PRACTICAL LOW-LATENCY KEY-VALUE STORES

by

Chinmay Satish Kulkarni

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Chinmay Satish Kulkarni**
has been approved by the following supervisory committee members:

Ryan Stutsman ,	Chair(s)	__ Date Approved
Robert Ricci ,	Member	__ Date Approved
John Regehr ,	Member	__ Date Approved
Kobus Van Der Merwe ,	Member	__ Date Approved
Badrish Chandramouli ,	Member	__ Date Approved

by **Mary Hall** , Chair/Dean of
the Department/College/School of **Computing**
and by **David B Kieda** , Dean of The Graduate School.

ABSTRACT

The last decade of computer systems research has yielded efficient kernel-bypass stores with throughput and access latency thousands of times better than conventional stores. These gains come from careful attention to detail in request processing, so these systems often start with simple and stripped-down designs to achieve their performance goals. Hence, they trade off features that would make them more practical and cost effective at cloud scale, such as load (re)distribution, multi-tenancy, and expressive data models.

This thesis demonstrates that this trade off is unnecessary. It presents mechanisms for reconfiguration, multi-tenancy and expressive data models that would make these systems more practical and efficient at cloud scale while preserving their performance benefits.

Rocksteady is a live migration technique for scale-out in-memory key-value stores. It balances three competing goals: it migrates data quickly, it minimizes response time impact, and it allows arbitrary, fine-grained splits. Rocksteady allows a key-value store to defer all repartitioning work until the moment of migration, giving it precise and timely control for load balancing.

Shadowfax is a system that allows distributed key-value stores to transparently span DRAM, SSDs, and cloud blob storage while serving 130 Mops/s/VM over commodity Azure VMs using conventional Linux TCP. Beyond high performance, Shadowfax uses a unique approach to distributed reconfiguration that avoids any server-side key ownership checks or cross-core coordination both during normal operation and migration.

Splinter is a system that allows clients to extend low-latency key-value stores by migrating (pushing) code to them. Splinter is designed for modern multi-tenant data centers; it allows mutually distrusting tenants to write their own fine-grained extensions and push them to the store at runtime. The core of Splinter's design relies on type- and memory-safe extension code to avoid conventional hardware isolation costs. This still allows for bare-metal execution, avoids data copying across trust boundaries, and makes granular storage functions that perform less than a microsecond of compute practical.

Friends are the family we choose for ourselves

– Edna Buchanan

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

CHAPTER 1

INTRODUCTION

The last decade of computer systems research has yielded efficient kernel-bypass stores with throughput and access latency thousands of times better than conventional stores. Today, these systems can execute millions of operations per second with access times of 5 µs or less [29, 68, 83]. These gains come from careful attention to detail in request processing, so these systems often start with simple and stripped-down designs to achieve their performance goals.

However, for these low-latency stores to be practical in the long-term, they must evolve to include many of the features that conventional data center and cloud storage systems have while preserving their performance benefits. Key features include the ability to reconfigure and (re)distribute load (and data) in response to load imbalances and failures, which occur frequently in practice; the ability to perform such reconfiguration in the cloud, where networking stacks have been historically slow, and where resources are shared by multiple tenants; and the ability to support a diverse set of such tenants with varying access patterns, data models and performance requirements.

Load Distribution: When it comes to load distribution, hash partitioning records across servers is often the norm since it is a simple, efficient, and scalable way of distributing load across a cluster of machines. Most systems tend to pre-partition records and tables into coarse hash buckets, and then move these buckets around the cluster in response to load imbalances [26]. However, coarse pre-partitioning can lead to high request fan-out when applications exhibit temporal locality in the records they access, hurting performance and cluster utilization [58]. Therefore, in order to be able to support a diverse set of applications with different access patterns, these systems need to be more flexible and lazy about how they partition and distribute data.

Flexible and lazy partitioning creates a unique challenge for kernel-bypass storage systems. Once a decision to partition is made, the partition must be quickly moved to it's new home with minimum impact to performance. Doing so is hard; these systems offer latencies as low as 5 µs, so even a few cache misses will significantly hurt performance.

