Systems and Algorithms for High-Performance, Cost-Efficient Key-Value Storage

Xiaozhou Li

A Dissertation

Presented to the Faculty

of Princeton University

in Candidacy for the Degree

of Doctor of Philosophy

Recommended for Acceptance

by the Department of

Computer Science

Adviser: Michael J. Freedman

June 2016

Abstract

Key-value storage systems are increasingly essential building blocks of modern cloud and big data applications. The workloads these systems support often require random access to small objects over massive datasets with highly skewed and dynamic key popularity. It is challenging for a storage cluster to serve these workloads with both high performance and low-cost operations. Today's systems usually sacrifice one for the other. In this dissertation, we present novel approaches to improve both the performance and cost-efficiency of key-value systems by combining new hardware and software techniques with careful architectural design and algorithmic optimizations.

First, at cluster scale, we build SwitchKV, a heterogeneous system that uses small high-end cache nodes to guarantee load balancing across many SSD-based backend nodes under nearly-arbitrary workloads. The cache nodes absorb the hottest queries so that no individual backend node is over-burdened or underutilized. The system exploits OpenFlow switches to enable efficient content-aware routing so that it can achieve scalable high throughput, low tail latency, and high availability. It uses new algorithms to keep the cache and switch forwarding rules updated with low overhead, and to ensure stable high performance under rapidly changing workloads. SwitchKV can meet the service level objectives for many cloud services more efficiently than traditional systems.

Second, to improve the efficiency of each individual multi-core server, we build a high-throughput and memory-efficient concurrent hash table based around optimistic cuckoo hashing. Our re-design minimizes critical section length, reduces interprocessor coherence traffic, and enables effective prefetching through careful algorithm and data structure engineering. We explore hardware transactional memory and fine-grained locking for concurrency control, and find that both of them require the same level of algorithmic efforts to achieve high performance. Our new hash table design greatly outperforms other optimized concurrent hash tables for both read- and write-heavy workloads, even while using substantially less memory for small key-value items.

Acknowledgements

The past few years of my life have been an incredible journey. I consider myself extremely fortunate to have been accompanied by my fantastic advisors, collaborators, friends and family. I wish to express my most sincere gratitude and appreciation to everyone who helped and supported me during my time at graduate school.

I am deeply indebted to my adviser, Mike Freedman, for his invaluable guidance and support. Mike is always sharp, passionate and inspiring. He motivated me to pursue system research, encouraged me to explore different paths, and guided me to solve hard but interesting problems. He provides a role model for me to follow, not only in research, but also in life outside of work. Without Mike, I would not be even close to my current state as a seasoned researcher and software engineer who enjoy my everyday life.

My dissertation research was done in collaboration with my adviser, Dave Andersen, Michael Kaminsky, and Raghav Sethi. Dave and Michael also served as my unofficial secondary advisers. They taught me a lot in every aspect of being a successful researcher, from finding the right problems and challenging every assumptions, to writing good papers and making impressive presentations. Raghav made significant contributions to the SwitchKV project. He is a proficient hacker and an ideal collaborator.

I thank the other members of my committee—Kyle Jamieson, Kai Li, and Jennifer Rexford—for their helpful comments on improving my dissertation. I would like to especially recognize Jen as a great mentor who often provide insightful advice on research, graduate life and career directions.

I owe a tremendous amount to Boon Thau Loo, who advised me at the University of Pennsylvania. I would not have been in Princeton without his guidance and help.

Many thanks go to my fellow graduate students and postdocs in the SNS group for creating such a friendly and warn environment, especially to Haoyu Zhang, Aaron Blankstein, Annie Liu, Amy Tai, David Shue, Wyatt Lloyd, Siddhartha Sen, Jeff Terrace, Raghav Sethi, Matvey Arye and Rob Kiefer.

My time at Princeton has been immensely enjoyable, thanks in large part to the good friends I have made here, including Xin Jin, Linpeng Tang, Feng Liu, Yida Wang, Peng Sun, Nanxi Kang, Yichen Chen, Xuan Zou, Tianqiang Liu, Yiming Liu, Tianlong Wang, Xinyi Fan, Minlan Yu, Jidong Chen, Wei Wang, and all the SNS members.

The research in this dissertation was supported by National Science Foundation Awards CSR-0953197 (CAREER) and CCF-0964474, and by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC). I am also grateful to the university and the Siebel Scholars Foundation for providing generous fellowships.

Most of all, I would like to thank my wonderful family. To my parents Wanxia Liu and Zuocheng Li, thank you for your unconditional love and dedication all the time. None of my accomplishments would be possible without you. To my parents-in-law Runqiu Guo and Yue Hao, thank you for your unwavering support and faith in me. Finally, to my beloved wife Rui Hao, thank you for being my partner in life and always standing by me no matter what. Our journey continues on.

To my family.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Key-value storage systems provide operations to store and retrieve data (values) identified by unique names (keys). Their simple APIs (e.g., `get`, `put`, `delete`) form a fundamental building block of modern large-scale cloud services such as Google (BigTable [11]), Facebook (Memcached [59]), Amazon (Dynamo [21]), LinkedIn (Voldemort [75]), and many others (Cassandra [10], Redis [68]).[1] They are also critical components of big data analytics such distributed machine learning (parameter server [49]).

These systems usually seek high performance. Ensuring high throughput and low tail latency is important because many higher-level applications rely on the key-value storage layer to achieve reliable high service quality. It is equally important to keep the system cost-efficient because it determines whether the service is able to easily scale its performance and capacity as the data keeps growing at increasingly accelerated rates.

Though key-value storage systems have been extensively studied and optimized by the research community and industry, with various focuses on performance [55, 63], availability [10, 59], consistency [26, 71], cost [4, 54], it is still challenging to build systems that can provide both *high performance* and *low cost* operations to serve the workloads of today's cloud and big data applications, which share similar characteristics:

[1] Some of these stores support more complex operations such as transactions and range queries. This dissertation only focuses on supporting simple `get`, `put` and `delete` queries, which are the most essential operations of key-value storage.

- *Parallel random access to small objects over large datasets* [4, 6, 59]. While the data volumes of many systems are reaching to terabytes or petabytes and beyond, the size of each object is typically small, mostly within tens to hundreds of bytes. These objects are accessed randomly through many concurrent, mostly-independent client requests.

- *Highly skewed and dynamic key popularity* [6, 9, 16, 28]. Many workloads have power-law access frequency distribution, where a small portion of keys account for the majority of the queries. These hot spots often change quickly. Many applications further experience unpredictable flash crowds or adversarial access patterns.

These highly I/O intensive, concurrent, skewed, and fast-changing workloads pose great challenges to both scaling out the system capacity and performance with large clusters, and improving the efficiency of each individual storage server. Most current systems offer high performance or low cost, but not both. Some systems use DRAM aggressively in order to meet the high performance goals, and substantially over-provision the resources to handle the dynamic and unpredictable workload skew. They are often expensive and power-hungry. Other systems choose to use a large cluster of low-power servers in order to reduce the cost. These clusters usually have poor performance and face severe load balancing problems, which leads to most servers being either over- or underutilized.

The goal of this dissertation research is to build systems that are cheaper and more power-efficient than conventional architectures, while providing the same or even better performance under real-world workloads. This requires the systems to use hardware resources efficiently, both in each individual server and as the cluster scales out. To achieve this goal, our key-value storage systems face three major challenges:

1. How to ensure dynamic load balancing as the cluster scales out, so that the system can fully utilize all the storage servers regardless of the workload distribution?

2. How to use different types of memory (e.g., DRAM, flash) efficiently, so that the system can meet the performance and capacity requirements with minimal cost?

3. How to achieve maximum parallelism in multi-core systems, so that each individual server can fully utilize the processing power of modern CPUs?

We first explain and analyze these main challenges in the next section, and then address these challenges in the rest of this dissertation with our careful system design and algorithmic optimizations.

1.1 Current Approaches and Challenges

In this section, we summarize the approaches and problems of current system solutions according to the challenges described above—load balancing, memory efficiency, and parallel access. We will discuss the principles and directions to solve these problems and meet these challenges in Section 1.2.

1.1.1 Key-Value Cluster and Load Balancing

Cloud service providers are building increasingly large storage clusters to support their rapidly growing active key-value datasets. For example, Apple runs over 75,000 Cassandra nodes storing over 10 PB of data, with a single cluster spanning over 1,000 nodes [67]; Google runs BigTable and GFS with with 1000 to 7000 nodes in each storage cell [30]; Facebook runs Memcached as their in-memory caching solution, with over thousands of servers within a cluster [59].[2] These systems scale out by partitioning data across cluster nodes, where each node handles only a subset of the data set; these designs usually achieve high availability against node or network failures by replicating each piece of data over multiple nodes [11, 21, 48].

[2]In this dissertation, we use the term node and server indistinguishably. Each storage server is considered a node in the cluster.

Key-value workloads for cloud applications are often skewed and rapidly changing. For example, workload analysis of Facebook's Memcached cluster shows that 50% of keys occur in only 1% of all requests, while 5% of keys account for 80% of requests [6]. Yahoo's YCSB framework models many workloads as Zipfan distribution [16]. Prior study shows that Internet-scale data-intensive sites often experience "data spikes": a sudden increase in demand for certain objects, or more generally, a pronounced change in the distribution of object popularity on the site [9].

As a result, dynamic load balancing becomes a key challenge as the storage cluster scales out, because the system service quality is often bottlenecked by the performance of the overloaded (slowest) node. For example, a web server may need to contact 10s to 100s of storage nodes with many sequential queries when responding to a single page request [59], and the tail latency of the queries can significantly degrade the service performance [19]. The system aggregate throughput would also be severely affected by the overloaded nodes. Therefore, service providers often choose to provision each storage node based on peak load [59], even though at most times the actual load is far below it. This greatly increases the system cost.

Good load balancing is necessary to ensure that the cluster can meet its performance goals without substantial over-provisioning. Consistent hashing [44] and virtual nodes [18] are popular and effective techniques to balance the static load and space utilization, but are unable to balance the dynamic load with skewed query distributions. Traditional dynamic load balancing methods often migrate data between nodes [13, 45, 70]. They are limited in their ability to deal with large skew, are usually too slow to handle rapid workload changes, and often introduce consistency challenges and system overhead for migration or replication.

Frontend caching can provide effective backend load balancing [5, 28], because the cache can directly serve and filter out the hot spots so that the load across the backends can be much more uniform. Prior research further proved that the size of the cache required

to provide good load balance is only $O(n \log n)$, where n is the total number of backend nodes [28]. This means that a large cluster can have guaranteed load balance by only using one or a few small high-end cache servers. However, this is a theorem without practical realizations. It introduces new challenges to the systems because the cache is small and the hit ratio could be low, which breaks the assumptions of existing caching architectures. It is difficult to keep the small cache updated with low overhead and serve queries for the uncached keys efficiently. One of the main contributions of this dissertation is to address these challenges with new system designs and algorithms, and build a highly efficient load-balanced key-value storage cluster based on this theoretical result.

1.1.2 Memory-Efficiency for Big Data Storage

In pursuit of meeting aggressive latency and throughput performance goals for Internet services, providers have increasingly turned to in-memory [59, 63, 68] or flash-based [4, 69] key-value storage systems as caches or primary data stores. These systems can offer microseconds of latency and provide throughput hundreds to thousands of times that of the hard-disk-based approaches of yesteryear.

The choice of flash vs DRAM comes with important differences in throughput, latency, persistence, and cost-per-gigabyte. Given the performance requirements, some systems keep data entirely in memory [63, 68], with disks used only for failure recovery; others put a significant fraction of data in cache to achieve high hit ratio [59]. Systems that aggressively use DRAM are both expensive and consume a surprising amount of power. Google once reported that DRAM accounts for 30% of the power usage in its data centers [7], and the power draw of storage clusters is becoming an increasing fraction of their cost—up to 50% of the three-year total cost of owning a cluster [4].

Meanwhile, SSDs are 10x cheaper and 100x more power-efficient than DRAM [4]. Recent advances in SSD performance, including new hardware technologies such as NVMe [61] and optimized software stacks such as RocksDB [69], are opening up new

points in the design space of storage systems that were formerly the exclusive domain of DRAM-based systems. It is now possible to build SSD-based key-value storage clusters that can meet the performance goals of many cloud services. This would lead to significant cost savings, especially for services that require the data to be stored persistently. A key contribution of this dissertation is to build such a cluster with performance optimizations in each SSD-based storage node and efficient techniques to ensure load balancing across many nodes as the cluster grows.

Of course, SSDs cannot replace DRAM everywhere, as their performance gap is still significant. Many applications use DRAM to store all their data or cache the hot objects in order to meet their high performance goals. Flash-based storage applications often use DRAM to index the massive data stored in SSDs. Yet as the growth of DRAM capacity is slower than the growth of big data volume and flash storage capacity [54], it is increasingly important to have memory-efficient key-value data structures such as hash tables. This is especially challenging for small objects, because any metadata per object, including pointers, may lead to huge waste of memory. Another key contribution of this dissertation is to explore the memory-efficient cuckoo hash table [64, 27], and optimize its performance for both read- and write-intensive workloads.

1.1.3 Parallelism in Multi-Core Systems

In addition to using DRAM and flash efficiently, another critical aspect of improving the performance and efficiency of each individual sever is to fully utilize each modern multicore system. As we scale systems on a storage server to greater numbers of cores and threads, the ability to exploit the maximum parallelism enabled by these cores becomes increasingly important. There are two different access models: exclusive access and concurrent access.

Exclusive access has been well-studied and proven to be very efficient for networked key-value storage [8, 43, 55]. By partitioning the data, each core can exclusively access its own partition in parallel, without worrying about any inter-core communication or lock synchronization. Modern network interfaces and drivers such as Intel DPDK [40] allow the NIC to deliver incoming packets to the specific cores based on the packet header fields, so that the query packets can be directly sent to the right cores for processing.

One concern of this approach is that the performance would degrade if the load across partitions is imbalanced [27, 56, 74]. This can be solved by exploiting CPU caches and packet burst I/O to eliminate the penalty from skewed workloads [55], or applying load balancing techniques to make the load across cores more uniform. The key-value cluster proposed in this dissertation explores both techniques for its cache and storage servers.

Concurrent access is also used by many key-value systems [20, 27, 56, 59], where all cores can access all the data. It supports more generic abstractions and has wider usage. However, it requires careful data structure design and algorithm engineering to achieve high concurrency and make sure everything behaves correctly.

This dissertation studies the most widely used data structure—hash tables. It focuses on cuckoo hashing, an open-addressing hash table that can be extremely memory-efficient for small key-value objects [27, 64]. The previous state-of-art implementation of cuckoo hash table can support concurrent read operations efficiently, but has to serialize all the write operations. As a result, the performance degrades quickly as the write ratio increases. Extensive algorithmic optimizations are needed in order to support fast concurrent writes.

1.2 Contributions

The goals of our research are 1) to achieve *high performance with low memory overhead* on each multi-core server, and 2) to keep *high performance without substantial overprovisioning under widely varying workloads* on a cluster with many storage servers. These

goals are deeply coupled with the challenges discussed above. Meeting the first goal requires efficient usage of DRAM and flash storage, and achieving maximum parallelism in multicore systems. Meeting the second goal needs an efficient dynamic load balancing mechanism to ensure that each cluster node can be fully utilized under any workloads.

Fortunately, new hardware technologies such as OpenFlow switches [62] and hardware transactional memory [77] offer us new opportunities to meet the challenges. However, through careful system design and engineering, we find that naively applying these hardware techniques introduces more problems than they solve. Both architectural and algorithmic optimizations are necessary to realize the hardware benefits and maximize the system performance and cost-efficiency.

With deep understanding of current system challenges, existing solutions, and emerging technical trends, our dissertation research improves the performance and efficiency of key-value storage systems by combining new hardware and infrastructure capabilities with carefully-crafted algorithmic techniques. We make the following two major contributions:

- *At cluster scale, we build SwitchKV, a fast, scalable and efficiently load-balanced SSD-based key-value system for widely varying real-world workloads.* [52]

  SwitchKV combines high-performance cache nodes with resource-constrained SSD-based backend nodes to provide load balancing in the face of unpredictable workload skew. The cache nodes absorb the hottest queries so that no individual backend node is over-burdened. Compared with previous designs, SwitchKV exploits SDN techniques and deeply optimized switch hardware to enable efficient content-based routing. Programmable network switches keep track of cached keys and route requests to the appropriate nodes at line speed, based on keys encoded in packet headers. A new hybrid caching strategy keeps cache and switch forwarding rules updated with low overhead and ensures that system load is always well-balanced under rapidly changing workloads. SwitchKV can meet the performance goals for many cloud ser-

vices efficiently by being able to fully utilize the capacity of all SSD-based backend servers under nearly-arbitrary workloads.

- *For each individual multi-core server, we build a fast and memory-efficient cuckoo hash table for both highly concurrent read- and write-intensive workloads.* [51]

  We present the design, implementation, and evaluation of a high-throughput and memory-efficient concurrent hash table that supports multiple readers and writers. The design arises from careful attention to systems-level optimizations such as minimizing critical section length and reducing interprocessor coherence traffic through algorithm re-engineering. As part of the architectural basis for this engineering, we include a discussion of our experience and results adopting Intel's recent hardware transactional memory (HTM) support to this critical building block. We find that naively allowing concurrent access using a coarse-grained lock on existing data structures reduces overall performance with more threads. While HTM mitigates this slowdown somewhat, it does not eliminate it. Algorithmic optimizations that benefit both HTM and designs for fine-grained locking are needed to achieve high performance. Our performance results demonstrate that our new hash table design—based around optimistic cuckoo hashing—outperforms other optimized concurrent hash tables by up to 2.5x for write-heavy workloads, even while using substantially less memory for small key-value items.

Taken together, SwitchKV and concurrent cuckoo hashing provide in-depth exploration of building highly-efficient key-value systems to meet the high performance goals for real-world applications at different scale: scalable clusters and many-core servers. Chapters 2 and 3 describe these two contributions in detail respectively. Chapter 4 summarizes our research and discusses possible directions for future work.

Chapter 2

SwitchKV

SwitchKV is a new cluster-level key-value store architecture that can achieve high efficiency under widely varying and rapidly changing workloads. SwitchKV uses a mix of server classes, where specially-configured high-performance nodes serve as fast, small in-memory caches for data that is hash partitioned across resource-constrained SSD-based backend nodes. The cache absorbs the hottest queries and ensure the load across the backends well-balanced [28].

At the heart of SwitchKV's design is an efficient content-based routing mechanism that uses software-defined networking (SDN) techniques to serve requests with minimal overhead. Clients encode keys into packet headers and send the requests to OpenFlow switches. These switches maintain forwarding rules for all cached items, and route requests directly to the cache or backend nodes as appropriate based on the embedded key information.

SwitchKV achieves high performance by *moving the cache out of the data path* and by exploiting switch hardware that has already been optimized to match (on query keys) and forward traffic to the right node at line rate with low latency. All responses return within one round-trip, and there is no overhead for the significant volume of queries for keys that are not in the cache. SwitchKV can scale-out by adding more cache nodes and switches, and is resilient to cache crashes.

The benefits of using OpenFlow switches come at a price: the update rate of forwarding rules in hardware is much lower than that of in-memory caches. Our solution includes an efficient hybrid cache update mechanism that minimizes the cache churn, while still reacting quickly to rapid changes in key popularity. Backends send periodic reports to the cache nodes about their recent hot keys as well as instant reports about keys that suddenly become very popular. Cache nodes maintain query statistics for the cached keys, add or evict the appropriate keys when they receive reports, and instruct SDN controllers to update switch forwarding rules accordingly.

