Resource-Efficient Data-Intensive System Designs for High Performance and Capacity

Hyeontaek Lim

CMU-CS-15-132

September 2015

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
David G. Andersen, Chair
Michael Kaminsky, Intel Labs
Andrew Pavlo
Eddie Kohler, Harvard University

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation under grant numbers ANI-0331653, CCF-0964474, CNS-1040801, and CNS-1345305, and the US Army Research Office under grant number W911NF0910273. Hyeontaek Lim was in part supported by the Korea Foundation for Advanced Studies and by the Facebook Fellowship.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Log Structure, Hash Table, Indexing, Concurrent Data Structure, I/O, Remote Procedure Call

Abstract

Data-intensive systems are a critical building block for today's large-scale Internet services. These systems have enabled high throughput and capacity, reaching billions of requests per second for trillions of items in a single storage cluster. However, many systems are surprisingly inefficient; for instance, memcached, a widely-used in-memory key-value store system, handles 1–2 million requests per second on a modern server node, whereas an optimized software system could achieve over 70 million requests per second using the same hardware. Reducing such inefficiencies can improve the cost effectiveness of the systems significantly.

This dissertation shows that by leveraging modern hardware and exploiting workload characteristics, data-intensive storage systems that process a large amount of fine-grained data can achieve an order of magnitude higher performance and capacity than prior systems that are built for generic hardware and workloads. As examples, we present SILT and MICA, which are resource-efficient key-value stores for flash and memory. SILT provides flash-speed query processing and 5.7X higher capacity than the previous state-of-the-art system. It employs new memory-efficient indexing schemes including ECT that requires only 2.5 bits per item in memory, and a system cost model built upon new accurate and fast analytic primitives to find workload-specific system configurations. MICA offers 4X higher throughput over the network than previous in-memory key-value store systems by performing efficient parallel request processing on multi-core processors and low-overhead request direction with modern network interface cards, and by using new key-value data structures designed for specific workload types.

Acknowledgments

I am very privileged to have David G. Andersen and Michael Kaminsky as my advisors. Dave is a computer-science alchemist who can convert even a complicated problem into a simple puzzle and answer it with a succinct solution almost instantly. Many important ideas in this dissertation would have not been born without his magic. I appreciate Michael for imbuing my work with rigorousness that has helped realizing my ideas. It is fortunate for me to have been able to work with them throughout my PhD study.

Andy Pavlo and Eddie Kohler have provided valuable feedback to make this dissertation possible to exist.

The Parallel Data Lab and the systems and networking lunch seminar series have helped me improve my work and communicate my ideas. I specially thank Garth Gibson, Greg Ganger, Peter Steenkiste, and Srinivasan Seshan. So long, and thanks for all the pizza.

A large fraction of my most memorable time has been with my colleges at Carnegie Mellon University. Bin Fan's analytic thinking has shed light on how mathematics can depict complex systems with simple expressions. Dongsu Han has been always willing to share his ambitious vision and inspire me. Vyas Sekar mentored me during my early PhD study. I thank Yoshihisa Abe, Athula Balachandran, Henggang Cui, Fahad Dogar, Charlie Garrod, Kiryong Ha, Philgoo Han, Aaron Harlap, Q Hong, Junchen Jiang, Anuj Kalia, U Kang, Gunhee Kim, Jin Kyu Kim, Soonho Kong, Aapo Kyrölä, Jeehyung Lee, Seunghak Lee, Soo Bum Lee, Suk-Bok Lee, Conglong Li, Mu Li, Michel Machado, Iulian Moraru, Matthew K. Mukerjee, David Naylor, George Nychis, Jun Woo Park, Swapnil Patil, Richard Peng, Amar Phanishayee, Kai Ren, Wolfgang Richter, Raja R. Sambasivan, Vivek Seshadri, Dafna Shahaf, Julian Shun, Alexey Tumanov, Vijay Vasudevan, Xiaohui Wang, Gabriel Weisz, Lin Xiao, Lianghong Xu, Erik Zawadzki, Huanchen Zhang, Dong Zhou, Timothy Zhu, and all the others for sharing their invaluable insight with me. People in the eXpressive Internet Architecture project have helped me deepen my knowledge on large-scale networking and enabled my work involving fast network processing.

I appreciate Srikanth Kandula and Peter Bodik for being as great mentors while I was at Microsoft Research.

My friends at KAIST are hidden contributors to my PhD study. They helped me decide to study in the US and make it possible.

Deborah A. Cavlovich and Mor Harchol-Balter have been one of the greatest supporters in the Computer Science Department and helped me complete my PhD study. Kathy McNiff, Karen Lindenfelser, Joan Digney, and Angela Miller have streamlined my activities in the Computer Science Department and the Parallel Data Lab.

I cannot thank my parents and brother more. They ignited my interest in computer science with early access to computers, and granted me the best educational environment and resources. Their love has been the most important ingredient in my life.

This dissertation includes and extends prior published work:

- Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, October 2011 [91].
- Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Practical batch-updatable external hashing with sorting. In *Proceedings of Meeting on Algorithm Engineering and Experiments (ALENEX)*, January 2013 [92].
- Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2014 [93].

Many parts of this dissertation have benefited from contributions by numerous people. The following is a partial list of such contributions:

- David G. Andersen and Michael Kaminsky have contributed to the aforementioned work.
- Bin Fan has contributed to the SOSP version of the SILT, in particular to the design and implementation of partial-key cuckoo hashing and the analysis of the multi-store architecture of SILT.
- Dongsu Han has contributed to the NSDI version of MICA, in particular to the motivation of the work.
- Anonymous reviewers of SOSP, ALENEX, and NSDI have provided invaluable feedback to improve the published work.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

As a core component of today's large-scale Internet services, *data-intensive* systems process and store a large amount of data on many cluster nodes in a distributed manner. At the same time, many of them waste a comparably large amount of system resources that could have been converted into useful work. In many previous systems, the individual node performance is less than 25% of what can be achieved by highly optimized software designs running on the same hardware. This dissertation aims to answer why such inefficiencies occur and how to avoid them.

Data-intensive systems often consist of thousands of nodes [108]. Building a cluster would require a multi-million dollar investment for the equipment alone. Their tremendous capabilities—being able to handle petabytes of data [31] and process billions of requests per second for trillions of items [108]—have enabled Facebook to serve almost a billion users every day by 2015 [57].

Yet, prior data-intensive systems fail to achieve high efficiency. The efficiency of a system for a task is the ratio of the system capability for the task to the amount of the low-level system resources that are the most significant bottlenecks in fulfilling the task. memcached [99], a widely-used networked in-memory storage system, handles up to 1–2 million operations per second on a single node. This system is inefficient for query processing because the underlying hardware can achieve about 78 million I/Os per second where I/O is one of the most critical bottlenecks for key-value query processing. On the contrary, a highly efficient system would achieve higher performance that approaches this packet I/O speed (performance that is achievable, as demonstrated in this thesis).[1]

A key factor in the inefficiencies of prior data-intensive systems is that their design is mostly agnostic of the hardware and workloads. They typically use unified software designs that run on diverse hardware and for different workloads without major modifications. Ensuring uniformity has attractive benefits such as easy and cheap software development, deployment, and maintenance. However, such a design has important consequences in the context of data-intensive systems: The huge scale of data-intensive systems using many nodes will significantly amplify even the slightest overhead, compromising their cost effectiveness.

[1] It is important to note the difference between efficiency and (raw) performance. memcached would have been said to have good efficiency if the underlying system could perform only a few million packet I/Os per second. Similar observations should be made for systems using different storage devices—e.g., DRAM, flash, and hard disk drives.

To reduce inefficiencies, we argue that it is necessary to (1) *leverage modern hardware* and (2) *exploit workload characteristics*. Modern hardware offers new opportunities including multi-core processors, specialized CPU instructions, new storage devices, and advanced NIC (network interface card) features. Taking advantage of these opportunities requires extensive effort and great care because they are often exposed via unconventional interfaces; using them naïvely could degrade performance. Applying optimizations can become effective under specific workloads with certain properties; workload skew is typically considered harmful because it can create hot spots and load imbalance, but it can also improve system performance when exploited properly.

This dissertation shows that storage systems that store and process large-scale, fine-grained data can provide an order of magnitude higher performance and capacity than conventional systems that are built for generic hardware and workloads. Such improvements can be achieved by using new algorithms, data structures, and software architectures that leverage modern hardware and exploit workload characteristics.

We devise and evaluate new techniques to do so, ranging from efficient indexing algorithms to full key-value (hash table-like) store designs and implementations:

- **SILT** improves the capacity of unordered on-flash key-value storage, realizing 5.7X higher memory efficiency than the state-of-the-art system at that time while preserving high query performance. Key design components of SILT include the combination of multiple stores with different I/O and indexing characteristics, and compact data structures that realize memory-efficient indexing.
- **ECT** provides SILT with a memory-efficient indexing scheme. It requires only 2.5 bits per key in memory to index hashed keys, which is 36% smaller than the best external perfect hashing scheme at that time. ECT possesses a desirable property of using a fully sorted array that simplifies the incremental merge of large arrays.
- **Accurate and fast analytic primitives** provide tools to build models of SILT and SILT-like systems. The system model allows estimating performance metrics on the scale of seconds. The accuracy and speed of the estimation facilitates exploiting the skew of workloads by finding appropriate system parameters, which takes only up to a few minutes for a large workload containing hundreds of millions of data items.
- **MICA** increases the throughput of networked, unordered in-memory key-value storage, achieving 4X higher speed than the state-of-the-art system at that time. MICA avoids expensive concurrent writes across multiple cores and uses advanced NIC features for the hardware to deliver client requests directly to appropriate server cores for key-value processing. MICA uses different key-value data structures that are designed for different key-value workloads.

The rest of this chapter introduces the background of data-intensive systems, examines modern hardware trends, discuss opportunities and challenges the trends impose, and outlines how this dissertation tackles problems that arise therein.

1.1 Data-Intensive Systems

The rapid growth of Internet services in the 2000s has required computer systems to handle an explosively increasing amount of data for acquiring, storing, and processing. Today, production-level services exist that handle petabytes of data [31] and process billions of requests per second for trillions of items [108]. Services such as Facebook have evolved to serve almost a billion users every day in 2015 [57].

A core component of large-scale Internet services is *data-intensive systems* that process and store a large number of fine-grained data items, such as web pages and web analytics [31], product catalogs and sales rank, user preferences, shopping carts and sessions [44], and database query results [108]. These tasks handle a vast amount of data items in comparison to the small amount of computation applied to each item. In this context, efficient data storage and I/O for fine-grained items is extremely important. Furthermore, most data items are independent of each other, enabling *data parallelism* that makes it easy for systems to process individual data items in a distributed manner. Data-intensive systems employ a large-scale clustered architecture that builds a cluster of many nodes to achieve *high aggregate throughput and capacity*.

Data-intensive systems are not new in the types of requests they can handle, but are distinguished by how efficiently they can handle the requests. Traditional DBMS (database management systems) could process the same types of requests data-intensive systems do. Large-scale Internet services do not make more complex requests than relational database queries; the Internet services rather make simpler requests—they are often as simple as hash table operations in essence. In this simpler but higher volume setting, the sheer amount of data and data parallelism make data-intensive systems more favorable for achieving the required scale in a cost-effective manner.

In addition to meeting high performance and capacity goals, data-intensive systems have two important goals: *cost effectiveness* and *scalability*.

The scale of data-intensive systems makes them expensive to provision and operate. A single cluster consisting of thousands of nodes would cost tens of millions of dollars for the equipment, and running the large cluster will incur high electricity and cooling costs plus many other management costs. Thus, the system architects should keep the total cost, including the equipment, operation, and risk management cost, as low as possible while satisfying other important goals. Any small increase in the cost of a single node can be amplified to threaten the cost effectiveness of the entire cluster; on the other hand, a few percent of improvement in performance and capacity can significantly reduce the total cost of data-intensive systems.

Data-intensive systems are expected to scale well. Demands for processing and storage of data vary over time, and additional applications may begin to use existing data-intensive systems. The system should be able to offer higher performance and capacity by scaling horizontally ("scale-out") as the demands increase.

System designers, however, should be careful about avoiding side effects that arise in a large cluster, to ensure *load balance*, *fault tolerance and high availability*, and *low tail latency*.

Load balance affects how evenly data-intensive systems distribute the work to cluster nodes. In an ideal scenario, every node uses the same amount of local system resources, such as processing power and storage space. This equal load allows achieving high utilization of each node's power and reducing wasted resources. Many practical workloads, however, contain workload skew (nonuniform demands for different data items) and flash crowds (traffic spikes; sudden increases

in load) [8], which makes it difficult to distribute the load evenly across nodes. Load balance is increasingly difficult with many slow nodes in the cluster because each node can easily become a "hotspot" that experiences excessive load that the node cannot handle.

Data-intensive systems are built upon many nodes that can fail individually. As a cluster uses more nodes, it experiences more frequent node failures every hour or day. The whole cluster may suffer downtime and even lose the data permanently if the failure propagates to other nodes or is not resolved quickly, which can cost the system operator a huge amount of money and time. Ideally, the system as a whole should be always available to the user regardless of how distributed the load is across multiple nodes, while this goal is very challenging for large clusters with many nodes.

Many applications demand low tail latency (the high-percentile response time) while it is difficult to achieve it in a large cluster. Even a half a second increase in the response time of interactive services can cause traffic and revenue to drop by 20% [70]. Applications therefore attempt to make the maximum response time no higher than a certain threshold (e.g., 100 ms) [82]. The tail latency becomes worse as a cluster uses more nodes [41]. This phenomenon occurs because there exists variance in the response time of multiple nodes, and an application must often wait until the last response arrives before proceeding; using more nodes increases the variance and thus increases the chance of experiencing high response time with one of the nodes.

In this dissertation, we aim to preserve the strengths of data-intensive systems while reducing their side effects. In particular, we explore how we can improve individual systems to offer higher performance and capacity.

By boosting the capability of individual nodes, we hope to obtain several desirable direct and indirect benefits. The most direct consequence is that the system is more robust to workload skew and flash crowds; each node can handle more load, and this spare capability can reduce the chance of creating a hotspot. With individually powerful nodes, the system can use a fewer number of nodes to provide the same aggregate performance and capacity as before; with the smaller cluster size, the system can expect (1) lower equipment and operational cost, (2) easier failure recovery and higher availability, and (3) low tail latency.

Improving the performance or efficiency of individual nodes is a time-tested goal in computer science. Faster processors and larger storage space have for decades allowed processing more requests and storing more data on a single system. However, modern hardware and workload trends necessitate a new approach to improving node capabilities, as we explain in the next two sections.

1.2 Challenges and Opportunities with Modern Hardware

Moore's law described—and, indeed, prescribed, as industry scrambles to fulfill it—a promising vision of hardware evolution: The number of transistors per integrated circuit doubles about every two years [104]. Along with this growth, Dennard scaling predicts that the power density of transistors remain similar as they shrink [45]. An implication from two laws is that processors required roughly the same power on the same area while providing exponentially increasing computational power. Chip designers capitalized on this to boost clock rates while preserving the

TDP (thermal design power; the amount of heat generated by a processor) by reducing the chip size.

Unfortunately, Dennard scaling has slowed considerably. As transistors become smaller, they experience more current leakage caused by tunneling current [24]. Chips began to generate too much heat to be cooled efficiently. This physical limitation of high-density chips is referred to as the "power wall" [129]. The breakdown of Dennard scaling made it no longer viable to increase the clock rate to obtain fast processors. "The free lunch is over" [132]—software can no longer enjoy the clock rate increase that previously brought automatic performance improvements.

Two efforts have been made to mitigate the effect of the power wall. *Multi-core architectures* introduce multiple sets of processing logic in a single chip package. Multi-core processor designs improve the total processing power of a single chip for parallel processing without increasing the clock rate. Processors are also increasingly equipped with *advanced processing features* such as an extended instruction set for vectorized operations and specialized computation such as data encryption and CRC32 checksumming [78, 80]. A corollary of these efforts is "dark silicon," which activates only a portion of the chip to limit the overall power density [51]. Unfortunately, it is difficult to take advantages of these new changes because programming with multi-core processors requires great caution to avoid contention at shared resources, and different CPU implementations can exhibit different performance characteristics for concurrency controls [40].

As the processors become more powerful, the discrepancy grows between the processor speed and the memory access latency. Upon a cache miss (an attempt to access data that does not currently reside in the CPU cache), the processor can experience a "stall" while waiting for the memory controller to fetch the data from the main memory to the CPU cache. Each stall wastes a significant number of CPU cycles because of an increasing gap between the CPU and memory speed—the process speed has improved by about 75% per year while the memory speed has improved by only about 7% per year [97]. To mitigate this "memory wall" [146], today's processors are equipped with a large cache whose size is huge compared to previous generations' (tens of MBs vs. a couple of MBs). More cache unfortunately does not solve the memory access problem for applications whose working set does not fit in the cache, which is common in data-intensive systems. For example, main-memory storage systems often have a huge working set size of GBs, which is far beyond the cache size, as certain workloads require them to actively access the entire stored data. Therefore, the intelligent use of *prefetch instructions*, which give the processor a hint before the data is accessed [30], has become more critical to the performance of such applications than ever. Nevertheless, it is difficult to make effective use of prefetching because the system designer needs in-depth knowledge about how data is accessed and processed in the system; prefetching must be requested before the data access with an appropriate time gap (about the memory access latency) to make prefetching effective and avoid cache pollution that increases cache misses for other frequently accessed data.

New storage technologies have blossomed, raising questions for system balance. *SSD (solid-state drives) based on flash* first became practical in the 2000s and began to replace HDDs (hard disk drives) [6]. They are persistent, have very low random access time of tens of microseconds as secondary storage (c.f., a few milliseconds for HDDs), and have good sequential transfer speed of hundreds of MBs per second. Their downsides include costing about ten times more per unit storage space than HDDs; a wear-out property that makes the drive unusable after excessive writes; and relatively slow random write speed because SSDs lack support for block-granularity

data overwrite and require coarse-grained erasure operations. The storage space offered by an SSD has increased rapidly, exceeding the increase rate of the DRAM capacity; this leads to burden systems that accelerate query processing by spending the memory proportional to the total data size on flash.

Fast network devices have evolved as well. High-speed commodity NICs (network interface cards) offer up to 56 Gbps speed in the mid-2010s [98]. It has become critical to use multiple cores to perform I/O because a single core does not provide enough processing power to perform I/O at such a high speed [46]. Therefore, modern NICs implement *multiple receive and transmit queues*. Each queue can be accessed by a core to allow parallel network I/O across cores. NICs with multiple queues commonly include advanced features such RSS (receive-side scaling) to automatically distribute incoming packets across multiple receive queues [46], and more programmable features such as "flow steering" allow the software to supply packet classification rules for the packet distribution [116]. These features are often underutilized in the context of data-intensive systems because they are designed for other contexts (e.g., flow-oriented communication). To use new NIC features, a data-intensive system must close the gap between the hardware feature and the application context.

We speculate that these hardware trends will continue, and become more significant in the near future. The introduction of new hardware will expose previous system designs to different challenges while offering new opportunities to further improve their performance and capacity.

1.3 Diversity in Workloads of Data-Intensive Systems

Many data-intensive systems commonly expose a simple and generic interface to a variety of target applications. Bigtable provides the same API to many applications including web document crawlers, web analytics, and Google Earth [31]. Similarly, Dynamo [44] has a simple interface of storing and retrieving a key-value pair; this interface is used in various areas, as we revisit it in the next section. With the shared infrastructure, system operators can facilitate the provisioning, maintenance, and development of the infrastructure.

While multiple applications share the same interface, they can impose different workload characteristics to the storage system. Prior study on Facebook's production memcached deployment [8] reveals that there exists a huge difference in the length of keys and values as well as their access patterns in different usages of the key-value storage. A pool storing the user-account status information has tiny key-value items (16 B or 21 B keys and 2 B values) is highly read-intensive (99.8% of operations are reads). On the other hand, a pool storing the browser information handles more varied key-value item sizes and serves 3X more writes than reads.

Using similar interfaces for different workloads does not necessarily result in the same storage and processing efficiency under the workloads. The workloads have different performance requirements and different types of data and requests. For example, applications using a storage system as an in-memory cache storing DBMS query results would expect high throughput and low latency that are comparable to the memory speed (e.g., millions of operations per second, hundreds of microseconds of response time), whereas applications using a storage system to keep persistent data may be more interested in storing a high number of items (e.g., billions of items) on the same physical storage space. A storage system optimized for read-intensive workloads may

perform poorly under write-heavy workloads. Workloads may also have different skew properties; some workloads require uniform access to stored items, while others access a small set of items more frequently, creating skewed access patterns. Ignoring the skew may lose an opportunity to adjust the behavior of the storage system and benefit from side effects of the skew (e.g., being able to deduplicate redundant items that occur frequently under high skew).

Considering such differences in workloads can make more balanced use of available system resources and improve system capabilities without requiring more powerful hardware. The system can introduce techniques that exhibit a high efficiency under certain characteristics implied by the target workload. Using carefully tuned system parameters for the specific workload can improve performance even when the system implementation remains unchanged. Similar to exploiting modern hardware, these techniques require effort to understand workload characteristics and apply them to the system design.

1.4 Key-Value Stores

We use *key-value stores* as examples to demonstrate how we can achieve high performance and capacity by leveraging modern hardware and workload characteristics.

Key-value stores are high-performance, highly available, and scalable storage services with a simple abstraction allowing data-parallel system designs with a cluster design [44]. Key-value stores expose a hash table-like interface, such as `SET(key,value)` (store a key-value pair in the system) and `GET(key)` (retrieve the value associated with the key), to clients either locally and remotely. Key-value stores have become important in diverse areas, from their initial applications to web object caching [8, 44, 108] to more recent applications such as state storage for machine learning [1, 8].

Despite their importance as a building block for large services, conventional key-value store designs struggle in achieving high efficiency. Prior key-value stores are designed to run on generic hardware, missing opportunities to make efficient use of modern hardware and adapt to new hardware balance; for instance, prior key-value stores require a large amount of memory and stick to the standard socket I/O that fails to take advantage of fast packet processing assisted by modern NICs. Prior work is mostly agnostic to workload characteristics; for example, previous key-value systems use a uniform system design and parameter set regardless of the target workload. As a consequence, prior systems suffer from inefficiencies, achieving no more than about 25% of the performance and capacity achievable by new, efficient key-value store designs that we present in this dissertation.

1.5 Contribution Summary

This dissertation presents two systems, SILT and MICA, for different underlying storage devices.

SILT (Small Index Large Table) is a persistent flash-based key-value store for workloads with unordered retrievals. It provides high throughput for a large number of stored items, converting up to 99% SSD's random read speed into query performance while requiring only 0.7 bytes of memory per item; thus, SILT allows serving 5.7 times as many fine-grained items for the same

amount of the main memory as the previous state-of-the-part system without sacrificing query performance. SILT guarantees long lifetime of SSDs by combining three different key-value data structures with a different emphasis on memory efficiency and write-friendliness and balancing the use of these three components.

One important of SILT's key design components is **ECT** (Entropy-Coded Tries), a memory-efficient indexing scheme. By exploiting unordered retrievals, ECT provides a compact mapping between hashed keys and their item indices in a sorted array, using 2.5 bits of memory per item, which can be further reduced when a single storage block contains multiple small items. ECT trades computation for space by using compression; its lookup is slower than typical hash functions and binary search trees, but ECT's compression ratio can be adjusted to match the random read speed of SSDs. ECT lookups can be performed in parallel on multiple CPU cores to support fast SSDs.

We devise **accurate and fast analytic primitives** that help estimating performance metrics of SILT and SILT-like key-value stores that combine multiple stores. Models built using these primitives consider the skew in the workloads; therefore, for specific workloads, a system designer can improve the system design to better exploit the skew of the workloads and automatically tune system parameters within minutes even for large workloads with hundreds of millions of items. SILT can offer 26% better memory efficiency for a skewed workload by choosing workload-specific system parameters based on the analysis result.

MICA (Memory-store with Intelligent Concurrent Access) is a non-persistent in-memory key-value store for unordered access. MICA can serve fine-grained key-value items over the network at up to 76.9 million operations per second using a single machine, which is 99% of the packet I/O speed the machine can achieve without performing key-value processing. This throughput is 4 times as high as previous in-memory stores provide. It uses an efficient parallel architecture that avoids expensive inter-communication between CPU cores. MICA takes advantage of fast modern NIC's features and client-side assists to deliver remote client requests directly to appropriate server cores on the hardware level. For fast local key-value processing, MICA employs new data structures that are specific to the required interface semantics (cache mode and store mode).

The following gives an outline of the rest of this work:

- Chapter 2 presents SILT's multi-store design, individual store designs, describes an analytic method that determines SILT's system parameters, and examines the performance and memory efficiency of the SILT implementation.
- Chapter 3 extends the description of ECT and compares it with the state-of-the-art external hashing algorithm by building stores with them.
- Chapter 4 proposes fast and accurate analysis primitives quickly build models of SILT and SILT-like systems. Using the models, a system designer can quickly estimate system performance metrics under different workloads and optimize system parameters for specific workloads.
- Chapter 5 describes MICA's system design and new data structures, and evaluates MICA and previous in-memory key-value stores.
- Chapter 6 concludes.

Chapter 2

SILT (Small Index Large Table)

Key-value storage systems have become a critical building block for today's large-scale, high-performance data-intensive applications. High-performance key-value stores have therefore received substantial attention in a variety of domains, both commercial and academic: e-commerce platforms [44], data deduplication [5, 42, 43], picture stores [13], web object caching [9, 99], and more.

To achieve low latency and high performance, and make best use of limited I/O resources, key-value storage systems require efficient indexes to locate data. As one example, Facebook engineers recently created a new key-value storage system that makes aggressive use of DRAM-based indexes to avoid the bottleneck caused by multiple disk operations when reading data [13]. Unfortunately, DRAM is up to 8X more expensive and uses 25X more power per bit than flash, and, as Table 2.1 shows, is growing more slowly than the capacity of the disk or flash that it indexes. As key-value stores scale in both size and importance, index memory efficiency is increasingly becoming one of the most important factors for the system's scalability [13] and overall cost effectiveness.

Recent proposals have started examining how to reduce per-key in-memory index overhead [5, 6, 9, 42, 43, 107, 147], but these solutions either require more than a few bytes per key-value entry in memory [5, 6, 9, 42], or compromise performance by keeping all or part of the index on flash or disk and thus require many flash reads or disk seeks to handle each key-value lookup [43, 107, 147] (see Figure 2.1 for the design space). We term this latter problem *read amplification* and explicitly strive to avoid it in our design.

This chapter presents a new flash-based key-value storage system, called SILT (Small Index Large Table), that significantly reduces per-key memory consumption with predictable system

Metric	**2008 → 2011**	**Increase**
CPU transistors	731 → 1,170 M	60 %
DRAM capacity	0.062 → 0.153 GB/$	147 %
Flash capacity	0.134 → 0.428 GB/$	219 %
Disk capacity	4.92 → 15.1 GB/$	207 %

Table 2.1: From 2008 to 2011, flash and hard disk capacity increased much faster than either CPU transistor count or DRAM capacity.

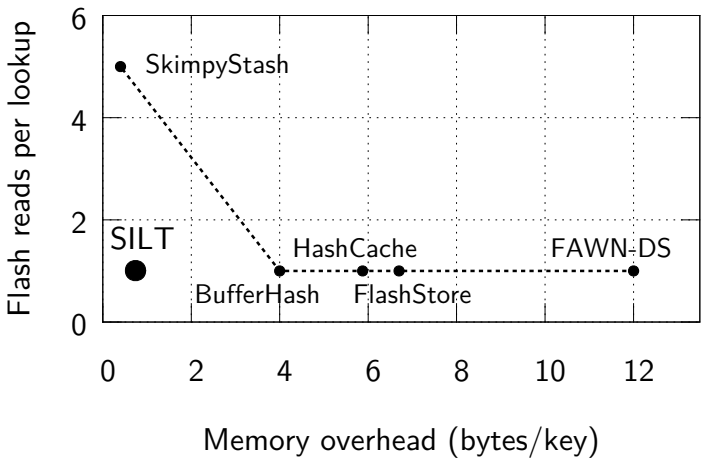


Figure 2.1: The memory overhead and lookup performance of SILT and the recent key-value stores. For both axes, smaller is better.

performance and lifetime. SILT requires approximately 0.7 bytes of DRAM per key-value entry and uses on average only 1.01 flash reads to handle lookups. Consequently, SILT can saturate the random read I/O on our experimental system, performing 46,000 lookups per second for 1024-byte key-value entries, and it can potentially scale to billions of key-value items on a single host. SILT offers several knobs to trade memory efficiency and performance to match available hardware.

This chapter makes three main contributions:

- The design and implementation of three basic key-value stores (LogStore, HashStore, and SortedStore) that use new fast and compact indexing data structures (partial-key cuckoo hashing and Entropy-Coded Tries), each of which places different emphasis on memory-efficiency and write-friendliness.
- Synthesis of these basic stores to build SILT.
- An analytic model that enables an explicit and careful balance between memory, storage, and computation to provide an accurate prediction of system performance, flash lifetime, and memory efficiency.

2.1 SILT Key-Value Storage System

Like other key-value systems, SILT implements a simple exact-match hash table interface including `PUT` (map a new or existing key to a value), `GET` (retrieve the value by a given key), and `DELETE` (delete the mapping of a given key).

For simplicity, we assume that keys are *uniformly distributed* 160-bit hash values (e.g., pre-hashed keys with SHA-1) and that data is fixed-length. This type of key-value system is widespread in several application domains such as data deduplication [5, 42, 43], and is applicable to block storage [39, 119], microblogging [53, 137], WAN acceleration [5], among others. In systems with lossy-compressible data, e.g., picture stores [13, 61], data can be adaptively compressed

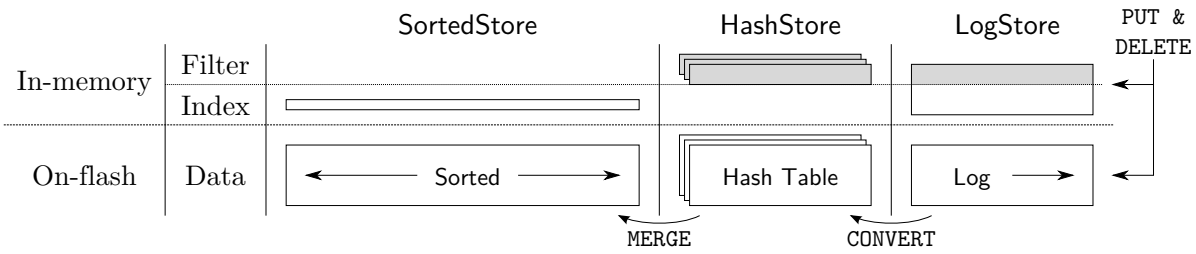


Figure 2.2: Architecture of SILT.

to fit in a fixed-sized slot. A key-value system may also let applications choose one of multiple key-value stores, each of which is optimized for a certain range of value sizes [44]. We discuss the relaxation of these assumptions in Section 2.3.

Design Goals and Rationale The design of SILT follows from five main goals:

1. **Low read amplification**: Issue $1+\varepsilon$ flash reads in expectation for a single `GET`, where ε is configurable and small (e.g., 0.01).
   Rationale: Random reads remain the read throughput bottleneck when using flash memory. Read amplification therefore directly reduces throughput.
2. **Controllable write amplification and favoring sequential writes**: It should be possible to adjust how many times a key-value entry is rewritten to flash over its lifetime. The system should issue flash-friendly, large writes.
   Rationale: Flash memory can undergo only a limited number of erase cycles before it fails. Random writes smaller than the SSD log-structured page size (typically 4 KiB[1]) cause extra flash traffic.

   Optimizations for memory efficiency and garbage collection often require data layout changes on flash. The system designer should be able to select an appropriate balance of flash lifetime, performance, and memory overhead.
3. **Memory-efficient indexing**: SILT should use as little memory as possible (e.g., less than one byte per key stored).
   Rationale: DRAM is both more costly and power-hungry per gigabyte than Flash, and its capacity is growing more slowly.
4. **Computation-efficient indexing**: SILT's indexes should be fast enough to let the system saturate the flash I/O.
   Rationale: System balance and overall performance.
5. **Effective use of flash space**: Some data layout options use the flash space more sparsely to improve lookup or insertion speed, but the total space overhead of any such choice should remain small – less than 20% or so.
   Rationale: SSDs remain relatively expensive.

[1]For clarity, binary prefixes (powers of 2) will include "i", while SI prefixes (powers of 10) will appear without any "i".

	SortedStore (Section 2.2.3)	**HashStore** (Section 2.2.2)	**LogStore** (Section 2.2.1)
Mutability	Read-only	Read-only	Writable
Data ordering	Key order	Hash order	Insertion order
Multiplicity	1	≥ 0	1
Typical size	$> 80\%$ of total entries	$< 20\%$	$< 1\%$
DRAM usage	0.4 bytes/entry	2.2 bytes/entry	6.5 bytes/entry

Table 2.2: Summary of basic key-value stores in SILT.

In the rest of this section, we first explore SILT's high-level architecture, which we term a "multi-store approach", contrasting it with a simpler but less efficient single-store approach. We then briefly outline the capabilities of the individual store types that we compose to form SILT, and show how SILT handles key-value operations using these stores.

Conventional Single-Store Approach A common approach to building high-performance key-value stores on flash uses three components:

1. *an in-memory filter* to efficiently test whether a given key is stored in this store before accessing flash;
2. *an in-memory index* to locate the data on flash for a given key; and
3. *an on-flash data layout* to store all key-value pairs persistently.

Unfortunately, to our knowledge, no existing index data structure and on-flash layout achieve all of our goals simultaneously. For example, HashCache-Set [9] organizes on-flash keys as a hash table, eliminating the in-memory index, but incurring random writes that impair insertion speed. To avoid expensive random writes, systems such as FAWN-DS [6], FlashStore [42], and SkimpyStash [43] append new values sequentially to a log. These systems then require either an in-memory hash table to map a key to its offset in the log (often requiring 4 bytes of DRAM or more per entry) [6, 43]; or keep part of the index on flash using multiple random reads for each lookup [43].

Multi-Store Approach BigTable [31], Anvil [95], and BufferHash [5] chain multiple stores, each with different properties such as high write performance or inexpensive indexing.

Multi-store systems impose two challenges. First, they require effective designs and implementations of the individual stores: they must be efficient, compose well, and it must be efficient to transform data between the store types. Second, it must be efficient to query multiple stores when performing lookups. The design must keep read amplification low by not issuing flash reads to each store. A common solution uses a compact in-memory filter to test whether a given key can be found in a particular store, but this filter can be memory-intensive—e.g., BufferHash uses 4–6 bytes for each entry.

SILT's multi-store design uses a series of *basic key-value stores*, each optimized for a different purpose.

1. Keys are inserted into a write-optimized store, and over their lifetime flow into increasingly more memory-efficient stores.
2. Most key-value pairs are stored in the most memory-efficient basic store. Although data outside this store uses less memory-efficient indexes (e.g., to optimize writing performance), the average index cost per key remains low.
3. SILT is tuned for high performance for hard cases—a lookup found in the last and largest store. As a result, SILT can avoid using an in-memory filter on this last store, allowing all lookups (successful or not) to take $1+\varepsilon$ flash reads.

SILT's architecture and basic stores (the LogStore, HashStore, and SortedStore) are depicted in Figure 2.2. Table 2.2 summarizes these stores' characteristics.

LogStore is a write-friendly key-value store that handles individual `PUT`s and `DELETE`s. To achieve high performance, writes are appended to the end of a log file on flash. Because these items are ordered by their insertion time, the LogStore uses an in-memory hash table to map each key to its offset in the log file. The table doubles as an in-memory filter. SILT uses a memory-efficient, high-performance hash table based upon cuckoo hashing [114]. As described in Section 2.2.1, our *partial-key cuckoo hashing* achieves 93% occupancy with very low computation and memory overhead, a substantial improvement over earlier systems such as FAWN-DS and BufferHash that achieved only 50% hash table occupancy. Compared to the next two read-only store types, however, this index is still relatively memory-intensive, because it must store one 4-byte pointer for every key. SILT therefore uses only one instance of the LogStore (except during conversion to HashStore as described below), with fixed capacity to bound its memory consumption.

Once full, the LogStore is converted to an immutable *HashStore* in the background. The HashStore's data is stored as an on-flash hash table that does not require an in-memory index to locate entries. SILT uses multiple HashStores at a time before merging them into the next store type. Each HashStore therefore uses an efficient in-memory filter to reject queries for nonexistent keys.

SortedStore maintains key-value data in sorted key order on flash, which enables an extremely compact index representation (e.g., 0.4 bytes per key) using a novel design and implementation of *Entropy-Coded Tries*. Because of the expense of inserting a single item into sorted data, SILT periodically merges in bulk several HashStores along with an older version of a SortedStore and forms a new SortedStore, garbage collecting deleted or overwritten keys in the process.

Key-Value Operations Each `PUT` operation inserts a `(key,value)` pair into the LogStore, even if the key already exists. `DELETE` operations likewise append a "delete" entry into the LogStore. The space occupied by deleted or stale data is reclaimed when SILT merges HashStores into the SortedStore. These lazy deletes trade flash space for sequential write performance.