Preserve Performance at Scale: Several of these stores exploit many-core hardware to ingest and index events at high rates – 100 million operations (Mops) per second (s) per machine [52, 65, 68, 88]. However, they rely on application-specific hardware acceleration, making them impossible to deploy on today's cloud platforms. Furthermore, these systems only store data in DRAM, and they do not scale across machines; adding support to do so without cutting into normal-case performance is not straightforward. For example, many of them statically partition records across cores to eliminate cross-core synchronization. This optimizes normal-case performance, but it makes concurrent operations like migration and scale out impossible; transferring record data and ownership between machines and cores requires a stop-the-world approach due to these systems' lack of fine-grained synchronization.

Achieving this level of performance while fulfilling all of these requirements on commodity cloud platforms requires solving two key challenges simultaneously. First, workloads change over time and cloud VMs fail, so systems must tolerate failure and reconfiguration. Doing this without hurting normal-case performance at 100 Mops/s is hard, since even a single extra server-side cache miss to check key ownership or reconfiguration status would cut throughput by tens-of-millions of operations per second. Second, the high CPU cost of processing incoming network packets easily dominates in these workloads, especially since, historically, cloud networking stacks have not been designed for high data rates and high efficiency.

Expressive Data Model and Multi-tenancy: Since the end of Dennard scaling, disaggregation has become the norm in the datacenter. Applications are typically broken into a compute and storage tier separated by a high speed network, allowing each tier to be provisioned, managed, and scaled independently. However, this approach is beginning to reach its limits. Applications have evolved to become more data intensive than ever. In addition to good performance, they often require rich and complex data models such as social graphs, decision trees, vectors [66, 79] etc. Storage systems, on the other hand,

have become faster with the help of kernel-bypass [30, 83], but at the cost of their interface – typically simple point lookups and updates. As a result of using these simple interfaces to implement their data model, applications end up stalling on network round-trips to the storage tier. Since the actual lookup or update takes only a few microseconds at the storage server, these round-trips create a major bottleneck, hurting performance and utilization. Therefore, to fully leverage these fast storage systems, applications will have to reduce round-trips by pushing compute to them.

Pushing compute to these fast storage systems is not straightforward. To maximize utilization, these systems need to be shared by multiple tenants, but the cost for isolating tenants using conventional techniques is too high. Hardware isolation requires a context switch that takes approximately 1.5 microseconds on a modern processor [59]. This is roughly equal to the amount of time it takes to fully process an RPC at the storage server, meaning that conventional isolation can hurt throughput by a factor of 2.

1.1 Contributions

Low-latency stores adopt simple, stripped down designs that optimize for normal case performance, and in the process, trade off features that would make them more practical and cost effective at cloud scale. This thesis shows that this trade off is unnecessary. Carefully leveraging and extending new and existing abstractions for scheduling, data sharing, lock-freedom, and isolation will yield feature-rich systems that retain their primary performance benefits at cloud scale.

This thesis presents horizontal and vertical mechanisms for rapid low-impact reconfiguration, multi-tenancy and expressive data models that would help make low-latency storage systems more practical and efficient at cloud scale. It is structured into three key pieces:

1. **Fast Data Migration:** The first piece presents *Rocksteady* [58], a horizontal mechanism for rapid reconfiguration and elasticity. Rocksteady is a live migration technique for scale-out in-memory key-value stores. It balances three competing goals: it migrates data quickly, it minimizes response time impact, and it allows arbitrary, fine-grained splits. Rocksteady allows a key-value store to defer all repartitioning work until the

moment of migration, giving it precise and timely control for load balancing.

2. **Low Cost Coordination:** The second piece presents *Shadowfax* [57], a system with horizontal and vertical mechanisms that allow distributed key-value stores to transparently span DRAM, SSDs, and cloud blob storage while serving 130 Mops/s/VM over commodity Azure VMs using conventional Linux TCP. Beyond high single-VM performance, Shadowfax uses a unique approach to distributed reconfiguration that avoids any server-side key ownership checks or cross-core coordination both during normal operation and migration.