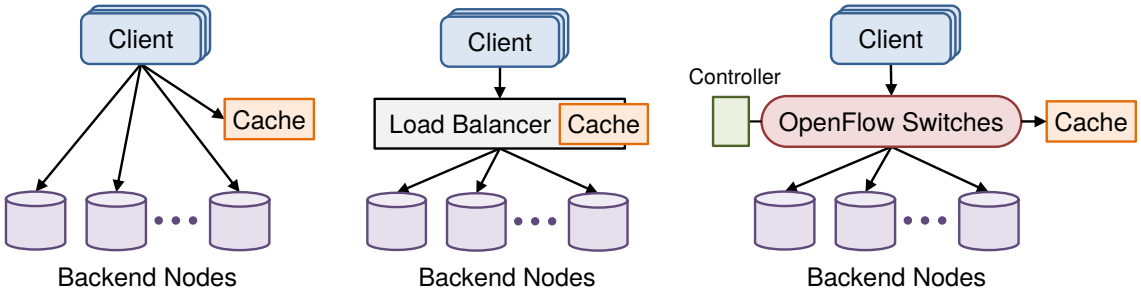
Our evaluation and analysis shows that SwitchKV can handle the traffic characteristics of modern cloud applications more efficiently than conventional systems, and can achieve up to 5× throughput and 3× latency improvements over traditional caching architectures.

Our SwitchKV prototype uses low-power backend nodes. The same design principles and evaluation results also apply to clusters with more powerful backends, by using high-end cache servers [50] that can keep the same order of performance (and cost) gap between the cache node and backend node.

2.1 Background and Related Work

Load Balancing. Dynamic load balancing is a key challenge to scaling out storage systems under skewed and fast changing real-world workloads [6, 9, 16]. The system performance must not become bottlenecked due to unevenly partitioned load across cluster nodes. Conventional static data partitioning techniques such as consistent hashing [44] do not help with dynamic load imbalance caused by skewed and rapidly-changing key popularity. Load balancing techniques that reactively replicate or transfer hot data across nodes often introduce performance and complexity overheads [45].

Prior research shows that a small, fast frontend cache can provide effective dynamic load-balancing by directly serving the most popular items without querying the backend



(a) Look-aside (b) Look-through (c) SwitchKV

Figure 2.1: Different cache architectures.

nodes, making the load across the backends much more uniform [28]. That work proves that the cache needs to store only the $O(n \log n)$ hottest items to guarantee good load balance, where n is the total number of *backend nodes* (independent of the number of keys). This theoretical result inspired the design of SwitchKV.

Caching Architectures. Look-aside [29] and on-path look-through [28] are the two typical caching architectures, shown in Fig. 2.1 and compared in Table 2.1. They suffer a major drawback when using a frontend cache for load balancing. In these architectures, clients must send all read requests to the cache first. This approach imposes high overhead when the hit ratio is low, which could be the case when the cache is small and used primarily for load balancing.

A cache miss in a look-aside architecture results in an additional round-trip of latency, as the query must be sent back to the client with a cache miss notification, and then resent to the backend. Look-through architectures reduce this latency by placing the cache in the on-path load balancer, however, the cache still must process each incoming request to determine whether to forward or serve it. Additionally, the load balancers become new critical failure points, which are far less reliable and durable than network switches [31].

	Look-aside	**Look-through**	**SwitchKV**
Clients' responsibilities	handle cache misses	nothing (transparent)	encode keys in packet headers
Cache load	*100% queries*	*100% queries*	cache hits only (likely <40% queries)
Latency with cache miss	*three* machine transits	*two* machine transits	only one machine transit
Failure points	switches	*cache*, switches	switches
Cache update involves	cache, backends	cache, backends	cache, backends, *switches*
Cache update rate limit	high	high	*low* (less than 10K/s in switch hardware)

Table 2.1: Comparison of different cache architectures.

2.2 SwitchKV Design

The primary design goal of SwitchKV is to remove redundant components on the query path such that latency can be minimized for all queries, throughput can scale out with the number of backend nodes, and availability is not affected by cache node failures.

The key to achieving this goal is the observation that specialized programmable network switches can play a key role in the caching system. Switch hardware has been optimized for decades to perform basic lookups at high speed and low cost. This simple but efficient function is a perfect match to the first step of a query processing: determine whether the key is cached or not.

The core of our new architectural design is an effective content-based routing mechanism. All clients, cache nodes, and backend nodes are connected with OpenFlow switches, as shown in Fig. 2.1c. Clients encode keys in query packet headers, and send packets to the cluster switch. Switches have forwarding rules installed, including exact match rules for each cached key and wildcard rules for each backend, to route queries to the right node

at line rate. Table 2.1 summarizes the significant benefits of this new architecture over traditional ones.

Exploiting SDN and fast switch hardware benefits system performance, efficiency and robustness. However, it also adds complexity and limitations. The switches have limited rule space and a relatively slow rule update rate. Therefore, cached keys and switch forwarding rules must be managed carefully to realize the full benefits of this new architecture. The rest of this section describes SwitchKV's query-processing flow and mechanisms to keep the cache up-to-date.

2.2.1 Content Routing for Queries

We first describe how SwitchKV handles client queries, assuming both cache and switch forwarding rules are installed and up-to-date. The process of updating cache and switch rules will be discussed in Section 2.2.2.

Query operations are performed over UDP, which has been widely used in large-scale, high-performance in-memory key-value systems for low latency and low overhead [55, 59]. Because UDP is connectionless, queries can be directed to different servers by switches without worrying about connection states. With a well-provisioned network, packet loss is rare [59], and simple application-driven loss recovery is sufficient to ensure both reliability and high throughput [55].

2.2.1.1 Key Encoding and Switch Forwarding

An essential system component to enable content-based routing is the programmable network switches that can install new *per cached key* forwarding rules on the fly. These switches can use both TCAM and L2/L3 tables for packet processing. The TCAM is able to perform flexible wildcard matches, but it is expensive and power hungry to include on switches. Thus, the size of the TCAM table is usually limited to a few thousand

entries [46, 66].[1] The L2 table, however, matches only on destination MAC addresses; it can be more cost-effectively implemented in switches and is more power-efficient. Modern commodity switches support 128K [66] or more L2 entries. These sizes may be insufficient for environments where a large percentage of data must be cached, but is a large enough cache size to ensure good load balancing in SwitchKV.

Key Encoding in Packet Headers. Because MAC addresses have more bits for key encoding and switches usually have large enough L2 tables to store forwarding rules for all cached keys, clients encode query keys in the destination MAC addresses of UDP packets. The MAC consists of a small *prefix* and a *hash* of the key, computed by the same consistent hashing used to partition the keyspace across the backends.

The prefix is used to identify the packet as being a request destined for SwitchKV, and to let the switches distinguish different types of queries. Only `get` queries coming directly from the clients may need to be forwarded to the cache nodes. Other types of queries should be forwarded to the backends, including `put` queries, `delete` queries, and `get` queries from a cache node due to cache misses. Therefore, get queries from the clients use one prefix, and all other queries use a different one.

In order to forward queries to the appropriate backend nodes, each client tracks the mapping between keyspace partitions and the backend nodes, and encodes *identifiers of backends* for the query keys into the destination IP addresses. This mapping changes only when backend nodes are added or removed, so client state synchronization has very low overhead.

Finally, the client's address and identity information is stored in the packet payload so that the node that serves the request knows where to send responses.

[1]Some high-end switches advertise larger TCAM table (e.g., 125K to 1 million entries [60]), albeit at higher cost and power consumption. Such capabilities would not meaningfully change our design, as our design primarily relies on exact-match rules.

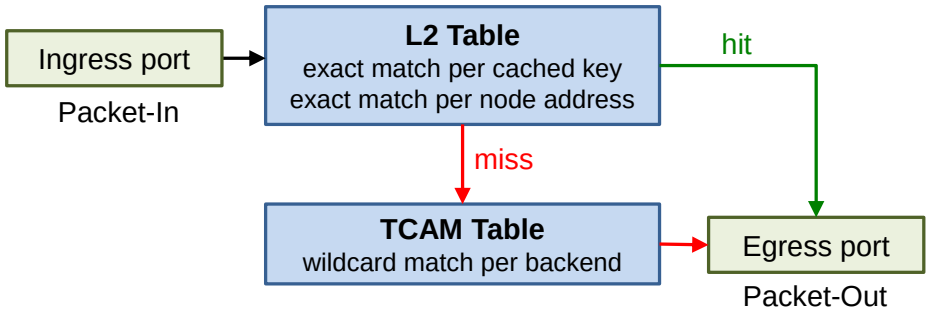


Figure 2.2: Packet flow through a switch.

Switch Forwarding. There are three classes of rules in switches, which are used to forward `get` queries for the cached keys to the cache nodes, other queries to the backends, and non-query packets (e.g., query responses, cache updates) to the destination node respectively. Fig. 2.2 shows the packet flow through a switch. The L2 table stores exact match rules on destination MAC addresses for each cached key and each cache and backend node. The TCAM table stores wildcard match rules on destination IP addresses for each backend node.

The L2 table is set to have a higher priority. A switch will first look for an exact match in the L2 table and will forward the packet to an egress port if either the packet was addressed directly to a node or it is a `get` query for a cached key. If there is no match in the L2 table, the switch will then look for a wildcard match in the TCAM and forward the packet to the appropriate backend node.

Below are the detailed switch forwarding rules:

- Exact match rules in L2 table for all cached keys. We use `pre1` to denote the prefix for `get` queries from clients. For each cached key in cache node:

  ```
  match:<mac_dst = pre1-keyhash>
  action:<port_out = port_cache_node>
  ```

- Exact match rules in L2 table for all clients, caches, and backends. For each node:

  ```
  match:<mac_dst = mac_node>
  action:<port_out = port_node>
  ```

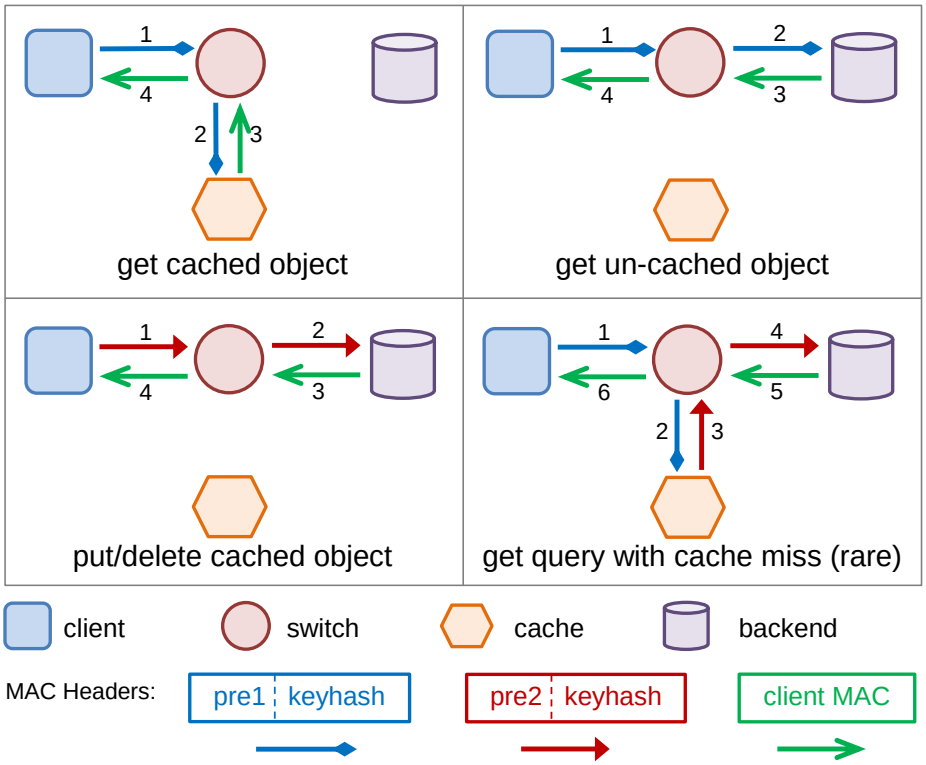


Figure 2.3: Query packets flows and destination MAC addresses. Internal messages for cache consistency during `put` or `delete` operations are not included. A cache miss only occurs due to key hash collision or temporarily outdated switch rules.

- Wildcard match rules in TCAM table for all backend nodes. For each backend node:

```
match:<ip_dst/mask = id_node>

action:<port_out = port_backend_node>
```

2.2.1.2 Query Flow Through the System

A main benefit of SwitchKV is that it can send queries to the appropriate nodes with minimal overhead, especially for queries on uncached keys which make up most of the traffic. Fig. 2.3 shows the possible packet flows of queries.

Handle Read Requests. SwitchKV targets read-heavy workloads, so the efficiency of handling read requests is critical to the system performance. Switches route `get` queries to the cache or backends based on match results in the forwarding tables. When it receives

a `get` query, the cache or backend node will look for the key in its local store, either in memory or SSD. The backend will send a reply message with the destination MAC set as the client address. The cache node will also reply if the key is found. This reply will be forwarded back to the client.

In most cases, queries sent to the cache node will hit the cache, because queries for keys not in the cache were filtered out by the switches. However, it is possible for a cache node to receive a `get` query but not find the key in its local in-memory store. This may occur due to a small delay in rule removal from the switch, or a rare hash collision with another key. When this happens, the cache node must forward the packet to the backends. To do so, the cache will send the query packet back to the switch, with the appropriate destination MAC address prefix (e.g., from `pre1` to `pre2` in Fig. 2.3). This prevents the packet from matching the same L2 rule in switches again, so that the query can be forwarded to the appropriate backend node via a wildcard match in TCAM.

Handle Write Requests. Clients send `put` and `delete` queries with a MAC prefix that is different from the prefix of `get` queries (as shown in Fig. 2.3), so that the packets will not trigger a rule in the L2 table of switches, and will be forwarded directly to the backends. When a backend node receives a `put` or `delete` query for a key, it will update its local data store and reply to the client.

Each backend node keeps track of which keys in its local store are also being cached. If a `put` or `delete` request for a cached key arrives, the backend will send messages to update the cache node before replying to the clients. The cache node is then responsible for communicating the update to the network controller for switch rule updates. This policy ensures that data items in the cache and backends are consistent to the client, but allows temporary inconsistency between cached keys and switch forwarding rules. The next section describes the detailed mechanism of cache update and consistency.

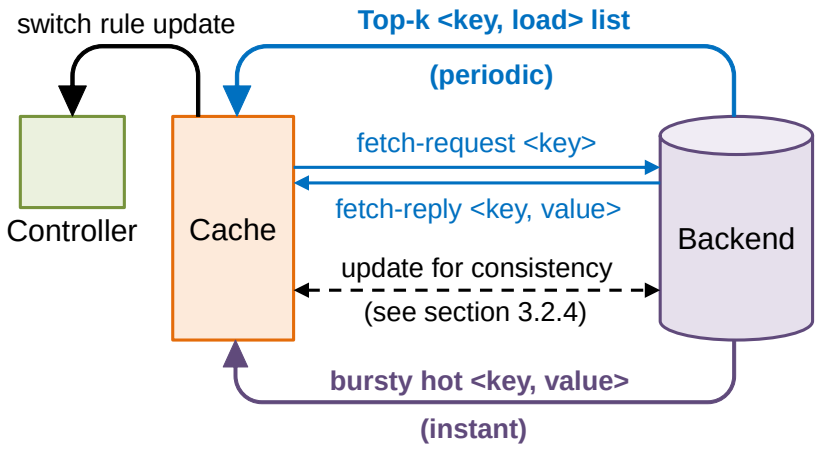


Figure 2.4: Cache update overview.

2.2.2 Hybrid Cache Update

As our goal is to build a system that is robust for (nearly) arbitrary workloads, the limited forwarding rule update rate poses challenges for the caching mechanism. Since each cache addition or eviction requires a switch rule installation or removal, the rule update rate in switches directly limits the cache update rate, which affects how quickly SwitchKV can react to workload distribution changes. Though switches are continuously being optimized to speed up their rule update and some switches can now achieve 12K updates per second [60], they are still too slow to support traditional caching strategies that insert each recently-visited key to the cache.

To meet this new challenge, we designed new hybrid cache update algorithms and protocols to minimize unnecessary cache churn. The cache update mechanism consists of three components: 1) Backends *periodically* report recent hot keys to the cache nodes. 2) Backends *immediately* report keys that suddenly become very hot to the cache nodes. 3) Cache nodes add selected keys from reports and evict appropriate keys when necessary, and they instruct the network controller to make corresponding switch rule updates through REST APIs. Cache addition is prioritized over eviction in order to react quickly to sudden workload distribution changes at the cost of some additional buffer switch rule space. Fig. 2.4 shows our cache update mechanism at a high level.

2.2.2.1 Update with Periodic Hot Key Report

In most caching systems, a query for a key that is not in the cache would bring that key into cache and evict another key if the cache is full. However, many recently visited keys are not hot and will not be accessed again in the near future. This would result in unnecessary cache churn, which would harm the performance of SwitchKV because its cache update rate is limited.

Instead, we use a different approach to add objects to the cache less aggressively. Each backend node maintains an efficient top-k load tracker to track recent popular keys. Backend nodes periodically (e.g., every second) report their recent hot keys and loads to the cache nodes. Each cache node maintains an in-memory data store and frequency counter for the cached items with the same load metric. The cache node keeps a load threshold based on the load statistics of cached keys. Upon receiving the reports, the cache node selects keys whose loads are above the threshold to add to the cache. It sends `fetch` requests for the selected keys to the corresponding backend nodes to get the values. It then updates the cache and instructs the network controller to update switch rules based on the received `fetch` responses.

Time-segmented Top-K Load Tracker. Each backend node maintains a key-load list with k entries to store its approximated hottest k keys and their loads. It also keeps a local frequency counter for the recently visited keys, so that it can know what are the most popular keys and how frequently they are queried. A backend node cannot afford to keep counters for all keys in memory. Instead, since only information about hot keys is needed, we can use memory-efficient top-k algorithms to find frequent items in the query stream [17].

To keep track of *recent* hot keys, we segment the query stream into separate intervals. At the end of each interval, the frequency counter extracts the top-k list of its current segment, then clears itself for the next segment. The key-load list is updated by the top-k

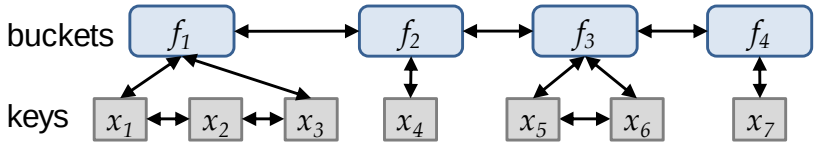


Figure 2.5: Structure of top-k frequency counter.

list of the new segment using weighted average. Suppose the frequency of key x in the new segment is f_x, and the current load of x is L'_x, then the new load of x is

$$L_x = \alpha \cdot f_x + (1 - \alpha) \cdot L'_x, \tag{2.1}$$

where α represents the degree of weighting decrease. A higher α discounts previous load faster. Only keys in the new top-k list will be kept in the new updated key-load list. L'_x is zero for keys not in the previous key-load list.

The frequency counter uses a "space-saving algorithm" [58] to track the heavy hitters of the query stream in each time segment and approximate the frequencies of these keys. Fig. 2.5 shows the data structure of the frequency counter.