To handle `GET`, SILT searches for the key in the LogStore, HashStores, and SortedStore in sequence, returning the value found in the youngest store. If the "deleted" entry is found, SILT will stop searching and return "not found."

Partitioning Finally, we note that each physical node runs multiple SILT instances, responsible for disjoint key ranges, each with its own LogStore, SortedStore, and HashStore(s). This parti-

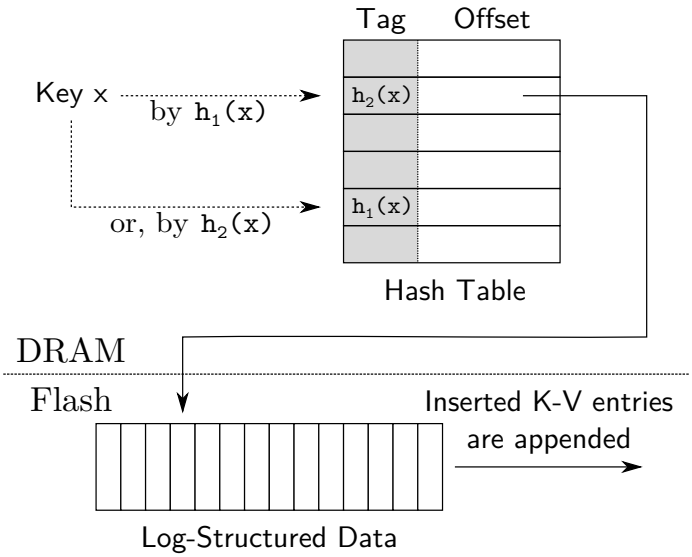


Figure 2.3: Design of LogStore: an in-memory cuckoo hash table (index and filter) and an on-flash data log.

tioning improves load-balance (e.g., virtual nodes [130]), reduces flash overhead during merge (Section 2.2.3), and facilitates system-wide parameter tuning (Section 2.4).

2.2 Basic Store Design

2.2.1 LogStore

The LogStore writes `PUT`s and `DELETE`s sequentially to flash to achieve high write throughput. Its in-memory partial-key cuckoo hash index efficiently maps keys to their location in the flash log, as shown in Figure 2.3.

Partial-Key Cuckoo Hashing The LogStore uses a new hash table based on *cuckoo hashing* [114]. As with standard cuckoo hashing, it uses two hash functions h_1 and h_2 to map each key to two candidate buckets. On insertion of a new key, if one of the candidate buckets is empty, the key is inserted in this empty slot; if neither bucket is available, the new key "kicks out" the key that already resides in one of the two buckets, and the displaced key is then inserted to its own alternative bucket (and may kick out other keys). The insertion algorithm repeats this process until a vacant position is found, or it reaches a maximum number of displacements (e.g., 128 times in our implementation). If no vacant slot found, it indicates the hash table is almost full, so SILT freezes the LogStore and initializes a new one without expensive rehashing.

To make it compact, the hash table does not store the entire key (e.g., 160 bits in SILT), but only a "tag" of the actual key. A lookup proceeds to flash only when the given key matches the tag, which can prevent most unnecessary flash reads for non-existing keys. If the tag matches, the full key and its value are retrieved from the log on flash to verify if the key it read was indeed the correct key.

Although storing only the tags in the hash table saves memory, it presents a challenge for cuckoo hashing: moving a key to its alternative bucket requires knowing its other hash value. Here, however, the full key is stored only on flash, but reading it from flash is too expensive. Even worse, moving this key to its alternative bucket may in turn displace another key; ultimately, each displacement required by cuckoo hashing would result in additional flash reads, just to insert a single key.

To solve this costly displacement problem, our *partial-key cuckoo hashing* algorithm stores the index of the alternative bucket as the tag; in other words, partial-key cuckoo hashing uses the tag to reduce flash reads for non-existent key lookups *as well as* to indicate an alternative bucket index to perform cuckoo displacement without any flash reads. For example, if a key x is assigned to bucket $h_1(x)$, the other hash value $h_2(x)$ will become its tag stored in bucket $h_1(x)$, and vice versa (see Figure 2.3). Therefore, when a key is displaced from the bucket a, SILT reads the tag (value: b) at this bucket, and moves the key to the bucket b without needing to read from flash. Then it sets the tag at the bucket b to value a.

To find key x in the table, SILT checks if $h_1(x)$ matches the tag stored in bucket $h_2(x)$, or if $h_2(x)$ matches the tag in bucket $h_1(x)$. If the tag matches, the `(key,value)` pair is retrieved from the flash location indicated in the hash entry.

Associativity Standard cuckoo hashing allows 50% of the table entries to be occupied before unresolvable collisions occur. SILT improves the occupancy by increasing the associativity of the cuckoo hashing table. Each bucket of the table is of capacity four (i.e., it contains up to 4 entries). Our experiments show that using a 4-way set associative hash table improves space utilization of the table to about 93%,[2] which matches the known experimental result for various variants of cuckoo hashing [49]; moreover, 4 entries/bucket still allows each bucket to fit in a single cache line.[3]

Hash Functions Keys in SILT are 160-bit hash values, so the LogStore finds $h_1(x)$ and $h_2(x)$ by taking two non-overlapping slices of the low-order bits of the key x.

By default, SILT uses a 15-bit key fragment as the tag. Each hash table entry is 6 bytes, consisting of a 15-bit tag, a single valid bit, and a 4-byte offset pointer. The probability of a false positive retrieval is 0.024% (see Section 2.4 for derivation), i.e., on average 1 in 4,096 flash retrievals is unnecessary. The maximum number of hash buckets (not entries) is limited by the key fragment length. Given 15-bit key fragments, the hash table has at most 2^{15} buckets, or $4 \times 2^{15} = 128$ Ki entries. To store more keys in LogStore, one can increase the size of the key fragment to have more buckets, increase the associativity to pack more entries into one bucket, and/or partition the key-space to smaller regions and assign each region to one SILT instance with a LogStore. The tradeoffs associated with these decisions are presented in Section 2.4.

[2]Space utilization here is defined as the fraction of used entries (not used buckets) in the table, which more precisely reflects actual memory utilization.

[3]Note that, another way to increase the utilization of a cuckoo hash table is to use more hash functions (i.e., each key has more possible locations in the table). For example, FlashStore [42] applies 16 hash functions to achieve 90% occupancy. However, having more hash functions increases the number of cache lines read upon lookup and, in our case, requires more than one tag stored in each entry, increasing overhead.

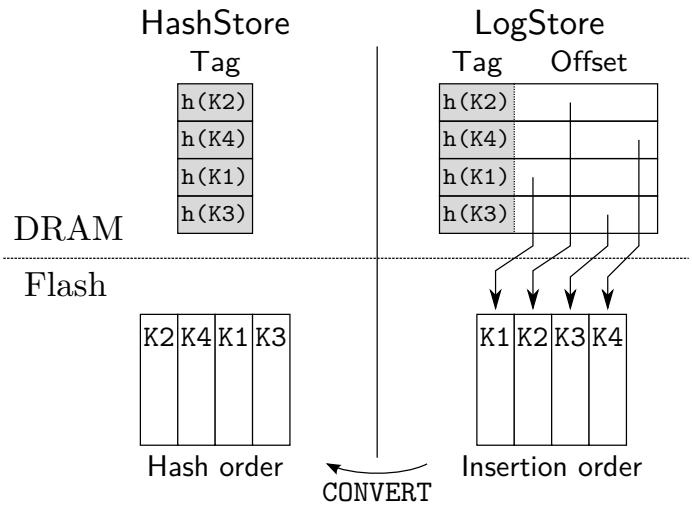


Figure 2.4: Convert a LogStore to a HashStore. Four keys K1, K2, K3, and K4 are inserted to the LogStore, so the layout of the log file is the insert order; the in-memory index keeps the offset of each key on flash. In HashStore, the on-flash data forms a hash table where keys are in the same order as the in-memory filter.

Note that the latest version of SILT use improved partial-key cuckoo hashing [58]. In this version, the alternative bucket index of an item is obtained by applying `XOR` to the current bucket index and the hash of the tag. The tag is unchanged when moving the item between its two alternative locations. This effectively removes the size limit of the hash table imposed by the tag size, allowing more flexible system configurations.

2.2.2 HashStore

Once a LogStore fills up (e.g., the insertion algorithm terminates without finding any vacant slot after a maximum number of displacements in the hash table), SILT freezes the LogStore and converts it into a more memory-efficient data structure. Directly sorting the relatively small LogStore and merging it into the much larger SortedStore requires rewriting large amounts of data, resulting in high write amplification. On the other hand, keeping a large number of LogStores around before merging could amortize the cost of rewriting, but unnecessarily incurs high memory overhead from the LogStore's index. To bridge this gap, SILT first converts the LogStore to an immutable HashStore with higher memory efficiency; once SILT accumulates a configurable number of HashStores, it performs a bulk merge to incorporate them into the SortedStore. During the LogStore to HashStore conversion, the old LogStore continues to serve lookups, and a new LogStore receives inserts.

HashStore saves memory over LogStore by eliminating the index and reordering the on-flash `(key,value)` pairs from insertion order to hash order (see Figure 2.4). HashStore is thus an on-flash cuckoo hash table, and has the same occupancy (93%) as the in-memory version found in LogStore. HashStores also have one in-memory component, a filter to probabilistically test whether a key is present in the store without performing a flash lookup.

Memory-Efficient Hash Filter Although prior approaches [5] used Bloom filters [22] for the probabilistic membership test, SILT uses a hash filter based on partial-key cuckoo hashing. Hash filters are more memory-efficient than Bloom filters at low false positive rates. Given a 15-bit tag in a 4-way set associative cuckoo hash table, the false positive rate is $f = 2^{-12} = 0.024\%$ as calculated in Section 2.2.1. With 93% table occupancy, the effective number of bits per key using a hash filter is $15/0.93 = 16.12$. In contrast, a standard Bloom filter that sets its number of hash functions to optimize space consumption requires at least $1.44\log_2(1/f) = 17.28$ bits of memory to achieve the same false positive rate.

HashStore's hash filter is also efficient to create: SILT simply copies the tags from the LogStore's hash table, in order, discarding the offset pointers; on the contrary, Bloom filters would have been built from scratch, hashing every item in the LogStore again.

2.2.3 SortedStore

SortedStore is a static key-value store with very low memory footprint. It stores `(key,value)` entries sorted by key on flash, indexed by a new *Entropy-Coded Trie* data structure that is fast to construct, uses 0.4 bytes of index memory per key on average, and keeps read amplification low (exactly 1) by directly pointing to the correct location on flash.

We dedicate Chapter 3 to describe the design of Entropy-Coded Tries further and compare it with the current state-of-the-art hash function.

Using Sorted Data on Flash Because of these desirable properties, SILT keeps most of the key-value entries in a single SortedStore. The Entropy-Coded Trie, however, does not allow for insertions or deletions; thus, to merge HashStore entries into the SortedStore, SILT must generate a new SortedStore. The construction speed of the SortedStore is therefore a large factor in SILT's overall performance.

Sorting provides a natural way to achieve fast construction:

1. Sorting allows efficient bulk-insertion of new data. The new data can be sorted and sequentially merged into the existing sorted data.
2. Sorting is well-studied. SILT can use highly optimized and tested sorting systems such as Nsort [109].

Indexing Sorted Data with a Trie A trie, or a prefix tree, is a tree data structure that stores an array of keys where each leaf node represents one key in the array, and each internal node represents the longest common prefix of the keys represented by its descendants.

When fixed-length key-value entries are sorted by key on flash, a trie for the *shortest unique prefixes* of the keys serves as an index for these sorted data. The shortest unique prefix of a key is the shortest prefix of a key that enables distinguishing the key from the other keys. In such a trie, some prefix of a lookup key leads us to a leaf node with a direct index for the looked up key in sorted data on flash.

Figure 2.5 shows an example of using a trie to index sorted data. Key prefixes with no shading are the shortest unique prefixes which are used for indexing. The shaded parts are ignored for indexing because any value for the suffix part would not change the key location. A lookup

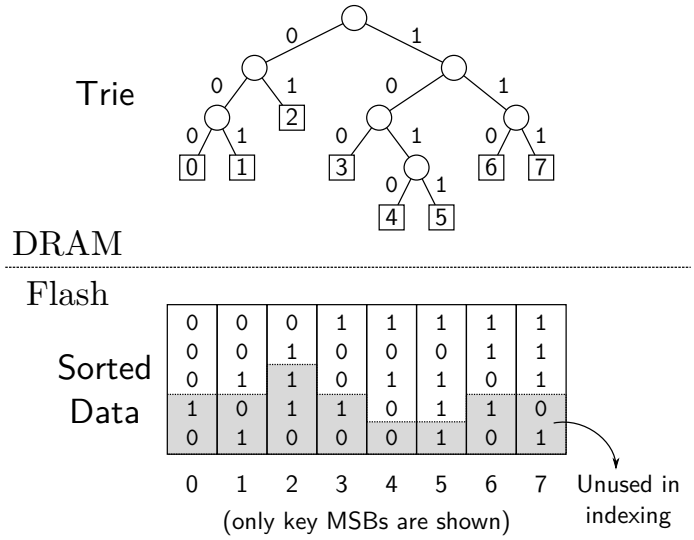


Figure 2.5: Example of a trie built for indexing sorted keys. The index of each leaf node matches the index of the corresponding key in the sorted keys.

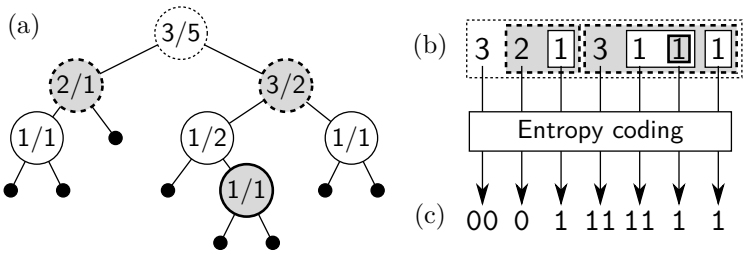


Figure 2.6: (a) Alternative view of Figure 2.5, where a pair of numbers in each internal node denotes the number of leaf nodes in its left and right subtries. (b) A recursive form that represents the trie. (c) Its entropy-coded representation used by SortedStore.

of a key, for example, `10010`, follows down to the leaf node that represents `100`. As there are 3 preceding leaf nodes, the index of the key is 3. With fixed-length key-value pairs on flash, the exact offset of the data is the obtained index times the key-value pair size (see Section 2.3 for extensions for variable-length key-value pairs). Note that a lookup of similar keys with the same prefix of `100` (e.g., `10000`, `10011`) would return the same index even though they are not in the array; the trie guarantees a correct index lookup for stored keys, but says nothing about the presence of a lookup key.

Representing a Trie A typical tree data structure is unsuitable for SILT because each node would require expensive memory pointers, each 2 to 8 bytes long. Common trie representations such as level-compressed tries [7] are also inadequate if they use pointers.

```
# @param  T  array of sorted keys
# @return    trie representation
def construct(T):
  if len(T) == 0 or len(T) == 1:
    return [-1]
  else:
    # Partition keys according to their MSB
    L = [key[1:] for key in T if key[0] == 0]
    R = [key[1:] for key in T if key[0] == 1]
    # Recursively construct the representation
    return [len(L)] + construct(L) + construct(R)
```

Algorithm 1: Trie representation generation in Python-like syntax. `key[0]` and `key[1:]` denote the most significant bit and the remaining bits of `key`, respectively.

SortedStore uses a compact recursive representation to eliminate pointers. The representation for a trie T having L and R as its left and right subtries is defined as follows:

$$\texttt{Repr}(T) \;:=\; |L|\ \texttt{Repr}(L)\ \texttt{Repr}(R)$$

where $|L|$ is the number of leaf nodes in the left subtrie. When T is empty or a leaf node, the representation for T is an empty string. (We use a special mark (`-1`) instead of the empty string for brevity in the simplified algorithm description, but the full algorithm does not require the use of the special mark.)

Figure 2.6 (a,b) illustrates the uncompressed recursive representation for the trie in Figure 2.5. As there are 3 keys starting with `0`, $|L| = 3$. In its left subtrie, $|L| = 2$ because it has 2 keys that have `0` in their second bit position, so the next number in the representation is 2. It again recurses into its left subtrie, yielding 1. Here there are no more non-leaf nodes, so it returns to the root node and then generates the representation for the right subtrie of the root, 3 1 1 1.

Algorithm 1 shows a simplified algorithm that builds a (non-entropy-coded) trie representation from sorted keys. It resembles quicksort in that it finds the partition of keys and recurses into both subsets. Index generation is fast ($\geq$ 7 M keys/sec on a modern Intel desktop CPU, Section 2.5).

Looking up Keys Key-lookup works by incrementally reading the trie representation (Algorithm 2). The function is supplied the lookup key and a trie representation string. By decoding the encoded next number, *thead*, SortedStore knows if the current node is an internal node where it can recurse into its subtrie. If the lookup key goes to the left subtrie, SortedStore recurses into the left subtrie, whose representation immediately follows in the given trie representation; otherwise, SortedStore recursively decodes and discards the entire left subtrie and then recurses into the right. SortedStore sums *thead* at every node where it recurses into a right subtrie; the sum of the *thead* values is the offset at which the lookup key is stored, if it exists.

For example, to look up `10010`, SortedStore first obtains 3 from the representation. Then, as the first bit of the key is `1`, it skips the next numbers (2 1) which are for the representation of the left subtrie, and it proceeds to the right subtrie. In the right subtrie, SortedStore reads the next number (3; not a leaf node), checks the second bit of the key, and keeps recursing into its left subtrie. After reading the next number for the current subtrie (1), SortedStore arrives at a leaf

```
# @param  key    lookup key
# @param  trepr  trie representation
# @return        index of the key
#                in the original array
def lookup(key, trepr):
  (thead, ttail) = (trepr[0], trepr[1:])
  if thead == -1:
    return 0
  else:
    if key[0] == 0:
      # Recurse into the left subtrie
      return lookup(key[1:], ttail)
    else:
      # Skip the left subtrie
      ttail = discard_subtrie(ttail)
      # Recurse into the right subtrie
      return thead + lookup(key[1:], ttail)

# @param  trepr  trie representation
# @return        remaining trie representation
#                with the next subtrie consumed
def discard_subtrie(trepr):
  (thead, ttail) = (trepr[0], trepr[1:])
  if thead == -1:
    return ttail
  else:
    # Skip both subtries
    ttail = discard_subtrie(ttail)
    ttail = discard_subtrie(ttail)
    return ttail
```

Algorithm 2: Key lookup on a trie representation.

node by taking the left subtrie. Until it reaches the leaf node, it takes a right subtrie only at the root node; from $n = 3$ at the root node, SortedStore knows that the offset of the data for `10010` is $(3 \times \text{key-value-size})$ on flash.

Compression Although the above uncompressed representation uses up to 3 integers per key on average, for *hashed* keys, SortedStore can easily reduce the average representation size to 0.4 bytes/key by compressing each $|L|$ value using *entropy coding* (Figure 2.6 (c)). The value of $|L|$ tends to be close to half of $|T|$ (the number of leaf nodes in T) because fixed-length hashed keys are uniformly distributed over the key space, so both subtries have the same probability of storing a key. More formally, $|L| \sim \mathit{Binomial}(|T|, \frac{1}{2})$. When $|L|$ is small enough (e.g., ≤ 16), SortedStore uses static, globally shared Huffman tables based on the binomial distributions. If $|L|$ is large, SortedStore encodes the difference between $|L|$ and its expected value (i.e., $\frac{|T|}{2}$) using Elias gamma coding [48] to avoid filling the CPU cache with large Huffman tables. With this entropy coding optimized for hashed keys, our Entropy-Coded Trie representation is about twice as compact as the previous best recursive tree encoding [35].

When handling compressed tries, Algorithm 1 and 2 are extended to keep track of the number of leaf nodes at each recursion step. This does not require any additional information in the representation because the number of leaf nodes can be calculated recursively using $|T| = |L| + |R|$. Based on $|T|$, these algorithms choose an entropy coder for encoding `len(L)` and decoding `thead`. It is noteworthy that the special mark (`-1`) takes no space with entropy coding, as its entropy is zero.

Ensuring Constant Time Index Lookups As described, a lookup may have to decompress the entire trie, so that the cost of lookups would grow (large) as the number of entries in the key-value store grows.

To bound the lookup time, items are partitioned into 2^k buckets based on the first k bits of their key. Each bucket has its own trie index. Using, e.g., $k = 10$ for a key-value store holding 2^{16} items, each bucket would hold in expectation $2^{16-10} = 2^6$ items. With high probability, no bucket holds more than 2^8 items, so the time to decompress the trie for bucket is both bounded by a constant value, and small.

This bucketing requires additional information to be stored in memory: (1) the pointers to the trie representations of each bucket and (2) the number of entries in each bucket. SILT keeps the amount of this bucketing information small (less than 1 bit/key) by using a simpler version of a compact select data structure, semi-direct-16 [21]. With bucketing, our trie-based indexing belongs to the class of data structures called monotone minimal perfect hashing [16, 26] (Section 2.6).

Further Space Optimizations for Small Key-Value Sizes For small key-value entries, SortedStore can reduce the trie size by applying *sparse indexing* [47]. Sparse indexing locates the *block* that contains an entry, rather than the exact offset of the entry. This technique requires scanning or binary search within the block, but it reduces the amount of indexing information. It is particularly useful when the storage media has a minimum useful block size to read; many flash devices, for instance, provide increasing I/Os per second as the block size drops, but not past a certain limit (e.g., 512 or 4096 bytes) [106, 117]. SILT uses sparse indexing when configured for key-value sizes of 64 bytes or smaller.

SortedStore obtains a sparse-indexing version of the trie by pruning some subtries in it. When a trie has subtries that have entries all in the same block, the trie can omit the representation of these subtries because the omitted data only gives in-block offset information between entries. Pruning can reduce the trie size to 1 bit per key or less if each block contains 16 key-value entries or more.

Merging HashStores into SortedStore SortedStore is an immutable store and cannot be changed. Accordingly, the merge process generates a new SortedStore based on the given HashStores and the existing SortedStore. Similar to the conversion from LogStore to HashStore, HashStores and the old SortedStore can serve lookups during merging.

The merge process consists of two steps: (1) sorting HashStores and (2) sequentially merging sorted HashStores data and SortedStore. First, SILT sorts all data in HashStores to be merged. This task is done by enumerating every entry in the HashStores and sorting these entries. Then,

Comparison	"Deleted"?	Action on K_{SS}	Action on K_{HS}
$K_{SS} < K_{HS}$	any	copy	–
$K_{SS} > K_{HS}$	no	–	copy
$K_{SS} > K_{HS}$	yes	–	drop
$K_{SS} = K_{HS}$	no	drop	copy
$K_{SS} = K_{HS}$	yes	drop	drop

Table 2.3: Merge rule for SortedStore. K_{SS} is the current key from SortedStore, and K_{HS} is the current key from the sorted data of HashStores. "Deleted" means the current entry in K_{HS} is a special entry indicating a key of SortedStore has been deleted.

this sorted data from HashStores is sequentially merged with already sorted data in the SortedStore. The sequential merge chooses newest valid entries, as summarized in Table 2.3; either copy or drop action on a key consumes the key (i.e., by advancing the "merge pointer" in the corresponding store), while the current key remains available for the next comparison again if no action is applied to the key. After both steps have been completed, the old SortedStore is atomically replaced by the new SortedStore. During the merge process, both the old SortedStore and the new SortedStore exist on flash; however, the flash space overhead from temporarily having two SortedStores is kept small by performing the merge process on a single SILT instance at the same time.

In Section 2.4, we discuss how frequently HashStores should be merged into SortedStore.

Application of False Positive Filters Since SILT maintains only one SortedStore per SILT instance, SortedStore does not have to use a false positive filter to reduce unnecessary I/O. However, an extension to the SILT architecture might have multiple SortedStores. In this case, the trie index can easily accommodate the false positive filter; the filter is generated by extracting the key fragments from the sorted keys. Key fragments can be stored in an in-memory array so that they have the same order as the sorted data on flash. The extended SortedStore can consult the key fragments before reading data from flash.

2.3 Extending SILT Functionality

SILT can support an even wider range of applications and workloads than the basic design we have described. In this section, we present potential techniques to extend SILT's capabilities.

Crash Tolerance SILT ensures that all its in-memory data structures are backed-up to flash and/or easily re-created after failures. All updates to LogStore are appended to the on-flash log chronologically; to recover from a fail-stop crash, SILT simply replays the log file to construct the in-memory index and filter. For HashStore and SortedStore, which are static, SILT keeps a copy of their in-memory data structures on flash, which can be re-read during recovery.

SILT's current design, however, does not provide crash tolerance for *new* updates to the data store. These writes are handled asynchronously, so a key insertion/update request to SILT may complete before its data is written durably to flash. For applications that need this additional

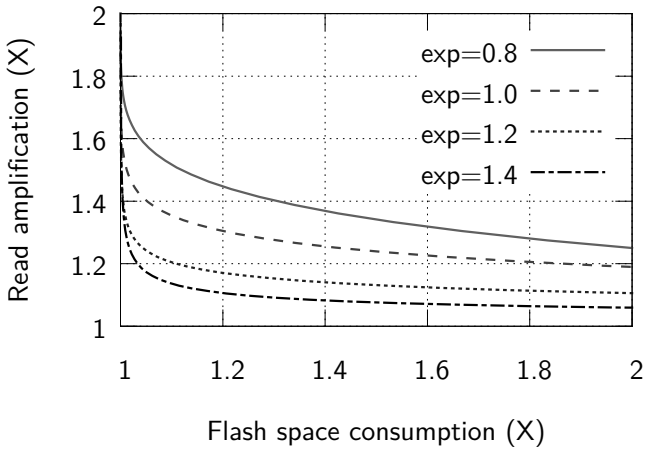


Figure 2.7: Read amplification as a function of flash space consumption when inlining is applied to key-values whose sizes follow a Zipf distribution. "exp" is the exponent part of the distribution.

level of crash tolerance, SILT would need to support an additional synchronous write mode. For example, SILT could delay returning from write calls until it confirms that the requested write is fully flushed to the on-flash log.

Variable-Length Key-Values For simplicity, the design we presented so far focuses on fixed-length key-value data. In fact, SILT can easily support variable-length key-value data by using *indirection with inlining*. This scheme follows the existing SILT design with fixed-sized slots, but stores `(offset, first part of (key, value))` pairs instead of the actual `(key, value)` in HashStores and SortedStores (LogStores can handle variable-length data natively). These offsets point to the remaining part of the key-value data stored elsewhere (e.g., a second flash device). If a whole item is small enough to fit in a fixed-length slot, indirection can be avoided; consequently, large data requires an extra flash read (or write), but small data incurs no additional I/O cost. Figure 2.7 plots an analytic result on the tradeoff of this scheme with different slot sizes. It uses key-value pairs whose sizes range between 4 B and 1 MiB and follow a Zipf distribution, assuming a 4-byte header (for key-value lengths), a 4-byte offset pointer, and a uniform access pattern.

For specific applications, SILT can alternatively use segregated stores for further efficiency. Similar to the idea of simple segregated storage [143], the system could instantiate several SILT instances for different fixed key-value size combinations. The application may choose an instance with the most appropriate key-value size as done in Dynamo [44], or SILT can choose the best instance for a new key and return an opaque key containing the instance ID to the application. Since each instance can optimize flash space overheads and additional flash reads for its own dataset, using segregated stores can reduce the cost of supporting variable-length key-values close to the level of fixed-length key-values.

In the subsequent sections, we will discuss SILT with fixed-length key-value pairs only.

Fail-Graceful Indexing Under high memory pressure, SILT may temporarily operate in a degraded indexing mode by allowing higher read amplification (e.g., more than 2 flash reads per lookup) to avoid halting or crashing because of insufficient memory.

(1) Dropping in-memory indexes and filters. HashStore's filters and SortedStore's indexes are stored on flash for crash tolerance, allowing SILT to drop them from memory. This option saves memory at the cost of one additional flash read for the SortedStore, or two for the HashStore.

(2) Binary search on SortedStore. The SortedStore can be searched without an index, so the in-memory trie can be dropped even without storing a copy on flash, at the cost of $\log(n)$ additional reads from flash.

These techniques also help speed SILT's startup. By memory-mapping on-flash index files or performing binary search, SILT can begin serving requests before it has loaded its indexes into memory in the background.

2.4 Analysis

Compared to single key-value store approaches, the multi-store design of SILT has more system parameters, such as the size of a single LogStore and HashStore, the total number of HashStores, the frequency to merge data into SortedStore, and so on. Having a much larger design space, it is preferable to have a systematic way to do parameter selection.

This section provides a simple model of the tradeoffs between write amplification (WA), read amplification (RA), and memory overhead (MO) in SILT, with an eye towards being able to set the system parameters properly to achieve the design goals from Section 2.1.

$$\text{WA} = \frac{\text{data written to flash}}{\text{data written by application}}, \tag{2.1}$$

$$\text{RA} = \frac{\text{data read from flash}}{\text{data read by application}}, \tag{2.2}$$

$$\text{MO} = \frac{\text{total memory consumed}}{\text{number of items}}. \tag{2.3}$$

Model A SILT system has a flash drive of size F bytes with a lifetime of E erase cycles. The system runs P SILT instances locally, each of which handles one disjoint range of keys using one LogStore, one SortedStore, and multiple HashStores. Once an instance has d keys in total in its HashStores, it merges these keys into its SortedStore.

We focus here on a workload where the total amount of data stored in the system remains constant (e.g., only applying updates to existing keys). We omit for space the similar results when the data volume is growing (e.g., new keys are inserted to the system) and additional nodes are being added to provide capacity over time. Table 2.4 presents the notation used in the analysis.

Write Amplification An update first writes one record to the LogStore. Subsequently converting that LogStore to a HashStore incurs $1/0.93 = 1.075$ writes per key, because the space occupancy of the hash table is 93%. Finally, d total entries (across multiple HashStores of one SILT instance)

Symbol	Meaning	Example
	SILT design parameters	
d	maximum number of entries to merge	7.5 M
k	tag size in bit	15 bits
P	number of SILT instances	4
H	number of HashStores per instance	31
f	false positive rate per store	2^{-12}
	Workload characteristics	
c	key-value entry size	1024 B
N	total number of distinct keys	100 M
U	update rate	5 k/sec
	Storage constraints	
F	total flash size	256 GB
E	maximum flash erase cycle	10,000

Table 2.4: Notation.

are merged into the existing SortedStore, creating a new SortedStore with N/P entries. The total write amplification is therefore

$$\text{WA} = 2.075 + \frac{N}{d \cdot P}. \tag{2.4}$$

Read Amplification The false positive rate of flash reads from a 4-way set associative hash table using k-bit tags is $f = 8/2^k$ because there are eight possible locations for a given key—two possible buckets and four items per bucket.

This 4-way set associative cuckoo hash table with k-bit tags can store 2^{k+2} entries, so at 93% occupancy, each LogStore and HashStore holds $0.93 \cdot 2^{k+2}$ keys. In one SILT instance, the number of items stored in HashStores ranges from 0 (after merging) to d, with an average size of $d/2$, so the average number of HashStores is

$$H = \frac{d/2}{0.93 \cdot 2^{k+2}} = 0.134 \, \frac{d}{2^k}. \tag{2.5}$$

In the *worst-case* of a lookup, the system reads once from flash at the SortedStore, after $1+H$ failed retrievals at the LogStore and H HashStores. Note that each LogStore or HashStore rejects all but an f fraction of false positive retrievals; therefore, the expected total number of reads per lookup (read amplification) is:

$$\text{RA} = (1+H)f + 1 = \frac{8}{2^k} + 1.07\frac{d}{4^k} + 1. \tag{2.6}$$

By picking d and k to ensure $1.07d/4^k + 8/2^k < \varepsilon$, SILT can achieve the design goal of read amplification $1+\varepsilon$.

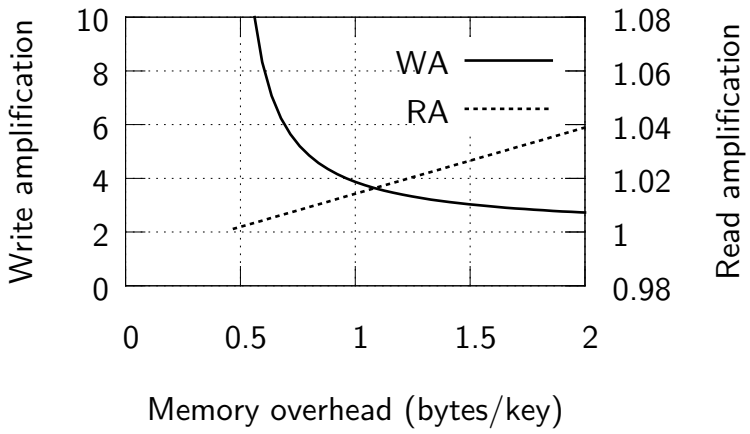


Figure 2.8: WA and RA as a function of MO when N=100 M, P=4, and k=15, while d is varied.

Memory Overhead Each entry in LogStore uses $(k+1)/8+4$ bytes (k bits for the tag, one valid bit, and 4 bytes for the pointer). Each HashStore filter entry uses $k/8$ bytes for the tag. Each SortedStore entry consumes only 0.4 bytes. Using one LogStore, one SortedStore, and H HashStores, SILT's memory overhead is:

$$\begin{aligned} \text{MO} &= \frac{\left(\left(\frac{k+1}{8}+4\right)\cdot 2^{k+2}+\frac{k}{8}\cdot 2^{k+2}\cdot H+0.4\cdot\frac{N}{P}\right)\cdot P}{N} \\ &= \left((16.5+0.5k)2^k+0.067\ kd\right)\frac{P}{N}+0.4. \end{aligned} \tag{2.7}$$

Tradeoffs Improving either write amplification, read amplification, or memory amplification comes at the cost of one of the other two metrics. For example, using larger tags (i.e., increasing k) reduces read amplification by reducing both f the false positive rate per store and H the number of HashStores. However, the HashStores then consume more DRAM due to the larger tags, increasing memory overhead. Similarly, by increasing d, SILT can merge HashStores into the SortedStore less frequently to reduce the write amplification, but doing so increases the amount of DRAM consumed by the HashStore filters. Figure 2.8 illustrates how write and read amplification change as a function of memory overhead when the maximum number of HashStore entries, d, is varied.

Update Rate vs. Flash Life Time The designer of a SILT instance handling U updates per second wishes to ensure that the flash lasts for at least T seconds. Assuming the flash device has perfect wear-leveling when being sent a series of large sequential writes [32], the total number of writes, multiplied by the write amplification WA, must not exceed the flash device size times its erase cycle budget. This creates a relationship between the lifetime, device size, update rate, and memory overhead:

$$U\cdot c\cdot \text{WA}\cdot T \quad\leq\quad F\cdot E. \tag{2.8}$$

Example Assume a SILT system is built with a 256 GB MLC flash drive supporting 10,000 erase cycles [10] ($E = 10000$, $F = 256 \times 2^{30}$). It is serving $N = 100$ million items with $P = 4$ SILT instances, and $d = 7.5$ million. Its workload is 1 KiB entries, 5,000 updates per second ($U = 5000$).

By Eq. (2.4) the write amplification, WA, is 5.4. That is, each key-value update incurs 5.4 writes/entry. On average the number of HashStores is 31 according to Eq. (2.5). The read amplification, however, is very close to 1. Eq. (2.6) shows that when choosing 15 bits for the key fragment size, a `GET` incurs on average 1.008 of flash reads even when all stores must be consulted. Finally, we can see how the SILT design achieves its design goal of memory efficiency: indexing a total of 102.4 GB of data, where each key-value pair takes 1 KiB, requires only 73 MB in total or 0.73 bytes per entry (Eq. (2.7)). With the write amplification of 5.4 from above, this device will last 3 years.

Considering Workload Characteristics SILT performs better with specific workloads where the workload skew is high, i.e., where a few keys are popular while the other keys appear less frequently. Popular keys are typically deduplicated in LogStore, which delays the conversion from LogStore to HashStore.

Analyzing the effect of workload characteristics on the system behaviors requires an additional set of analysis primitives. We explore the new primitives in Chapter 4 and evaluate SILT for skewed workloads.

2.5 Evaluation

Using macro- and micro-benchmarks, we evaluate SILT's overall performance and explore how its system design and algorithms contribute to meeting its goals. We specifically examine (1) an end-to-end evaluation of SILT's throughput, memory overhead, and latency; (2) the performance of SILT's in-memory indexing data structures in isolation; and (3) the individual performance of each data store type, including flash I/O.

Implementation SILT is implemented in 15 k lines of C++ using a modular architecture similar to Anvil [95]. Each component of the system exports the same, basic key-value interface. For example, the classes which implement each of the three stores (LogStore, HashStore, and SortedStore) export this interface but themselves call into classes which implement the in-memory and on-disk data structures using that same interface. The SILT system, in turn, unifies the three stores and provides this key-value API to applications. (SILT also has components for background conversion and merging.)

