In this way users will only receive correct data. On the other hand, for DELETE or UPDATE operations, cache flushes have to be used to remove or invalidate the corresponding KV items persistently. In this way, these items will never re-surface even after an unexpected server crash and restart.

While tolerance of data loss by being a cache provides opportunity of mostly removing requirement of immediately persisting data, a KV cache has another unique property that poses bigger design challenges. Once a KV cache is full, it runs its replacement algorithm to constantly identify and evict victim items to reclaim space. To minimize the use of flush, we aim to not immediately persist changes reflecting the space reclamation. However, there are three issues to address to achieve this objective. First, a KV cache often chains its items in linked lists in a hash table. If an item on a list is being evicted, one may need to use a flush to maintain the list integrity. Even if the aforementioned checksum technique is applied to detect data corruption, all of the items following the deleted one on the list can be lost after a crash. While a cache allows data loss, the amount of loss should be minimized. Second, when an item’s space is reclaimed and becomes available, the metadata recording free spaces for allocation, such as free list or bitmaps, needs to be updated. If the update is not promptly persisted with cache flush, possibility of losing the information due to a system crash can compromise the entire allocation metadata after a restart. This demands a complete walking over all cached items to recover the correct
memory usage, making the KV caching service unavailable for an extended period of time. Third, when the cache decides to replace a KV item, it updates certain metadata, such as pointer or status bit, to reflect this replacement. If flush is performed for each replacement, the number of flushes can be as large as number of items inserted into the cache (once the cache is full). However, if these updates are not immediately persisted, there is a possibility that the correctness of the system is violated. Let’s consider this scenario: when the status about a replaced KV item is still cached in the processor cache, a deletion request regarding this item is processed according to the in-processor-cache information indicating that the item is no longer present in the KV cache. Consequently it is deemed that further actions for the deletion request, such as persistently invalidating the in-memory key, is not necessary before the request is acknowledged. If at this moment the system crashes, the item that supposedly has been deleted is still in the non-volatile memory and it will re-surface after a restart. A user deletes an item from the cache usually because it becomes invalid and should not be used. Making the deleted item available without user’s consent is an intolerable mistake.

We propose a NVM-based KV cache design, named NVMcached, that takes advantage of its tolerance of data loss to minimize use of flushes, addresses the issues caused by its replacement operations to minimize data loss, and uses near-zero flushes except for processing DELETE and UPDATE requests.

In summary, we make three contributions in the work.

- We identify opportunities and challenges of building an efficient non-volatile-memory-based key-value cache. Consistent with recent efforts of improving cache locality on building very-high-performance KV caches, we show that keeping data in the processor cache for longer time period with minimal use of cache flushes is critical for an efficient NVM KV cache.
- We design NVMcached, an NVM-based KV cache that uses checksums to weed out corrupted data so that most cache flushes can be avoided. In response to the issues with item replacement and space reclamation, NVMcached adopts a batched approach. Specifically, it allocates and reclaims memory in log-organized zones of fixed size (e.g., 32 MB). It uses zone versioning to enable efficient batched space reclamation, which makes it unnecessary to use flush on each to-be-deleted item. Furthermore, leveraging KV-cache’s highly-skewed request pattern, NVMcached minimizes garbage collection cost.
- We prototype NVMcached and extensively evaluate it with Facebook’s memcached traces and Zipfian workloads generated by YCSB. NVMcached can improve the system throughput by up to $2.8 \times$ for real-world workloads and up to $4.5 \times$ for synthetic Zipfian workloads.
5.2 Design of NVMcached

In the design of NVMcached, for correctness it is guaranteed that a KV item that has been deleted (or invalidated by an UPDATE request) will not come back alive after a system crash. To this end, a DELETE/UPDATE request must be accompanied with at least one cache flush if we assume that the client issuing the request has to be immediately acknowledged. Considering this requirement, NVMcached sets its primary performance goal as producing almost zero flushes beyond ones necessitated by DELETE/UPDATE requests. As a secondary goal, it maximizes the number of KV items surviving a system crash.

To achieve the goals, there are three design principles to follow. First, NVMcached should not use (long) linked lists, because without using immediate flushes a system crash may break multiple lists and make many KV items on the lists unreachable (lost). Second, space must be reclaimed in a temporally and spatially batched fashion. That is, multiple physically contiguous KV items should be replaced in batch so that cost of necessary flushes for recording the replacement can be amortized. Third, memory allocation must also be in a batched fashion for a quick recovery of memory allocation metadata even without their immediate persistency using flushes.

