Enhancing Transactional Key-Value Storage Systems in Datacenters using Precise Clocks and Software-Defined Storage

by

Pulkit A. Misra

Department of Computer Science
Duke University

Date: ___________________________
Approved:

____
Alvin R. Lebeck, Supervisor

____
Jeffrey S. Chase

____
Jun Yang

____
Ashwin Machanavajjhala

____
Ricardo Bianchini

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2019

Abstract

Enhancing Transactional Key-Value Storage Systems in Datacenters using Precise Clocks and Software-Defined Storage

by

Pulkit A. Misra

Department of Computer Science
Duke University

Date: ______________________
Approved:

____
Alvin R. Lebeck, Supervisor

____
Jeffrey S. Chase

____
Jun Yang

____
Ashwin Machanavajjhala

____
Ricardo Bianchini

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University
2019

Abstract

Transactional key-value storage is an important service offered by cloud service providers for building applications (e.g., Amazon DynamoDB, Microsoft CosmosDB, Google Spanner). This type of service is popular because it provides high-level guarantees like consistency, scalability and fault-tolerance to ease application development and deployment on the cloud. Unfortunately, providing high performance without high complexity entails several challenges for transactional key-value storage systems in datacenters due to several sophisticated protocols that provide the high-level guarantees (e.g., transaction and replication), and the overheads incurred by traversing multiple abstraction layers.

We leverage two emerging datacenter capabilities — precise synchronized clocks and software-defined storage — to address the performance and complexity challenges with transactional key-value storage systems in datacenters. To this end, we use a cross-layer approach that investigates all levels of the storage stack, from developer APIs to underlying hardware. We show that this methodology opens avenues for synergistic interactions between software and the underlying hardware, and leads to simpler system designs with better performance.

This dissertation presents 4 systems — Semel, Milana, Kairos and SkimpyFTL. Semel is a multi-version key-value storage system that exploits remap-on-write property of flash-based Solid State Drives for device-integrated multi-versioning and uses a simplified, unordered (inconsistent) replication protocol for fault tolerance. Milana supports serializable ACID transactions over Semel using an enhanced Optimistic Concurrency Control protocol that leverages intra-datacenter precisely synchronized clocks to reduce transaction abort rate and enable local validation of read-only transactions. Kairos builds over Milana and adds support for inter-transaction caching and sharded transaction validation; cache consistency in Kairos is based on a simple, stateless, time-to-live protocol with leases, without having to track sharers or send invalidations like with directory-based cache consistency protocols. Finally, SkimpyFTL builds over Semel and adds support for memory-efficient data indexing in flash-based key-value storage systems.

To my father and lifelong mentor, Ambikanandan Misra.

Contents

List of Tables

List of Figures

Acknowledgements

This dissertation would not have been possible without the guidance and support of many people. First and foremost I would like to thank my advisor, Alvin Lebeck. I am extremely grateful to him for giving me the freedom of thought, and for teaching me the importance of looking at the big picture rather than just fretting about the minute details. Also, I am constantly amazed by his knack of providing critical insights on my half-baked ideas, far beyond my own understanding. I hope that some day I can master this skill like him.

I would also like to express my gratitude to many amazing collaborators who taught me how to perform high-quality research. I would like to thank Jeff Chase for improving all aspects of my work. His presentation skills have had a lasting impact. Johannes Gehrke taught me the importance of choosing the “right” problem. Ricardo Bianchini and Íñigo Goiri taught me how to perform large-scale systems research and showed how much fun and rewarding this can be during my two internships at Microsoft Research.

My life at Duke would not be complete without the amazing staff and graduate students. Marilyn Butler was just a door knock away for advise on all aspects of life. Pam Spencer made it extremely easy to schedule events. Joe Shamblin helped out with managing servers on multiple occasions; he even came in on a weekend to power cycle my server while I was on the west coast. I also greatly benefited from the interactions with my fellow graduate students, especially Ramin Bashizade, Xiangyu

Zhang, Bryan Prosser, Tamara Lehman, George Mappouras, Sean Murray, Atefeh Mehrabi, Jack Snyder, Sahba Tashakkori, Sudarshan Balaji and Amir Gilad. I will also cherish the high-intensity table tennis matches with Sam Haney.

My family has been a great source of encouragement and support throughout the years. I would like to thank my dad, Ambikanandan Misra, for setting the best example; I would have never reached this point without his advise and support. My mom, Sarita Misra, is always there for guidance. I want to thank her for her unwavering belief in me, which has been extremely beneficial through tough times.

Finally, I would like to thank my best friend and wife, Jharna Modi, for always taking care of me. She has been my anchor to reality and has provided more encouragement and support than I deserve. I hope I can partly return the favor in the years to come.

1

Introduction

Large-scale datacenters provide the computational infrastructure that underlies the increasing use of cloud services. A key aspect in many datacenters is the use of commodity hardware to provide scale-out cloud infrastructure for services such as Software as a Service(SaaS), Hardware as a Service (HaaS) and the more generalized Anything as a Service (XaaS). Today's datacenters exhibit properties of both loosely coupled distributed systems and tightly coupled supercomputers. For example, networking infrastructure now mimics early supercomputers with low latency and high bandwidth per link, high bisection bandwidth Fat-Tree topologies [8, 77], and remote memory operations (e.g., RoCE). We believe that the rapid increase in cloud computing is driving a trend to move further toward supercomputing-like capabilities. Nonetheless, the scale and criticality of today's systems demands a distributed service architecture that is simple, scalable, provides good performance and is resilient to failures, even within the datacenter.

In keeping with this model it is crucial to continually examine existing, new and emerging features available to improve support for cloud services. Several software and hardware trends have emerged to this end. For example, a trend in software is to shift from designing large, complex, monolithic services to using a large number of simple, single-purpose, loosely-coupled microservices for providing the same end-to-end service functionality, because this approach improves software modularity, deployment flexibility and performance debugging [119, 1, 28, 49]. Another trend is to leverage specialized hardware to improve performance of large-scale cloud services. For example, GPUs [7, 6] and reconfigurable FPGAs [106, 20] have been used to accelerate web search. FPGAs and specialized hardware accelerators have also been leveraged to enable machine learning in real time in datacenters [48, 63].

We focus on transactional key-value storage service inside datacenters. Key-value storage is a fundamental building block for many modern-day, data-intensive applications and is used in a variety of domains, such as web-indexing [23, 33], e-commerce [37], data deduplication [34], photo stores [14], social networking [97, 19, 114], online gaming [35], messaging [56, 55] and more. Key-value storage systems are designed for linear outward scalability to thousands of servers. Data in these systems is stored as an independent collection of key-value pairs, that are distributed over a cluster of servers. Earlier versions of key-value storage systems provided a simple GET and PUT interface to access single key-value pairs, without any support for transactions, i.e. guaranteeing *atomicity*, *consistency*, *isolation*, and *durability* (ACID) of data accesses [37, 29, 70]. This limitation was because transactions were thought to be incompatible with the scalability goals of these systems [121]. However, several recent works show that ACID transactions can be a practical extension to key-value storage systems, and this extension helps application developers manage complexity and concurrency in distributed environments [31, 116, 67, 39, 130, 43].

Transactional key-value storage systems are popular because they provide high-level guarantees like consistency, scalability and fault-tolerance to ease application development. Unfortunately, providing high performance without high complexity entails several challenges for transactional key-value storage systems in datacenters due to several sophisticated protocols that provide the high-level guarantees (e.g., transaction and replication), and the the overheads incurred by traversing multiple abstraction layers.

We leverage 2 emerging datacenter capabilities — precise synchronized clocks and software-defined storage — to address the performance and complexity challenges with transactional key-value storage systems. To this end, we use a cross-layer approach that spans from low-level software close to the hardware up through developer APIs as we believe that re-examining existing abstractions exposes new opportunities for enhancing this important cloud service.

The remainder of this chapter is organized as follows. Section 1.1 provides an overview of transactional key-value storage systems inside datacenters, and describes the challenges introduced by the different protocols used by these systems. We describe our key contributions in addressing the challenges in Section 1.2.

1.1 Overview and Challenges

Transactional key-value storage systems inside datacenters are arranged in a client-server architecture, as illustrated in Figure 1.1. Requests of external clients (users) to a cloud-based application are sent to application servers in the nearest datacenter. Within a datacenter, a load balancer distributes user requests across the application server tier. The application servers are the *clients* of the storage system. User requests are executed as transactions on a client; each client has a front-end library to issue transaction operations to the storage servers, where the key-value data resides.

For fault tolerance, each key-value pair is replicated across multiple storage servers. The application key space is partitioned (sharded) across servers; sharding enables horizontal scaling of the storage system because capability (e.g., storage capacity, peak throughput) can be increased by adding new servers and assigning key shards to them. All the servers (client and storage) in the system are monitored

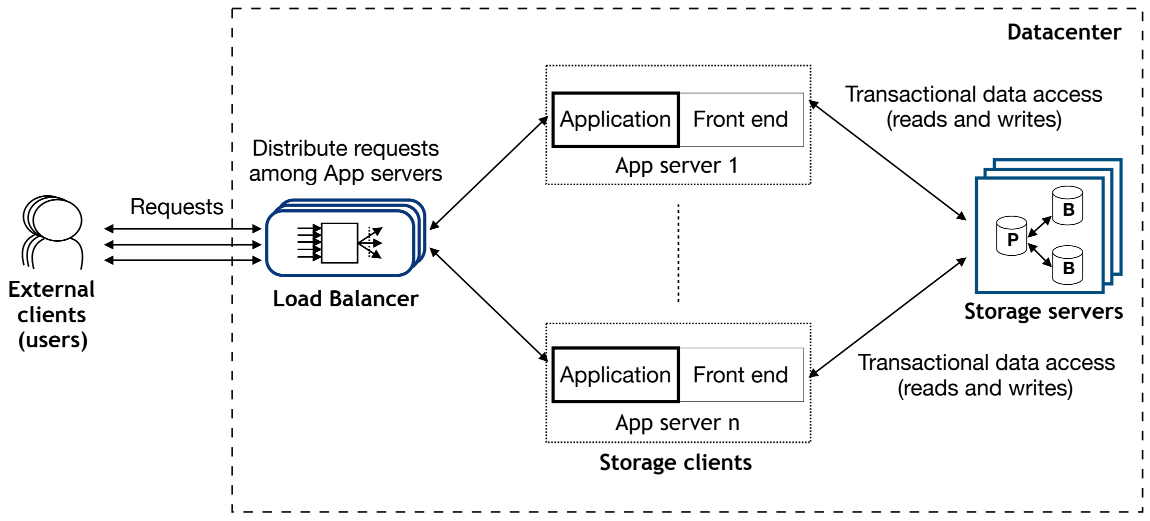


FIGURE 1.1: Structure of a transactional key-value storage system in a datacenter

by a global master, which facilitates scaling and handling failures in the system.

Transactional key-value storage systems use several protocols to provide high-level guarantees (e.g., consistency, fault tolerance) and good performance to the application. They use a storage protocol for data management on each storage server. A replication protocol is used for fault tolerance. These systems layer a transaction protocol over replicated servers for providing ACID transactions. Finally, they use an inter-transaction caching protocol for providing good performance. Below we describe each protocol and also highlight the challenges with the protocol.

Storage Protocol. Transactional key-value storage systems typically use a multi-version storage to increase concurrency in the system since read requests can be satisfied using a prior (older) version, while writes create new versions [15, 31, 95, 44, 125]. However, there are several challenges in designing a multi-version storage. First, the extra versions require additional capacity. Second, these systems need an indexing mechanism to map versions of a key to their value. A naïve approach is to store the entire index in main memory (DRAM); this approach provides the lowest read latency but has a high space overhead. An efficient indexing technique needs to tradeoff between lookup latency and memory requirement for indexing. Third, multi-versioning necessitates a version management scheme for effective capacity utilization. The scheme needs to strike a balance between keeping and discarding prior versions for servicing read requests and capacity reclamation, respectively.

Replication Protocol. Key-value storage systems use a replication protocol for fault tolerance and high availability. Typical replication protocols, such as Paxos [71] and Viewstamped Replication [98], cluster a group of servers into an ensemble called a *replica group*. One server in a replica group acts as the primary (or leader), and the remaining act as backups. A leader drives operations in sequence order to the backups for consensus (strong consistency). However, this approach to achieving consensus incurs high latency because each operation commits only after a majority of replicas accept it, and the ordering requirement prevents a replica from accepting an operation until it has accepted all prior operations.

Transaction Protocol. Key-value storage systems layer a transaction protocol over a group of replicated servers for supporting ACID transactions. A transaction protocol includes a concurrency control mechanism to enforce isolation among concurrent transactions. This isolation can be provided in two ways: 1) Two Phase Locking (2PL), and 2) Optimistic Concurrency Control (OCC) [68]. 2PL is a pessimistic approach since it requires a client to acquire a read/write lock for each key accessed by a transaction. However, acquiring locks limits concurrency and is also prone to deadlocks. In contrast, OCC eschews acquiring locks and therefore enhances concurrency relative to 2PL. A transaction in OCC is *speculatively* executed without acquiring any locks and is *validated* before commit to identify any conflicts due to concurrent modification by other transactions. Validation typically occurs on storage servers [3, 75, 130, 92] and any transaction that fails validation is aborted.

A prior work, Thor [3], integrates loosely-synchronized clocks with OCC and shows how timestamps can be used to detect conflicts during transaction validation;

many systems use a variant of the OCC techniques pioneered in Thor [41, 39, 130, 42, 75, 25, 92, 129]. Unfortunately, clock skew between servers can impact abort rates with OCC as it increases the risk of timestamp ordering conflicts during validation [3]. With advancements in network technologies and newer storage mediums (e.g., Non-volatile Memory), data access (read/write) latencies in key-value storage systems can be in order of 10s of μs [53]. Whereas, typical clock synchronization protocols, such as Network Time Protocol (NTP), provide a synchronization accuracy of 10s of milliseconds, which is very coarse grained and can lead to spurious aborts. Such aborts lead to increased application latencies and lower throughput.

Inter-Transaction Caching Protocol. The popularity of data items in real-world workloads often exhibits a power law distribution [30, 11]. In such distributions, a small subset of the data (key-value pairs) receives a disproportionately high number of accesses and can cause the storage server(s) storing this frequently-accessed data to bottleneck the entire system. Such bottlenecks cause longer transaction execution times, which, in turn, increases the likelihood of contention among transactions, leading to higher abort rate and performance degradation [129].

Caching frequently-accessed data on clients can alleviate such workload-induced hotspots. However, systems (e.g., Thor [3]) that support inter-transaction caching typically use *explicit invalidation* for client cache consistency. This approach requires servers to track sharers (client caches with a copy) of each object and send invalidations (callbacks) to sharers for removing the cached copy on each update. However, explicit invalidation introduces substantial performance and scalability overhead in transactional key-value storage systems. As a result, many recent systems do not address inter-transaction caching at all [31, 116, 41, 75, 39, 130, 92].

1.2 Contributions

We leverage two emerging datacenter capabilities — precise synchronized clocks and software-defined storage — to address the following challenges with transactional key-value storage systems: 1) multi-version storage protocol, 2) ordering constraint of replication protocol, 3) high abort rate due to clock skew with the transaction protocol (Optimistic Concurrency Control), and 4) explicit invalidation overhead with the inter-transaction caching protocol. Below we summarize our key contributions.

Chapter 3 describes two systems: SEMEL and MILANA. SEMEL is a multi-version key-value storage system that provides non-transactional access to single keys. It addresses the capacity and version management challenge with storage by implementing multi-versioning in the Flash Translation Layer (FTL) of a Solid State Drive (SSD). Furthermore, it leverages precise synchronized clocks to design an inconsistent replication protocol that performs update ordering only on server failures.

Our SEMEL implementation utilizing Precision Time Protocol (PTP) [60] and the LightNVM Open-Channel SSD emulation framework [18] reveals a 20-45% increase in throughput and up to 7× lower GET latency on a single machine compared to a naïve multi-version key-value storage system implemented over a standard FTL for read heavy workloads (50-100% GET ops).

MILANA adds OCC to support ACID transactions over SEMEL. MILANA addresses the challenge of high abort rate due to clock skew with OCC. We show that clock skew with Network Time Protocol (NTP) [91] is too high in modern datacenters and that PTP enables use of OCC with low abort rates. Furthermore, MILANA uses precise synchronized clocks to eliminate server-side validation of read-only transactions, which reduces the number of messages and improves performance.

Evaluation of MILANA prototype using Retwis [73] workload show up to 43% reduction in abort rates using PTP vs. NTP due to tighter clock synchronization.

Moreoever, local client validation of read-only transactions in Milana reduces transaction latency by 35% and increases throughput by 55% for read-heavy workloads.

In Chapter 4, we describe Kairos, a system that builds on the approach of Milana by using precise synchronized clocks for OCC and adds support for inter-transaction caching, without the cost of tracking sharers and explicit invalidations. Precise synchronized clocks enable a simple time-to-live (TTL) protocol with leases for cache consistency. Furthermore, Kairos leverages sharded validation [39] to decouple transaction validation from the servers, and adapts it to work with inter-transaction caching.

Evaluation of a Kairos prototype using a YCSB workload [30] reveals that inter-transaction caching alone improves throughput by 1.86x relative to a baseline system with only intra-transaction caching; adding sharded validation further improves throughput by a factor of 2.28 under a workload with a hotspot that saturates a storage server. Furthermore, our evaluation shows that lease-based inter-transaction caching can operate at a 62.5% higher scale while providing 1.55x the throughput of a system with explicit invalidation-based caching in workloads with hot keys.

Chapter 5 describes SkimpyFTL, a system that builds on top of Semel and addresses the challenge with memory-efficient indexing in multi-version storage. SkimpyFTL uses a hash-based approach for indexing and offloads portions of the index to flash for enabling a tradeoff between memory capacity and lookup latency for indexing.

A SkimpyFTL prototype utilizing LightNVM Open-Channel SSD emulation framework [18] reveals SkimpyFTL provides 72-91% throughput of Semel for read-dominant key-value workloads (75-100% reads), while reducing the memory requirement for indexing by 95%. For a transactional YCSB [30] workload, SkimpyFTL provides 85% peak throughput of Semel. Finally, SkimpyFTL outperforms a naïve multi-version key-value store implemented over a standard FTL.

1.3 Summary

The trend of datacenters to exhibit properties of both tightly coupled supercomputers and loosely coupled distributed systems presents unique opportunities to optimize cloud service implementations. We examine transactional key-value storage, an important service in datacenters. Unfortunately, providing high performance without high complexity entails several challenges for these systems due to use of sophisticated protocols and various levels of abstraction. To address these challenges, we leverage two emerging capabilities — precise synchronized clocks and software-defined storage — and use a cross-layer approach of investigating all levels of the storage stack, from developer APIs to underlying hardware. We show that this methodology opens avenues for synergistic interactions between software and underlying hardware, and leads to simpler system designs and better performance.

2 Background and Motivation

Cloud computing has emerged as a successful and ubiquitous paradigm for service oriented computing and has revolutionized the way computing infrastructure is abstracted and used. Several cloud abstractions have gained popularity over the years e.g., Software as a Service (SaaS), Infrastructure as a Service (IaaS), and the generalized Anything as a Service (XaaS). These services provide many enabling features (e.g., *elasticity*, *high availability*, *low time to market*, and *transfer of risks* etc.) that have made cloud computing a ubiquitous paradigm for deploying applications spanning all aspects of the human endeavor. Analysts forecast that global cloud services revenue will reach $410 billion by 2020 [50, 127].

The scale and criticality of these services demands a distributed system architecture that is simple, scalable, provides good performance and is resilient to failures. Therefore, it is crucial to continually examine existing, new and emerging features available to enhance these services. For example, GPUs [7, 6] and reconfigurable FPGAs [106, 20] have been used to accelerate web search. FPGAs and specialized accelerators have also been leveraged to enable machine learning in real time [48, 63]. Similarly, smart Network Interface Cards (NICs) [79] and switches [85, 61] have been

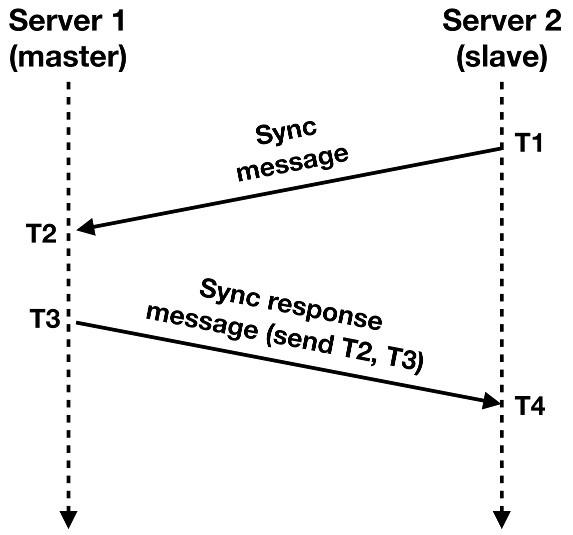


Figure 2.1: Exchange of messages for clock synchronization

used to optimize storage services in datacenters.

In this same spirit, our work leverages two emerging datacenter capabilities —- precise synchronized clocks and software-defined storage — to architect transactional key-value storage, an important service inside datacenters. This chapter describes these 2 emerging technologies. We start by describing precise synchronized clocks in Section 2.1; software-defined storage is defined in Section 2.2.

2.1 Precise Synchronized Clocks

Several clock synchronization protocols with varying level of synchronization accuracy — from 10s of ms to 100s of ns — have been proposed in the literature [91, 60, 76, 53]. We start by describing how these protocols synchronize clocks in Section 2.1.1. Section 2.1.2 describes the techniques used by various protocols to improve clock synchronization accuracy. We conclude this section by describing how we use precise synchronized clocks in Section 2.1.3.

2.1.1 Basics of Clock Synchronization

Figure 2.1 illustrates the messages exchanged between two servers for clock synchronization, where one server acts as the master (time source) and the other as a slave, who synchronizes its clock with the master. Different clock synchronization protocols use different terminologies and sets of messages but the underlying principle is the same: exchange messages to calculate one way delay (OWD) between the servers and clock offset with the master. The figure shows a slave sending a *sync* message at time $T1$, which is received by the master at time $T2$. Later, the master sends back a *sync response* message at time $T3$, which is received by the slave at time $T4$; the response message contains the timestamps $T3$ and $T4$. At the end of the exchange, a slave has 4 timestamps needed for synchronizing its clock with the master. Equation 2.1 shows how to calculate the OWD between the servers; the calculation assumes that the latency to send a message between a master and slave is symmetric i.e., it takes similar time, irrespective of the sender (master or slave). After calculating OWD, a slave can use Equation 2.2 to determine its clock offset from the master. Typically, clock synchronization protocols take several samples of these 4 timestamps and use average values for synchronization; they may also filter outliers [53].

$$\begin{aligned} latency_{SM} &= T2 - T1 \\ latency_{MS} &= T4 - T3 \\ OWD &= \frac{latency_{SM} + latency_{MS}}{2} \end{aligned} \tag{2.1}$$

$$clockOffset = latency_{SM} - OWD \tag{2.2}$$

2.1.2 Improving Clock Synchronization Accuracy

Although synchronization protocols use a variant of the above described mechanism to calculate clock offset, the synchronization accuracy can vary significantly. For example, Network Time Protocol [91] achieves a synchronization accuracy of several ms, whereas Precision Time Protocol [60] delivers a < 1 μs accuracy. This is because any queuing delays at any point in the communication path between the master and a slave can cause a slave to estimate inaccurate values for OWD and clock offset, and thereby impact synchronization accuracy.

In NTP, the clock synchronization messages are timestamped on the host processor. Consequently, it is susceptible to any queuing delays in the operating system stack of the host (e.g., a context switch after a message has been timestamped but before it is sent over the wire) or the network stack (e.g., queuing delays in the input or output port of a network switch in the communication path between the master and a slave). In contrast, synchronization messages are timestamped on the Network Interface Card (NIC) in PTP, just before the message is sent/received over the network; this approach eliminates queuing delays in the operating system stack on the host. To eliminate queuing delays in the network stack, PTP uses specialized "transparent" switches, which record the ingress and egress time of each clock synchronization packet to account for queuing latencies in the network. As a result, PTP yields a < 1 μs synchronization accuracy.

Furthermore, recent research demonstrates $\leqslant 150$ ns skew across a datacenter [76, 53]. DTP [76] achieves precise clock synchronization on servers by exploiting IEEE 802.3 Ethernet standard's natural clock synchronization mechanism between the transmitter and receiver PHYs at either end of a wire. DTP improves synchronization skew beyond PTP to less than $160ns$ throughout a datacenter and less than $30ns$ for directly connected servers. However, like PTP, DTP requires specialized hardware at every PHY in the datacenter. In contrast, Huygens [53] achieves $<$ 100 ns clock synchornization accuracy between servers across a datacenter without using any specialized network switches. It leverages several techniques towards this end. First, it filters out “noisy” synchronization messages that suffer from queuing delays. Second, it uses a machine learning classifier to accurately estimate OWD and clock offset. Finally, it exploits a natural network effect — the idea that a group of pair-wise synchronized clocks must be transitively synchronized — to detect and correct synchronization errors even further.

2.1.3 Using Precise Synchronized Clocks

Precise synchronized clocks enable distributed applications to operate on a common time axis, which, in turn, enables key functions like consistency, event ordering, causality and the scheduling of tasks and resources with precise timing [53].

An early paper by Liskov [83] describes many fundamental uses of precise synchronized clocks in distributed systems. Thor [3] pioneered the technique to provide transactions in distributed storage systems using Optimistic Concurrency Control (OCC) and synchronized clocks. In recent times, Google's Spanner [31] used synchronized clocks to provide external consistency in geo-distributed storage systems. Benefits of synchronized clocks have also been shown in the network. In software-defined networks, synchronized clocks create an order of forwarding rule updates, which helps avoid routing loops [87]. Precise synchronized clocks can also be used to assign time slots for sending packets and thereby achieve high bandwidth and near-zero queuing delays [104].

Our work uses precise synchronized clocks to simplify protocols and improve performance of transactional key-value storage systems within a datacenter. With advances in network technology, the one-way network latency is $< 10\ \mu$s [53]. Storage access latencies (e.g., solid state drives) are on similar scales. In this scenario, clock

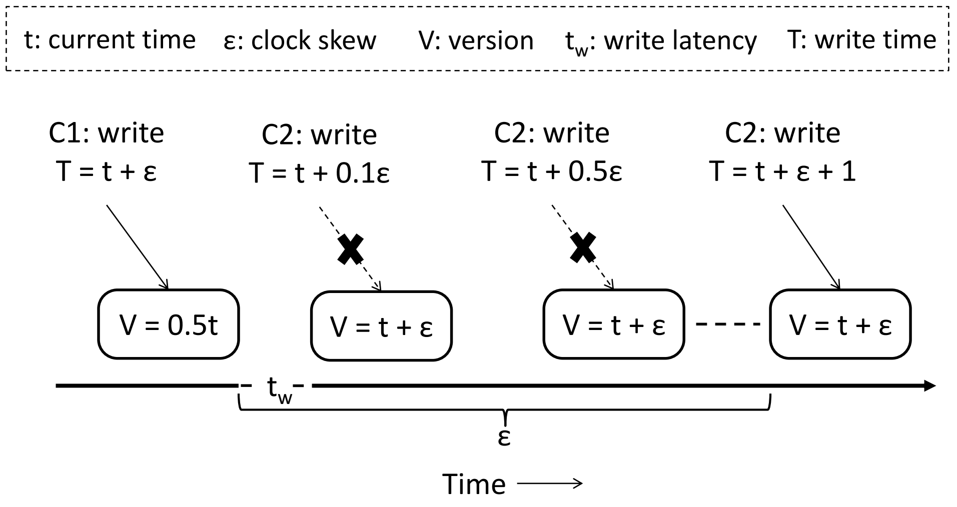


FIGURE 2.2: Impact of clock skew

skew becomes critical for performance. Figure 2.2 illustrates such a scenario; it shows an example of a shared object updated by two clients C_1 and C_2, ϵ is the clock skew and t_w is the write latency. The system in the figure rejects any writes that attempt to create a new version with timestamp less than the timestamp of the current version, just like OCC. Since the client with a leading clock (C_1) updates the object first, the lagging client (C_2) has to wait for a duration $> \epsilon$ before it can successfully update the shared object. If $\epsilon >> t_w$ then there are spurious aborts even though the storage system is capable of satisfying a new write request.