The counter consists of a linked list of buckets, each with a unique frequency number f. The buckets are sorted by their frequency in increasing order (e.g., $f_1 < f_2$). Each bucket has a linked list of keys that have been visited for the same number of times, f. Keys in the same bucket are sorted by their most recent visited time, with newest key at the tail of the list. With this structure, getting a list of top-k hot keys and their load is straightforward. For example, the top-5 list in Fig. 2.5 is $[\langle x_7, f_4\rangle\ \langle x_6, f_3\rangle\ \langle x_5, f_3\rangle\ \langle x_4, f_2\rangle\ \langle x_3, f_1\rangle]$.

The counter has a configurable size limit N, which is the maximum number of keys it can track. Algorithm 1 describes how to update the counter. When processing (e.g., create, delete, move) buckets and keys, the orders described above are always maintained. The counter requires $O(N)$ memory, and has O(1) running time for each query. To reduce the computational overhead, we can randomly sample the query packets, and only update the

Algorithm 1 Update Frequency Counter

```
1:  function SeeQuery(x)
2:      if x is not tracked in the counter then
3:          if the counter is not full then
4:              create a bucket with f = 1 if not exists
5:              add x to the first bucket;
6:              return
7:          y ← first key of first bucket, which is the least visited key
8:          replace y with x and keep the same frequency
9:      Update(x)
10: function Update(x)
11:     ⟨b, f⟩ ← current ⟨bucket, frequency⟩ of x
12:     if next bucket of b has frequency f + 1 then
13:         move x to the next bucket
14:     else if x is the only key of b then
15:         increase frequency of b to f + 1
16:     else  creat a new bucket with frequency f + 1 and move x to it
17:     if b is empty then delete b
```

counter for a small fraction of the queries. Sampling can provide a good approximation of the ranking of heavy hitters in highly skewed workloads.

Cache Adds Selected Keys from Reports. The cache also tracks the load for all cached keys. In order to be comparable with the load of reported keys, it must keep the same parameters (e.g., time segment interval, average weights, sampling rate) with the tracker in the backends. Cache nodes update a load threshold periodically based on the loads of cached keys, and send `fetch` queries for the reported keys with load higher than the threshold.

Too big of a threshold would prevent caching hot keys, while a too small of one would cause frequent unnecessary cache churn. To compute a proper load threshold in practice, the cache samples a certain number of key loads and uses the load at a certain rank (e.g., 10^{th} percentile from the lowest) as the threshold value. This process runs in the background periodically, so it does not introduce overhead to serving queries or updating cached data.

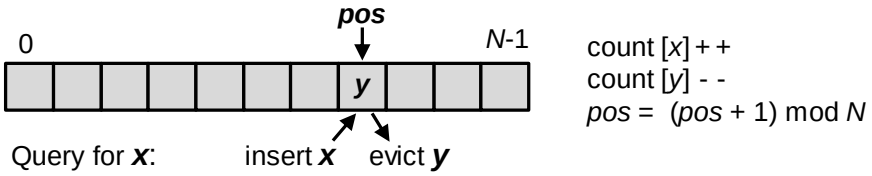


Figure 2.6: Circular log and counter.

2.2.2.2 Update for Bursty Hot Keys

Periodic reports can update the cache effectively with low communication and memory overhead, but cannot react quickly when some keys suddenly become popular. In addition to periodic reports, the backends also send instant reports to the cache to report bursty queries, so that those queries can be offloaded to the cache immediately.

Each backend maintains a circular log to track the recently visited keys, and a hash table that keeps only entries for keys currently in the log and tracks the number of occurrence of these keys. As shown in Fig. 2.6, when a key is queried, it is inserted into the circular log, with the existing key at that position evicted. The hash table updates the count of the keys accordingly and adds or deletes related entries when necessary. If the count of a key exceeds a threshold and the node's overall load is also above a certain threshold, the key and its value are *immediately sent and added to the cache*. The size of the circular log and hash table could be small (e.g., a few hundreds of entries), which introduces little overhead to query processing.

2.2.2.3 Handle Burst Change with Rule Buffer

The distribution changes in real-world workloads are not constant. Sudden changes in the key popularity may lead a large number of cache updates in a short period of time. In traditional caching algorithms, a cache addition when the cache is full would also trigger a cache eviction, which in SwitchKV would mean that each addition involves two forwarding rule updates in the switch. As a result, the cache would only able to add keys at half of the switch update rate on average.

In order to react quickly to sudden workload changes, we prioritize cache addition over eviction. Cache evictions and switch rule deletion requests are queued and executed after cache additions and rule installations until a maximum delay is reached. In this way, we can reduce the required peak switch update rate for bursty cache updates to half, so that new hot keys can be added to cache more quickly. For example, if the switch update rate limit is 2000 rules per second, and the maximum delay for rule deletions is one second, then the cache can update at 1000 keys/second on average, and a maximum of 2000 keys/second for a short period (one second).

To allow delay in switch rule deletions, a rule buffer must be reserved in the L2 table. The size of this buffer is the maximum switch update rate times the duration of maximum delay. In the example above, the switch should reserve space for at least 2000 rules, which is small compared to the available L2 table size in switches.

Delaying rule deletion may result in stale forwarding rules in the L2 table. The stale rules will produce a temporary cache miss for some queries, as shown in the lower right block of Fig. 2.3. The miss overhead is small, however, because the evicted or deleted keys are (by definition) less likely to be frequently visited.

2.2.2.4 Cache Consistency

SwitchKV always guarantees consistent responses to clients. As a performance optimization, it allows temporary inconsistency between switch forwarding rules and cached keys, which (as described above) can introduce temporary overheads for a small number of queries, but never causes inconsistent data access.

In traditional cache systems such as Memcached [29], when a client sends a `put` or `delete` request, it will also send a request to the cache to either update or invalidate the item if it is in cache. The cache in SwitchKV is small and it is possible that most requests are for uncached keys, so forwarding each `put` or `delete` request to the cache introduces unnecessary overhead.

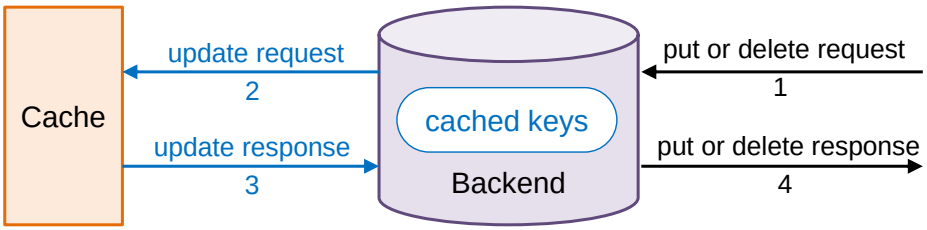


Figure 2.7: Updates to keep cache consistency.

The backends avoid this overhead by tracking, in-memory, which keys in its local store are currently cached. The backend only updates the cache when it receives requests for one of these cached keys. Keys are added to the set whenever the backend receives a `fetch` request, or sends an instant hot object detected by the circular-log counter. When the cache evicts a key, or decides not to add the item from a `fetch` response or instant report, it sends a message to the backend so that the backend can remove this key from its cached key set.

We use standard leasing mechanisms to ensure consistency when there are cache or backend failures or network partitions [33]. Backends grant the cache a short-term lease on each cached key. The cache periodically renews its leases and only return a cached value while the lease is still valid. When a backend receives a `put` or `delete` request for a cached key, it will send an `update` request to the cache, as shown in Fig. 2.7, and will wait for the response or until the lease expires before it replies to the client. We choose to update the cached data rather than invalidate it for a `put` request to reduce the cache churn and rule update burden on switches.

2.2.3 Local Storage and Networking

Optimizing the single-node local performance of cache and backends is not our primary goal, and has been extensively researched [54, 55, 69]. Nevertheless, we made several design choices on local storage and networking to maximize the potential performance of each server, which we discuss here.

2.2.3.1 Parallel Data Access

Exploiting the parallelism of multi-core systems is critical for high performance. Many key-value systems use various concurrent data structures so that all cores can access the shared data in parallel [20, 27, 59]. However, they usually scale poorly with writes and can introduce significant overhead and complexity to our cache update algorithms that require query statistics tracking.

Instead, SwitchKV partitions the data in each cache and backend node based on key hash. Each core has *exclusive access* to its own partition, and runs its own load trackers. This greatly improves both the concurrency and simplicity of the local stores. Prior work [27, 56, 74] observed that partitioning may lower the performance when the load across partitions is imbalanced. In SwitchKV, however, backend nodes do not face high skew in key popularity. By exploiting CPU caches and packet burst I/O, a cache node that serves a small number of keys can handle different workload distributions [55].

2.2.3.2 Network Stack

SwitchKV uses Intel® DPDK [40] instead of standard socket I/O, which allows our user-level libraries to control NICs, modify packet headers, and transfer packet data with minimal overhead [55].

Since each core in the cache and backend nodes has exclusive access to its own partition of data, we can have the NIC deliver each query packet to the appropriate RX queue based on the key. SwitchKV can achieve this by using Receive Side Scaling (RSS) [23, 36] or Flow Director (FDir) [55, 65].[2] Both methods require information about the key in packet headers for the NIC to identify which RX queue should the packet be sent to. This requirement is automatic in SwitchKV where key hashes are already part of the packet header.

[2]Our prototype uses RSS. FDir enables more flexible control of the network stack, but it is not supported in the Mellanox NICs that we use.

2.2.4 Cluster Scaling

To scale system performance, the cluster will require multiple caches and OpenFlow switches. This section briefly sketches a design (not yet implemented) for a scale-out version of SwitchKV.

Multiple Caches. We can increase SwitchKV's total system throughput by deploying additional cache nodes. As each individual node can deliver high throughput because of its small dataset size (especially when keys fit within its L3 cache), we do not replicate keys across nodes and instead simply partition the cache across the set of participating nodes. Note that while we *are* very concerned about load amongst our backend nodes, our cache nodes have orders-of-magnitude higher performance, and thus the same load-balancing concerns do not arise. Each cache node is responsible for multiple backends, and each backend reports only to its dedicated cache node. As such, we do not require any cache coherency protocols between the cache nodes.

If the mapping between backends and cache nodes changes, the relevant backends will delete their cached items from their old cache nodes, and then report to the new ones. If the change is due to a cache crash, the network controller will detect the failed node and delete all forwarding rules to it.

Network Scaling. To scale network throughput, we can use the well-studied multi-rooted fat-tree [3, 35]. Such an architecture may require exact match rules for cached keys to be replicated at multiple switches. This approach may sacrifice performance until the rule updates complete, but does not compromise correctness (the backends may need to serve the keys temporarily).

On the other hand, if the switching bottleneck is in terms of rule space (as opposed to bandwidth), then each switch must be configured to store only rules for a subset of the backend nodes, i.e., we partition the backends, and thus the rule space, across our switches.

In this case, queries for keys in a backend node must be sent through a switch associated with that key's backend (i.e., that has the appropriate rules); that switch can be identified easily by the query packets' destination IP addresses.

2.3 Evaluation

In this section, we demonstrate how our new architecture and algorithms significantly improve the overall performance of a key-value storage cluster under various workloads. Our experiments answer three questions:

- How well does a fast small cache improve the cluster load balance and overall throughput? (§2.3.2)
- Does SwitchKV improve system throughput and latency compared to traditional architectures? (§2.3.3)
- Can SwitchKV's new cache update mechanism react quickly to workload changes? (§2.3.4)

Our SwitchKV prototype is written in C/C++ and runs on x86-64 Linux. Packet I/O uses DPDK 2.0 [40]. In order to minimize the effects of implementation (rather than architectural) differences, we implemented the look-aside and look-through caches used in our evaluation simply by changing the query data path in SwitchKV.

2.3.1 Evaluation Setup

Platform. Our testbed consists of four server machines and one OpenFlow switch. Each machine is equipped with dual 8-core CPUs (Intel® Xeon® E5-2660 processors @ 2.20 GHz), 32 GB of total system memory, and one 40Gb Ethernet port (Mellanox ConnectX-3 EN) that is connected to one of the four 40GbE ports on a Pica8 P-3922 switch. Fig. 2.8

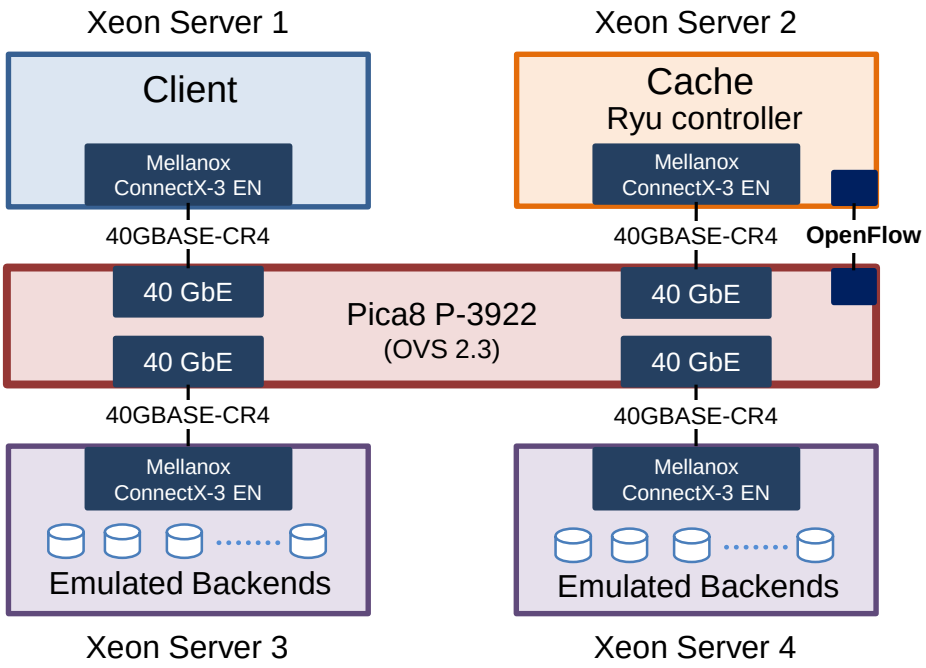


Figure 2.8: Evaluation platform.

diagrams our evaluation platform. One machine serves as the client, one machine as the cache, and two machines emulate many backends nodes.

We derived our emulated performance from experimental measurements on a backend node that fits our target configuration: an Intel® Atom™ C2750 processor paired with an Intel® DC P3600 PCIe-based SSD. On this SSD-based target backend, we ran RocksDB [69] with 120 million 1KB key-value pairs, and measured its performance against a client over a 1Gb link. The backend could serve 99.4K queries per second on average.

Each emulated backend node in our experiments runs its own isolated in-memory data structures to serve queries, track workloads, and update the cache. It has a configurable maximum throughput enforced by a fine-grained rate limiter. Since it is hard to predict the performance bottleneck at a backend node if its load is skewed, we assume backends have a fixed throughput limit as measured under uniform workloads. The emulated backends do not store the actual key-value pairs due to limited memory space. Instead, they reply to the client or update the cache with a fake random value for each key. In most experiments (except Fig. 2.14), we emulate a total of 128 backend nodes in the two server machines,

Number of backend nodes	128
Max throughput of each backend	100 KQPS
Workload distribution	Zipf (0.99)
Number of items in cache	10000

Table 2.2: Default experiment settings unless otherwise specified

and limit each node to serve at most 100K queries per second. Table 2.2 summarize the default experiment settings unless otherwise specified.

Workloads and Method. We evaluate both skewed and uniform workloads in our experiments, and focus mainly on skewed workloads. Most skewed workloads use a non-uniform key popularity that follows a Zipf distribution of skewness 0.99, which is the same that used by YCSB [16]. The request generator uses approximation techniques to quickly generate workloads with a Zipf distribution [34, 55]. The keyspace size is 10 billion, so each of the 128 backend nodes is responsible for serving approximately 78 million unique keys. The mapping of a given key to a backend is decided by the key hash. We use fixed 16-byte keys and 128-byte values.

Most experiments (except Fig. 2.12) use read-only workloads, since SwitchKV aims to load balance read requests. All write requests have to be processed by the backends, so they cannot be load balanced by the cache.

To find the maximum effective system throughput, the client tracks the packet loss rate, and adjusts its sending rate every 10 milliseconds to keep the loss rate between 0.5% to 1%. This self-adjusted rate control enables us to evaluate the real-time system performance.

Our server machines can send packets at 28 Mpps, but receive at only 15 Mpps. To avoid the system being bottlenecked by the client's receiving rate, the backends and cache node fully process all incoming queries, but send only half of the responses back to the client. The client doubles its receiving rate before computing the loss rate.

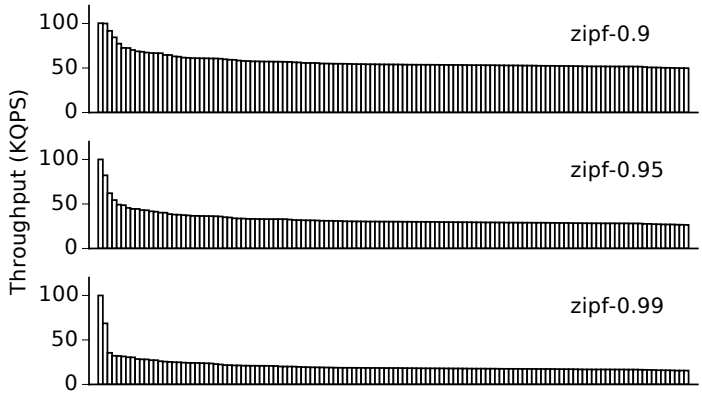


Figure 2.9: Throughput of each backend node without cache under workloads with different Zipf skewness. Node IDs (x-axis) are sorted according to their throughput.

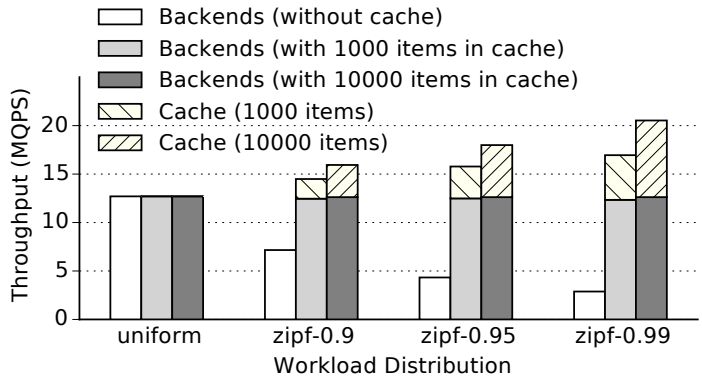


Figure 2.10: System throughput with and without the use of a cache. Figure illustrates the portion of total throughput handled by the cache and that by backend nodes.

2.3.2 Load Balancing with a Small Cache

We first evaluate the effectiveness of introducing a small cache for reducing load imbalances.

Fig. 2.9 shows a snapshot of the individual backend node throughput with caching disabled under workloads of varying skewness. We observe that the load across the backend nodes is highly imbalanced.

Fig. 2.10 shows how caching affects the system throughput. Under uniform random workload, the backends total throughput can reach near the maximum capacity (128 backends $\times$ 100 KQPS). However, when the workload is skewed, the system throughput without the cache is bottlenecked by the overloaded node and significantly reduced. Adding a small

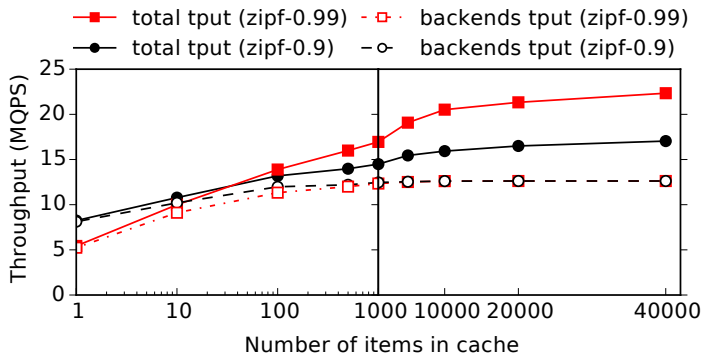


Figure 2.11: System throughput as cache size increases. Even a modest-sized cache of 10,000 items achieves significant gains.