5.2.1 Storing Data Checksums to Avoid Ordered Writes

One potential use of long lists in a KV cache is to chain KV items on collision at a hash table entry. One example use is in memcached, where each KV item contains a pointer to the next one on a list. Such a list serves two purposes. One is to store the KV items, and the other is to index the items for their location. To avoid use of lists, we separate storage and indexing of items into two data structures. As shown in Figure 5.2, a list of chained KV items from a hash table entry are replaced with an array of KV-item descriptors.
Each descriptor includes a pointer\textsuperscript{2} to actual storage of the corresponding KV item. In other words, KV items are individually pointed by pointers in a descriptor array, and a system crash breaking one pointer will affect only one item. Each descriptor also contains a hash-tag, which is calculated from the key of the corresponding item. When searching in an array, only the descriptors whose hash-tag matches the key’s hash-tag will be considered for full-key matching. The array is pre-allocated with a limited number of slots. A slot can be empty when the array has not yet been filled or the KV item associated with it has been deleted. For higher lookup efficiency, a bitmap is attached to each descriptor array, with each bit for one slot in the array. A bit is set only when its corresponding slot is occupied by a valid descriptor. In this way, lookup can be quickly performed by skipping the empty slots. The bitmap also helps with allocation by skipping the non-empty slots in an array. In the rare cases where a new slot is demanded when all slots in an array are filled, a second array is allocated and linked to the first one. This expansion of descriptor array will be persisted with an immediate flush. As sufficient number of slots should have been pre-allocated according to the KV cache’s capacity, the expansion only imposes small overhead to the system.

Though it is acceptable to lose some items in a KV cache due to a system crash, some data’s persistency order still matters. Each KV item has to be persisted on the NVM before updating its corresponding pointer to prevent the incomplete data from being reachable. While an intuitive method of enforcing the desired write order is to use flush, it can be too expensive. Instead, we compute 32 bit-checksum of each KV item’s on-NVM contents (including both key and value) using the CRC32 hash function \cite{111}. The approach has been applied to efficiently provide crash consistency in file systems \cite{138,115}. We store the checksum and the item’s size as part of a descriptor along with the pointer to avoid the use of cache flushes for persisting each KV item. When accessing KV items, those with mismatched checksums are detected so that corrupted items will not be returned to the client. Note that it is not necessary to compute checksum and perform the comparison upon every KV-item read. A KV item should be checked only when its integrity cannot be determined, which only happens after a crash and restart. To this end, we maintain a volatile bitmap in DRAM for each descriptor array, each bit for one descriptor, which is zeroed out after system boot (preceded by normal shutdown or unexpected system crash). When accessing an item for the first time its integrity is verified by examining its CRC32 checksum. Its corresponding bit is then set to avoid repeated verification. In this way, the use of checksums is of very low cost. The pseudo code of how to process \texttt{GET} requests is described in Algorithm 1.

\textsuperscript{2}Because KV items are allocated in segments of memory named \textit{zone}, the pointer of an item is actually computed from its zone-id and its offset in the zone.
Figure 5.3: Throughput of writes accompanied with checksum calculations or flush operations.

**Algorithm 1** Handling GET request

```plaintext
function process_GET(cache, key)
  index ← hash(key)
  entry ← cache.hashtable[index]
  for array ← entry do
    for d ← array do
      if d.hashtag ≠ key.hashtag then
        continue
      if array.bitmap[d.id] = 0 then
        verify_checksum(d)
      if d.data_size = 0 then
        continue
      dv ← d.zone_version
      zone ← cache.zones[d.zone_id]
      zv ← zone.version
      if dv ≠ zv then
        d.data_size ← 0
        continue
      ptr ← zone.base + d.offset
      kv ← check_and_get(ptr)
      if kv ≠ NULL then
        return kv
```

However, one might be concerned that as checksums need to be calculated for every new KV item admitted in the KV cache, it may add excessive overhead to a high-performance KV system. We run a micro-benchmark to investigate the impact of checksum
calculation for data writes in DRAM. In each operation a value of fixed size is copied to a randomized location within a pre-allocated memory space (a working set). In addition to the data write, we either compute a CRC32 checksum or use cache flush instructions to compare the two costs. As shown in Figure 5.3a, with a small working set that fits in the processor cache, calculation of checksums can reduce the write throughput by roughly 30% to 70%. The seemingly significant overhead is due to the fact that the data movements are all conducted within the cache and DRAM is mostly not involved in the data movement. However, checksum calculation is at least five times faster than using cache flushes. When we increase the working set size, as shown in Figure 5.3b, the throughput of writes with checksum calculation becomes much closer to that of pure memory writes as both of them need to frequently access DRAM. However, cache flush still shows significant overhead with a throughput of less than a quarter of that of the others.