Our MILANA work shows that NTP time skew is too high for modern low-latency datacenters and that PTP enables use of OCC with low abort rates, even in high-contention scenarios. Furthermore, precise synchronized clocks enable SEMEL to simplify and optimize the various protocols used by key-value storage systems. SEMEL simplifies the replication protocol to just provide fault tolerance without any ordering guarantees. SEMEL and MILANA are described in Chapter 3. Finally, our KAIROS work leverages precise synchronized clocks to enable inter-transaction caching with soft leases and dynamic self-invalidation, which helps improve performance and alleviate workload-induced hotspots in transactional key-value storage systems. Chapter 4 describes Kairos.

Next, we describe software-defined storage and how we leverage it to address the storage protocol challenges in transactional key-value storage systems.

2.2 Software-Defined Storage

A key service in datacenters is reliable persistent storage—historically provided by replicated hard disk drives (HDDs). The performance of HDDs has lagged far behind processors, memory and networks; since 1970, the performance of a microprocessor has increased by 200,000x, whereas, in the same time period, disk access latency has improved by only 9x and while throughput has improved by 163x [21]. This performance gap made the performance of software layers that manage storage — file systems, device drivers, storage networks, databases—relatively unimportant to overall system performance [115].

However, emerging persistent memory technologies, such as battery-backed DRAM, byte-addressable non-volatile memories (e.g., PCM, STTRAM etc.), and even coarser block-addressable Solid State Drives (SSDs), provide characteristic access latency between 100 ns and 10s of μs. These mediums alter the landscape of storage entirely and require software designers to rethink the importance and role of software in storage systems. Various tradeoffs have emerged to this end. For example, the NVMe standard enables per core user-space access to flash-based SSDs and thereby avoids inter-core communication and also bypasses the inefficient operating system I/O stack. Furthermore, the standard defines a protocol for direct access to flash storage via the network, without involvement of the host processor. In contrast, Software-defined Flash (SDF) is a more radical approach that enables tailoring SSDs according to application requirements. It exposes the internal geometry of the SSD to applications, and allows applications to participate in scheduling I/Os and flash management.

Our work leverages SDF for exploiting an intrinsic characteristic —remap-on-write— of flash-based SSDs. We start by briefly describing the internals of flash-based SSDs and the intrinsic characteristic that we exploit in Section 2.2.1. Section 2.2.2 describes how we use SDF.

2.2.1 Internals of a Flash-Based Solid State Drive

A SSD comprises several flash memory chips and a programmable controller that executes the Flash Translation Layer (FTL). SSD capacity is increasing rapidly through the use of vertical stacking. Simultaneously, increased throughput and decreased latency is achieved by using new queue-based PCIe interfaces (e.g., NVMe), similar to those used in high-performance network interfaces (e.g., Infiniband). SSDs can achieve $\approx$1M IOPs with capacities near 1TB per drive, and latencies of $\approx 50-100\mu$s at less than \$1.00/GB. These advances in SSD design and implementation further improve their ability to handle high-throughput big data processing. Flash continues to see increased use within datacenters [51, 57, 120].

Flash memory is organized as an array of blocks where each block contains some number of pages. Typically pages are 2-16KB in size and each block contains 128-256 pages. The page size is the smallest unit for reads and writes. Without loss of generality, we consider a single-level flash bit cell that can be written in only one direction (0 to 1). A page write operation only sets values to 1, and thus if the data changes from a 1 to 0, the page must first be erased and then the new data written (*erase-before-write*). Unfortunately, erase operations on flash occur only at a block granularity (*block-grained erase*). Moreover, flash has limited endurance: each block can be erased only a certain number of times before the cells wear out.

To accommodate the above characteristics and limitations, the Flash Translation Layer (FTL) of flash-based SSDs provides a dynamic mapping of logical addresses

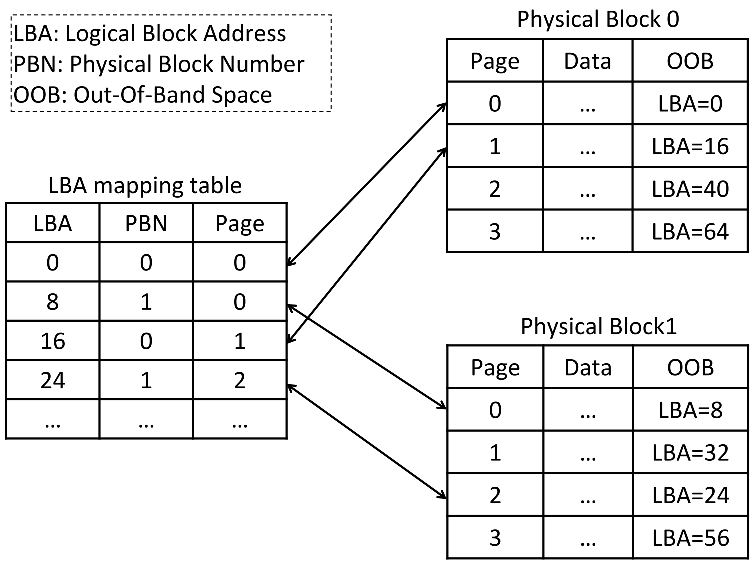


FIGURE 2.3: Flash Translation Layer (FTL) Mapping of Logical Blocks to Physical Pages & Blocks

to physical locations. The FTL presents a block device interface to the Operating System (OS) [5] and maps a Logical Block Address (LBA) to a page in flash memory, as shown in Figure 2.3. This level of indirection allows the FTL to remap a logical block to a new physical page on each write, leaving the old value in place pending garbage collection. The FTL's garbage collector may remap existing (current and valid) pages as needed to free complete blocks to erase. The FTL also implements *wear-leveling*: it distributes writes uniformly across physical locations, so the flash cells wear at the same rate.

Historically, the FTL was implemented entirely within the SSD enclosure and exposed a traditional block abstraction to software. This structure enables tight control over garbage collection and wear-leveling, aspects that can influence warranty guarantees for vendors, and allows close integration with flash read/write circuitry for read/write operations; however, it can limit flexibility.

2.2.2 Software-Defined Flash

Software-Defined Flash (SDF) is a technique to separate the FTL functionality and enable applications to participate in I/O scheduling and flash management [101, 62, 100, 22], with several vendors providing some form of this capability (e.g., CNEX Labs, SanDisk/FusionIO, Radian Memory). This approach enables several optimizations across traditional system boundaries. First, it eliminates one level of indirection since applications, such as key-value storage systems, do not need to consider SSDs as a block device and can directly map its logical blocks (e.g., keys) to pages on flash. Several works exploit this observation [88, 58, 131, 62, 111]. Next, customized mapping techniques that exploit application or system specific information can provide new functionality and/or improve performance. Prior work exploits this observation for providing snapshot capability for flash-based storage [113]. Another work leverages SDF to exploit the inherent parallelism of SSDs by mapping log-structured merge (LSM) tree [99] operations on to different SSD channels [100].

We leverage SDF to design a lightweight multi-version storage. Semel exploits the remap-on-write property of flash-based SSDs to maintain multiple versions of keys and exposes these versions to the application. Our Milana work leverages the multi-version storage of Semel to support transactions using multi-version concurrency control [15]. Semel and Milana are described in Chapter 3. Our SkimpyFTL work explores multi-version indexing techniques in flash-based key-value storage systems. Specifically, we explore the tradeoff between DRAM capacity for mapping multiple versions on the host and performance. We describe SkimpyFTL in Chapter 5.

Finally, although our work focuses on flash-based SSDs, the approach generalizes to other storage technologies as well that naturally preserve multiple versions of each key as it is updated (remap-on-write property). Our work can be applied to such storage technologies to build high-performance, reliable, low-cost, scale-out storage.

2.3 Summary

The scale and criticality of large-scale cloud services demands a distributed service architecture that is simple, scalable, provides good performance and is resilient to failures. Therefore, it is crucial to continually examine existing, new and emerging features available to enhance these services. Our work leverages two emerging capabilities — precise synchronized clocks and software-defined storage — to architect transactional key-value storage systems in datacenters. Precise synchronized clocks provide server clock skew in 100s of nanoseconds and enables distributed applications to operate on a common time axis, which, in turn, enables simplifying and optimizing protocols used in transactional key-value storage systems. Software-defined storage allows applications to participate in I/O scheduling and data management, and thereby enables creating application-tailored storage.

3

Semel and Milana

Distributed transactional storage is an important service in today's datacenters. Achieving good performance without high complexity is often a challenge for these systems due to use of several sophisticated protocols to provide high level guarantees (e.g., consistency, fault tolerance) and the presence of multiple layers of abstraction. This chapter shows how to combine two emerging datacenter capabilities—precise synchronized clocks and software-defined storage—to address several challenges with transactional key-value storage in datacenters.

We presents 2 systems: Semel and Milana. Semel[1] is multi-version key-value storage system that addresses the capacity and version management challenge with designing a multi-version storage; it also relaxes the ordering constraint in the replication protocol used for providing fault tolerance. Milana[2] is a lightweight transactional layer over Semel that reduces clock-skew related spurious aborts with the transaction protocol.

In Semel, each version of a key's value is timestamped using precise synchronized

[1] Semel means once in Latin.

[2] Milana means consistency in Bengali.

clocks. These timestamps enable a lightweight primary-backup replication protocol that moves update ordering off the critical path. Milana adds optimistic concurrency control (OCC [68]) to support serializable ACID transactions over Semel, adapted to a client-server setting based on techniques pioneered in Thor [3]. However, OCC transactions may be forced to abort/rollback under contention due to timestamp ordering conflicts with other transactions, and this risk increases with clock skew [3]. Our results show that in this setting precision time (PTP) achieves a lower rate of spurious aborts due to false conflicts (and therefore higher peak throughput) when compared to clock synchronization using NTP—the current state of the art.

Moreover, timestamp-based concurrency control enables use of multi-version approaches [15] to further improve concurrency by enabling snapshot-isolated read-only transactions with low cost. Semel leverages the erase-before-write (remap-on-write) behavior of flash SSDs to enable light-weight multi-version storage. Semel and Milana are based on an extended SSD Flash Translation Layer (FTL) that writes updated values in a log-structured fashion on physical storage, maps keys directly to values at physical locations, and integrates version management with FTL garbage collection.

Semel and Milana reflect the state of the art in sharded, replicated, transactional key-value storage systems, but embody a unique set of design tradeoffs for low-latency intra-datacenter storage with SSDs and precision time. For example, Milana is similar to a prior work — TAPIR [130] — that uses OCC in conjunction with an unordered replication protocol, which has potential for lower latency than ordered consensus (as in Thor). Consensus forces all replicas to agree on operations in a total order, which is not necessary to preserve transactional consistency. However, in contrast to TAPIR, Semel uses primary-backup replication. This choice reduces OCC validation costs in Milana: write validation occurs only on the primary replica for each affected shard, and read-only transactions validate locally at the client. The price of this efficiency is that read-write transactions require an extra round-trip latency (though not extra messages) to sequence through the primary, which is a small price for providing low latency reads, given the prevalence of read-dominated workloads [97, 11].

Our Semel implementation utilizing PTP and the LightNVM Open-Channel SSD emulation framework [18] reveals a 20-45% increase in IOPs and up to 7X lower GET latency on a single machine using unified version and flash management compared to a naive multi-version KV-store implemented using a standard FTL for read heavy workloads (50-100% GET ops). Our experiments running Retwis [73] with Milana show up to 43% reduction in abort rates using PTP vs. NTP due to tighter clock synchronization. Furthermore, local client validation in Milana reduces transaction latency by 35% and increases throughput by 55% for read-heavy workloads.

The remainder of this chapter is organized as follows. The design of Semel and Milana are described in Section 3.1 and Section 3.2. We evaluate our prototype systems in Section 3.3. Section 3.4 discusses related work and we present a summary of this chapter in Section 3.5.

3.1 Semel: A Replicated Multi-version Key-Value Store

This section presents Semel, a replicated multi-version key-value store that exploits precision time and software-defined storage. Semel provides safety guarantees for ordering of operations to individual keys. Section 3.2 shows how to support transactional atomicity and consistency for operations on multiple keys, layered above Semel.

The Semel design targets an intra-datacenter client-server storage model. The persistent memory (SSDs) reside on storage servers. The key space is sharded among the storage servers, and each shard is replicated for availability and fault tolerance. The clients of SEMEL are application servers. Each client has a unique ID and runs a SEMEL library that exposes the key-value storage API and issues read and write operations to the storage servers. The client library coordinates with a global master to map each key to a data shard and to the shard's primary replica using standard techniques (e.g., consistent hashing [65]). The master maintains the shard maps based on its global view of participating servers. The master can be implemented using standard techniques (e.g., Apache Zookeeper [59]).

Values for each key are stored as a sequence of versions timestamped by the client that issued the write. Versions are ordered by the version number, which is a $V = \langle timestamp, clientID \rangle$ tuple. The clientID induces a total order over simultaneous writes from different clients, and also supports linearizability (Section 3.1.3). SEMEL uses these timestamps to maintain a coherent view across all clients for each key. We do not expect timestamp wraparound to be an issue if we use 64-bit timestamps. Assuming $\approx$ 100ns resolution for timestamps, a 64-bit timestamp does not overflow for nearly 60 *thousand* years.

The application API to SEMEL client library is defined below. We use $t_{current}$ to denote a client's view of the current time.

- **put(key, value)**: Create a new version for the given key.
- **get(key) → value**: Return a version with timestamp $\leqslant t_{current}$.
- **delete(key)**: Delete all versions of the key.

The client library assigns a timestamp $t_{current}$ to all *get* and *put* requests. This timestamp is used for creating a new version $V = \langle t_{current}, clientID \rangle$ of a key on a put request. For a get request, SEMEL uses $t_{current}$ to read the *youngest* version with timestamp $\leqslant t_{current}$. MILANA (Section 3.2) extends the SEMEL client to issue reads for a specific timestamp other than $t_{current}$ as required for the transaction protocol.

3.1.1 Multi-version Flash Translation Layer

Recall, an SSD Flash Translation Layer (FTL) maps a Logical Block Address (LBA) to a page in flash memory. A page is uniquely identified by a Physical Block Number (PBN) and page number within the physical block. A map table in the FTL is consulted to determine the flash page for each I/O operation on an SSD. For example, a read operation performs a $Key \rightarrow Page$ conversion and then reads the data from the physical page. Flash SSDs do not overwrite pages on a write operation, due to the need for block-grained erase-before-write. A standard FTL writes each modified page to a freshly erased block and remaps the page. Therefore, flash SSDs may naturally provide multiple versions of a given key with little additional overhead [113].

Semel leverages the Open-Channel SSD framework [18] to extend the FTL for multi-version key-value storage. A key-value storage system implemented using traditional SSDs requires two mapping steps: $Key \rightarrow LBA \rightarrow \langle PBN, Page\rangle$. Software-Defined Flash (SDF) enables modifying the FTL to collapse this two-step translation into a single translation [58, 131, 62, 88], so that it maps a key directly to a physical address with a single map table access.

Mapping table:. The mapping table in Semel FTL maintains multiple versions of a key as a linked list. Each version is assigned a 64-bit create timestamp; the linked list is sorted in descending order of create timestamps of the versions. Semel writes new values in a log-structured fashion on flash. Figure 3.1 shows the mapping table and data layout in Semel. For small key-value pairs, Semel packs multiple pairs into a single page. The mapping table maintains the page and the offset within the page where the version is stored.

Semel FTL assumes that adequate server DRAM is available to store the entire mapping table in main memory. Chapter 5 describes how we address this drawback using memory-efficient indexing.

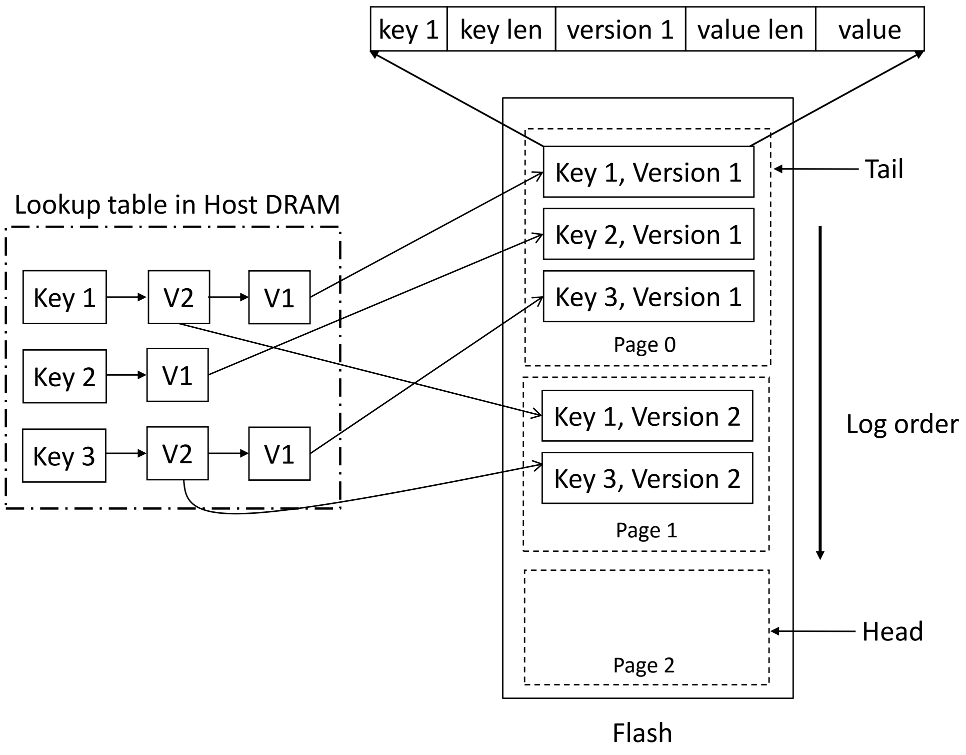


FIGURE 3.1: Mapping Table and Data Layout in SEMEL

Garbage collection:. Keeping versions around longer than necessary on flash-based systems may cause wasteful remapping (moving) during garbage collection. Ideally we want to balance remapping cost with the desire to provide historical versions within a certain threshold (window size), e.g., keep all versions that are less than 5 seconds old. The window size can be tunable to keep older versions as needed, e.g., for read-only analytics workloads. SEMEL utilizes watermarking [39], which establishes a lower bound on the client clocks. Each client periodically broadcasts the timestamp of its last acknowledged operation to all storage servers and the minimum of all these timestamps is the watermark in SEMEL. Since NTP/PTP clocks are monotonic, no client issues a new operation with a timestamp below the watermark. Therefore, the garbage collection algorithm needs to keep only the youngest write with a timestamp less than the watermark; it is safe to discard all prior versions.

3.1.2 Lightweight Inconsistent Replication

Key-value storage systems use a replication protocol for fault tolerance and high availability. Typical replication protocols, such as Paxos [71] and Viewstamped Replication [98], cluster a group of servers into an ensemble called a *replica group*. One storage server in a replica group acts as the primary (or leader), and the remaining servers act as backups. Such replication protocols can tolerate f storage server failures, for $2f + 1$ servers in a replica group. A primary handles all incoming read/write requests and acts as the serialization point. To provide strong consistency (consensus), a primary imposes a total order on all writes from the clients (using sequence numbers) and multicasts each write to all the backups. The backups process write requests in sequence order only and discard any write that comes out of order. A backup sends an acknowledgement to the primary for each in-order write request; a write request is considered complete by a primary only after at least f backups have acknowledged the receipt of the write.

The update life-cycle for such protocols is:

1. The application sends an update to the primary.

2. The primary assigns a sequence number to the update (to enforce ordering), which is higher than the sequence number of all previous updates. The primary then propagates the update and the sequence number to the other replicas (backups).

3. A backup replica acknowledges an update if it has *seen all the prior sequence numbers* and consequently all prior updates. Otherwise, the backup replica responds with an error.

4. If the primary receives a success from f replicas before a timeout, it acknowledges the update to the application. Otherwise, it retries the update or sends

an error to the application.

SEMEL uses primary-backup replication with a designated primary for each shard, and exploits tightly synchronized clocks to *relax the ordering requirement* and commit each update as soon as a majority of replicas receive (and acknowledge) it. Since the replicated SEMEL operations are timestamped writes to independent versions of independent data items, there is no need to maintain ordering: the ordering is explicit in the version timestamps, which are recovered along with the data.

TAPIR [130] is based on a similar *inconsistent replication* approach that decouples replication from ordering (see Section 3.2.7 for a comparison against TAPIR). Inconsistent replication reduces latency because each replica can execute and acknowledge an unordered operation as soon as it receives it, even if it is missing earlier operations. Such an approach does not violate consistency after failover since all acknowledged updates can be recovered if a majority of replicas are available, and a correct ordering can be constructed based on version numbers. We explain recovery in Section 3.2.6.

3.1.3 Linearizability with Global Clocks

SEMEL leverages precise clocks to enable a simple and general timestamping approach to linearizable RPCs on objects. RPC calls on an object execute serially at the primary for the object's shard. The timestamp of a write request persists with the new object version, even across primary failover. The version stamps allow the server to ensure idempotence for retransmitted requests; the client ID distinguishes requests from different clients with the same timestamp. The server executes reads on the named version and rejects writes with timestamps older than the current version, guaranteeing at-most-once semantics. Thus writes execute in a serial timestamp order that is consistent with the real-time ordering. SEMEL also permits snapshot reads in the past, which are not linearizable, but they allow higher concurrency and it is a client's choice to use them.

SEMEL's approach to linearizable RPC is similar in spirit to RIFL [75], which also timestamps requests at the client and persists a *completion record* containing each request's timestamp with the object. The key difference is that SEMEL's request timestamps are *global* and synchronized across the clients. Precise clocks enable us to simplify the ordering protocol. For example, it becomes safe to garbage-collect old versions and their timestamps at any time. If a client replays a completed request after the server discards its version, a simple timestamp comparison blocks it from overwriting an earlier request on the same object: the client receives a rejection for the retransmitted request (not idempotent), but at-most-once semantics are preserved.

When precision timestamps are available, ordering with global clocks is simpler than approaches based on leases and/or causal information [75, 89, 46]. The key tradeoff is that clients with lagging clocks may see their requests rejected under contention, forcing them to retry more often. The SEMEL approach is suitable when the expected clock skew is less than the request cost, which is the case for operations on flash-based SSDs with PTP-based precision timestamps. Note that ordering with global clocks depends on low clock skew only for performance, and not for correctness.

3.2 MILANA: A Transactional Key-Value Storage System

This section shows how to use SEMEL's timestamped values to support transactions in a software layer above SEMEL. Our transaction system—called MILANA—supports transactions that update keys in multiple shards atomically using a classical two-phase commit (2PC) protocol. MILANA leverages SEMEL's precision timestamps for Optimistic Concurrency Control (OCC) [68] adapted to a client-server setting [3]. OCC is an alternative to locking (two-phase locking or 2PL): OCC enhances concurrency relative to 2PL, and is not prone to 2PL's blocking and deadlocks. OCC

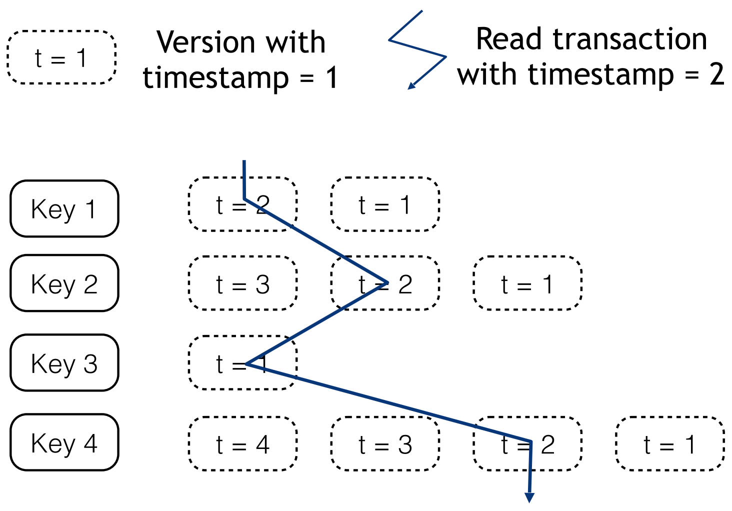


FIGURE 3.2: Snapshot reads in MILANA

systems *validate* each transaction T before commit by comparing T's timestamped data accesses—T's *read set* and *write set*—to those of other transactions to identify any access conflicts that violate a serializable ordering. Conflicting transactions are aborted and then restarted at the client. Other client-server OCC transaction systems include Thor [3], Centiman [39], and TAPIR [130].

MILANA benefits directly from PTP because low clock skew reduces the incidence of false aborts in client-server OCC [3]. For example, a false abort occurs if a late-arriving transaction (e.g., a commit request from a client with a lagging clock) conflicts with an already-committed transaction with a later timestamp. As explained previously, PTP is particularly important for storage services based on low-latency persistent memory, e.g., flash-based SSDs and emerging NVM technologies. MILANA exploits PTP to improve performance, but it is not required for correctness.

To implement OCC, MILANA assigns precision (PTP) timestamps *begin* (ts_{begin}) and *commit* (ts_{commit}) to each transaction T at the client; all read operations for T are issued at the SEMEL layer with T's *begin* timestamp and all write operations create a new version with T's *commit* timestamp. MILANA also leverages SEMEL's multi-version flash SSD store to support *snapshot reads*: Milana satisfies T's reads for a key K by returning a version that is current as of T's ts_{begin}, even if a writer has written a new version of K with a later timestamp. This approach reduces false conflicts and further improves concurrency and throughput. Figure 3.2 illustrates how consistent snapshots are read in Milana. A transaction with $ts_{begin} = 2$ reads all versions with $ts_{version} \leqslant 2$, while new versions are created by other write transactions.

Milana is optimized for read-heavy workloads, which typically dominate within a datacenter [97, 11]. In particular, Milana servers return sufficient version information to enable a client to perform *local validation* for read-only transactions (Section 3.2.3). Local validation allows a read-only transaction T to commit if and only if the values in T's read set are from a consistent snapshot: each value for a key K in T's read set is the *youngest committed* version of K with timestamp $\leqslant ts_{begin}$, and no key K in the read set has a prepared version with timestamp $\leqslant ts_{begin}$. Local validation ensures a serializable transaction ordering for read-only transactions, but it does not necessarily provide external consistency. Milana provides both serializability and external consistency for read-write transactions, which validate on the servers.

3.2.1 Transaction Protocol

Milana adds an extended transaction API to the Semel primary servers, and an enhanced client library to use it. A Milana primary maintains a *transaction table* recording the status of transactions that have prepared but for which commits have not yet been acknowledged: updates to this table are logged in persistent memory as they occur and are replicated to the backup servers using the Semel replication protocol. If the primary fails, a new primary recovers the transaction table before continuing (Section 3.2.6).

Milana uses the version stamps provided by Semel, $V = \langle timestamp, clientID \rangle$.

This approach provides monotonically increasing timestamps with a total order. We assign each transaction T two timestamps ts_{begin} and ts_{commit} at T's begin and commit time respectively. T's ts_{begin} is assigned to all GET (read) requests and ts_{commit} is assigned to all PUT (write) requests.

In addition, a MILANA primary server also maintains in DRAM a $ts_{latestRead}$, $ts_{prepared}$ and $ts_{latestCommitted}$ timestamp for each active key. $ts_{latestRead}$ is set on a get request if $ts_{get} > ts_{latestRead}$. $ts_{prepared}$ and $ts_{latestCommitted}$ timestamps are set after a successful validation and commit, respectively. None of these values are persisted; Section 3.2.6 explains how to recover them.