Evaluation System We evaluate SILT on Linux using a desktop equipped with:

CPU	Intel Core i7 860 @ 2.80 GHz (4 cores)
DRAM	DDR SDRAM / 8 GiB
SSD-L	Crucial RealSSD C300 / 256 GB
SSD-S	Intel X25-E / 32 GB

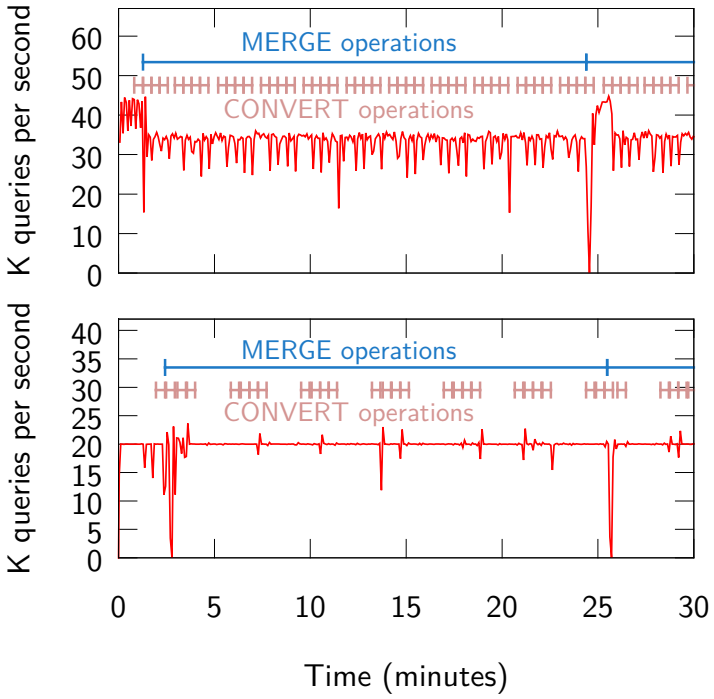


Figure 2.9: `GET` throughput under high (upper) and low (lower) loads.

The 256 GB SSD-L stores the key-value data, and the SSD-S is used as scratch space for sorting HashStores using Nsort [109]. The drives connect using SATA and are formatted with the ext4 filesystem using the `discard` mount option (TRIM support) to enable the flash device to free blocks from deleted files. The baseline performance of the data SSD is:

Random Reads (1024 B)	48 k reads/sec
Sequential Reads	256 MB/sec
Sequential Writes	233 MB/sec

2.5.1 Full System Benchmark

Workload Generation We use YCSB [38] to generate a key-value workload. By default, we use a 10% PUT / 90% GET workload for 20-byte keys and 1000-byte values, and we also use a 50% PUT / 50% GET workload for 64-byte key-value pairs in throughput and memory overhead benchmarks. To avoid the high cost of the Java-based workload generator, we use a lightweight SILT client to replay a captured trace file of queries made by YCSB. The experiments use four SILT instances ($P = 4$), with 16 client threads concurrently issuing requests. When applicable, we limit the rate at which SILT converts entries from LogStores to HashStores to 10 k entries/second, and from HashStores to the SortedStore to 20 k entries/second in order to prevent these background operations from exhausting I/O resources.

Throughput: SILT can sustain an average insert rate of 3,000 1 KiB key-value pairs per second, while simultaneously supporting 33,000 queries/second, or 69% of the SSD's random read capacity. With no inserts, SILT supports 46 k queries per second (**96**% of the drive's raw

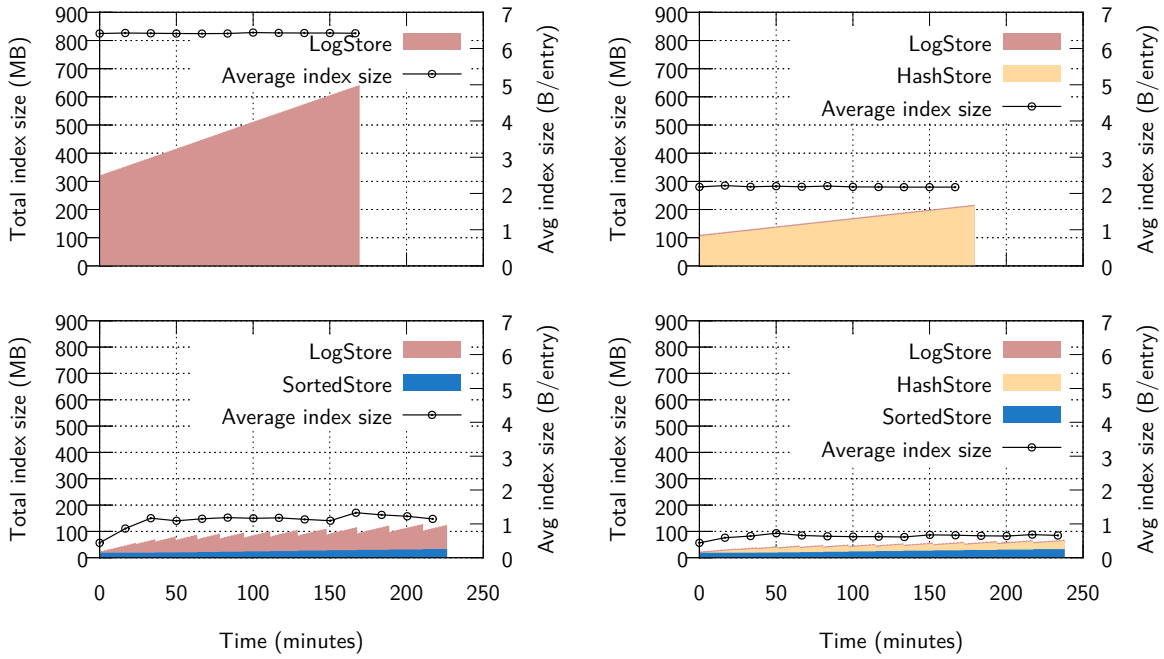


Figure 2.10: Index size changes for four different store combinations while inserting 50 M new entries.

capacity), and with no queries, can sustain an insert rate of approximately 23 k inserts per second. On a deduplication-like workload with 50% writes and 50% reads of 64 byte records, SILT handles 72,000 requests/second.

SILT's performance under insert workloads is limited by the time needed to convert and merge data into HashStores and SortedStores. These *background operations* compete for flash I/O resources, resulting in a tradeoff between query latency and throughput. Figure 2.9 shows the sustainable query rate under both high query load (approx. 33 k queries/second) and low query load (22.2 k queries/second) for 1 KiB key-value pairs. SILT is capable of providing predictable low latency, or can be tuned for higher overall throughput. The middle line shows when SILT converts LogStores into HashStores (periodically, in small bursts). The top line shows that at nearly all times, SILT is busy merging HashStores into the SortedStore in order to optimize its index size.[4] In Section 2.5.3, we evaluate in more detail the speed of the individual stores and conversion processes.

Memory overhead: SILT meets its goal of providing high throughput with low memory overhead. We measured the time and memory required to insert 50 million new 1 KiB entries into a table with 50 million existing entries, while simultaneously handling a high query rate. SILT used at most 69 MB of DRAM, or 0.69 bytes per entry. (This workload is worst-case because it is never allowed time for SILT to compact all of its HashStores into the SortedStore.) For the 50%

[4]In both workloads, when merge operations complete (e.g., at 25 minutes), there is a momentary drop in query speed. This is due to bursty TRIMming by the ext4 filesystem implementation (`discard`) used in the experiment when the previous multi-gigabyte SortedStore file is deleted from flash.

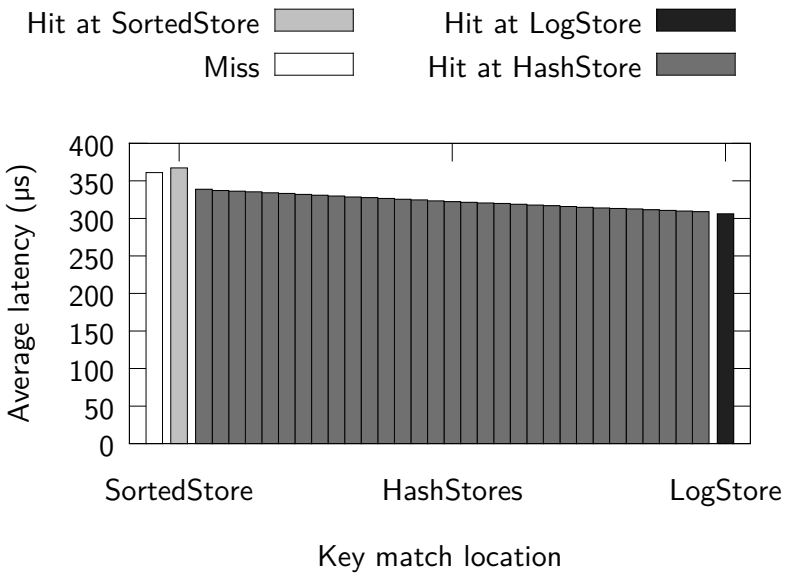


Figure 2.11: GET query latency when served from different store locations.

PUT / 50% GET workload with 64-byte key-value pairs, SILT required at most 60 MB of DRAM for 100 million entries, or 0.60 bytes per entry.

The drastic improvement in memory overhead from SILT's three-store architecture is shown in Figure 2.10. The figure shows the memory consumption during the insertion run over time, using four different configurations of basic store types and 1 KiB key-value entries. The bottom right graph shows the memory consumed using the full SILT system. The bottom left configuration omits the intermediate HashStore, thus requiring twice as much memory as the full SILT configuration. The upper right configuration instead omits the SortedStore, and consumes four times as much memory. Finally, the upper left configuration uses only the basic LogStore, which requires nearly 10X as much memory as SILT. To make this comparison fair, the test generates unique new items so that garbage collection of old entries cannot help the SortedStore run faster.

The figures also help understand the modest cost of SILT's memory efficiency. The LogStore-only system processes the 50 million inserts (500 million total operations) in under 170 minutes, whereas the full SILT system takes 40% longer–about 238 minutes–to incorporate the records, but achieves an order of magnitude better memory efficiency.

Latency: SILT is fast, processing queries in 367 μs on average, as shown in Figure 2.11 for 100% GET queries for 1 KiB key-value entries. GET responses are fastest when served by the LogStore (309 μs), and slightly slower when they must be served by the SortedStore. The relatively small latency increase when querying the later stores shows the effectiveness (reducing the number of extra flash reads to $\varepsilon < 0.01$) and speed of SILT's in-memory filters used in the Log and HashStores.

In the remaining sections, we evaluate the performance of SILT's individual in-memory indexing techniques, and the performance of the individual stores (in-memory indexes plus on-flash data structures).

Type	Cuckoo hashing (K keys/s)	Trie (K keys/s)
Individual insertion	10182	–
Bulk insertion	–	7603
Lookup	1840	208

Table 2.5: In-memory performance of index data structures in SILT on a single CPU core.

2.5.2 Index Microbenchmark

The high random read speed of flash drives means that the CPU budget available for each index operation is relatively limited. This microbenchmark demonstrates that SILT's indexes meet their design goal of computation-efficient indexing.

Experiment Design This experiment measures insertion and lookup speed of SILT's in-memory partial-key cuckoo hash and Entropy-Coded Trie indexes. The benchmark inserts 126 million total entries and looks up a subset of 10 million random 20-byte keys.

This microbenchmark involves memory only, no flash I/O. Although the SILT system uses multiple CPU cores to access multiple indexes concurrently, access to individual indexes in this benchmark is single-threaded. Note that inserting into the cuckoo hash table (LogStore) proceeds key-by-key, whereas the trie (SortedStore) is constructed en mass using bulk insertion. Table 2.5 summarizes the measurement results.

Individual Insertion Speed (Cuckoo Hashing) SILT's cuckoo hash index implementation can handle 10.18 M 20-byte key insertions (`PUT`s or `DELETE`s) per second. Even at a relatively small, higher overhead key-value entry size of 32-byte (i.e., 12-byte data), the index would support 326 MB/s of incoming key-value data on one CPU core. This rate exceeds the typical sequential write speed of a single flash drive: inserting keys into our cuckoo hashing is unlikely to become a bottleneck in SILT given current trends.

Bulk Insertion Speed (Trie) Building the trie index over 126 million pre-sorted keys required approximately 17 seconds, or 7.6 M keys/second.

Key Lookup Speed Each SILT `GET` operation requires a lookup in the LogStore and potentially in one or more HashStores and the SortedStore. A single CPU core can perform 1.84 million cuckoo hash lookups per second. If a SILT instance has 1 LogStore and 31 HashStores, each of which needs to be consulted, then one core can handle about 57.5 k `GET`s/sec. Trie lookups are approximately 8.8 times slower than cuckoo hashing lookups, but a `GET` triggers a lookup in the trie only after SILT cannot find the key in the LogStore and HashStores. When combined, the SILT indexes can handle about $1/(1/57.5\text{ k} + 1/208\text{ k}) \approx 45$ k `GET`s/sec with one CPU core.

Insertions are faster than lookups in cuckoo hashing because insertions happen to only a few tables at the same time and thus benefit from the CPU's L2 cache; lookups, however, can occur to any table in memory, making CPU cache less effective.

Type	Speed (K keys/s)
LogStore (by PUT)	204.6
HashStore (by CONVERT)	67.96
SortedStore (by MERGE)	26.76

Table 2.6: Construction performance for basic stores. The construction method is shown in the parentheses.

Type	SortedStore (K ops/s)	HashStore (K ops/s)	LogStore (K ops/s)
GET (hit)	46.57	44.93	46.79
GET (miss)	46.61	7264	7086

Table 2.7: Query performance for basic stores that include in-memory and on-flash data structures.

Operation on Multiple Cores Using four cores, SILT indexes handle 180 k GETs/sec in memory. At this speed, the indexes are unlikely to become a bottleneck: their overhead is on-par or lower than the operating system overhead for actually performing that many 1024-byte reads per second from flash. As we see in the next section, SILT's overall performance is limited by sorting, but its index CPU use is high enough that adding many more flash drives would require more CPU cores. Fortunately, SILT offers many opportunities for parallel execution: Each SILT node runs multiple, completely independent instances of SILT to handle partitioning, and each of these instances can query many stores.

2.5.3 Individual Store Microbenchmark

Here we measure the performance of each SILT store type in its entirety (in-memory indexing plus on-flash I/O). The first experiment builds multiple instances of each basic store type with 100 M key-value pairs (20-byte key, 1000-byte value). The second experiment queries each store for 10 M random keys.

Table 2.6 shows the construction performance for all three stores; the construction method is shown in parentheses. LogStore construction, built through entry-by-entry insertion using PUT, can use 90% of sequential write bandwidth of the flash drive. Thus, SILT is well-suited to handle bursty inserts. The conversion from LogStores to HashStores is about three times slower than LogStore construction because it involves bulk data reads and writes from/to the same flash drive. SortedStore construction is slowest, as it involves an external sort for the entire group of 31 HashStores to make one SortedStore (assuming no previous SortedStore). If constructing the SortedStore involved merging the new data with an existing SortedStore, the performance would be worse. The large time required to create a SortedStore was one of the motivations for introducing HashStores rather than keeping un-merged data in LogStores.

Table 2.7 shows that the minimum GET performance across all three stores is 44.93 k ops/s. Note that LogStores and HashStores are particularly fast at GET for non-existent keys (more than 7 M ops/s). This extremely low miss penalty explains why there was only a small variance in the

average GET latency in Figure 2.11 where bad cases looked up 32 Log and HashStores and failed to find a matching item in any of them.

2.6 Related Work

Hashing *Cuckoo hashing* [114] is an open-addressing scheme to resolve hash collisions efficiently with high space occupancy. Our partial-key cuckoo hashing—storing only a small part of the key in memory without fetching the entire keys from slow storage on collisions—makes cuckoo hashing more memory-efficient while ensuring high performance.

Minimal perfect hashing is a family of collision-free hash functions that map n distinct keys to n consecutive integers $0 \ldots n-1$, and is widely used for memory-efficient indexing. In theory, any minimal perfect hash scheme requires at least 1.44 bits/key [62]; in practice, the state-of-the-art schemes can index any static data set with 2.07 bits/key [16]. Our Entropy-Coded Trie achieves 3.1 bits/key, but it also preserves the lexicographical order of the keys to facilitate data merging. Thus, it belongs to the family of monotone minimal perfect hashing (MMPH). Compared to other proposals for MMPH [14, 15], our trie-based index is simple, lightweight to generate, and has very small CPU/memory overhead.

External-Memory Index on Flash Recent work such as MicroHash [147] and FlashDB [107] minimizes memory consumption by having indexes on flash. MicroHash uses a hash table chained by pointers on flash. FlashDB proposes a self-tuning B^+-tree index that dynamically adapts the node representation according to the workload. Both systems are optimized for memory and energy consumption of sensor devices, but not for latency as lookups in both systems require reading multiple flash pages. In contrast, SILT achieves very low memory footprint while still supporting high throughput.

Key-Value Stores HashCache [9] proposes several policies to combine hash table-based in-memory indexes and on-disk data layout for caching web objects. FAWN-DS [6] consists of an on-flash data log and in-memory hash table index built using relatively slow CPUs with a limited amount of memory. SILT dramatically reduces DRAM consumption compared to these systems by combining more memory-efficient data stores with minimal performance impact. FlashStore [42] also uses a single hash table to index all keys on flash similar to FAWN-DS. The flash storage, however, is used as a cache of a hard disk-backed database. Thus, the cache hierarchy and eviction algorithm is orthogonal to SILT. To achieve low memory footprint (about 1 byte/key), SkimpyStash [43] moves its indexing hash table to flash with linear chaining. However, it requires on average 5 flash reads per lookup, while SILT only needs $1+\varepsilon$ per lookup.

More closely related to our design is BufferHash [5], which keeps keys in multiple equal-sized hash tables—one in memory and the others on flash. The on-flash tables are guarded by in-memory Bloom filters to reduce unnecessary flash reads. In contrast, SILT data stores have different sizes and types. The largest store (SortedStore), for example, does not have a filter and is accessed at most once per lookup, which saves memory while keeping the read amplification low. In addition, writes in SILT are appended to a log stored on flash for crash recovery, whereas inserted keys in BufferHash do not persist until flushed to flash in batch.

Several key-value storage libraries rely on caching to compensate for their high read amplifications [12, 65], making query performance depend greatly on whether the working set fits in the in-memory cache. In contrast, SILT provides uniform and predictably high performance regardless of the working set size and query patterns.

Distributed Key-Value Systems Distributed key-value storage clusters such as BigTable [31], Dynamo [44], and FAWN-KV [6] all try to achieve high scalability and availability using a cluster of key-value store nodes. SILT focuses on how to use flash memory-efficiently with novel data structures, and is complementary to the techniques used in these other systems aimed at managing failover and consistency.

Modular Storage Systems BigTable [31] and Anvil [95] both provide a modular architecture for chaining specialized stores to benefit from combining different optimizations. SILT borrows its design philosophy from these systems; we believe and hope that the techniques we developed for SILT could also be used within these frameworks.

2.7 Summary

SILT combines new algorithmic and systems techniques to balance the use of memory, storage, and computation to craft a memory-efficient, high-performance flash-based key value store. It uses two new in-memory index structures—partial-key cuckoo hashing and Entropy-Coded Tries—to reduce drastically the amount of memory needed compared to prior systems. SILT chains the right combination of basic key-value stores together to create a system that provides high write speed, high read throughput, and uses little memory, attributes that no single store can achieve alone. SILT uses in total only 0.7 bytes of memory per entry it stores, and makes only 1.01 flash reads to service a lookup, doing so in under 400 microseconds. Our hope is that SILT, and the techniques described herein, can form an efficient building block for a new generation of fast data-intensive services.

Chapter 3

ECT (Entropy-Coded Tries)

As the amount of data stored in large-scale storage systems continues to grow, so does the importance of algorithms and data structures to provide resource-efficient access to that data. In this thesis, we address the question of *external hashing*: using a small amount of RAM to efficiently locate data stored on a (large) external disk or flash drive. Our solution, *Entropy-Coded Tries*, provides three properties that are important in practice: It is memory and disk efficient; it provides fast access; and, by using sorting as its fundamental organizing method, is easy to engineer for high performance and reliability.

Traditional external hashing schemes put emphasis on the construction and evaluation speed of the hash function or the asymptotic size of the generated hash function [14, 15, 25]. However, in modern data-intensive systems, external hashing faces three challenges driven by technology and application trends:

- *Flash drives*: Flash provides fast random access speed—typically 20,000 to 1,300,000 I/Os per second (IOPS) [63, 79], compared to only a few hundred IOPS for hard drives. Flash also has fast sequential I/O, often exceeding 250 MB/s. Prior studies of external hashing only considered its use on slow hard disks; these systems, therefore, were not designed to take full advantage of high flash performance.
- *Growing cost of memory*: DRAM has become increasingly expensive and power-hungry relative to both disk and flash storage. As a result, the need for memory-efficient external hashing has grown—not only in asymptotic space use, but also in actual space use when storing millions to billions of keys.
- *Mission-critical storage systems*: Services ranging from Google and Facebook to end-user software such as web browsers use external hashing to build a highly memory-efficient external dictionary, which stores a large number of key-value pairs on disk or flash and provides fast retrieval of the data items. Ensuring the correctness and high performance of this dictionary is both critical and difficult; using a well-studied software component—sorting—as the main building block of external hashing can help achieve these goals.

The contributions of this chapter are three-fold:

1. **Entropy-Coded Tries (ECT)**: ECT is a minimal perfect hashing (MPH) scheme that uses an array of highly-compressed tries on a sorted dataset in external storage. Given n distinct input keys, we assign unique values in the range $[0, n-1]$ to the input keys,

whose order (or index) is determined by the hash values of the keys; we represent the mapping from the keys to their indices as tries, each of which is compressed using a new trie compression technique—a combination of Huffman coding and Elias-gamma coding that exploits the underlying statistical properties of tries created from hashed keys. ECT requires only 2.5 bits per key on average for an in-memory index (close to minimal perfect hashing's information-theoretic lower bound of approximately 1.44 bits per key [27]) and 7 microseconds to lookup a key. This CPU requirement can be reduced for very fast SSDs by trading some space efficiency.

2. **Sort as a basic building block of efficient external hashing**: We demonstrate that the use of a well-engineered sort implementation can provide high-performance indexing in ECT and even speed up other external hashing schemes that are not originally based on sorting. In particular, we develop a modified version of EPH [25] that uses sorting instead of partitioning, and show substantial performance improvements over its original partition-based scheme. ECT, which natively uses sorting, in turn outperforms both partitioning and sorting versions of EPH. In addition, sort provides two engineering benefits: (1) The system design becomes easy to understand and implement; and (2) the system can drop the index and fall back to binary search whenever desired. Further, a simple assessment of the system's correctness is possible by using a sort implementation that has been widely adopted in data processing systems.
3. **Incremental updates for dynamic datasets**: In workloads dealing with dynamic datasets, it is crucial to incorporate a series of updates into the existing dataset quickly. We show the benefits of supporting incremental updates in ECT as well as in our version of EPH. To handle batched updates, we do not perform the entire index construction on the pre-existent data; we partition/sort new changes only, and then sequentially merge the new changes into the previously indexed items while generating new in-memory structures during the merge. This incremental process incurs less I/O than the full construction of the dataset, which reduces update time.

3.1 Design

This section presents the design of Entropy-Coded Tries (ECT). It first presents a high-level overview of ECT and then describes major components of the ECT scheme in detail.

3.1.1 Overview

ECT supports $O(1)$ retrieval of data items stored in external storage using approximately 2.5 bits of DRAM per key on average. It does so by keeping the data sorted in *hash order* (the order of the keys after applying a hash function to them), grouping adjacent keys into *virtual buckets* (each of which contains keys that share the same hash prefix of a certain length), and creating an efficient trie-based index for each virtual bucket to quickly locate the index of a particular pre-hashed key within that virtual bucket.

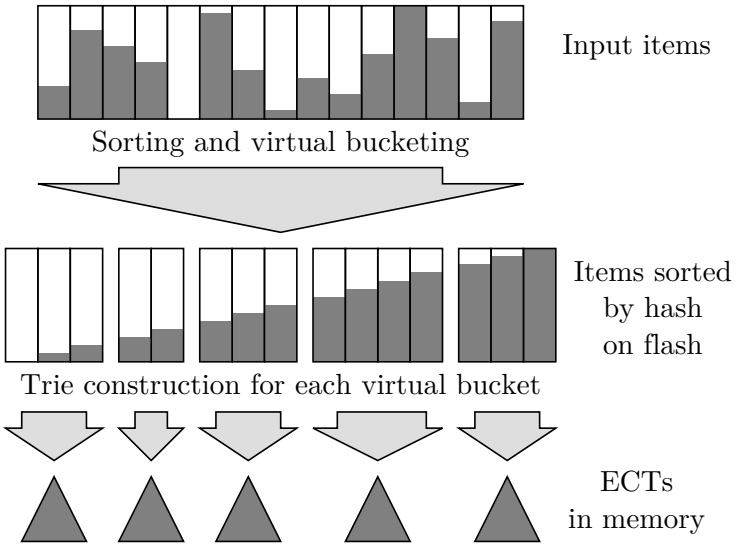


Figure 3.1: Workflow of the construction of ECT.

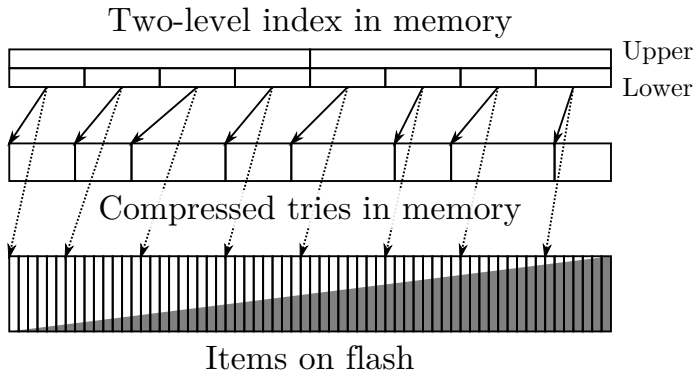


Figure 3.2: Constructed ECT data structures in memory and on flash.

Figure 3.1 depicts the ECT construction process. Each box represents an item, and the height of the gray bar indicates the relative order of the item's hashed key. Our scheme sorts the dataset on flash and builds a trie for each disjoint group of adjacent keys (virtual bucket) in memory. During construction, the only data manipulation is sort, as virtual bucketing and trie construction do not move data on flash.

Figure 3.2 shows the data structures generated by the ECT scheme. A two-level index stores internal indexing information: the in-memory location of the compressed trie representation and the on-flash location of the first item of each virtual bucket. This internal index has an additional benefit in that it allows dense storage of other ECT data structures: (1) Compressed tries, one of which is generated for each virtual bucket, are stored contiguously in memory to achieve a small memory footprint; and (2) sorted hashkeys (and associated values) are kept on flash without wasting storage space.

In the following subsections, we detail the algorithms and data structures used by ECT.

3.1.2 Sorting in Hash Order

ECT uses external sort as its sole data manipulation mechanism. Unlike many other indexing schemes, ECT does not require a specific procedure to construct a structured data layout (e.g., B-tree) or a custom data layout (e.g., EPH [25]). ECT directly applies external sort to the input dataset to obtain the final data layout for which ECT constructs an index. As sort is a well-established building block for many algorithms and systems, ECT can take advantage of readily-available sort implementations to achieve high performance and reliability [109, 133]. These sort implementations are carefully tuned for a wide range of systems (e.g., low I/O and computational demands), which greatly reduces the engineering effort needed to deliver high performance with ECT.

The unique feature of ECT with respect to sort is that it does not use the original key as the sort key. Instead, it *hashes each key* and uses this hashed result as the sort key. The choice of the hashing function is flexible as long as it provides a reasonably uniform distribution of hash values with very low or zero probability of collisions. Ideally, universal hashing [3] yields the best result for the subsequent steps, while using a conventional cryptographic hash function (e.g., SHA-1) also produces acceptable results for the workloads in our experiments.

We will refer to the hash value of each original key as a *hashkey* or simply as a key. Using hashed keys enables two important design aspects of ECT, as we explain further.

3.1.3 Virtual Bucketing

After sorting the hashkeys,[1] ECT groups sets of adjacent keys into *virtual buckets*, partitioned by their k most significant bits (MSBs), as illustrated in Figure 3.1. As shown in Figure 3.2, the process of looking up a key involves first examining its MSBs to determine which virtual bucket it falls into. ECT consults a two-level index to find, for each virtual bucket, the location of the compressed trie index in memory, and the starting location on flash where keys for this bucket are stored.

The virtual bucketing process does not move data on storage: It merely represents a grouping for the trie indexing. Each virtual bucket's trie is compressed and decompressed independently. Compressing and reading a trie takes time roughly linear in the size of the trie, but larger tries compress more efficiently than small tries. As a result, the virtual bucket size g, which determines how many bits k should be used in bucketing, provides a way to trade space savings and lookup time. We show in Section 3.1.7 that the size of the largest bucket, which relates to the maximum amount of computation required for each lookup, is proportional to the average size of the buckets.

Virtual bucketing adds marginal memory overhead to index the buckets. Since both the in-memory and on-flash locations of the buckets are monotonically increasing, ECT can efficiently encode these offsets by using a two-level index. ECT records the absolute offset of every 64th bucket (when each bucket contains 256 keys on average; i.e., $g = 256$, $k = \log_2(n/256)$), and for each group of 64 buckets, it maintains the offset of individual buckets relative to the first bucket in the same bucket group. The upper level offsets use wide integers (64-bit integers in our implementation), and the lower level offsets use smaller 16-bit integers to save memory. Because

[1] In our implementation, this step occurs while sort is *emitting* the sorted keys; the process need not be complete for virtual bucketing to begin, which eliminates the need for re-reading sorted hashkeys from flash.

of the narrow data type of the lower level, larger buckets (i.e., more than 256 keys per bucket on average) would require more upper level offsets (e.g., every 32nd bucket or more frequent) to avoid an overflow in the lower level. When the average bucket size is 256 keys, the two-level index for in-memory and on-flash locations adds insignificant space overhead of approximately 0.133 bits per key for a practical number ($< 2^{64}$) of keys.

3.1.4 Compressed Tries

Hashkeys in each bucket are indexed by a binary trie. Similar to conventional tries, this trie represents a key with the path from the root node to a leaf node corresponding to that key, where each edge represents one bit of the key. As a result, the trie has a one-to-one correspondence between its leaf nodes and hashkeys; the i-th leaf node among all leaf nodes in the trie corresponds to the i-th smallest hashkey. A key lookup on the trie is essentially finding a leaf node corresponding to the key and counting the other leaf nodes to the left of this leaf node.

However, unlike in the tries used for text search, ECT uses *unique prefixes* of the hashkeys to remove unused information. If a leaf node does not have a sibling node, the leaf node is always removed from the trie, and its parent node becomes a new leaf node. This process significantly decreases the number of nodes in the trie while preserving the relationship between leaf nodes and hashkeys. Further, any internal node that becomes a new leaf node is an ancestor to exactly one leaf node in the original trie of full-length hashkeys. Accordingly, ECT can obtain the correct index of a lookup key by using a prefix of the key when traversing the trie: ECT can stop descending as soon as it arrives at a leaf node.

ECT reduces the amount of memory used to store the trie by applying compression. The compressed trie is represented recursively as follows:

$$Rep(T) = \begin{cases} EC(0, |T|) & \text{if } |T| = 0 \text{ or } 1, \\ EC(|Left(T)|, |T|)\ Rep(Left(T))\ Rep(Right(T)) & \\ & \text{otherwise,} \end{cases}$$

where

$$\begin{aligned} |T| &= \text{Number of keys indexed by } T \\ &\quad \text{(i.e., leaf nodes in } T\text{)}, \\ Left(T) &= \text{Left subtrie of } T, \\ Right(T) &= \text{Right subtrie of } T, \\ EC(S,C) &= \text{Entropy code for symbol } S \text{ in context of } C. \end{aligned}$$

This representation has a pre-order traversal structure similar to the recursive encoding for binary trees [35], but ours differs from the previous technique in the way it describes the left subtrie. When measuring the size of a trie, we use the number of *keys* (i.e., leaf nodes), excluding internal nodes, of the left subtrie, and this enables an efficient entropy coding. Since the trie uses hashkeys, the keys are uniformly distributed over the entire key space; in other words, each bit of every key has the same probability of being 0 and 1. $|Left(T)|$ therefore follows the binomial

distribution of $N = |T|$ and $P = 1/2$. Based on this known statistical distribution, we can encode the key counts using entropy coding. Entropy coding ($EC(S,C)$) uses the contextual information $C = |T|$ to find a suitable code for $S = |Left(T)|$ because the distribution of $|Left(T)|$ depends on $|T|$. Note that for the trivial tries of $|T| \leq 1$, the entropy codes are simply an empty string, as the trivial tries have only one form (i.e., no child nodes).

When decompressing the trie representation for a lookup, C is initially set to the total number of keys in the trie. $|Left(T)|$ can be decoded from the head of the representation, and $|Right(T)|$ can be calculated by $C - |Left(T)|$, since $|Left(T)| + |Right(T)| = |T| = C$. Once $|Left(T)|$ and $|Right(T)|$ are restored, ECT recurses into the subtries by setting C to each subtrie size and using the rest of the trie representation.

The implementation of $EC(S,C)$ varies by the subtrie size ($C = |T|$). In order to use the CPU cache efficiently, we use Huffman coding only for small tries. If Huffman coding is applied to large tries, the Huffman tree or table size required for compressing and decompressing counts grows superlinearly, and this burdens the CPU cache space. Thus, we avoid using Huffman coding for a trie whose size is larger than a certain threshold. We term this threshold *hmax*; when a trie contains no more than *hmax* leaf nodes (i.e., $C \leq hmax$), we use static Huffman tables based on the binomial distributions; for larger tries, we apply Elias-gamma coding [48]. The combination of these two entropy coding techniques reduces cache misses by ensuring Huffman tables can comfortably fit in the CPU cache; we investigate the effect of this combination in Section 3.3.5.

To shrink the trie representation size further, we use the same code for two special cases in each trie size: $|Left(T)| = 0$ and $|Left(T)| = |T|$. That is, if a trie has all keys on either left side or right side, ECT treats them as the same. This does not affect the correctness of the lookup result: It is equivalent to changing the lookup at that location to a "wildcard bit" that can match on either 0 or 1. Such a change preserves correctness because the correct index result will still be found for any key that has been indexed. The change does prevent the trie from being able to reject, in rare cases, keys that are not indexed, but as this is not a design goal of the trie, nor frequent, we judge a worthwhile optimization. This optimization reduces the compressed trie size by 0.4 bits/key.

We do not have to focus on the asymptotic space consumption of the compressed trie representations because the number of keys in a trie is practically small (e.g., around 256 keys), even when ECT indexes a large number of keys (e.g., billions of keys) across many virtual buckets. Therefore, we focus on the expected space consumption for tries in this range. Figure 3.3 plots the expected number of bits required to store a trie as a function of the number of keys stored in it. In this analysis, *hmax* is fixed to 64, and the result does not include the space to store the total number of keys in the trie (i.e., $|T|$ for the top-level). As the number of keys increases, the representation size approaches 2.5 bits/key. Compared to the underlying entropy of the trie, the combination of Huffman and Elias-gamma coding used in ECT achieves good coding efficiency.

3.1.5 Incremental Updates

By employing sort as the data manipulation method, ECT can perform incremental updates efficiently, reusing the previously built sorted dataset on flash. Figure 3.4 illustrates the workflow of incremental updates. Only new items (not the entire dataset) are sorted and are then sequentially merged with the existing sorted items. As this sequential merge does not reorder data globally, it better uses the buffer memory for sort and thus reduces the total amount of I/O. These savings

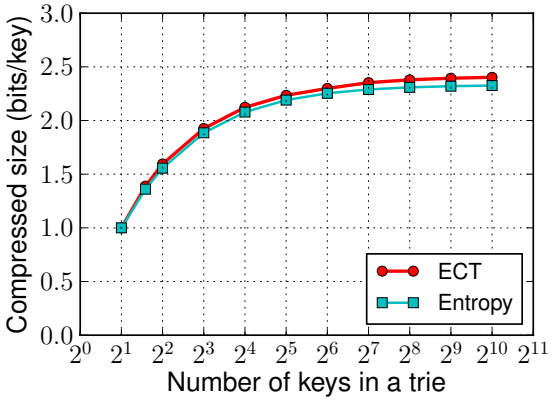


Figure 3.3: Expected size of the compressed representation of a single trie containing a varying number of keys.