Another potential use of long linked lists is to organize KV items for recording access history for the replacement algorithm to identify items of weak locality for eviction. One example is the LRU list of the LRU algorithm adopted by memcached. Without using flushes, the list(s) can be easily broken and become unusable after a crash. To make matters even worse, the list(s) need to be updated even with GET requests. While using the CLOCK replacement algorithm can eliminate the list(s) [56], one still needs to frequently update the CLOCK data structure for each GET request. To address the issue, we move the LRU list to the DRAM to remove requirement on its crash consistency, as we assume a hybrid memory where a fraction of its space is composed of DRAM. However, by doing so all access history would be lost after a crash, and cache efficacy can be compromised accordingly. To address this issue, NVMcached lays out KV-items in the NVM to roughly reflect their access locality, which provides clues for the replacement algorithm after a crash. Furthermore, it helps to reduce garbage collection cost (more details in Section 5.2.2).

5.2.2 Managing Memory in Zones for its Efficient Allocation and Reclamation

A KV cache requires a memory allocation mechanism for space allocation and deallocation. One readily available choice is off-the-shelf memory allocators such as the Glibc’s malloc() and its alternatives [87, 55, 60]. Their implementation usually involves long lists for organizing free and allocated spaces. It needs extensive use of cache flushes and even hardware supports for its performance optimization [106], which is inconsistent with NVMcached’s design principle. Another possible choice is memcached’s slab-based allocation mechanism, in which slabs of a fixed size are assigned to different classes and each class holds KV items in a certain size range [57]. However, this can lead to serious
internal space fragmentation. Even worse, slabs need to be constantly moved across classes to simulate the effect of a global replacement algorithm [68], raising the demand on flushes.

For memory allocation, it has been shown that a log-structured approach to memory management can achieve a high memory utilization [121]. NVMcached also adopts the general approach. In addition to the high utilization, it also helps to level wearing on NVM, which usually has limited write endurance. However, there are two issues to address to enable efficient persistent memory allocation. First, garbage collection is a major source of inefficiency in a log-structured system, as it needs to migrate live data while collecting garbage data. When invalidated data is of only a small fraction, the migration can be very inefficient. Second, in a log-structured allocator the free spaces released by evicting unpopular KV items can be scattered across the log. They cannot be immediately reused until the garbage collection process reaches them, making re-collection of the spaces very inefficient. To address the issues, we integrate garbage collection and replacement operations in a zone, and exploit strong access locality exhibited in KV cache workloads.

5.2.3 Batched KV item replacement and space reclamation in Zones

In NVMcached, we organize the log for space allocation and reclamation in zones of fixed size. In a zone to be cleaned, there are two types of KV items for garbage collection. One includes those that have been deleted or invalidated by DELETE or UPDATE requests. The other includes those that have been evicted by the replacement algorithm due to their weak access locality. In NVMcached, the replacement algorithm is not independent of the garbage collection operation. Instead, the replacement of KV items in a zone is performed when the zone is being cleaned to increase efficiency of space reclamation and reduce garbage collection cost. In addition, this enables batched replacement of KV items, which amortizes the cost of a cache flush demanded by each replacement operation.

Each zone has a unique identifier (Zone-ID) and a version number (Zone-Version), as shown in Figure 5.2. Each KV-item descriptor contains a Zone-ID, a corresponding version number of the zone, the item’s offset in the zone, and the item size. The purpose of associating a version to each zone is to quickly remove all items in a zone from being accessed. To access an item in a zone, in addition to the checksum verification, the zone version number in its descriptor must match its counterpart on the zone. Otherwise, mismatched zone version indicates that the descriptor is invalid. Once a zone’s version number is incremented, all the KV items in it become inaccessible due to mismatched version numbers. Therefore, we only need to immediately persist the updated version number of the zone with a cache flush. Using only one flush, all of the items in the zone can be replaced without the risk of allowing deleted items to resurface. In Section 5.1, we have explained that a flush is required for each KV-item replacement to prevent deleted items from being re-surfaced. NVMcached’s batched KV item replacement significantly amortizes the cost.
After incrementing a zone’s version number, each KV item in the zone will have one of two possible fates. One is to be removed from the zone if it is already a deleted or invalidated item (by DELETE and UPDATE requests, respectively) or if it is of weak access locality and should be evicted. The other is being retained within the zone if it is of strong locality. Those KV items are to be revitalized by incrementing the version number in their descriptor. To enable efficient space reclamation and replacement, we need to address two issues. One is how to determine the locality strength of an item. The other is how to reduce data migration cost for the items to be retained as the KV items that are qualified to be retained are likely to be scattered throughout the zone.