Here is the application API to the MILANA client library. Timestamp $t_{current}$ represents the client's local view of the current time as given by PTP.

- **beginTransaction()**: Start a new transaction T. Assign a begin timestamp to T ($ts_{begin} = t_{current}$), and initialize an empty read and write set for T.
- **abortTransaction()**: Discard the read and write set maintained for the current transaction and remove all state.
- **commitTransaction() → Success / Fail**: Assign commit timestamp $ts_{commit} = t_{current}$ for the current transaction and initiate the commit protocol, which either succeeds or fails.
- **put(key, value)**: Buffer the key-value pair; add key to the current transaction's write set.
- **get(key) → value**: Return a consistent value for a key; add key to the current transaction's read set.

We now describe the transaction protocol for processing transactions. A MILANA client performs the following for each transaction.

1. The application starts a transaction by invoking the *beginTransaction* method in the client library.

2. The application then transitions to the processing stage of the transaction.

   Write operation (put):. The client adds the key-value pair to the write set of the transaction, which is buffered in DRAM.

   Read operation (get):. The client returns the value from write or read set, if present. Otherwise it uses ts_{begin} to obtain a consistent version of the key from the primary. The primary returns the youngest committed version with a timestamp $ts_{commit} \leqslant ts_{begin}$ and a boolean that indicates if there is a prepared version for that key with a timestamp $ts_{prepared} \leqslant ts_{begin}$. Note that $ts_{commit} < ts_{prepared} \leqslant ts_{begin}$.

3. The processing stage finishes when the application invokes *commitTransaction* or *abortTransaction*. If the application invokes *commitTransaction* then the transaction either commits or aborts based on the validation result. The application cannot arbitrarily determine to abort the transaction after validation on all participants (primaries) is successful.

4. Validation: the client performs the following steps to validate the requested commit:

   Read-only transaction:. The client performs local validation for a read-only transaction. The local validation check ensures that the keys read in the transaction are from a consistent snapshot (§3.2.3).

   Read-Write transaction:. Like read-only transactions, the client can abort a transaction if the read set was not from a consistent snapshot. Otherwise, the client sets $ts_{commit} = t_{current}$, assigns version $V = \langle ts_{commit}, clientID \rangle$ to all the keys in the write set and starts the 2PC protocol, with the client as the

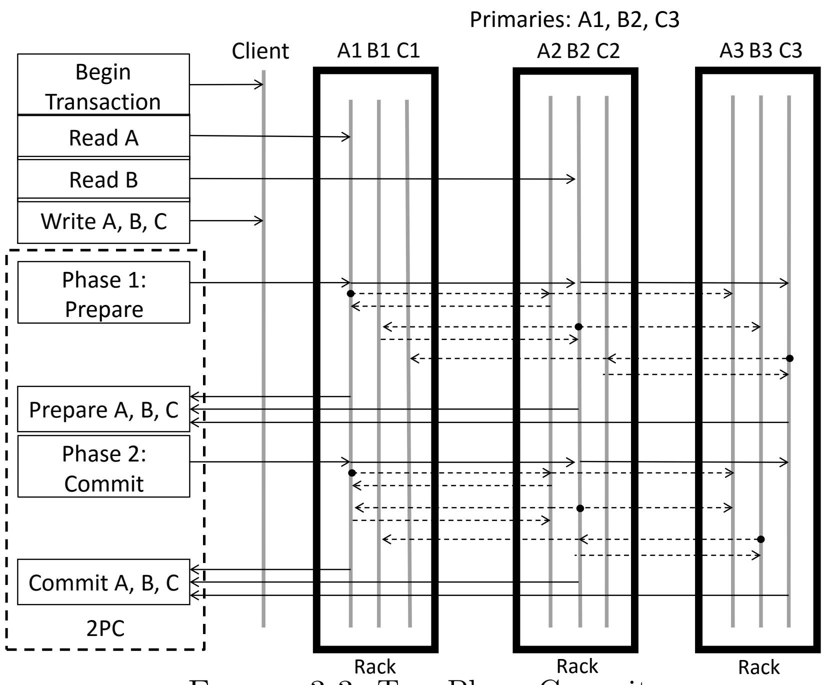


FIGURE 3.3: Two Phase Commit

coordinator, as described below. The client returns the status (SUCCESS / ABORT) to the application after the first (prepare) phase of 2PC.

3.2.2 Two-Phase Commit: Write Validation

Figure 3.3 shows an example transaction with a standard two phase commit (2PC) protocol. On a commit request for a read-write transaction T, the client library initiates 2PC and acts as the coordinator. It first sends a *Prepare()* request to the primary of each participant shard, passing each primary all keys in T's read and write sets for shards that the primary controls. It also passes a list of other affected shards for possible use in recovery (Section 3.2.6).

Each primary uses Algorithm 1 to validate T's keys. T fails validation if it has conflicts that violate transactional serializability. It then propagates the validation decision (SUCCESS/ABORT) along with the write set (on successful validation) and shard list to the backup replicas, waits for f (out of $2f$) backups to respond, and

Algorithm 1 MILANA Primary Validation Algorithm

```
procedure VALIDATE(transaction)
    for each (key, version) ∈ transaction.readSet do
        if key.prepared ≠ NONE then
            return ABORT
        else if key.latestCommitted ≠ version then
            return ABORT
        end if
    end for
    newVersion = transaction.commitTimestamp
    for each (key, version) ∈ transaction.writeSet do
        if key.prepared ≠ NONE then
            return ABORT
        else if key.latestRead ⩾ newVersion then
            return ABORT
        else if key.latestCommitted ⩾ newVersion then
            return ABORT
        end if
    end for
    return SUCCESS
end procedure
```

then reports the decision as its vote to the client/coordinator. If a primary votes to commit T then T is *prepared* at that primary.

The client accumulates the votes from all primaries and determines the outcome: T commits if and only if all primaries vote to commit, else T aborts. The client reports the outcome to the application and then asynchronously notifies all primaries of the outcome.

3.2.3 Local Validation of Read-only Transactions

As mentioned earlier, a MILANA client performs local validation for read-only transactions. Local validation eliminates *two round trips* at validation time: client to primary and primary to backups.

As a transaction T runs, the client issues a read (get) to the primary server for each key K read by T, and satisfies subsequent reads to K from its cache. The client issues gets with T's begin timestamp ts_{begin}. On a get, the primary returns the youngest committed version of K with a timestamp $K.ts_{commit} \leqslant T.ts_{begin}$, and a boolean that indicates if there is a prepared version of K with a timestamp

$K.ts_{prepared} \leqslant T.ts_{begin}$. Note that $K.ts_{commit} < K.ts_{prepared} \leqslant T.ts_{begin}$. The primary also records the read timestamp ts_{begin} in DRAM if it is $> K.ts_{latestRead}$.

Local validation works because a MILANA primary aborts any late-arriving transaction S that attempts to commit a new value for a key K with an earlier timestamp $S.ts_{commit} \leqslant K.ts_{latestRead}$ (see Algorithm 1). Therefore, if K did not have a prepared version when it was read, then it is guaranteed that there can be no prepared version with a timestamp less than T's begin timestamp (ts_{begin}). Thus the client has all the information needed to locally validate T: it can commit T if and only if none of the keys in T's read set had a prepared version at T's read time (ts_{begin}).

Local validation is aided by both SDF and PTP. SDF provides a lightweight mechanism to maintain multiple versions of a key, thus enabling snapshot reads. PTP's low clock skew makes local validation practical from a *performance* standpoint. For a given clock skew ϵ, if a client with a leading clock reads a key K, then a client with a lagging clock has to wait up to ϵ duration before it can commit a transaction that updates K. For NTP, ϵ is on the order of milliseconds, while PTP reduces it to microseconds.

3.2.4 Snapshot Reads

This section describes two scenarios in which an application reads a consistent snapshot even when there are conflicting writes on the same set of keys. For each scenario consider two contending transactions T_1 and T_2 that operate on keys K_1 and K_2. SEMEL holds a version of both keys before either T_1 or T_2 start.

In the first scenario T_1 and T_2 are read-only and read-write transactions, respectively. The start time of T_1 is ts_{begin1} and end time of T_2 is ts_{end2}, where $ts_{begin1} < ts_{end2}$. T_1 starts processing and reads one key (K_1) and has not issued a read for K_2 as yet. T_2 overlaps in execution with T_1 and T_2 is able to commit and update the values of both K_1 and K_2 while T_1 is still executing. Since SEMEL maintains multiple versions, it is able to serve T_1's read of K_2 and the data returned is from a consistent snapshot at time ts_1, even though there is a committed value for K_2 at the later time ts_2. T_1 behaves as though it executed completely before T_2 even started, which is a consistent execution.

In this second example T_1 is a read-write transaction and T_2 is a read-only transaction. The end time of T_1 is ts_{end1} and start time of T2 is ts_{begin2}, where $ts_{end1} < ts_{begin2}$. T_1 starts its 2PC process and sends the commit request for K_1 to shard A, but is delayed in sending the commit for K_2 to shard B. T_2 overlaps in execution with T_1, performing read operations on keys K_1 and K_2. For K_1, T_2 obtains the value that T_1 wrote, since the shard received the commit. However, shard B contains both a prepared value and the latest committed value for K_2, since it does not know if T_1 will commit. In this case, shard B returns the last committed value and a boolean (= true) that indicates that there is a prepared version of the key. The client can then determine that it did not read from a consistent snapshot and would abort T_2.

3.2.5 Version Management

Milana leverages Semel's watermarking-based garbage collection to manage versions and satisfy long-running read-only transactions. Each Milana client periodically broadcasts the timestamp t_d of its latest decided (committed or aborted) transaction to all primaries. The minimum over the t_ds becomes the watermark t_w. Since PTP time increases monotonically, no client can have a transaction begin time that is less than the watermark. Therefore, the Semel garbage collector only needs to keep the youngest version with a timestamp $\leqslant t_w$ and can discard all prior versions.

Consider an active long-running read-only transaction T with a begin timestamp ts_{begin}. Then the watermark $t_w < ts_{begin}$. Therefore, a Milana server retains at least the youngest version of any key K with a timestamp $\leqslant ts_{begin}$, so T can read a version from a consistent snapshot at its ts_{begin}. The watermarking scheme dynamically tunes the number of versions kept for all keys and is a function of the duration of transactions: fewer versions are kept when transactions are short, and the threshold increases as longer transactions are added to the mix.

3.2.6 Recovery

This section describes what happens if a client or storage server (e.g., a primary) fails during the process of committing a transaction. Milana assumes fail-stop (non-byzantine) failures.

Client Failure.. If the client fails during 2PC, then the participants (primaries) time out waiting for a commit or abort decision for a prepared transaction T. T is blocked until its commit/abort status is known. This situation does not affect any transactions operating on key sets that are disjoint from T's read/write sets. However, the participating primaries are forced to abort any transaction that attempts to read/write any of the keys in T's read or write set, until T's commit/abort status is known. In such a case, one of the participating primaries is designated as a backup coordinator for T. The backup coordinator can use the Cooperative Termination Protocol (CTP) [16] to determine if T should commit. The backup coordinator queries the other participating primaries for the status of T, and takes appropriate action. The states are *Received Commit, Received Abort, Prepared, Sent Commit, Sent Abort* and the actions can be any of the following:

1. If any primary received a commit or abort then T should be committed or aborted since the client made a decision only after receiving a response from all the primaries.
2. If any primary did not receive a prepare request for T, then all primaries can

agree to abort T because the client does not commit a transaction until it receives a response from all primaries for its shards.

3. If any primary responded with ABORT to the prepare request, then all primaries abort T.
4. If all primaries responded SUCCESS for the prepare request then the backup coordinator commits T.

Replica Failure / Recovery.. If a backup replica of a participant shard fails during 2PC, it does not block any transaction as long as a majority of replicas for a shard are available to store transactions. However if a primary of a participant shard fails then it would block all transactions involving that shard. A new primary must be elected (failover) in order to unblock any running transactions and resume service.

Distributed transactions require a protocol to ensure atomicity and consistency of keys and shards across failures of servers and clients. Many storage systems that provide transactional semantics and fault tolerance use both a transaction protocol and a replication protocol, which enforce a serial ordering in two places: transactions across shards and updates among replicas. This redundancy can add latency and reduce throughput. Since the transaction protocol enforces ordering among the transactions and consequently the updates, the replication protocol does not need to also enforce ordering. This observation was previously exploited to reduce write transaction latency in TAPIR [130] by allowing inconsistent replication.

Semel and Milana replicate using a primary-backup approach: all the updates to a shard flow through the primary. As a result, the Milana primary has the consistent view (an up-to-date transaction table) needed to validate transactions without involving the backups, reducing validation costs and abort rates. Since the backups play no role in validating or executing transactions, their only purpose is to provide fault tolerance, as in Semel, and not consistency, which is handled by the

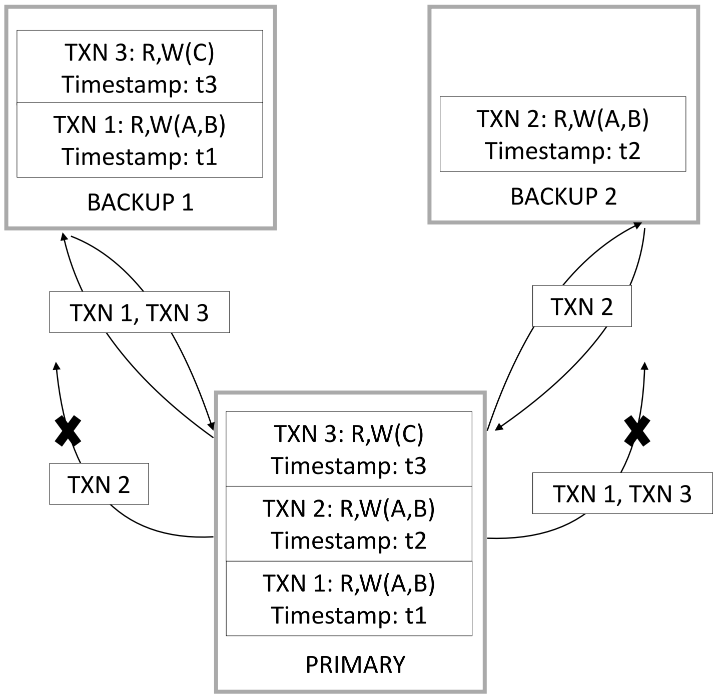


Figure 3.4: Milana Relaxed Backup Updates

Milana code on the primary.

Once the primary validates a transaction, it can propagate updates and prepare records to the replicas in any order, as long as a new primary can rebuild the transaction table during failover. Figure 3.4 shows how Milana relaxes backup update ordering and how this can tolerate transient failures. In this example there are three storage servers, a primary and two backups. The primary requires only one of the two backups to acknowledge a prepare and commit of a transaction. In this case, backup 1 acknowledges prepare and commit of transactions 1 and 3, while backup 2 acknowledges prepare and commit of transaction 2. In another scenario, backup 1 acknowledges prepare of transactions 1,2 and 3, and backup 2 acknowledges the commit for these transactions. In both cases, traditional replication would be forced to signal an error, and possibly abort transactions since the backups did not receive updates in sequence order. Milana eliminates these scenarios by reconstructing the

Algorithm 2 Milana Recovery Merge Algorithm

```
procedure mergeLog(transactions, table = NULL)
    for each T ∈ transactions do
        if T.status == COMMITTED then
            table.insert(T)
        else if T.status == PREPARED then
            if T.participants == 1 then
                table.insert(T)
            else
                decision = queryParticipant(T, participants)
                if $decision \in COMMIT, PREPARED$ then
                    table.insert(T)
                end if
            end if
        end if
    end for
    return table
end procedure
```

correct overall order during failure recovery.

Since Semel does not enforce strict global ordering for all updates during replication, therefore in Milana the new primary must be brought to a consistent state before it can start servicing transaction requests. The new primary can always reach a consistent state if there are $f+1$ replicas available (out of $2f+1$), which are needed for a majority quorum. For $f+1$ available replicas, there must always be at least one replica that has seen any given transaction committed or prepared by the previous primary. Therefore the new primary has access to all the transactions and can rebuild the data versions and transaction table by merging the updates from all the replicas, as shown in Algorithm 2. If a transaction prepare or commit record is not present on at least one replica at recovery, then the previous primary could not have obtained a majority for the operation, and therefore could not have acknowledged the request. In this case, 2PC recovery restores the state of these transactions, as detailed above (CTP).

The new primary can then apply all the successfully committed transactions without any validation. It can also apply a successfully prepared transaction that included a single shard because that would have been committed. For a prepared transaction involving multiple shards, the new primary must contact the primary of the other shards to determine whether the transaction committed. The transaction is committed or aborted if any shard responds with a COMMIT or ABORT, respectively. If all participants respond with a prepared status then the transaction is still outstanding and should be committed and a response sent to the client.

After creating the transaction table, the new primary propagates the table to the backups to bring them to a consistent state. It then populates $ts_{prepared}$ and $ts_{latestCommitted}$ values for each key. These values can be inferred from prepare requests obtained from other replicas during recovery and from the version stamps included with each write (see Figure 3.1), respectively.

The new primary cannot populate $ts_{latestRead}$ for keys because these values are not persisted nor are the backups informed about the latest read timestamp of a key while servicing a get request. Validating new transactions without these values can violate serializability. Consider the following scenario: a read-only transaction T_a issues a get request for key K with a timestamp t_2, a primary returns a version of K with a timestamp t_0 and T_a commits. Now a failover occurs and a new primary allows a read-write transaction T_b to commit that creates a new version of K with timestamp t_1 ($t_0 < t_1 \leqslant t_2$). This violates serializability because T_a (already committed) should have read T_b's write.

Milana uses *leases* [54] to avoid this scenario. A primary in Milana obtains a periodically renewed lease from at least f backups to process any get request with a timestamp $< t_{lease}$. After recovery, the new primary waits for its local clock to advance past t_{lease} before servicing transaction requests for its shard. As an optimization, we can combine this mechanism with leases used for avoiding spurious failovers in primary/backup based replicated state machine protocols [84].

In all failure scenarios (client, primary, backup replica or some combination of the three) a decision can be made on any outstanding transaction and service can be resumed as long as a majority of replicas ($f + 1$) of all shards are available.

3.2.7 Comparison with TAPIR

There are some similarities between Semel/ Milana and TAPIR [130]. Like Semel, TAPIR is based on an inconsistent replication approach that decouples replication from ordering for lowering write latencies. The Semel inconsistent replication protocol differs from TAPIR in that Semel uses primary/backup replication with a designated primary for each shard, rather than having the client propagate the operation to symmetric replicas, as in TAPIR. Both Milana and TAPIR build on top of inconsistent replication to provide transactional semantics and use OCC for ordering operations. To reduce read latencies, TAPIR clients read data from the nearest replica during a transaction. This approach also helps balance the read load across replicas. In contrast, all reads in Milana are serviced by the primary but this requirement can be relaxed for read-write transactions, which can read data from the nearest replica and validate at the primary before commit.

Validation in TAPIR succeeds only after a majority of replicas for each affected shard agree to validate the transaction. TAPIRs approach of eliminating the primary saves a round-trip latency for each prepare. This may be a substantial saving if the primary resides in a different datacenter, but it requires all replicas to maintain additional state and validate both read-only and read-write transactions. This is less energy-efficient since additional compute and memory resources are needed, and also consumes precious memory / storage bandwidth on all replicas. In contrast, Milana clients validate read-only transactions locally, which eliminates two round trips. For read-write transactions, once validation on a primary is complete, the updates and prepare records can propagate to backups in any order. Since the backups play no role in validating transactions, their only purpose is to provide fault tolerance: validation imposes no memory or compute cost on the backups.

In summary, TAPIRs design is suitable for a geo-replicated system, where eliminating the primary saves a cross-datacenter round trip. In contrast, Milana targets intra-datacenter storage and its approach reduces total validation costs: every transaction validates on a single node and not on all replicas as in TAPIR. The tradeoff is that all read-write transactions require an extra round trip (from primary to backups), but no extra messages are sent.

3.3 Evaluation

We present preliminary results for our prototype implementations of Semel and Milana. We use the Open-Channel SSD framework [18] for our SDF implementation. In software-only mode, the framework emulates the internals of a NVMe SSD and supports timing simulation of I/O operations. We extend the timing-only simulation with functional emulation by adding support for storing data values and IOCTLs that provide a get, put and erase functionality for flash blocks. We also added the capability to specify latencies for read page, write page and erase block operations. These operations are simulated by adding a timer interrupt in the kernel and the request is acknowledged after the timer expires.

The performance of SDF suffers due to emulation limitations: crossing the kernel boundary for submitting each I/O request and lack of batched interrupts for request completion. An open-channel compatible NVMe SSD does not suffer from these limitations since the NVMe standard provides user-space queues for submitting requests and also supports batching interrupts to reduce overhead.

Experimental Setup:. We use a set of Linux virtual machines from a single ExoGENI [13] site for both clients and servers. The client VMs have 2 cores, clocked at 2.6 GHz and 6 GB of DRAM. The system clocks on client servers are synchronized using PTP software timestamping or NTP. The storage server VMs have 8 cores, clocked at 2.6 GHz and 32 GB of DRAM. The cores of all storage VMs are pinned to allocate from a specific NUMA zone on the host. Each storage VM is configured with an emulated SSD, backed by 12 GB DRAM, with a hardware queue depth of 128. The SSD has a page size of 4KB and there are 32 pages in a block. A page read, write time is 50 μs and 100 μs respectively and it takes 1 ms to erase a block. We use Ubuntu 14.04 with kernel 4.6 on all VMs and our code is compiled using gcc version 4.9 with -O3 flag enabled.

In all our experiments, we set the key size to 16B and a $\langle key, value, version \rangle$ tuple is 512B. Although our implementation supports variable-sized keys and values, we decided to use a fixed size for evaluation since it allows efficient packing of data. As a flash page is 4KB in size, we employ a *packing logic* in the FTL that waits for up to 1 ms (tunable) to pack data of multiple keys into a page (see Figure 3.1 for data layout). We run each experiment for 15 minutes to ensure that garbage collection is running in the background (for all runs with non-zero put request %).

3.3.1 SEMEL *Evaluation*

To elucidate the advantages of implementing multi-versioning within the FTL, we implement a single-version generic FTL and a separate multi-version KV store on top of a generic FTL. This multi-version KV layer implements its own lookup, request handling and garbage collection logic that is *separate* from that of the FTL. The multi-version layer operates at 4KB granularity and uses a log-based approach to write data to the SSD. We refer to the single-version generic FTL as SFTL, the split multi-version layer on top of a generic FTL as VFTL and the unified multi-version FTL as MFTL. To ease garbage collection, SFTL, MFTL and the multi-version KV layer in VFTL reserve 10% of available capacity for remapping data.

We first measure the throughput and latency for KV operations by emulating a single SSD for both MFTL and VFTL. For these experiments, we populate the device with 2 million keys and use a micro-benchmark to issue KV requests for varying get

Table 3.1: Single SSD Multi-version FTL Performance

Get %	Throughput Kilo Reqs/Sec		Average Latency (μs)			
			Get		Put	
	VFTL	MFTL	VFTL	MFTL	VFTL	MFTL
100	351	456	68.1	59.9		
75	295	430	363.1	62.9	568.5	872.8
50	217	277	516.6	70.3	673.8	859.0
25	215	189	435.6	77.7	659.8	895.8

request percentages. Table 3.1 shows our results. As expected, MFTL delivers up to 45% higher throughput, and up to 7x lower latency compared to VFTL. For 25% get rate, VFTL performs better since it has a lower packing delay. A key-value pair is 512B in size, thus our packing logic waits for up to 1 ms to pack data of multiple keys (puts or remapped keys) into a page. Since VFTL has less available space compared to MFTL (10% capacity reserved at two levels), it performs more garbage collection, has more data (keys) to remap and therefore, incurs less packing delay. For 25% get rate, VFTL remaps 15% more data than MFTL.

We also obtain latency and throughput of a distributed KV store, for three SEMEL implementations (DRAM, MFTL, VFTL). DRAM is an in-memory solution that uses a simple hash table to maintain versions in main memory, and implements the same garbage collection as the others. We use 5 clients to issue asynchronous get/put requests (queue depth: 32) to a varying number of storage server shards. Each shard consists of 3 storage servers: 1 primary and 2 backups. The primary of a shard is responsible for servicing client requests and replicates all put requests before acknowledgment. These experiments again vary the fraction of get requests, and each run is 10 mins. We present results for 3 shards (3 primaries and 6 backups) in Table 3.2. Results for 1 and 2 shards are qualitatively similar.

Table 3.2: Distributed Multi-version KV Store Performance

	Throughput	Average Latency (ms)	
Get %	Kilo Reqs/Sec	Get	Put
DRAM			
100	390	0.41	
75	285	0.46	0.87
50	223	0.48	0.93
25	190	0.52	0.99
MFTL			
100	297	0.55	
75	202	0.51	1.64
50	118	0.76	1.88
25	91	0.84	1.99
VFTL			
100	240	0.68	
75	154	0.8	1.73
50	110	0.88	1.99
25	95	0.76	1.97

Table 3.3: Retwis Configuration

Transaction Type	Num GETs	Num PUTs	Workload %
Add User	1	2	5
Follow User	2	2	10
Post Tweet	3	5	35
Get Timeline	rand (1,10)	0	50

3.3.2 MILANA *Evaluation*

We evaluate MILANA by exploring the impact of multi-versioning and precise time on transaction abort rates. Our first experiment evaluates the impact of multi-versioning vs. using a single version FTL (MFTL vs. SFTL). We use a single VM for this experiment to eliminate clock skew. The VM hosts a storage layer and runs varying number of clients that issue transactions to the storage layer. We populate the storage layer with 2 million keys. The clients run the Retwis benchmark [73] with the transaction mix shown in Table 3.3. Each client has one outstanding transaction and all transactions are executed sequentially. We run this experiment for varying number of clients and to simulate key-sharing, we also vary the Retwis Contention

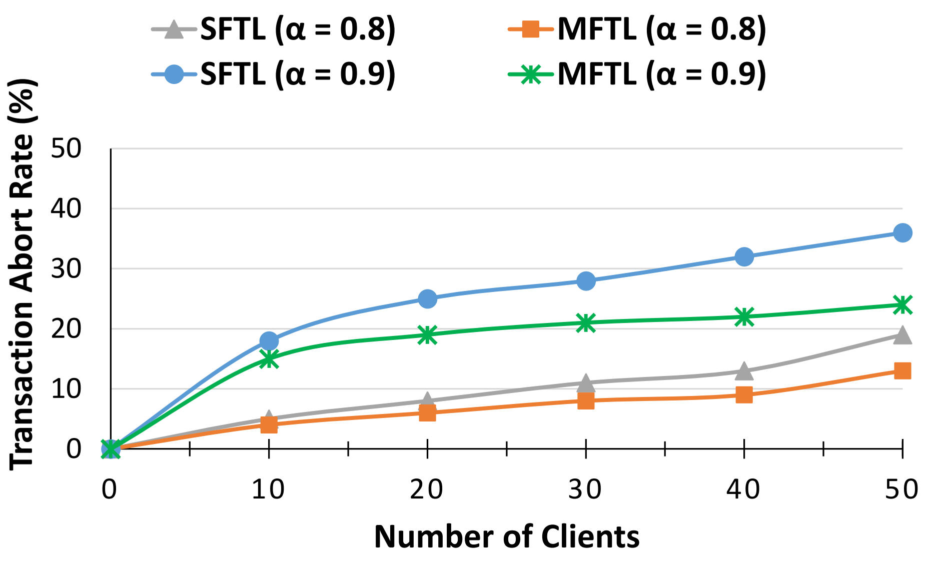


FIGURE 3.5: Transaction abort rate for varying number of clients

parameter (α).

Figure 3.5 shows transaction abort rates versus number of clients for a single and multi-version FTL. From the results we see that with increased key contention, a multi-version FTL helps reduce abort rates since tardy read-only transactions are able to read from a consistent snapshot and commit whereas these transactions are aborted on a single-version FTL. Abort rates using VFTL are qualitatively similar to MFTL and omitted for clarity.