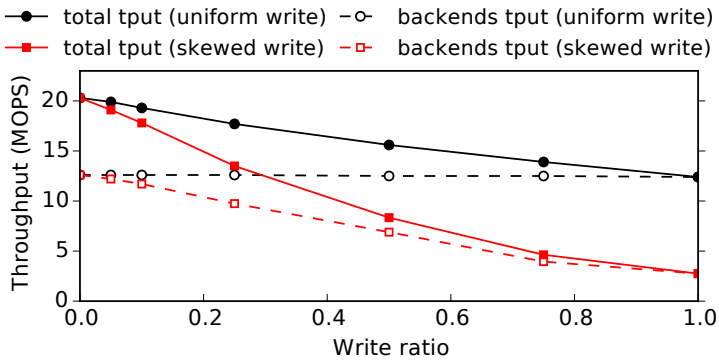


Figure 2.12: System throughput with different write ratio.

cache can help the system achieve good load balance across all of nodes: A cache with only 10,000 items can improve the system's overall throughput by 7× for workloads with Zipf skewness of 0.99.

Fig. 2.11 investigates how different numbers of cached items affect the system throughput. The backends' load quickly becomes well balanced as the number of cached items grows to 1000. Then, the system throughput continues to grow as more items are cached, but the benefits from increased cache size diminish (as one expects given a Zipf workload). The system would require significantly more memory at the cache node or many more cache nodes to further increase the hit ratio. We choose to cache 10,000 items for the rest of the experiments.

Fig. 2.12 plots the systems throughput with different write ratios and write workloads. We assume the backend nodes have the same performance for read and write operations,

and use two types of write workload: write queries uniformly distributed across all keys and write queries according to the same Zipf 0.99 distribution as read queries. Write workloads cannot be balanced by the cache, so the system throughput with skewed write workload quickly decreases as the write ratio increases. With the uniform write workload, load across the backends is always uniform, so increasing the write ratio only decreases the effective throughput of the cache.

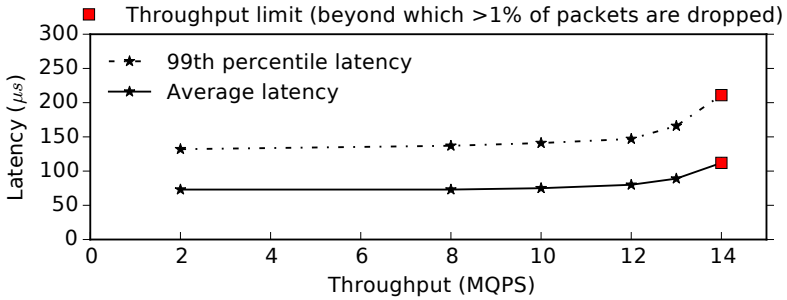
2.3.3 Benefits of the New Architecture

This section compares the system performance between SwitchKV and traditional look-aside and on-path look-through architectures. As summarized in Table 2.1, compared to traditional architectures in which the cache handles all queries first, the cache in SwitchKV is only involved when the requested key is already cached (with high likelihood), and thus uncached items are served with only a single machine transit. As a result, we expect SwitchKV to have both lower latency and higher throughput than traditional architectures, which is strongly supported by our experimental results.

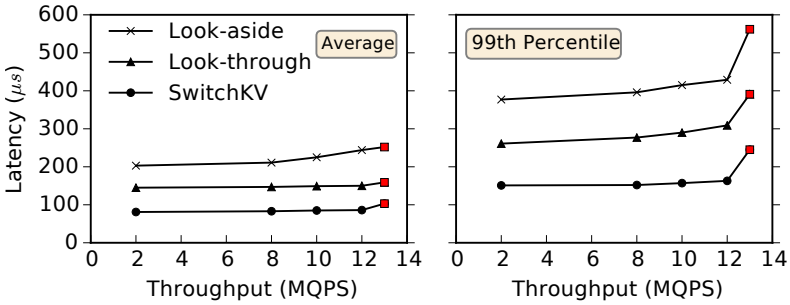
Latency. We first compare the average and 99^{th} percentile latency of different architectures, as shown in Fig. 2.13. To measure the end-to-end latency, the client tags each query packet with the current timestamp. When receiving responses, the client compares the current timestamp and the previous timestamp echoed back in the responses. To measure latency under different throughputs, we disable the client's self rate adjustment, and manually set different send rates.

Fig. 2.13a shows the latency when the client only sends queries for keys in the cache. In all three architectures, the queries will be forwarded to the cache by the switch and the cache reply directly to the client. Accordingly, they have the same latency for cache hits.

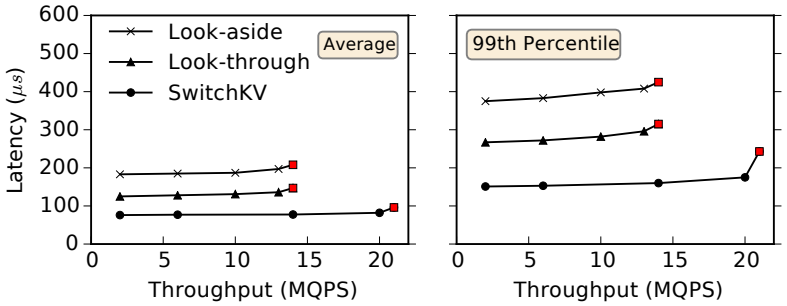
Fig. 2.13b shows the latency when the client generates uniform workloads and the cache is empty, which results in all queries missing the cache. Look-aside has the highest latency



(a) Queries for cached keys with Zipf 0.99 workloads.



(b) Queries for uncached keys with uniform workloads.



(c) Zipf 0.99 workloads with 10000 items in cache.

Figure 2.13: End-to-end latency as a function of throughput.

because it takes three machine transits (cache→client→backend) to handle a cache miss. Look-through also has high latency because it takes two machine transits (cache→backend) to handle a cache miss. In comparison, queries for uncached keys in SwitchKV cache will directly go to the backend nodes.

Fig. 2.13c shows the overall latency for a Zipf 0.99 workload and 10000-item cache. As shown in Fig. 2.10, about 38% of queries will hit the cache under these settings. The average latency is within the range of cache hits and cache misses. The 99^{th} percentile latency is about the same as cache miss latency. As all queries must go through the cache in look-aside and look-through architectures, we cannot collect latency measurements beyond

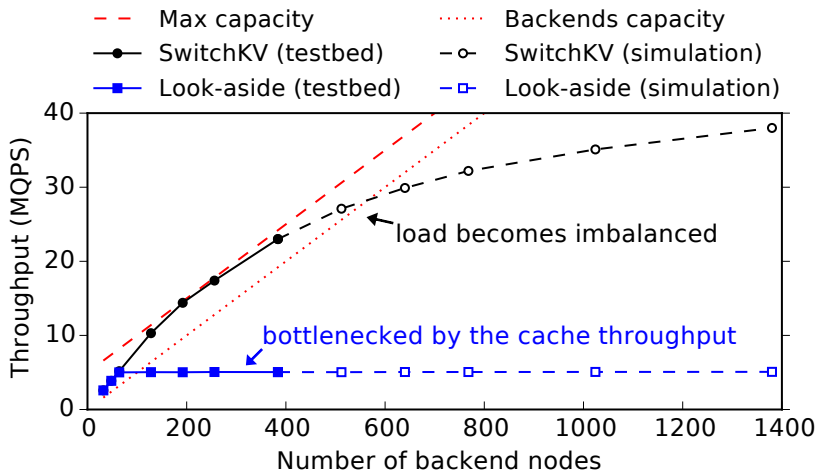


Figure 2.14: System throughput scalability as the number of backend nodes increases, for SwitchKV and look-aside architecture with Zipf 0.99 workload and at most 10000 items in cache. On-path look-through has the same throughput as look-aside. Each backend node is rate limited at 50K queries per second, cache is rate limited at 5 million queries per second. Look-through has similar performance to look-aside.

the 14 million QPS mark for them, as the cache is unable to handle more traffic. This result illustrates one of the major benefits of the SwitchKV design: requests for uncached keys are simply not sent to the cache, allowing a single cache node to support more backends (higher aggregate system throughput).

Throughput. We then compare the full system throughput under a Zipf 0.99 workload as the number of backend nodes increase, for different architectures. For each architecture, the cache node stores at most 10000 items.

In order to emulate more backend nodes in this experiment, we scale down the rate capacity of each backend node to at most 50K queries per second, and limit the cache to serve at most 5 million queries per second. The performance improvement ratio of SwitchKV to other architectures will be the same as long as the performance ratio of the cache to a backend node is 100:1. To achieve the maximum system throughput, the cache may store fewer items when it becomes the performance bottleneck as the backend cluster size increases.

Fig. 2.14 shows the experiment results. The throughput of the look-aside architecture is bottlenecked quickly by the cache capacity when the number of backend nodes increases to 64, while the throughput of SwitchKV can scale out to much larger cluster sizes. When the number of backend nodes goes beyond 400, the throughput begins to drop below the maximum system capacity, because the cache is insufficient for providing good load balance for such a cluster. To retain linear scalability as the cluster grows, we would need to have a more powerful cache node or increase the number of cache nodes.

Less skewed workloads will yield better scalability for SwitchKV, but will hit the same performance bottleneck for both look-aside and look-through architectures. Due to space constraints, we omit these results.

2.3.4 Cache Updates

This section evaluates the effectiveness of SwitchKV's hybrid cache-update mechanisms. In these experiments, we keep the workload distribution (Zipf 0.99) the same, and change only the popularity of each key. The workload generator in the client actually generates key indices with fixed popularity ranks. We change the query workloads by changing the mapping between indices and key strings. We use three different workload change patterns:

1. ***Hot-in***: Move N cold keys to the top of the popularity ranks, and decrease the ranks of other keys accordingly. This change is radical, as cold keys suddenly become the hottest ones in the cluster.

2. ***Hot-out***: Move N hottest keys to the bottom of the popularity ranks, and increase the ranks of other keys accordingly. This change is more moderate, since the new hottest keys are most likely already in the cache if N is smaller than the cache size.

3. ***Random***: Replace N random keys in the top K hottest keys with cold keys. We typically set K to the cache size. This change is typically moderate when N is not large, since the probability that most of the hottest keys are changed at once is low.

A note about our experimental infrastructure, which affects SwitchKV's performance under rapid workload changes: The Pica8 P-3922 switch's L2 rule update is poorly implemented. The switch performs an unnecessary linear scan of all existing rules before each rule installation, which makes the updates very slow as the L2 table grows. We benchmark the switch and find it can only update about 400 rules/second when the there are about 10K existing rules, which means the cache can only update 200 items/second on average. Some other switches can update their rules much faster (e.g., 12K updates/second [60]). Though still too slow to support the update rate needed by traditional caching algorithms, these switches would provide much higher performance with SwitchKV under rapidly changing workloads.

All experiments use Zipf 0.99 workloads and a 10000-item-sized cache. Each experiment begins with a pre-populated cache containing the top 10,000 hot items. Each backend node sends reports to the cache as follows: its top five hot keys every second, and keys that were visited more than eight times within the last two hundred queries instantly. The choice of parameters for periodic and instant updates is flexible, determined by the performance goals, cache size, and update rate limit. For example, the size and threshold of the ring counter for instant reports determines when a key is hot enough to be immediately added to the cache. A threshold that is too low may cause unnecessary cache churn, while a threshold that is too high may make the cache slow to respond to bursty workload changes. We also compare SwitchKV with a traditional update method, in which backends try to add every queried key to the cache.

We first evaluate system throughput under the *hot-in* change pattern. Since this is a radical change, we do not expect it to happen frequently. Thus, we move 200 cold keys to the top of the popularity ranks every ten seconds. Fig. 2.15 shows the system throughput over time. A traditional cache update method has very poor performance, as it performs many cache updates for recently-visited yet non-hot keys. With periodic top-k reports alone, a backend's hot keys are not added to the cache until its next report (once per second). The

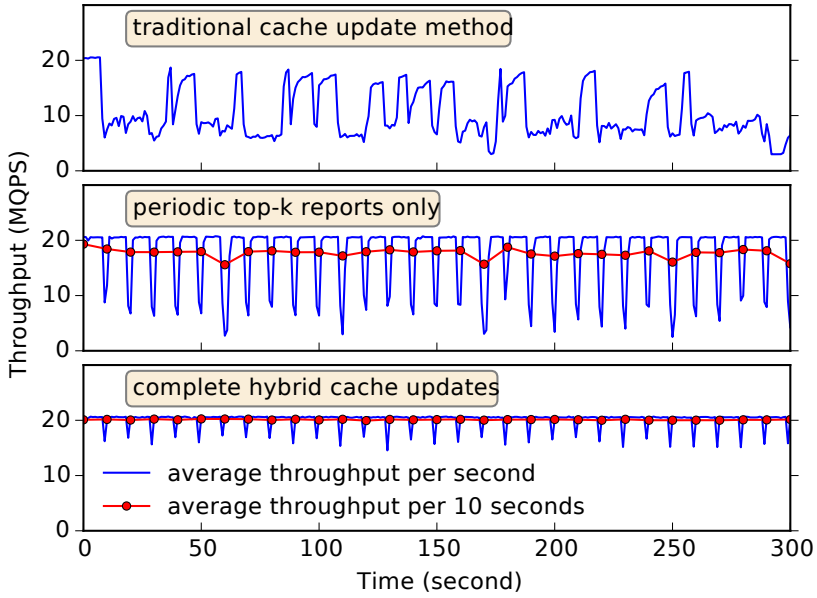


Figure 2.15: Throughput with *hot-in* workload changes, i.e., change 200 cold keys into the hottest keys every 10 seconds.

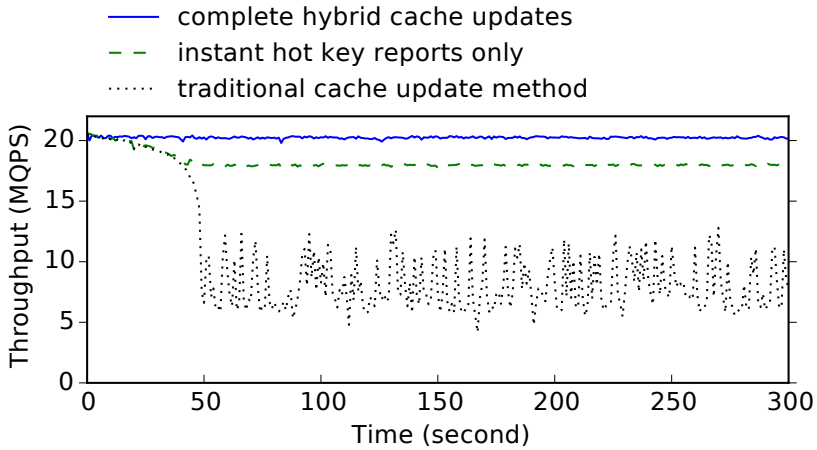


Figure 2.16: Throughput with *hot-out* workload changes, i.e., move out 200 hottest keys every second.

throughput is reduced to less than half after the workload changes, and recovers in 1-2 seconds. The bottom subfigure shows SwitchKV's throughput using its complete cache update mechanism, which includes the instant hot key reports. The new hot keys are immediately added to the cache, resulting in a lower performance drop and a much faster recovery after a sudden workload change. This demonstrates that SwitchKV is robust enough to meet the performance goals even with certain adversarial changes in key popularity.

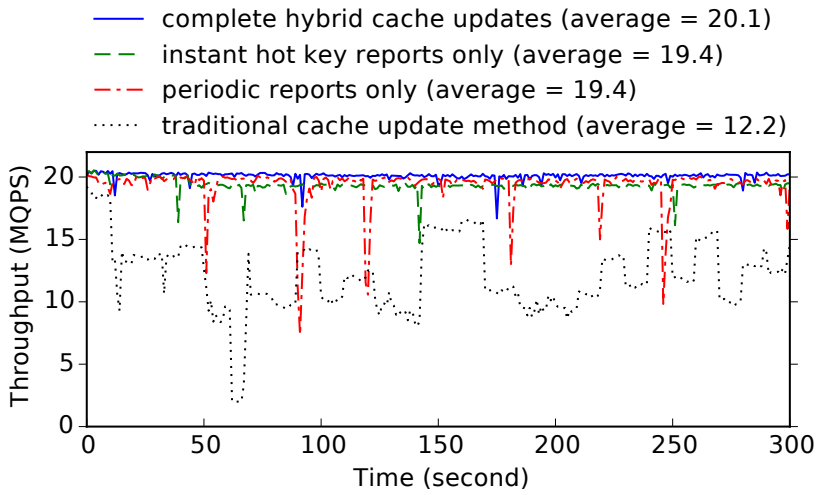


Figure 2.17: Throughput with *random* workload changes, i.e., replace 200 out of the top 10000 keys every second.

Our next experiment evaluates SwitchKV's throughput under a *hot-out* change pattern. Every second, the 200 hottest keys suddenly go cold, and we thus increase the popularity ranks of all other keys accordingly. As shown in Fig. 2.16, the complete update mechanism can handle this change well. With instant reports only and no periodic reports, the system cannot achieve its maximum throughput: the circular log counter can detect only very hot keys, not the keys just entering the bottom of the top-10000 hot-key list. These keys are only added to cache as they further increase in their popularity when more of the hottest keys move out. Note that this gap becomes particularly apparent as the system reaches its steady state 50 seconds into the experiment; at this point, none of the pre-populated cached keys remain in the cache.

Fig. 2.17 shows the throughput with a *random* change pattern, in which we randomly replace 200 keys in the top 10000 popular keys every second. The complete update mechanism is able to handle the workload changes. There are occasionally short-term small performance drops, which occur when the hottest keys are replaced. The throughput would be lower, however, if SwitchKV were to omit either its instant or periodic reports.

Fig 2.18 shows the effectiveness of SwitchKV's rule buffer in handling bursty workload changes (see §2.2.2.3). The maximum delay for cache eviction and rule deletion is set to

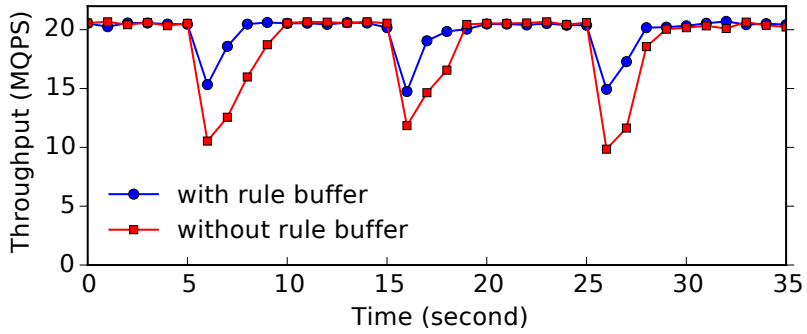


Figure 2.18: Throughput with *hot-in* workload changes with 600 new hottest keys every time, which requires 1200 rule updates and will take the switch at least three seconds to finish them.