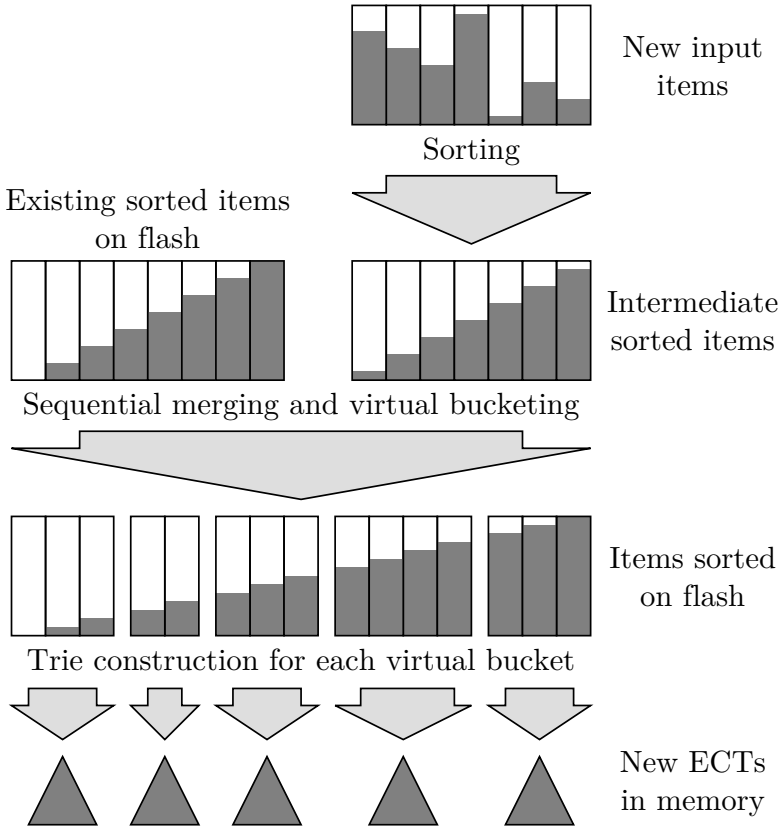


Figure 3.4: Incremental updates of new items to the existing dataset in ECT.

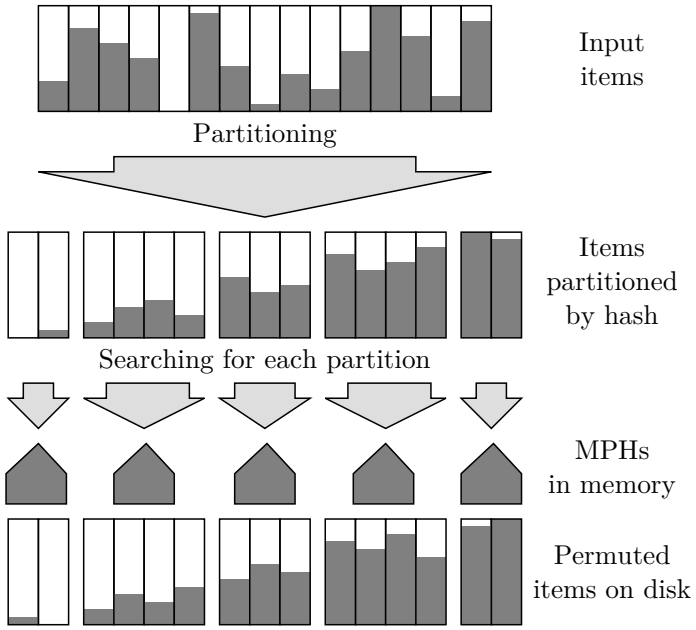


Figure 3.5: Comparison to the workflow of the index construction in External Perfect Hashing (EPH) [25]. Within each bucket, items are not ordered by global hash order, but permuted by a local PHF or MPHF specific to each bucket.

become significant as the size of the existing dataset grows larger than the size of the new dataset, as demonstrated in our evaluation (Section 3.3.4).

3.1.6 Sparse Indexing

As an optimization for small items, ECT can generate more compact indexes by applying the idea of *sparse indexes*: addressing items in the disk block level, rather than the byte level. Since disk and flash drives are block-access oriented devices, they have a certain minimum I/O size (e.g., a block of 512 bytes) defined by their interface. Overheads from I/O operation processing and the physical movement of mechanical parts (e.g., disk heads) further increase the efficient minimum I/O size. Therefore, byte addresses returned by an index save no I/O compared to block addresses; by locating items in the block level, a sparse index can yield the same performance while making it more compact than byte-level dense indexes.

ECT realizes sparse indexing by pruning any subtrie that belongs to the same block. When generating the representation of a subtrie, ECT keeps track of the start and end locations of the items indexed by that subtrie. If the subtrie contains items of the same block (i.e., the start and end locations share the same block), ECT simply omits further generation of the subtrie representation. When handling a lookup, ECT repeats the same process of location tracking; when it finds a subtrie whose items fit in a single block, ECT stops decoding subtries and reads the block from the device so that it can search the queried item within the block.

Figure 3.6 shows expected compressed trie sizes by using sparse indexing. The trie contains 256 keys, and *hmax* is 64. With 8 or more items per block, a trie requires fewer than 1 bit per key in expectation.

The sparse indexing in ECT is stricter than k-perfect hashing [16], which allows *up to* k hash collisions. ECT generates an index with *exactly* k hash collisions (with an exception of the last block that may contain fewer than k items), and this leads to more efficient use of storage space. Further, as this sparse indexing uses the same data layout on flash, ECT can generate a new dense or sparse index on already indexed items without repeating construction process, allowing it to adapt to memory and performance requirements quickly.

3.1.7 Maximum Size of Virtual Buckets

The size of the largest virtual buckets is $\Theta(n/m)$, where n is the total number of keys indexed by ECT, and m is the number of virtual buckets ($m = 2^k$ when using the k MSBs to determine the virtual bucket of a key). We can formulate the problem using a conventional balls-into-bins model by treating each key as a ball (total n balls) and each virtual bucket as a bin (total m bins). It is known that when $n \geq m \log_2 m$, the maximum number of balls in each bin is $\Theta(n/m)$ [3, 120]. In our setting, let m be n/w (each bin contains w balls on average), where w is the width of a machine word (e.g., 64)—this will require using $\log_2(n/w)$ MSBs for determining to which virtual bucket each key belongs. Since $n/w = 2^{\log_2 n - \log_2 w}$, $m \log_2 m = (n/w)\log_2(n/w) = 2^{\log_2 n - \log_2 w} \cdot (\log_2 n - \log_2 w) = 2^{\log_2 n - \log_2 w + \log_2(\log_2 n - \log_2 w)}$. Observe $n \leq 2^w$ (due to addressing items by a machine word), $2^{\log_2 n - \log_2 w + \log_2(\log_2 n - \log_2 w)} \leq 2^{\log_2 n - \log_2 w + \log_2(w - \log_2 w)} < 2^{\log_2 n} = n$. Thus, $n \geq (n/w)\log_2(n/w)$, and we get the maximum load of each virtual bucket of $\Theta(n/(n/w)) = \Theta(w)$. In other words, as long as the average bucket size ($g = n/m$) is no smaller than the number of bits in a machine word, the size of the largest bucket will remain proportional to the average size of all buckets.

3.2 Comparison to Other Schemes

In this section, we discuss similarities and differences between ECT and other fast external hashing schemes, summarized in Table 3.1.[2]

3.2.1 External Perfect Hashing

External Perfect Hashing (EPH) [25] is a scalable external hashing scheme that requires $O(1)$ I/O operations for item retrieval and uses a small amount of memory for the index. As shown in Figure 3.5, EPH first partitions items into buckets using the hash values of their keys. For each bucket, EPH constructs a perfect hash function (PHF) or a minimal perfect hash function (MPHF), depending on the memory and storage space requirement. Since PHFs and MPHFs result in a specific offset for each key, the keys in the buckets are permuted (reordered) in their PHF/MPHF order and stored on disk. EPH achieves 3.8 bits/key of memory use, which is 27% higher than 2.5 bits/key used by ECT.

[2]The CPU requirement of MMPH is from the original MMPH paper [14].

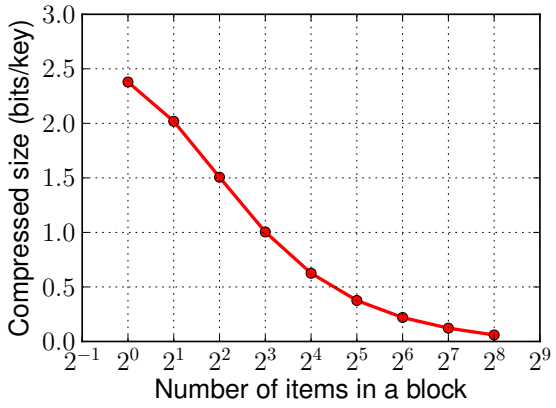


Figure 3.6: Expected size of the compressed representation of a single trie containing 256 items with a varying number of items in a block.

The external memory (EM) algorithm [27], which is an EPH variant that shares the same partitioning approach, still requires 3.1–3.3 bits/key as MPH to index a large number of keys that do not fit in memory.

Unlike EPH, ECT sorts all items in hash order, eliminating the additional permutation step. As we show in Section 3.3, EPH can benefit from being modified to use external sort, but the permutation step adds overhead as indicated in the experimental results (Section 3.3.2).

When performing incremental updates, EPH requires additional steps to resolve item movement within and between buckets. There is no stable ordering within a bucket across incremental updates in EPH because the order of each key is determined by an MPHF specific to the bucket, and this involves re-permutation of the items within a bucket whenever a key is added to or removed from the bucket. Worse, when the size of a bucket exceeds a certain limit or shrinks to a very small size, EPH must split or merge the buckets; as this data manipulation must be done separately from the partitioning step of EPH, support for incremental updates in EPH complicates the system design and implementation.

3.2.2 Monotone Minimal Perfect Hashing

Monotone Minimal Perfect Hashing (MMPH) [14, 15, 69], like ECT, uses sorting as its data manipulation method, and builds its index on a sorted table. While ECT uses hash order for sorting and trie construction, MMPH uses original keys, and thus, requires a variable amount of computation and memory space depending on the key distribution and characteristics. Among various low-level indexing methods for MMPH, "hollow tries" provides the smallest hash function size—4.4 bits/key—for random keys [14], which is the best workload for minimizing the index size.

Although MMPH shares several characteristics with ECT in that both use sort, we focus our evaluation on comparisons between ECT and EPH because both yield smaller index sizes than MMPH.

	External Perfect Hashing (MPH version)	Monotone Minimal Perfect Hashing (MMPH)	Entropy-Coded Tries (ECT)
Data manipulation method	Partitioning with hash + permutation by MPHFs	Sorting in original key order	Sorting in hash order
Low-level index type	MPH	Various	Compressed trie
I/O complexity for construction	$O(n)$	$O(n)$	$O(n)$
Memory requirement for a large set of keys	3.8 bits/key	$\geq$ 4.4 bits/key	2.5 bits/key
CPU requirement for an in-memory lookup	0.6 µs/lookup	7 µs/lookup	7 µs/lookup

Table 3.1: Summary of the comparison between ECT and other fast external hashing schemes.

Component	Specification
CPU	Intel Core i7 860 (quad-core) @ 2.80 GHz
DRAM	DDR3 4 GiB
SSD	RAID-0 array of three Intel SSD 510 120 GB (MLC)

Table 3.2: Experimental system setup.

3.3 Evaluation

This section presents the performance evaluation of an external dictionary using ECT as its index. As a comparison, we modified the original EPH implementation, which only supported a hash function interface, to provide the full functionality of an external dictionary. Our modified version of EPH can operate using its custom partitioning as well as using external sort similarly to ECT, as sort results are compatible with partition results. In addition, we implemented optional incremental construction in ECT and extended EPH to support the same incremental construction functionality, because the original EPH implementation only supported index construction from scratch.

Throughout the evaluation, we explicitly indicate binary prefixes (powers of 2) using "i" (e.g., MiB, GiB) to avoid confusion with SI prefixes (powers of 10) (e.g., MB, GB).

3.3.1 Experiment Setup

Table 3.2 shows the hardware configuration for the experiments. All input, output, and temporary data are stored on a RAID-0 array consisting of three SATA SSDs, each of which provides up to 265 MB/s sequential read and 200 MB/s sequential write throughput.

We use the `WEBSPAM-UK2007` URL list [142] as input keys (on average 111 bytes per URL) and associated values of 1000 bytes with each key.[3] Due to the large size of the key-value pairs, we use up to 24 million unique URLs at the beginning of the URL list; as we use hashkeys, using

[3]Note that using a 1000-byte value for each key significantly increases both data size and construction cost compared to prior studies [14, 25], which stored no key (and value) data in external storage; our study stores both key and value data on flash to provide a dictionary interface (retrieval of a value associated with a key).

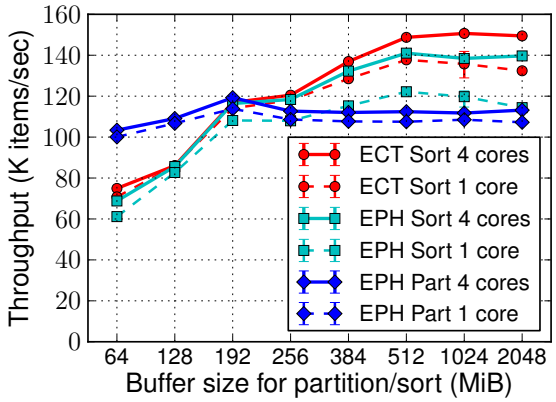


Figure 3.7: Construction performance with varying buffer size for partition/sort.

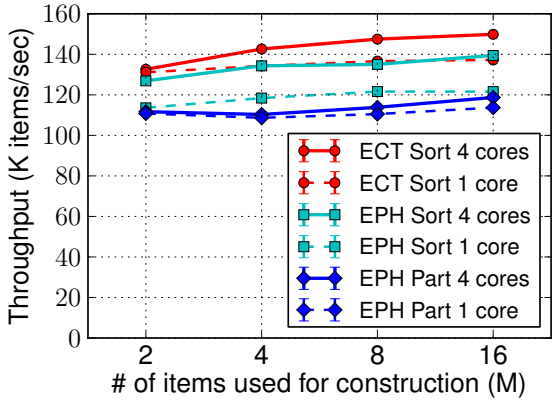


Figure 3.8: Construction performance with a varying number of items.

the first part of the URL list does not introduce skew in the input data. Unless specified, this input is the standard workload in this section.

In addition, to examine the performance with a large number of items, we use 1 billion small key-value pairs, which consist of 64-byte keys and 4-byte values. Each key is hashed into 12 bytes at the beginning of the index construction, constituting a 16-byte key-value pair on flash during and after construction to support later key-value retrieval. While we could use the sparse indexing technique (Section 3.1.6), we avoid using it for ECT to make a fair comparison with EPH, which does not have an implementation for the technique. This workload is denoted as `small-item`, and its experiment result is shown in Section 3.3.6.

All experiments were performed on Ubuntu 11.04 (64-bit), and we used Nsort [109] as an external sort implementation. Each configuration of the experiments involving time measurements had three runs, and the range of the results is indicated with an error bar.

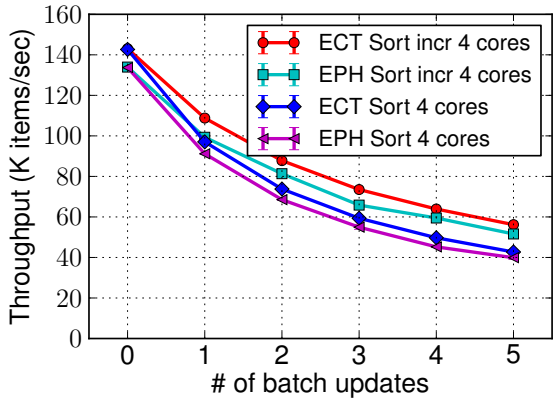


Figure 3.9: Construction performance for different combinations of indexing and incremental construction schemes. Each batch contains 4 M new items.

3.3.2 Construction

Initial construction of the external hashing involves a large amount of I/O for input data, temporary files, and output data. I/O is a major bottleneck for the overall construction process. Both Nsort and EPH provide a settable parameter that controls the size of in-memory buffer space to use I/O efficiently for sorting and partitioning; thus, determining the appropriate buffer size is important. An excessively large buffer may decrease overall performance because the OS page cache, which competes for the same main memory space, is also important for reading the input data and writing the final output.

Figure 3.7 shows the construction performance when using different buffer sizes for the partition and sort stage. This experiment used 16 million key-value pairs, whose size (about 16 GB) far exceeds the total memory size (4 GiB). For both ECT and EPH, the sort versions that use only 1 core (dotted lines) achieve the best performance using 512 MiB for buffering, while the partition-based EPH (EPH Part) performs best with a smaller buffer size of 192 MiB (Table 3.3). This difference is because Nsort internally uses direct I/O that bypasses the OS page cache, and thus its performance is less susceptible to the reduced amount of the OS page cache when the in-memory sort buffer size is large. With 4 cores (solid lines), ECT works best with 1024 MiB buffer size. From this result, we can observe major performance boosts when using multiple cores due to Nsort's internal optimization, but not with EPH's custom partitioning, because EPH's custom data partitioning code uses 1 core and does not directly benefit from multiple cores. This demonstrates the performance and engineering advantages of using a well-understood and well-optimized primitive such as sorting for the main data manipulation.

In the subsequent experiments, we used the best buffer memory size for each configuration. Figure 3.8 plots the construction performance with varying dataset size; we set the buffer size based on the large dataset of 16 M items, which shows that smaller dataset sizes have an insignificant effect on the construction speed.

	Throughput @ Buffer size
ECT Sort 4 cores	151 k items/s @ 1024 MiB
ECT Sort 1 core	138 k items/s @ 512 MiB
EPH Sort 4 cores	141 k items/s @ 512 MiB
EPH Sort 1 core	122 k items/s @ 512 MiB
EPH Part 4 cores	119 k items/s @ 192 MiB
EPH Part 1 core	114 k items/s @ 192 MiB

Table 3.3: Best construction performance of each scheme.

	Throughput	Total CPU time
EPH	69.16 k queries/s	139.76 s
ECT	64.02 k queries/s	217.75 s
(difference)	-7.4%	+55.8%

Table 3.4: Random lookup performance with random queries using 16 threads.

3.3.3 Index Size

Despite its faster construction speed, ECT generates a smaller index than EPH. To index 16 million items of the given dataset, ECT used 5.02 MB (2.51 bits/key); EPH required 7.83 MB (3.92 bits/key) excluding a fixed-size lookup table of 3 MiB, or 10.98 MB including the lookup table (5.49 bits/key in total). ECT's space consumption is at least 36% lower than EPH's and is close to the analytical result shown in Figure 3.3. These differences can determine whether an index fits in the CPU cache, and they grow proportionally to the number of items in larger datasets.

3.3.4 Incremental Updates

For datasets that are not fully static, incremental updates improve the reconstruction speed of the external hashing substantially. Figure 3.9 plots performance boosts with incremental updates. The workload constructs an index using 4 M items. Then, it adds another 4 M items on each batch to update the existing dataset. It repeats this update up to 5 times until the final dataset size reaches 24 M items. As clearly shown, using incremental updates outperforms versions that must rebuild the sorted/partitioned dataset from scratch rather than reusing the previously organized data. The performance gap increases as the number of updates increases—more than 20% with 5 updates. Therefore, allowing incremental updates is important to improving the performance of storage systems with dynamic data.

3.3.5 Space vs. Lookup Speed Tradeoff

Table 3.4 shows the full-system lookup performance difference between ECT and EPH, using 8 M random queries on 16 M items. Each lookup includes data item retrieval from flash, thus incurring I/O. The experiment used 16 threads to take advantage of the I/O parallelism of flash drives [91]. While both algorithms exhibit lookup speed exceeding 64 k queries/s, ECT is 7.4%

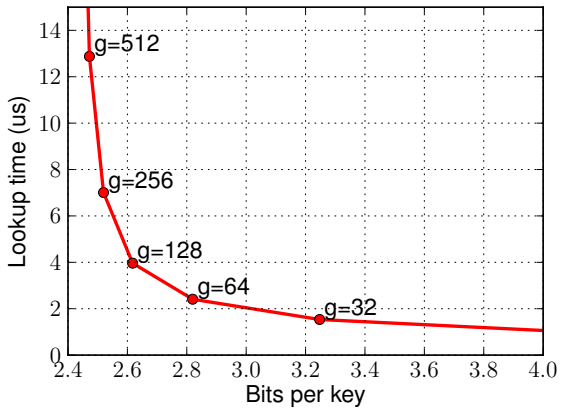


Figure 3.10: The tradeoff between size and in-memory lookup performance on a single core when varying average bucket size (g) with $hmax = 64$.

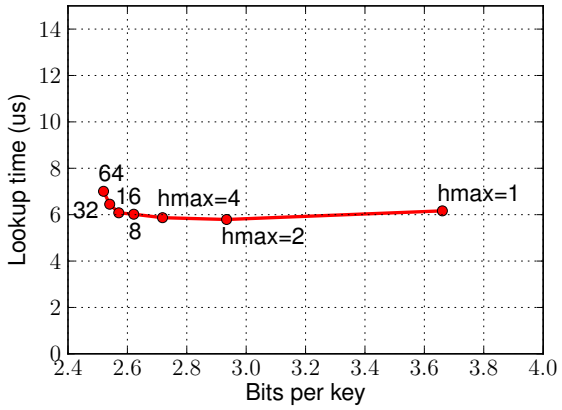


Figure 3.11: The tradeoff between size and in-memory lookup performance on a single core when varying maximum trie size for Huffman coding ($hmax$) with $g = 256$.

slower than EPH. This difference mostly comes from the higher cost of ECT's lookup process, as shown in the total CPU time, which also includes the CPU overhead of I/O processing in the kernel. However, the difference in the actual lookup speed is much smaller than the difference in the total CPU time, since the system's CPU is largely underutilized (i.e., external lookup is an I/O-bound task). Therefore, for a small extra end-to-end lookup cost, ECT brings considerable savings in the index size (36% smaller index size than EPH). The amount of I/O performed was the same in both schemes, as they incur a single random I/O per lookup.

When a bigger and faster array of flash drives is used, slow hash function evaluation may cause the underutilization of the storage array. ECT provides two knobs—the virtual bucket size (g) and the maximum trie size at which to apply Huffman coding ($hmax$)—to allow the system to obtain a balance between the in-memory data structure size and per-lookup computation. To highlight the effect of adjusting knobs, we measure in-memory (no I/O) lookup performance using a single core only.

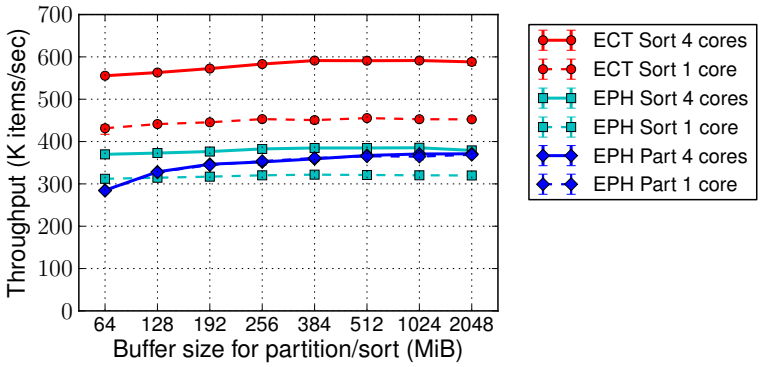


Figure 3.12: Construction performance with varying buffer size for partition/sort for small items.

	Throughput @ Buffer size
ECT Sort 4 cores	591 k items/s @ 1024 MiB
ECT Sort 1 core	455 k items/s @ 512 MiB
EPH Sort 4 cores	385 k items/s @ 1024 MiB
EPH Sort 1 core	322 k items/s @ 384 MiB
EPH Part 4 cores	371 k items/s @ 2048 MiB
EPH Part 1 core	369 k items/s @ 2048 MiB

Table 3.5: Best construction performance of each scheme for small items.

Figure 3.10 shows that by setting the average virtual bucket size (g) to 256 keys, ECT requires as little as 2.5 bits per item and CPU consumption for an in-memory lookup of 7 µs. This lookup speed allows the system to support up to 67 k queries per second per core without considering I/O costs. With such low memory use by ECT, if a system has 4 GiB of DRAM to store the ECT index, it can index 13.7 billion items. By having a smaller number of keys in each virtual bucket, the system can trade the memory efficiency for even higher lookup speed.

Figure 3.11 shows diminishing returns of using a high value for the maximum trie size to apply Huffman coding (*hmax*). With a moderate *hmax* value (8 to 64), the system can achieve small index size as well as low lookup time.

3.3.6 Small Items

With small items of 64-byte key-value pairs, the construction process is more dependent on computation than with large items. As shown in Figure 3.12 and Table 3.5, ECT using 4 cores exceeds any EPH scheme's performance by far; ECT's simple construction process handles 591 k key-value items per second (28 minutes in total), achieving 54% higher construction speed than the best EPH scheme.

ECT demonstrated its memory efficiency again with small items. ECT required 35% less memory space than EPH; ECT's index size was 314 MB (2.51 bits/key), whereas EPH required

481 MB (3.85 bits/key) when excluding a fixed-size lookup table of 3 MiB, or 484 MB with the lookup table (3.87 bits/key in total).

3.4 Summary

Entropy-Coded Tries (ECT) provide a practical external hashing scheme by using a new trie-based indexing technique on hash-sorted items. Our trie compression yields a very compact hash function occupying 2.5 bits per key, with fast lookup time (around 7 µs), making ECT suitable for datasets stored on high-speed storage devices such as flash drives. Using sort as the main data manipulation method substantially saves engineering effort, and this enables ECT to use I/O and multi-core CPUs efficiently for index construction and incremental index updates.

Chapter 4

Analyzing Multi-Stage Log-Structured Designs

Log-structured store designs provide fast write and easy crash recovery for block-based storage devices that have considerably higher sequential write speed than random write speed [123]. In particular, multi-stage versions of log-structured designs, such as LSM-tree [111], COLA [17], and SAMT [128], strive to balance read speed, write speed, and storage space use by segregating fresh and old data in multiple append-only data structures. These designs have been widely adopted in modern datastores including LevelDB [65], RocksDB [54], BigTable [31], HBase [135], and Cassandra [88].

Given the variety of multi-stage log-structured (MSLS) designs, a system designer is faced with a problem of plenty, raising questions such as: Which design is best for this workload? How should the systems' parameters be set? How sensitive is that choice to changes in workload? Our goal in this chapter is to move toward answering these questions and more through an improved—both in quality and in speed—analytical method for understanding and comparing the performance of these systems. This analytical approach can help shed light on how different design choices affect the performance of today's systems, and it provides an opportunity to optimize (based on the analysis) parameter choices given a workload. For example, in Section 4.6, we show that a few minutes of offline analysis can find improved parameters for LevelDB that decrease the cost of inserts by up to 9.4–30.5%. As another example, in Section 4.7, we reduce the insert cost in RocksDB by up to 32.2% by changing its system design based upon what we have learned from our analytic approach.

Prior evaluations of MSLS designs largely reside at the two ends of the spectrum: (1) asymptotic analysis and (2) experimental measurement. *Asymptotic analysis* of an MSLS design typically gives a big-O term describing the cost of an operation type (e.g., query, insert), but previous asymptotic analyses do not reflect real-world performance because they assume the worst case. *Experimental measurement* of an implementation produces accurate performance numbers, which are often limited to a particular implementation and workload, with lengthy experiment time to explore various system configurations.

This chapter proposes a new evaluation method for MSLS designs that provides accurate and fast evaluation without needing to run the full implementations. Our approach uses new analytic primitives that help model the dynamics of MSLS designs. We build upon this model

by combining it with a nonlinear solver to help automatically optimize system parameters to maximize performance.

This chapter makes four key contributions:

- New analytic primitives to model creating the log structure and merging logs with redundant data (Section 4.2);
- System models for LevelDB, COLA, and SAMT, representative MSLS designs, using the primitives (Section 4.3, Section 4.4);
- Optimization of system parameters with the LevelDB model, improving the real system performance (Section 4.6); and
- Application of lessons from the LevelDB model to the RocksDB system to reduce its write cost (Section 4.7).

In addition, we revisit SILT's multi-store design in Section 4.5) to provide its accurate analysis model.

4.1 Background

This section introduces a family of multi-stage log-structured designs and their practical variants, and explains metrics commonly used to evaluate these designs.

4.1.1 Multi-Stage Log-Structured Designs

A multi-stage log-structured (MSLS) design is a storage system design that contains multiple append-only data structures, each of which is created by sequential writes; for instance, several designs use sorted arrays and tables that are often called *SSTables* [65, 128]. These data structures are organized as *stages*, either logically or physically, to segregate different classes of data—e.g., fresh data and old data, frequently modified data and static data, small items and large items, and so forth. *Components* in LSM-tree [111] and *levels* in many designs [17, 65, 128] are examples of stages.

MSLS designs exploit the fast sequential write speed of modern storage devices. On hard disk and flash drives, sequential writes are up to an order of magnitude faster than random writes. By restricting most write operations to incur only sequential I/O, MSLS can provide fast writes.

Using multiple stages reduces the I/O cost for data updates. Frequent changes are often contained within a few stages that either reside in memory and/or are cheap to rewrite—this approach shares the same insight as the generational garbage collection used for memory management [84, 90]. The downside is that the system may have to search in multiple stages to find a single item because the item can exist in any of these stages. This can potentially reduce query performance.

The system moves data between stages based upon certain criteria. Common conditions are the byte size of the data stored in a stage, the age of the stored data, etc. This data migration typically reduces the total data volume by merging multiple data structures and reducing the redundancy between them; therefore, it is referred to as "compaction" or "merge."

MSLS designs are mainly classified by how they organize log structures and how and when they perform compaction. The data structures and compaction strategy significantly affect the cost of various operations, characterizing strengths and weaknesses of store designs.

Log-Structured Merge-Tree

The log-structured merge-tree (LSM-tree) [111] is a write-optimized store design with two or more components, each of which is a tree-like data structure [110]. One component (C_0) resides in memory, and the rest of the components (C_1,C_2,...) are stored on disk. Each component can hold a set of items, and multiple components can contain multiple items of the same key. A lower-numbered component always stores a newer version of the item than any higher-numbered component does.

For query processing, LSM-tree searches in potentially multiple components. It starts from C_0 and stops as soon as the desired item is found in one of the components.

Handling inserts involves updating the in-memory component and merging the data between components. A new entry is inserted into C_0 (and is also logged to disk for crash recovery), and the new item is migrated over time from C_0 to C_1, from C_1 to C_2, and so on. Note that frequent and redundant updates of the same items are coalesced in C_0 without spilling them to the disk; cold data, in contrast, remains in C_1 and later components which reside on low-cost disk.

The data merge in LSM-tree is mostly a sequential I/O operation. The data from C_l is read and merged into C_{l+1}, using a "rolling merge" that creates and updates nodes in the C_{l+1} tree incrementally in the key space.

The authors of LSM-tree suggested maintaining component sizes to follow a geometric progression. The size of a component is r times larger than the previous component size, where r is commonly referred to as a "growth factor" that typically lies between 10 and 20. With such size selection, the expected I/O cost per insert by the data migration is $O((r+1)\log_r N)$, where N is the size of the largest component, i.e., the total number of unique keys. The worst-case lookup incurs $O(\log_r N)$ random I/O by accessing all components, if finding an item in a component costs $O(1)$ random I/O.

LevelDB [65] is an important variant of the LSM-tree.

COLA and SAMT

The cache-oblivious lookahead array (COLA) [17] is a generalized and improved binomial list [18]. Like LSM-tree, COLA has multiple levels whose count is $\lceil \log_r N \rceil$, where r is the growth factor. Each level contains zero or one SSTable. Unlike LSM-tree, however, COLA uses the merge count as the main compaction criterion; a level in COLA accepts $r-1$ merges with the lower level before the level is merged into the next level. COLA uses fractional cascading [33] to reduce lookup costs by storing samples of later levels in earlier levels.

COLA has roughly similar asymptotic complexities to LSM-tree's. A query in COLA may cost $O(\log_r N)$ random I/O per lookup if looking up a level costs $O(1)$ random I/O. COLA's data migration costs $O((r-1)\log_r N)$ I/O per insert. COLA's growth factor is usually chosen between 2 and 4.

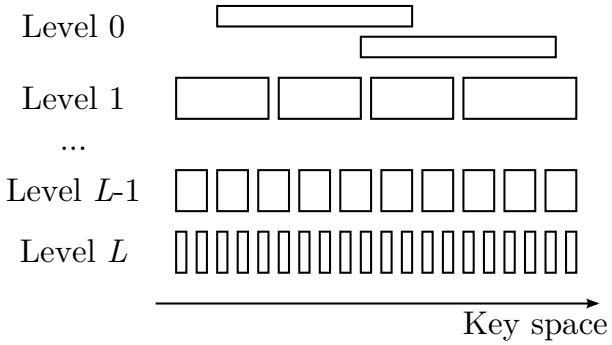


Figure 4.1: A simplified overview of LevelDB data structures. Each rectangle is an SSTable. Note that the x-axis is the key space; the rectangles are not to scale to indicate their byte size. The memtable and logs are omitted.

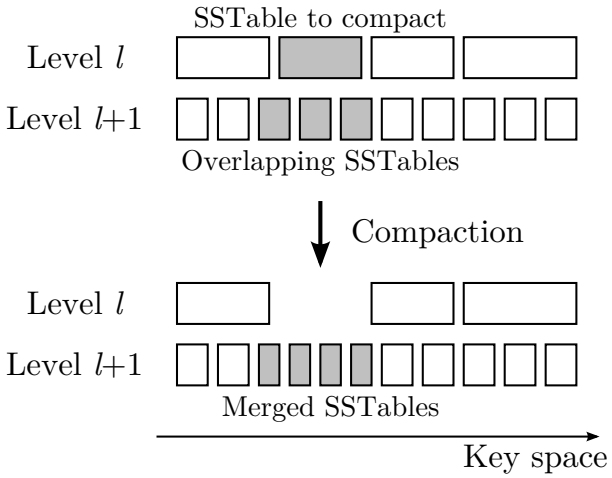


Figure 4.2: Compaction between two levels in LevelDB.

The Sorted Array Merge Tree (SAMT) [128] is similar to COLA but peforms compaction differently. Instead of eagerly merging data to have a single log structure per level, SAMT keeps up to r SSTables before merging them and moving the merged data into the next level. Therefore, a lookup costs $O(r \log_r N)$ random I/O, whereas the per-update I/O cost decreases to $O(\log_r N)$.

A few notable systems implementing a version of COLA and SAMT are HBase [135] and Cassandra [88, 134].

LevelDB

LevelDB [65] is a well-known variant of LSM-tree. It uses an in-memory table called a memtable, on-disk log files, and on-disk SSTables. The memtable plays the same role as C_0 of LSM-tree, and write-ahead logging is used for recovery. LevelDB organizes multiple levels that correspond to the components of LSM-tree; however, as shown in Figure 4.1, LevelDB uses a set of SSTables instead of a single tree-like structure for each level, and LevelDB's first level (level-0) is similar to SAMT's level because it can contain duplicate items across multiple SSTables.

Handing data updates in LevelDB is mostly similar to LSM-tree with a few important differences. Newly inserted data is stored in the memtable and appended to a log file. When the log size exceeds a threshold (e.g., 4 MiB[1]), the content of the memtable is converted into an SSTable and inserted to level-0. When the table count in level-0 reaches a threshold (e.g., 4), LevelDB begins to migrate the data of level-0 SSTables into level-1. For level-1 and later levels, when the aggregate byte size of SSTables in a level reaches a certain threshold, LevelDB picks an SSTable from that level and merges it into the next level. Figure 4.2 illustrates the compaction process; it takes all next-level SSTables whose key range overlaps with the SSTable being compacted, replacing the next-level SSTables with new SSTables containing merged items.

The SSTables created by compaction follow several invariants. A new SSTable has a size limit (e.g., 2 MiB), which makes the compaction process incremental. An SSTable cannot have more than a certain amount of overlapping data (e.g., 20 MiB) in the next level, which limits the future cost of compacting the SSTable.

LevelDB compacts SSTables in a circular way within the key space for each level. Fine-grained SSTables and round-robin SSTable selection have interesting implications in characterizing LevelDB's write cost as discussed in Section 4.3.

There are several variants of LevelDB. A popular version is RocksDB [54], which claims to improve write performance with better support for multithreading. Unlike LevelDB, RocksDB picks the largest SSTable available for concurrent compaction. We discuss the impact of this strategy in Section 4.7. RocksDB also supports "universal compaction," an alternative compaction strategy that trades read performance for faster writes by taking a compaction strategy is closer to COLA/SAMT than to LSM-tree/LevelDB.

We choose to apply our analytic primitives and modeling techniques to LevelDB in Section 4.3 because it creates interesting and nontrivial issues related to its use of SSTables and incremental compaction. We briefly discuss the model of COLA and SAMT in Section 4.4.

4.1.2 Common Evaluation Metrics

We briefly describe common metrics to evaluate MSLS designs. This chapter focuses on analytic metrics (e.g., per-insert cost factors) more than on experimental metrics (e.g., insert throughput represented in MB/s or kOPS).

Queries and inserts are two common operation types. A query asks for one or more data items, which can simply return "not found" as well. An insert stores new data or update existing item data. While it is hard to define a cost metric for every type of query and insert operation, prior studies have extensively used two metrics defined for the *amortized I/O cost per processed item*, namely read amplification and write amplification.

Read amplification (RA) is the expected number of random read I/O operations to serve a lookup query under an assumption that the total data size is much larger than the system memory size, which translates to the expected I/O overhead of query processing [29, 91]. RA is based on the fact that random I/O access on disk and flash is a critical resource in handling a query and imposes a high impact on the query performance.

[1]Mi denotes 2^{20}. k, M, and G denote 10^3, 10^6, and 10^9, respectively.