To evaluate the impact of clock skew on transaction abort rates, we use 3 storage (1 primary and 2 backups) and 5 client VMs and synchronize clocks on the client VMs using either PTP software timestamping mode or NTP. The NTP daemon is free to choose the best master based on NTP's criterion (lowest jitter). We populate the storage VMs with 2 million keys. Each client VM runs 4 independent instances of the Retwis benchmark (20 instances in total). Each instance executes one transaction at a time and retries an aborted transaction with the same set of keys and without any wait.

Figure 3.6 shows transaction abort rates versus the amount of contention in the

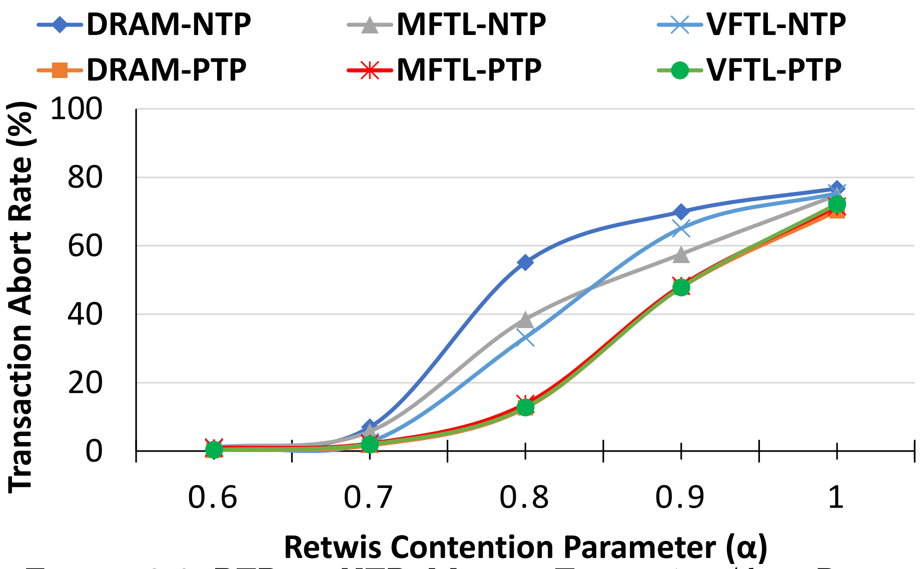


FIGURE 3.6: PTP vs. NTP: MILANA Transaction Abort Rates

Retwis benchmarks using MILANA with a DRAM backend, VFTL and SEMEL's MFTL. From these results we see that PTP provides superior performance for all storage backends due to the tighter clock synchronization. For NTP the DRAM backend incurs the highest abort rates, as expected, since the faster write time requires lower clock skew across clients. VFTL also incurs slightly higher abort rates compared to MFTL due to lower write latency (see Table 3.1). NTP shows an average skew of 1.51ms among clients, while software timestamped PTP has average skew of 53.2 μs.

To evaluate throughput and latency, we deploy Milana over 3 shards where each shard has 3 replicas. We populate the system with 6 million keys and ran the Retwis workload with 75% read-only transactions (5%, 10%, 10% and 75% breakdown - Table 3.3) for an increasing number of clients. We perform this evaluation for all 3 storage backends (DRAM, VFTL and MFTL) and also measure the impact of local validation (LV).

Figure 3.7 shows the average transaction latency vs. throughput for the 3 storage backends. From the results we see that MILANA with local validation is able

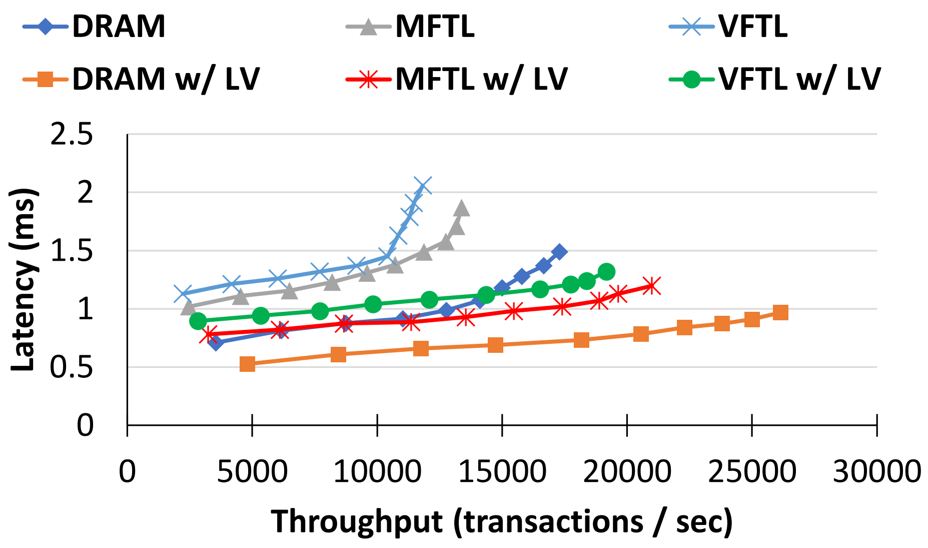


FIGURE 3.7: Retwis Transaction Latency vs. Throughput

to achieve up to 55% higher throughput and 35% lower latency. Local validation enables a MILANA client to independently make a commit or abort decision for a read only transaction, without affecting consistency. This saves *two* round-trip times for validation (§ 3.2.3) and helps reduce transaction latency. These results also show that MFTL achieves 15% higher throughput and 10% lower latency compared to VFTL. VFTL w/ local validation achieves higher throughput than MFTL w/o local validation, showing the importance of local validation.

3.3.3 Comparison of Local Validation Techniques

This experiment compares Centiman's local validation approach [39] with that of MILANA. Centiman uses a watermark-based technique that allows a client to locally validate a read-only transaction, if it read a consistent snapshot of keys with timestamp < watermark. Otherwise, a Centiman client reverts to remote validation. Centiman's approach works well for low contention scenarios. But under high contention, more key-sharing between transactions increases the probability of a read-only transaction reading a version younger than the watermark and failing the local validation

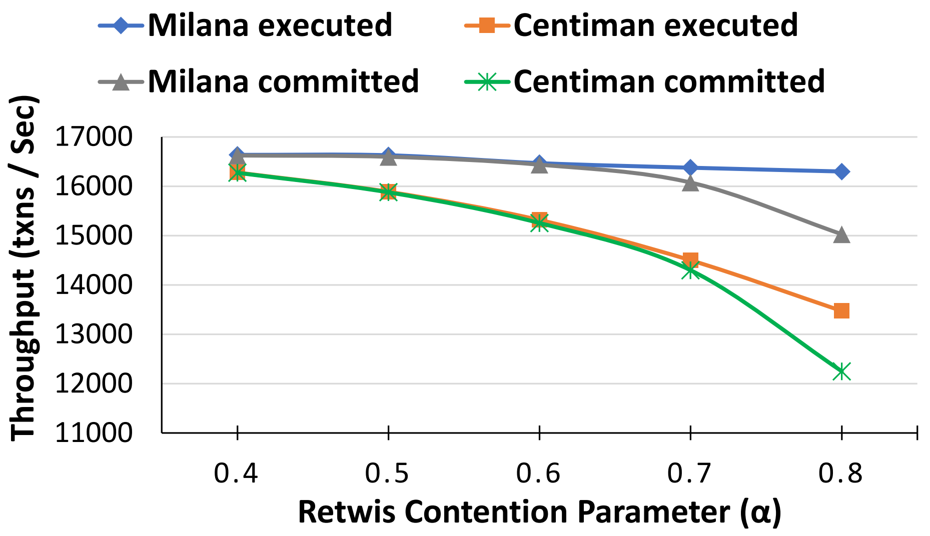


FIGURE 3.8: Comparison of Local Validation Techniques

check. Centiman can counteract this by faster dissemination of watermarks, but this increases coordination overhead.

For this experiment, we use 3 storage and 5 client VMs. The storage VMs are used for creating 3 shards, each VM stores data on SSD (MFTL). We populate the shards with 6 million keys. To eliminate impact on throughput, we use the same number of validators (3) with Centiman (one per shard) and these validators run on the storage VMs. We do not use replication in MILANA since Centiman's validators do not replicate. On each client VM, we run 6 independent instances of the Retwis benchmark (30 instances in total) with 75% read-only transaction workload and vary α to simulate key contention. Clients disseminate watermark after every 1,000 transactions. The clocks on client VMs are synchronized using PTP software timestamping.

Figure 3.8 shows the results. Under low contention ($\alpha = 0.4$), Centiman achieves a similar throughput as MILANA. However, the throughput drops under increasing contention as Centiman's local validation check fails thereby forcing a remote validation. Centiman locally validates 89% of read-only transactions for $\alpha = 0.4$ and this value drops to 25% for $\alpha = 0.8$. One the other hand, MILANA can perform local validation for *all* read-only transactions and therefore achieves 20% higher throughput under high contention settings. Both systems observe similar abort rates.

3.4 Related Work

There is a long history of work exploring distributed systems, datacenter services and flash storage systems. This section places our work in context relative to a subset of distributed transactional storage systems and flash-based storage systems.

There are numerous client-server distributed systems, such as key-value services; however, many of these systems lack support for updating multiple objects atomically [29, 37, 70] or restrict partitioning [12] due to the complexity of supporting distributed transactions.

Thor [3] introduced loosely synchronized clocks for OCC and performed validation on the storage servers. Our approach differs by maintaining multiple versions of a key, which helps avoid conflicts between concurrent read and write transactions and performs local validation at the client for all read-only transactions.

TAPIR [130] and Spanner [31] along with some of the differences from our work were discussed previously. A main difference is our focus on intra-datacenter operation vs. inter-datacenter. Centiman [39] uses OCC to support intra-datacenter distributed transactions but validations are performed on a different set of servers called validators. This helps in a multi-tenant datacenter where different applications can have their own validators. However this approach involves increased coordination and suffers from limited availability since transactions are made durable on a single client before they are committed. It optimizes for local validation of read only transactions using watermarks but needs remote validation under high-contention settings.

Other intra-datacenter systems focus on in-memory computation [93, 122, 64, 41, 42, 110, 75]. RamCloud [110, 75] is a key-value storage system that provides exactly once semantics like SEMEL and transactional semantics like MILANA. However, it does not maintain multiple versions of a key. FaRM [41, 42] is optimized for performance over RDMA, it maintains multiple versions of an object and supports strictly serializable distributed transactions. Neither of these systems have MILANA's inconsistent replication.

Calvin [116] buffers transaction requests and creates a transaction schedule from the received requests. All replicas then execute transactions deterministically using the defined schedule. However this approach restricts the type of transactions since it needs the read and write set of a transaction to be pre-declared in the transaction request. MILANA does not have this requirement.

Hyder [17] uses a shared-storage made up of flash chips to store data. Clients record transactions on the flash storage and also broadcast their intent to all the other clients, which allows clients to then determine if a transaction can commit. The broadcast can be a scalability issue and our approach differs since we allow clients or storage servers to scale independently and we try to minimize coordination wherever possible.

Flash based KV stores have been proposed in prior works [10, 34, 36, 82, 88]. Other systems eliminate the FTL indirection [88, 58, 131, 62, 111]. Each of these systems has some aspects included in SEMEL; however SEMEL differs from these works by maintaining multiple versions of a key and providing transactional semantics for updating multiple keys. Previous systems that maintain multiple versions [113, 124, 123] require a snapshot activation to access prior versions. Several studies [88, 101, 100, 62, 112, 22, 132, 27] propose a cooperative hardware and software based approach to exploit the performance of non-volatile memories. SEMEL and MILANA leverage the functionalities accorded by these designs.

3.5 Summary

This chapter presents a distributed transactional key-value storage system (called Milana) as a layer above a durable multi-version key-value store (called Semel) for read-heavy workloads within a datacenter. Both systems exploit precise synchronized clocks and software-defined flash to simplify protocols and remove layers of abstraction. Semel exploits precise synchronized clocks to relax the ordering constraint in primary-backup replication protocols, and uses software-defined flash to exploit remap-on-write behavior of SSDs to maintain a time-ordered sequence of versions for each key efficiently and durably. Milana adds a variant of optimistic concurrency control above Semel's API to service read requests from a consistent snapshot and to enable clients to make fast local commit or abort decisions for read-only transactions.

Evaluations of our prototype implementations reveal that Semel achieves 20%-50% higher IOPs than a traditional separate version and flash management approach. Furthermore, by using PTP, Milana reduces abort rates by up to 43% over NTP for transactions with high-contention, due to the tighter clock synchronization across servers. We also demonstrate that Milana's use of local client validation reduces latency by 35% and increases throughput by 55%.

4

Kairos

This chapter proposes a new approach to *inter*-transaction caching and concurrency validation for scalable low-latency stores. Client caching of active data is standard in client-server transactional stores since Thor [3]. While *intra*-transaction caching is trivial with concurrency control for serializable transactions, Thor and later systems that support *inter*-transaction caching typically use *explicit invalidation* to keep client caches consistent across transaction boundaries. This approach is similar to network file systems and other client-server storage systems using classical callback leases [54] (see See 4.1). However, explicit invalidation introduces substantial cost for high-performance stores with fine-grained concurrency control, such as transactional stores in the datacenter (Section 4.4). It also complicates the implementation. As a result, many recent transactional key-value stores do not address inter-transaction caching at all [31, 116, 41, 75, 39, 130, 92].

Full support for caching is important for performance, particularly under read-dominated workloads. Several recent works emphasize the importance of caching for datacenter services. For example, auto-sharding systems (load balancers) like Slicer [4] seek to bound the *spread* of requests to each data item across application servers, and show substantial improvements to cache effectiveness. (This idea is a form of locality-aware request distribution [102].) NetCache [61] embeds caching of hot data into the network to reduce hotspots caused by skewed power-law popularity distributions, which are common in standard workloads [30, 11]. However, NetCache does not provide transactional semantics. It is an open question how best to obtain the benefits of client caching with transactions (Section 4.5.)

This chapter presents KAIROS[1], a transactional key-value store that supports inter-transaction caching *without* explicit invalidations and sharded transaction validation. KAIROS builds on the approach of Milana [92] by using precise synchronized clocks [60, 31, 53, 76] to enable physical time-based consistency integrated with transactional concurrency control and adds support for inter-transaction caching and sharded validation

KAIROS is a client-server transaction system that implements transactional serializability using optimistic concurrency control (OCC [68]) based on physical clocks, a technique pioneered by Thor [3]. KAIROS leverages *sharded validation* from Centiman [39] to decouple transaction validation from the servers, so that validation scales independently of the storage tier. KAIROS adapts this sharded validation to support inter-transaction caching (see §4.3.6), without the cost of explicit invalidation.[2]

Precise synchronized clocks also enable a simple, stateless, time-to-live (TTL) protocol for cache consistency in KAIROS. Storage servers in KAIROS hand out leases to cache popular keys in the usual fashion. We refer to KAIROS leases as "soft" because the lease manager need not track leases or send invalidations (callbacks), although it may do so as an optimization for write-heavy keys[3] Instead, cache consistency in

[1] KAIROS means "appropriate time" in Greek

[2] It is important to distinguish two similar terms that are independent: *validation* refers to a step of optimistic concurrency control that occurs when a client transaction prepares to commit, while *invalidation* refers to a server callback to flush a stale value from a client cache.

[3] Called *tear-off* blocks in a hardware coherence protocol [74].

Kairos is based on low-cost *self-invalidation* [74] when the lease expires. With soft leases, a client may read stale data from its cache with some probability; Kairos uses OCC validation as a fallback to restart any transaction that reads stale data. *The central challenge for this approach is to set lease times to balance the hit ratio with the cost of stale reads.* Kairos servers use the observed inter-access (read and write) times of popular keys to adapt lease durations *dynamically* (see §4.2.3) for each key to optimize this tradeoff according to an analytical model. The classic paper on lease-based consistency [54] suggested adapting lease times based on access parameters and an analytical model, but we are not aware of any work that develops this idea

Evaluation of a Kairos prototype under a YCSB workload [30] reveals that inter-transaction caching alone improves throughput by 1.86x relative to a baseline system with only intra-transaction caching; adding sharded validation further improves throughput by a factor of 2.28 under a workload with a hotspot that saturates a storage primary. Furthermore, our evaluation shows that lease-based inter-transaction caching can operate at a 62.5% higher scale while providing 1.55x the throughput of classical callback leases (explicit invalidation) in workloads with hot keys.

The rest of this chapter is organized as follows. Section 4.1 covers background. Section 4.2 presents caching with leases in Kairos. We present the design of Kairos in Section 4.3. Section 4.4 presents results from a prototype of Kairos. We discuss the related work in Section 4.5. Finally, we summarize in Section 4.6.

4.1 Background

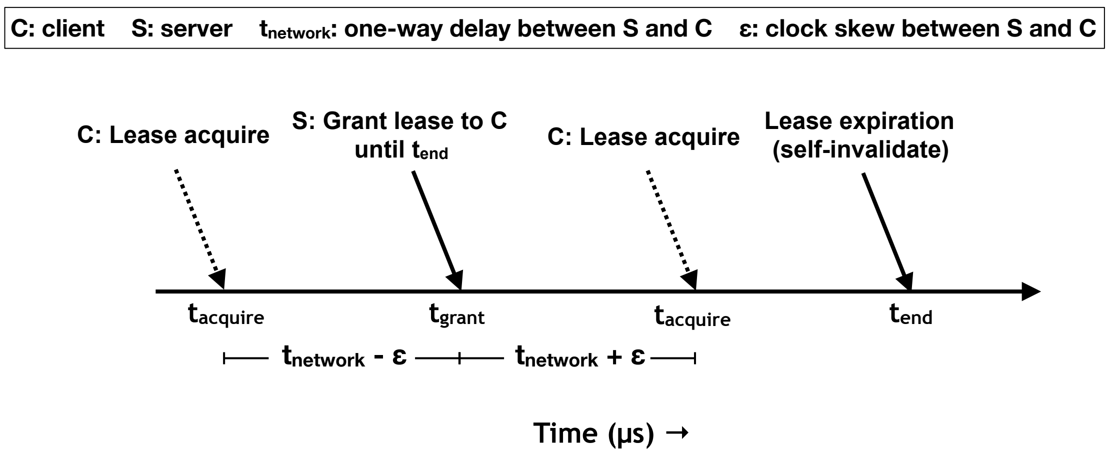
This section describes how Kairos leverages precise clocks for cache consistency.

Cache consistency. Many storage systems implement cache consistency using classical callback leases [54]. A server S grants a lease to a client C to cache an object O and records the lease. If S receives a request to update O from another client, it retrieves its record of C's lease, sends C a callback on the lease, and waits for a reply (synchronous) before processing the update. Each lease is valid for a *duration* (term) chosen by the server: a lease specifies a timestamp after which the lease expires. C considers its cached copy of O to be stale when its lease expires. In a network file system (e.g., [86] or NFSv4) the lease terms may be tens of seconds.

The key observation underlying cache consistency with *dynamic self-invalidation* in KAIROS is that OCC frees the server from the need to send callbacks. If the lease term is "short enough", then the client marks its copy of O as stale (self-invalidates) before another client updates O. If the lease term is "too long", then any client transaction that reads the stale data fails the OCC validation checks and is aborted. The *ideal* term is one that allows the lease holder (client) to cache the data long enough to reap some cache hits, and then self-invalidate before it reads stale data. KAIROS servers adapt the lease terms for popular keys in a dynamic way according to an analytical model that considers the key reference frequency and read/write ratio (see §4.2). Dynamic self-invalidation offers lightweight cache consistency without the server overhead to maintain state records and without the network cost and latency of callbacks. The key challenge is for a system to choose lease terms close to the ideal, as measured by the rates of fresh hits and aborts due to stale reads.

Precise self-invalidation. KAIROS meets this challenge by using precise clocks to timestamp transaction operations and to set lease terms. With advances in network technology, the one-way network latency ($t_{network}$) is < 10 μs [53]. Storage latencies for stores based on DRAM, NVMs, or SSDs are on similar scales. Therefore, the inter-access times to objects in a transaction can also be in the μs range. Consequently, ideal lease durations may reflect similar time scales.

In this scenario, clock skew becomes a critical issue for lease-based self-invalidation. Figure 4.1 illustrates the impact of clock skew on a lease duration perceived by a client. A client with a lagging clock may perceive the lease expiration time as fur-



C: client S: server $t_{network}$: one-way delay between S and C ε: clock skew between S and C
C: Lease acquire
S: Grant lease to C until t_{end}
C: Lease acquire
Lease expiration (self-invalidate)
$t_{acquire}$
t_{grant}
$t_{acquire}$
t_{end}
$t_{network}$ - ε
$t_{network}$ + ε
Time (μs) →

FIGURE 4.1: Impact of clock skew, $\epsilon >> t_{network}$ with NTP

ther in the future, so it holds the lease for longer, which increases the probability of reading stale data; similarly, if the client has a leading clock, it expires the lease early, compromising its hit rate.

In the standard Network Time Protocol (NTP), pairs of hosts synchronize their clocks with messages, yielding clock skew $\epsilon >> t_{network}$ because queuing delays (on servers and within the network) impact the messaging time. The PTP standard avoids this drawback by assigning timestamps to packets on a server NIC and using "transparent" switches which record the ingress and egress time of each clock synchronization packet to account for queuing latencies accurately. As a result PTP yields $\epsilon \leqslant t_{network}$.

4.2 Inter-Transaction Caching

This section describes lease-based caching (Section 4.2.1), compares it with other techniques (Section 4.2.2) and presents an analytical model to calculate *ideal* lease duration (Section 4.2.3).

4.2.1 Self-Invalidation with Soft Leases

In lease-based caching, storage servers hand out leases for caching keys on clients. A lease for a key allows a client to cache and read the key from its local cache until lease expiration, without requiring any remote communication with the server.

Kairos calculates lease duration (term) for a key K based on the observed inter-access (read and write) times of K. We expect that updates to K are independent rather than arriving at regular intervals (although this may occur in some scenarios). Therefore, the inter-arrival times follow a probability distribution (e.g., exponential) (Section 4.2.3). Any chosen lease duration for K leaves some probability that an update to K arrives before the lease expires. In this case, the value is still active on one or more client caches, leaving a window for a client to read a stale value for K. Stale hits impact forward progress and lead to lower transaction commit rates.

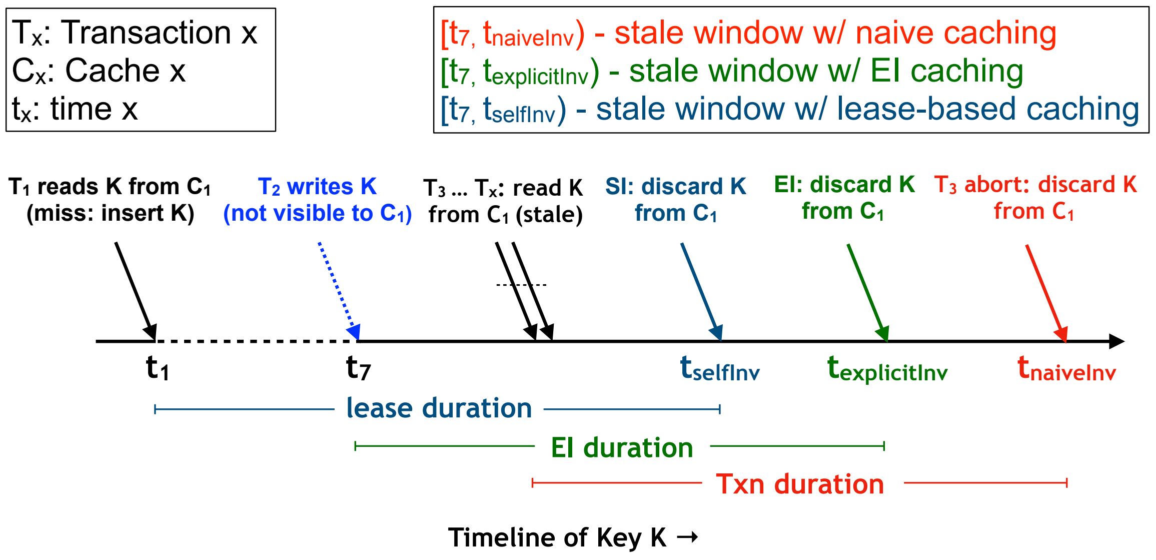
Kairos approximates an *ideal* term for each lease (Section 4.2.3). The ideal lease duration maximizes the expected number of fresh hits and minimizes stale hits.

4.2.2 Comparison of Caching Techniques

Here we use an example to describe the impact of leases on client cache consistency and set our approach in context with other caching techniques: 1) naïve caching, and 2) explicit invalidation (EI).

Assumptions. We assume look-through client-side caches [80] in the example. All read requests in a transaction are first queried in the cache and hits are serviced immediately. On a miss, the data is fetched from the server and stored in the cache. The design choice to have look-through caches has minimal latency impact because caches are local to the client and no remote communication is involved in querying a cache.

Motivating example. Figure 4.2 shows the timeline of operations on a key K. K is brought into cache C_1 at time t_1 by transaction T_1, which successfully commits



Tx: Transaction x
Cx: Cache x
tx: time x
[t7, tnaiveInv) - stale window w/ naive caching
[t7, texplicitInv) - stale window w/ EI caching
[t7, tselfInv) - stale window w/ lease-based caching
T1 reads K from C1 (miss: insert K)
T2 writes K (not visible to C1)
T3 ... Tx: read K from C1 (stale)
SI: discard K from C1
EI: discard K from C1
T3 abort: discard K from C1
t1
t7
tselfInv
texplicitInv
tnaiveInv
lease duration
EI duration
Txn duration
Timeline of Key K →

FIGURE 4.2: Impact on client cache consistency with various caching techniques

soon after (not shown). At time t_7, another transaction T_2 commits and updates K from a different client (not visible to C_1), creating a new version and rendering the cached copy in C_1 stale; any transaction that reads K from C_1 after t_7 reads a stale value and aborts at validation time.

The *stale window* is the interval from t_7 to the time C_1 discards K. This window determines the number of stale hits and the abort rates. Next we consider the stale window with different caching techniques.

Naïve caching. Naïve caching serves as a baseline straw man. In this approach, a client caches aggressively and discards a cached key only when it learns of a stale read by a local transaction that fails validation. Inter-transaction caching in Sundial [129] is similar to naïve caching (Section 4.5).

Figure 4.2 shows the consistency challenge with naïve caching: a value is determined to be stale only after the first transaction to read the stale value completes (T_3 in the figure). In other words, the stale window with naïve caching is proportional to the length of a transaction; longer transactions result in more stale hits and higher transaction abort rates.

Explicit Invalidation (EI). In EI caching, servers track *sharers* (client caches) that cache a copy of a key K and send an invalidation request (callback) to all sharers on each update to K; a sharer cache discards K when it receives the callback (and the value is re-fetched from the server on the next cache miss for K). Classical leases [54] and Thor [3] use EI to maintain client cache consistency.

Figure 4.2 shows the stale window with EI caching. Key K is updated at time t_7 and C_1 discards K when it receives the server's callback at $t_{explicitInv}$. Thus, the stale window is $[t_7,\ t_{explicitInv})$. The length of this window is generally the one-way network latency between a server and the client; however, it may be longer if the callback encounters queuing delays. Therefore, any resource constraints (e.g., network bandwidth, CPU cycles) on servers or even the clients can impact the stale window.

Lease-based caching. Figure 4.2 shows the stale window with self-invalidating leases. Key K is updated at time t_7 and C_1 discards K when the lease expires at $t_{selfInv}$. Lease-based caching does not suffer from the drawbacks of naïve caching as the stale window is bounded by the lease end time and is independent of the length of transactions. Moreover, resource constraints do not affect the the stale window nor does the technique require servers to track sharers or send callbacks. Therefore, lease-based caching does not suffer from the drawbacks of EI caching. However, it is sensitive to clock skew. A leading clock causes the lease to expire sooner and a lagging clock extends the stale window and increases abort risk.