3. **Extensibility and Multi-Tenancy:** The final piece presents *Splinter* [59], a system that provides clients with a vertical mechanism to extend low-latency key-value stores by migrating (pushing) code to them. Splinter is designed for modern multi-tenant data centers; it allows mutually distrusting tenants to write their own fine-grained extensions and push them to the store at runtime. The core of Splinter's design relies on type- and memory-safe extension code to avoid conventional hardware isolation costs. This still allows for bare-metal execution, avoids data copying across trust boundaries, and makes granular storage functions that perform less than a microsecond of compute practical.

1.2 Fast Data Migration

Rocksteady is a live migration technique for scale-out in-memory key-value stores. Built on top of RAMCloud [83], Rocksteady's key insight is to leverage application skew to speed up data migration while minimizing the impact to performance. When migrating a partition from a source to a target, it first migrates ownership of the partition. Doing so moves load on the partition from the source to the target, creating headroom on the source that can be used to migrate data. To keep the partition online, the target pulls records from the source on-demand; since applications are skewed – most requests are for a small set of hot records – this on-demand process converges quickly.

To fully utilize created headroom, Rocksteady carefully schedules and pipelines data migration on both the source and target. Migration is broken up into tasks that work in parallel over RAMCloud's hash table; doing so keeps the pre-fetcher happy, improving

cache locality. A shared-memory model allows these tasks to be scheduled on any core, allowing any idle compute on the source and target to be used for migration. To further speed up migration, Rocksteady delays re-replication of migrated data at the target to until after migration has completed. Fault tolerance is guaranteed by maintaining a dependency between the source and target at RAMCloud's coordinator (called lineage) during the migration, and recovering all data at the source if either machine crashes. Recovery must also include the target because of early ownership transfer; the target could have served and replicated writes on the partition since the migration began. Putting all these parts together results in a protocol that migrates data 100x faster than the state-of-the-art while maintaining tail latencies 1000x lower.

Overall, Rocksteady's careful attention to ownership, scheduling, and fault tolerance allow it to quickly and safely migrate data with low impact to performance. Experiments show that it can migrate at 758 MBps while maintaining tail latency below 250 microseconds; this is equivalent to migrating 256 GB of data in 6 minutes, allowing for quick scale-up and scale-down of a cluster. Additionally, early ownership transfer and lineage help improve migration speed by 25%. These results have important implications on system design; fast storage systems can use Rocksteady as a mechanism to enable flexible, lazy partitioning of data.

1.3 Low Cost Coordination

Shadowfax allows distributed key-value store to transparently span DRAM, SSDs, and cloud blob storage. Its unique approach to distributed reconfiguration avoids any cross-core coordination during normal operation and data migration both in its indexing and network interactions. In contrast to totally-ordered or stop-the-world approaches used by most systems, cores in Shadowfax avoid stalling to synchronize with one another, even when triggering complex operations like scale-out, which require defining clear before/after points in time among concurrent operations. Instead, each core participating in these operations – both at clients and servers – independently decides a point in an *asynchronous global cut* that defines a boundary between operation sequences in these complex operations. Shadowfax vertically extends asynchronous cuts from cores within one process [16] to servers and clients in a cluster. This helps coordinate server and client

threads in Shadowfax's low-coordination data migration and reconfiguration protocol.

In addition to reconfiguration, Shadowfax has mechanisms that help it achieve high throughput of 130 Mops/s/VM over commodity Azure VMs [19]. First, all requests from a client on one machine to Shadowfax are asynchronous with respect to one another all the way throughout Shadowfax's client- and server-side network submission/completion paths and servers' indexing and (SSD and cloud storage) I/O paths. This avoids all client- and server-side stalls due to head-of-line blocking, ensuring that clients can always continue to generate requests and servers can always continue to process them. Second, instead of partitioning data among cores to avoid synchronization on record accesses [51, 68, 98, 105], Shadowfax partitions network sessions across cores; its lock-free hash index and log-structured record heap are shared among all cores. This risks contention when some records are hot and frequently mutated, but this is more than offset by the fact that no software-level inter-core request forwarding or routing is needed within server VMs.