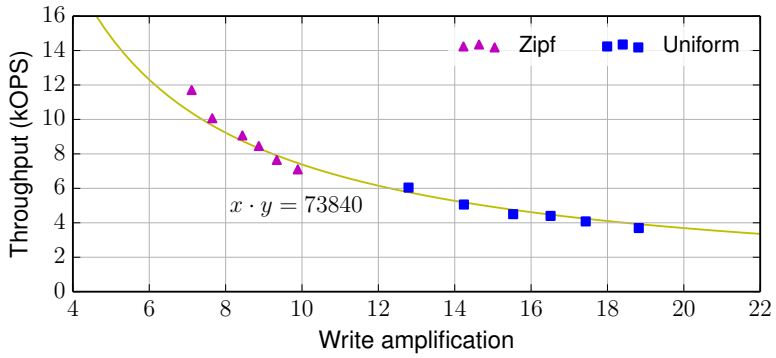


Figure 4.3: Write amplification is an important metric; an increase in write amplification leads to a decrease in insert throughput on LevelDB.

Write amplification (WA) is the expected amount of data written to disk or flash per insert, which measures the I/O overhead of insert processing. Its concept originates from a metric to measure the efficiency of the flash translation layer (FTL), which stores blocks in a log structure-like manner; WA has been adopted later in key-value store studies to project insert throughput and estimate the life expectancy of underlying flash drives [54, 65, 91].

Write amplification and insert throughput are inversely related. Figure 4.3 shows LevelDB's insert throughput for 1 kB items on a fast flash drive.[2] We vary the total data volume from 1 GB to 10 GB and examine two distributions for the key popularity, uniform and Zipf. Workloads that produce higher write amplification (e.g., larger data volume and/or uniform workloads) have lower throughput.

In this chapter, our main focus is write amplification. Unlike read amplification whose effect on actual system performance can be reduced by dedicating more memory for caching, the effect of write amplification cannot be mitigated easily without changing the core system design because the written data must be eventually flushed to disk/flash to ensure durability.

4.2 Analytic Primitives

Our goal in later sections is to create simple but accurate models of the write amplification of different MSLS designs. To reach this goal, we first present three new analytic primitives, Unique, Unique^{-1}, and Merge, that form the basis for those models. In Sections 4.3 and 4.4, we show how to express the insert and growth behavior of LevelDB and COLA/SAMT using these primitives.

4.2.1 Roles of Redundancy

Redundancy has an important effect on the behavior of an MSLS design. Any given table store (SSTable, etc.) contains at most one entry for a single key, no matter how many inserts were applied for that key. Similarly, when compaction merges tables, the resulting table will also

[2] We use Intel SSDSC2BB160G4T with `fsync` enabled for LevelDB.

contain only a single copy of the key, no matter how many times it appeared in the tables that were merged. An accurate model must therefore consider redundancy.

Asymptotic analyses in prior studies ignore the roles of redundancy in determining operation costs. Most analyses assume that compactions observe no duplicate keys from insert requests and input tables being merged [17, 65]. One consequence of such assumptions is that asymptotic analysis gives the same answer regardless of skew in the key popularity; it ignores whether all keys are equally popular or some keys are more popular than others. It also estimates only an upper bound on the compaction cost—duplicate keys mean that less total data is written, lowering real-world write amplification.

We first clarify our assumptions and then explain how we quantify the effect of redundancy.

4.2.2 Notation and Assumptions

Let K be the key space. Without loss of generality, K is the set of all integers in $[0, N-1]$, where N is the total number of unique keys that the workload uses.

A discrete random variable X maps an insert request to the key referred to by the request. f_X is the probability mass function for X, i.e., $f_X(k)$ for $k \in K$ is the probability of having a specific key k for each insert request, assuming the keys in the requests are independent and identically distributed (i.i.d.) and have no spatial locality in popularity. As an example, a Zipf key popularity is defined as $f_X(h(i)) = \frac{1/i^s}{\sum_{n=1}^{N} 1/n^s}$, where s is the skewness and h maps the rank of each key to the key.[3] Since there is no restriction on how f_X should look, it can be built from a key popularity distribution inferred by an empirical workload characterization [8, 124, 145].

Without loss of generality, $0 < f_X(k) < 1$. We can remove any key k satisfying $f_X(k) = 0$ from K because k will never appear in the workload. Similarly, $f_X(k) = 1$ degenerates to a workload with only a single unique key, which is trivial to analyze.

A table is a set of the items that contains no duplicate keys. Tables are constructed from a sequence of insert requests or merges of other tables.

L refers to the total number of standard levels in an MSLS design. *Standard* levels include only the levels that follow prevailing invariants of the design; for example, the level-0 in LevelDB does not count towards L because level-0 contains overlapping tables, while other levels do not, and has a different compaction trigger that is based on the table count in the level, not the aggregate size of tables in a level. L has close relationship with the lookup performance, i.e., read amplification; an MSLS design may have to make L random I/Os to retrieve an item that exists only in the last level (unless the design uses additional data structures such as Bloom filters [22]).

To avoid complicating the analysis, we assume that all operations are synchronous and all items have equal size (e.g., 1000 bytes). This assumption is consistent with YCSB [38], a widely-used key-value store benchmark. We discuss this limitation in Section 4.8.

[3] Note that $s = 0$ leads to a *uniform* key popularity, i.e., $f_X(k) = 1/N$. We use $s = 0.99$ frequently to describe a "skewed" or simply "Zipf" distribution for the key popularity, which is the default skewness in YCSB [38].

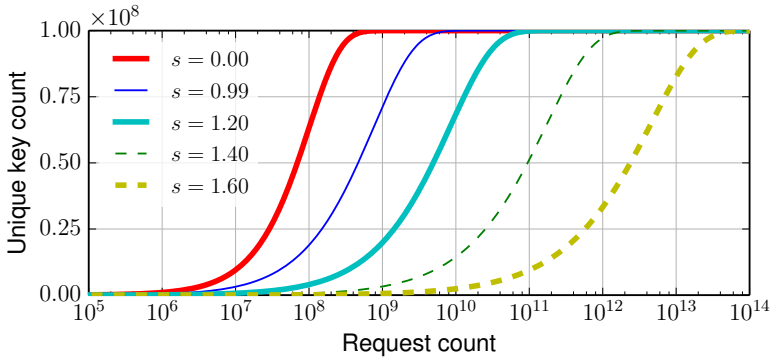


Figure 4.4: Unique key count as a function of request count for 100 million unique keys, with varying Zipf skewness (s).

4.2.3 Counting Unique Keys

A sequence of insert requests may contain duplicate keys. The requests with duplicate keys overwrite or modify the stored values. When storing the effect of the requests in a table, only the final (combined) results survive. Thus, a table can be seen as a set of distinct keys in the requests.

We first formulate Unique, a function describing the expectation of the number of unique keys that appear in p requests:

Theorem 4.1.

$$\text{Unique}(p) = N - \sum_{k \in K} (1 - f_X(k))^p \text{ for } p \geq 0.$$

Proof. Key k counts towards the unique key count if k appears at least once in a sequence of p requests, whose probability is $1-(1-f_X(k))^p$. Therefore, $\text{Unique}(p) = \sum_{k \in K} (1-(1-f_X(k))^p) = N - \sum_{k \in K} (1-f_X(k))^p$. □

Figure 4.4 plots the number of unique keys as a function of the number of insert requests for 100 million unique keys ($N = 10^8$). We use Zipf distributions with varying skewness. The unique key count increases as the request count increases, but the increase slows down as the unique key count approaches the total unique key count. The unique key count with less skewed distributions increases more rapidly than with more skewed distributions until it is close to the maximum.

In the context of MSLS designs, Unique gives a hint about how many requests (or how much time) it takes for a level to reach a certain size from an empty state. With no or low skew, a level quickly approaches its full capacity and the system initiates compaction; with high skew, however, it can take a long time to accumulate enough keys to trigger compaction.

We examine another useful function, Unique^{-1}, which is the inverse function of Unique. $\text{Unique}^{-1}(u)$ estimates the expected number of requests to observe u unique keys in the requests.[4] By extending the domain of Unique to the real numbers, we can ensure the existence of Unique^{-1}:

Lemma 4.2. $\text{Unique}^{-1}(u)$ exists for $0 \leq u < N$.

[4]The solution of Unique^{-1} is often similar to that of a generalized coupon collector's problem (CCP) [59]; however, it is *not* always identical to CCP. A generalized CCP terminates *as soon as* a certain number of unique items has been collected, whereas Unique^{-1} is merely defined as the inverse of Unique. In numeric analysis, a solution of the generalized CCP is typically smaller than that of Unique^{-1} due to the eager termination of CCP.

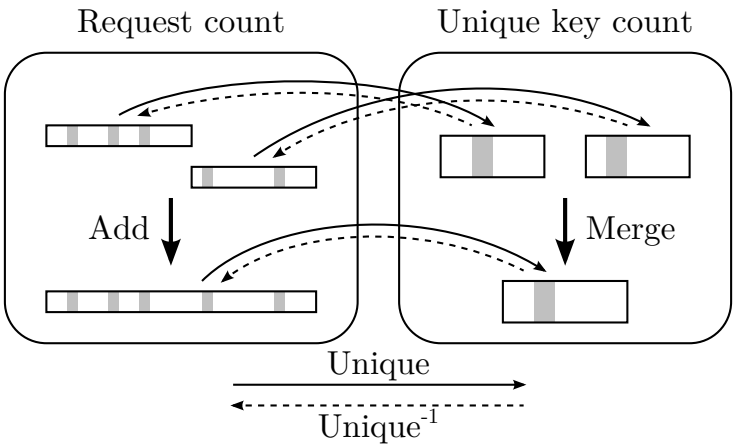


Figure 4.5: Isomorphism of Unique. Gray bars indicate a certain redundant key.

Proof. Suppose $0 \leq p < q$. $(1 - f_X(k))^q < (1 - f_X(k))^p$ because $0 < 1 - f_X(k) < 1$. $\text{Unique}(q) - \text{Unique}(p) = -\sum_{k \in K}(1 - f_X(k))^q + \sum_{k \in K}(1 - f_X(k))^p > 0$. Thus, Unique is a strictly monotonic function that is defined over $[0, N)$. □

We further extend the domain of Unique^{-1} to include N by using limits. $\text{Unique}(\infty) := \lim_{p \to \infty} \text{Unique}(p) = N$; $\text{Unique}^{-1}(N) := \lim_{u \to N} \text{Unique}^{-1}(u) = \infty$.

It is straightforward to compute the value of $\text{Unique}^{-1}(u)$ either by solving $\text{Unique}(p) = u$ for p numerically or by approximating Unique and Unique^{-1} with piecewise linear functions.

4.2.4 Merging Tables

Compaction in MSLS takes multiple tables and creates a new set of tables that contain no duplicate keys. Nontrivial cases involve tables with overlapping key ranges. For such cases, we can estimate the size of merged tables using a combination of Unique and Unique^{-1}:

Theorem 4.3. $\text{Merge}(u, v) = \text{Unique}(\text{Unique}^{-1}(u) + \text{Unique}^{-1}(v))$ for $0 \leq u, v \leq N$.

Proof. Let p and q be the expected numbers of insert requests that would produce tables of size u and v, respectively. The merged table is expected to contain all $k \in K$ except those missing in both request sequences. Therefore, $\text{Merge}(u, v) = N - \sum_{k \in K}(1 - f_X(k))^p (1 - f_X(k))^q = \text{Unique}(p + q)$. Because $p = \text{Unique}^{-1}(u)$ and $q = \text{Unique}^{-1}(v)$, $\text{Merge}(u, v) = \text{Unique}(\text{Unique}^{-1}(u) + \text{Unique}^{-1}(v))$. □

In the worst-case analysis, merging tables of size u and v results in a new table of size $u + v$, which assumes that the input tables contain no duplicate keys. Such an assumption creates a larger error as u and v approach N and as the key popularity has more skew. For example, with 100 million (10^8) total unique keys and Zipf skewness of 0.99, $\text{Merge}(10^7, 9 \times 10^7) \approx 9.03 \times 10^7$ keys, whereas the worst-case analysis expects 10^8 keys.

Finally, Unique is an isomorphism as shown in Figure 4.5. Unique maps the length of a sequence of requests to the number of unique keys in it, and Unique^{-1} forms the opposite direction. The addition of request counts corresponds to applying Merge to unique key counts; the addition calculates the length of concatenated request sequences, and Merge obtains the number

```
// @param  L     maximum level
// @param  wal   write-ahead log file size
// @param  c0    level-0 SSTable count
// @param  size  level sizes
// @return       write amplification
function estimateWA_LevelDB(L, wal, c0, size[]) {
  local l, WA, interval[], write[];

  // mem -> log
  WA = 1;

  // mem -> level-0
  WA += unique(wal) / wal;

  // level-0 -> level-1
  interval[0] = wal * c0;
  write[1] = merge(unique(interval[0]), size[1]);
  WA += write[1] / interval[0];

  // level-l -> level-(l+1)
  for (l = 1; l < L; l++) {
    interval[l] = interval[l-1] + dinterval(size, l);
    write[l+1] = merge(unique(interval[l]), size[l+1]) + unique(interval[l]);
    WA += write[l+1] / interval[l];
  }

  return WA;
}
```

Algorithm 3: Pseudocode of a model that estimates WA of LevelDB.

of unique keys in the merged table. Translating the number of requests to the number of unique keys and vice versa make it easy to build an MSLS model, as presented in the next section.

4.3 Modeling LevelDB

This section applies our analytic primitives to model a practical MSLS design, LevelDB. We explain how the dynamics of LevelDB components can be incorporated into the LevelDB model. We compare the analytic estimate with the actual performance measured with a LevelDB simulator and the original implementation.

We assume that the dictionary-based compression [37, 64, 67] is not used in logs and SSTables. Using compression can reduce the write amplification (WA) by a certain factor; its effectiveness depends significantly on how compressible the stored data is.

Algorithm 3 summarizes the WA estimation for LevelDB. **unique**(), **uniqueinv**(), and **merge**() calculate Unique, Unique^{-1}, and Merge as defined in Section 4.2. **dinterval**() calculates DInterval, which we define in this section.

4.3.1 Logging

The write-ahead logging (WAL) of LevelDB writes roughly the same amount as the data volume of inserts. We do not need to take into account the key redundancy because logging does not perform redundancy removal. As a consequence, logging contributes to 1 unit of WA (line #10). Using an advanced WAL scheme [36] can lower the logging cost below 1 unit.

4.3.2 Constructing Level-0 SSTables

LevelDB stores the contents of the memtable as a new SSTable in level-0 whenever the current log size reaches a threshold *wal*, which is 4 MiB in LevelDB by default.[5] Because an SSTable contains no redundant keys, we use Unique to compute the expected size of the SSTable corresponding to the accumulated requests; for every *wal* requests, LevelDB creates an SSTable of Unique(wal), which adds Unique(wal)/wal to WA (line #13).

4.3.3 Compaction

LevelDB compacts one or more SSTables in a level into the next level when one or more of the following conditions are satisfied: (1) when level-0 has at least $c0$ SSTables; (2) when the aggregate size of SSTables in a level-l ($1 \leq l \leq L$) reaches Size(l) bytes; or (3) after an SSTable has observed a certain number of seeks caused by query processing. The original LevelDB defines $c0$ to be 4 SSTables[6] and Size(l) to be 10^l MiB. A particular level to compact is chosen based on how high the ratio of the current SSTable count or level size to the triggering condition is, and this can be approximated as prioritizing levels in their order from 0 to L in the model. The seek trigger depends on the distribution of queries as well as of insert requests, which is beyond the scope of this chapter.

We examine two quantities to estimate the amortized compaction cost: (1) a certain interval (insert requests), denoted as Interval(l), and (2) the expected amount of data written to level-$(l+1)$ during that interval, which we denote as Write$(l+1)$. The contribution to WA by the compaction from level-l to level-$(l+1)$ is given by Write$(l+1)$/ Interval(l) by the definition of WA (line #18, line #24).

Compacting Level-0 SSTables

LevelDB picks a level-0 SSTable and other level-0 SSTables that overlap with the first SSTable picked. It chooses overlapping level-1 SSTables as the other compaction input, and it can possibly choose more level-0 SSTables as long as the number of overlapping level-1 SSTables remains unchanged. Because level-0 contains overlapping SSTables with a wide key range, it is common to have multiple level-0 SSTables selected for a single compaction event; to build a concise model,

[5] We use the byte size and the item count interchangeably based on the assumption of fixed item sizes, as described in Section 4.2.2.

[6] LevelDB begins compaction with 4 level-0 SSTables, and new insert requests stall if the compaction of level-0 is not fast enough that the level-0 SSTable count reaches 12.

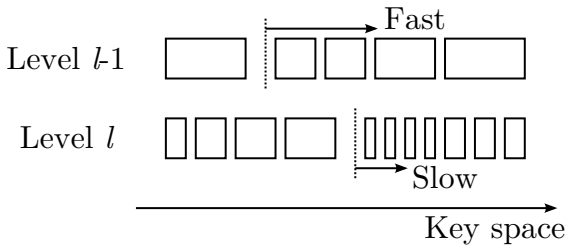


Figure 4.6: Non-uniformity of the key density caused by the different compaction speed of two adjacent levels in the key space. Each rectangle represents an SSTable. Vertical dotted lines indicate the last compacted key; the rectangles right next to the vertical lines will be chosen for compaction next time.

we assume that all level-0 SSTables are chosen for compaction whenever the compaction trigger for level-0 is met.

Let Interval(0) be the interval of creating $c0$ SSTables, where Interval$(0) = wal \cdot c0$ (line #16). Compaction performed for that duration merges the SSTables created from Interval(0) requests, which contain Unique(Interval(0)) unique keys, into level-1 with Size(1) unique keys. Therefore, Write(1) = Merge(Unique(Interval(0)), Size(1)) (line #17).

Compacting Non-Level-0 SSTables

While the compaction from level-l to level-$(l+1)$ $(1 \leq l < L)$ follows similar rules as level-0 does, it is more complicated because of how LevelDB chooses the next SSTable to compact. LevelDB remembers LastKey(l), the last key of the SSTable used in the last compaction for level-l and picks the first SSTable whose smallest key succeeds LastKey(l); if there exists no such SSTable, LevelDB picks the SSTable with the smallest key in the level. This compaction strategy chooses SSTables in a circular way in the key space for each level.

Non-uniformity arises because of this round-robin compaction strategy. Compaction removes items from a given level, but its effect is localized in the key space of that level. Compaction from a lower level into this given level, however, tends to push items across the entire key space of the receiving level because the lower level makes faster progress compacting the entire key space because of the lower level's smaller size. Figure 4.6 depicts this process. As a result, the part of the key space that was recently compacted has a lower chance of having items (i.e., low density), whereas the other part of the key space that has not been compacted for a long period of time has a higher chance of having items (i.e., high density). Because of the constraint on the maximum SSTable size, the low density area has SSTables covering a wide key range, and the high density area has SSTables with a narrow key range.

This non-uniformity makes compaction less costly. Compaction occurs for an SSTable at the dense part of the key space. The narrow key range of the dense SSTable means a relatively small number of overlapping SSTables in the next level. Therefore, the compaction of the SSTable results in less data written to the next level.

Some LevelDB variants [75] explicitly pick an SSTable that maximizes the ratio of the size of that SSTable to the size of all overlapping SSTables in the next level, in hope of making the compaction cost smaller. In fact, due to the non-uniformity, LevelDB already *implicitly* realizes a

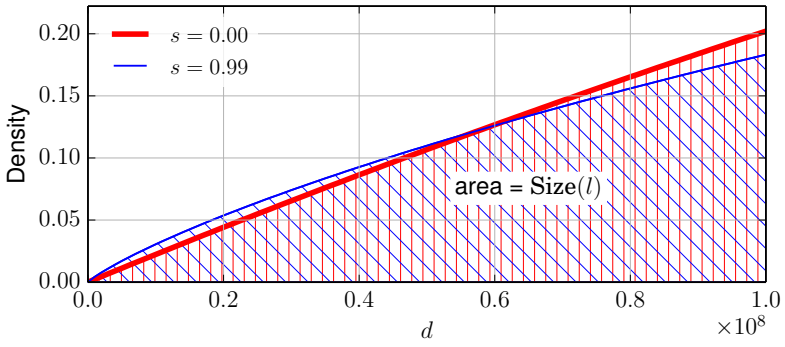


Figure 4.7: Density as a function of the distance (d) from the last compacted key in the key space, with varying Zipf skewness (s): $y = \text{Density}(l,d)$, where $l = 4$ is the second-to-last level. $\text{Size}(l) = 10 \cdot 2^{20}$. Using 100 million unique keys, 1 kB item size.

similar compaction strategy. Our simulation results (not shown) indicate that the explicit SSTable selection brings a marginal performance gain over LevelDB's circular SSTable selection.

To quantify the effect of the non-uniformity to compaction, we model the density distribution of a level. Let $\text{DInterval}(l)$ be the expected interval between compaction of the same key in level-l. This is also the interval to merge the level-l data into the entire key space of level-$(l+1)$. We use d to indicate the unidirectional distance from the most recently compacted key $\text{LastKey}(l)$ to a key in the key space, where $0 \le d < N$. $d = 0$ represents the key just compacted, and $d = N-1$ is the key that will be compacted next time. Let $\text{Density}(l,d)$ be the probability of having an item for the key with distance d in level-l. Because we assume no spatial key locality, we can formulate Density by approximating $\text{LastKey}(l)$ to have a uniform distribution:

Theorem 4.4. Assuming $\text{P}(\text{LastKey}(l) = k) = 1/N$ for $1 \le l < L$, $k \in K$, then $\text{Density}(l,d) = \text{Unique}(\text{DInterval}(l) \cdot d/N)/N$ for $1 \le l < L$, $0 \le d < N$.

Proof. Suppose $\text{LastKey}(l) = k \in K$. Let k' be $(k-d+N) \bmod N$. Let r be $\text{DInterval}(l) \cdot d/N$. There are r requests since the last compaction of k'. Level-l has k' if any of r requests contains k', whose probability is $1-(1-f_X(k'))^r$.

By considering all possible k and thus all possible k', $\text{Density}(l,d) = \sum_{k \in K} (1/N)(1-(1-f_X(k))^r) = \text{Unique}(\text{DInterval}(l) \cdot d/N)/N$. □

We also use a general property of the density:

Lemma 4.5. $\sum_{d=0}^{N-1} \text{Density}(l,d) = \text{Size}(l)$ for $1 \le l < L$.

Proof. The sum over the density equals to the expected unique key count, which is the number of keys level-l maintains, i.e., $\text{Size}(l)$. □

Figure 4.7 depicts the relationship between the density and level size. The workload skew affects the density distribution, while the area between the line and x-axis remains constant, i.e., $\text{Size}(l)$. The value of $\text{DInterval}(l)$ can be obtained by solving it numerically using Theorem 4.4 and Lemma 4.5.

We see that $\text{DInterval}(l)$ is typically larger than $\text{Unique}^{-1}(\text{Size}(l))$ that represents the expected interval of compacting the same key *without* non-uniformity. For example, with $\text{Size}(l) = 10$ Mi,

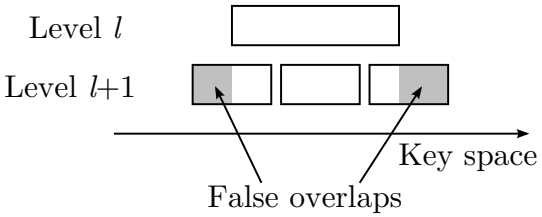


Figure 4.8: False overlaps that occur during the LevelDB compaction. Each rectangle indicates an SSTable; its width indicates the table's key range, not the byte size.

$N = 100$ M (10^8), and a uniform key popularity distribution, DInterval(l) is at least twice as large as Unique^{-1}(Size(l)): 2.26×10^7 vs. 1.11×10^7. This confirms that non-uniformity does slow down the progression of LastKey(l), improving the efficiency of compaction.

Interval(l), the actual interval we use to calculate the amortized WA, is cumulative and increases by DInterval, i.e., Interval(l) = Interval($l-1$) + DInterval(l) (line #22). Because compacting lower levels is favored over compacting upper levels, an upper level may contain more data than its compaction trigger as an *overflow* from lower levels. We use a simple approximation to capture this behavior by adding the cumulative term Interval($l-1$).

False overlaps are another effect caused by the incremental compaction using SSTables in LevelDB. Unlike non-uniformity, they increase the compaction cost slightly. For an SSTable being compacted, overlapping SSTables in the next level may contain items that lie outside the key range of the SSTable being compacted, as illustrated in Figure 4.8. Even though the LevelDB implementation attempts to reduce such false overlaps by choosing more SSTables in the lower level without creating new overlapping SSTables in the next level, false overlaps may add extra data writes whose size is close to that of the SSTables being compacted, i.e., Unique(Interval(l)) for Interval(l). Note that these extra data writes caused by false overlaps are more significant when Unique for the interval is large, i.e., under low skew, and they diminish as Unique becomes small, i.e., under high skew.

Several proposals [4, 127] strive to further reduce false overlaps by reusing a portion of input SSTables, essentially trading storage space and query performance for faster inserts. Such techniques can reduce WA by up to 1 per level, and even more if they address other types of false overlaps; the final cost savings, however, largely depend on the workload skew and the degree of the reuse.

By considering all of these factors, we can calculate the expected size of the written data. During Interval(l), level-l accepts Unique(Interval(l)) unique keys from the lower levels, which are merged into the next level containing Size($l+1$) unique keys. False overlaps add extra writes roughly as much as the compacted level-l data. Thus, Write($l+1$) = Merge(Unique(Interval(l)), Size($l+1$)) + Unique(Interval(l)) (line #23).

4.3.4 Sensitivity to the Workload Skew

To examine how our LevelDB model reacts to the workload skew, we compare our WA estimates with the asymptotic analysis results. For asymptotic analysis results, we use the worst-case scenario with zero redundancy, where merging two SSTables yields an SSTable whose size is

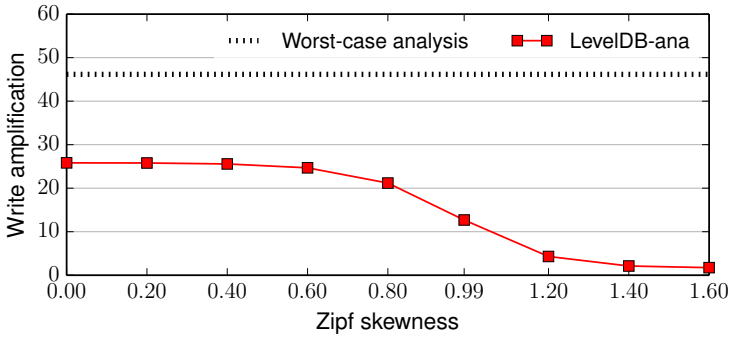


Figure 4.9: Effects of the workload skew on WA. Using 100 million unique keys, 1 kB item size.

exactly the same as the sum of the input SSTable sizes. In other words, compacting levels of size u and v results in $u+v$ items in the worst case.

Figure 4.9 plots the estimated WA for different Zipf skew parameters. Because our analytic model ("LevelDB-ana") considers the key popularity distribution of the workload in estimating WA, it clearly shows how WA decreases as LevelDB handles more skewed workloads; in contrast, the asymptotic analysis ("Worst-case analysis") gives the same result regardless of the skew.

4.3.5 Comparisons with the Asymptotic Analysis, Simulation, and Experiment

We compare analytic estimates of WA given by our LevelDB model with the estimates given by the asymptotic analysis, and the measured cost by running experiments on a LevelDB simulator and the original implementation.

We built a LevelDB simulator that follows the LevelDB design specification [66] and uses system parameters extracted from the LevelDB source code. This simulator does not intend to capture every detail of LevelDB implementation behaviors; instead, it realizes the high-level design components as explained in the LevelDB design document. The major differences are (1) our simulator runs in memory; (2) it performs compaction synchronously without concurrent request processing; and (3) it does not implement several opportunistic optimizations: (a) reducing false overlaps by choosing more SSTables in the lower level, (b) bypassing level-0 and level-1 for a newly created SSTable from the memtable if there are no overlapping SSTables in these levels, and (c) dynamically allowing more than 4 level-0 SSTables under high load.

For the measurement with the LevelDB implementation, we instrumented the LevelDB code (version 1.18) to report the number of bytes written to disk via system calls. We use an item size that is 18 bytes smaller than we do in the analysis and simulation, to compensate for the increased data writes due to LevelDB's own storage space overhead. For fast experiments, we disable `fsync` and checksumming, which showed no effects on WA in our experiments. We also avoid inserting items at an excessive rate that can overload level-0 with many SSTables and cause a high lookup cost.

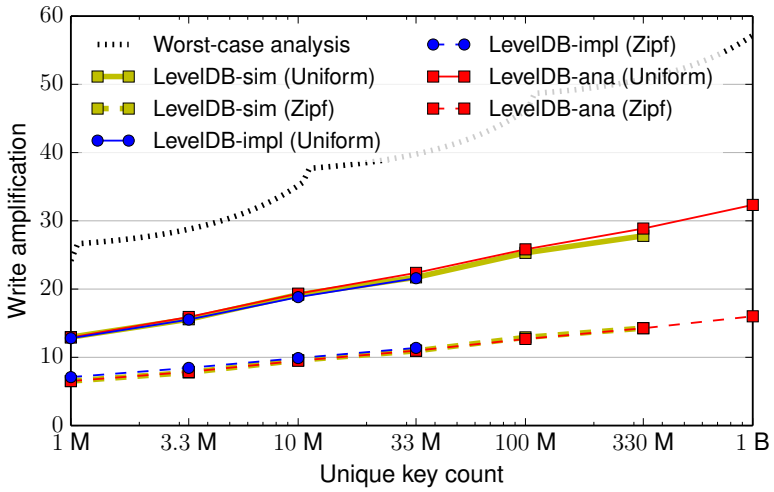


Figure 4.10: Comparison of WA between the estimation from our LevelDB model, the asymptotic analysis, LevelDB simulation, and implementation results, with a varying number of total unique keys. Using 1 kB item size. Simulation and implementation results with a large number of unique keys are unavailable due to excessive runtime.

Both LevelDB simulator and implementation use a YCSB [38]-like workload generator written in C++. Each experiment initializes the system by inserting all keys once and then measures the average WA of executing random insert requests whose count is 3 times the total unique key count.

Figure 4.10 shows WA estimation and measurement with a varying number of total unique keys. Due to excessive experiment time, the graph excludes some data points for simulation ("LevelDB-sim") and implementation ("LevelDB-impl") with a large number of unique keys. The graph shows that our LevelDB model successfully estimates WA that agrees almost perfectly with the simulation and implementation results. The most significant difference occurs at 330 M unique keys with the uniform popularity distribution, where the estimated WA is only 3.8% higher than the measured WA. The standard asymptotic analysis, however, significantly overestimates WA by 1.8–3.6X compared to the actual cost, which highlights the accuracy of our LevelDB model.

Figure 4.11 compares results with different *write buffer* size (i.e., the memtable size), which determines how much data in memory LevelDB accumulates to create a level-0 SSTable (and also affects how long crash recovery may take). In our LevelDB model, *wal* reflects the write buffer size. We use write buffer sizes between LevelDB's default size of 4 MiB and 10% of the last level size. The result indicates that our model estimates WA with good accuracy, but the error increases as the write buffer size increases for uniform key popularity distributions. We suspect that the error comes from the approximation in the model to take into account temporal overflows of levels beyond their maximum size; the error diminishes when level sizes are set to be at least as large as the write buffer size. In fact, avoiding too small level-1 and later levels has been suggested by RocksDB developers [55], and our optimization performed in Section 4.6 typically results in

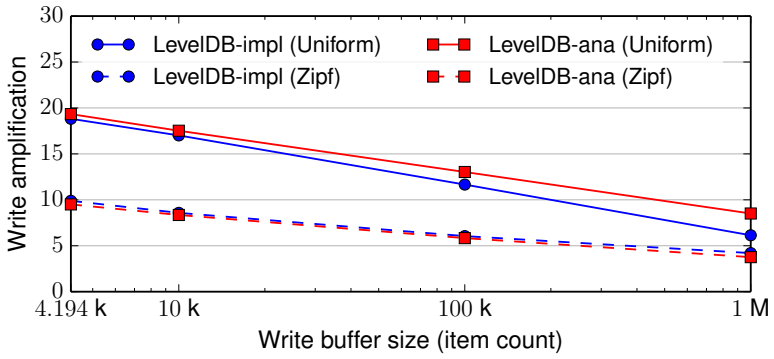


Figure 4.11: Comparison of WA between the estimation from LevelDB simulation and implementation results, with varying write buffer sizes. Using 10 million unique keys, 1 kB item size.

moderately large sizes for lower levels under uniform distributions, which makes this type of error insignificant for practical system parameter choices.

4.4 Modeling COLA and SAMT

To illustrate the power of our primitives to model MSLS systems, we present two additional examples, COLA and SAMT, in Algorithm 4. Both models assume that the system uses write-ahead log files whose count is capped by the growth factor r. In COLA, line #15 calculates the amount of writes for a level that has already accepted j merges ($0 \leq j < r-1$). Compaction of the second-to-last level is treated specially because the last level must be large enough to hold all unique keys and has no subsequent level (line #19). The SAMT model is simpler because it defers merging the data in the same level.

4.5 Modeling SILT

We revisit the multi-store design of SILT and build an accurate model of SILT. The estimated write amplification by this model for the uniform workload is essentially the same as the estimate given in Section 2.4. However, this model provides an estimate for skewed workloads instead of giving the worst-case estimate. For a workload with Zipf skewness 0.99, this model estimates WA to be 3.0, whereas the worst-case analysis always gives WA of 5.4 regardless of the skew, when using the same workload size and system parameters as we do in Section 2.4 ($hs = 2^{17}, occ = 0.93, hc = 31 * 2$).

SILT can achieve even lower memory overheads under skewed workloads. By reducing the number of HashStores (hc) from 62 to 25, SILT can maintain the same level of WA (5.4) for the skewed workload whose skewness is 0.99. This parameter selection stores more data in the memory-efficient SortedStore and lowers the memory use by 26%, which lowers the memory overhead to 0.56 bytes per item.

```
// @param  L     maximum level
// @param  wal   write-ahead log file size
// @param  r     growth factor
// @return       write amplification
function estimateWA_COLA(L, wal, r) {
  local l, j, WA, interval[], write[];
  // mem -> log
  WA = 1;
  // mem -> level-1; level-l -> level-(l+1)
  interval[0] = wal;
  for (l = 0; l < L - 1; l++) {
    interval[l + 1] = interval[l] * r;
    write[l + 1] = 0;
    for (j = 0; j < r - 1; j++)
      write[l + 1] += merge(unique(interval[l]), unique(interval[l] * j));
    WA += write[l + 1] / interval[l + 1];
  }
  // level-(L-1) -> level-L
  WA += unique(∞) / interval[L - 1];
  return WA;
}

function estimateWA_SAMT(L, wal, r) {
  local l, WA, interval[], write[];
  // mem -> log
  WA = 1;
  // mem -> level-1; level-l -> level-(l+1)
  interval[0] = wal;
  for (l = 0; l < L - 1; l++) {
    interval[l + 1] = interval[l] * r;
    write[l + 1] = r * unique(interval[l]);
    WA += write[l + 1] / interval[l + 1];
  }
  // level-(L-1) -> level-L
  WA += unique(∞) / interval[L - 1];
  return WA;
}
```

Algorithm 4: Pseudocode of models that estimate WA of COLA and SAMT.

4.6 Optimizing System Parameters

Compared to simulators and implementations, an analytic model offers fast estimation of cost metrics for a given set of system parameters. To demonstrate fast evaluation of the analytic model, we use an example of optimizing LevelDB system parameters to reduce WA using our LevelDB model.

Note that the same optimization effort could be made with the full LevelDB implementation by substituting our LevelDB model with the implementation and a synthetic workload generator. However, it would take prohibitively long to explore the large parameter space, as examined in Section 4.6.4.

```
// @param  hs    LogStore and HashStore's table size
// @param  occ   table occupancy
// @param  hc    maximum HashStore count
// @return       write amplification
function estimateWA_SILT(hs, occ, hc) {
  local WA, interval;

  // conversion interval
  interval = uniqueinv(hs * occ);

  // LogStore
  WA = 1;

  // LogStore -> HashStore
  WA += hs / interval;

  // HashStore -> SortedStore
  WA += unique(∞) / (interval * hc);
  return WA;
}
```

Algorithm 5: Pseudocode of models that estimate WA of SILT.

4.6.1 Parameter Set to Optimize

An important set of system parameters in LevelDB are the level sizes, Size(l). They determine when LevelDB should initiate compaction for standard levels and affect the overall compaction cost of the system. The original LevelDB design uses a geometric progression of Size(l) $= 10^l$ MiB. Interesting questions are (1) what level sizes different workloads favor; and (2) whether the geometric progression of level sizes is the optimal for all workloads.

Using different level sizes does not necessarily trade query performance or memory use. The log size, level-0 SSTable count, and total level count—the main determinants of query performance—are all unaffected by this system parameter.