The locality of the items in a zone must be quantified in comparison with that of other items in the cache. To this end, we maintain a global LRU list in the DRAM to record the access history for GET requests. Because items are of variable sizes, the list’s capacity is defined by the aggregate size of all recorded items, rather than by the item count. In a regular caching system, it might be acceptable to replace truly least-recently-used (LRU) items to make only enough room for new items. However, this approach can be excessively expensive in a log-structured memory management where the replaced LRU items can be dispersed in different zones. Therefore, in NVMcached any not-recently-accessed (NRU) items are eligible for replacement. To this end, we only maintain a small LRU list for the most-recently-used (MRU) items. This list is accordingly named the MRU list. It is worth noting that removing an item from this MRU list does not mean it will be immediately evicted from the KV cache. Instead, they will be evicted from the cache during batched zone cleaning process.

Figure 5.4 illustrates essential data structures related to the zone cleaning operation. When a zone is to be cleaned, only the items recorded in the MRU list will be considered as ones of strong locality and be immune to replacement. All other live items in a zone will be cleaned along with deleted and invalidated items in the zone. To minimize the cost on identifying hot items belonging to a zone, an additional list is maintained for each zone to record the descriptors pointing to the same zone. Note that the MRU list is maintained in the DRAM. So it does not require flushes with its updates for crash consistency. However, after a crash all access history recorded in the list is lost. To address the issue, NVMcached lays out items in a zone in such a way that the history can be approximately reflected even without the MRU list. To retain items in the MRU list, or hot items, in a zone when it is cleaned, we move them to the beginning of the zone, and revitalize them by populating their descriptors with the zone’s new version number. The offset at the end of the area for hot items is named hot line. With a hot line recorded in each zone, NVMcached at least knows what items should not be replaced right after a recovery from a crash. In the meantime, the remaining space following the hot line is reclaimed and ready for new allocations.

3 Populating the new zone version does not need flushes as loss of items is allowed.
Figure 5.4: Data structures for zone cleaning. In the shown scenario where Zone 0 is being cleaned, the hot items A and C are recognized by visiting the corresponding per-zone list. A and C are moved to the beginning of Zone 0 and their metadata in their descriptors are updated accordingly.

As the allocation is in a log-structured manner, new items will be written in the zone in a back-to-back manner and the end of last writing position is a critical metadatum about memory allocation. Usually loss of such metadata would have a severe consequence—all of the data objects in the memory have to be visited before free space can be identified and used for admitting new writes. `memcached` uses a bitmap to record which slots in a slab are allocated or not. A straightforward implementation of `memcached` in NVM would be using flush to promptly persist the corresponding bit once a slot allocation is made, which is too expensive. In contrast, `NVMcached` does not have this concern. When allocation begins in a zone, the zone’s identifier is immediately persisted to indicate that it is the current zone for space allocation, named as active zone. After that, space allocation will not require flushing of corresponding metadata. After a crash, the information about the current allocation status will be lost. It would be inefficient to scan the items in the active zone to find out where the last allocation happened and from where the allocation can resume for new requests. Instead, after the restart we immediately switch to the next zone in the log to start allocation. As the MRU list in the DRAM is already lost due to the crash, the hot line in the new zone is used to do a quick cleaning. In the meantime, the previous active zone remains untouched so that future GET requests could still retrieve valid KV items, which are verified by checksum matching.

A major concern on the log-structured space allocation is on its potentially high cleaning cost, or specifically the cost of moving live data out of the area being cleaned. `NVMcached` has effectively addressed the issue by taking advantage of a property of KV
cache workloads, which is that access of KV items is highly skewed. That is, a small percentage of items receives most of the accesses. This is reported in the study of memcached’s workload [20] and widely assumed in studies of KV caches [94, 93, 56, 53, 91, 144]. This property supports NVMcached’s choice of using NRU items, instead of pure LRU items, for replacement. More importantly, it implies that set of the hot items before the hot line, or named hot set, is highly likely to be stable. When NVMcached starts to clean a zone, it first identifies current hot set comprising this zone’s items in the MRU list, and then it is compared with the last hot set comprising items before the hot line. If most of the items in the last hot set (80% by default) still remain in the new hot set, then items in the last hot set will not be relocated and the rest of the new hot items will be relocated right after the old hot line. In this way, most data movement for zone cleaning can be avoided.

5.2.4 Caching at DRAM

In the design of NVMcached, we assume a hybrid memory with a fraction of it being DRAM. Depending on disparity of access speeds (read or write) between DRAM and NVM, we may use DRAM as a cache for NVM. As reported, for the two most promising NVM technologies, PCM and STT-RAM, the major issues are related to write. For example, PCM has much higher density than DRAM (by at least 2-4×) [155]. However, its write can be 10-100× slower than that of DRAM, while its read is only 2-4× slower [116]. Additionally, its energy consumption for write can be 10-50× more than that of DRAM [88, 137, 84]. For STT-RAM its read latency and energy consumption are comparable to those of DRAM. However, its write latency is 1.25-2× higher than DRAM and it has 5-10× higher energy consumption [84, 40, 90].