4.2.3 Ideal Lease Duration for a Key

The lease duration of a key impacts the stale and overall (fresh + stale) hit rate. Shorter leases incur fewer stale hits but may also reduce the overall cache hit rate. On the other hand, longer leases yield higher overall hit rates but increase the abort rate due to stale hits.

To select the *ideal* lease duration, we choose a term that maximizes the expected number of fresh hits. A more sophisticated solution might consider the weighted cost of transaction aborts due to stale hits in order to choose a suitable level of risk to balance the reward. We leave that evaluation to future work.

Arrival rate model for a key. To approximate the ideal lease duration, we need a model of the arrival rate of accesses (read and write) for key K. We use a Poisson process to model independent requests for K. Poisson is a standard stochastic process for independent arrivals, and is used in popular key-value storage benchmarks (e.g., YCSB [30], Retwis [73], TPC-C [78]) [26, 126, 105, 107]. In these benchmarks, each client processes transaction arrivals at a configured *transaction arrival rate* (λ). Transactions access keys sequentially according to configured logical relationships among keys, e.g., checking the status of an order in TPC-C. However, the decision to access a given key (or set) is independent for each transaction and has no correlation with the access. The inter-arrival times in a Poisson process are exponentially distributed and the mean inter-arrival time is λ^{-1} [118].

Thus λ for a given key K depends on its relative *popularity*, which is typically modeled as a power law distribution [30, 11]. The rate of reads and writes for K depends on its *read/write ratio*, which may vary across keys.

Calculating fresh hit rate. In practice, a server needs two parameters to evaluate a candidate lease duration for K. Figure 4.3 illustrates these parameters. First, the global write arrival rate (λ_{write}^{global}) of K is needed because a write from any client causes all cached copies to become stale. Second, the per cache mean read inter-arrival time (R_{mean}^{cache}) of K allows the server to compute the expected *per cache* fresh hit rate for K. The global read and write arrival rate for K is the sum of the read and write rates for K across all caches. Clients may observe these values directly and report them to the server.

A server approximates the ideal lease term for K by generating candidate terms

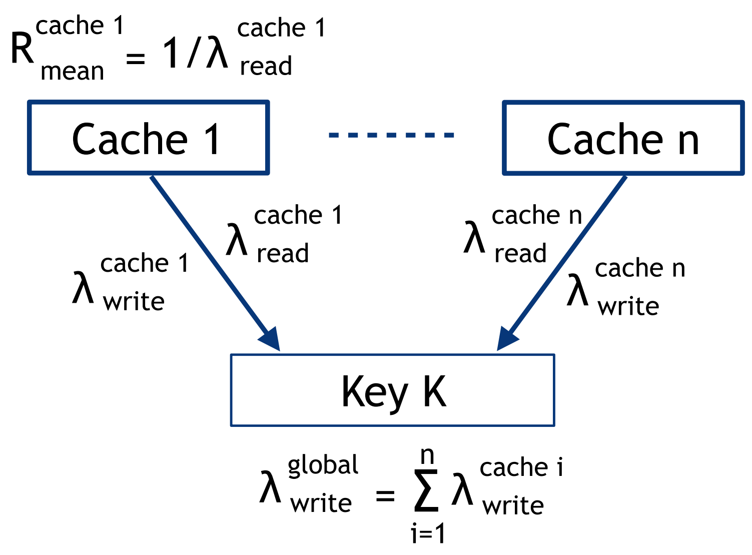

$R^{cache\ 1}_{mean} = 1/\lambda^{cache\ 1}_{read}$
Cache 1
Cache n
$\lambda^{cache\ 1}_{write}$
$\lambda^{cache\ 1}_{read}$
$\lambda^{cache\ n}_{read}$
$\lambda^{cache\ n}_{write}$
Key K
$\lambda^{global}_{write} = \sum_{i=1}^{n} \lambda^{cache\ i}_{write}$

FIGURE 4.3: Arrival rates and inter-access times for a key

and evaluating their expected effectiveness. Each lease duration d has an expected number of hits within d (given by Equation 4.1), and a probability of inter-update time being less/greater than d. If W is an exponentially distributed random variable that models the inter-write times of K, then the probability that *no* update arrives within d is given by Equation 4.2.

$$E[Hits(d)] = \frac{d}{R^{cache}_{mean}} \tag{4.1}$$

$$Pr(W > d) = e^{-\lambda^{global}_{write} \times d} \tag{4.2}$$

$$\begin{aligned} Pr(W \leqslant d) &= 1 - Pr(W > d) \\ &= 1 - e^{-\lambda^{global}_{write} \times d} \end{aligned} \tag{4.3}$$

Equation 4.3 gives the probability of an update arriving within d, i.e., an update arriving while a lease is still active. We call a lease period in which an update arrives a *stale lease period*. However, even within a stale lease period, any cache hits that occur before the update are still fresh. Figure 4.4 shows the example of a stale lease period and the fresh hit duration (d_{fresh}) within the stale lease period. Equation 4.4

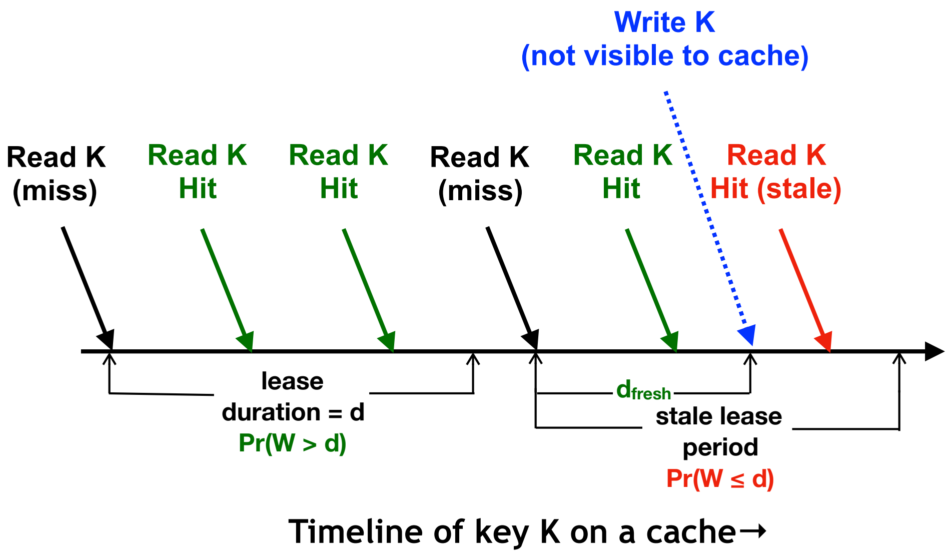


FIGURE 4.4: Fresh hit duration (d_{fresh}) in a stale lease period

gives the expected value of d_{fresh} in a stale lease period, where $\lambda = \lambda^{global}_{write}$. We use relative times to simplify the equation, i.e., we assume without loss of generality that the stale lease period starts at 0 and ends at d.

The fresh hit rate for a lease duration d is the weighted sum of the expected hit rate in a lease period with no update (all hits are fresh) and the expected fresh hit rate in a stale lease period. A server calculates the fresh hit rate using Equation 4.5. Each component is multiplied by the probability that any given lease period is a stale lease period (i.e., an update occurs during the lease term). The denominator is the expected number of cache hits in lease duration d plus the first read miss that fetches the data into the cache. Finally, equation 4.6 gives the stale hit rate for a lease duration d. Stale hits occur only in stale lease periods; within such periods, the expected number of stale hits is the number of reads in duration (d - d_{fresh}).

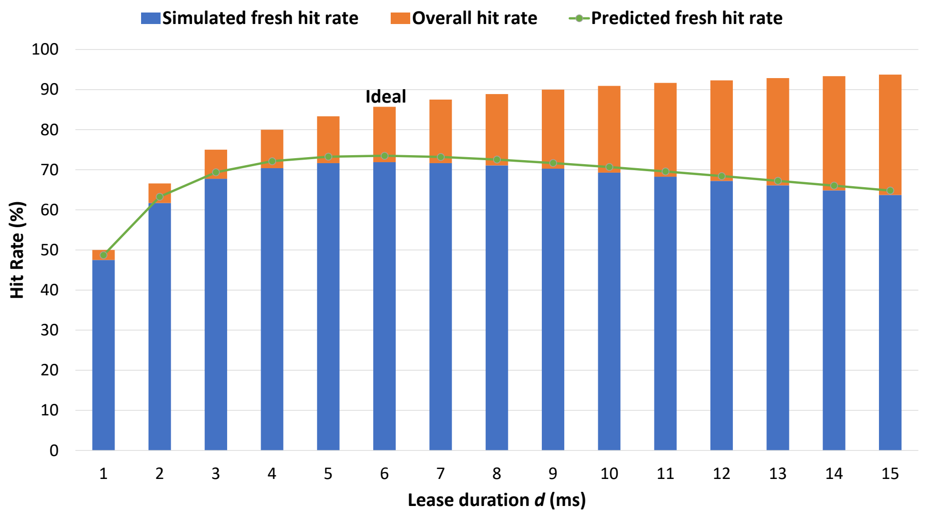


FIGURE 4.5: Ideal lease, $R^{cache}_{mean} = 1$ ms, $W^{global}_{mean} = 19$ ms.

$$\begin{aligned} E[d_{fresh} | 0 \leqslant d_{fresh} < d] &= \frac{\int_0^d \lambda x e^{-\lambda x} dx}{\int_0^d \lambda e^{-\lambda x} dx} \\ &= \frac{1 - (\lambda d + 1) e^{-\lambda d}}{\lambda (1 - e^{-\lambda d})} \end{aligned} \tag{4.4}$$

$$\begin{aligned} FreshHitRate(d) &= (Pr(W > d) \times HitRate(d)) \\ &\quad + (Pr(W \leqslant d) \times HitRate(d_{fresh})) \\ HitRate(d_x) &= \frac{E[Hits(d_x)]}{E[Hits(d)] + 1}, \; d_x = d \text{ or } d_{fresh} \end{aligned} \tag{4.5}$$

$$StaleRate(d) = Pr(W \leqslant d) \frac{E[Hits(d - d_{fresh})]}{E[Hits(d)] + 1} \tag{4.6}$$

Finding ideal lease duration. To illustrate, Figure 4.5 shows the fresh hit rate for varying lease durations for a key with $R^{cache}_{mean} = 1$ ms and $W^{global}_{mean} = 19$ ms. It also shows prediction accuracy by comparing predictions using Equation 4.5 with results from a Monte Carlo simulation. The simulator takes R^{cache}_{mean}, W^{global}_{mean} and a

Algorithm 3 Find ideal lease duration

```
 1: procedure FIND_IDEAL_LEASE_DURATION($R^{cache}_{mean}$, $\lambda^{global}_{write}$)
 2:     bestLeaseDuration = 0
 3:     bestFreshHitRate = 0
 4:     expectedHitsPerLease = 1
 5:     while true do
 6:         leaseDuration = expectedHitsPerLease × $R^{cache}_{mean}$
 7:         freshHitRate = get_fresh_hit_rate(leaseDuration, $R^{cache}_{mean}$, $\lambda^{global}_{write}$)
 8:         if freshHitRate < bestFreshHitRate then
 9:             break                                         ▷ Stop searching
10:         end if
11:         bestFreshHitRate = freshHitRate
12:         bestLeaseDuration = leaseDuration
13:         expectedHitsPerLease += 1                 ▷ Increase lease duration
14:     end while
15:     return {bestLeaseDuration, bestFreshHitRate}
16: end procedure
```

candidate lease duration as the input, and generates read and write arrivals from an exponential distribution with the specified inter-arrival times. Data is cached for the lease duration after each miss; the read arrival times are used to determine if a read is a cache hit, and write arrival times are used to determine whether a hit is fresh. We simulate 10M accesses for each lease duration. As seen from the figure, the predicted values are always optimistic, and the average difference with the simulation results is 1.4%.

The trend from figure 4.5 shows that the overall hit rate (fresh + stale hits) increases with the lease duration (d); the fresh hit rate increases initially, hits a peak, which is the ideal value for d, and any subsequent increase in d only increases the number of stale hits and therefore the fresh hit rate starts to drop. We use this trend to design a simple gradient algorithm to find the ideal lease duration for a key K (Algorithm 3). The algorithm takes the per-cache mean inter-read time and the global write rate of K and returns the ideal lease duration and highest fresh hit rate. Other possible considerations (e.g., minimum fresh hit rate, maximum stale rate) are left to future work.

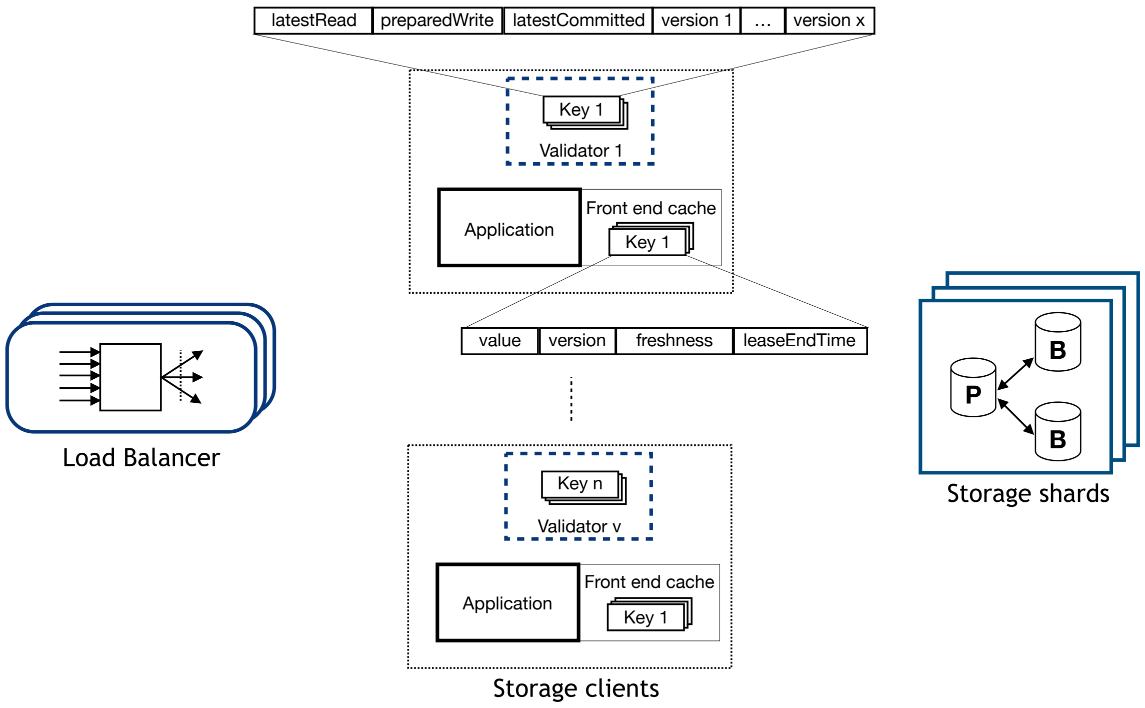


FIGURE 4.6: KAIROS architecture

4.3 KAIROS: A Transactional Key-Value Storage System

This section presents KAIROS, a transactional key-value storage system that leverages inter-transaction caching and sharded validation to improve performance and alleviate workload-induced hotspots.

4.3.1 System Architecture

Figure 4.6 shows the architecture of KAIROS. The KAIROS design targets an intra-data center client/server storage model. Data is stored on DRAM, NVM or SSDs on the storage servers. The key space is sharded across the storage servers, and each shard is replicated for availability and fault tolerance. Applications run on one or more multi-tenant client servers. A frontend load balancer (e.g., Slicer [4]) distributes user transactions across application instances on different client servers, while balancing load or exploiting any locality of accesses.

Kairos allows lease-based inter-transaction caching on the client. Storage servers hand out leases for caching frequently-read keys and a cached key is self-invalidated by client caches on lease expiration. Lease durations provide probabilistic guarantees of staleness, a key may be updated while a lease is still active and cause stale hits on caches, but system safety is not compromised because the transaction protocol aborts all transactions that read stale data. Kairos servers predict the ideal lease durations based on the inter-access (read and write) times of keys (§ 4.2.3).

Kairos uses optimistic concurrency control (OCC [68]) adapted to a client/server setting [3] to support serializable ACID transactions. OCC enhances concurrency by allowing transactions to access data without acquiring any locks and validation checks are performed before a transaction commits to detect any conflicts. A transaction validation request consists of a commit timestamp, and a read and write set of keys accessed by the transaction. The read set consists of version timestamps of all the keys read in the transaction, and the write set contains the keys the transaction intends to write to. Validation of transactions in OCC is typically performed on the storage servers [3, 130, 92]. However, Kairos offloads the validation workload to client-side validators, and the storage servers are involved in a backup validation path when the validators do not have the requisite state to validate a transaction.

Figure 4.6 shows the state maintained on a client cache and validator. Each client cache in Kairos operates independently and popular data may be replicated across caches. Each key in a client cache is associated with a value, version timestamp ($ts_{version}$), lease end time ($ts_{leaseEndTime}$) and freshness timestamp ($ts_{freshness}$). $ts_{version}$ is the commit timestamp of the transaction that wrote the version, $ts_{leaseEndTime}$ indicates the time at which the entry will be self-invalidated from the cache and $ts_{freshness}$ indicates the latest timestamp for which the client knows that the cached version is fresh i.e., there are no superseding writes with timestamp t in the interval $(ts_{version}, ts_{freshness}]$ (Section 4.3.2).

Algorithm 4 Processing phase of a transaction T. For brevity, we use $ts_{version}$ = vts, $ts_{leaseEndTime}$ = lts, $ts_{freshness}$ = fts

```
1:  procedure READ(T, key)
2:      if key ∈ T.writeSet then
3:          return T.writeSet[key].value
4:      else if key ∈ T.readSet then
5:          return T.readSet[key].value
6:      else
7:          if key ∉ Cache or Cache[key].lts < $t_{current}$ then
8:              Cache[key].{value, vts, lts} = get_from_server(key)
9:              Cache[key].fts = max(Cache[key].vts, $ts^{global}_{watermark}$)
10:         end if
11:         T.readSet[key].{value, vts, fts} = Cache[key].{value, vts, fts}
12:         return T.readSet[key].value
13:     end if
14: end procedure

15: procedure WRITE(T, key, value)
16:     T.writeSet[key].value = value
17: end procedure
```

Key ownerships are distributed across client-side validators. A validator maintains a latest read timestamp ($ts_{latestRead}$), a *preparedWrite* flag, a latest committed timestamp ($ts_{latestCommitted}$) and a list of version timestamps for each key that it owns and is responsible for validating. $ts_{latestRead}$ indicates the highest commit timestamp of a transaction that read the key, *preparedWrite* indicates if there is a successfully validated but uncommitted transaction and $ts_{latestCommitted}$ is the commit timestamp of the last transaction that wrote the key.

In addition, each validator maintains a garbage collection threshold timestamp ts_{GC} and discards all key versions older than ts_{GC}. ts_{GC} indicates that the validator has sufficient state to validate all transactions with freshness timestamps $\geqslant ts_{GC}$ (Section 4.3.4).

4.3.2 Transaction Protocol

KAIROS executes transactions in a similar manner to other client-server storage systems with OCC [3, 130, 39]. Algorithm 4 shows how a transaction T's reads and write requests are handled during the processing phase of T. For a read request, a value is returned if the key exists in T's write or read set. Otherwise, the transaction checks the local cache for the key. On a cache miss, the read request is sent to the remote server and the returned data is placed in the local cache; the server returns the value, $ts_{version}$ and $ts_{leaseEndTime}$. The $ts_{freshness}$ value of a cached key is set to the max of the timestamp of the returned version and the client's view of the global watermark, i.e., $ts_{freshness} = max(ts_{version}, ts^{global}_{watermark})$; KAIROS guarantees that no writes older than $ts^{global}_{watermark}$ will occur in the system (§4.3.4 describes watermarks in greater detail). For each key read in the transaction, the read set tracks its value, $ts_{version}$ and $ts_{freshness}$. All transaction writes during the processing phase are buffered and made visible only after a transaction commits, as is typical in OCC.

The processing phase finishes when the application invokes commit transaction. Before initiating the commit process, the client assigns a freshness and commit timestamp to the transaction T. The freshness timestamp of a transaction ($T_{freshness}$) is the minimum freshness timestamp from all keys in the read set, i.e., $T_{freshness} = \min_{key \in readSet} key.fts$. The commit timestamp ($T_{commit}$) depends on the type of transaction (read-only or read-write); for a read-only transaction, the commit timestamp is max freshness timestamp from all keys in the read set i.e., $T_{commit} = \max_{key \in readSet} key.fts$ and for a read-write transaction, $T_{commit} = t_{current}$, where $t_{current}$ is the current time on the client. After assigning the timestamps, the client initiates and acts as the coordinator in the the two-phase commit (2PC) protocol; the 2PC participants include the storage servers and validators for all keys in either set. The validation decisions of transactions are logged on the client.

Algorithm 5 Validation Algorithm

```
 1: procedure VALIDATE(txn)
 2:     if txn.freshness < $ts_{GC}$ then
 3:         return ABORT                          ▷ Not enough state to validate
 4:     end if
 5:     for each (key, version) ∈ txn.readSet do
 6:         if key.preparedWrite then
 7:             return ABORT
 8:         else if key.latestCommitted ≠ version then
 9:             return ABORT
10:         end if
11:     end for
12:     newVersion = txn.commitTimestamp
13:     for each (key, version) ∈ txn.writeSet do
14:         if key.preparedWrite then
15:             return ABORT
16:         else if key.latestRead ⩾ newVersion then
17:             return ABORT
18:         else if key.latestCommitted ⩾ newVersion then
19:             return ABORT
20:         end if
21:     end for
22:     return COMMIT
23: end procedure
```

4.3.3 Transaction Validation

Algorithm 5 shows the validation algorithm used by the sharded client-side validators in KAIROS. A validator simply aborts a transaction T if $T_{freshness} < ts_{GC}$ because it does not have state to validate T as it discards versions behind ts_{GC}. $T_{freshness}$ provides a time bound on the oldest key read by T from its client cache; T can be validated only if $T_{freshness} \geqslant ts_{GC}$ because if T read any stale data from its cache, i.e., it missed a superseding write w, then w's timestamp must be later than $T_{freshness}$, and if $T_{freshness} \geqslant ts_{GC}$ then the validator must remember w, and the validator aborts T.

Validation of read-only transactions is same as in other OCC-based systems [3, 130, 39, 75], below we describe our protocol for validating read-write transactions.

Read-write transaction. Figure 4.7 shows an example of the distributed two phase commit (2PC) protocol used to commit read-write transactions, with the client acting as the coordinator. The key difference is that in phase 1 of 2PC, a client

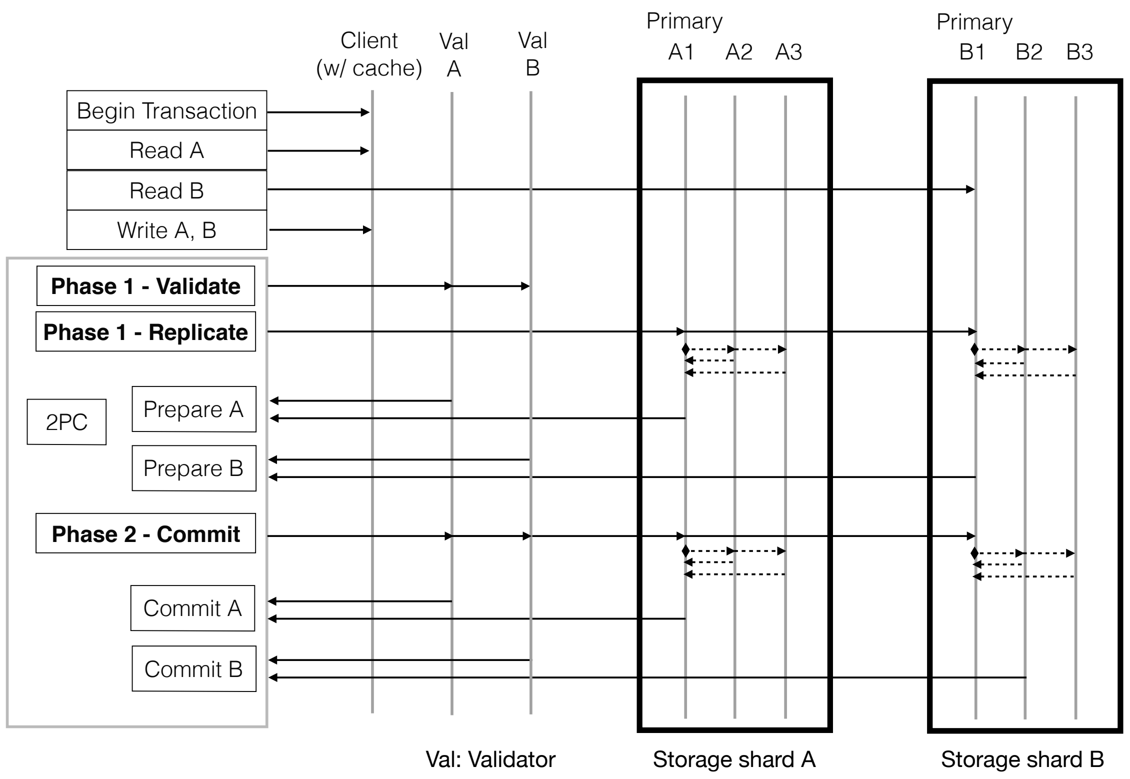


FIGURE 4.7: Two Phase Commit (2PC)

(coordinator) sends, in parallel, a validate request to the participant validators and a replicate request to primaries of participant storage shards. Replicate requests are quorum replicated before a primary of a storage shard returns a response. A client accumulates all validate and replicate responses before starting phase 2 of 2PC. A transaction decision is COMMIT only if *all* validators respond with a COMMIT during phase 1, otherwise the decision is to ABORT. In phase 2, a client informs the transaction decision to the application and all participant validators and primaries of storage shards.

Although not shown in the algorithm, after successful validation of a read-only or read-write transaction, a validator sets $key.latestRead = txn.commitTimestamp$ for all validated keys in the read set and $key.preparedWrite = true$ for all keys in the write set of the transaction. A validator resets $key.preparedWrite$ on receiving the transaction decision during 2nd phase of 2PC and also updates *key.latestCommitted* for a COMMIT decision.

4.3.4 Watermarks and Version Management

Kairos uses watermarks [39] for version management on storage servers and validators. Each client c in Kairos maintains a watermark timestamp $ts^{c}_{watermark}$. The meaning of $ts^{c}_{watermark}$ is that every transaction associated with c with commit timestamp $t \leqslant ts^{c}_{watermark}$ has already completed. A client also caches the watermarks of other clients in the system; the cached information must be refreshed periodically, for example using gossip protocol [38]. Each client computes a global watermark using the individual watermarks of clients in the system. Let C be the set of all clients in the system, then the global watermark is $ts^{global}_{watermark} = \min_{c \in C} ts^{c}_{watermark}$. By construction, any transaction with commit timestamp $t \leqslant ts^{global}_{watermark}$ has already completed.

Clients use $ts^{global}_{watermark}$ to assign a freshness timestamp to newly cached keys (see Algorithm 4). Let $c_{freshness}$ be the minimum freshness timestamp from all cached keys in a client. The meaning of $c_{freshness}$ is that the oldest cached value in a client is known to be fresh as of $c_{freshness}$. In other words, the client may have missed writes to cached keys with timestamp $> c_{freshness}$, but knows about all writes with timestamps $< c_{freshness}$.

Each client periodically broadcasts $c_{freshness}$ to all storage servers and validators in the system. In turn, the validators and storage servers use the individual $c_{freshness}$ value to calculate a garbage collection timestamp ts_{GC}, where $ts_{GC} = \min_{c \in C} c_{freshness}$. Versions with timestamps $\leqslant ts_{GC}$ can be safely discarded by the storage servers and validators since there can be no transactions with freshness timestamps $< ts_{GC}$.

4.3.5 Recovery

Prior work [3, 92] describes recovering from client or storage shard failures. Here, we describe how to recover from validator failures.