4.6.2 Optimizer

We implemented a system parameter optimizer based on our analytic model. The objective function to minimize is the estimated WA. Input variables are Size(l), excluding Size(L), which will be equal to the total unique key count. After finishing the optimization, we use the new level sizes to obtain new WA estimates and measurement results on our analytic model and simulator. We also force the LevelDB implementation to use the new level sizes and measure WA. Our optimizer is written in Julia [20] and uses Ipopt [140] for nonlinear optimization. To speed up Unique, we use a compressed key popularity distribution which groups keys with similar probabilities and stores their average probability.[7]

[7]For robustness, we optimize using both the primal and a dual form of the LevelDB model presented in Section 4.3. The primal optimizes over Size(l) and the dual optimizes over Unique^{-1}(Size(l)). We pick the result of whichever model produces the smaller WA.

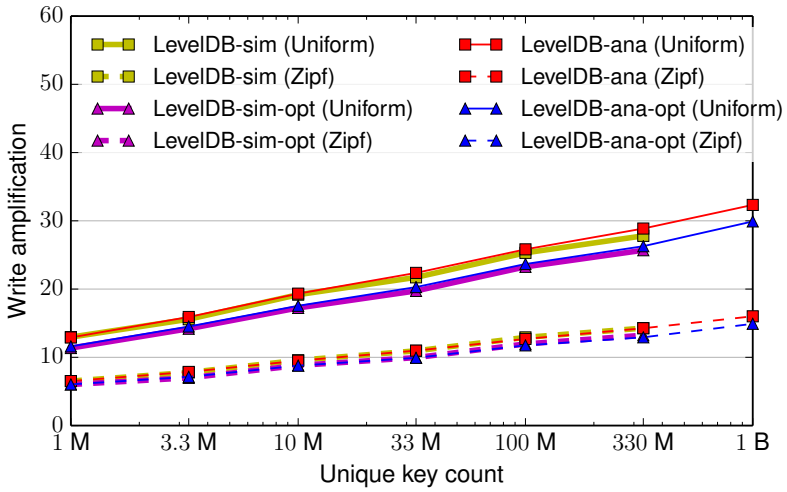


Figure 4.12: Improved WA using optimized level sizes on our analytic model and simulator for LevelDB.

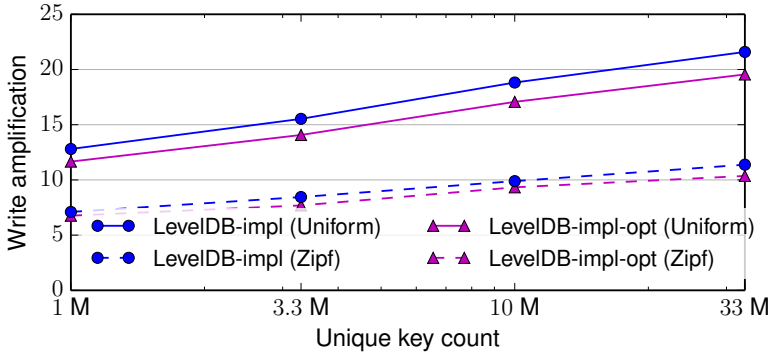


Figure 4.13: Improved WA using optimized level sizes on the LevelDB implementation.

4.6.3 Optimization Results

Our level size optimization successfully reduces the insert cost of LevelDB. Figures 4.12 and 4.13 plot WA with optimized level sizes. Both graphs show that the optimization ("LevelDB-ana-opt," "LevelDB-sim-opt," and "LevelDB-impl-opt") improves WA by up to 9.4%. The analytic estimates and simulation results agree with each other as before, and the LevelDB implementation exhibits lower WA across all unique key counts.

The optimization is effective because level sizes differ by the workload skew, as shown in Figure 4.14. Having larger low levels is beneficial for relatively low skew. On the other hand, high skew favors smaller low levels and level sizes that grow faster than the standard geometric progression. This result suggests that it is suboptimal to use fixed level sizes for different workloads and that using a geometric progression of level sizes is not always the best design to minimize WA.

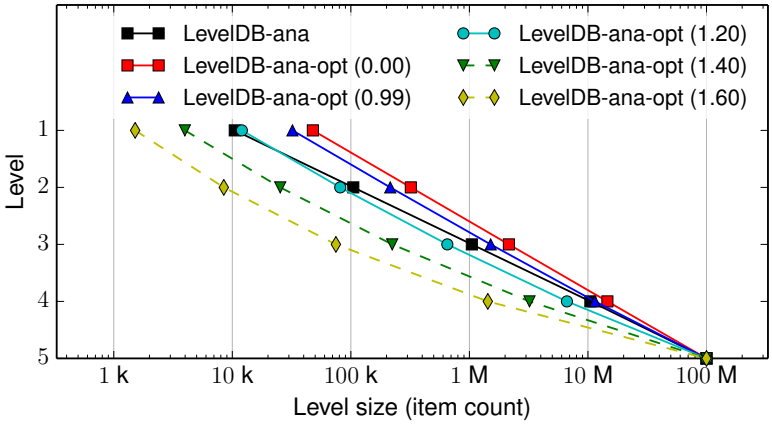


Figure 4.14: Original and optimized level sizes with varying Zipf skewness. Using 100 million unique keys, 1 kB item size.

Source	Analysis		Simulation	
	No opt	Opt	No opt	Opt
mem→log	1.00	1.00	1.00	1.00
mem→level-0	1.00	1.00	1.00	1.00
level-0→1	1.62	3.85	1.60	3.75
level-1→2	4.77	4.85	4.38	4.49
level-2→3	6.22	4.82	6.04	4.66
level-3→4	6.32	4.65	6.09	4.55
level-4→5	4.89	3.50	5.20	3.81
Total	25.82	23.67	25.31	23.26

Table 4.1: Breakdown of WA sources on the analysis and simulation without and with the level size optimization. Using 100 million unique keys, 1 kB item size, and a uniform key popularity distribution.

Table 4.1 further examines how the optimization affects per-level insert costs, using the LevelDB model and simulation. Per-level WA tends to be more variable using the original level sizes, while the optimization makes them relatively even across levels except the last level. This result suggests that it may be worth performing a runtime optimization that dynamically adjusts level sizes to achieve the lower overall WA by reducing the variance of the per-level WA.

By lowering WA, the system can use fewer levels to achieve faster lookup speed without significant impact on insert costs. Figure 4.15 reveals how much extra room for query processing the optimization can create. This analysis changes the level count by altering the growth factor of LevelDB, i.e., using a higher growth factor for a lower level count. The result shows that the optimization is particularly effective with a fewer number of levels, and it can save almost a whole level's worth of WA compared to using a fixed growth factor. For example, with the optimized level sizes, a system can use 3 levels instead of 4 levels without incurring excessively high insert costs.

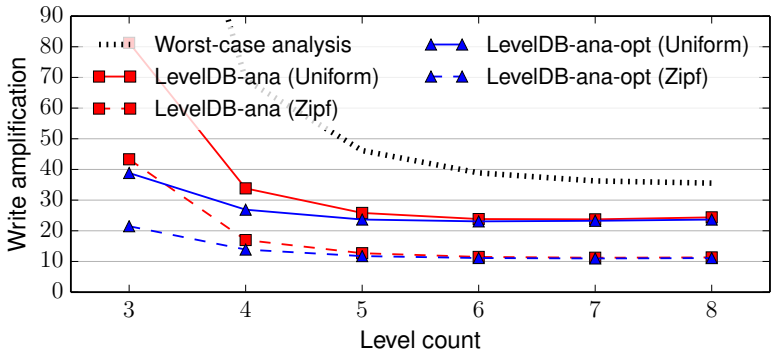


Figure 4.15: WA using varying numbers of levels. The level count excludes level-0. Using 100 million unique keys, 1 kB item size.

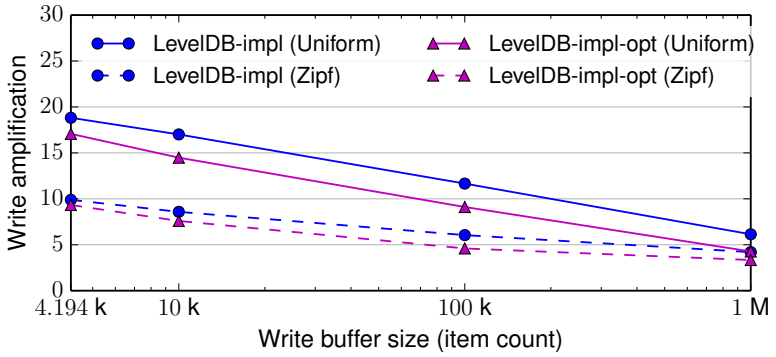


Figure 4.16: Improved WA using optimized level sizes on the LevelDB implementation, with a large write buffer. Using 10 million unique keys, 1 kB item size.

A LevelDB system with large memory can further benefit from our level size optimization. Figure 4.16 shows the result of applying the optimization to the LevelDB implementation, with a large write buffer. The improvement becomes more significant as the write buffer size increases, reaching 30.5% of WA reduction at the buffer size of 1 million items.

4.6.4 Optimizer Performance

The level size optimization requires little time due to the fast evaluation of our analytic model. For 100 million unique keys with a uniform key popularity distribution, the entire optimization took 3.17 seconds, evaluating 12,254 different parameter sets (3,866 evaluations per second) on a desktop-class machine. For the same-sized workload, but with Zipf skewness of 0.99, the optimization time increased to 118 seconds, which is far more than the uniform case, but is less than 2 minutes; for this optimization, the model was evaluated 12,645 times before convergence (107 evaluations per second).

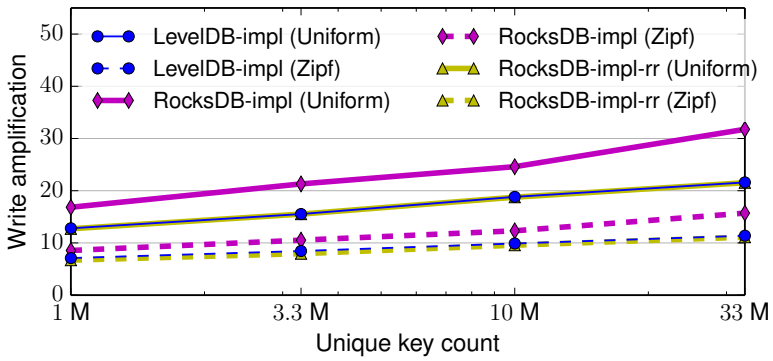


Figure 4.17: Comparison of WA between LevelDB, RocksDB, and a modified RocksDB with a LevelDB-like compaction strategy.

Evaluating this many system parameters using a full implementation—or even simulation—is prohibitively expensive. On server-class hardware, our in-memory LevelDB simulator takes about 12 minutes to measure WA for a *single* set of system parameters with 100 million unique keys. The full LevelDB implementation takes 35 minutes (without `fsync`) to 173 minutes (with `fsync`), for a smaller dataset with 10 million unique keys.

4.7 Improving RocksDB

In this section, we turn our attention to RocksDB [54], a well-known variant of LevelDB. RocksDB offers improved capabilities and multithreaded performance, and provides an extensive set of system configurations to temporarily accelerate bulk loading by sacrificing query performance or relaxing durability guarantees [52, 55]; nevertheless, there have been few studies of how RocksDB's *design* affects its performance. We use RocksDB version 3.9.1 and apply the same set of instrumentation, configuration, and workload generation as we do to LevelDB.

RocksDB supports "Level Style Compaction" that is similar to LevelDB's data layout, but differs in how it picks the next SSTable to compact. RocksDB picks the largest SSTable in a level for compaction, rather than keeping LevelDB's round-robin SSTable selection. However, we learned in Section 4.3 that LevelDB's compaction strategy is effective in reducing WA because it tends to pick SSTables that incur less WA.

To compare the compaction strategies used by LevelDB and RocksDB, we measure the insert cost of both systems in Figure 4.17. Unfortunately, the current RocksDB design produces higher WA ("RocksDB-impl") than LevelDB does ("LevelDB-impl"). In theory, the RocksDB approach may help multithreaded compaction because large tables may be spread over the entire key space so that they facilitate parallel compaction; this effect, however, was not evident in our experiments using multiple threads. The high insert cost of RocksDB is entirely caused by RocksDB's compaction strategy; implementing LevelDB's SSTable selection in RocksDB ("RocksDB-impl-rr") reduces RocksDB's WA by up to 32.2%, making it comparable to LevelDB's WA. This result confirms that LevelDB's strategy is good at reducing WA as our analytic model predicts.

Besides the intra-level table selection, RocksDB exhibits starvation in choosing a level to compact. It always favors compacting level-0 into level-1 if the count of level-0 SSTables reaches a certain threshold. By doing so, RocksDB tries to avoid high insert latencies that can occur when it hits a hard limit of the level-0 SSTable count. However, this prioritization accumulates a large number of level-1 SSTables without performing sufficient compaction in level-1, increasing the cost of compaction from level-0 to level-1. In our experiments, this (mis)prioritization results in an overall WA of over 70 for 10 million items; a WA this high defeats the purpose of the prioritization by prolonging the contention with the inefficient compaction. We mitigated this starvation by not favoring level-0; this fixed version serves as the baseline in all of our experiments above.

We have not found a scenario where RocksDB's current strategy excels, though some combinations of workloads and situations may favor it. LevelDB and RocksDB developers may or may have not intended any of the effects on the overall WA when they designed their systems. Either way, our analytic model provides quantitative evidence that LevelDB's table selection will perform well under a wide range of workloads despite being the "conventional" solution.

4.8 Discussion

Analyzing an MSLS design with an accurate model can provide useful insights on how one should design a new MSLS to exploit opportunities provided by workloads. For example, our analytic model reveals that LevelDB's byte size-based compaction trigger makes compaction much less costly under skew; such a design choice should be suitable for many real-world workloads with skew [8].

A design process that is supplemented with accurate analysis can help avoid false conclusions about a design's performance. LevelDB's per-level WA is far less (only up to 4–6) than assumed in the worst case (11–12 for the growth factor of 10), even for uniform workloads. Our analytical model justifies LevelDB's high growth factor, which turns out not to be as harmful for insert performance as the standard asymptotic analysis implies.

Our focus in this chapter is on estimating WA because it is more difficult to mitigate the effects of WA than RA. We believe, however, that our approach is also useful to understand other performance metrics. For instance, estimating the number of unique items of a certain level in the system can help one calculate read amplification by estimating how many levels on average the system needs to access to find items.

Our analytic primitives and modeling method, however, are not without limitations. The assumptions such as no spatial locality in requested keys may not hold if there are dependent keys that share the same prefix. A similar limitation applies when handling datasets with highly varying item sizes because our primitives use item count to calculate quantities instead of the item byte size. Finally, our primitives also do not take into account time-varying workload characteristics (e.g., flash crowds) or special item types such as tombstones that represent item deletion. We leave extending our primitives to accommodate such cases as future work.

4.9 Related Work

Over the past decade, numerous studies have proposed new multi-stage log-structured (MSLS) designs and evaluated their performance. In almost every case, the authors present implementation-level performance [4, 11, 17, 54, 65, 75, 91, 125, 126, 127, 128, 138, 141, 144]. Some employ analytic metrics such as write amplification to explain the design rationale, facilitate design comparisons, and generalize experiment results [4, 11, 54, 75, 91, 125], and most of the others also use the concept of per-operation costs. However, they eventually rely on the experimental measurement because their analysis fails to offer sufficiently high accuracy to make any meaningful performance comparisons. LSM-tree [111], LHAM [105], COLA [17], bLSM [125], and B-tree variants [28, 81] provide extensive analysis on their design, but their analyses are limited to asymptotic complexities or assume the worst-case scenario.

Despite such a large number of MSLS design proposals, there is little active research to devise improved evaluation methods for these proposals to fill the gap between asymptotic analysis and experimental measurement. The sole existing effort is limited to a specific system design [100], but does not provide general-purpose primitives. We are unaware of any prior studies that successfully capture the workload skew and the dynamics of compaction to the degree that the estimates are close to simulation and implementation results, as we present in this chapter.

4.10 Summary

We present new analytic primitives for modeling multi-stage log-structured (MSLS) designs, which can quickly and accurately estimate their performance. We have presented a model for the popular LevelDB system, which estimates write amplification very close to experimentally determined actual costs; using this model, we were able to find more favorable system parameters that reduce the overall cost of writes. Based upon lessons learned from the model, we propose changes to RocksDB to lower its insert costs. We believe that our analytic primitives and modeling method are applicable to a wide range of MSLS designs and performance metrics. The insights derived from the models facilitate comparisons of MSLS designs and ultimately help develop new designs that better exploit workload characteristics to improve performance.

Chapter 5

MICA (Memory-store with Intelligent Concurrent Access)

In-memory key-value storage is a crucial building block for many systems, including popular social networking sites (e.g., Facebook) [108]. These storage systems must provide high performance when serving many small objects, whose total volume can grow to TBs and more [8].

While much prior work focuses on high performance for read-mostly workloads [58, 96, 99, 112], in-memory key-value storage today must also handle write-intensive workloads, e.g., to store frequently-changing objects [1, 8, 108]. Systems optimized only for reads often waste resources when faced with significant write traffic; their inefficiencies include lock contention [99], expensive updates to data structures [58, 96], and complex memory management [58, 99, 108].

In-memory key-value storage also requires low-overhead network communication between clients and servers. Key-value workloads often include a large number of small key-value items [8] that require key-value storage to handle short messages efficiently. Systems using standard socket I/O, optimized for bulk communication, incur high network stack overhead at both kernel- and user-level. Current systems attempt to batch requests at the client to amortize this overhead, but batching increases latency, and large batches are unrealistic in large cluster key-value stores because it is more difficult to accumulate multiple requests being sent to the same server from a single client [108].

MICA (Memory-store with Intelligent Concurrent Access) is an in-memory key-value store that achieves high throughput across a wide range of workloads. MICA can provide either store semantics (no existing items can be removed without an explicit client request) or cache semantics (existing items may be removed to reclaim space for new items). Under write-intensive workloads with a skewed key popularity, a single MICA node serves 70.4 million small key-value items per second (Mops), which is 10.8X faster than the next fastest system. For skewed, read-intensive workloads, MICA's 65.6 Mops is at least 4X faster than other systems even after modifying them to use our kernel bypass. MICA achieves 75.5–76.9 Mops under workloads with a uniform key popularity. MICA achieves this through the following techniques:

Fast and scalable parallel data access: MICA's data access is fast and scalable, using data partitioning and exploiting CPU parallelism within and between cores. Its EREW mode (Exclusive Read Exclusive Write) minimizes costly inter-core communication, and its CREW mode (Concurrent Read Exclusive Write) allows multiple cores to serve popular data. MICA's tech-

niques achieve consistently high throughput even under skewed workloads, one weakness of prior partitioned stores.

Network stack for efficient request processing: MICA interfaces with NICs directly, bypassing the kernel, and uses client software and server hardware to direct remote key-value requests to appropriate cores where the requests can be processed most efficiently. The network stack achieves zero-copy packet I/O and request processing.

New data structures for key-value storage: New memory allocation and indexing in MICA, optimized for store and cache separately, exploit properties of key-value workloads to accelerate write performance with simplified memory management.

5.1 System Goals

In this section, we first clarify the non-goals and then discuss the goals of MICA.

Non-Goals: We do not change the *cluster* architecture. It can still shard data and balance load across nodes, and perform replication and failure recovery.

We do not aim to handle large items that span multiple packets. Most key-value items will fit comfortably in a single packet [8]. Clients can store a large item in a traditional key-value system and put a pointer to that system in MICA. This only marginally increases total latency; one extra round-trip time for indirection is smaller than the transfer time of a large item sending multiple packets.

We do not strive for durability: All data is stored in DRAM. If needed, log-based mechanisms such as those from RAMCloud [112] would be needed to allow data to persist across power failures or reboots.

MICA instead strives to achieve the following goals:

High single-node throughput: Sites such as Facebook replicate some key-value nodes purely to handle load [108]. Faster nodes may reduce cost by requiring fewer of them overall, reducing the cost and overhead of replication and invalidation. High-speed nodes are also more able to handle load spikes and popularity hot spots. Importantly, using fewer nodes can also reduce job latency by reducing the number of servers touched by client requests. A single user request can create more than 500 key-value requests [108], and when these requests go to many nodes, the time until all replies arrive increases, delaying completion of the user request [41]. Having fewer nodes reduces fan-out, and thus, can improve job completion time.

Low end-to-end latency: The end-to-end latency of a remote key-value request greatly affects performance when a client must send back-to-back requests (e.g., when subsequent requests are dependent). The system should minimize both local key-value processing latency and the number of round-trips between the client and server.

Consistent performance across workloads: Real workloads often have a Zipf-distributed key popularity [8], and it is crucial to provide fast key-value operations regardless of skew. Recent uses of in-memory key-value storage also demand fast processing for write-intensive workloads [1, 108].

Handle small, variable-length key-value items: Most key-value items are small [8]. Thus, it is important to process requests for them efficiently. Ideally, key-value request processing over the network should be as fast as packet processing in software routers—40 to 80 Gbps [46, 72].

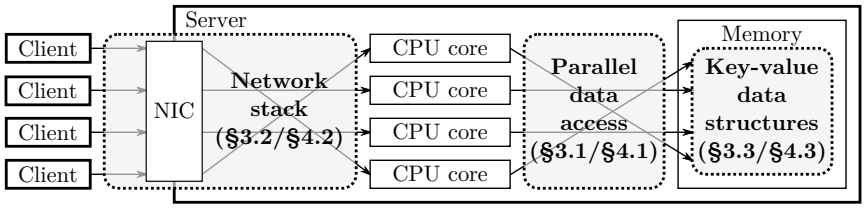


Figure 5.1: Components of in-memory key-value stores. MICA's key design choices in Section 5.2 and their details in Section 5.3.

Variable-length items require careful memory management to reduce fragmentation that can waste substantial space [8].

Key-value storage interface and semantics: The system must support standard single-key requests (e.g., `GET(key)`, `PUT(key,value)`, `DELETE(key)`) that are common in systems such as Memcached. In cache mode, the system performs automatic cache management that may evict stored items at its discretion (e.g., LRU); in store mode, the system must not remove any stored items without clients' permission while striving to achieve good memory utilization.

Commodity hardware: Using general-purpose hardware reduces the cost of development, equipment, and operation. Today's server hardware can provide high-speed I/O [46, 76], comparable to that of specialized hardware such as FPGAs and RDMA-enabled NICs.

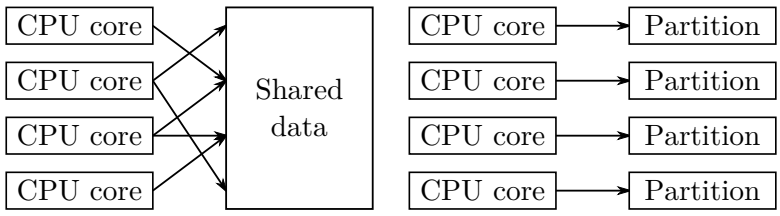
Although recent studies tried to achieve some of these goals, none of their solutions comprehensively address them. Some systems achieve high throughput by supporting only small fixed-length keys [101]. Many rely on client-based request batching [58, 96, 101, 108] to amortize high network I/O overhead, which is less effective in a large installation of key-value stores [56]; use specialized hardware, often with multiple client-server round-trips and/or no support for item eviction (e.g., FPGAs [23, 94], RDMA-enabled NICs [103]); or do not specifically address remote request processing [136]. Many focus on uniform and/or read-intensive workloads; several systems lack evaluation for skewed workloads [23, 101, 103], and some systems have lower throughput for write-intensive workloads than read-intensive workloads [96]. Several systems attempt to handle memory fragmentation explicitly [108], but there are scenarios where the system never reclaims fragmented free memory, as we describe in the next section. The fast packet processing achieved by software routers and low-overhead network stacks [46, 72, 73, 116, 122] set a bar for how fast a key-value system *might* operate on general-purpose hardware, but do not teach how their techniques apply to the higher-level processing of key-value requests.

5.2 Key Design Choices

Achieving our goals requires rethinking how we design *parallel data access*, the *network stack*, and *key-value data structures*. We make an unconventional choice for each; we discuss how we overcome its potential drawbacks to achieve our goals. Figure 5.1 depicts how these components fit together.

5.2.1 Parallel Data Access

Exploiting the parallelism of modern multi-core systems is crucial for high performance. The most common access models are concurrent access and exclusive access:



(a) Concurrent access (b) Exclusive access

Figure 5.2: Parallel data access models.

Concurrent access is used by most key-value systems [58, 96, 108]. As in Figure 5.2 (a), multiple CPU cores can access the shared data. The integrity of the data structure must be maintained using mutexes [108], optimistic locking [58, 96], or lock-free data structures [102].

Unfortunately, concurrent writes scale poorly: they incur frequent cache line transfer between cores, because only one core can hold the cache line of the same memory location for writing at the same time.

Exclusive access has been explored less often for key-value storage [19, 85, 101]. Only one core can access part of the data, as in Figure 5.2 (b). By partitioning the data ("sharding"), each core exclusively accesses its own partition in parallel without inter-core communication.

Prior work observed that partitioning *can* have the best throughput and scalability [96, 136], but cautions that it lowers performance when the load between partitions is imbalanced, as happens under skewed key popularity [58, 96, 136]. Furthermore, because each core can access only data within its own partition, *request direction* is needed to forward requests to the appropriate CPU core.

MICA's parallel data access: MICA partitions data and mainly uses exclusive access to the partitions. MICA exploits CPU caches and packet burst I/O to disproportionately speed more loaded partitions, nearly eliminating the penalty from skewed workloads. MICA can fall back to concurrent reads if the load is extremely skewed, but avoids concurrent writes, which are always slower than exclusive writes. Section 5.3.1 describes our data access models and partitioning scheme.

5.2.2 Network Stack

This section discusses how MICA avoids network stack overhead and directs packets to individual cores.

Network I/O

Network I/O is one of the most expensive processing steps for in-memory key-value storage. TCP processing alone may consume 70% of CPU time on a many-core optimized key-value store [101].

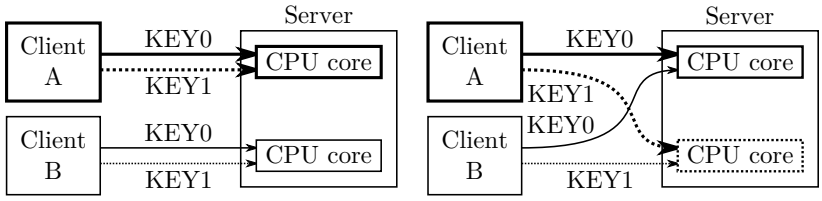
The **socket I/O** used by most in-memory key-value stores [58, 96, 101, 136] provides portability and ease of development. However, it underperforms in packets per second because it has high per-`read()` overhead. Many systems therefore often have clients include a batch of requests in a single larger packet to amortize I/O overhead.

Direct NIC access is common in software routers to achieve line-rate packet processing [46, 72]. This raw access to NIC hardware bypasses the kernel to minimize the packet I/O overhead. It delivers packets in bursts to efficiently use CPU cycles and the PCIe bus connecting NICs and CPUs. Direct access, however, precludes useful TCP features such as retransmission, flow control, and congestion control.

MICA's network I/O uses direct NIC access. By targeting only small key-value items, it needs fewer transport-layer features. Clients are responsible for retransmitting packets if needed. Section 5.3.2 describes such issues and our design in more detail.

Request Direction

Request direction delivers client requests to CPU cores for processing.[1] Modern NICs can deliver packets to specific cores for load balancing or core affinity using hardware-based packet classification and multi-queue support.



(a) Flow-level core affinity (b) Object-level core affinity

Figure 5.3: Request direction mechanisms.

Flow-level core affinity is available using two methods: Receive-Side Scaling (RSS) [46, 72] sends packets to cores based by hashing the packet header 5-tuple to identify which RX queue to target. Flow Director (FDir) [116] can more flexibly use different parts of the packet header plus a user-supplied table to map header values to RX queues. Efficient network stacks use affinity to reduce inter-core contention for TCP control blocks [73, 116].

Flow affinity reduces only *transport layer* contention, not *application-level* contention [73], because a single transport flow can contain requests for any objects (Figure 5.3 (a)). Even for datagrams, the benefit of flow affinity is small due to a lack of locality across datagrams [108].

[1] Because we target small key-value requests, we will use requests and packets interchangeably.

Object-level core affinity distributes requests to cores based upon the application's partitioning. For example, requests sharing the same key would all go to the core handling that key's partition (Figure 5.3 (b)).

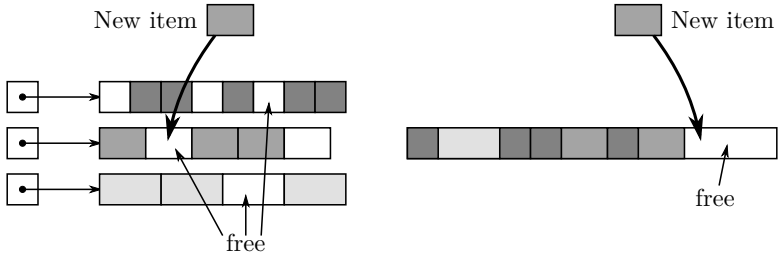
Systems using exclusive access require object-level core affinity, but commodity NIC hardware cannot directly parse and understand application-level semantics. Software request redirection (e.g., message passing [101]) incurs inter-core communication, which the exclusive access model is designed to avoid.

MICA's request direction uses Flow Director [77, 98]. Its *clients* then encode object-level affinity information in a way Flow Director can understand. Servers, in turn, inform clients about the object-to-partition mapping. Section 5.3.2 describes how this mechanism works.

5.2.3 Key-Value Data Structures

This section describes MICA's choice for two main data structures: allocators that manage memory space for storing key-value items and indexes to find items quickly.

Memory Allocator



(a) Dynamic object allocator (b) Append-only log structure

Figure 5.4: Memory allocators.

A **dynamic object allocator** is a common choice for storing variable-length key-value items (Figure 5.4 (a)). Systems such as Memcached typically use a slab approach: they divide object sizes into classes (e.g., 48-byte, 56-byte, ..., 1-MiB[2]) and maintain separate ("segregated") memory pools for these classes [58, 108]. Because the amount of space that each class uses typically varies over time, the systems use a global memory manager that allocates large memory blocks (e.g., 1 MiB) to the pools and dynamically rebalances allocations between classes.

The major challenge for dynamic allocation is the memory fragmentation caused when blocks are not fully filled. There may be no free blocks or free objects for some size classes while blocks from other classes are partly empty after deletions. Defragmentation packs objects of each object tightly to make free blocks, which involves expensive memory copy. This process is even more complex if the memory manager performs rebalancing concurrently with threads accessing the memory for other reads and writes.

Append-only log structures are write-friendly, placing new data items at the end of a linear data structure called a "log" (Figure 5.4 (b)). To update an item, it simply inserts a new item to the log

[2]Binary prefixes (powers of 2) end with an "i" suffix, whereas SI prefixes (powers of 10) have no "i" suffix.

that overrides the previous value. Inserts and updates thus access memory sequentially, incurring fewer cache and TLB misses, making logs particularly suited for bulk data writes. This approach is common in flash memory stores due to the high cost of random flash writes [5, 6, 91], but has been used in only a few in-memory key-value systems [112].

Garbage collection is crucial to space efficiency. It reclaims space occupied by overwritten and deleted objects by moving live objects to a new log and removing the old log. Unfortunately, garbage collection is costly and often reduces performance because of the large amount of data it must copy, trading memory efficiency against request processing speed.

MICA's memory allocator: MICA uses separate memory allocators for cache and store semantics. Its cache mode uses a log structure with inexpensive garbage collection and in-place update support (Section 5.3.3). MICA's allocator provides fast inserts and updates, and exploits cache semantics to eliminate log garbage collection and drastically simplify free space defragmentation. Its store mode uses segregated fits [118, 143] that share the unified memory space to avoid rebalancing size classes (Section 5.3.3).

Indexing: Read-oriented vs. Write-friendly

Read-oriented index: Common choices for indexing are hash tables [58, 101, 108] or tree-like structures [96]. However, conventional data structures are much slower for writes compared to reads; hash tables examine many slots to find a space for the new item [58], and trees may require multiple operations to maintain structural invariants [96].

Write-friendly index: Hash tables using *chaining* [101, 108] can insert new items without accessing many memory locations, but they suffer a time-space tradeoff: by having long chains (few hash buckets), an item lookup must follow a long chain of items, this requiring multiple random dependent memory accesses; when chains are short (many hash buckets), memory overhead to store chaining pointers increases. *Lossy data structures* are rather unusual in in-memory key-value storage and studied only in limited contexts [23], but it is the standard design in hardware indexes such as CPU caches [74].

MICA's index: MICA uses new index data structures to offer both high-speed read and write. In cache mode, MICA's lossy index also leverages the cache semantics to achieve high insertion speed; it evicts an old item in the hash table when a hash collision occurs instead of spending system resources to resolve the collision. By using the memory allocator's eviction support, the MICA lossy index can avoid evicting recently-used items (Section 5.3.3). The MICA lossless index uses *bulk chaining*, which allocates cache line-aligned space to a bucket for each chain segment. This keeps the chain length short and space efficiency high (Section 5.3.3).

5.3 MICA Design

This section describes each component in MICA and discusses how they operate together to achieve its goals.

5.3.1 Parallel Data Access

This section explains how CPU cores access data in MICA, but assumes that cores process only the requests for which they are responsible. Later in Section 5.3.2, we discuss how MICA assigns remote requests to CPU cores.

Keyhash-Based Partitioning

MICA creates one or more partitions per CPU core and stores key-value items in a partition determined by their key. Such horizontal partitioning is often used to shard *across* nodes [6, 44], but some key-value storage systems also use it across cores within a node [19, 85, 101].

MICA uses a *keyhash* to determine each item's partition. A keyhash is the 64-bit hash of an item's key calculated by the client and used throughout key-value processing in MICA. MICA uses the first few high order bits of the keyhash to obtain the partition index for the item.

Keyhash partitioning uniformly maps keys to partitions, reducing the request distribution imbalance. For example, in a Zipf-distributed population of size 192×2^{20} (192 Mi) with skewness 0.99 as used by YCSB [38],[3] the most popular key is 9.3×10^6 times more frequently accessed than the average; after partitioning keys into 16 partitions, however, the most popular partition is only 53% more frequently requested than the average.

MICA retains high throughput under this remaining partition-level skew because it can process requests in "hot" partitions more efficiently, for two reasons. First, a partition is popular *because* it contains "hot" items; these hot items naturally create locality in data access. With high locality, MICA experiences fewer CPU cache misses when accessing items. Second, the skew causes packet I/O to be more efficient for popular partitions (described in Section 5.3.2). As a result, throughput for the Zipf-distributed workload is 86% of the uniformly-distributed workload, making MICA's partitioned design practical even under skewed workloads.

Operation Modes

MICA can operate in EREW (Exclusive Read Exclusive Write) or CREW (Concurrent Read Exclusive Write). *EREW* assigns a single CPU core to each partition for all operations. No concurrent access to partitions eliminates synchronization and inter-core communication, making MICA scale linearly with CPU cores. *CREW* allows any core to read partitions, but only a single core can write. This combines the benefit of concurrent read and exclusive write; the former allows all cores to process read requests, while the latter still reduces expensive cache line transfer. CREW handles reads efficiently under highly skewed load, at the cost of managing read-write conflicts. MICA minimizes the synchronization cost with efficient optimistic locking [148] (Section 5.3.3).

Supporting cache semantics in CREW, however, raises a challenge for read (`GET`) requests: During a `GET`, the cache may need to update cache management information. For example, policies such as LRU use bookkeeping to remember recently used items, which can cause conflicts and cache-line bouncing among cores. This, in turn, defeats the purpose of using exclusive writes.

[3] i-th key constitutes $1/(i^{0.99} H_{n,0.99})$ of total requests, where $H_{n,0.99} = \sum_{i=1}^{n} (1/i^{0.99})$ and n is the total number of keys.

To address this problem, we choose an approximate approach: MICA counts reads only from the exclusive-write core. Clients round-robin CREW reads across all cores in a NUMA domain, so this is effectively a sampling-based approximation to, e.g., LRU replacement as used in MICA's item eviction support (Section 5.3.3).

To show performance benefits of EREW and CREW, our MICA prototype also provides the CRCW (Concurrent Read Concurrent Write) mode, in which MICA allows multiple cores to read and write any partition. This effectively models concurrent access to the shared data in non-partitioned key-value systems.

5.3.2 Network Stack

The network stack in MICA provides *network I/O* to transfer packet data between NICs and the server software, and *request direction* to route requests to an appropriate CPU core to make subsequent key-value processing efficient.

Exploiting the small key-value items that MICA targets, request and response packets use UDP. Despite clients not benefiting from TCP's packet loss recovery and flow/congestion control, UDP has been used widely for read requests (e.g., `GET`) in large-scale deployments of in-memory key-value storage systems [108] for low latency and low overhead. Our protocol includes sequence numbers in packets, and our application relies on the idempotency of `GET` and `PUT` operations for simple and stateless application-driven loss recovery, if needed: some queries may not be useful past a deadline, and in many cases, the network is provisioned well, making retransmission rare and congestion control less crucial [108].

Direct NIC Access

MICA uses Intel's DPDK [76] instead of standard socket I/O. This allows our user-level server software to control NICs and transfer packet data with minimal overhead. MICA differs from general network processing [46, 72, 116] that has used direct NIC access in that MICA is an application that processes high-level key-value requests.