Considering the challenges on NVM’s performance and endurance, NVMcached uses DRAM as a write cache to keep short-lived items from entering the NVM, including ones that can quickly be deleted or updated. However, there would be three issues if we let every new KV item first enter the write cache. First, it can cause expensive data movement for cold items as they will be first written into the write cache and soon be evicted to NVM, leading to doubled data write traffic. Second, UPDATE would cause expensive thrashing between DRAM and NVM. When processing UPDATE, if an old version of the item is in NVM, it should be first invalidated using flushes and then the UPDATE can be finished by inserting the new item into the write cache. Otherwise, after crash-restart the old item will re-surface, violating the correctness of the KV cache. As a cold item is to be evicted to NVM soon, holding it in the write cache has no benefit but leads to extra overhead. Third, the write cache would consume large DRAM space to have the capability to retain sufficient amount of really hot items. This is very inefficient as most of the items in the write cache should be cold, especially for highly-skewed workloads.
To address the aforementioned issues, we let the write cache record the hash-tag of recent SET requests in an MRU list\(^4\) which only consumes a few tens of bytes for each item. A hit count is also maintained in each list node, so hot items can be identified if it has a large hit count. An item that receives three or more hits is considered a hot item. A new KV item is written directly to the NVM unless it can be considered hot according to its hit count. In this way, we can make sure that only hot items will have a chance to stay in the write cache. There will be much less data movement between write cache and NVM. Also the use of hash-tag significantly reduces the DRAM usage for retaining both hot items and sufficient access history. However, the performance penalty of having the in-DRAM cache is that a GET request (for read) has to first access the write cache, and then access the NVM if it had a miss in the write cache. The penalty depends on the read latency disparity between DRAM and NVM. For example, for PCM whose read latency is 2-4× slower than DRAM, this penalty is small and is dwarfed by the benefit from reduced writes to the NVM.

### 5.3 Evaluation

In this section we evaluate the efficacy of NVMcached by answering three questions: How does a persistent KV cache preserve the hit ratio across a crash-restart time period even with expected data loss? Is it affordable to apply conventional approaches, such as using flushes, to provide crash consistency for an existing KV cache? Does NVMcached preserve high performance while keeping most of its data persistent after a crash?

To answer the above questions, we conducted extensive experiments on memcached and NVMcached. In the experiment, we issue requests from threads running at the KV cache server to exclude networking from the measurements. We had observed that performance and scalability of memcached are greatly constrained by its slow network stack, which is also reported in previous studies \([145, 128]\). In the state-of-the-art KV caches, network cost has been minimized, and the time spent on CPU and memory access have been the dominant cost \([104, 135]\). We remove the networking component from memcached in our experiments to expose the performance potential of non-volatile KV cache in a high-performance environment.

#### 5.3.1 Experiment Setup

The experiments run on a Dell PowerEdge R630 server with two Xeon E5-2680 v3 12-core processors of 30 MB L3 cache, and 256 GB (16×16 GB) DDR4 DRAM. The operating system runs Linux kernel 4.4.1. The hyper-threading feature is turned off from

\(^4\) This MRU list independently maintains history for SET requests without interfering the MRU that records GET requests.
BIOS. We currently do not have large-capacity NVM devices. Instead, DRAM is used to emulate NVM.

In the evaluation, we replay the traces collected at Facebook’s production memcached system [20] and Zipfian workloads generated from Yahoo’s YCSB benchmark suite [43].

We configure NVMcached’s hashtable to have $2^{26}$ hash entries, each with eight slots for pre-allocated descriptors. For comparison, we use three memcached-based KV caches, which are named as follows:

- **MC-DRAM** is the stock memcached running on DRAM. All data in the system will be lost after a crash.
- **MC-NVM** is the memcached running on an assumed NVM with data persistency. After every SET/DELETE operation, the new KV item and its associated metadata are promptly persisted with carefully ordered cache-flush instructions so that after a crash-restart data-structures are not corrupted and data loss is minimized to only affecting unfinished operations. However, the LRU operations are not persisted in MC-NVM as the loss of LRU does not affect the volatility of the KV items.
- **MC-NVLRU**, is based on MC-NVM and it keeps the LRU list in the NVM and maintains its crash consistency.

### 5.3.2 Evaluating the Impact of Crash-restart with Real Traces

To examine the impact of a crash-restart in a volatile KV system, we replay four of Facebook’s memcached traces (APP, USR, ETC, and SYS). One of them (VAR) was not selected because it is write-dominant—only a few keys appear in the GET requests.