Validator Failure. If a participant validator fails during 2PC then the corresponding storage shards take the responsibility of validating transactions for keys owned by the validator. The primaries of storage shards can validate all outstanding read-only transactions but might not have enough state to validate read-write transactions.

Specifically, the primary may not have the correct $ts_{preparedRead}$ for a key since there might be a read-only transaction that updated this field on the failed validator but the storage shard did not receive any request for this transaction. Validating any read-write transaction without these values can violate serializability. Consider the following scenario: a read-only transaction T_a read key K and was successfully committed with a timestamp t_2. Now the validator for key K fails, and the primary allows a read-write transaction T_b to commit that creates a new version of K with timestamp t_1 ($t_0 < t_1 \leqslant t_2$). This violates serializability because T_a (already committed) should have read T_b's write.

Kairos uses leases to avoid this scenario. A validator in Kairos obtains a periodically renewed lease from storage shards (from at least *f+1* replicas within each shard) to validate transactions with an commit timestamp $< t_{lease}$. The primary waits for its local clock to advance past t_{lease} before servicing read-write transaction requests for its shard. All outstanding read-write transactions with commit timestamps $< t_{lease}$ can be either aborted or proposed to commit at a timestamp $> t_{lease}$. In all failure scenarios (client, primary, backup replica or some combination of the three) a decision can be made on any outstanding transaction and service can be resumed as long as a majority of replicas ($f + 1$) of all shards are available.

4.3.6 Comparison with Centiman

There are some similarities between Kairos and Centiman [39]. Kairos follows Centiman in decoupling validation from storage servers, so that validation scales independently of the storage tier. Like Centiman, Kairos also maintains multiple versions of (key, version) pairs on the sharded validators for providing transactional serializability. Finally, both Kairos and Centiman use watermarks for version management; watermarks are used for calculating a garbage collection timestamp (ts_{GC}) and versions with timestamp $< ts_{GC}$ are discarded.

However, Centiman keeps $ts_{GC} \approx ts^{global}_{watermark}$ (global watermark) to minimize state on validators as transactions always read fresh values from storage servers during execution as there is no inter-transaction caching. In contrast, Kairos aims at keeping $ts_{GC} << ts^{global}_{watermark}$ to maximize state on validators, which in turn enables inter-transaction caching on the clients. In Kairos, a validator cannot validate a transaction T if $T.freshness < ts_{GC}$ (see §4.3.3) because it does not know of all writes that are pertinent to T, that T might have missed due to stale cache reads, since the validator discards versions behind ts_{GC}. Therefore, keeping $ts_{GC} << ts^{global}_{watermark}$ enables Kairos to validate transactions that read "old" data out of the cache. Crucially, it also enables Kairos to use validation as a "fallback" for cache consistency: it is safe for T to read stale data without violating consistency because a validator will abort T.

4.4 Evaluation

This section presents results from our prototype implementation of Kairos running in an Azure cluster.

Implementation. We implemented caching on top of Milana, with local validation disabled (Section 6.2 describes a technique for combining inter-transaction caching with local validation of read-only transactions; is left for future directions to explore). To elucidate the advantages of lease-based caching, we also implemented the two other caching techniques discussed previously: naïve and explicit invalidation-based caching. Based on the cache configuration, servers in our system either hand out leases or track sharers and send invalidations after a read-write transaction commits; in naïve caching, servers hand out lease durations longer than the running time of our experiments, so cache management only occurs based on transaction decisions.

To compare the impact of different caching techniques in isolation, we fix the set of keys that can be cached in all cache configurations. This enables us to evaluate performance across all configurations while caching the same set of keys, with same access patterns; the keys are pre-computed using a top-k algorithm [24, 90]. For lease-based caching, servers track the mean inter-write time and client caches track the per-cache mean inter-read time of the top-k keys. A client cache periodically piggybacks the mean inter-read time for a key with a get request and the servers use this value along with its view of the global mean inter-write time to calculate ideal lease duration (see Algorithm 3). Our prototype assumes uniform access distribution across all client caches.

Finally, we implement a validator for sharded validation and make the necessary changes on servers for the modified 2PC protocol with sharded validation.

Experimental setup. We run all experiments on Microsoft Azure D4s.v3 nodes with 4 vCPUs, 16 GB of RAM and a high performance network. All experiments use 5 storage shards and up to 15 clients. Each shard has 1 primary and 2 backups; data is stored on DRAM. We co-locate the application and validators on a subset of the clients. The system clocks on all VMs are synchronized using PTP software timestamping mode. The average clock skew between VMs and one-way network latency is 400 μs and 500 μs, respectively.

Workload. We use a variant of YCSB [30] to model a social network application where the data of popular users is read more often and users have different rate of posting updates. The workload models this behavior by using different zipfian distribution coefficients for controlling popularity of keys in read-only (α_r) and read-write (α_{rw}) transactions. By default, $\alpha_r = 0.99$, $\alpha_{rw} = 0.75$ and 90% of transactions are read-only. We evaluate the impact of varying α_r and the rate of read-only transactions. Each transaction accesses 4 keys. We populate the system with 20M keys; each key is 16B in size and a value is 1KB.

4.4.1 Inter-Transaction Caching

This section evaluates impact of inter-transaction caching alone. We use the YCSB workload with default configuration for all experiments in this section. By default, we use 6 clients in each experiment; each client executes 4M transactions at a default rate of 5k transactions/sec. Transaction validation occurs on the storage servers for all experiments presented in this subsection. We present results with sharded validation in Section 4.4.2.

First, we evaluate the sensitivity to cache configurations.

Impact of cache size. Figure 4.8 shows the hit rate for varying cache size (% of total dataset size). Clients use a naïve caching approach in which cache consistency decisions are made based on transaction validation decisions; we observe similar trends with other caching techniques. As seen from the figure, initially the hit rate increases rapidly with the cache size, and plateaus after cache size = 0.1% of the total dataset size. Subsequently, the hit rate only increases by 1.4% even after doubling the cache size from 0.1% to 0.2%. Based on these results, we use a cache size of 0.1% of the total dataset size in all subsequent experiments.

Impact of using ideal lease duration. This experiment compares our ideal lease calculation technique with a static approach where leases are based on a fixed probability of an update arriving while a lease is still active. Figure 4.9 shows

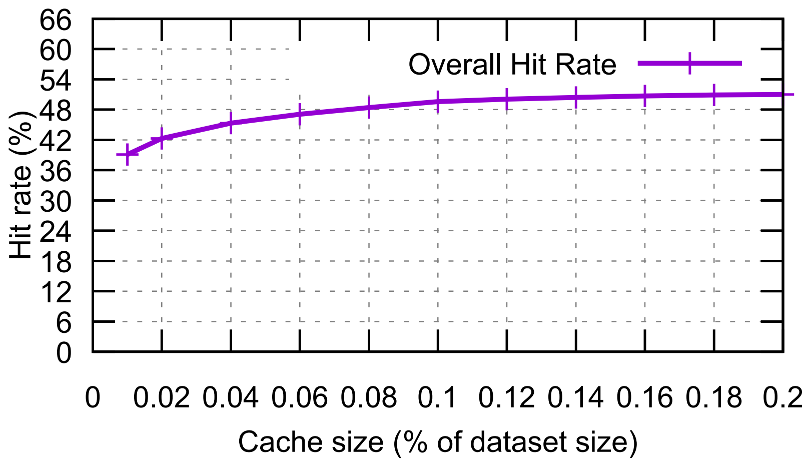


FIGURE 4.8: Client cache size vs hit rate

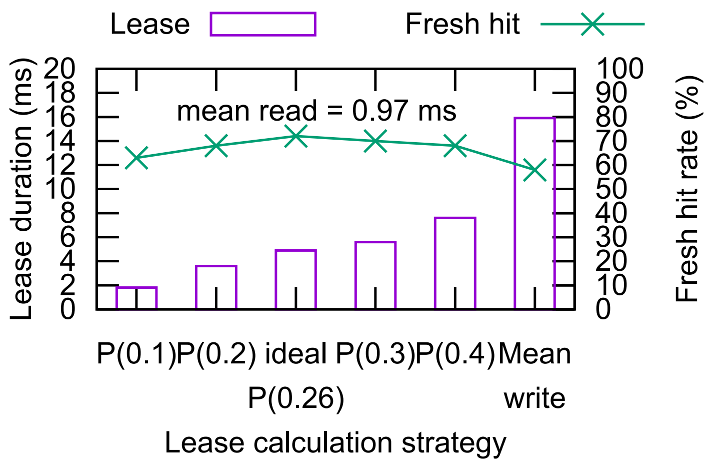


FIGURE 4.9: Fresh hit rate with varying lease durations for a key

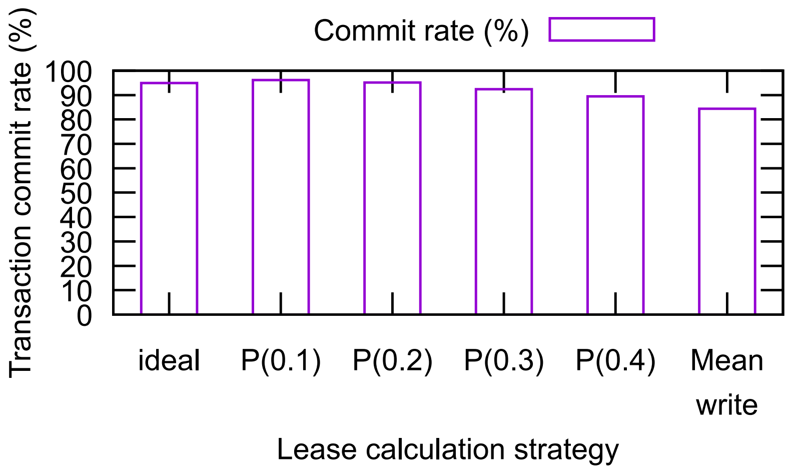


FIGURE 4.10: Commit rate: ideal vs static strategy for lease calculation

how lease duration and the fresh hit rate for a duration varies for different lease calculation strategies. In the graph, x in $P(x)$ corresponds to the probability of an update arriving within a lease duration d, i.e., $\Pr(W \leqslant d) = x$. The value of d is proportional to x, higher values of x lead to higher values of d. Note that x in P(x) is not the stale rate. In this experiment, for each key, we find the highest possible value of d that still satisfies the constraint set by the choice of x. The figure shows that ideal gives the highest fresh hit rate, with P(0.26).

Figure 4.10 shows the transaction commit rates with the various lease calculation strategies. Lease durations across all cached keys vary between 4 ms and 5 secs. As seen from the figure, the overall transaction commit rates drop with increasing values of x in P(x); ideal is able to achieve a commit rate within 1.4% of the commit rate of $P(0.1)$, while delivering a 6% higher fresh hit rate compared to P(0.1) (not shown in figure). These results show the trade-off between fresh hit rate and commit rate and necessitate finding a duration that maximizes fresh hit rate without sacrificing much on commit rate. Our ideal technique is able to achieve this goal through an analytical model (see §4.2.3), without a need for experimentally finding the best lease that give highest fresh hit rate.

Next, we compare our lease-based technique with naïve and explicit invalidation-based (EI) caching. Inter-transaction caching in Sundial [129] is similar to naïve caching and Thor [3] uses EI for cache consistency. *Transaction validation occurs on the storage servers with all caching techniques.*

Impact of offered load on performance. This experiment evaluates the impact of increasing offered load on performance with the different caching techniques. Figure 4.11 shows the commit rates with each technique for increasing offered load. There are two take aways from the figure. First, naïve caching performs worst as cache management is done only based on transaction decisions with this technique. Second, EI caching performs better than our lease-based technique at lower loads

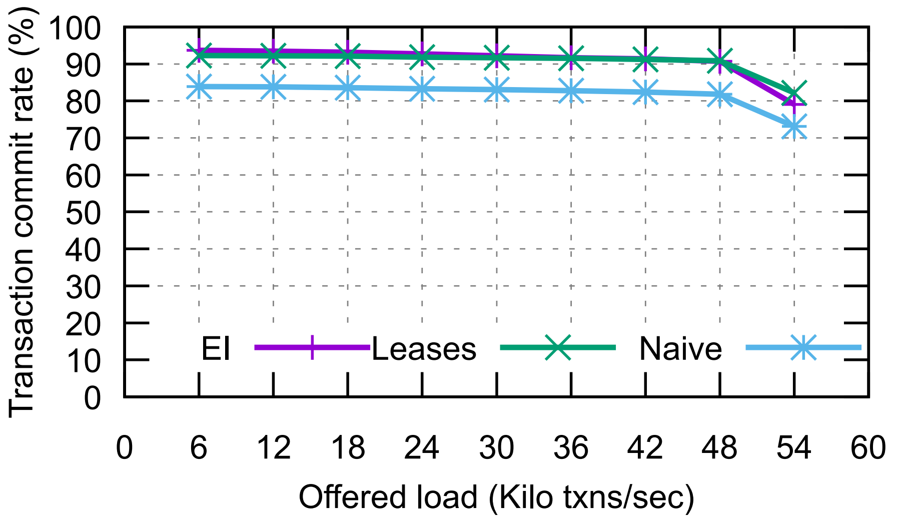


FIGURE 4.11: Commit rate with different caching technique

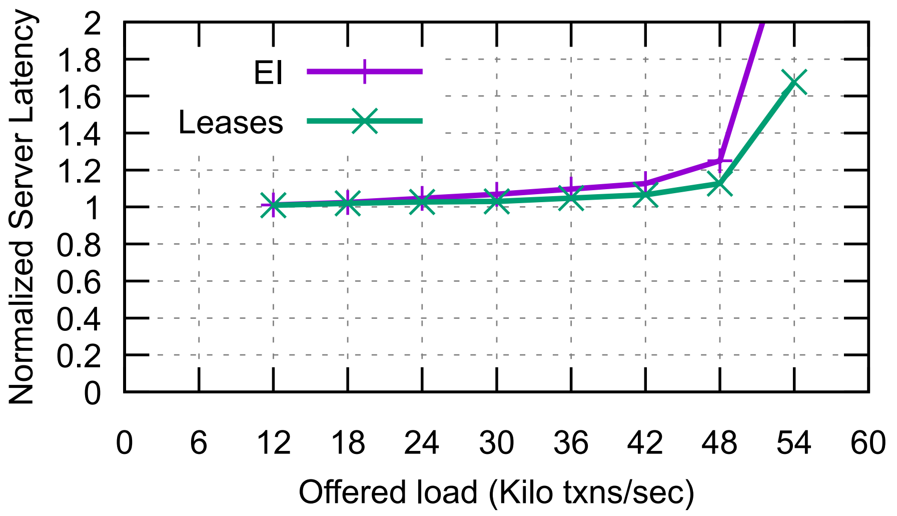


FIGURE 4.12: Normalized average server latency

but the commit rates with EI caching drop steadily as load is increased and there is a cross-over point at offered load of 42k transactions/sec after which lease-based caching technique offers better commit rates. The system (CPU on most loaded server) saturates at offered load of 54k requests/sec, irrespective of the caching technique.

Increasing load has a greater impact on EI caching because its stale window is impacted by queuing delays (Section 4.2.2). Figure 4.12 show the request processing latencies on the most loaded server with EI and lease-based caching. The values in the figure are normalized to the the latency for 6k transactions/sec with EI caching.

Impact on scalability. Here we evaluate the scalability impact of EI and lease-based caching, while keeping the offered load fixed at 48k transactions/sec (below the saturation point as seen from Figure 4.11).

Figure 4.13 shows the throughput with each technique for increasing number of clients (and caches). The system with EI caching is only able to support 8 clients before performance (throughput and latency) starts to degrade dramatically. Figure 4.14 (notice log scale on y-axis) shows the normalized transaction latency with each technique; the latencies are normalized to the transaction latency with EI caching and 6 clients. EI caching does not scale well because a server's work for maintaining cache consistency increases linearly with the number of client caches and this minimizes the gains from caching. Figure 4.15 shows the percentage of messages saved with each technique. The messages saved metric is calculated by subtracting the number of hits with number of invalidation requests needed to keep caches consistent (0 with lease-based and > 0 with EI caching) and dividing the result by the total number of get (read) requests. This metric captures the percentage of requests that were serviced locally, without any involvement or resource utilization (e.g., CPU, network etc) on the servers. The metric is of interest as the CPU on the server saturates in our setup during these experiments. As seen from the figure, messages saved (%) with EI caching drops almost linearly with increasing number of sharers, indicating that the gains from caching are minimized by the invalidations sent by the servers.

In contrast, lease-based caching scales better and is able to support 13 clients before saturation, while offering 55% higher throughput compared to EI caching. The messages saved (%) with lease-based caching also decrease with increasing number

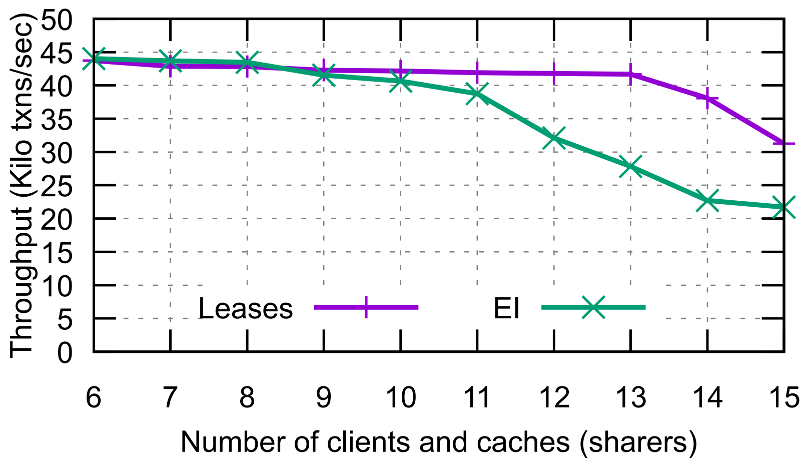


FIGURE 4.13: Throughput: explicit invalidation vs lease-based caching

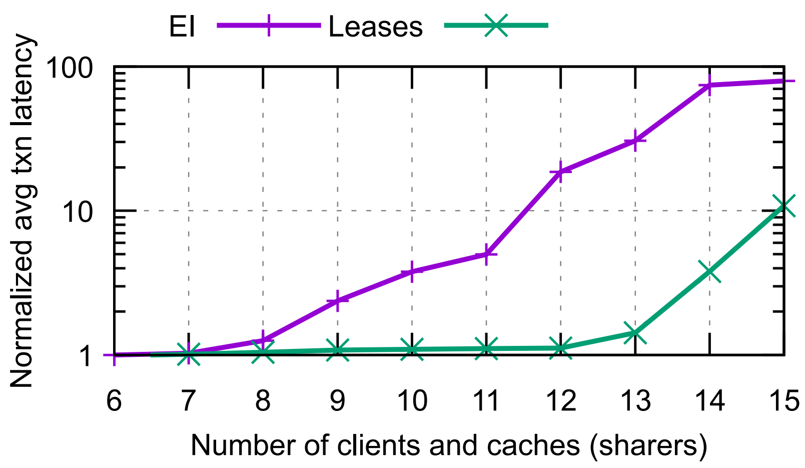


FIGURE 4.14: Normalized transaction latency: explicit invalidation vs lease-based caching

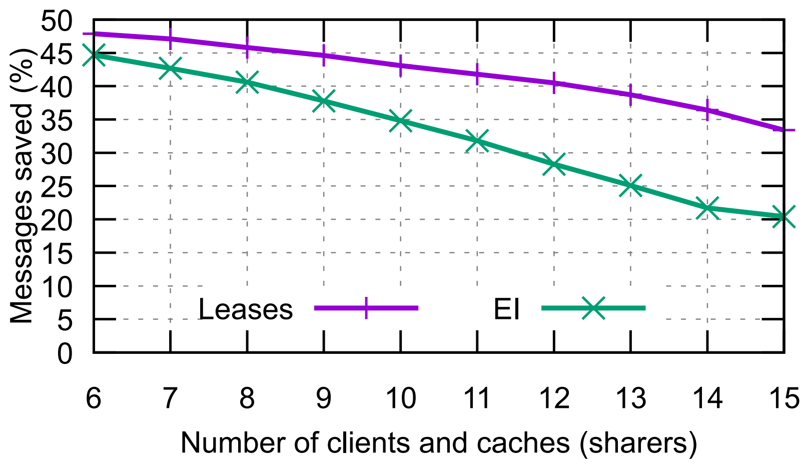


FIGURE 4.15: Messages saved: explicit invalidation vs lease-based caching

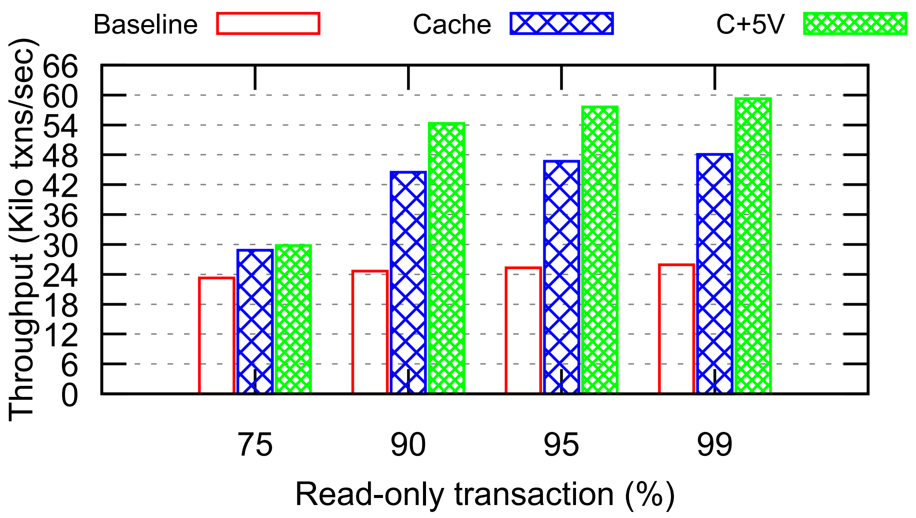


FIGURE 4.16: Throughput with varying read transaction %

of sharers because the mean inter-read time per cache increases and this reduces cache effectiveness. *This trend is independent of the caching technique and shows the importance of locality-aware request distribution for effective cache utilization [102, 4].*

We explored techniques to offload EI caching from the storage servers to the client (e.g., sharded validator) so as to compare EI and lease-based caching in a setup where servers are not involved in maintaining cache consistency. However, any design almost doubles (worst case) the number of messages in a transaction since all reads (cache misses) during execution would also have to be sent to the client-side cache manager for tracking sharers and writes need to be sent for triggering invalidations (if cache manager is separate from validator).

4.4.2 Comparison with a Baseline System

Here we evaluate KAIROS against a baseline system with *intra*-transaction caching only and no de-centralized validation. We use 6 clients and 5 storage shards in all experiments and vary the offered load per client. Each client caches 0.1% of the most

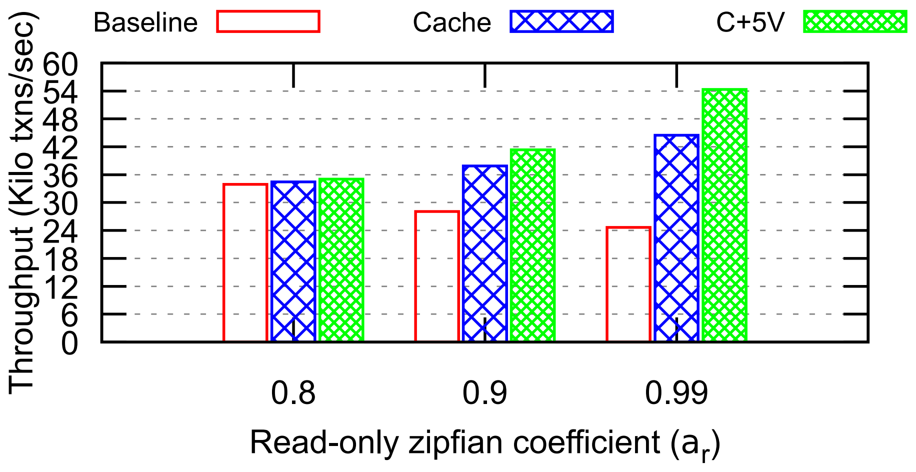


FIGURE 4.17: Throughput with varying α_r (90% read-only txn)

popular keys in the workload. All transactions are validated on client-side validators. We use 5 validators (same as number of storage shards) in a system with sharded validation.

Figure 4.16 shows the throughput with varying read-only transaction percentage for a baseline system, a system with inter-transaction caching only and a system with inter-transaction caching and sharded validation. As seen from the figure, the throughput of all systems increases for increasing read-only transaction percentage. A system with inter-transaction caching alone offers up to 1.86x throughput of the baseline system. Adding sharded validation (with caching) provides an additional 22.5% improvement in throughput and offers 2.28x throughput of the baseline system. Sharded validation improves performance by moving validation away from servers.

Figure 4.17 shows the throughput for varying zipfian coefficient (α_r) for selecting keys in read-only transactions. We fix the read-only transaction percentage to 90% for this experiment. Popularity distribution of keys is directly related to α_r. As α_r increases, the performance of the baseline system drops due to workload-induced hotspots as reads become more skewed. In contrast, the performance impact of inter-transaction caching improves because the hit rate on individual client caches increases when distribution is more skewed. Impact of sharded validation shows a similar trend to inter-transaction caching.

4.5 Related Work

Caching in distributed storage systems. Client caching is standard in distributed file systems [86, 103, 2] using variants of callback leases [54]. Prior works have also explored caching to improve performance and/or balance load in key-value stores [45, 97, 80, 85, 61, 52]. However, none of these works support transactions. Fan et al [45] prove that $O(n\log n)$ is a lower bound on the cache size to provide good load balance, where n is the total number of backend nodes. They use this result to design a small look-through cache that resides in a frontend load balancer and serves the most popular items to dynamically balance the load across the backend nodes. SwitchKV [80] improves performance of flash-based key-value stores by using an OpenFlow-capable switch to steer requests for popular data to an in-memory caching layer. However, SwitchKV maintains a single copy of each cached object and therefore is prone to hotspots in the caching layer.

IncBricks [85] and NetCache [61] use middleboxes and switches, respectively, to cache popular keys inside the network. Nishtala et al [97] propose techniques for scaling memcached. ccKVS [52] leverages skew for aggressively caching and replicating popular data to improve performance of key-value stores. All these systems allow keeping multiple copies of frequently-accessed data but require explicit invalidations for cache consistency.

Client-server transactions with OCC. Numerous works use optimistic concurrency control (OCC [68]) for ACID transactions [3, 41, 39, 130, 42**?** , 25, 92]. Kairos follows Thor [3] in using physical clocks for the OCC version stamps, as do many others (e.g., [130, 39, 92]). Among these, Milana [92] and Centiman [39] are most closely related. Milana uses precise clocks for optimizing replication and transaction protocols and also shows that they reduce abort rates in OCC-based systems. However, Milana does not support inter-transaction caching. Comparison with Centiman is presented in Section 4.3.6. KAIROS adopts Centiman's scalable sharded validation, and integrates support for inter-transaction caching: our results show that the combination yields more benefit than either technique alone for workloads with hot keys. Spanner [31] also uses physical clocks for transactions, but only for snapshot reads: Spanner does not use OCC.

Cache consistency. Thor [3] and some of its successors support inter-transaction caching using explicit invalidations (callbacks) to keep caches consistent. Thor shows that *asynchronous* callbacks are sufficient for transaction systems with OCC. Although asynchrony causes a transaction T to read stale data, consistency is not violated since T fails OCC validation and aborts. In essence, Thor uses OCC as a fallback for loose cache consistency. KAIROS takes this idea one step further by eliminating the callback entirely (or making it optional). Thor also shows that OCC with physical clocks leads a rate of spurious aborts that increases with clock skew. KAIROS leverages precise clocks to minimize these aborts (like Milana [92]) and also to support time-based consistency with a lightweight protocol that directs clients to self-invalidate cached keys at precise times ("soft" leases).