Measurements show that Shadowfax can shift load in 17 s to improve system throughput by 10 Mops/s with little disruption. Compared to the state-of-the-art, it has 8× better throughput (than Seastar+memcached) and scales out 6× faster. When scaled to a small cluster, Shadowfax retains its high throughput to perform 400 Mops/s, which, to the best of our knowledge, is the highest reported throughput for a distributed key-value store used for large-scale data ingestion and indexing.

1.4 Extensibility and Multi-Tenancy

Splinter provides clients with a vertical mechanism to extend low-latency key-value stores by migrating (pushing) code to them. Splinter relies on a type- and memory-safe language for isolation. Tenants push extensions – a tree traversal for example – written in the Rust programming language [95] to the system at runtime. Splinter installs these extensions. Once installed, an extension can be remotely invoked (executed) by the tenant in a single round-trip. For applications such as tree traversals which would ordinarily require round-trips logarithmic in the size of the tree, splinter can significantly improve both throughput and latency.

In addition to lightweight isolation, splinter consists of multiple mechanisms to make pushing compute feasible. Cross-core synchronization is minimized by maintaining *tenant*

locality; tenant requests are routed to preferred cores at the NIC [43] itself, and cores steal work from their neighbour to combat any resulting load imbalances. Pushed code (an extension) is scheduled *cooperatively*; extensions are expected to yield down to the storage layer frequently ensuring that long running extensions do not starve out short running ones. This approach is preferred over conventional multitasking using kthreads because preempting a kthread requires a context switch, making it too expensive for microsecond timescales. Uncooperative extensions are identified and dealt with by a dedicated watchdog core. Data copies are minimized by passing immutable references to extensions; the rust compiler statically verifies the lifetime and safety of these references. With the help of these mechanisms, Splinter can isolate 100's of granular tenant extensions per core while serving millions of operations per second with microsecond latencies.

Overall, Splinter adds extensibility to fast kernel-bypass storage systems, making it easier for applications to use them. An 800 line Splinter extension implementing Facebook's TAO graph model [11] can serve 2.8 million ops/s on 8 threads with an average latency of 30 microseconds. A significant fraction of TAO operations involve only a single round-trip. Implementing these on the client using normal lookups and implementing the remaining operations using the extension helps improve performance to 3.2 million ops/s at the same latency. This means that an approach that combines normal lookups/updates with Splinter's extensions is the best for performance; the normal lookups do not incur isolation overhead (no matter how low), and the extensions reduce the number of round-trips. In comparison, FaRM's [29] implementation of TAO performs 6.3 million ops/s on 32 threads with an average latency of 41 microseconds. This makes Splinter's approach, which performs 0.4 million ops/s per thread, competitive with FaRM's RDMA based approach, which performs 0.2 million ops/s per thread.

CHAPTER 2

FAST DATA MIGRATION

2.1 Introduction

The last decade of computer systems research has yielded efficient scale-out in-memory stores with throughput and access latency thousands of times better than conventional stores. Today, even modest clusters of these machines can execute billions of operations per second with access times of 6 µs or less [29, 83]. These gains come from careful attention to detail in request processing, so these systems often start with simple and stripped-down designs to achieve performance goals. For these systems to be practical in the long-term, they must evolve to include many of the features that conventional data center and cloud storage systems have *while* preserving their performance benefits.