In NUMA (non-uniform memory access) systems with multiple CPUs, NICs may have different affinities to CPUs. For example, our evaluation hardware has two CPUs, each connected to two NICs via a direct PCIe bus. MICA uses NUMA-aware memory allocation so that each CPU and NIC only accesses packet buffers stored in their respective NUMA domains.

MICA uses NIC multi-queue support to allocate a dedicated RX and TX queue to each core. Cores exclusively access their own queues without synchronization in a similar way to EREW data access. By directing a packet to an RX queue, the packet can be processed by a specific core, as we discuss in Section 5.3.2.

Burst packet I/O: MICA uses the DPDK's burst packet I/O to transfer multiple packets (up to 32 in our implementation) each time it requests packets from RX queues or transmits them to TX queues. Burst I/O reduces the per-packet cost of accessing and modifying the queue, while adding only trivial delay to request processing because the burst size is small compared to the packet processing rate.

Importantly, burst I/O helps handle skewed workloads. A core processing popular partitions spends more time processing requests, and therefore performs packet I/O less frequently. The

lower I/O frequency increases the burst size, reducing the per-packet I/O cost (Section 5.4.2). Therefore, popular partitions have more CPU available for key-value processing. An unpopular partition's core has higher per-packet I/O cost, but handles fewer requests.

Zero-copy processing: MICA avoids packet data copy throughout RX/TX and request processing. MICA uses MTU-sized packet buffers for RX even if incoming requests are small. Upon receiving a request, MICA avoids memory allocation and copying by reusing the request packet to construct a response: it flips the source and destination addresses and ports in the header and updates only the part of the packet payload that differs between the request and response.

Client-Assisted Hardware Request Direction

Modern NICs help scale packet processing by directing packets to different RX queues using hardware features such as Receiver-Side Scaling (RSS) and Flow Director (FDir) [46, 72, 116] based on the packet header.

Because each MICA key-value request is an individual packet, we wish to use *hardware* packet direction to directly send packets to the appropriate queue based upon the key. Doing so is much more efficient than redirecting packets in software. Unfortunately, the NIC alone cannot provide key-based request direction: RSS and FDir cannot classify based on the packet payload, and cannot examine variable length fields such as request keys.

Client assistance: We instead take advantage of the opportunity to co-design the client and server. The client caches information from a server directory about the operation mode (EREW or CREW), number of cores, NUMA domains, and NICs, and number of partitions. The client then embeds the request direction information in the packet header: If the request uses exclusive data access (read/write on EREW and write on CREW), the client calculates the partition index from the keyhash of the request. If the request can be handled by any core (a CREW read), it picks a server core index in a round-robin way (across requests, but in the same NUMA domain (Section 5.3.2)). Finally, the client encodes the partition or core index as the UDP destination port.[4] The server programs FDir to use the UDP destination port, without hashing, ("perfect match filter" [77]), as an index into a table mapping UDP port numbers to a destination RX queue. Key hashing only slightly burdens clients. Using fast string hash functions such as CityHash [34], a single client machine equipped with dual 6-core CPUs on our testbed can generate over 40 M requests/second with client-side key hashing. Clients include the keyhash in requests, and servers reuse the embedded keyhash when they need a keyhash during the request processing to benefit from offloaded hash computation.

Client-assisted request direction using NIC hardware allows efficient request processing. Our results in Section 5.4.5 show that an optimized software-based request direction that receives packets from any core and distributes them to appropriate cores is significantly slower than MICA's hardware-based approach.

[4]To avoid confusion between partition indices and the core indices, we use different ranges of UDP ports; a partition may be mapped to a core whose index differs from the partition index.

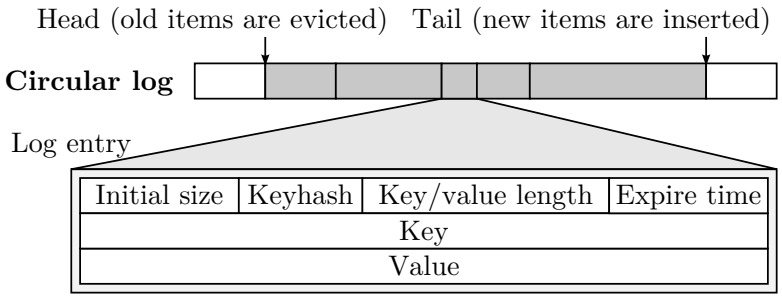


Figure 5.5: Design of a circular log.

5.3.3 Data Structure

MICA, in cache mode, uses *circular logs* to manage memory for key-value items and *lossy concurrent hash indexes* to index the stored items. Both data structures exploit cache semantics to provide fast writes and simple memory management. Each MICA partition consists of a single circular log and lossy concurrent hash index.

MICA provides a store mode with straightforward extensions using segregated fits to allocate memory for key-value items and *bulk chaining* to convert the lossy concurrent hash indexes into lossless ones.

Circular Log

MICA stores items in its circular log by appending them to the *tail* of the log (Figure 5.5). This results in a space-efficient packing. It updates items in-place as long as the new size of the key+value does not exceed the size of the item when it was first inserted. The size of the circular log is bounded and does not change, so to add a new item to a full log, MICA evicts the oldest item(s) at the *head* of the log to make space.

Each entry includes the key and value length, key, and value. To locate the next item in the log and support item resizing, the entry contains the initial item size, and for fast lookup, it stores the keyhash of the item. The entry has an expire time set by the client to ignore stale data.

Garbage collection and defragmentation: The circular log eliminates the expensive garbage collection and free space defragmentation that are required in conventional log structures and dynamic memory allocators. Previously deleted items in the log are automatically collected and removed when new items enter the log. Almost all free space remains contiguously between the tail and head.

Exploiting the eviction of live items: Items evicted at the head are not reinserted to the log even if they have not yet expired. In other words, the log may delete items without clients knowing it. This behavior is valid in cache workloads; a key-value store must evict items when it becomes full. For example, Memcached [99] uses LRU to remove items and reserve space for new items.

MICA uses this item eviction to implement common eviction schemes at low cost. Its "natural" eviction is FIFO. MICA can provide LRU by reinserting any requested items at the tail because only the least recently used items are evicted at the head. MICA can approximate LRU by reinserting requested items selectively—by ignoring items recently (re)inserted and close to the

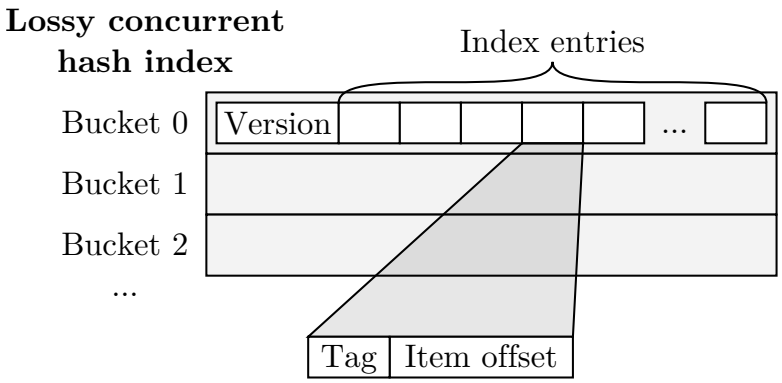


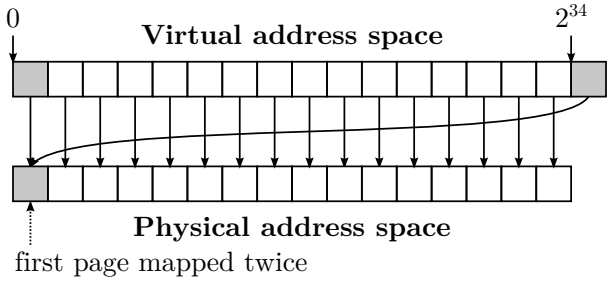
Figure 5.6: Design of a lossy concurrent hash index.

tail; this approximation offers eviction similar to LRU without frequent reinserts, because recently accessed items remain close to the tail and far from the head.

A second challenge for conventional logs is that any reference to an evicted item becomes dangling. MICA does not store back pointers in the log entry to discover all references to the entry; instead, it provides detection, and removes dangling pointers incrementally (Section 5.3.3).

Low-level memory management: MICA uses hugepages and NUMA-aware allocation. Hugepages (2 MiB in x86-64) use fewer TLB entries for the same amount of memory, which significantly reduces TLB misses during request processing. Like the network stack, MICA allocates memory for circular logs such that cores access only local memory.

Without explicit range checking, accessing an entry near the end of the log (e.g., at $2^{34} - 8$ in the example below) could cause an invalid read or segmentation fault by reading off the end of the range. To avoid such errors without range checking, MICA manually maps the virtual memory addresses right after the end of the log to the same physical page as the first page of the log, making the entire log appear locally contiguous:



Our MICA prototype implements this scheme in userspace by mapping a pool of hugepages to virtual addresses using the `mmap()` system call.

Lossy Concurrent Hash Index

MICA's hash index locates key-value items in the log using a set-associative cache similar to that used in CPU caches. As shown in Figure 5.6, a hash index consists of multiple buckets (configurable for the workload), and each bucket has a fixed number of index entries (configurable in the source code; 15 in our prototype to occupy exactly two cache lines). MICA uses a portion

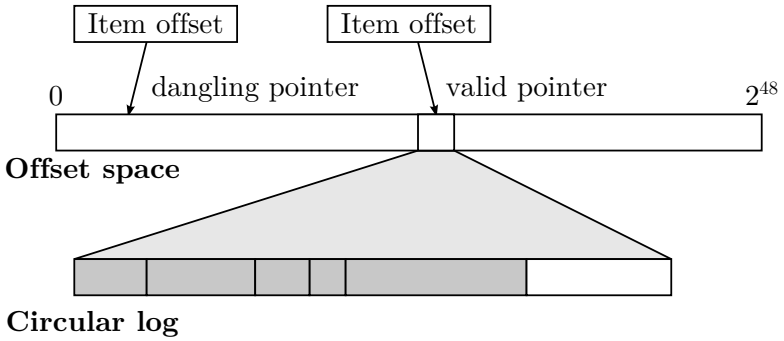


Figure 5.7: Offset space for dangling pointer detection.

of the keyhashes to determine an item's bucket; the item can occupy any index entry of the bucket unless there is a duplicate.

Each index entry contains partial information for the item: a tag and the item offset within the log. A tag is another portion of the indexed item's keyhash used for filtering lookup keys that do not match: it can tell whether the indexed item will never match against the lookup key by comparing the stored tag and the tag from the lookup keyhash. We avoid using a zero tag value by making it one because we use the zero value to indicate an empty index entry. Items are deleted by writing zero values to the index entry; the entry in the log will be automatically garbage collected.

Note that the parts of keyhashes used for the partition index, the bucket number, and the tag do not overlap. Our prototype uses 64-bit keyhashes to provide sufficient bits.

Lossiness: The hash index is lossy. When indexing a new key-value item into a full bucket of the hash index, the index evicts an index entry to accommodate the new item. The item evicted is determined by its age; if the item offset is most behind the tail of the log, the item is the oldest (or least recently used if the log is using LRU), and the associated index entry of the item is reclaimed.

This lossy property allows fast insertion. It avoids expensive resolution of hash collisions that lossless indexes of other key-value stores require [58, 101]. As a result, MICA's insert speed is comparable to lookup speed.

Handling dangling pointers: When an item is evicted from the log, MICA does not delete its index entry. Although it is possible to store back pointers in the log entry, updating the hash index requires a random memory write and is complicated due to locking if the index is being accessed concurrently, so MICA does not. As a result, index pointers can "dangle," pointing to invalid entries.

To address this problem, MICA uses large pointers for head/tail and item offsets. As depicted in Figure 5.7, MICA's index stores log offsets that are wider than needed to address the full size of the log (e.g., 48-bit offsets vs 34 bits for a 16 GiB log). MICA detects a dangling pointer before using it by checking if the difference between the log tail and the item offset is larger than the actual log size.[5] If the tail wraps around the 48-bit size, however, a dangling pointer may appear valid again, so MICA scans the index incrementally to remove stale pointers.

This scanning must merely complete a full cycle before the tail wraps around in its wide offset space. The speed at which it wraps is determined by the increment rate of the tail and

[5] $(\text{Tail} - \text{ItemOffset} + 2^{48}) \bmod 2^{48} > \text{LogSize}$.

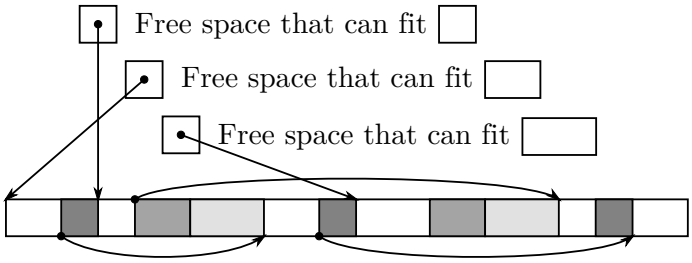


Figure 5.8: Segregated free lists for a unified space.

the width of the item offset. In practice, full scanning is infrequent even if writes occur very frequently. For example, with 48-bit offsets and writes occurring at 2^{30} bytes/second (millions of operations/second), the tail wraps every 2^{48-30} seconds. If the index has 2^{24} buckets, MICA must scan only 2^{6} buckets per second, which adds negligible overhead.

Supporting concurrent access: MICA's hash index must behave correctly if the system permits concurrent operations (e.g., CREW). For this, each bucket contains a 32-bit version number. It performs reads optimistically using this version counter to avoid generating memory writes while satisfying `GET` requests [58, 96, 148]. When accessing an item, MICA checks if the initial state of the version number of the bucket is even-numbered, and upon completion of data fetch from the index and log, it reads the version number again to check if the final version number is equal to the initial version number. If either check fails, it repeats the read request processing from the beginning. For writes, MICA increments the version number by one before beginning, and increments the version number by one again after finishing all writes. In CRCW mode, which allows multiple writers to access the same bucket, a writer also spins until the initial version number is even (i.e., no other writers to this bucket) using a compare-swap operation instruction.

Our MICA prototype uses different code to optimize locking. It uses conventional instructions to manipulate version numbers to exploit memory access ordering on the x86 architecture [148] in CREW mode where there is only one writer. EREW mode does not require synchronization between cores, so MICA ignores version numbers. Because of such a hard-coded optimization, the current prototype lacks support for runtime switching between the operation modes.

Multi-stage prefetching: To retrieve or update an item, MICA must perform request parsing, hash index lookup, and log entry retrieval. These stages cause random memory access that can significantly lower system performance if cores stall due to CPU cache and TLB misses.

MICA uses multi-stage prefetching to interleave computation and memory access. MICA applies memory prefetching for random memory access done at each processing stage in sequence. For example, when a burst of 8 RX packets arrives, MICA fetches packets 0 and 1 and *prefetches* packets 2 and 3. It decodes the requests in packets 0 and 1, and prefetches buckets of the hash index that these requests will access. MICA continues packet payload prefetching for packets 4 and 5. It then prefetches log entries that may be accessed by the requests of packets 0 and 1 while prefetching the hash index buckets for packets 2 and 3, and the payload of packet 6 and 7. MICA continues this pipeline until all requests are processed.

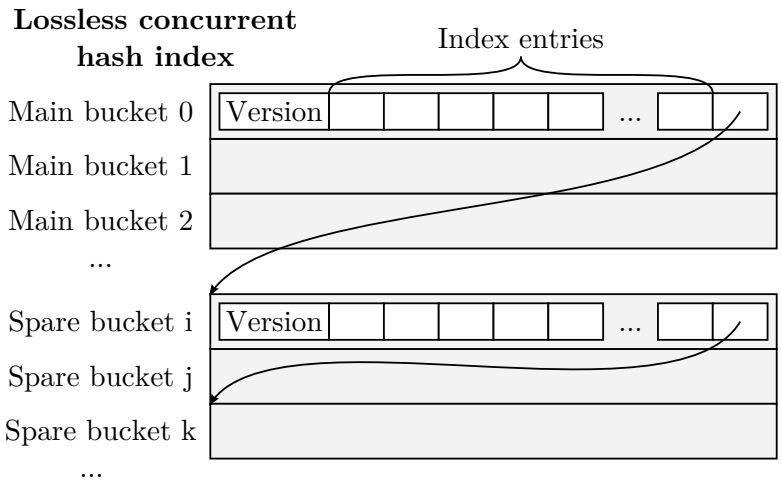


Figure 5.9: Bulk chaining in MICA's lossless hash index.

Store Mode

The store mode of MICA uses segregated fits [118, 143] similar to fast malloc implementations [89], instead of the circular log. Figure 5.8 depicts this approach. MICA defines multiple size classes incrementing by 8 bytes covering all supported item sizes, and maintains a freelist for each size class (a linked list of pointers referencing unoccupied memory regions that are at least as large as the size class). When a new item is inserted, MICA chooses the smallest size class that is at least as large as the item size and has any free space. It stores the item in the free space, and inserts any unused region of the free space into a freelist that matches that region's size. When an item is deleted, MICA coalesces any adjacent free regions using boundary tags [86] to recreate a large free region.

MICA's segregated fits differ from the simple segregated storage used in Memcached [58, 99]. MICA maintains a unified space for all size classes; on the contrary, Memcached's SLAB allocator dynamically assigns memory blocks to size classes, which effectively partitions the memory space according to size classes. The unified space of MICA eliminates the need to rebalance size classes unlike the simple segregated storage. Using segregated fits also makes better use of memory because MICA already has partitioning done with keyhashes; a SLAB allocator introducing another partitioning would likely waste memory by allocating a whole block for only a few items, resulting in low memory occupancy.

MICA converts its lossy concurrent hash index into a lossless hash index by using *bulk chaining*. Bulk chaining is similar to the traditional chaining method in hash tables; it adds more memory space to the buckets that contain an excessive number of items.

Figure 5.9 shows the design of the lossless hash index. MICA uses the lossy concurrent hash index as the main buckets and allocates space for separate spare buckets that are fewer than the main buckets. When a bucket experiences an overflow, whether it is a main bucket or spare bucket, MICA adds an unused spare bucket to the full bucket to form a bucket chain. If there are no more spare buckets available, MICA rejects the new item and returns an out-of-space error to the client.

This data structure is friendly to memory access. The main buckets store most of the items (about 95%), keeping the number of random memory read for an index lookup close to 1; as a

comparison, cuckoo hashing [114] used in improved Memcached systems [58] would require 1.5 random memory accesses per index lookup in expectation. MICA also allows good memory efficiency; because the spare buckets only store overflow items, making the number of spare buckets 10% of the main buckets allows the system to store the entire dataset of 192 Mi items in our experiments (Section 5.4).

5.4 Evaluation

We answer four questions about MICA in this section:

- *Does it perform well under diverse workloads?*
- *Does it provide good latency?*
- *How does it scale with more cores and NIC ports?*
- *How does each component affect performance?*

Our results show that MICA has consistently high throughput and low latency under a variety of workloads. It scales nearly linearly, using CPU cores and NIC ports efficiently. Each component of MICA is needed. MICA achieves 65.6–76.9 million operations/second (Mops), which is over 4–13.5X faster than the next fastest system; the gap widens as the fraction of write requests increases.

MICA is written in 12 k lines of C and runs on x86-64 GNU/Linux. Packet I/O uses the Intel DPDK 1.4.1 [76].

Compared systems: We use custom versions of open-source Memcached [99], MemC3 [58], Masstree [96], and RAMCloud [112]. The revisions of the original code we used are: Memcached: 87e2f36; MemC3: an internal version; Masstree: 4ffb946; RAMCloud: a0f6889.

Note that the compared systems often offer additional capabilities compared to others. For example, Masstree can handle range queries, and RAMCloud offers low latency processing on InfiniBand; on the other hand, these key-value stores do not support automatic item eviction as Memcached systems do. Our evaluation focuses on the performance of the standard features (e.g., single key queries) common to all the compared systems, rather than highlighting the potential performance impact from these semantic differences.

Modifications to compared systems: We modify the compared systems to use our lightweight network stack to avoid using expensive socket I/O or special hardware (e.g., InfiniBand). When measuring Memcached's baseline latency, we use its original network stack using the kernel to obtain the latency distribution that typical Memcached deployments would experience. Our experiments do not use any client-side request batching. We also modified these systems to invoke memory allocation functions though our framework if they use hugepages, because the DPDK requests all hugepages from the OS at initialization and would make the unmodified systems inoperable if they request hugepages from the OS; we kept other memory allocations using no hugepages as-is. Finally, while running experiments, we found that statistics collection in RAMCloud caused lock contention, so we disabled it for better multi-core performance.

5.4.1 Evaluation Setup

Server/client configuration: MICA server runs on a machine equipped with dual 8-core CPUs (Intel Xeon E5-2680 @2.70 GHz), 64 GiB of total system memory, and eight 10-Gb Ethernet ports (four Intel X520-T2's). Each CPU has 20 MiB of L3 cache. We disabled logical processor support ("Hyper-Threading"). Each CPU accesses the 32 GiB of the system memory that resides in its local NUMA domain over a quad-channel DDR3-1600 bus. Each CPU socket is directly connected to two NICs using PCIe gen2. Access to hardware resources in the remote NUMA domain uses an interconnect between two CPUs (Intel QuickPath).

We reserved the half of the memory (16 GiB in each NUMA domain) for hugepages regardless of how MICA and the compared systems use hugepages.

MICA allocates 16 partitions in the server, and these partitions are assigned to different cores. We configured the cache version of MICA to use approximate LRU to evict items; MICA reinserts any recently accessed item at the tail if the item is closer to the head than to the tail of the circular log.

Two client machines with dual 6-core CPUs (Intel Xeon L5640 @2.27 GHz) and two Intel X520-T2's generate workloads. The server and clients are directly connected without a switch. Each client is connected to the NICs from both NUMA domains of the server, allowing a client to send a request to any server CPU.

Workloads: We explore different aspects of the systems by varying the item size, skew, and read-write ratio.

We use three datasets as shown in the following table:

Dataset	Key Size (B)	Value Size (B)	Count
Tiny	8	8	192 Mi
Small	16	64	128 Mi
Large	128	1024	8 Mi

We use two workload types: *uniform* and *skewed*. Uniform workloads use the same key popularity for all requests; skewed workloads use a non-uniform key popularity that follows a Zipf distribution of skewness 0.99, which is the same as YCSB's [38].

Workloads have a *varied ratio between* `GET` *and* `PUT`. 50% `GET` (50% `PUT`) workloads are write-intensive, and 95% `GET` (5% `PUT`) workloads are read-intensive. They correspond to YCSB's A and B workloads, respectively.

Workload generation: We use our custom key-value request generator that uses similar techniques to our lightweight network stack to send more than 40 Mops of key-value requests per machine to saturate the link.[6] It uses approximation techniques of Zipf distribution generation [68, 113] for fast skewed workload generation.

To find the maximum *meaningful* throughput of a system, we adjust the workload generation rate to allow only marginal packet losses ($< 1\%$ at any NIC port). We could generate requests at the highest rate to cause best-effort request processing (which can boost measured throughput more than 10%), as is commonly done in throughput measurement of software routers [46, 72], but we avoid this method because we expect that real deployments of in-memory key-value stores

[6]MICA clients are still allowed to use standard socket I/O in cases where the socket overhead on the client machines is acceptable because the MICA server and clients use the plain UDP protocol.

would not tolerate excessive packet losses, and such flooding can distort the intended skew in the workload by causing biased packet losses at different cores.

The workload generator does not receive every response from the server. On our client machines, receiving packets whose size is not a multiple of 64 bytes is substantially slower due to an issue in the PCIe bus [71].

The workload generator works around this slow RX by sampling responses to perform fewer packet RX from NIC to CPU. It uses its real source MAC addresses for only a fraction of requests, causing its NIC to drop the responses to the other requests. By looking at the sampled responses, the workload generator can validate that the server has correctly processed the requests. Our server is unaffected from this issue and performs full packet RX.

5.4.2 System Throughput

We first compare the full-system throughput. MICA uses EREW with all 16 cores. However, we use a different number of cores for the other systems to obtain their best throughput because some of them (Memcached, MemC3, and RAMCloud) achieve higher throughput with fewer cores (Section 5.4.4). The throughput numbers are calculated from the actual number of responses sent to the clients after processing the requests at the server. We denote the cache version of MICA by MICA-c and the store version of MICA by MICA-s.

Figure 5.10 (top) plots the experiment result using *tiny key-value items*. MICA performs best, regardless of the skew or the `GET` ratio. MICA's throughput reaches 75.5–76.9 Mops for uniform workloads and 65.6–70.5 Mops for skewed ones; its parallel data access does not incur more than a 14% penalty for skewed workloads. MICA uses 54.9–66.4 Gbps of network bandwidth at this processing speed—this speed is very close to 66.6 Gbps that our network stack can handle when doing packet I/O only. The next best system is Masstree at 16.5 Mops, while others are below 6.1 Mops. All systems except MICA suffer noticeably under write-intensive 50% `GET`.

Small key-value items show similar results in Figure 5.10 (middle). However, the gap between MICA and the other systems shrinks because MICA becomes network bottlenecked while the other systems never saturate the network bandwidth in our experiments.

Large key-value items, shown in Figure 5.10 (bottom), exacerbates the network bandwidth bottleneck, further limiting MICA's throughput. MICA achieves 12.6–14.6 Mops for 50% `GET` and 8.6–9.4 Mops for 95% `GET`; note that MICA shows high throughput with lower `GET` ratios, which require less network bandwidth as the server can omit the key and value from the responses. Unlike MICA, however, all other systems achieve higher throughput under 95% `GET` than under 50% `GET` because these systems are bottleneck locally, not by the network bandwidth.

In those measurements, MICA's cache and store modes show only minor differences in the performance. We will refer to the cache version of MICA (MICA-c) simply as MICA in the rest of the evaluation for simplicity.

Skew resistance: Figure 5.11 compares the per-core throughput under uniform and skewed workloads of 50% `GET` with tiny items. MICA uses EREW. Several cores process more requests under the skewed workload than under the uniform workload because they process requests more efficiently. The skew in the workload increases the RX burst size of the most loaded core from 10.2 packets per I/O to 17.3 packets per I/O, reducing its per-packet I/O cost, and the higher data locality caused by the workload skew improves the average cache hit ratio of all cores from

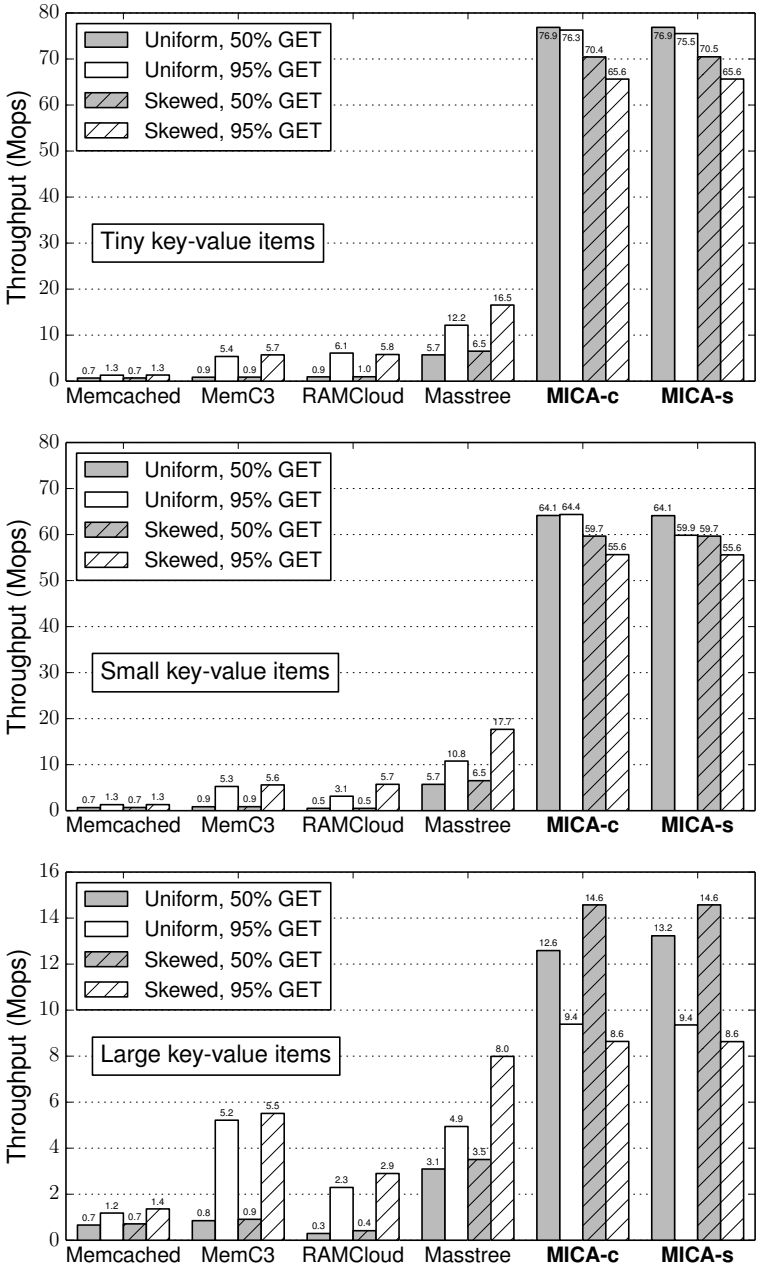


Figure 5.10: End-to-end throughput of in-memory key-value systems. All systems use our lightweight network stack that does not require request batching. The bottom graph (large key-value items) uses a different Y scale from the first two graphs'.

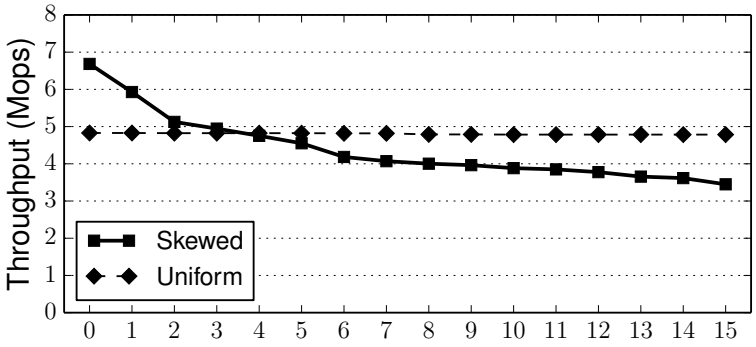


Figure 5.11: Per-core breakdown of end-to-end throughput.

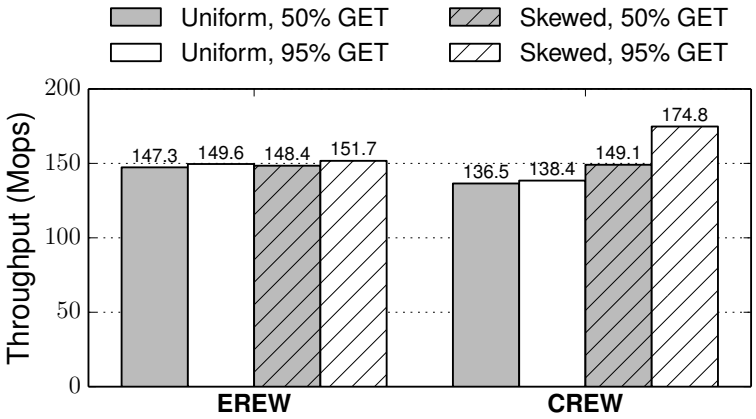


Figure 5.12: Local throughput of key-value data structures.

67.8% to 77.8%. A local benchmark in Figure 5.12 (without network processing) also shows that skewed workloads grant good throughput for local key-value processing due to the data locality. These results further justify the partitioned design of MICA and explains why MICA retains high throughput under skewed workloads.

Summary: MICA's throughput reaches 76.9 Mops, at least 4X faster than the next best system. MICA delivers consistent performance across different skewness, write-intensiveness, and key-value sizes.

5.4.3 Latency

To show that MICA achieves comparably low latency while providing high throughput, we compare MICA's latency with that of the original Memcached implementation that uses the kernel network stack. To measure the end-to-end latency, clients tag each request packet with the current timestamp. When receiving responses, clients compare the current timestamp and the previous timestamp echoed back in the responses. We use uniform 50% `GET` workloads on tiny items. MICA uses EREW. The client varies the request rate to observe the relationship between throughput and latency.

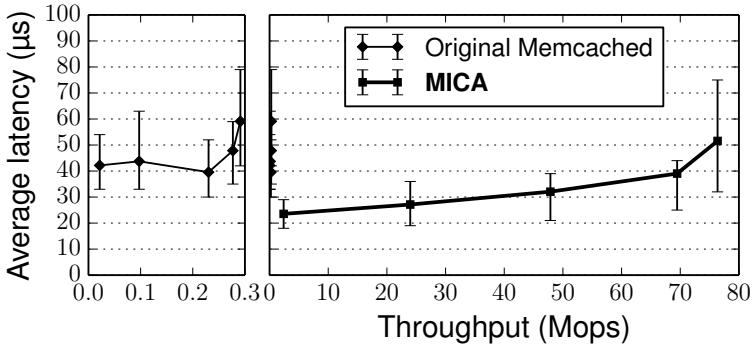


Figure 5.13: End-to-end latency of the original Memcached and MICA as a function of throughput.

Figure 5.13 plots the end-to-end latency as a function of throughput; the error bars indicate 5th- and 95th-percentile latency. The original Memcached exhibits almost flat latency up to certain throughput, whereas MICA shows varied latency depending on the throughput it serves. MICA's latency lies between 24–52 µs. At the similar latency level of 40 µs, MICA shows 69 Mops—more than two orders of magnitude faster than Memcached.

Because MICA uses a single round-trip per request unlike RDMA-based systems [103], we believe that MICA provides best-in-class low-latency key-value operations.

Summary: MICA achieves both high throughput and latency near the network minimum.

5.4.4 Scalability

CPU scalability: We vary now the number of CPU cores and compare the end-to-end throughput. We allocate cores evenly to both NUMA domains so that cores can efficiently access NICs connected to their CPU socket. We use skewed workloads on tiny items because it is generally more difficult for partitioned stores to handle skewed workloads. MICA uses EREW.

Figure 5.14 (upper) compares core scalability of systems with 50% `GET`. Only MICA and Masstree perform better with more cores. Memcached, MemC3, and RAMCloud scale poorly, achieving their best throughput at 2 cores.

The trend continues for 95% `GET` requests in Figure 5.14 (lower); MICA and Masstree scale well as before. The rest also achieve higher throughput, but still do not scale. Note that some systems scale differently from their original papers. For example, MemC3 achieves 5.7 Mops at 4 cores, while the original paper shows 4.4 Mops at 16 cores [58]. This is because using our network stack instead of their network stack reduces I/O cost, which may expose a different bottleneck (e.g., key-value data structures) that can change the optimal number of cores for the best throughput.

Network scalability: We also change the available network bandwidth by varying the number of NIC ports we use for request processing. Figure 5.15 shows that MICA again scales well with high network bandwidth, because MICA can use almost all available network bandwidth for request processing. The `GET` ratio does not affect the result for MICA significantly. This result suggests that MICA can possibly scale further with higher network bandwidth (e.g., multiple

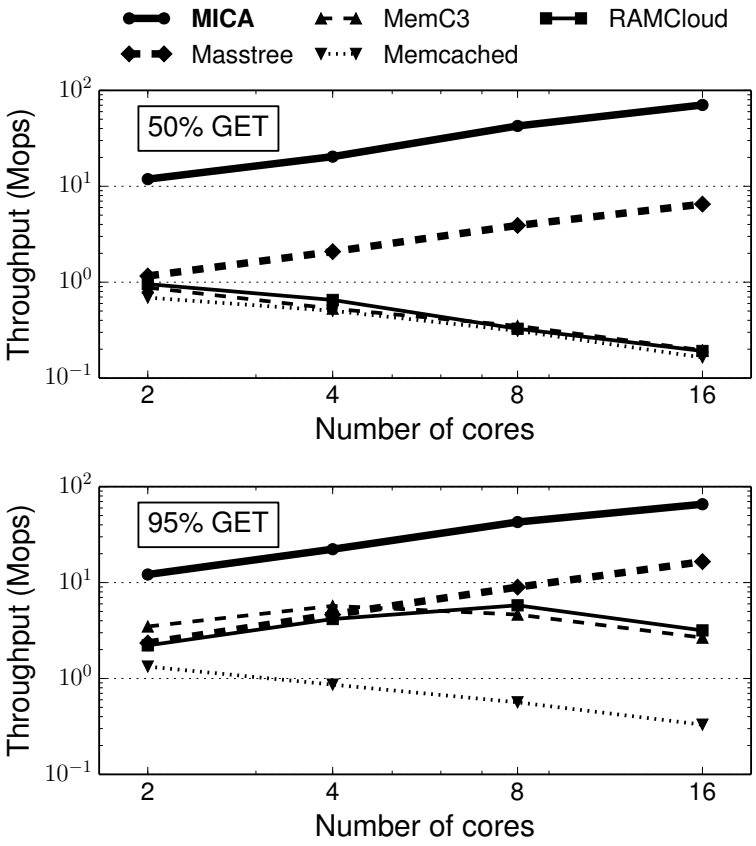


Figure 5.14: End-to-end throughput of in-memory key-value systems using a varying number of cores. All systems use our lightweight network stack.