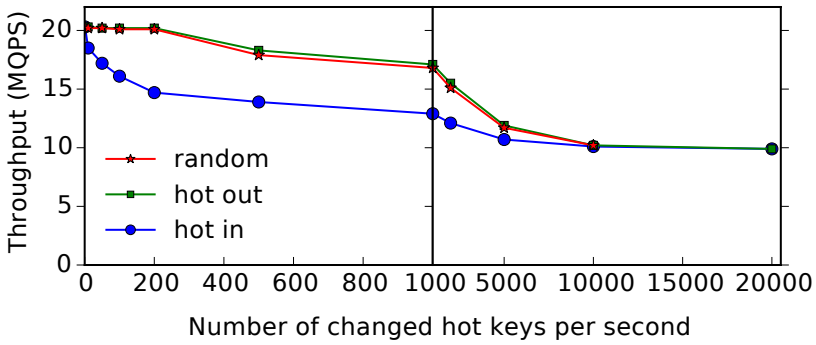


Figure 2.19: Throughput with different workload change patterns as a function of change rate.

2 seconds. With a switch rule buffer and prioritizing rule installation, the 600 new keys can be added to the cache within 1.5 seconds. Without the rule buffer, this installation time would double. The rule buffer thus reduces any throughput impact and allows faster recovery during bursty workload changes.

Fig. 2.19 shows the average throughput with different change patterns and rates. The switch can update 400 rules per second, which can support 200 cache updates per second. The system throughput is near maximum for random and hot-out change patterns when the change rate is within 200 keys per second, and then goes down as the change rate increases. Throughput drops quickly under increasing hot-in changes, as the cache is less effective when more of the hottest keys change every second. Once all patterns change more than 10000 of the hottest keys per second, all three patterns yield similar throughput, as all patterns replace the entire cache every second. Still, even at this point the cache can

still keep up to 200 of current hot keys, and most of the hottest keys are likely to added to the cache from the instant reports, so throughput is still much higher (by 3×) than that of the system lacking a cache. The performance under fast changing workloads would be higher with switches that can update their rules faster.

2.4 Conclusion

SwitchKV is a high-performance and cost-efficient SSD-based key-value storage cluster. It can maintain efficient load balancing under widely varying and rapidly changing real-world workloads. SwitchKV achieves stable high throughput and low tail latency in a cost-effective manner, both by combining fast small caches with new algorithm design, and by exploiting SDN techniques and switch hardware. All storage nodes in a SwitchKV cluster can be fully utilized regardless of the workload distributions, and all queries can be served with minimal overhead through efficient content-aware routing. We demonstrate SwitchKV can meet throughput and latency performance goals more efficiently than traditional systems.

Chapter 3

Fast Concurrent Cuckoo Hashing

As discussed before, the key to improve the performance and efficiency of each individual key-value storage server is to maximize memory efficiency and exploit multi-core parallelism. This chapter focuses on the most widely used key-value data structure—the hash table. We present a new memory-efficient concurrent hash table that can achieve high throughput for both read- and write-intensive workloads.

High-performance, concurrent hash tables are one of the fundamental building blocks for modern systems, used both in concurrent user-level applications and in system applications such as kernel caches. As we continue our hardware-driven race towards more and more cores, the importance of having high-performance, concurrency-friendly building blocks increases. Obtaining these properties increasingly requires a combination of algorithmic engineering and careful attention to systems issues such as internal parallelism, cache alignment, and cache coherency.

At the outset of this research, we hoped to capitalize on the recently introduced hardware transactional memory (HTM) support in Intel's new Haswell chipset, the TSX instructions [1]. Contrary to our expectations, however, we ended up implementing a design that performs well regardless of its use of HTM, and the bulk of our time was not spent dealing with concurrency mechanisms, but rather in algorithm and data structure engineer-

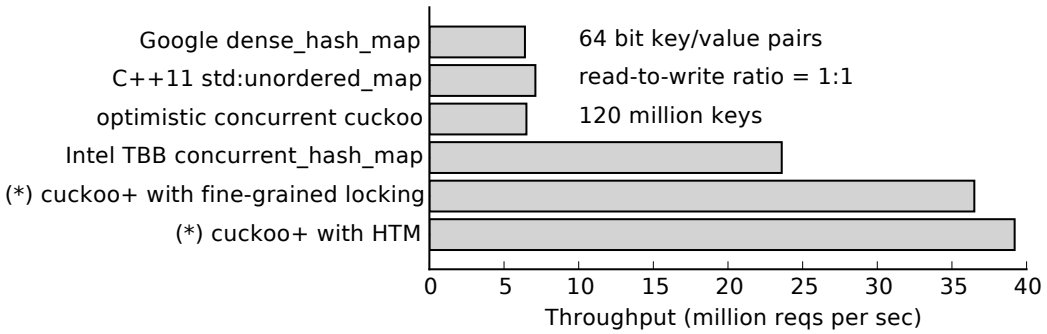


Figure 3.1: Highest throughput achieved by different hash tables on a 4-core machine. (*) are our new hash tables.

ing to optimize for concurrent access. For fast hash tables, HTM's biggest benefit may be to software engineering, by reducing the intellectual complexity of locking, with a modest performance gain as a secondary benefit.

As a result of these efforts, this chapter presents the design and implementation of the first high-performance, multiple-reader/writer hash table that achieves the memory efficiency of multi-way Cuckoo hashing [64]. Most fine-grained concurrent hash tables today store entries in a linked-list with per-bucket locks [2] or Read-Copy-Update (RCU) mechanisms [57, 72]. While often fast, the pointers used in these approaches add high overhead when the key/value items are small. In contrast, our Cuckoo-based design achieves high occupancy with no pointers.

We contribute a design that provides high throughput for multiple writers; prior work we build upon [27] allowed only a single writer, limiting the generality of the data structure. Our design is based on algorithmic engineering of Cuckoo hashing, combined with architectural tuning in the form of effective prefetching, use of striped fine-grained spinlocks, and an optimistic design that minimizes the size of the locked critical section during updates.

The result of these engineering efforts is a solid building block for key-value storage of small objects. On a 16-core machine, our table achieves almost 40 million inserts per second, outperforming the concurrent hash table in Intel's Thread Building Blocks by 2.5x, while using less than half of the memory for 64 bit key/value pairs. Figure 3.1 gives an

example of how our scheme (cuckoo+) outperforms other hash tables with mixed random read/write workloads. Section 3.5 presents a performance evaluation detailing the advantages of this cuckoo-based approach for multicore applications.

3.1 Background and Related Work

This section provides background information on hash tables and concurrency control mechanisms. We conclude with a brief performance evaluation of the effects of naively applying standard concurrency control techniques to several common hash table implementations. These results remind that high-performance concurrency is not trivial: careful algorithm engineering is important regardless of the underlying concurrency control mechanisms, and the algorithmic effects dominate the choice of concurrency mechanism.

3.1.1 Hash Tables

As used in this chapter, a *hash table* provides `Lookup`, `Insert`, and `Delete` operations for indexing all key-value objects. Hash tables do not support retrieval by any key ordering. Popular designs vary in their support for iterating through the hash table in the presence of concurrent modifications; we omit consideration of this feature.

Interface. On `Lookup`, a value is returned for the given key, or "does not exist" if the key cannot be found. On `Insert`, the hash table returns success, or an error code to indicate whether the hash table is too full or the key already exists. `Delete` simply removes the key's entry from the hash table. We focus on `Lookup` and `Insert`, as `Delete` is very similar to `Lookup`.

High-performance single-thread hash tables. As an example of a modern, extremely fast hash table, we compare in several places against Google's `dense_hash_map`, a hash

table available in the Google SparseHash [32] library. Dense hash sacrifices space efficiency for extremely high speed: It uses open addressing with quadratic internal probing. It maintains a maximum load factor of 0.5 with default configurations, and stores entries in a single large array.

C++11 introduces an `unordered_map` implemented as a separate chaining hash table. It has very fast lookup performance, but also at the cost of more memory usage.

The performance of these hash tables does not scale with the number of cores in the machine, as shown in Figure 3.1, because only one writer or one reader is allowed at the same time.

Multiple-reader, single-writer hash tables. As a middle ground between no thread safety and full concurrency, single-writer tables can be extended to permit many concurrent readers. Such designs often use optimistic techniques such as versioning or the read-copy-update (RCU) [57] techniques becoming widely used within the Linux kernel.

Our work builds upon one such hash table design and extends it to support multiple writers. Cuckoo hashing [64] is an open-addressed hashing technique with high memory efficiency and $O(1)$ amortized insertion time and retrieval. As a basis for its hashing, our work uses the multi-reader version of cuckoo hashing from MemC3 [27], which is optimized for high memory efficiency and fast concurrent reads (detailed in Section 3.3.2).

Scalable concurrent hash tables. The Intel Threading Building Blocks library (Intel TBB) [2] provides a `concurrent_hash_map` that allows multiple threads to concurrently access and update values. This hash table is also based upon the classic separate chaining design, where keys are hashed to a bucket that contains a linked list of entries. This design is quite popular for concurrent hash tables: Because a key hashes to one unique bucket, holding a per-bucket lock permits guaranteed exclusive modification while still allowing fine-grained access. Further care must be taken if the hash table permits expansion.

3.1.2 Concurrency Control Mechanisms

As noted earlier, part of our motivation was to explore the application of hardware transactional memory to this core data structure. All concurrent data structures require some mechanism for arbitrating concurrent access, which we briefly list below, focusing on those used in this work.

Locking. Multi-threaded applications take advantage of increasing number of cores to achieve high performance. To ensure thread-safety, multiple threads have to serialize their operations when accessing shared data, often through the use of a critical section protected by a lock.

The simplest form of locking is to wrap a coarse-grained lock around the whole shared data structure. Only one thread can hold the lock at the same time. This tends to be pessimistic, since the thread with the lock prevents any other threads from accessing the shared resource, even if they only want to read the data or make non-conflicting updates.

Another option is to use fine-grained locking by splitting the coarse-grained lock into multiple locks. Each fine-grained lock is responsible for protecting a region of the data, and multiple threads can operate on different regions of the data at the same time. Fine-grained locking can improve the overall performance of a concurrent system. However, it must be carefully designed and implemented to behave correctly without deadlock, livelock, starvation, etc.

Hardware Transactional Memory (HTM). It is often hard to write fast and *correct* multi-threaded code using fine-grained locking. Transactional memory [37] is designed to make the creation of reliable multi-threaded programs easier. Much like database transactions, all shared memory accesses and their effects are applied *atomically*, i.e., they are either committed together or discarded as a group. With transactional memory, threads no

longer need to take locks when accessing the shared data structures held in memory, yet the system will still guarantee thread safety.

Previous experience, implementations and evaluations of HTM include Sun's Rock [12, 22] processor, AMD advanced synchronization family [15, 14], IBM Blue Gene/Q [76] and System Z [42].

Recently, Intel released Transactional Synchronization Extensions (TSX) [1], an extension to the Intel 64 architecture that adds transactional memory support in hardware. Part of the recently-released Intel Haswell microarchitecture, TSX allows the processor to determine dynamically whether threads need to serialize through lock-protected critical sections, and to serialize *only when required*. With TSX, the program can declare a region of code as a transaction. A transaction executes and atomically commits all results to memory when the transaction succeeds, or *aborts* and cancels all the results if the transaction fails (e.g., conflicts occur). We focus on the use of Restricted Transactional Memory (RTM) interface of TSX, which gives the programmer the flexibility to start, commit and abort transactional execution. Intel evaluated TSX for high-performance computing workloads [77], already optimized for parallelism, and showed that TSX provides an average speedup of 1.41x.

3.1.3 Naive use of concurrency control fails

Before making deeper changes, we begin by examining the performance of several hash tables *without* algorithmic optimization, using both naive global locking and using Intel's TSX to optimize this approach. While the poor performance of these approaches is not surprising, their relative simplicity makes them an important starting baseline for understanding further improvements.

Haswell's hardware memory transactions are a best-effort model intended for fast paths. The hardware provides no guarantees as to whether a transactional region will ever successfully commit. Therefore, any transaction implemented with TSX needs a fallback path. The simplest fallback mechanisms is "lock elision": the program executes a lock-protected

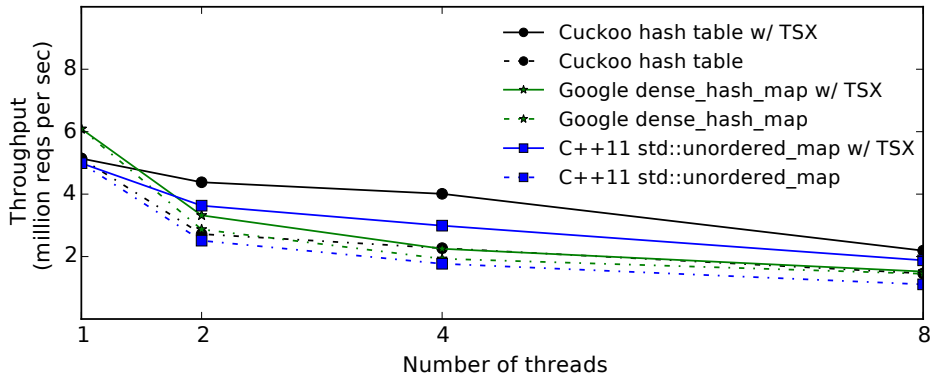


Figure 3.2: `Insert` throughput vs. number of threads for single writer hash tables with and without TSX lock elision. Each thread is pinned to a different hyper-threaded core. 16 million different keys are inserted in each table.

region speculatively as a transaction, and only falls back to use normal locking if the transaction does not succeed. An implementation of TSX lock elision for glibc [73] has been released. It adds a TSX elided lock as a new type of POSIX mutex. Applications linked against this new glibc library automatically have their pthread locks elided.

Lock elision may seem promising for designing a concurrent, multi-writer hash table: multiple threads may be able to update different sets of non-conflicting entries of the hash table as transactions at the same time. Through a set of experiments, we make two observations about TSX lock elision: It outperforms the naive use of a global lock, but it does not ensure that multicore concurrent writes are faster than single-core exclusive access.

We evaluated the `Insert` throughput of the optimistic cuckoo hash table in MemC3, `std::unordered_map` in C++11, and `dense_hash_map` in Google SparseHash [32] library, both with and without TSX lock elision, on a quad-core machine with hyperthreading enabled. All these hash tables allow only one writer at a time, as each `Insert` has to lock the entire table. Global counters were removed in cuckoo hash table and `dense_hash_map` to avoid obvious common data conflicts.

Figure 3.2 shows the results of our experiment. With global pthread locks, each hash table's multi-thread aggregate write throughput is much lower than that of a single thread, due to extensive lock contention. By enabling TSX lock elision, the aggregation write

throughput is higher than that with pthread global locks, but still much lower than the single thread throughput. This is because most transactions fail and abort, forcing the program to take the fallback lock frequently, resulting in sequential behavior. According to Intel Performance Counter Monitor [41], the transactional abort rates are above 80% for all three hash tables with 8 concurrent writers. We will discuss the reasons for transactional aborts and how to reduce the abort rate in Section 3.4.

Through this experiment, we find that naively making a data structure concurrent may harm its performance. Simply applying lock elision using hardware transactional memory could mitigate the performance degradation caused by lock contention, but may not be able to scale up throughput as more cores access the same lock protected data structure.

3.2 Principles to Improve Concurrency

Given that naive application of global locking with or without hardware transactional memory support fails to provide scalable performance, what must be done? In this section we present our design principles to improve the concurrent performance of data structures. Although these principles are general and well known, we state them here to illustrate the framework within which our algorithmic engineering discussed in the next section optimizes for concurrent access in cuckoo hashing. In general, the key to improving concurrency for a data structure is to reduce lock contention. We present three principles to help achieve this reduction:

P1. *Avoid unnecessary or unintentional access to common data.* When possible, make globals thread-local; for example, disable instant global statistics counters in favor of lazily aggregated per-thread counters. These simple optimizations are already included in our results for cuckoo hash table and Google `dense_hash_map` in Figure 3.2. Without them, concurrent performance was much worse.

P2. *Minimize the size and execution time of critical sections.* A promising strategy is to move data accesses out of the critical section whenever possible. As we show in the following section, an optimistic approach can work well here if there are search-like operations that must be performed: Perform the entire search outside of a critical section, and then transactionally execute by only verifying that the found value remains unchanged.

P3. *Optimize the concurrency control mechanism.* Tune the concurrency control implementation to match the expected behavior of the data structure. For example, because the critical sections of our optimized hash tables are all very short, we use lightweight splinlocks and lock striping in the fine-grained locking implementation, and optimize TSX lock elision to reduce transactional abort rate when applying it to the coarse-grained locking implementation.

By following these principles, data structures can reduce the possibility of multiple threads attempting to access data protected by a shared lock or within a same transactional region, thus improve the concurrent performance with either fine-grained or coarse-grained locking. We show how to apply these principles to the design of a concurrent cuckoo hash table in the next two sections, to greatly improve multi-threaded read/write throughput.

3.3 Concurrent Cuckoo Hashing

We now present the design of a multi-reader/multi-writer cuckoo hash table that is optimized for fast concurrent writes. By applying the principles previously described, our resulting design achieves high and scalable multi-threading performance for both read- and write-heavy workloads.

We begin by presenting the basic operation of cuckoo hashing [64], followed by the multiple-reader/single-writer version that we build upon to create our final solution [27].

3.3.1 Cuckoo Hashing

Cuckoo hashing [64] is an open-addressed hash table design. All items are stored in a large array, with no pointers or linked lists. To resolve collisions, two techniques are used: First, items can be stored in one of two buckets in the array, and they can be moved to their other location if the first is full. Second, in common use, the hash buckets are multi-way set associative, i.e., each bucket has B "slots" for items. $B = 4$ is a common value in practice.[1] A lookup for `key` proceeds by computing two hashes of `key` to find buckets b_1 and b_2 that could be used to store the key, and examining all of the slots within each of those buckets to determine if the key is present. A basic "2,4-cuckoo" hash table (two hash functions, four slots per bucket) is shown in Figure 3.3.

A consequence of this is that `Lookup` operations are both fast and predictable, always checking $2B$ keys.

To `Insert` a new key into the table, if either of the two buckets has an empty slot, it is then inserted in that bucket; if neither bucket has space, a random key from one candidate bucket is displaced by the new item. The displaced item is then relocated to its own alternate location, possibly displacing another item, and so on, until a maximum number of displacements is reached. If no vacant slot is found, the hash table is considered too full to insert and an expansion process is scheduled.

We call the sequence of displaced keys in an `Insert` operation a *cuckoo path*, as illustrated in Figure 3.3. Write performance of cuckoo hashing degrades as the table occupancy increases, since the cuckoo path length will increase, and more random reads/writes are needed for each `Insert`.

[1] Without set-associativity, basic cuckoo hashing allows only 50% percent of the table entries to be occupied before unresolvable collisions occur. It is possible to improve the space utilization to over 90% by using 4-way (or higher) set associative hash table [25].