Each experiment has two phases. In the first phase, the cache is warmed up by replaying the trace until it produces a stable hit ratio. In the second phase, we emulate an event of system crash and observe the variation of the miss ratio until it becomes stable again.

In this experiment we compare two systems. One is the volatile memcached that runs on DRAM, named MC-DRAM. The other is the NVMcached. While all data will be lost in volatile memcached, NVMcached can still retain most of its data after a crash on an assumed NVM. As the NVM is emulated, the crash of NVMcached has to be simulated by destroying data that are likely still in the processor at the time of the crash. To be conservative, we remove a total amount of 60 MB of the most recently inserted KV items, which is two times larger than the last-level cache of CPU (30 MB). We set the capacity of the KV caches to 64 GB, the same configuration of the servers where the traces were collected.

Figure 5.5 shows the miss-ratio change after the crash. For three of the workloads, APP (Figure 5.5a), USR (Figure 5.5b), and ETC (Figure 5.5c), memcached shows a significant increase of miss ratio after the crash. However we do not observe 100% miss ratio,
because during the time interval that the hits/misses are being collected, the most popular items have been restored and contribute to the hit count for subsequent GET requests. For the skewed access pattern, the less popular keys continue to contribute to misses for an extended period of time. The miss ratio slowly reduces but it does not obtain a complete recovery even after billions of requests have been served. Instead of running into high miss ratio, NVMcached does not show any disruption on miss ratio after the crash. The reason for its strong resilience is that the lost data is just of a tiny fraction of the total amount of data – only about 0.1% (60MB data loss vs. 64GB cache size).

For the SYS workload (Figure 5.5d), the miss ratio shows a small spike and is recovered quickly after the crash. This workload has a small working set compared with the other workloads, and over 50% of its requests are SETs. The two factors make this workload easier to recover from a crash without undergoing a long recovery process. Therefore, NVMcached keeps a consistent low miss-ratio without showing any visible disruption.

5.3.3 Performance of NVMcached with Synthetic Workloads

We have shown Non-volatile KV cache can avoid a sudden increase of miss ratio after a system crash. However, this benefit does come with additional cost and the
performance impact could be a potential concern as expensive flushes have to be used. We evaluate the performance of memcached and NVMcached to see how NVMcached improves the throughput by (mostly) removing the overhead. We run micro-benchmarks on NVMcached and the three memcached-based systems (described in Section 5.3.1).

We use Zipfian workloads with mixed GET/SET requests with different compositions. Figure 5.6 shows the results with one thread and three different value sizes (100 B, 1000 B, and 10000 B). MC–DRAM’s performance is the best among all the systems as it is by nature designed for DRAM with no cost on maintaining crash-consistency. MC–NVM performs well for GET-dominant workloads but the throughput deteriorates quickly as the percentage of SET requests increases. With additional cost of persisting LRU and ensuring its consistency, MC–NVLRU performs even worse and its throughput for GET-only workloads is also reduced. However, with larger value size, the cost of persisting the KV data dominates the execution time and the cost of persisting LRU history becomes less significant.

NVMcached maintains a consistently high throughput across all the tests. However, the design of NVMcached also adds extra overheads. For example, the write cache adds extra lookup cost for both GET and SET requests. While removing the write cache can improve performance of GET, frequent SET operations can add even more cost to NVM devices. As DRAM is used in the experiments to emulate the slower NVM, we believe
the write cache should be more cost-effective in a real hybrid DRAM+NVM configuration. Another overhead is due to that each 16-byte descriptor doubles the space of each pointer, which is 8-byte in 64-bit systems. The increased memory usage reduces the cache utilization of the hot items, especially for those running skewed workloads. With modestly reduced throughput (compared to that MC-DRAM), NVMcached enables the persistency of KV data. It also preserves the approximate access locality by efficient zone cleaning. In contrast, after a crash-restart the entire data set is lost in MC-DRAM and the access history in MC-NVM is also lost.