To that end, we present *Rocksteady*, a fast migration and reconfiguration system for the RAMCloud scale-out in-memory store. Rocksteady facilitates cluster scale-up, scale-down, and load rebalancing with a low-overhead and flexible approach that allows data to be migrated at arbitrarily fine-grained boundaries and does not require any normal-case work to partition records. Our measurements show that Rocksteady can improve the efficiency of clustered accesses and index operations by more than 4×: operations that are common in many real-world large-scale systems [22, 79]. Several works address the general problem of online (or *live*) data migration for scale-out stores [6, 22, 23, 26, 31, 32, 109], but hardware trends and the specialized needs of an in-memory key value store make Rocksteady's approach unique:

Low-latency Access Times. RAMCloud services requests in 6 µs, and predictable, low-latency operation is its primary benefit. Rocksteady's focus is on 99.9th-percentile response times but with 1,000× lower response times than other tail latency focused

systems [26]. For clients with high fan-out requests, even a millisecond of extra tail latency would destroy client-observed performance. Migration must have minimum impact on access latency distributions.

Growing DRAM Storage. Off-the-shelf data center machines pack 256 to 512 GB per server with terabytes coming soon. Migration speeds must grow along with DRAM capacity for load balancing and reconfiguration to be practical. Today's migration techniques would take hours just to move a fraction of a single machine's data, making them ineffective for scale-up and scale-down of clusters.

High Bandwidth Networking. Today, fast in-memory stores are equipped with 40 Gbps networks with 200 Gbps [74] arriving in 2017. Ideally, with data in memory, these systems would be able to migrate data at full line rate, but there are many challenges to doing so. For example, we find that these network cards (NICs) struggle with the scattered, fine-grained objects common in in-memory stores (§2.3.2). Even with the simplest migration techniques, moving data at line rate would severely degrade normal-case request processing.

In short, the faster and less disruptive we can make migration, the more often we can afford to use it, making it easier to exploit locality and scaling for efficiency gains.

Besides hardware, three aspects of RAMCloud's design affect Rocksteady's approach; it is a high-availability system, it is focused on low-latency operation, and its servers internally (re-)arrange data to optimize memory utilization and garbage collection. This leads to the following three design goals for Rocksteady:

Pauseless. RAMCloud must be available at all times [81], so Rocksteady can never take tables offline for migration.

Lazy Partitioning. For load balancing, servers in most systems internally pre-partition data to minimize overhead at migration time [26, 29]. Rocksteady rejects this approach for two reasons. First, deferring all partitioning until migration time lets Rocksteady make partitioning decisions with full information at hand; it is never constrained by a set of pre-defined splits. Second, DRAM-based storage is expensive; during normal operation, RAMCloud's log cleaner [94] constantly reorganizes data

physically in memory to improve utilization and to minimize cleaning costs. Forcing a partitioning on internal server state would harm the cleaner's efficiency, which is key to making RAMCloud cost-effective.

Low Impact With Minimum Headroom. Migration increases load on source and target servers. This is particularly problematic for the source, since data may be migrated away to cope with increasing load. Efficient use of hardware resources is critical during migration; preserving headroom for rebalancing directly increases the cost of the system.

Four key ideas allow Rocksteady to meet these goals:

Adaptive Parallel Replay. For servers to keep up with fast networks during migration, Rocksteady fully pipelines and parallelizes all phases of migration between the source and target servers. For example, target servers spread incoming data across idle cores to speed up index reconstruction, but migration operations yield to client requests for data to minimize disruption.

Exploit Workload Skew to Create Source-side Headroom. Rocksteady prioritizes migration of hot records. For typical skewed workloads, this quickly shifts some load with minimal impact, which creates headroom on the source to allow faster migration with less disruption.

Lineage-based Fault Tolerance. Each RAMCloud server logs updated records in a distributed, striped log which is also kept (once) in-memory to service requests. A server does not know how its contents will be partitioned during a migration, so records are intermixed in memory and on storage. This complicates fault tolerance during migration: it is expensive to synchronously reorganize on-disk data to move records from the scattered chunks of one server's log into the scattered chunks of another's. Rocksteady takes inspiration from Resilient Distributed Datasets [119]; servers can take dependencies on portions of each others' recovery logs, allowing them to safely reorganize storage asynchronously.

Optimization for Modern NICs. Fast migration with tight tail latency bounds requires careful attention to hardware at every point in the design; any "hiccup" or extra

load results in latency spikes. Rocksteady uses kernel-bypass for low overhead migration of records; the result is fast transfer with reduced CPU load, reduced memory bandwidth load, and more stable normal-case performance.