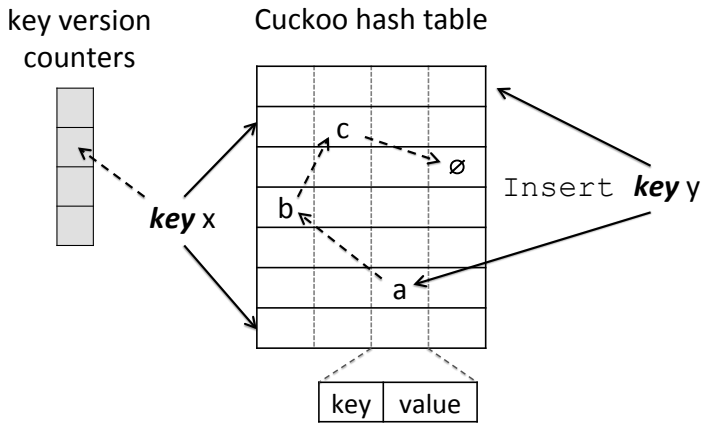


Figure 3.3: Cuckoo hash table overview: Each key is mapped to 2 buckets by hash functions and associated with 1 version counter. ∅ represents an empty slot. "$a \rightarrow b \rightarrow c \rightarrow \emptyset$" is a cuckoo path to make one bucket available to insert key y.

3.3.2 Prior Work in Concurrent Cuckoo

Basic cuckoo hashing does not support concurrent access. Our work builds upon the two major prior approaches to concurrent cuckoo hashing: Herlihy's lock-striped approach [38], and the optimistic cuckoo with separated path discovery and item movement from MemC3 [27]. Our resulting design realizes the strengths of each: The low space overhead of MemC3's approach and the concurrent writer support of Herlihy's approach.

Our starting point was MemC3's table, which used three building blocks:

- *To eliminate reader/writer false misses, change the order of the basic cuckoo hashing insertions.* Allow concurrent reads and cuckoo movement by moving "holes" backwards along the cuckoo path instead of moving "items" forward along the cuckoo path. This ensures that an item can always be found by a reader thread; if it is undergoing concurrent cuckoo movement, it may be present twice in the table, but will never be missing.

  Providing this property requires separating the process of searching for a cuckoo path from using it: find the empty slot, and *then* use the path. As we show, this has a second benefit: This searching process can be moved outside of the critical section.

- *Implement efficient concurrency control by using lock striping.* Lock striping [38] uses a smaller vector of locks (or, in MemC3, version counters) that each maps to a set of items in the hash table. To lock a bucket, a writer thread computes the lock stripe entry corresponding to the bucket and locks that entry. By using reasonable size lock tables, such as 1K-8K entries, the locking can be both very fine-grained and low-overhead.

- *Allow reads to be performed with no cache line writes by using optimistic locking [47].* Instead of locking for reads, the hash table uses a lock-striped version counter associated with the buckets, updates it upon insertion or displacement, and looks for a version change during lookup.

By using these techniques with *only* version counters and a simple global lock for writers, MemC3 provided substantial gains for read-intensive workloads, but still performed poorly for write-heavy workloads. Unfortunately, the basic scheme used in MemC3 was not obviously amenable to fine-grained locking:

1. The cuckoo path can be very long. Grabbing a few hundred locks in the right order to avoid deadlock and livelock is tricky. There is also a nontrivial probability that a path becomes invalid, and the execution of `Insert` needs to restart, further complicating locking, increasing the risk of livelock, and harming performance.

2. The `Insert` procedure for optimistic concurrent cuckoo hashing in MemC3 [27] involves nested locks if fine-grained locking is implemented, which can easily cause deadlocks.

Algorithm 2 MemC3 Cuckoo Insert Procedure.
Region between dashed lines is the largest possible critical section.

```
1: function INSERT(h, x)                          ▷ Insert key x to table h
2:     b1, b2 ← two buckets mapped by key x
3:     LOCK(h)
       - - - - - - - - - - - - - - - - - - - - - - - -
4:     if ADD(h, b1, x) or ADD(h, b2, x) then
5:         UNLOCK(h); return true
6:     if path ←SEARCH(h, b1, b2) then
7:         EXECUTE(h, path)
       - - - - - - - - - - - - - - - - - - - - - - - -
8:         UNLOCK(h); return true
9:     UNLOCK(h); return false
```

3.3.3 Algorithmic Optimizations

3.3.3.1 Lock After Discovering a Cuckoo Path

In MemC3 cuckoo hashing, each `Insert` operation locks the hash table at the very beginning of the process, and releases the lock after the insertion completes. The separated phases of search and execution of the cuckoo path are all protected by the lock within one (big) critical section.

To reduce the size of critical sections, our first optimization was to search for an empty slot before acquiring the lock, then only lock the table when displacing the items along the cuckoo path and inserting the new item. In this way, multiple `Insert` threads can look for their cuckoo paths at the same time without interfering with each other. Inserts are still serialized, but the critical section is smaller.

Algorithm 2 shows the basic `Insert` procedure that allows concurrent reads. ADD(h, b, x) tries to insert key x to bucket b, returns $true$ on success or $false$ if the bucket is full. SEARCH(h, b_1 b_2) searches for a cuckoo path that makes either bucket b_1 or b_2 available to insert a new item. EXECUTE(h, $path$) moves items backwards along the cuckoo path, and then inserts key x to the bucket made available. The critical section of this algorithm is the whole process. When the table occupancy is high, this may involve hundreds of bucket reads to search for a cuckoo path, followed by hundreds of item

Algorithm 3 Cuckoo Insert – lock after discovering a path.
Region between dashed lines is the largest possible critical section.

```
 1: function INSERT(h, x)                                  ▷ Insert key x to table h
 2:     b1, b2 ← two buckets mapped by key x
 3:     for i ← 1,2 do
 4:         if AVAILABLE(h, bi) then                       ▷ if bi has an empty slot
 5:             LOCK(h)
 6:             if ADD(h, bi, x) then
 7:                 UNLOCK(h); return true
 8:             UNLOCK(h)
 9:     while path ←SEARCH(h, b1, b2) do
10:         LOCK(h)
            - - - - - - - - - - - - - - - - - - - - - - - -
11:         if VALIDATE_EXECUTE(h, path) then
            - - - - - - - - - - - - - - - - - - - - - - - -
12:             UNLOCK(h); return true
13:         UNLOCK(h)
14:     return false
```

displacements along that path, during which all `Insert` operations of other threads are blocked.

Algorithm 3 shows our new `Insert` procedure. The lock is acquired only when doing the actual writes to the hash table. As the search phase is not protected by the lock, there exists a potential race condition: After one thread reads a bucket to extend its cuckoo path, another thread can write to the same bucket and cause the first thread to read corrupted data. Therefore, `Insert` must re-check if the related entries have been modified before each item displacement in the execution phase, which is handled by VALIDATE_EXECUTE(h, $path$). If the existing path becomes invalid, it restarts and looks for a new path. Each displacement relocates only one item to its alternate bucket, so there is no undo needed if execution aborts. We omit the steps to check if key x already exists in both Algorithm 2 and 3, which should be proceeded within each critical section.

To summarize, each `Insert` optimistically searches for a cuckoo path, displacing items along the path with lock protection. Execution terminates at the end of the path or if the path becomes invalid (and then `Insert` restarts). We can estimate the probability of a

cuckoo path become invalid after being discovered, which is the probability that a path of one writer overlaps with paths of other writers.

Let N denote the number of entries in the hash table, L ($\ll N$) denote the maximum length of a cuckoo path, and T denote the number of concurrent writers. A cuckoo path has the highest possibility of overlapping with others when all the T paths are at their maximum length L. For a cuckoo path with length L, the probability that it does not overlap with another cuckoo path with length L is

$$P = \frac{\binom{N-L}{L}}{\binom{N}{L}} = \prod_{i=0}^{L-1} \frac{N-L-i}{N-i}. \tag{3.1}$$

The probability that the cuckoo path overlaps with at least one of other $(T-1)$ paths is

$$P_{invalid\_max} = 1 - P^{T-1} = 1 - \prod_{i=0}^{L-1} \left(\frac{N-L-i}{N-i}\right)^{(T-1)}. \tag{3.2}$$

Because $i \ll N$, we can assume $\frac{N-L-i}{N-i} \approx \frac{N-L}{N}$, so that

$$P_{invalid\_max} \approx 1 - \left((N-L)/N\right)^{L(T-1)}. \tag{3.3}$$

For example, the maximum length of a cuckoo path in MemC3 is $L = 250$. Suppose $N = 10$ million, $T = 8$, then $P_{invalid} < 4.28\%$. This upper bound assumes all paths are at maximum length, which occurs only rarely; the expected probability is much lower. It is, however, non-negligible. We apply further algorithmic optimizations next to reduce the odds of such a failure by several orders of magnitude.

3.3.3.2 Breadth-first Search for an Empty Slot

Basic cuckoo hashing searches for an empty slot using a greedy algorithm: if the current bucket is full, a random key is "kicked out" to its alternate location, and possibly kicks out another random key there, until a vacant position is found. Each bucket touched by

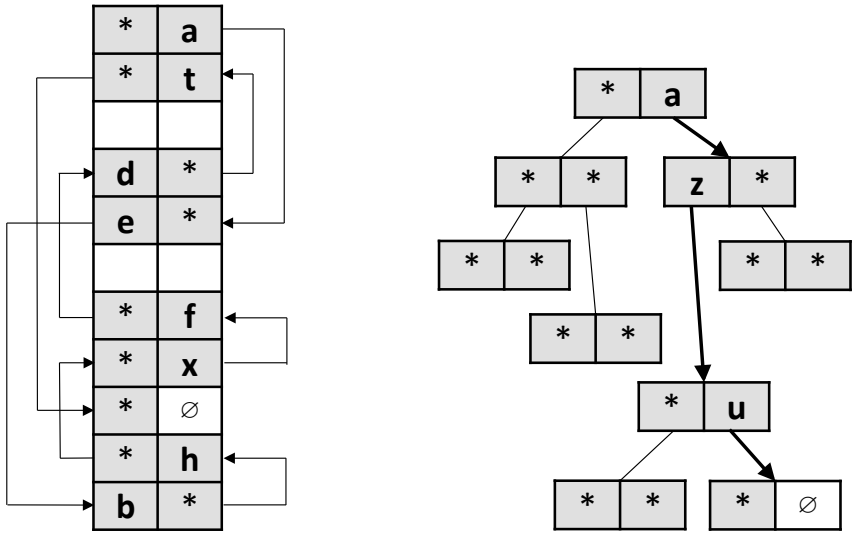
the process is a part of the cuckoo path. As table occupancy grows, the average length of cuckoo paths increases, because it needs to examine more buckets to find an empty slot. It may require hundreds of displacements for one `Insert`, which greatly slows down the performance.

A cuckoo hash table can be viewed as an undirected graph called a *cuckoo graph*, which has a vertex for each bucket, and an edge for each key in the table, connecting the two alternative buckets of the key. The "random displacements" scheme used by basic cuckoo hashing to look for an empty slot is thus a random *depth-first search* (DFS) of the graph. To reduce the number of item displacements and the size of critical sections, we use *breadth-first search* (BFS) instead. Each slot in a bucket is considered as a possible path, and extends its own path to alternate buckets in the same way.

Figure 3.4 shows an example of the two searching schemes in a 2-way set-associative hash table. Both schemes examine 18 slots (9 buckets) to find an empty slot in the search with no item actually moved. Figure 3.4a is the traditional searching scheme where each time only one random key is displaced. The cuckoo path discovered is $a \rightarrow e \rightarrow b \rightarrow h \rightarrow x \rightarrow f \rightarrow d \rightarrow t \rightarrow \varnothing$. Figure 3.4b uses BFS to look for an empty slot. While the number of examined slots are same, the BFS cuckoo path is $a \rightarrow z \rightarrow u \rightarrow \varnothing$, which is much shorter.

The prior work on MemC3 used an optimization of tracking two cuckoo paths in parallel, completing when either found an empty slot, but still used a DFS strategy. This strategy, in general, reduced the expected length of a cuckoo path by a factor of two. In contrast, the BFS strategy we present here reduces the expected length to a *logarithmic* factor.

Let B denote the set-associatitivity of the hash table, M denote the maximum number of slots to be examined when looking for an empty slot before declaring the table is full, L_{BFS} denote the maximum length of the cuckoo path. The search process expands to two BFS tree rooted by the two alternative buckets of the key to be inserted. Each tree has at



(a) Random displacements (b) Breadth-first search

Figure 3.4: Search for an empty slot by `Insert` in a 2-way set-associative hash table. Left(3.4a) is the traditional approach, right(3.4b) is our approach. Slots in gray are examined before the empty slot is found. Alphabet letters are keys selected to be moved to their alternate locations along the cuckoo path represented by the arrows (→).

most $M/2$ slots. Therefore,

$$B + B^2 + B^3 + \cdots + B^{L_{BFS}} \geq M/2, \tag{3.4}$$

which gives us

$$L_{\text{BFS}} = \lceil \log_B (M/2 - M/(2B) + 1) \rceil. \tag{3.5}$$

As used in MemC3, $B = 4,\ M = 2000$. With two-way DFS, the maximum number of displacements for a single `Insert` is 250, whereas with $L_{\text{BFS}} = 5$.

This optimization is key to reducing the size of the critical section: While the total number of slots examined is still M, this is work that can be performed without a lock held. With BFS, however, at most five buckets must be examined and modified with the lock actually held, reducing both the duration of the critical section and the number of cache lines dirtied while doing so.

Shorter cuckoo paths also reduce the chance of a path becoming invalid (and of transactional aborts). Based on Eq. 3.3, with $L_{\text{BFS}} = 5$, and the same settings of the example at the end of Section 3.3.3.1 , the new worst-case $P_{invalid} < 1.75 \times 10^{-5}$ — an extremely rare event.

Prefetching. BFS provides a second benefit: because the schedule of buckets to visit is predictable, we can prefetch buckets into cache before they are accessed to reduce the cache-miss penalty. In the *cuckoo graph*, each alternative bucket of the keys in the current bucket are considered *neighbors* of that bucket. BFS scans all neighbors of a bucket to extend the cuckoo path. Before scanning one neighbor, the processor can load the *next_neighbor* in cache, which will be accessed soon if no empty slot is found in the current neighbor. This cannot be done with the traditional DFS approach, because the next bucket location is unknown until one key in the current bucket is "kicked out".

3.3.3.3 Increase Set-associativity

As discussed in Section 3.3.1, higher set-associativity improves space utilization. Then cuckoo hash table in MemC3 is 4-way set-associative, which achieves 95% maximum load factor, and high performance for read-intensive workloads.

The impact of set-associativity on the read and write performance of cuckoo hashing is two-fold:

- Higher set-associativity leads to *lower* read throughput, since each `Lookup` must scan up to $2B$ slots from two buckets in an B-way set-associative hash table. If a bucket fits in a cache line, then the read throughput would not be affected too much.

- Higher set-associativity may *improve* write throughput, because each `Insert` can read fewer random buckets (with fewer cache misses) to find an empty slot, and needs fewer item displacements to insert a new item. However, the set-associativity cannot

be too high, since a `Lookup` is required to check if the new key already exits in the hash table before each `Insert`, which becomes slower as set-associativity increases.

To achieve a good balance between read- and write-heavy workloads, we use a 8-way set-associative hash table. Section 3.5 evaluates the performance with different set-associativities and different workloads. Our choice of 8-way associativity may require reading more than one cache line per bucket, but this extra cost is offset by the fact that the two lines can be fetched together, costing only memory bandwidth, not latency, and that sequential memory reads are substantially faster because they typically hit in the DRAM row buffer.

3.3.4 Fine-grained Locking

Fine-grained locking is often used to improve concurrency. However, it is non-trivial to implement fine-grained per-bucket locking for traditional cuckoo hashing. There are high deadlock and livelock risks.

In basic cuckoo hashing, it is not known before displacing the keys how many and which buckets will be modified, because each displaced key depends on the one previously kicked out. Therefore, standard techniques to make `Insert` atomic and avoid deadlock, such as acquiring all necessary locks in advance, are not obviously applicable. As noted earlier, simply using the optimization of finding the path in advance was not enough to solve this problem because of lingering locking complexity issues.

By reducing the length of the cuckoo path and reordering the locking procedure, our optimizations make fine-grained locking practical. To do so, we go back to the basic design of lock-striped cuckoo hashing and maintain an actual lock in the stripe *in addition* to the version counter (our lock uses the high-order bit of the counter). Here we favor spinlocks using compare-and-swap over more general purpose mutexes. A spinlock wastes CPU cycles spinning on the lock while other writers are active, but has low overhead, particularly

for uncontended access. Because the operations that our hash tables support are all very short and have low contention, very simple spinlocks are often the best choice.

To `Insert` each new key-value pair, there is at most one new item inserted and four item displacements. Each insert or displacement involves exactly two buckets. The `Insert` operation only locks the pair of buckets associated with ongoing insertion or displacement, and releases the lock immediately after it completes, before locking the next pair. Locks of the pair of buckets are ordered by the bucket id to avoid deadlock. If two buckets share the same lock, then only one lock is acquired and released during the process. In summary, a writer must only lock at most five (usually fewer than three) pairs of buckets sequentially for an `Insert` operation.

Although there is a small chance that any cuckoo insert will abort because of other concurrent inserts, it is likely to succeed on a re-try. It is worth noting that this design only avoids livelock probabilistically. A writer thread that encounters excessive insert aborts *could* pessimistically acquire a full-table lock by acquiring each of the 2048 locks in the lock-striped table, but we have never observed a condition where this would be warranted.

The combination of these techniques results in a cuckoo hash table that (i) retains high memory efficiency (the efficiency of the basic table plus the small additional lock-striping table), (ii) permits highly concurrent read-write access, and (iii) has a minimally-sized critical section that reads and dirties few cache lines while holding the lock or executing under hardware transactional memory.

3.4 Optimizing for Intel TSX

As shown in Section 3.1.3, naive use of TSX lock elision to hash tables with a global lock does not provide high multi-threaded throughput. The key to improving concurrent performance is to reduce the "transactional abort rate." In the Haswell implementation of TSX, the underlying hardware transactional memory system uses tags in the L1 cache to

track the read- and write-sets of transactions at a granularity of a cache line. Transactions abort for three common reasons:

1. *Data conflict on a transactionally accessed address.* A transaction encounters a conflict if a cache line in its read-set is written by another thread, or if a cache line in its write-set is read or written by another thread.

2. *Limited resources for transactional stores.* A transaction will abort if there is not enough space to buffer its reads and writes in cache. Current implementations can track only 16KB of data.

3. *TSX-unfriendly instructions.* Several instructions (e.g., XABORT, PAUSE) and system calls (e.g., mmap) cause transactions to abort.

For high performance, the program must minimize transactional aborts. From the first two causes, we draw several conclusions about general issues with transactional aborts:

- Transactions that touch more memory are more likely to conflict with others, as well as to exceed the L1-cache-limited capacity for transactional reads and writes.

- Transactions that take longer to execute are more likely to conflict with others.

- Sharing of commonly-accessed data, such as global statistics counters, can greatly increase conflicts.

- Because the hardware tracks reads and writes at the granularity of a cache line, false sharing can create transactional conflicts even if no data appears to be shared.

The observant reader will no doubt note that many of these same issues arise in cache-centric performance optimizations. Our solutions are similar but not identical. To address these issues and improve the multi-threaded concurrent performance of cuckoo hashing with coarse-grained locking and TSX lock elision enabled, we just need to follow principle

P1 and P2 presented in Section 3.2, which are detailed in Section 3.3. Our algorithmic optimizations can significantly reduce the size of the transactional region in a cuckoo `Insert` process from hundreds of bucket reads and writes to only a few bucket writes, which greatly reduces the transactional abort rate caused by data conflicts or limited transactional stores.