We then evaluate the scalability of NVMcached by using concurrent threads issuing requests. Each thread exclusively runs on a dedicated CPU core and we increase the number of threads to 12 threads. The value size in this experiment is set to 100 B. Figure 5.7 shows the results. All of the four systems show increased throughput when the number of threads increases. While NVMcached shows an approximately constant throughput with different proportions of GET/SET requests, memcached-based systems show better scalability for GET-dominant workloads. NVMcached outperforms MC-NVLRU by at least 60% and up to 3×. It also outperforms MC-NVM when SET requests is of 20% or more. This experiment suggests that increasing concurrency cannot reduce the cost of flushes.
5.3.4 Performance of NVMcached with Real-world Workloads

To get a more insightful analysis of how NVMcached helps with real-world KV cache systems, we replay the five Facebook KV traces on NVMcached and MC-NVM. Figure 5.8 shows the experiment results. While three of them—APP, ETC, and USR—show almost identical performance for NVMcached and MC-DRAM, the other two traces show more than 2× difference. To explain the differences we first need to take a look at the components of each trace. Table 5.1 lists the proportion of each type of requests in the five traces. Two of those traces, SYS and VAR, contain a significant percentage of SET requests. This suggests that NVMcached has effectively removed the flushes for the SET requests, improving the overall throughput by 2× for SYS (half SET and half GET), and by 2.85× for VAR (virtually SET-only). However, for the other three GET-dominant workloads, even with a large portion of DELETE requests, the throughput stays the same between NVMcached and MC-NVM. It is because in both systems cache flushes are unavoidable in processing DELETE requests. In summary, NVMcached effectively improves the performance for write-intensive workloads and maintains the high performance for read-dominant workloads.

5.4 Related Work

In this section we discuss previous studies on using NVM and its integration in KV systems.

5.4.1 Persistent Memory Allocator

Mnemosyne, NV-heaps, and the NVM Library provide general-purpose programming interfaces for accessing NVM [133, 41, 73]. They need to employ expensive undo/redo logging to maintain consistency for persistent transactional updates. NVMalloc tries to optimize the persistent memory allocation by minimizing metadata writes [106]. However, expensive cache flushes are still required to persist each allocation and it shows a roughly 75% performance degradation compared with non-persistent malloc. In addition to the cost on allocation, user data still requires extra cost for correct ordering and persistency. Instead of relying on the expensive allocation interfaces to provide consistency, NVMcached mostly removes the necessity of promptly persisting data/metadata by employing the KV item descriptor to verify the integrity of KV items after a crash-restart.

Slab allocator can greatly reduce external fragmentation for fixed-size objects [30]. However, it needs to maintain a free list for efficient allocation. If one wants to move the free list to DRAM to avoid its costly updates on NVM, the entire memory space managed by the allocator has to be scanned to recover the free slots. Log-structured memory can reduce the allocation cost to a single change in the log-tail [121]. However, the expen-
sive garbage-collection needs to consistently move data around the log. NVMcached also adopts a log-structured scheme for memory management. It enables bulk removal of a lot of KV items with a single cache line write (to increment the version number), which is much more efficient than above solutions. By replacing not-recently-used items, NVMcached greatly reduces the cost in garbage collection.

5.4.2 Persistent Data Structures

Write-Atomic B⁺-Tree enables single-write metadata update, which uses a single flush instead of using expensive logging/journaling for each update in the tree [37]. CDDS is a similar design that employs versioning to enable atomic metadata update in B⁺-tree [131]. NV-Tree allows the keys in each B⁺-Tree node to stay unsorted to enable atomic metadata update [147]. Among these works, the atomic operations (using only one cache flush) only maintain integrity of the B⁺-tree itself, the actual data—the keys and also the values—are not protected by the B⁺-tree. As a result, the system has to pay additional cost for maintaining consistency and persistency. NVMcached is a complete design of a KV cache system that eliminates most expensive flushes except for the unavoidable ones for deletion and invalidation.

5.4.3 Hardware Supports for NVM

While battery-backed-up DRAM has been proposed for decades [34, 44], it requires the operating system to flush the dirty cache lines to the non-volatile storage at power loss. As a result, software failure or system crash would still lead to incomplete writes to NVM and application has to explicitly enforce data consistency by itself.

The commodity x86 CPUs provide essential but limited support for NVM. CLFLUSH instruction forces writing a cache line to memory and invalidates that cache line [75]. In addition, memory fences (such as MFENCE instruction) should be used to enforce order between writes. The use of CLFLUSH has significant impact on workloads with strong locality because cache-line write-backs always lead to their invalidation. A recently proposed instruction (CLWB) enables write-back of a cache line without invalidation [75]. However, our experiment suggests that even with weak locality, explicit cache write-back still introduces significant overhead (see Figure 5.1). NVMcached leverages the unique property of KV cache to remove the need of flushes except for deletion and invalidation.