Sundial [129] uses leases based on logical time and also integrates inter-transaction caching with OCC for serializable transactions. Sundial is similar to naïve caching (see §4.2.2), but it reorders transactions to avoid some stale hits, and disables caching for a key if its rate of stale hits exceeds an arbitrary configured threshold. KAIROS uses precise clocks to adjust lease terms dynamically for effective caching on a per-key basis. Furthermore, KAIROS can also support external consistency since transactions commit in physical timestamp order. Finally, KAIROS scales validation independent of storage.

Self invalidation in coherent caches/shared memory systems. Prior works have used self-invalidation to improve performance of coherence protocols in shared memory multiprocessors [74, 94, 69, 108]. Ninan et al [96] use cooperative leases to maintain consistency in content distribution networks. Lease management is done hierarchically to avoid overloading lease brokers and the duration of a lease is based on the the degree of freshness desired for cached objects. Mirage [47] uses static leases to reduce coherence overhead in software distributed shared memory systems.

4.6 Summary

Distributed, transactional storage systems scale by sharding data across servers. However, workload-induced hotspots result in contention, leading to higher abort rates and performance degradation.

This chapter presents KAIROS, a transactional key-value storage system that leverages client-side inter-transaction caching and sharded transaction validation to balance the dynamic load and alleviate workload-induced hotspots in the system. KAIROS utilizes precise synchronized clocks to implement self-invalidating leases for cache consistency and avoids the overhead and potential hotspots due to maintaining sharing lists or sending invalidations.

Experiments show that inter-transaction caching alone provides 1.86x the throughput of a baseline system with only intra-transaction caching; adding sharded validation further improves the throughput by a factor of 2.28 over baseline. We also show that lease-based caching can operate at a 62.5% higher scale while providing 1.55x the throughput of the state-of-the-art explicit invalidation-based caching.

5

SkimpyFTL

Transactional key-value storage systems use a multi-version storage to increase concurrency [32] and reduce abort rates [92, 40] since reads can be satisfied from a consistent snapshot in the past (old versions), while writes create new versions. Figure 5.1 illustrates the impact of single vs multi-versioning on transaction abort rate for a social network application; workload and methodology are described in §5.3. As seen from the figure, multi-versioning provides 2× reduction in abort rates compared to single-version storage, and its benefit increases with offered load.

However, there are several challenges in designing a multi-version storage, including: 1) additional capacity, 2) index for mapping versions to values, and 3) version management. First, the extra versions require additional capacity, which can be prohibitively expensive with in-memory (DRAM) storage systems. Second, these systems need an index to map versions of a key to their value so reads can be serviced from a consistent snapshot. A naïve approach is to store the entire index in DRAM; this approach provides the lowest lookup latencies but has a high space overhead. An efficient indexing technique needs to find a tradeoff between lookup latencies and DRAM requirement for indexing. Third, multi-versioning necessitates a version

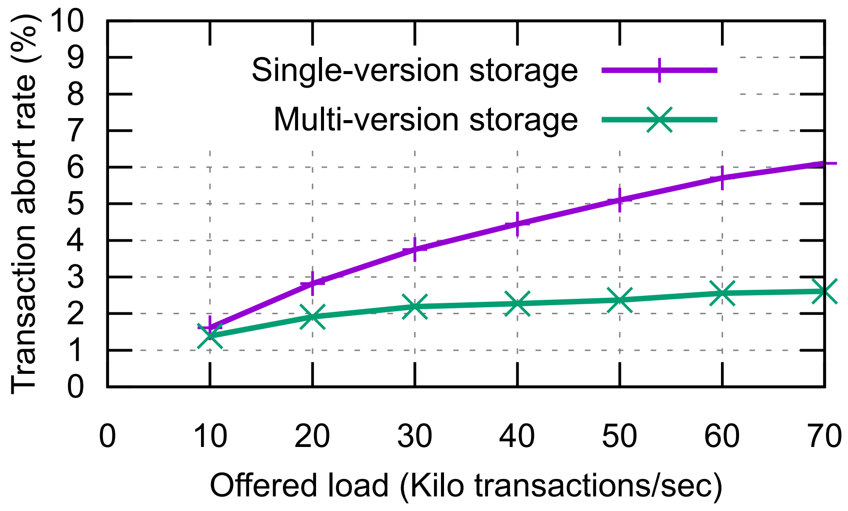


FIGURE 5.1: Transaction abort rate with single vs multi-version storage

management (garbage collection) scheme for effective capacity utilization. The version management scheme needs to strike a balance between capacity reclamation by discarding old versions and servicing reads from a consistent snapshot.

SEMEL (Chapter 3) addresses the capacity and version management challenges with designing a multi-version storage system. It uses Solid State Drives (SSDs) for storing multiple versions of data and a watermark-based version management for effective capacity utilization as watermarks provide a bound on the oldest version that can be read by the application. This chapter addresses the indexing challenge.

SSDs are a more attractive proposition for multi-versioning than DRAM or newer non-volatile memory (NVM) technologies (e.g., Intel Optane) because they provide TB capacity per drive for less than \$1.00/GB. In addition, newer standards like Software Defined Flash (SDF) allow designing flash storage systems based on application requirements [62, 131, 111, 88, 57, 92]. These advances in flash storage enable storing more data per server, while delivering better performance compared to spinning disks and at a lower cost compared to DRAM and NVMs.

Furthermore, SEMEL exploits an intrinsic property of flash-based SSDs — remap-on-write — to implement multi-versioning in an SSDs Flash Translation Layer (FTL) using SDF. Semel's approach removes abstractions and provides better performance compared to a naïve approach of stacking a multi-version software layer over a standard FTL. However, Semel incurs a high space overhead since it maintains the entire index for mapping a version of a key to its location on flash in host DRAM.

This chapter presents SkimpyFTL, a system that addresses all the 3 challenges with designing a multi-version key-value storage system. It builds on top of Semel by leveraging SSDs for FTL-integrated multi-versioning along with a watermark-based version management scheme and addresses the indexing challenge by providing a tradeoff between DRAM capacity and lookup latency. SkimpyFTL uses a hash table for mapping key versions to their values. It follows SkimpyStash [35] in offloading the hash collision list on flash to reduce DRAM requirement for indexing and adds support for multi-versioning using version pointers, which are also stored on flash along with the collision list.

A SkimpyFTL prototype utilizing LightNVM Open-Channel SSD emulation framework [18] reveals SkimpyFTL provides 72-91% throughput of Semel for read-dominant key-value workloads (75-100% reads), while reducing the memory requirement for indexing by a factor of 0.95×. For a transactional YCSB [30] workload, SkimpyFTL provides 85% peak throughput of Semel. Finally, SkimpyFTL outperforms a naïve multi-version key-value store implemented over a standard FTL on both workloads.

The rest of this chapter is organized as follows. Section 5.1 provides an overview on Semel FTL. We present the design of Kairos in Section 5.2. Section 5.3 presents results from a prototype of Kairos. We discuss the related work in Section 5.4. Finally, we summarize in Section 5.5.

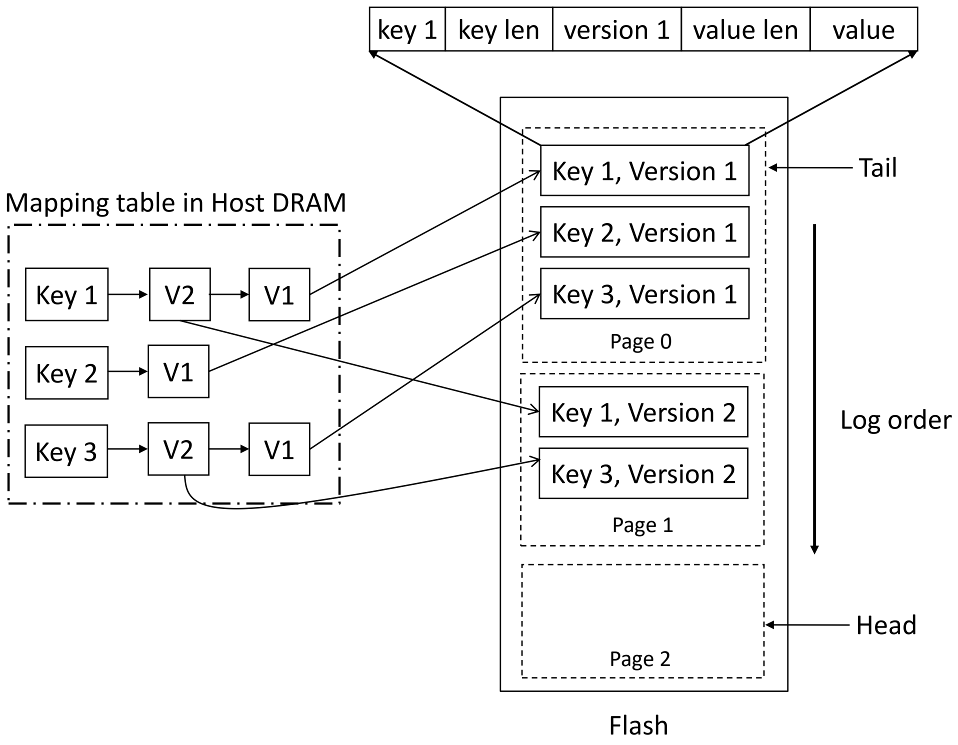


FIGURE 5.2: Mapping table and data layout on flash in SEMEL

5.1 Background

This section briefly describes how SEMEL uses SDF for designing a multi-version storage system. More details can be found in Section 3.1.

SEMEL is a lightweight multi-version key-value store based on SDF. It writes new values in a log-structured fashion [109], densely packed in pages on flash. Figure 5.2 shows the mapping table and data layout in SEMEL. As seen from the figure, SEMEL maintains a linked list in DRAM with an entry for each version of a key. Each version is assigned a 64-bit create timestamp and maps directly to a page on flash and the version's offset within the page, removing a level of indirection.

SEMEL's DRAM-based mapping table is prohibitively expensive. Each version entry is 20B in size (4B page address, 8B version timestamp and 8B pointer to prior version): for a 1 TB SSD and 512B key-value pairs, SEMEL consumes 40 GB of DRAM to map the entire SSD. The goal of SkimpyFTL is to provide DRAM-efficient version mapping with performance as close as possible to Semel.

5.2 SkimpyFTL: A Multi-Version Flash Translation Layer

This section describes how SkimpyFTL addresses the need for memory-efficient dynamic indexing of a multi-version store like Semel. Our approach combines a flash-based mapping table (§5.2.1) with a DRAM-based mapping translation cache for hot keys (§5.2.2). The two structures operate together to handle reads and writes efficiently in the common case (§5.2.3). The scheme also extends the garbage collection protocol (§5.2.4).

SkimpyFTL API is similar to other multi-version key-value stores, it takes a timestamp as an argument with get and put requests to return a consistent snapshot and set timestamp of a new version, respectively. Below we describe the SkimpyFTL API.

- **put(key, value, t_{put})**: Create a new version for the given key with version timestamp = t_{put}.
- **get(key, t_{get}) → value**: Return a version with timestamp $\leqslant t_{get}$.
- **delete(key)**: Delete all versions of the key.

5.2.1 Mapping Table

SkimpyFTL indexes the key-value pairs on flash with a hash-based mapping table whose buckets are rooted in host DRAM. Multiple key-value pairs may hash to the same bucket; SkimpyFTL handles such collisions using *linear chaining*, where key-value pairs in the same bucket are chained using a linked list. Rather than maintaining this collision list in DRAM, SkimpyFTL offloads it to flash; a hash table bucket in DRAM points to the head of the linear list on flash, and each entry

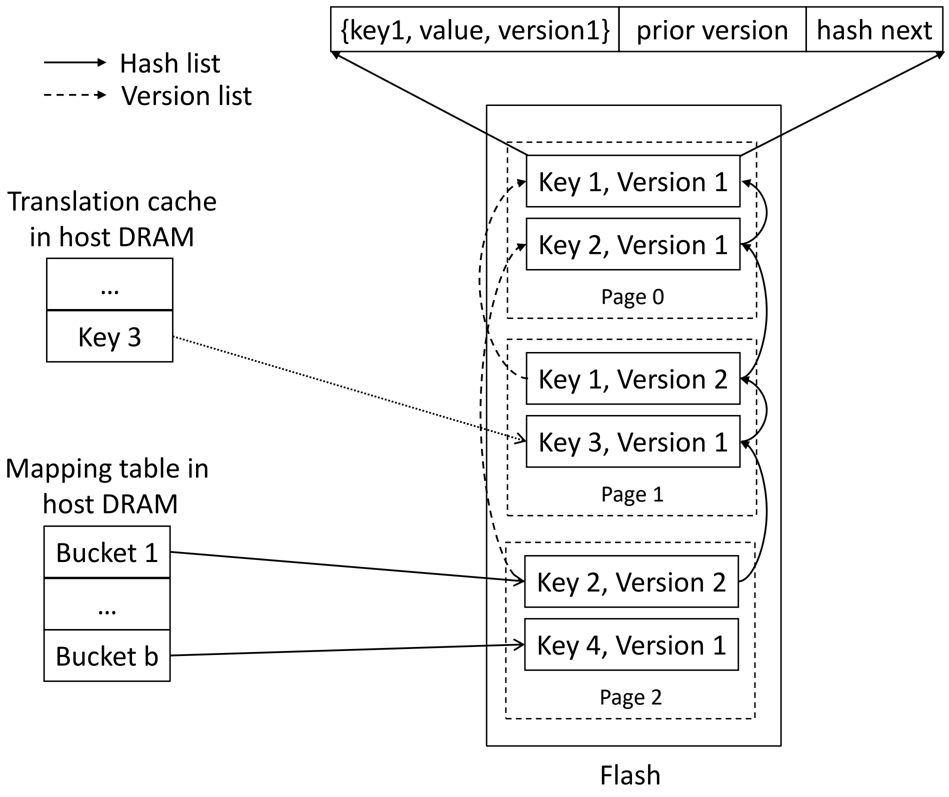


FIGURE 5.3: Mapping table, translation cache and data layout on flash in SKIMPYFTL

in the list on flash points to the next entry. This approach is inspired by a prior work [35]. SKIMPYFTL allows configuring the number of hash buckets to achieve a desired balance of DRAM cost and lookup time.

Figure 5.3 shows the mapping table and the data layout on flash. SKIMPYFTL writes data versions to flash in a log-structured fashion, as in SEMEL. In addition to the usual fields for each key-value pair (version), SKIMPYFTL stores two pointers: a *hash next* pointer to the next entry in the bucket's collision list (hash chain) and a *prior version* pointer to a prior (older) version of the key. The prior version pointer is an optimization: SKIMPYFTL can traverse a hash chain to find a requested version, but it is often faster to traverse the prior version pointer from a later version of the desired key, as described next.

5.2.2 Mapping Translation Cache

Prior characterization studies suggest that datacenter workloads tend to be read-dominated [11, 97]. Furthermore, the popularity of data items in real-world workloads often follows a power law distribution, where a small subset of the keys receive a large portion of the accesses [30, 11]. Such workloads would cause significant read amplification to traverse the collision lists. In particular, a key that is updated infrequently tends to migrate to the end of the list, since new versions are written to the front of the list.

To mitigate this cost, SkimpyFTL caches key translations in a *mapping translation cache* in DRAM. A translation cache entry for a key stores the location of its latest version on flash.

5.2.3 Request Life Cycle

Figure 5.3 illustrates how the DRAM-based translation cache operates in conjunction with the mapping table to handle GET (read) and PUT (write) requests efficiently.

A GET request first does a lookup to locate the latest version of the requested key. Then, if the request is a snapshot read for a previous version, it traverses the prior version pointer(s) to locate the requested version. A lookup for a hot or recent key hits in the translation cache, which returns the pointer to the latest version. On a cache miss, SkimpyFTL hashes the key to a bucket and then follows the hash chain until it finds an entry for the key, which is the latest version. It then caches the mapping in the translation cache.

For example, for a GET of {key 1, version 1} in Figure 5.3, SkimpyFTL first hashes key 1 to bucket 1 and traverses the hash chain: {key 2, version 2} → {key 3, version 1} → {key 1, version 2}, until it encounters key 1. It then updates the translation cache to point to the latest version of key 1.

A PUT request hashes the key to a bucket and links the new value to the front of the hash chain: it sets its hash next field to the current index pointed to by the bucket and updates the current index. To populate the prior version field, SkimpyFTL probes the translation cache for the most recent version of the key. This lookup typically results in a hit in the common case of a read-modify-write operation on the key. On a miss, SkimpyFTL sets the prior version field to a special value that indicates to a GET request that prior versions may exist and must be retrieved by searching further in the chain. It fills in the missing prior version pointers during remapping (§5.2.4).

For example, for a PUT request for key 2 in Figure 5.3, SkimpyFTL writes the new version to the end of the log on flash, hashes it to bucket 1, sets the hash next of key 2 to point to key 3 (on page 1) and updates bucket 1 to point to the new version. It sets the prior version field of key 2 to point to a prior version.

5.2.4 Garbage Collection

The garbage collection process starts from the tail of the log and remaps valid versions (key-value pairs) to the head of the log. For each key, SkimpyFTL retains the youngest version with a timestamp less than the current *watermark*, and discards older versions. The watermark is a timestamp that advances continuously. In the Semel transactional key-value store, the watermark is the minimum timestamp that could appear in any future request from a client, following Centiman [39]. Each client periodically passes its timestamp for its last acknowledged operation. The minimum of these timestamps is the watermark.

SkimpyFTL extends Semel's garbage collection to use the bucket collision lists and version lists, and to maintain their integrity. To determine whether to remap or discard a version, SkimpyFTL probes the translation cache and version list for a more recent version that is younger than the watermark. On a miss, it must traverse a bucket hash chain. For each remapped version, it must also update the hash next pointer of its predecessor in the chain to point at the new location. For this reason, SkimpyFTL garbage collects an entire hash bucket at a time by sweeping its hash chain and remapping retained versions in reverse temporal order, updating their pointers in the usual way as it goes.

To maintain fidelity of the hash chains and version lists, SkimpyFTL blocks any new writes to a bucket during the process of traversing its hash chain and remapping valid data. The remapping process packs the retained versions of a bucket's keys densely into flash pages, reducing lookup time for later GET requests.

5.3 Evaluation

We use a modified Open-Channel SSD framework [18] from our prior work [92] for all FTL implementations. In software-only mode, the emulator supports storing data values in DRAM, and provides IOCTLs for get, put and erase functionality for flash blocks. It also allows specifying latencies for read page, write page and block erase operations.

All our FTL implementations use 32 bits (4 bytes) for storing the location of a key on flash. To allow an FTL to pack multiple key-value pair within a page; we divide each page into fixed number of chunks and use 3 out of the 32 bits for storing the start chunk for a key in a page. The remaining 29 bits are used for addressing a page.

Workload. We use 2 types of workloads for evaluation: 1) key-value operations workload, and 2) transactional YCSB [30] workload. We implement a micro-benchmark for the key-value operations workload; the micro-benchmark issues non-transactional get and put requests to single keys, for a varying get request percentage. Popularity of keys in the benchmark is controlled using a zipfian distribution; we set the zipfian coefficient α to 0.99 in our micro-benchmark, a frequently used value in key-value workloads [30, 11]. Our transactional YCSB [30] workload models a social network application, where the data of popular users is read more often and users have different rate of posting updates. The workload models this behavior by using different values of α for controlling popularity of keys in read-only (α_r) and read-write (α_{rw}) transactions. Each read-only transaction accesses from 1 to 10 keys, and a read-write transaction operates on 1 to 5 keys.

Experimental Setup. All experiments are run on a server with a 16 core Intel Xeon E5-2640 processor clocked and 128 GB DRAM. The SSD emulator is backed by 32 GB DRAM, with a hardware queue depth of 128. The SSD has a page size of 4KB and there are 32 pages in a block. A flash page read, write time is 50 μs and 100 μs respectively and it takes 1 ms to erase a flash block. To ease garbage collection, all FTL implementations reserve 10% of available capacity for remapping data.

For all experiments, we populate the system with 20M keys; the key size is 16B and packed data on flash is 512B, which includes key, value, version, and pointers to prior version and next entry in hash collision chain. Since a flash page is 4KB in size, we employ a *packing logic* in the FTL that waits for up to 1 ms (tunable) to pack data of multiple keys into a page. There are up to 8 keys packed into a page, which increases the garbage collection overhead per page. Each run is of 15 mins and we pre-condition the SSD so garbage collection runs in the background in all experiments that perform writes.

Our SEMEL prototype stores the entire mapping table in DRAM. It needs $\sim$ 67M mappings (=32GB/512B) to address the entire SSD, and each mapping is 20B (4B page address, 8B version timestamp, and 8B prior version pointer) in size.

We perform sensitivity analysis to determine the size of translation cache and number of buckets in SKIMPYFTL. Figure 5.4 shows translation cache size vs hit rate under our key-value micro-benchmark with 10% writes and for a varying zipfian key selection coefficient α, with number of buckets = number of keys. As seen from

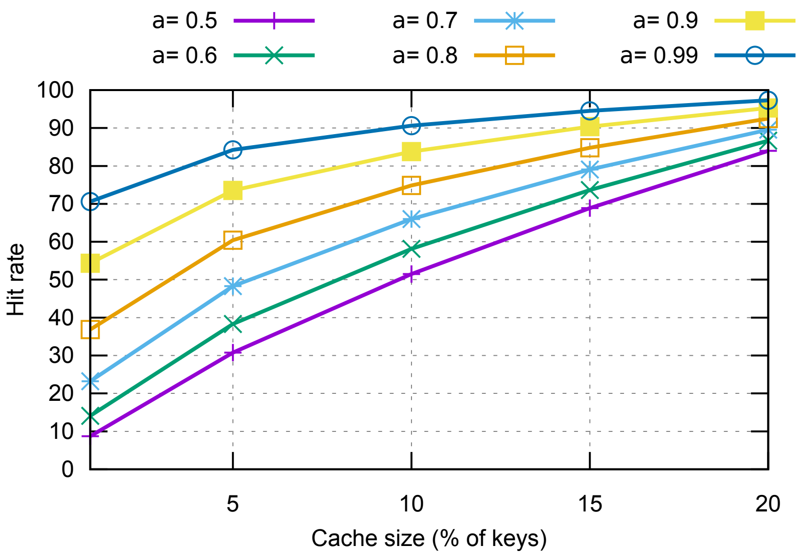


FIGURE 5.4: Translation cache size vs hit rate for varying values of α

the figure, the hit rate increases for a given cache size with increasing values of α. This is expected because the popularity skew in the workload increases with increasing values of α, which causes a small subset of keys in the workload to get disproportionate number of accesses. For $\alpha = 0.99$, a translation cache size of 10% of the number of keys in the workload gives a $\sim$ 90% hit rate. Consequently, we set the translation cache size to 10% for all experiments. After fixing the size of the translation cache, we evaluate the impact of number of buckets in the hashmap for the same workload, with 10% writes and $\alpha = 0.99$. Figure 5.5 shows the impact of average number of keys / bucket on throughput. Based on the results, we size the mapping table to have 5 keys / bucket i.e., 4M buckets. Each entry in the mapping table is 4B (page address), and 16B (8B page address, 16B LRU pointers) in the translation cache. SKIMPYFTL uses $\sim$ 5% of the memory used by SEMEL. We assume the memory requirement for storing keys is the same in both systems.

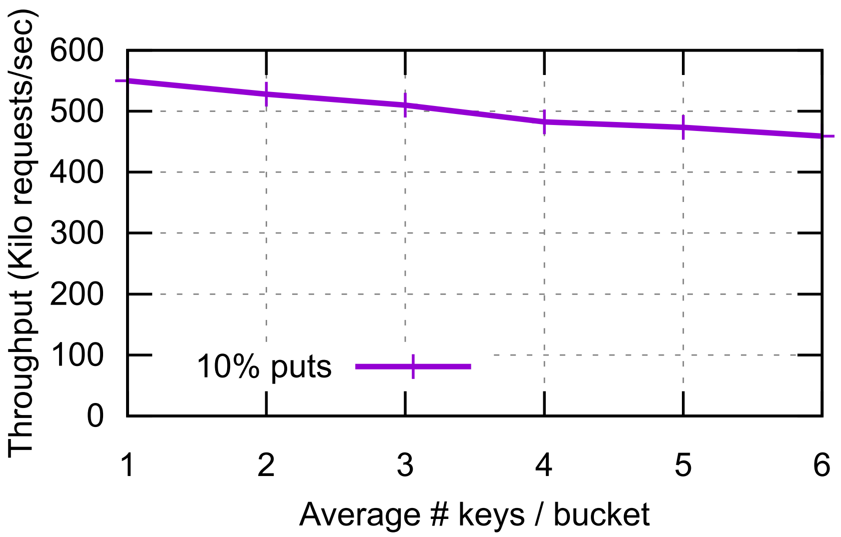


FIGURE 5.5: Impact of # keys / bucket

5.3.1 Key-Value Workload

We first evaluate the performance of all systems with a non-transactional key-value workload, for a varying put request percentage.

Figure 5.6 shows the throughput for varying put request % with our key-value micro-benchmark. The throughput of all 3 systems — SEMEL, VFTL and SKIMPYFTL—

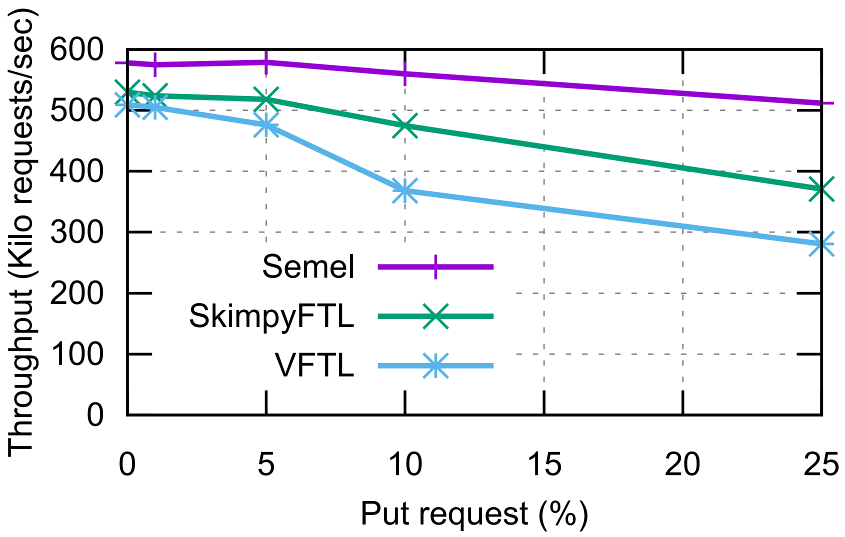


FIGURE 5.6: Throughput with key-value micro-benchmark

drops as put request percentage is increased because writes increase the garbage collection overhead. As expected, Semel provides the highest throughput since it implements multi-versioning in flash and maps all versions of all keys in DRAM. SkimpyFTL provides from 72% - 91% of the throughput of Semel, while only using 5% of the memory. The throughput degradation is worse in SkimpyFTL as put request percentage increases because garbage collection overhead — traversing entire hash collision chains for determining data to discard and remap — is more frequent. We plan to explore other approaches to garbage collection, with lower overheads, in the future. Interestingly, the throughput of VFTL, which is implemented by stacking a multi-version layer over a generic FTL, is even lower than SkimpyFTL. VFTL provides the lowest throughput because it suffers from log stacking [128] — the log in multi-version layer and generic FTL operate independently, which leads to uncoordinated garbage collection and randomization of writes.

5.3.2 Transactional Workload

To evaluate performance for a transactional workload, we stack a layer over the 3 systems that supports transactions using MVCC [15]. The memory overhead of the layer is the same for all systems. Our YCSB [30] workload issues 90% read-only transactions, with zipfian coefficient for read-only and read-write transactions set to $\alpha_r = 0.99$ and $\alpha_{rw} = 0.75$, respectively.