The third cause of transactional abort indicates that a program should *minimize the occurrence of TSX-unfriendly instructions within transactional regions.* A common example is if dynamic memory allocation must invoke a system call such as `brk`, `futex`, or `mmap`. While our implementation of Cuckoo hashing does not do this, we observed this problem when testing TSX using chained hashing and Masstree [56]. It is therefore useful to preallocate structures that may be needed inside the transactional region. If they are not used, one can simply store them in a per-thread cache and use for a subsequent transaction (or preallocate and free if using a malloc that already does this, such as `tcmalloc`). This is an application of principle P3.

Further, we *use a tuned version of TSX lock elision that matches the expected behavior of the data structure.* The generic glibc version of TSX lock elision for pthread mutexes can be improved substantially if the application's transactional behavior is known in advance, as is the case for our optimized cuckoo hash table, in which every transaction is small. This is another application of principle P3. The following subsection details our implementation of TSX lock elision.

3.4.1 Optimized TSX lock elision

Intel TSX provides two interfaces for transactional memory. The first is Hardware Lock Elision (HLE), a legacy compatible instruction set extension that allows easy conversion of lock-based programs to transactional programs. The second mode is Restricted Transactional Memory (RTM), a new instruction set interface with more complete transactional memory implementation. It provides three explicit instructions—XBEGIN, XEND, and XABORT—for programmers to start, commit, and abort a transactional execution, respec-

tively. RTM is not backwards compatible, but it allows much finer control of the transactions than HLE. We focus on the use of RTM since it is more powerful and flexible than HLE and can serve as a upper bound of the performance improvements one may realize through TSX.

The released TSX RTM lock elision implementation for glibc [73] can be improved by specializing it for our hash tables. As a generic implementation, it is designed to work well for any mix of transactions, including the case of a mix of short transactions that must potentially coexist with long-running ones. In contrast, in the hash table workloads, all transactions are short. We further observed that the generic version misuses the EAX abort status code for RTM and takes the fallback lock too frequently. This causes performance to suffer because whenever a fallback lock is taken by one core, *all* the other cores have to abort their concurrent transactions.

We implemented our own TSX elision wrapper around existing lock functions. It is optimized for short transactions and elides the lock more aggressively. Figure 3.5 shows the implementation of our RTM elision wrapper, a modified version of the released glibc one [73]. It is a small library separated from glibc pthread, and thus does not require building a new glibc library. Its fallback lock can be of any type, including the custom spinlocks we use for cuckoo hashing.

Implementation details. `_xbegin()`, `_xabort()`, and `_end()` calls are wrappers around the special instructions that begin, abort, and commit the transaction. `_xbegin()` returns `_XBEGIN_STARTED` if the transaction begins successfully. `_ABORT_RETRY` is an EAX abort status code which indicates the transaction may succeed on a retry. We found that even if `_ABORT_RETRY` is not set in the EAX register, the transaction may succeed still on a retry. Whenever `_ABORT_RETRY` is not set, however, the glibc TSX lock elision aborts the transaction and takes the fallback lock immediately, forcing all other concurrent transactions to

```
void elided_lock_wrapper(lock) {
    xbegin_retry = 0; abort_retry = 0;
    while (xbegin_retry < _MAX_XBEGIN_RETRY) {
        if (status=_xbegin() == _XBEGIN_STARTED) {  // Start transaction
            if (lock is free)           // Check lock and put into read-set
                return;                 // Execute in transaction
            _xabort (_ABORT_LOCK_BUSY); // Abort transaction as lock is busy
        }
        if (!(status & _ABORT_RETRY)) { // Transaction may not succeed on a retry
            if (abort_retry >= _MAX_ABORT_RETRY) // There is no chance for a retry
                break;
            abort_retry ++ ;
        }
        xbegin_retry ++;
    }
    take fallback lock;
}

void elided_unlock_wrapper(lock) {
    if (lock is free)
        _xend();  // Commit transaction
    else
        unlock lock;
}
```

Figure 3.5: Optimized TSX lock elision

abort. Instead, we always retry several times before taking the fallback lock (using more retries if `_ABORT_RETRY` is set).

3.5 Evaluation

In this section, we investigate how the proposed techniques and optimizations contribute to the improvements of read and write performance in cuckoo hashing.

Platform. Most experiments (except Figure 3.8) run on a 4-core Haswell-microarchitecture Intel i7-4770 at 3.4GHz. This is the highest core count currently available with TSX support. The L1 D-cache is 32KB; the L2 cache is 256KB, the L3 cache is 8MB. The machine is equipped with 16GB of DDR3 SDRAM.

Method and Workloads. 8 byte keys and 8 byte values are used for most experiments. The default cuckoo hash table is 8-way set-associative with $2^{27} = 134,217,728$ slots, which uses about 2 GB memory. Each bucket has all the keys come first and then the values, and fits exactly two cache lines: one for 8 keys and another for 8 values. We evaluate different set-associativities in Section 3.5.3 and key-value sizes in Section 3.5.4.

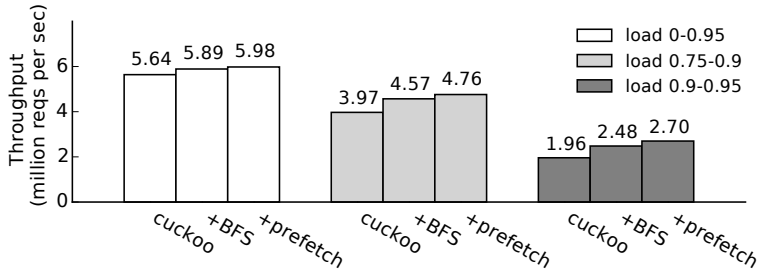
We focus on the performance benefit from our optimizations and TSX support for workloads with concurrent writes by measuring the aggregate throughput of multiple threads accessing the same hash table. We focus on three workloads: *a*) 100% `Insert`, *b*) 50% `Insert` and 50% `Lookup`, and *c*) 10% `Insert` and 90% `Lookup`.

Each experiment first creates an empty cuckoo hash table and then fills it to 95% capacity, with random mixed concurrent reads and writes as per the specified insert/lookup ratio. Because Cuckoo hashing slows down as the table fills (more items must be moved), we measure both overall throughput and throughput for certain load factor intervals (e.g., empty to 50% full). Each data point in the graphs of this section is the average of 10 runs. We observed that the performance is always stable, so we do not include error bars in the graphs.

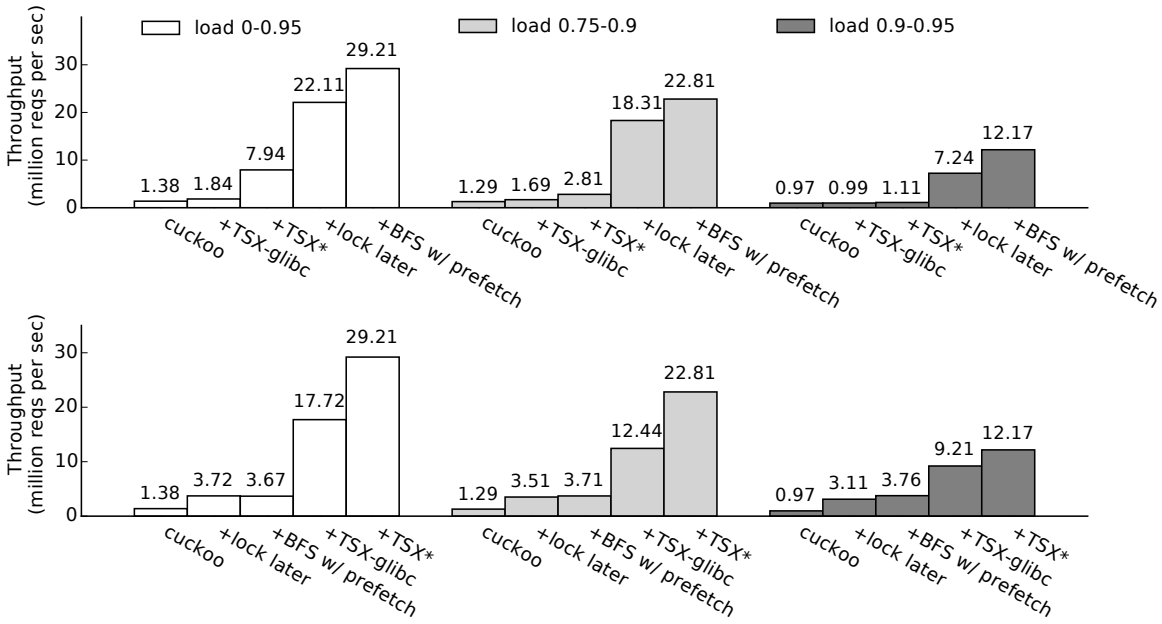
3.5.1 Factor Analysis of Insert Performance

This experiment investigates how much our optimizations and the use of Intel TSX improve the `Insert` performance of cuckoo hashing. We break down the performance gap between basic optimistic cuckoo hashing and our optimized concurrent cuckoo hashing. We measure different hash table designs with the `Insert`-only workload starting from the basic cuckoo and adding optimizations cumulatively as follows:

- **cuckoo**: The optimistic concurrent multi-reader/single-writer cuckoo hashing used in MemC3 [27]. Each `Insert` locks the whole hash table.
- **+lock later**: Lock after discovering a cuckoo path.



(a) Single thread `Insert` performance (all locks disabled)

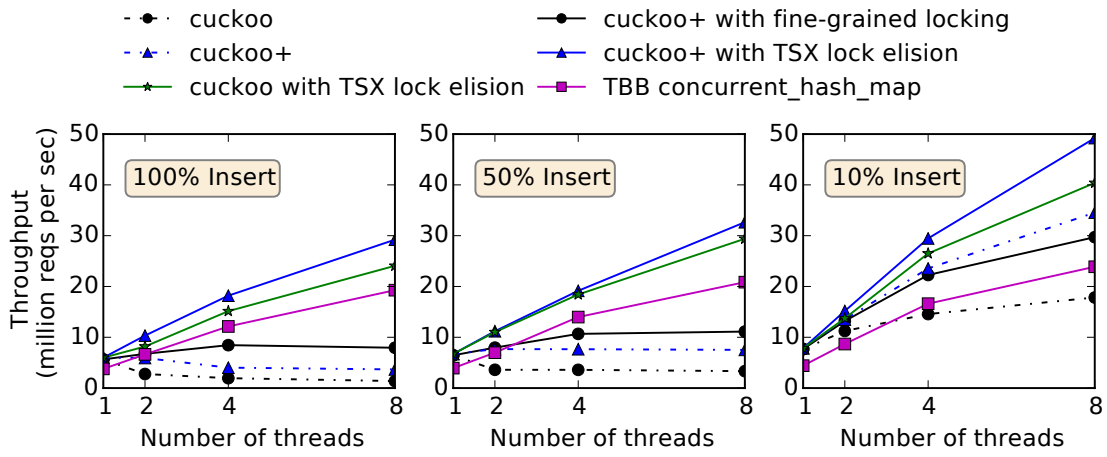


(b) Aggregate `Insert` performance of 8 threads, with locking

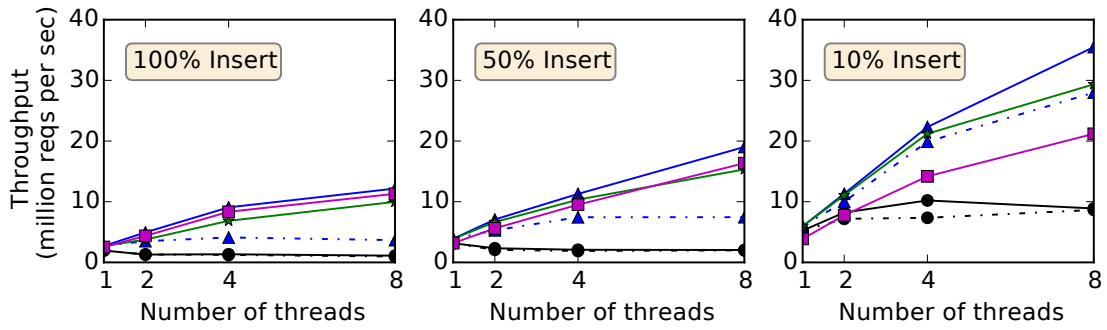
Figure 3.6: Contribution of optimizations to the hash table `Insert` performance. Optimizations are cumulative.

- **+BFS**: Look for an empty slot by breadth-first search.
- **+prefetch**: Prefetch the next bucket into cache.
- **+TSX-glibc**: Use the released glibc TSX lock elision [73] to support concurrent writers.
- **+TSX***: Use our TSX lock elision implementation that is optimized for short transactions (Section 3.4.1) instead of TSX-glibc.

Single-thread `Insert` performance is shown in Figure 3.6a. All locks are disabled, so "lock later" and "TSX" do not apply here. At high load factors, BFS improves single-



(a) Average throughput to fill the table from 0% to 95% occupancy.



(b) Average throughput at high table occupancy (0.9% – 0.95%).

Figure 3.7: Throughput vs. number of threads. "cuckoo" is the optimistic cuckoo hashing used in MemC3, "cuckoo+" is cuckoo with optimizations in Section 3.3.3. TSX lock elision is the optimized version in Section 3.4.1. The cuckoo hash table is 2 GB with ~ 134.2 million slots. Table occupancy is for cuckoo hashing only. TBB concurrent_hash_map is inserted with the same number and size of key-value pairs, with 2× to 3× more memory used than cuckoo hash table.

thread write performance by ~ 26%, and data prefetching further increases the throughput by ~ 9%.

At low table occupancy, these optimizations are less important. In most cases, there are plenty of empty slots, and so no keys need to be moved. Further, when the cuckoo paths are all short, there is no savings in item motion to outweigh the slightly increased search cost of BFS over DFS. At high occupancy, BFS substantially reduces the number of item displacements, and prefetching is more useful because more buckets need to be evaluated as insertion candidates.

Multi-thread insert performance is shown in Figure 3.6b, measured by aggregating the throughput from 8 threads accessing the same hash table. A global lock is used for each `Insert` in the optimistic cuckoo hashing. Due to lock contention, the multi-threaded aggregate throughput of the optimistic cuckoo hashing is much lower than the single-thread throughput. The performance difference between the original optimistic cuckoo hashing scheme and optimized cuckoo hashing with TSX lock elision is roughly 20×.

To understand the source of these benefits, the upper plot of Figure 3.6b shows the optimization sequence with lock elision enabled first and algorithmic optimizations applied later. With no algorithmic optimizations, using the customized TSX* elision improves overall throughput by ~ 4.3× over basic TSX lock elision. Comparing the top and bottom figures, when TSX* is applied *after* our algorithmic changes, it still improves throughput by almost 2x. This demonstrates the importance of using TSX in a way that is well-matched to the properties of the transactions it is handling. The improvements from fine-grained locking (not shown) are similar to those from applying TSX*, but slightly slower.

Simply reducing the size of the critical section *without* TSX or fine-grained locking results in only modest improvements (bottom graph, far left): from 1.38 to 3.7 million operations per second. However, once the system is capable of supporting fine-grained concurrent access, the improvement from algorithmic improvements is large (top graph, far left): from 7.94 to 29.2 million operations per second.

High performance is a consequence of both sufficiently fine-grained concurrency *and* data structures optimized to make that concurrency efficient. Neither of these optimizations alone was able to achieve more than 8 million operations per second, but they combine to achieve almost 30 million. Of particular note was that the algorithmic improvements needed here were concurrency-specific: Without concurrency, for example, the BFS changes were performance-neutral, but with fine-grained locking, BFS increased performance by over 30%.

This latter conclusion is particularly true under high contention: The rightmost graphs in the figure show the performance improvements for the highly-loaded portion of the hash table fill, growing from 90% to 95% (a load factor that might occur with a heavy insert/delete workload). In this case, the performance gains of the algorithmic engineering are even more important: The high contention means that TSX alone encounters frequent aborts, only improving performance by about 10%. The algorithmic optimizations then provide a roughly 11x improvement.

3.5.2 Multi-core Scaling Comparison

This section evaluates hash table performance under an increasing number of cores, comparing both our original and optimized table, and also the Intel TBB [2] `concurrent_hash_map` for comparison. We initialize the TBB table with the same number of buckets and key-value type, then operate with the same workloads.

Cuckoo+ scales well as the number of cores increases, on both our 4-core Haswell machine (Figure 3.7), as well as when using fine-grained locking on a 16-core Xeon machine without TSX support (Figure 3.8). On the Haswell machine, the performance increase from 4 to 8 cores is slightly lower than up to 4 cores because there are only 4 physical cores.

In comparison, the basic *optimistic cuckoo hash table* scales poorly for a write-heavy workload, even using TSX lock elision. As shown in Figure 3.7, its total `Insert` throughput actually drops as more cores are used, except for the read-heavy workloads (rightmost graphs) for which its optimistic design works well. Notably, however, even under 10% inserts, cuckoo+ still substantially outperforms optimistic cuckoo.

The fine-grained locking version of Cuckoo+ also scales well for all workloads. Its absolute performance is up to 20% less than the TSX-optimized version, however, suggesting that there is a non-negligible benefit from hardware support.

To put these numbers in perspective, we also compare against the Intel Thread Building Blocks hash table. This comparison is slightly unfair: TBB supports concurrent iteration

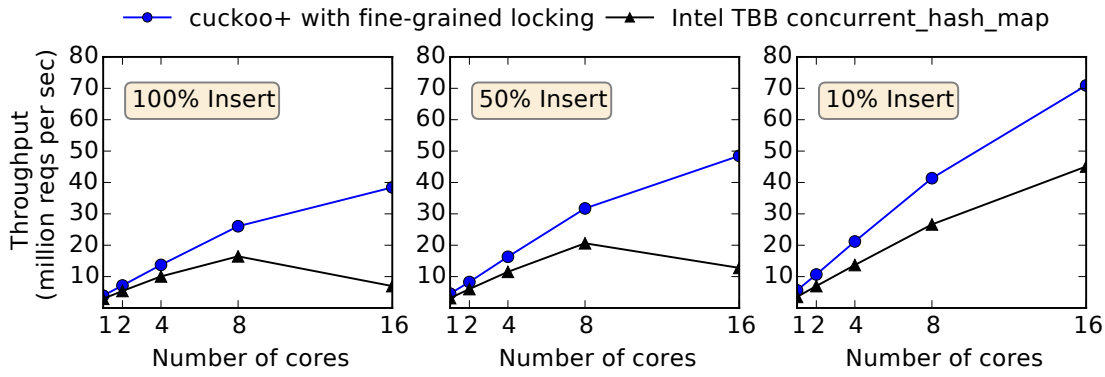


Figure 3.8: Overall throughput vs number of cores. On a 16-core machine without TSX support.

and other features that our hash table does not, but at a high level, it demonstrates both that our table's performance is good (it outperforms TBB substantially), particularly for read-intensive workloads, and that Cuckoo+ retains the memory efficiency advantages of the core Cuckoo design: It uses $2-3\times$ less memory for these small key-value objects, occupying only about 2GB of DRAM versus TBB's 6GB.

The results in Figure 3.8 show that these results also extend to larger machines, using a dual-socket Xeon server with 16 total cores, each a bit slower than those in the Haswell machine. Neither server has perfect speedup after 8 cores—memory operations begin to traverse the QPI interconnect between the sockets—but Cuckoo+ continues to scale for write-heavy workloads where TBB scales only for read-heavy workloads.

3.5.3 Set-associativity and Load Factor

In this section, we evaluate the impact of set-associativity and load factor on cuckoo hashing performance, using the optimized cuckoo hashing with TSX lock elision. The experiments use the same workloads and hash table with same number of slots as before.