5.5 Summary

We introduce NVMcached, a KV cache for non-volatile memory that can significantly reduce expensive cache flushes. This is achieved by a systematical design that uses
checksums to enable efficient data integrity checking and memory zones for bulk data removal while preserving access locality history after crash. NVMcached can improve the system performance by up to $2.85 \times$ for real-world workloads, while preventing the miss-ratio spike after a system crash and restart.
CHAPTER 6
CONCLUSIONS AND FUTURE WORK

The goal of this dissertation is to identify root causes of performance bottlenecks in current KV systems and to remove them by developing new systems of much improved efficiency. We aim to enable an efficient and easy-to-use infrastructure for today’s demanding applications. The methodology in this dissertation is to comprehensively re-examine all layers of a computer system in the context of today’s hardware and software trends and to experimentally reveal the issues and to explore room for improvements. This dissertation covers system aspects from CPU cache [139], DRAM [144], to the storage devices (HDDs and SSDs) [143], and trending technology (byte-addressable NVM) [140]. In the rest of this chapter I will first summarize the contributions of this dissertation and then discuss the directions of future work.

6.1 Contributions
The contributions of this dissertation can be summarized as follows:
- We identify the issue of false temporal locality and false spatial locality in the use of major index data structures and design Search Lookaside Buffer to improve index search by removing the false localities.
- We design zExpander to overcome the barriers on employing compression in KV caches to serve the purpose of reducing misses in KV cache by increasing effective cache size.
- We propose a trie-based data structure, named as LSM-trie, to address the issue of high write-amplification in LSM-tree based KV stores, which enables fast data accesses in large KV stores.
- We design NVMcached, a persistent KV cache on NVM. It removes all expensive FLUSH operations in a persistent KV cache and it enables low-cost GC by integrating cache replacement with garbage collection.

6.2 Future Work
In the next decade we are going to witness rapid change of almost every aspect of computer systems ranging from processor, memory, and network to storage technologies. Every year new hardware and software will be developed and deployed. Today’s design
guidelines and assumptions will have to be redefined to adapt to tomorrow’s changes. My future research will base on the background of computer systems and with a focus on exploring new possibilities and designing new systems to meet trending and practical needs.

6.2.1 Redesigning stores and caches for big data

One research focus I find especially exciting is redesigning NoSQL stores and caches to meet the needs of today’s big-data computing. To facilitate big-data computing, NoSQL systems must be able to manage massive amount of data for efficient storage and retrieval. The rapid improvement of hardware’s capabilities has created many opportunities and also challenges to the existing designs of NoSQL systems. Fast storage and network techniques will offer very high capacity and speed. Unlike traditional NoSQL designs that usually focus on bottlenecks at the slow hardware, new systems of high demands now are more likely to see bottlenecks scattered over many layers in the software stack spanning across systems and applications. In general I am interested in revisiting classic NoSQL designs and building new ones to meet the ever growing demands. I plan to extend my research to support high-performance transactional processing and efficient migration for NoSQL stores. I am also interested in exploring energy efficiency in key-value services for resource-constrained environments.

6.2.2 Supporting new applications with NoSQL systems

NoSQL systems have already shown their potential to support high performance applications in various areas. For example, deep learning systems use high-performance KV stores, such as LMDB, to organize training samples. Time-series processing in IoT applications rely on NoSQL stores and caches to organize the unstructured data generated from massive sensors. However, the full potential of NoSQL systems has not yet tapped. On one hand, today’s programs still heavily rely on an address-based model to manage their data structures, such as virtual memory for data on DRAM and file abstractions for data on disks. On the other hand, SQL databases process relational data organized in their own data structures, using a unified interface for users to access. NoSQL databases sit in between of these two schemes. I believe with the support of new hardware and efficient software designs, NoSQL data storage and access model can be extended to be a general programming abstraction and to serve as a building block for applications demanding both rich functionalities and high performance. I am interested in analyzing demands from various important applications and enriching NoSQL systems to meet their needs.
6.2.3 Leveraging Virtualization for high-performance big-data computing

Virtualization has been an enabling technique for cloud computing by consolidating services while providing sufficient protections through isolation. While big data computing usually runs on a large cluster of servers, virtualization can offer several key features that are very important to big-data applications. This includes effective sandboxing for computing tasks, dynamic provisioning of computing resources, and transparent support of fault-tolerance. However, employing virtualization for performance-sensitive big-data tasks is challenged by the performance degradation due to expensive abstractions. In the Selfie and TotalCOW projects [141, 142], we identified and addressed a few critical performance bottlenecks in virtual block devices and overlay file systems, respectively. The ultimate goal of my research is to reduce or even remove the abstraction overhead of virtualization so that its rich functionalities can be available to large-scale big-data computing with little performance compromise.
REFERENCES


Xingbo Wu was born in Beijing, China, in 1986. He received his B.S. degree of Computer Science and Engineering from Beihang University, Beijing, China, July 2009. He received his Ph.D. degree of Computer Engineering from The University of Texas at Arlington, June 2018. His research interests include high-performance key-value stores and caches, index data structures, storage and file systems, operating systems, and virtualization.