Figure 5.7 shows the throughput and latency with the 3 storage systems for an increasing offered load. The trend in performance of the systems is similar to the key-value workload. Semel provides the best performance; SkimpyFTL provides 85% of the peak throughput of Semel. VFTL provides the lowest peak throughput and highest latency. Thus, showing the disadvantage of the naïve approach of implementing multi-versioning, agnostic of the underlying storage medium. All approaches have similar transaction commit rates ($\sim$ 93%).

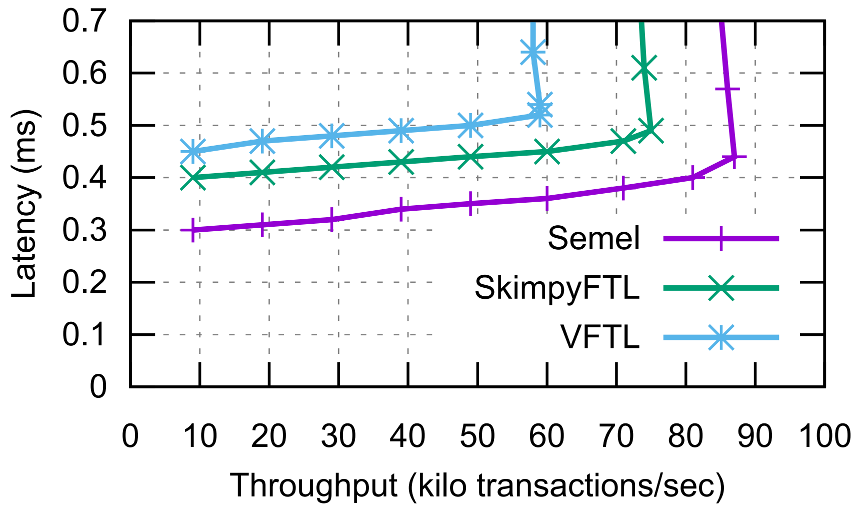


FIGURE 5.7: Throughput vs Latency for a transactional workload

5.4 Related Work

Prior works have proposed indexes for flash-based key-value storage systems [10, 9, 34, 81, 82, 35, 117, 88, 72]. However, none of these works support multi-versioning.

Several works use a hashmap for indexing. FAWN [10] stores key-value pairs in an append-only log on the SSD, and uses an in-memory hash-based index for fast lookups. BufferHash [9] uses multiple in-memory hash-based indexes, with bloom filters to choose which hash index to use for a lookup. SILT [82] stores data in multiple tiers; as key-value pairs age, they are compacted with other pairs and transitioned to other skimpy memory-optimized tiers. FlashStore [34] and SkimpyStash [35] also use a hash-based index, but optimize its size; FlashStore uses cuckoo hashing and compact key signatures to reduce the size of the index, while SkimpyStash uses a configurable number of hash buckets to reduce DRAM requirement and moves the hash collision list to flash. SKIMPYFTL builds on top of SkimpyStash's approach and adds support for multi-versioning and a translation cache. NVMKV [88] is a FTL-aware key-value store which uses native FTL capabilities, such as sparse addressing, and transactional supports to directly map keys to values on flash.

Tree-based data structures have also been proposed for indexing in flash-based SSDs. WiscKey [72] and FD-tree [81] use a tree-based data structure for indexing data on SSDs. WiscKey decouples keys and values from an LSM [99] tree for reducing read and write amplification in SSDs. FD-tree proposes a new tree-based data structure for indexing key-value pairs on SSD that uses logarithmic method for storing data (like LSM) and adds fractional cascading to speed up read operations. FD+tree [117] adds new features to FD-tree, such as level skipping for fast reads and tightening for tree compaction, to make them practical.

5.5 Summary

Maintaining multiple versions of data is popular in key-value stores as it increases concurrency and improves performance. However, designing a multi-version key-value store entails several challenges, such as additional capacity for storing extra versions and an indexing mechanism for mapping versions of a key to their values.

This chapter presents SkimpyFTL, a system that addresses all the 3 challenges with designing a multi-version key-value storage system. It builds on top of Semel by leveraging SSDs for FTL-integrated multi-versioning along with a watermark-based version management scheme and addresses the challenge with indexing by providing a tradeoff between DRAM capacity and lookup latency.

Our evaluation reveals SkimpyFTL provides 72-91% throughput of Semel for a key-value workload (75-100% reads), while reducing memory requirement by a factor of 0.95x. For a transactional workload, SkimpyFTL provides 85% peak throughput of Semel. We also show the benefit of implementing multi-versioning in the FTL — SkimpyFTL outperforms a naïve multi-version key-value store implemented over a standard FTL on both workloads.

6

Conclusion

Large-scale datacenters provide the computational infrastructure that underlies the increasing use of cloud services. A key aspect in many datacenters is the use of commodity hardware to provide scale-out cloud infrastructure for services such as Software as a Service(SaaS), Hardware as a Service (HaaS) and the more generalized Anything as a Service (XaaS). These services provide many enabling features (e.g., *elasticity*, *high availability*, *low time to market*, and *transfer of risks* etc.) that make cloud computing a ubiquitous paradigm for deploying applications spanning all aspects of the human endeavor. Analysts forecast that global cloud services revenue will reach $410 billion by 2020 [50].

Given the scale and criticality of cloud services, it is crucial to continually examine existing, new and emerging features available to enhance these services. Our work focuses on transactional key-value storage, an important service inside datacenters [31, 116, 41, 39, 130, 75, 42]. Transactional key-value storage systems are popular because they provide high-level guarantees like consistency, scalability and fault-tolerance to ease application development. Unfortunately, providing good performance (high throughput and low latency) without high complexity is often a challenge for these storage systems due to several sophisticated protocols for providing the high-level guarantees (e.g., transaction and replication protocol), and the overheads incurred by traversing various abstraction levels.

We leverage two emerging datacenter capabilities — precise synchronized clocks and software-defined storage — to address the performance and complexity challenges with transactional key-value storage systems. To this end, we use a cross-layer approach that investigates all levels of the storage stack, from developer APIs to underlying hardware. We show that this methodology opens avenues for synergistic interactions between software and underlying hardware, and leads to simpler system designs and better performance.

6.1 Key Contributions

We address the following following challenges with transactional key-value storage systems: 1) multi-version storage protocol, 2) ordering constraint in replication protocol, 3) high abort rate due to clock skew with transaction protocol (Optimistic Concurrency Control), and 4) explicit invalidation overhead with inter-transaction caching protocol.

This dissertation presents 4 systems — Semel, Milana, Kairos and SkimpyFTL— each designed to address a particular aspect of the complexity and performance challenge with transactional key-value storage systems. Below we summarize the key contributions made by each system.

Semel is a multi-version key-value storage system that provides access to single key-value objects, without any support for transactions. It leverages precise synchronized clocks to simplify the replication protocol used by key-value storage systems. All writes in Semel are timestamped with precision time; these timestamps enable a lightweight primary-backup *inconsistent* replication protocol that moves update ordering off the critical path — all writes are immediately acknowledged by backup storage servers, irrespective of the order they come in. The correct order is re-created using timestamps at the time of a storage primary failure. Furthermore, Semel addresses the storage capacity and version management challenge with a multi-version storage by exploiting remap-on-write property of flash-based SSDs for multi-versioning and a watermark-based version management scheme for effective capacity utilization, respectively.

Evaluation of a Semel prototype using the Precision Time Protocol [60] and the LightNVM Open-Channel SSD emulation framework [18] reveals a 20-45% increase in throughput and up to 7× lower GET latency on a single machine using unified version and flash management compared to a naïve multi-version key-value storage system implemented over a standard Flash Translation Layer (FTL) for read heavy workloads (50-100% GET ops).

Milana adds OCC to support serializable ACID transactions over Semel. Milana uses timestamps from precise synchronized clocks to address the problem of high abort rates with OCC. We show that clock skew with Network Time Protocol (NTP) [91] is too high for modern low-latency datacenters and that the Precision Time Protocol (PTP) [60] enables use of OCC with low abort rates, even in high-contention scenarios. Furthermore, Milana uses precise synchronized clocks to eliminate server-side validation of all read-only transactions — all such transactions are validated locally on the client, which reduces load on servers and improves performance and eliminates two round-trip of messages (for validation).

Evaluation of our Milana prototype using Retwis [73] shows up to 43% reduction in abort rates using PTP vs. NTP due to tighter clock synchronization. The local client validation optimization for all read-only transactions in Milana reduces transaction latency by 35% and increases throughput by 55% for read-heavy workloads.

Kairos builds on the approach of Milana by using precise synchronized clocks to enable physical time-based consistency integrated with transactional concurrency control, and adds support for inter-transaction caching and sharded validation. Precise synchronized clocks enable a lease-based inter-transaction caching protocol, without tracking sharers or sending any invalidations; clients dynamically self-invalidate data on lease expiration. The central challenge for a lease-based approach is to set lease times to balance the hit ratio with the cost of stale reads. Kairos addresses this challenge using the observed inter-access (read and write) times of popular keys to adapt lease durations dynamically for each key to optimize this tradeoff according to an analytical model. In addition, KAIROS also leverages sharded validation from Centiman [39] to decouple transaction validation from the servers, so that validation scales independently of the storage tier; KAIROS adapts this sharded validation to support inter-transaction caching.

Evaluation of a KAIROS prototype using a YCSB workload [30] reveals that inter-transaction caching alone improves throughput by 1.86x relative to a baseline system with only intra-transaction caching; adding sharded validation further improves throughput by a factor of 2.28 under a workload with a hotspot that saturates a storage primary. Furthermore, our evaluation shows that lease-based inter-transaction caching can support 62.5% more clients while providing 1.55x the throughput of classical callback leases (explicit invalidation) in workloads with hot keys.

SKIMPYFTL builds on top of SEMEL and addresses the requirement for memory-efficient indexing in multi-version storage. SKIMPYFTL uses a hashmap-based approach for enabling a tradeoff between memory capacity and lookup latency for indexing. It reduces memory requirement for indexing by tuning the size of the hash index (number of buckets); a smaller index reduces memory requirement, but causes more keys to be hashed to the same bucket, with the collision list of a bucket stored on flash. This memory size reduction impacts the lookup latency for indexing because more flash reads need to be performed to lookup a key.

Evaluation of a SkimpyFTL prototype under a key-value micro-benchmark shows that SkimpyFTL provides 72-91 % of the throughput of Semel for read-dominant workloads (75-100% reads), while reducing the memory requirement for indexing by a factor of 0.95×. For a transactional YCSB [30] workload, SkimpyFTL provides 85% of peak throughput of Semel. Finally, SkimpyFTL outperforms a naïve multi-version key-value storage system implemented over a standard FTL on both workloads.

6.2 Directions for Future Work

Precise synchronized clocks and Non-volatile Memory. Research advances continue to tighten the bounds on clock skew: recent work demonstrates ≈ 100-150 ns skew across a datacenter [76, 53]. These tolerances are within the access latencies of newer non-volatile memory technologies (e.g., STTRAM, PCM etc.) and are fast approaching those of DRAM. One research direction is using precise synchronized clocks to design distributed shared memories that leverage the low clock skew for deterministic operation execution. For example, precise clocks can be used for operation ordering and enforcing a memory system consistency model.

Distributed Flash Management. Software-defined storage enables tailoring storage devices (e.g., SSDs) according to application requirements. Semel and SkimpyFTL leverage software-defined storage to design a lightweight multi-version storage using SSDs. However, both systems run the modified Flash Translation Layer on storage server processors. With presence of multiple general-purpose cores on storage controllers [66] and newer standards like NVME over Fabric (NVMEoF) for communicating with a storage controller over the network, an interesting research avenue is to explore how to distribute the flash management from the storage server to the clients of the storage system. One approach is to distribute flash blocks on an SSD(s) to the clients of the system; each client reads, writes and performs garbage collection of the blocks that it owns through remote commands sent to the storage controller using NVMEoF. However, this approach needs a communication framework across the clients to synchronize accesses (reads and writes) to any shared data.

Integrating caching with local validation of read-only transactions. Milana supports local validation of read-only transactions, which reduces validation load on the server and eliminates one round-trip of messages and latency for the client. On the other hand, Kairos supports inter-transaction caching, which reduces the server load for data accesses along with latency reduction for clients as frequently-read data can be serviced from a client's cache. However, Kairos requires server-side validation of all transactions (including read-only) as client caches can contain stale data, and needs to be validated by a centralized authority (server or validator) to determine whether a transaction read any stale data. An interesting avenue of research is integrating inter-transaction caching with local validation to enable servicing read-only transactions entirely on the client, without requiring any communication with storage servers. One approach is to define epochs for keys; a key can be in a read or write epoch, the duration of which can be based on inter-access times. Clients can aggressively cache keys in read epochs, and perform local validation of transactions that only access keys in read epochs.

Multi-version indexing in flash-based SSDs. Other data structures can be used for indexing in Semel and SkimpyFTL, such as log-structured merge (LSM) trees [99]. Like the hash-based index in SkimpyFTL, LSM trees reduce DRAM footprint as large portions of the tree can be stored on flash. In addition, they can also potentially reduce the garbage collection overhead of the SkimpyFTL's hashmap-based approach as keys can be remapped independently. Finally, LSM trees provide better performance for range queries. However, maintaining multiple versions of data in an LSM tree is a challenge. A naïve approach of searching successive levels for a consistent snapshot of a key will provide poor performance. In contrast, a version pointer-based approach (like currently used in SkimpyFTL) wherein each version points to a prior version can be leveraged to improve performance but maintaining fidelity of these chains is a challenge since compaction of levels will break the version pointers. An interesting research direction is to explore techniques for incorporating multi-versioning with LSM-based indexing.

6.3 Summary

Distributed transactional storage is an important service in datacenters. Unfortunately, providing good performance without high complexity entails several challenges for these storage systems due to use of several sophisticated protocols and various levels of abstraction.

We leverage two emerging capabilities — precise synchronized clocks and software-defined storage — to architect transactional key-value storage systems in datacenters. This dissertation presents 4 systems — Semel, Milana, Kairos and SkimpyFTL— each designed to address a particular aspect of the complexity and performance challenge with transactional key-value storage. To this end, we use a cross-layer approach that investigates all levels of the storage stack, from developer APIs to underlying hardware. We show that this methodology opens avenues for synergistic interactions between software and underlying hardware, and leads to simpler system designs and better performance.

Bibliography

[1] The evolution of microservices. `https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference`.

[2] Atul Adya, William J Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R Douceur, Jon Howell, Jacob R Lorch, Marvin Theimer, and Roger P Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *OSDI*, 2002.

[3] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 23–34. ACM, 1995.

[4] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. Slicer: Auto-sharding for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 739–753, Savannah, GA, 2016. USENIX Association.

[5] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 57–70. USENIX Association, 2008.

[6] Sandeep R. Agrawal, Christopher M. Dee, and Alvin R. Lebeck. Exploiting accelerators for efficient high dimensional similarity search. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 3:1–3:12, New York, NY, USA, 2016. ACM.

[7] Sandeep R. Agrawal, Valentin Pistol, Jun Pang, John Tran, David Tarjan, and Alvin R. Lebeck. Rhythm: Harnessing data parallel hardware for server workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 19–34, New York, NY, USA, 2014. ACM.

[8] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

[9] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. Cheap and large cams for high performance data-intensive networked systems. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 29–29, Berkeley, CA, USA, 2010. USENIX Association.

[10] 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*, SOSP '09, pages 1–14. ACM, 2009.

[11] 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*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[12] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, volume 11, pages 223–234, 2011.

[13] Ilya Baldin, Jeff Chase, Yufeng Xin, Anirban Mandal, Paul Ruth, Claris Castillo, Victor Orlikowski, Chris Heermann, and Jonathan Mills. *ExoGENI: A Multi-Domain Infrastructure-as-a-Service Testbed*, pages 279–315. Springer International Publishing, Cham, 2016.

[14] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 47–60, Berkeley, CA, USA, 2010. USENIX Association.

[15] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control - theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, December 1983.

[16] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[17] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. Hyder - A transactional record manager for shared flash. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 9–20, 2011.

[18] Matias Bjørling, Javier González, and Philippe Bonnet. Lightnvm: The linux open-channel ssd subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, 2017.

[19] Dhruba Borthakur. Under the hood: Building and open-sourcing rocksdb. *Facebook Engineering Notes*, 2013.

[20] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[21] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 385–395, Dec 2010.

[22] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 387–400, New York, NY, USA, 2012. ACM.

[23] 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 - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[24] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, ICALP '02, pages 693–703, London, UK, UK, 2002. Springer-Verlag.

[25] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 26:1–26:17, New York, NY, USA, 2016. ACM.

[26] Houssem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, and Maria S Perez. Harmony: Towards automated self-adaptive consistency in cloud storage. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 293–301. IEEE, 2012.

[27] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From aries to mars: Transaction support for next-generation, solid-state drives. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 197–212, New York, NY, USA, 2013. ACM.

[28] Adrian Cockroft. Microservices workshop: Why, what, and how to get there. `https://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference`.

[29] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[30] 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*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[31] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264. USENIX Association, 2012.

[32] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's Globally Distributed Database. In *OSDI*, 2012.

[33] Jeffrey Dean and Sanjay Ghemawat. leveldb–a fast and lightweight key/value database library by google, 2011.

[34] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: High throughput persistent key-value store. *Proc. VLDB Endow.*, 3(1-2):1414–1425, September 2010.

[35] Biplob Debnath, Sudipta Sengupta, and Jin Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 25–36, New York, NY, USA, 2011. ACM.

[36] Biplob Debnath, Sudipta Sengupta, and Jin Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 25–36. ACM, 2011.

[37] 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, pages 205–220, New York, NY, USA, 2007. ACM.

[38] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 1–12, New York, NY, USA, 1987. ACM.

[39] Bailu Ding, Lucja Kot, Alan Demers, and Johannes Gehrke. Centiman: Elastic, high performance optimistic concurrency control by watermarking. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 262–275. ACM, 2015.

[40] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, 2017.

[41] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 401–414, Berkeley, CA, USA, 2014. USENIX Association.

[42] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 54–70, New York, NY, USA, 2015. ACM.

[43] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Warp: Lightweight multi-key transactions for key-value stores. *CoRR*, abs/1509.07815, 2015.

[44] Jose M. Faleiro and Daniel J. Abadi. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow.*, 8(11):1190–1201, July 2015.

[45] 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 '11, pages 23:1–23:12, New York, NY, USA, 2011. ACM.

[46] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):5666, 1988.

[47] B. Fleisch and G. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 211–223, New York, NY, USA, 1989. ACM.

[48] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, June 2018.

[49] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayantara Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Yuan He, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. In *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2019.

[50] Gartner. Gartner forecasts worldwide public cloud services revenue to reach $410 billion in 2020, 2017. `https://www.gartner.com/newsroom/id/3815165`.

[51] Alex Gartrell, Mohan Srinivasan, Bryan Alger, and Kumar Sundararajan. Mcdipper: A key-value cache for flash storage. `https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920`.

[52] Vasileios Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijayanand Nagarajan. *Scale-Out ccNUMA: Exploiting Skew with Strongly Consistent Caching.* 1 2018.

[53] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 81–94, Renton, WA, 2018. USENIX Association.

[54] 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 '89, pages 202–210, New York, NY, USA, 1989. ACM.

[55] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis of HDFS under hbase: A facebook messages case study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 199–212, Santa Clara, CA, 2014. USENIX.

[56] Apache HBase. A distributed database for large datasets. *The Apache Software Foundation, Los Angeles, CA. URL http://hbase. apache. org*, 4(4.2).

[57] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds. In *FAST'17*. USENIX, February 2017.

[58] Jian Huang, Anirudh Badam, Moinuddin K. Qureshi, and Karsten Schwan. Unified address translation for memory-mapped ssds with flashmap. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 580–591. ACM, 2015.

[59] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11. USENIX Association, 2010.

[60] IEEE. Ieee standard for a precision clock synchronization protocol for networked measurement and control systems. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pages 1–269, 2008.

[61] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, New York, NY, USA, 2017. ACM.

[62] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. Dfs: A file system for virtualized flash storage. *Trans. Storage*, 6(3):14:1–14:25+, September 2010.

[63] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[64] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, New York, NY, USA, 2014. ACM.

[65] 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 '97, pages 654–663, New York, NY, USA, 1997. ACM.

[66] Gunjae Koo, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: Trading communication with computing near storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 219–231, New York, NY, USA, 2017. ACM.

[67] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. ACM.

[68] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[69] An-Chow Lai and Babak Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 139–148, New York, NY, USA, 2000. ACM.

[70] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Operating Systems Review*, 44(2):35–40, April 2010.

[71] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[72] Lanyue Lu and Thanumalayan Sankaranarayana Pillai and Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, February 2016. USENIX Association.

[73] Costin Leau. Spring data redis retwis-j, 2013. `http://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/`.

[74] Alvin R. Lebeck and David A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 48–59, New York, NY, USA, 1995. ACM.

[75] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 71–86, New York, NY, USA, 2015. ACM.

[76] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *To appear in Proceedings of ACM SIGCOMM*, August 2016.

[77] Charles E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, 34(10):892–901, October 1985.

[78] Scott T Leutenegger and Daniel Dias. *A modeling study of the TPC-C benchmark*, volume 22. ACM, 1993.

[79] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th*

Symposium on Operating Systems Principles, SOSP '17, pages 137–152, New York, NY, USA, 2017. ACM.

[80] 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)*, pages 31–44, Santa Clara, CA, 2016. USENIX Association.

[81] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. Tree indexing on solid state drives. *Proc. VLDB Endow.*, 3(1-2):1195–1206, September 2010.

[82] 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 '11, pages 1–13. ACM, 2011.

[83] Barbara Liskov. Practical uses of synchronized clocks in distributed systems. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '91, pages 1–9, New York, NY, USA, 1991. ACM.

[84] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012.

[85] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 795–809, New York, NY, USA, 2017. ACM.

[86] Rick Macklem. Not quite nfs, soft cache consistency for nfs. In *USENIX Winter*, pages 261–278, 1994.

[87] Ratul Mahajan and Roger Wattenhofer. On consistent updates in software defined networks. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 20:1–20:7, New York, NY, USA, 2013. ACM.

[88] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. Nvmkv: A scalable and lightweight flash aware key-value store. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, Philadelphia, PA, June 2014. USENIX Association.

[89] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[90] 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'05, pages 398–412, Berlin, Heidelberg, 2005. Springer-Verlag.

[91] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, Oct 1991.

[92] Pulkit A. Misra, Jeffrey S. Chase, Johannes Gehrke, and Alvin R. Lebeck. Enabling lightweight transactions with precision time. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 779–794, New York, NY, USA, 2017. ACM.

[93] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 103–114, Berkeley, CA, USA, 2013. USENIX Association.

[94] Shubhendu S. Mukherjee and Mark D. Hill. Using prediction to accelerate coherence protocols. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, pages 179–190, Washington, DC, USA, 1998. IEEE Computer Society.

[95] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 677–689, New York, NY, USA, 2015. ACM.

[96] Anoop Ninan, Purushottam Kulkarni, Prashant Shenoy, Krithi Ramamritham, and Renu Tewari. Cooperative leases: Scalable consistency maintenance in content distribution networks. In *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, pages 1–12, New York, NY, USA, 2002. ACM.

[97] 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 *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[98] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17. ACM, 1988.

[99] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[100] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-defined Flash for Web-scale Internet Storage Systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 471–484, New York, NY, USA, 2014. ACM.

[101] Xiangyong Ouyang, David Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K. Panda. Beyond block i/o: Rethinking traditional storage primitives. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 301–311, Washington, DC, USA, 2011. IEEE Computer Society.

[102] Vivek S. Pai, Mohit Aron, Gaurov Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 205–216, New York, NY, USA, 1998. ACM.

[103] Brian Pawlowski, David Noveck, David Robinson, and Robert Thurlow. The nfs version 4 protocol. In *In Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000*, 2000.

[104] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized "zero-queue" datacenter network. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 307–318, New York, NY, USA, 2014. ACM.

[105] R. Pitchumani, S. Frank, and E. L. Miller. Realistic request arrival generation in storage benchmarks. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, May 2015.

[106] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 13–24, Piscataway, NJ, USA, 2014. IEEE Press.

[107] Zujie Ren, Biao Xu, Weisong Shi, Yongjian Ren, Feng Cao, Jiangbin Lin, and Zheng Ye. igen: A realistic request generator for cloud file systems benchmark-

ing. In *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*, pages 343–350. IEEE, 2016.

[108] Alberto Ros and Stefanos Kaxiras. Complexity-effective multicore coherence. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 241–252, New York, NY, USA, 2012. ACM.

[109] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 1–15. ACM, 1991.

[110] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for dram-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 1–16, Berkeley, CA, USA, 2014. USENIX Association.

[111] Mohit Saxena, Michael M. Swift, and Yiying Zhang. Flashtier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 267–280. ACM, 2012.

[112] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable ssd. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 67–80, Berkeley, CA, USA, 2014. USENIX Association.

[113] Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Snapshots in a flash with iosnap. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 23:1–23:14. ACM, 2014.

[114] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 18–18, Berkeley, CA, USA, 2012. USENIX Association.

[115] Steven Swanson and Adrian Caulfield. Refactor, reduce, recycle: Restructuring the i/o stack for the future of storage. *Computer*, 46(8):52–59, August 2013.

[116] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12. ACM, 2012.

[117] Risi Thonangi, Shivnath Babu, and Jun Yang. A practical concurrent index for solid-state drives. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pages 1332–1341, New York, NY, USA, 2012. ACM.

[118] Kishor S Trivedi. *Probability & statistics with reliability, queuing and computer science applications*. John Wiley & Sons, 2008.

[119] Twitter. Decomposing twitter: Adventures in service- oriented architecture. `https://www.slideshare.net/InfoQ/decomposing-twitter-adventures-in-serviceoriented-architecture`.

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

[121] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.

[122] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 87–104, New York, NY, USA, 2015. ACM.

[123] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Vinay Sridhar, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. MjÖlnir: Collecting trash in a demanding new world. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '15, pages 4:1–4:10. ACM, 2015.

[124] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Anvil: Advanced virtualization for modern non-volatile memory devices. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 111–118. USENIX Association, 2015.

[125] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10(7):781–792, March 2017.

[126] P. Xiong, Y. Chi, S. Zhu, H. J. Moon, C. Pu, and H. Hacgm. Smartsla: Cost-sensitive management of virtualized resources for cpu-bound database services. *IEEE Transactions on Parallel and Distributed Systems*, 26(5):1441–1451, May 2015.

[127] Karim Yaghmour and Jean-Hugues Deshchenes. Linux trace toolkit reference manual, 2004.

[128] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't stack your log on my log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*, Broomfield, CO, 2014. USENIX Association.

[129] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sanchez, Larry Rudolph, and Srinivas Devadas. Sundial: Harmonizing concurrency control and caching in a distributed oltp database management system. *Proc. VLDB Endow.*, 11(10):1289–1302, June 2018.

[130] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 263–278. ACM, 2015.

[131] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 1–1. USENIX Association, 2012.

[132] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 3–18, New York, NY, USA, 2015. ACM.

Biography

Pulkit A. Misra earned his B.E. in Electronics from The Maharaja Sayajirao University of Baroda, Vadodara, India in 2010. His first foray in to graduate school resulted in a M.S. in Computer Science degree from Northeastern University, Boston MA in 2013. He also worked at VeloBit, a computer storage startup company, while pursuing his masters. At VeloBit, he worked on similarity detection and delta compression of data for reducing writes and increasing lifetime of NAND flash-based Solid State Drives (SSDs). His work was instrumental in the company's acquisition by HGST Inc. in 2013. He filed several patents while working for HGST and VeloBit, 4 of which have been granted by USPTO.

He came back to graduate school in 2015 for getting a PhD in Computer Science from Duke University. His PhD research was on leveraging emerging datacenter capabilities for enhancing transactional storage service in datacenters, for which he received the Outstanding Research Initiation Project (RIP) Award by the Computer Science Department in 2017. He also interned at Microsoft Research during his PhD, where he worked on creating and managing tail latency in datacenter-scale filesystems. After his PhD, he will be joining the Cloud Efficiency group at Microsoft Research in April 2019.