Figure 3.9 shows the aggregate `Lookup`-only throughput of 8 threads for 4- 8- and 16-way set associative hash tables, all at 95% table occupancy. As expected, lower associativity improves throughput, because each reader needs to check fewer slots in order to find the key. Each `Lookup` in a 4-way set-associative hash table needs at most two cache line reads

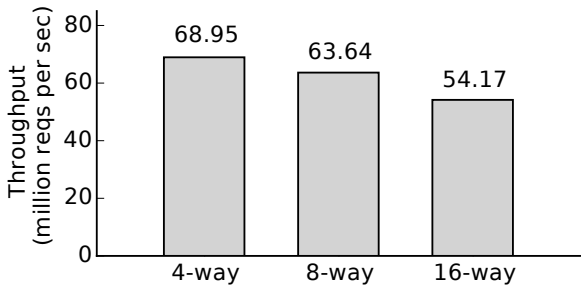


Figure 3.9: 8-thread aggregate `Lookup` throughput of hash tables with different set-associativities at 95% occupancy. Use optimized cuckoo hashing with TSX lock elision.

to find the key and get the value. Each `Lookup` in a 8-way set-associative hash table needs at most two cache line reads to find the key and one more cache line read to get the value. Each `Lookup` in a 16-way set-associative hash table needs at most four cache line reads to find the key and one more cache line read to get the value.

Figure 3.10 shows the 8-thread aggregate throughput of table with different set-associativities, for different workloads at different table occupancy. Write performance degrades as the table occupancy increases, since an `Insert` operation has to read more buckets to find an empty slot, and needs more item displacements to insert the new key.

The load factor is important in this discussion because of the different use modes for hash tables: Some applications may simply fill the table in one go and then use it (perhaps modifying inserted values but not deleting keys), thus caring more about total insert rate. Others may issue inserts and deletes to a table at high occupancy, thus caring more about 90%-95% insert throughput.

Our results show that 8-way set-associativity has the best overall performance. It always outperforms 4-way set-associativity for 100% and 50% `Insert` workloads, and for 10% `Insert` workloads when the load factor is above 0.85. 16-way set-associativity always performs worst at low or moderate table occupancy. It starts to outperform 4-way set-associativity when the load factor is above 0.75, and achieves the highest throughput for write-heavy workloads when the load factor is above 0.92. We therefore use 8-way associativity as our default because of its generality.

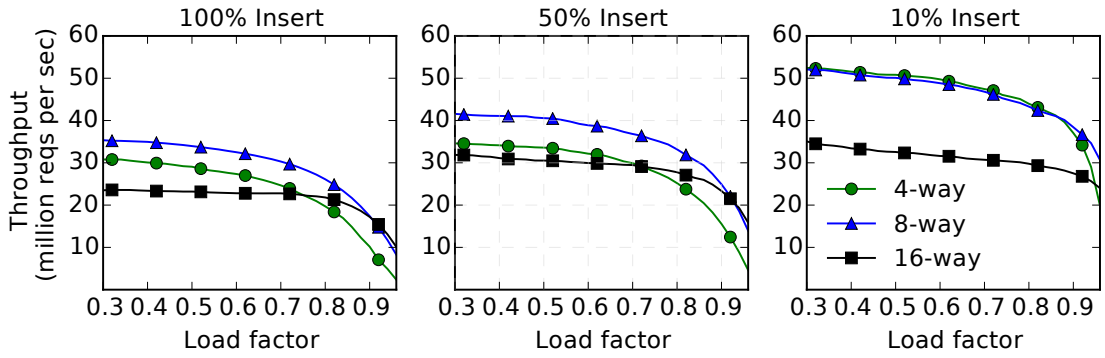


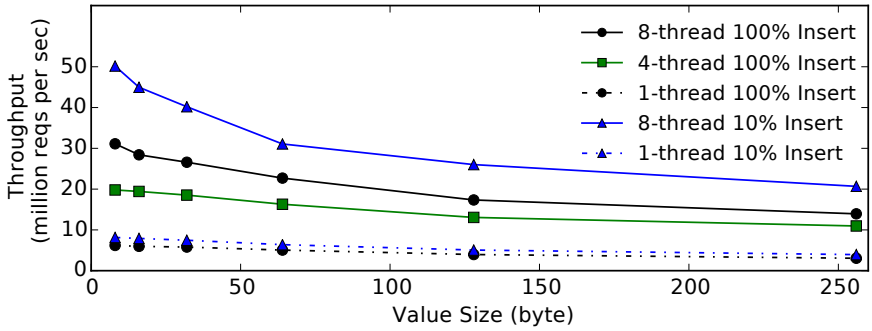
Figure 3.10: 8-thread aggregate throughput of hash tables with different set-associativities at different table occupancy. Use optimized cuckoo hashing with TSX lock elision.

3.5.4 Different Key-Value Sizes

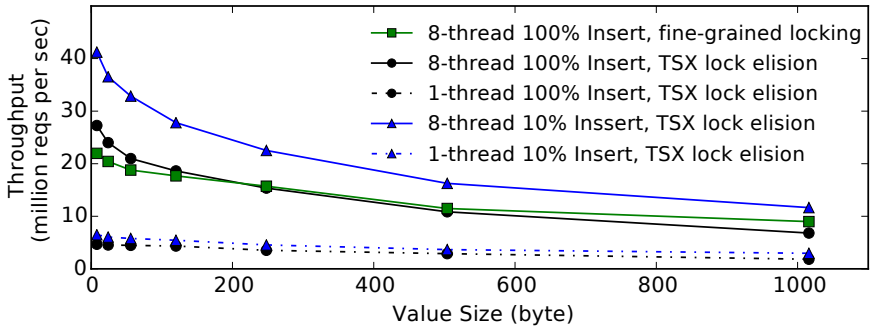
All previous experiments used workloads with 8 byte keys and 8 byte values. In this section, we evaluate the cuckoo hash table performance with different value sizes. Figure 3.11 shows the results of our two experiments.

In Figure 3.11a, we configure the hash table with 2^{25} entries, show throughput as the value size increases from 8 bytes to 256 bytes. As expected, the throughput decreases as the value size increases because of the increased memory bandwidth needed. On our 4-core machine, hyperthreading becomes much less effective with large values, because the machine runs out of memory bandwidth, and so performance scales only to the point of running one thread on each of the 4 physical cores. For example, with 256 byte values, single-thread throughput is 3.05 millions reqs per second, 4-thread throughput is 3.6× higher than 1-thread throughput, but 8-thread throughput is only 27% higher than 4-thread throughput.

Figure 3.11b reveals an interesting consequence of our current design when used with TSX: Large values increase the amount of memory touched during the transaction and therefore increase the odds of a transactional abort. For this experiment, we fix the hash table at 4GB and increase the key-value pair size to 1024 bytes. TSX lock elision outperforms fine-grained locking with small key-value sizes, but is worse at 1024 bytes. Improving our table design to reduce this effect seems a worthwhile area of future improvement.



(a) Hash table with fixed number (~ 33.4 million) of entries, using optimized cuckoo hashing with TSX lock elision.



(b) Hash table with fixed size (4 GB), using optimized cuckoo hashing with fine-grained locking or TSX lock elision.

Figure 3.11: Throughput with 8 byte keys and different sizes of values. *thr* stands for thread, *ins* for insert.

3.6 Discussion and Implementation Availability

Our results about TSX can be interpreted in two ways. On one hand, in almost all of our experiments, hardware transactional memory provided a modest but significant speedup over either global locking or our best-engineered fine-grained locking, and it was easy to use. This confirms other recent results showing, e.g., a "free" 1.4x speedup from using TSX in HPC workloads [77]. On the other hand, the benefits of data structure engineering for efficient concurrent access contributed substantially more to improving performance, but also required deep algorithmic changes to the point of being a research contribution on their own.

The focus of this chapter was on the algorithmic and systems changes needed to achieve the highest possible hash table performance. As is typical in a research prototype, this re-

sults in a fast, but somewhat "bare-bones" building block with several limitations, such as supporting only short fixed-length key-value pairs. To facilitate the wider applicability of our results, one of our colleagues has, subsequent to the work described herein, incorporated this design into an open-source C++ library, libcuckoo [53]. The libcuckoo library offers an easy-to-use interface that supports variable length key value pairs of arbitrary types, including those with pointers or strings, provides iterators, and dynamically resizes itself as it fills. The price of this generality is that it uses locks for reads as well as writes, so that pointer-valued items can be safely dereferenced, at the cost of a 5-20% slowdown. Specialized applications will, of course, still get the most performance using the hybrid locking/optimistic approach described herein, and part of our future work will be to provide one implementation that provides the best of both of these worlds.

3.7 Conclusion

This chapter describes a new high-performance, memory-efficient concurrent hash table based on cuckoo hashing. This hash table can serve as a critical component to improve the performance and cost-efficiency of each individual key-value storage server. We demonstrate that careful algorithmic and data structure engineering is a necessary first step to achieving increased performance. Our re-design minimizes the size of the hash table's critical sections to allow for significantly increased parallelism. These improvements, in turn, allow for two very different concurrency control mechanisms: fine-grained locking and hardware transactional memory. On a 16-core machine with write-heavy workloads, our system outperforms existing concurrent hash tables by up to 2.5x while using less than half of the memory for small key-value objects.

Chapter 4

Conclusion

4.1 Summary of Contributions

This dissertation contributes a number of techniques to build high-performance and cost-efficient key-value storage for large-scale clusters and multi-core servers. The design arises from deep understanding of real-world workloads, current system challenges and emerging technical trends. Our systems achieve high performance and efficiency by 1) *minimizing memory overhead and maximizing parallelism in each server*; and 2) *ensuring efficient dynamic load balancing across a cluster of servers under nearly-arbitrary workloads*. Our system prototypes achieve better performance than conventional systems with lower resource consumption. More specifically, this dissertation made following contributions.

- The design of a new cost-effective, large-scale, SSD-based key-value store architecture that uses fast, small cache to ensure dynamic load balancing without substantial over-provisioning, and exploits SDN and OpenFlow switch hardware capabilities to achieve scalable throughput, low tail latency, and high system availability with efficient content-aware routing.

- An efficient hybrid cache admission mechanism for our new small-cache-based load-balanced key-value storage cluster to meet the challenges imposed by the update

rate limits in switch hardware and the small cache size. It can keep the cache and switch forwarding rules updated with low overhead, and ensure stable high system performance under rapidly changing workloads.

- Algorithmic improvements for fast concurrent cuckoo hashing to achieve both high memory-efficiency and high throughput for both read- and write-heavy workloads. The optimizations include minimizing the size of critical sections, minimizing the number of memory writes per insert operation, enabling effective prefetching, and optimizing the concurrency control mechanisms.

- Experience from using hardware transactional memory to build concurrent hash tables, with summarized reasons for transactional aborts and principles to minimize the abort rate. We find that HTM provides primarily *engineering* benefits, not *performance* benefits. Algorithmic optimizations are needed to achieve high performance for both HTM and fine-grained locking implementations.

Collectively, the contributions in this dissertation provide system solutions and lessons to building key-value storage for cloud and big data applications. They demonstrate how to integrate different hardware and software techniques with new system and algorithm design to achieve high performance at low cost.

4.2 Open Issues and Future Work

Most of this dissertation has focused on providing efficient dynamic load balancing for key-value clusters, and improving the concurrency for cuckoo hash tables. There are still many questions and challenges that deserve future investigation.

Scalable design and deployment of SwitchKV. In Chapter 2, we only sketch a design for a scale-out version of SwitchKV with multiple cache servers and OpenFlow switches.

The detailed design and deployment in real data centers merit further investigation. It would be interesting and challenging to explore the interaction of network topologies, cache and backend server placements, OpenFlow switch forwarding rule management, and the content-aware routing protocols.

Full exploration of the design space of key-value storage cluster. To achieve high-efficiency, we moved from one extreme to another: instead of aggressively using DRAM, SwitchKV serves most (almost all) data from SSD, and only uses a very small amount of DRAM to serve the very few hot objects. It might be worth to fully explore the design space of key-value systems, with different choices of using flash and DRAM, and different designs of architectures.

A more challenging question is given certain expected workloads (e.g., object sizes, read/write ratio, query distributions), service level objectives (e.g., throughput, latency, consistency), and infrastructure constraints (e.g., network topologies, number of clients), can we quickly come up with a cluster design that can meet the service level objectives with lowest cost?

Better hash tables. We have demonstrated that with algorithmic optimizations, cuckoo hash table can be highly memory-efficient and very fast with concurrent requests. It would be interesting to know if there is any other hash table designs that could be better in terms of performance and memory-efficiency on multi-core systems. One data structure worth exploring and comparing with is hopscotch hashing [24, 39]. In theory, hopscotch hashing has better cache locality but requires more memory writes per insert operation when the table occupancy is high. Therefore, hopscotch hashing should have better performance when the memory usage is low but worse performance when the memory usage is high. It is unclear if we can further improve any of these two data structures to achieve better performance with high table occupancy.

4.3 Concluding Remarks

The highly I/O intensive, massively parallel, deeply skewed, and rapidly changing workloads of modern cloud and big data applications pose great challenges to scaling key-value storage performance in a cost-efficient manner. Such systems need to ensure load balancing without substantial over-provisioning as the number of storage servers increases, and they must maximize the memory efficiency and resource utilization of each individual server.

This dissertation argues for new approaches to building key-value systems that can meet these performance goals at low cost. We do so by coupling new hardware and infrastructure capabilities (e.g., OpenFlow, Intel DPDK, Intel TSX) with careful architectural design (e.g., content-aware routing) and algorithm engineering (e.g., hybrid cache update, concurrent cuckoo hashing). We believe high-performance and cost-efficient key-value storage for real production systems is worthy of further research, and the insights of this work can be applied to a wider range of systems.

Bibliography

[1] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253665-047US. Intel Corporation, June 2013.

[2] Intel Threading Building Block. `https://www.threadingbuildingblocks.org/`.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2008.

[4] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[5] Ismail Ari, Bo Hong, Ethan L. Miller, Scott A. Brandt, and Darrell D. E. Long. Managing flash crowds on the Internet. In *Proceedings of the 11th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '03)*, 2003.

[6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, 2012.

[7] Luiz André Barroso and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. 2009.

[8] Mateusz Berezecki, Eitan Frachtenberg, Mike Paleczny, and Kenneth Steele. Many-core key-value store. In *Proceedings of the Second International Green Computing Conference*, 2011.

[9] Peter Bodik, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. Characterizing, Modeling, and Generating Workload Spikes for Stateful Services. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC)*, 2010.

[10] Cassandra. `http://cassandra.apache.org/`.

[11] 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 *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '06, 2006.

[12] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. Rock: A High-Performance Sparc CMT Processor. *IEEE Micro*, 29(2):6–16, March 2009.

[13] Yue Cheng, Aayush Gupta, and Ali R. Butt. An In-memory Object Caching Framework with Adaptive Load Balancing. In *Proceedings of the Tenth European Conference on Computer Systems*, 2015.

[14] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Rivière. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *Proc. 5th EuroSys*, pages 27–40, 2010.

[15] Jaewoong Chung, Luke Yen, Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, David Christie, and Dan Grossman. ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory. In *Proc. 43rd MICRO*, pages 39–50, 2010.

[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, 2010.

[17] Graham Cormode and Marios Hadjieleftheriou. Finding frequent items in data streams. *Proc. VLDB Endow.*, 1(2), August 2008.

[18] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[19] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Commun. ACM*, 56(2), February 2013.

[20] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: High throughput persistent key-value store. *Proc. VLDB Endow.*, 3(1-2), September 2010.

[21] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, 2007.

[22] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *Proc. 14th ASPLOS*, pages 157–168, 2009.

[23] 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 *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[24] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2014)*, April 2014.

[25] Ulfar Erlingsson, Mark Manasse, and Frank McSherry. A Cool and Practical Alternative to Traditional Hash Tables. In *Proc. 7th Workshop on Distributed Data and Structures (WDAS'06)*, Santa Clara, CA, January 2006.

[26] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A Distributed, Searchable Key-value Store. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2012.

[27] Bin Fan, David G. Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[28] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC)*, 2011.

[29] Brad Fitzpatrick. Distributed Caching with Memcached. *Linux J.*, 2004(124):5–, August 2004.

[30] Daniel Ford, Francois Labelle, Florentina Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[31] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2011.

[32] Google SparseHash. `https://code.google.com/p/sparsehash/`.

[33] C. Gray and D. Cheriton. Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP)*, 1989.

[34] 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*, 1994.

[35] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2009.

[36] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: A GPU-accelerated Software Router. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2010.

[37] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proc. 20th ISCA*, pages 289–300, 1993.

[38] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[39] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch Hashing. In *Proceedings of the 22Nd International Symposium on Distributed Computing*, DISC '08, 2008.

[40] Intel Data Plane Development Kit (DPDK). `http://dpdk.org/`.

[41] Intel Performance Counter Monitor. `www.intel.com/software/pcm`.

[42] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional Memory Architecture and Implementation for IBM System Z. In *Proc. 45th MICRO*, pages 25–36, 2012.

[43] 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*, SoCC '12, 2012.

[44] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing (STOC)*, 1997.

[45] Markus Klems, Adam Silberstein, Jianjun Chen, Masood Mortazavi, Sahaya Andrews Albert, P.P.S. Narayan, Adwait Tumbde, and Brian Cooper. The yahoo!: Cloud datastore load balancer. In *Proceedings of the Fourth International Workshop on Cloud Data Management (CloudDB)*, 2012.

[46] Diego Kreutz, F.M.V. Ramos, P. Esteves Verissimo, C. Ẽsteve Rothenberg, S. Azodolmolky, and S. Uhlig. Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE*, 103(1), January 2015.

[47] H. T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[48] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2), April 2010.

[49] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[50] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, O. Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-value Store Server Platform. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, 2015.

[51] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, 2014.

[52] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be Fast, Cheap and in Control with SwitchKV. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, NSDI '16, 2016.

[53] libcuckoo. `https://github.com/efficient/libcuckoo`.

[54] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[55] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2014.

[56] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*, 2012.

[57] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read-Copy Update. In *In Ottawa Linux Symposium*, pages 338–367, 2001.

[58] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *Proceedings of the 10th International Conference on Database Theory (ICDT)*, 2005.

[59] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[60] NoviSwitch. `http://noviflow.com/products/noviswitch/`.

[61] NVM Express. `http://www.nvmexpress.org/`.

[62] OpenFLow. `https://www.opennetworking.org/`.

[63] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, January 2010.

[64] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. *Journal of Algorithms*, 51(2):122–144, May 2004.

[65] 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 (EuroSys)*, 2012.

[66] Pica8. `http://www.pica8.com/`.

[67] Recap of Cassandra Summit 2014. `http://opensourceconnections.com/blog/2014/09/17/cassandra-summit-2014/`.

[68] Redis. `http://redis.io/`.

[69] RocksDB. `http://rocksdb.org/`.

[70] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained Elastic Partitioning for Distributed Transaction Processing Systems. *Proc. VLDB Endow.*, 8(3), November 2014.

[71] Jeff Terrace and Michael J. Freedman. Object Storage on CRAQ: High-throughput Chain Replication for Read-mostly Workloads. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, 2009.

[72] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *Proc. USENIX ATC*, pages 11–11, 2011.

[73] TSX lock elision for glibc. `https://github.com/andikleen/glibc`.

[74] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[75] Voldemort. `http://www.project-voldemort.com/`.

[76] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Proc. 21st PACT*, pages 127–136, 2012.

[77] Richard M. Yoo, Christopher J. Hughes, Konrad Laiz, and Ravi Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High-Performance Computing. In *Proc. SC*, 2013.