40 Gbps NICs). MICA and Masstree achieve similar performance under the 95% `GET` workload when using 2 ports, but Masstree and other systems do not scale well with more ports.

Summary: MICA scales well with more CPU cores and more network bandwidth, even under write-intensive workloads where other systems tend to scale worse.

5.4.5 Benefits of the Holistic Approach

In this section, we demonstrate how each component of MICA contributes to its performance. Because MICA is a coherent system that exploits the synergy between its components, we compare different approaches for one component while keeping the other components the same.

Parallel data access: We use end-to-end experiments to measure how different data access modes affect the system performance. We use tiny items only. Figure 5.16 shows the end-to-end results. EREW shows consistently good performance. CREW achieves slightly higher throughput with high `GET` ratios on skewed workloads compared to EREW (white bars at 95% `GET`) because despite the overheads from bucket version management, CREW can use multiple cores to read popular items without incurring excessive inter-core communication. While CRCW performs

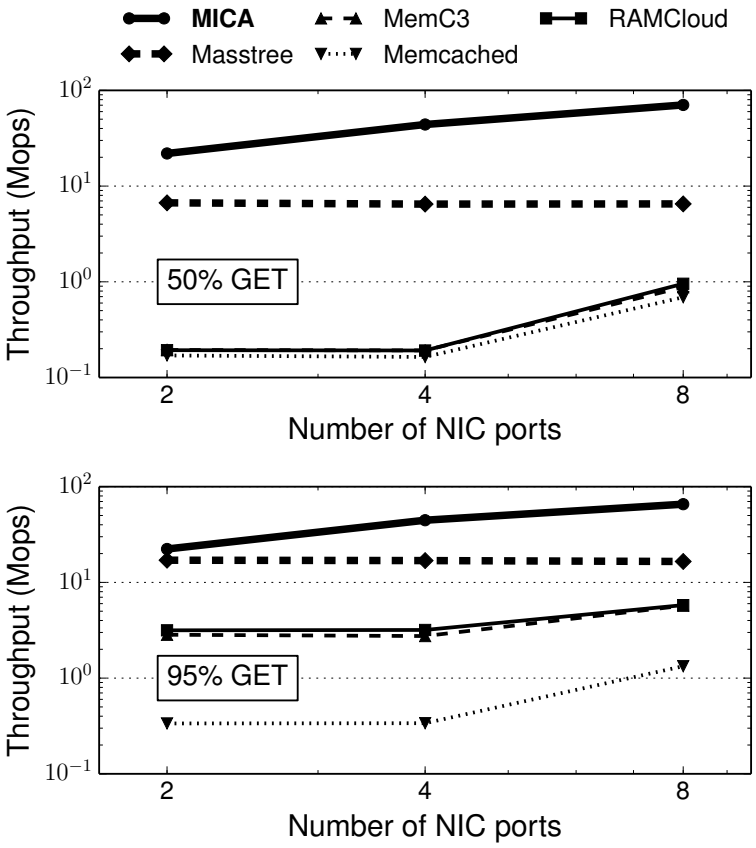


Figure 5.15: End-to-end throughput of in-memory key-value systems using a varying number of NIC ports. All systems use our lightweight network stack.

better than any other compared systems (Section 5.4.2), CRCW offers no benefit over EREW and CREW; this suggests that we should avoid CRCW.

Network stack: As shown in Section 5.4.2, switching Masstree to our network stack resulted in much higher throughput (16.5 Mops without request batching) than the throughput from the original paper (8.9 Mops with request batching [96]); this indicates that our network stack provides efficient I/O for key-value processing.

The next question is how important it is to use hardware to direct requests for exclusive access in MICA. To compare with MICA's client-assisted hardware request direction, we implemented software-only request direction: clients send requests to any server core in a round-robin way, and the server cores direct the received requests to the appropriate cores for EREW data access. We use Intel DPDK's queue to implement message queues between cores. We use 50% `GET` on tiny items.

Table 5.1 shows that software request direction achieves only 40.0–44.1% of MICA's throughput. This is due to the inter-core communication overhead of software request direction. Thus, MICA's request direction is crucial for realizing the benefit of exclusive access.

Key-value data structures: MICA's circular logs, lossy concurrent hash indexes, and bulk chaining permit high-speed read and write operations with simple memory management. Even

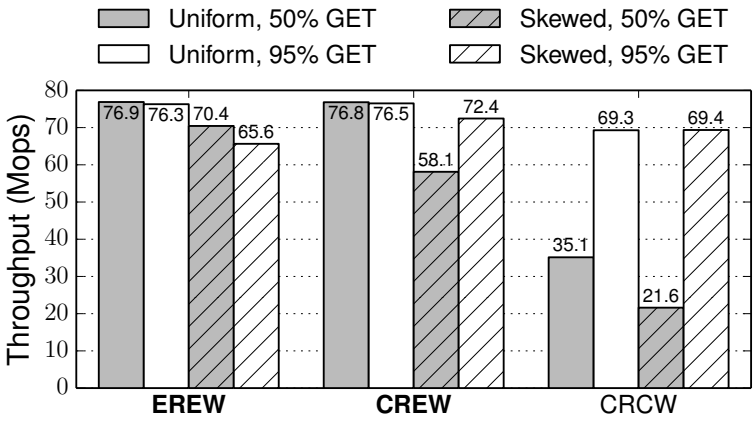


Figure 5.16: End-to-end performance using MICA's EREW, CREW, and CRCW.

Method	Workload	Throughput
Software-only	Uniform	33.9 Mops
	Skewed	28.1 Mops
Client-assisted hardware-based	Uniform	76.9 Mops
	Skewed	70.4 Mops

Table 5.1: End-to-end throughput of different request direction methods.

CRCW, the slowest data access mode of MICA, outperforms the second best system, Masstree (Section 5.4.2).

We also demonstrate that partitioning existing data structures does not simply grant MICA's high performance. For this, we compare MICA with "partitioned" Masstree, which uses one Masstree instance per core, with its support for concurrent access disabled in the source code. This is similar to MICA's EREW. We also use the same partitioning and request direction scheme.

Table 5.2 shows the result with skewed workloads on tiny items. Partitioned Masstree achieves only 8.2–27.3% of MICA's performance, with the throughput for 50% `GET` even lower than non-partitioned Masstree (Section 5.4.2). This indicates that to make best use of MICA's parallel data access and network stack, it is important to use key-value data structures that perform high-speed writes and to provide high efficiency with data partitioning.

In conclusion, the holistic approach is beneficial in achieving high performance; any missing component significantly degrades performance.

5.5 Related Work

Most DRAM stores are not partitioned: Memcached [99], RAMCloud [112], MemC3 [58], Masstree [96], and Silo [136] all have a single partition for each server node. Masstree and Silo show that partitioning can be efficient under some workloads but is slow under workloads with a skewed key popularity and many cross-partition transactions. MICA exploits burst I/O and

Method	Workload	Throughput
Partitioned	50% GET	5.8 Mops
Masstree	95% GET	17.9 Mops
MICA	50% GET	70.4 Mops
	95% GET	65.6 Mops

Table 5.2: End-to-end throughput comparison between partitioned Masstree and MICA using skewed workloads.

locality so that even in its exclusive EREW mode, loaded partitions run faster. It can do so because the simple key-value requests that it targets do not cross partitions.

Partitioned systems are fast with well-partitioned data. Memcached on Tilera [19], CPHash [101], and Chronos [85] are partitioned in-memory key-value systems that exclusively access partitioned hash tables to minimize lock contention and cache movement, similar to MICA's EREW partitions. These systems lack support for other partitioning such as MICA's CREW that can provide higher throughput under read-intensive skewed workloads.

H-Store [131] and VoltDB [139] use single-threaded execution engines that access their own partition exclusively, avoiding expensive concurrency control. Because workload skew can reduce system throughput, they require careful data partitioning, even using machine learning methods [115], and dynamic load balancing [85]. MICA achieves similar throughput under both uniform and skewed workloads without extensive partitioning and load balancing effort because MICA's keyhash-based partitioning mitigates the skew using and its request processing for popular partitions exploits burst packet I/O and cache-friendly memory access.

Several in-memory key-value systems focus on low latency request processing. RAMCloud achieves 4.9–15.3 µs end-to-end latency for small objects [121], and Chronos exhibits average latency of 10 µs and a 99th-percentile latency of 30 µs, on low latency networks such as InfiniBand. Pilaf [103] serves read requests using one-sided RDMA reads on a low-latency network. Our MICA prototype currently runs on 10-Gb Ethernet NIC whose base latency is much higher [60]; we plan to evaluate MICA on a low-latency network.

Prior work studies providing a high performance reliable transport service using low-level unreliable datagram services. The Memcached UDP protocol relies on application-level packet loss recovery [108]. Low-overhead user-level implementations for TCP such as mTCP [83] can offer reliable communication to Memcached applications without incurring high performance penalties. Low-latency networks such as InfiniBand often implement hardware-level reliable datagrams [103].

Affinity-Accept [116] uses Flow Director on the commodity NIC hardware to load balance TCP connections across multiple CPU cores. Chronos [85] directs remote requests to server cores using client-supplied information, similar to MICA; however, Chronos uses software-based packet classification whose throughput for small key-value requests is significantly lower than MICA's hardware-based classification.

Strict or complex item eviction schemes in key-value stores can be so costly that it can reduce system throughput significantly. MemC3 [58] replaces Memcached [99]'s original LRU with a CLOCK-based approximation to avoid contention caused by LRU list management. MICA's circular log and lossy concurrent hash index use its lossy property to support common eviction

schemes at low cost; the lossy concurrent hash index is easily extended to support lossless operations by using bulk chaining.

A worthwhile area of future work is applying MICA's techniques to semantically richer systems, such as those that are durable [112], or provide range queries [50, 96] or multi-key transactions [136]. Our results show that existing systems such as Masstree can benefit considerably simply by moving to a lightweight network stack; nevertheless, operations in these systems may cross partitions, it remains to be seen how to best harness the speed of exclusively accessed partitions.

5.6 Summary

MICA is an in-memory key-value store that provides high-performance, scalable key-value storage. It provides consistently high throughput and low latency for read/write-intensive workloads with a uniform/skewed key popularity. We demonstrate high-speed request processing with MICA's parallel data access to partitioned data, efficient network stack that delivers remote requests to appropriate CPU cores, and new lossy and lossless data structures that exploit properties of key-value workloads to provide high-speed write operations without complicating memory management.

Chapter 6

Conclusion

This dissertation demonstrates that new algorithms, data structures, and software architectures that are designed to exploit modern hardware and workload characteristics can store and process a larger number of fine-grained items at a higher speed than conventional systems built for generic hardware and agnostic of workloads. We use key-value stores as examples and present SILT and MICA. As an on-flash key-value store, SILT is parsimonious in its use of memory for indexing to allow storing and serving a large number of items while maintaining high performance. SILT uses ECT, a new indexing scheme to achieve memory efficient indexing. Accurate and fast analytic primitives for SILT and SILT-like systems help building their models to estimate performance metrics, improve the system design, and discover appropriate system parameters for higher performance. MICA is an in-memory key-value store that performs efficient parallel processing enabled by the use of advanced NIC features, and uses new key-value data structures, improving the throughput of key-value processing over the network by up to an order of magnitude compared to previous in-memory key-value systems.

6.1 Discussion and Future Work

We briefly discuss additional issues and highlight related future work.

6.1.1 Performance Impacts Caused by Supporting Complex Features

While this dissertation shows that SILT and MICA achieve the best-in-class performance and capacity, their interface to applications is relatively simple among key-value storage systems. Both systems focus on single-key, unordered key-value operations; this interface is complete in that their main applications and workloads (e.g., memcached clients for MICA) can keep using the interface. Recall that many algorithms and data structures in SILT and MICA are specifically designed for this interface.

However, there do exist applications that use richer key-value storage semantics. They often require ordered retrieval (e.g., finding the smallest key larger than a given lookup key, retrieving all values within a key range) and/or transactions (e.g., committing or aborting multiple changes atomically, obtaining a consistent set of values for multiple keys).

Generally speaking, systems supporting complex features [65, 96, 112, 136] exhibit lower performance and capacity than SILT and MICA do. This may be simply an outcome of implementing richer semantics. For example, it is ineffective to use Bloom filters [22] for optimizing ordered retrieval because ordered retrieval may use non-existent lookup keys that are never stored in the system; accordingly, a system designer may have to employ more complex data structures such as ARFs [2]. On the other hand, it could be due to a suboptimal system design and configuration; we have shown that there exists room for improvement in SILT-like systems by optimizing their design and configuration with their system models built upon accurate and fast analytic primitives.

A key question is: *What fraction of performance and capacity losses in systems are attributable to the support for complex semantics?* Is it possible to quantify the effect of the support? Can we build a system that supports an "equivalent" interface enough to run target applications while minimizing performance and capacity losses? If we can modify applications while preserving their core functionalities, what extent can we improve this feature-capability tradeoff?

6.1.2 Balancing Generality and Specialization

In a broad sense, this dissertation exploits characteristics of target applications by employing specialized solutions. This approach implies that (1) there may exist applications that cannot be handled by the specialized solutions as we discuss above, and (2) specialization may involve more development and operation effort.

The second point can become particularly significant because a system designer may have to build a new system for each of numerous niche applications. For example, RocksDB [54] contains many configuration options to handle different applications—it can use typical LevelDB-like compaction, but it can also perform FIFO-style eviction of old data. If we develop multiple systems that are optimized to provide the maximum performance for different scenarios, the development cost may be prohibitively high considering the limited applicability of the individual systems. It would also require long development time, which will hinder rapid development and deployment of a new solution to the production environment.

One approach that can minimize the side effect of specialization is modularization. As explored early in Click [87] and Anvil [95], a modular framework can enable fast designing and implementation of specialized systems. They allows creating a full system by combining smaller components; e.g., Anvil builds a simple DBMS-like system by composing different implementations of the "dTable" interface, while these implementations can be reused to make another storage system.

However, this modular approach faces a new challenge nowadays. There exists a (possibly fundamental) tradeoff between the modularity and the system performance as demonstrated by the history of the operating system development. This tradeoff is significant as we use multi-core CPUs and design memory-speed storage systems. Multi-core systems have complex data flows with concurrency control, which makes it difficult to define a clean modular interface. Memory-speed storage systems hinders the conventional module composition using virtual functions or shared object files, which cause high function call overheads and prevent inlining across function calls.

In other words, we need *a new modular framework in the era of multi-core architectures and memory-speed applications.* How would such a framework differ from previous modular

frameworks? What is the role of language and compiler-supported approaches in solving this problem? Would it be still worthwhile specializing systems or should we fall back to generic systems for niche applications?

6.1.3 Easy Configuration of Systems without In-Depth Knowledge

It is always challenging to configure a complex system to attain its best performance. While we propose new analytic primitives to model SILT and SILT-like systems in this dissertation, they are insufficient to cover all the configuration issues in modern complex systems. Configuring a system appropriately still requires not only the in-depth knowledge of the system design and implementation, but also the effort to understand the target workloads by performing measurement and analysis on the workloads.

We claim that there should be *an easy configuration method that does not demand expertise in system internals and the operational environment.* Ideally, one should be able to configure a system automatically, while a guided optimization of a system may be almost equally effective in achieving the same goal. Combined with the previous discussion topic, one can imagine building and tuning a system without having to understand the function and behavior of individual system components and target workloads, and still obtaining high system capabilities that is close to what an expert can achieve.

Bibliography

[1] Amr Ahmed, Moahmed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and Alexander J. Smola. Scalable inference in latent variable models. In *Proceedings of the fifth ACM international conference on Web search and data mining*, February 2012. [Cited on pages 7, 79, and 80.]

[2] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson. Adaptive range filters for cold data: Avoiding trips to Siberia. *Proc. VLDB Endowment*, 6(14), September 2013. [Cited on page 106.]

[3] N. Alon, M. Dietzfelbinger, P.B. Miltersen, E. Petrank, and G. Tardos. Linear hash functions. *Journal of the ACM (JACM)*, 46(5):667–683, 1999. [Cited on pages 38 and 43.]

[4] Hrishikesh Amur, David G. Andersen, Michael Kaminsky, and Karsten Schwan. Design of a write-optimized data store. Technical Report GIT-CERCS-13-08, Georgia Tech CERCS, 2013. [Cited on pages 66 and 77.]

[5] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. Cheap and large CAMs for high performance data-intensive networked systems. In *Proc. 7th USENIX NSDI*, San Jose, CA, April 2010. [Cited on pages 9, 10, 12, 17, 33, and 85.]

[6] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009. [Cited on pages 5, 9, 12, 33, 34, 85, and 86.]

[7] A. Andersson and S. Nilsson. Improved behaviour of tries by adaptive branching. *Information Processing Letters*, 46(6):295–300, 1993. [Cited on page 18.]

[8] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the SIGMETRICS'12*, June 2012. [Cited on pages 4, 6, 7, 59, 76, 79, 80, and 81.]

[9] Anirudh Badam, KyoungSoo Park, Vivek S. Pai, and Larry L. Peterson. HashCache: cache storage for the next billion. In *Proc. 6th USENIX NSDI*, Boston, MA, April 2009. [Cited on pages 9, 12, and 33.]

[10] Mahesh Balakrishnan, Asim Kadav, Vijayan Prabhakaran, and Dahlia Malkhi. Differential raid: Rethinking raid for ssd reliability. In *In Proceedings of Fifth ACM European Conference on Computer Systems. (Eurosys '10)*, Paris, France, April 2010. [Cited on

page 27.]

[11] Basho Technologies. Basho LevelDB. https://github.com/basho/leveldb, 2015. [Cited on page 77.]

[12] Berkeley DB. http://www.oracle.com/technetwork/database/berkeleydb/, 2011. [Cited on page 34.]

[13] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *Proc. 9th USENIX OSDI*, Vancouver, Canada, October 2010. [Cited on pages 9 and 10.]

[14] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Theory and practise of monotone minimal perfect hashing. In *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments*, ALENEX '09. Society for Industrial and Applied Mathematics, 2009. [Cited on pages 33, 35, 43, 44, and 45.]

[15] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: searching a sorted table with O(1) accesses. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 785–794. Society for Industrial and Applied Mathematics, 2009. [Cited on pages 33, 35, and 44.]

[16] Djamal Belazzougui, Fabiano Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In *Proceedings of the 17th European Symposium on Algorithms*, ESA '09, pages 682–693, 2009. [Cited on pages 21, 33, and 43.]

[17] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming B-trees. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, 2007. [Cited on pages 53, 54, 55, 59, and 77.]

[18] Jon L. Bentley and James B. Saxe. Decomposable searching problems I: Static-to-dynamic transformations. *Journal of Algorithms*, 1(4):301–358, December 1980. [Cited on page 55.]

[19] Mateusz Berezecki, Eitan Frachtenberg, Mike Paleczny, and Kenneth Steele. Many-core key-value store. In *In Proceedings of the Second International Green Computing Conference*, Orlando, FL, August 2011. [Cited on pages 82, 86, and 103.]

[20] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. December 2014. http://arxiv.org/abs/1411.1607. [Cited on page 71.]

[21] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. An experimental analysis of a compact graph representation. In *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments*, ALENEX '04, pages 49–61, 2004. [Cited on page 21.]

[22] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970. [Cited on pages 17, 59, and 106.]

[23] Michaela Blott, Kimon Karras, Ling Liu, Kees Vissers, Jeremia Bär, and Zsolt István. Achieving 10Gbps line-rate key-value stores with FPGAs. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, June 2013. [Cited on pages 81 and 85.]

[24] Mark Bohr. A 30 year retrospective on Dennard's MOSFET scaling paper. *Solid-State Circuits Society Newsletter, IEEE*, 12(1):11–13, 2007. [Cited on page 5.]

[25] Fabiano C. Botelho and Nivio Ziviani. External perfect hashing for very large key sets. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, CIKM '07, pages 653–662. ACM, 2007. [Cited on pages xi, 35, 36, 38, 42, 43, and 45.]

[26] Fabiano C. Botelho, Anisio Lacerda, Guilherme Vale Menezes, and Nivio Ziviani. Minimal perfect hashing: A competitive method for indexing internal memory. *Information Sciences*, 2009. [Cited on page 21.]

[27] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Practical perfect hashing in nearly optimal space. *Information Systems*, 2013. [Cited on pages 36 and 44.]

[28] Gerth Stolting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2003. [Cited on page 77.]

[29] Mark Callaghan. Read Amplification Factor. http://mysqlha.blogspot.com/2011/08/read-amplification-factor.html, 2011. [Cited on page 57.]

[30] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991. [Cited on page 5.]

[31] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. 7th USENIX OSDI*, Seattle, WA, November 2006. [Cited on pages 1, 3, 6, 12, 34, and 53.]

[32] Li-Pin Chang. On efficient wear leveling for large-scale flash-memory. storage systems. In *SAC '07 Proceedings of the 2007 ACM symposium on Applied computing*, 2007. [Cited on page 26.]

[33] Bernard Chazelle and LeonidasJ. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(1-4):133–162, 1986. [Cited on page 55.]

[34] CityHash. http://code.google.com/p/cityhash/, 2013. [Cited on page 88.]

[35] David Richard Clark. *Compact PAT trees*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 1998. [Cited on pages 20 and 39.]

[36] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *Proc. 24th*

ACM Symposium on Operating Systems Principles (SOSP), Farmington, PA, November 2013. [Cited on page 63.]

[37] Yann Collet. LZ4. https://github.com/Cyan4973/lz4, 2015. [Cited on page 62.]

[38] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. 1st ACM Symposium on Cloud Computing (SOCC)*, Indianapolis, IN, June 2010. [Cited on pages 28, 59, 68, 86, and 95.]

[39] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, October 2001. [Cited on page 10.]

[40] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013. [Cited on page 5.]

[41] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56 (2):74–80, February 2013. [Cited on pages 4 and 80.]

[42] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: high throughput persistent key-value store. *Proc. VLDB Endow.*, 3:1414–1425, September 2010. [Cited on pages 9, 10, 12, 15, and 33.]

[43] Biplob Debnath, Sudipta Sengupta, and Jin Li. SkimpyStash: RAM space skimpy key-value store on flash. In *Proc. ACM SIGMOD*, Athens, Greece, June 2011. [Cited on pages 9, 10, 12, and 33.]

[44] Guiseppe DeCandia, Deinz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, October 2007. [Cited on pages 3, 6, 7, 9, 11, 23, 34, and 86.]

[45] R.H. Dennard, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9 (5):256–268, October 1974. [Cited on page 4.]

[46] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009. [Cited on pages 6, 80, 81, 83, 87, 88, and 95.]

[47] Jirun Dong and Richard Hull. Applying approximate order dependency to reduce indexing space. In *Proceedings of the 1982 ACM SIGMOD international conference on Management of data*, SIGMOD '82, pages 119–127, New York, NY, USA, 1982. ACM. ISBN 0-89791-073-7. [Cited on page 21.]

[48] Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, March 1975. [Cited on pages 20 and 40.]

[49] U. Erlingsson, M. Manasse, and F. Mcsherry. A cool and practical alternative to traditional hash tables. In *Proc. Seventh Workshop on Distributed Data and Structures (WDAS'06)*, CA, USA, January 2006. [Cited on page 15.]

[50] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A distributed, searchable key-value store. In *Proc. ACM SIGCOMM*, Helsinki, Finland, August 2012. [Cited on page 104.]

[51] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011. [Cited on page 5.]

[52] Facebook. Performance Benchmarks. https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks, 2014. [Cited on page 75.]

[53] Facebook. http://www.facebook.com/, 2015. [Cited on page 10.]

[54] Facebook. RocksDB. http://rocksdb.org/, 2015. [Cited on pages 53, 57, 58, 75, 77, and 106.]

[55] Facebook. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide, 2015. [Cited on pages 68 and 75.]

[56] Facebook's memcached multiget hole: More machines != more capacity. http://highscalability.com/blog/2009/10/26/facebooks-memcached-multiget-hole-more-machines-more-capacit.html, 2009. [Cited on page 81.]

[57] Facebook Stats. https://newsroom.fb.com/company-info/, 2015. [Cited on pages 1 and 3.]

[58] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proc. 10th USENIX NSDI*, Lombard, IL, April 2013. [Cited on pages 16, 79, 81, 82, 83, 84, 85, 91, 92, 93, 94, 99, 102, and 103.]

[59] Philippe Flajolet, Danièle Gardy, and Loÿs Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics*, 39(3), November 1992. [Cited on page 60.]

[60] Mario Flajslik and Mendel Rosenblum. Network interface design for low latency request-response protocols. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, June 2013. [Cited on page 103.]

[61] Flickr. http://www.flickr.com/, 2011. [Cited on page 10.]

[62] Edward A. Fox, Lenwood S. Heath, Qi Fan Chen, and Amjad M. Daoud. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35:105–121, January 1992. [Cited on page 33.]

[63] Fusion-io ioDrive Octal. https://www.fusionio.com/platforms/iodrive-octal/, 2012. [Cited on page 35.]

[64] Jean-Loup Gailly and Mark Adler. zlib. http://www.zlib.net/, 2013. [Cited on page 62.]

[65] Google. LevelDB. https://github.com/google/leveldb, 2014. [Cited on pages 34, 53, 54, 55, 56, 58, 59, 77, and 106.]

[66] Google. LevelDB file layout and compactions. https://github.com/google/leveldb/blob/master/doc/impl.html, 2014. [Cited on page 67.]

[67] Google. Snappy. https://github.com/google/snappy, 2015. [Cited on page 62.]

[68] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, May 1994. [Cited on page 95.]

[69] Roberto Grossi and Giuseppe Ottaviano. Fast compressed tries through path decompositions. In *Proceedings of the Annual Meeting on Algorithm Engineering and Experiments*, ALENEX '12. Society for Industrial and Applied Mathematics, 2012. [Cited on page 44.]

[70] James Hamilton. The cost of latency. http://perspectives.mvdirona.com/2009/10/the-cost-of-latency/, 2009. [Cited on page 4.]

[71] Dongsu Han, Ashok Anand, Fahad Dogar, Boyan Li, Hyeontaek Lim, Michel Machado, Arvind Mukundan, Wenfei Wu, Aditya Akella, David G. Andersen, John W. Byers, Srinivasan Seshan, and Peter Steenkiste. XIA: Efficient support for evolvable internetworking. In *Proc. 9th USENIX NSDI*, San Jose, CA, April 2012. [Cited on page 96.]

[72] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-accelerated software router. In *Proc. ACM SIGCOMM*, New Delhi, India, August 2010. [Cited on pages 80, 81, 83, 87, 88, and 95.]

[73] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: a new programming interface for scalable network I/O. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, Hollywood, CA, October 2012. [Cited on pages 81 and 83.]

[74] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989. [Cited on page 85.]

[75] HyperDex. HyperLevelDB. http://hyperdex.org/performance/leveldb/, 2013. [Cited on pages 64 and 77.]

[76] Intel. Intel Data Plane Development Kit (Intel DPDK). http://www.intel.com/go/dpdk, 2013. [Cited on pages 81, 87, and 94.]

[77] Intel 82599 10 Gigabit Ethernet Controller: Datasheet. http://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html, 2013. [Cited on pages 84 and 88.]

[78] Intel Advanced Encryption Standard Instructions (AES-NI). https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni/, 2012. [Cited on page 5.]

[79] Intel Solid-State Drive 510 Series. http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-510-series.html, 2012. [Cited on page 35.]

[80] Intel SSE4 programming reference. https://software.intel.com/sites/default/files/m/8/b/8/D9156103.pdf, 2007. [Cited on page 5.]

[81] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. BetrFS: A right-optimized write-optimized file system. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, Santa Clara, CA, February 2015. [Cited on page 77.]

[82] Myeongjae Jeon, Saehoon Kim, Seung-won Hwang, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. Predictive parallelization: Taming tail latencies in web search. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, 2014. [Cited on page 4.]

[83] EunYoung Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *Proc. 11th USENIX NSDI*, Seattle, WA, April 2014. [Cited on page 103.]

[84] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 2011. [Cited on page 54.]

[85] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, October 2012. [Cited on pages 82, 86, and 103.]

[86] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., 1997. First edition published in 1968. [Cited on page 93.]

[87] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000. [Cited on page 106.]

[88] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating System Review*, 44:35–40, April 2010. [Cited on pages 53 and 56.]

[89] Doug Lea. A memory allocator. http://g.oswego.edu/dl/html/malloc.html, 2000. [Cited on page 93.]

[90] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6), June 1983. [Cited on page 54.]

[91] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, October 2011. [Cited on pages vi, 48, 57, 58, 77, and 85.]

[92] Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Practical batch-updatable external hashing with sorting. In *Proc. Meeting on Algorithm Engineering and Experiments (ALENEX)*, January 2013. [Cited on page vi.]

[93] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proc. 11th USENIX NSDI*, Seattle, WA, April 2014. [Cited on page vi.]

[94] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. Thin servers with Smart Pipes: Designing SoC accelerators for Memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, June 2013. [Cited on page 81.]

[95] Mike Mammarella, Shant Hovsepian, and Eddie Kohler. Modular data storage with Anvil. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009. [Cited on pages 12, 27, 34, and 106.]

[96] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proc. 7th ACM European Conference on Computer Systems (EuroSys)*, Bern, Switzerland, April 2012. [Cited on pages 79, 81, 82, 83, 85, 92, 94, 101, 102, 104, and 106.]

[97] Sally A. McKee. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers*, 2004. [Cited on page 5.]

[98] Mellanox ConnectX-3 product brief. `http://www.mellanox.com/related-docs/prod_adapter_cards/ConnectX3_EN_Card.pdf`, 2013. [Cited on pages 6 and 84.]

[99] A distributed memory object caching system. `http://memcached.org/`, 2011. [Cited on pages 1, 9, 79, 89, 93, 94, 102, and 103.]

[100] Prashanth Menon, Tilmann Rabl, Mohammad Sadoghi, and Hans-Arno Jacobsen. Optimizing key-value stores for hybrid storage architectures. In *Proceedings of CASCON*, 2014. [Cited on page 77.]

[101] Zviad Metreveli, Nickolai Zeldovich, and M. Frans Kaashoek. CPHash: a cache-partitioned hash table. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2012. [Cited on pages 81, 82, 83, 84, 85, 86, 91, and 103.]

[102] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, July 2002. [Cited on page 82.]

[103] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proceedings of the 2013 conference on USENIX Annual technical conference*, June 2013. [Cited on pages 81, 99, and 103.]

[104] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 1965. [Cited on page 4.]

[105] Peter Muth, Patrick O'Neil, Achim Pick, and Gerhard Weikum. The LHAM log-structured history data access method. *The VLDB Journal*, 8(3-4):199–221, February 2000. [Cited on page 77.]

[106] Suman Nath and Phillip B. Gibbons. Online maintenance of very large random samples on flash storage. In *Proc. VLDB*, Auckland, New Zealand, August 2008. [Cited on page 21.]

[107] Suman Nath and Aman Kansal. FlashDB: Dynamic self-tuning database for NAND flash. In *Proceedings of ACM/IEEE International Conference on Information Processing in Sensor Networks*, Cambridge, MA, April 2007. [Cited on pages 9 and 33.]

[108] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proc. 10th USENIX NSDI*, Lombard, IL, April 2013. [Cited on pages 1, 3, 7, 79, 80, 81, 82, 83, 84, 85, 87, and 103.]

[109] C. Nyberg and C. Koester. Ordinal Technology - Nsort home page. http://www.ordinal.com, 2012. [Cited on pages 17, 28, 38, and 46.]

[110] Patrick E. O'Neil. The SB-tree: An index-sequential structure for high-performance sequential access. *Acta Inf.*, 29(3):241–265, 1992. [Cited on page 55.]

[111] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, 1996. [Cited on pages 53, 54, 55, and 77.]

[112] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, October 2011. [Cited on pages 79, 80, 85, 94, 102, 104, and 106.]

[113] Optimized approximative pow() in C / C++. http://martin.ankerl.com/2012/01/25/optimized-approximative-pow-in-c-and-cpp/, 2013. [Cited on page 95.]

[114] R. Pagh and F.F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, May 2004. [Cited on pages 13, 14, 33, and 94.]

[115] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD '12: Proceedings of the 2012 international conference on Management of Data*, May 2012. [Cited on page 103.]

[116] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM european conference on Computer Systems*, April 2012. [Cited on pages 6, 81, 83, 87, 88, and 103.]

[117] Milo Polte, Jiri Simsa, and Garth Gibson. Enabling enterprise solid state disks performance. In *Proc. Workshop on Integrating Solid-state Memory into the Storage Hierarchy*, Washington, DC, March 2009. [Cited on page 21.]

[118] P.W. Purdom, S.M. Stigler, and Tat-Ong Cheam. Statistical investigation of three storage allocation algorithms. *BIT Numerical Mathematics*, 11(2), 1971. [Cited on pages 85 and 93.]

[119] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, pages 89–101, Monterey, CA, January 2002. [Cited on page 10.]

[120] Martin Raab and Angelika Steger. "Balls into bins" — a simple and tight analysis. In *Proceedings of the Second International Workshop on Randomization and Approximation Techniques in Computer Science*, RANDOM '98, pages 159–170, London, UK, 1998. Springer-Verlag. [Cited on page 43.]

[121] RAMCloud Project wiki: clusterperf November 12, 2012. https://ramcloud.stanford.edu/wiki/display/ramcloud/clusterperf+November+12%2C+2012, 2012. [Cited on page 103.]

[122] Luigi Rizzo. netmap: a novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, June 2012. [Cited on page 81.]

[123] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992. [Cited on page 53.]

[124] Trausti Saemundsson, Hjortur Bjornsson, Gregory Chockler, and Ymir Vigfusson. Dynamic performance profiling of cloud caches. In *Proc. 5th ACM Symposium on Cloud Computing (SOCC)*, Seattle, WA, November 2014. [Cited on page 59.]

[125] Russell Sears and Raghu Ramakrishnan. bLSM: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012. [Cited on page 77.]

[126] Russell Sears, Mark Callaghan, and Eric Brewer. Rose: Compressed, log-structured replication. *Proc. VLDB Endowment*, 1(1), August 2008. [Cited on page 77.]

[127] Pradeep Shetty, Richard Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building workload-independent storage with VT-trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, 2013. [Cited on pages 66 and 77.]

[128] Richard P. Spillane, Pradeep J. Shetty, Erez Zadok, Sagar Dixit, and Shrikar Archak. An efficient multi-tier tablet server storage architecture. In *Proc. 2nd ACM Symposium on*

Cloud Computing (SOCC), Cascais, Portugal, October 2011. [Cited on pages 53, 54, 56, and 77.]

[129] Phillip Stanley-Marbell, Victoria Caparros Cabezas, and Ronald Luijten. Pinned to the walls: Impact of packaging and application properties on the memory and power walls. In *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design*, ISLPED '11, 2011. [Cited on page 5.]

[130] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM*, San Diego, CA, August 2001. [Cited on page 14.]

[131] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era (it's time for a complete rewrite). In *Proc. VLDB*, Vienna, Austria, September 2007. [Cited on page 103.]

[132] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. http://www.gotw.ca/publications/concurrency-ddj.htm, 2005. [Cited on page 5.]

[133] Sort benchmark home page. http://sortbenchmark.org/, 2012. [Cited on page 38.]

[134] The Apache Software Foundation. Apache Cassandra. https://cassandra.apache.org/, 2015. [Cited on page 56.]

[135] The Apache Software Foundation. Apache HBase. https://hbase.apache.org/, 2015. [Cited on pages 53 and 56.]

[136] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proc. 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, November 2013. [Cited on pages 81, 82, 83, 102, 104, and 106.]

[137] Twitter. http://www.twitter.com, 2011. [Cited on page 10.]

[138] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. LogBase: A scalable log-structured database system in the cloud. *Proc. VLDB Endowment*, 5(10), June 2012. [Cited on page 77.]

[139] VoltDB, the NewSQL database for high velocity applications. http://voltdb.com/, 2014. [Cited on page 103.]

[140] Andreas Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006. [Cited on page 71.]

[141] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems*, 2014. [Cited on page 77.]

[142] WEBSPAM-UK2007. http://barcelona.research.yahoo.net/webspam/datasets/uk2007/links/, 2007. [Cited on page 45.]

[143] P.R. Wilson, M.S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. *Lecture Notes in Computer Science*, 1995. [Cited on pages 23, 85, and 93.]

[144] WiredTiger. WiredTiger. http://www.wiredtiger.com/, 2014. [Cited on page 77.]

[145] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, and Andrew Warfield. Characterizing storage workloads with counter stacks. In *Proc. 11th USENIX OSDI*, Broomfield, CO, October 2014. [Cited on page 59.]

[146] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1), March 1995. [Cited on page 5.]

[147] Demetrios Zeinalipour-Yazti, Song Lin, Vana Kalogeraki, Dimitrios Gunopulos, and Walid A. Najjar. MicroHash: An efficient index structure for flash-based sensor devices. In *Proc. 4th USENIX Conference on File and Storage Technologies*, San Francisco, CA, December 2005. [Cited on pages 9 and 33.]

[148] Dong Zhou, Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proc. 9th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, December 2013. [Cited on pages 86 and 92.]