We start by motivating Rocksteady (§4.2) and quantifying the gains it can achieve. Then, we show why state of the art migration techniques are insufficient for RAMCloud including a breakdown of why RAMCloud's simple, pre-existing migration is inadequate (§2.2.3). We describe Rocksteady's full design (§2.3) including its fault tolerance strategy, and we evaluate its performance with emphasis on migration speed and tail latency impact (§2.4). Compared to prior approaches, Rocksteady transfers data an order of magnitude faster ($>$ 750 MB/s) with median and tail latencies 1,000$\times$ lower ($<$ 40 µs and 250 µs, respectively); in general, Rocksteady's ability to use *any* available core for *any* operation is key for both tail latency and migration speed.

2.2 Background and Motivation

RAMCloud [83] is a key-value store that keeps all data in DRAM at all times and is designed to scale across thousands of commodity data center servers. Each server can service millions of operations per second, but its focus is on low access latency. End-to-end read and durable write operations take just 6 µs and 15 µs respectively on our hardware (§2.4).

Each server (Figure 2.1) operates as a *master*, which manages RAMCloud objects in its DRAM and services client requests, and a *backup*, which stores redundant copies of objects from other masters on local disk. Each cluster has one quorum-replicated *coordinator* that manages cluster membership and table-partition-to-master mappings [80].

RAMCloud only keeps one copy of each object in memory to avoid replication in expensive DRAM; redundant copies are logged to (remote) flash. It provides high availability with a fast distributed recovery that sprays the objects previously hosted on a failed server across the cluster in 1 to 2 seconds [81], restoring access to them. RAMCloud manages in-memory storage using an approach similar to that of log-structured filesystems, which allows it to sustain 80-90% memory utilization with high performance [94].

RAMCloud's design and data model tightly intertwine with load balancing and

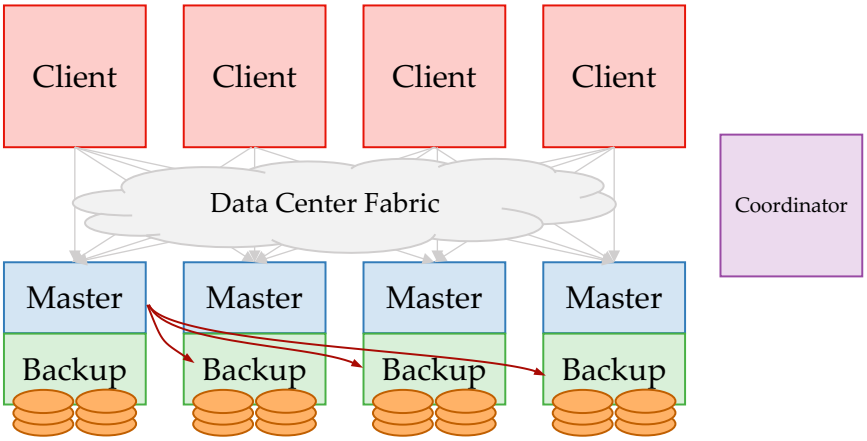


Figure 2.1: The RAMCloud architecture. Clients issue remote operations to RAMCloud storage servers. Each server contains a master and a backup. The master component exports the DRAM of the server as a large key-value store. The backup accepts updates from other masters and records state on disk used for recovering crashed masters. A central coordinator manages the server pool and maps data to masters.

migration. Foremost, RAMCloud is a simple variable-length key-value store; its key space is divided into unordered tables and tables can be broken into *tablets* that reside on different servers. Objects can be accessed by their primary (byte string) key, but ordered secondary indexes can also be constructed on top of tables [53]. Like tables, secondary indexes can be split into *indexlets* to scale them across servers. Indexes contain primary key hashes rather than records, so tables and their indexes can be scaled independently and needn't be co-located (Figure 2.2). Clients can issue multi-read and multi-write requests that fetch or modify several objects on one server with a single request, and they can also issue externally consistent and serializable distributed transactions [62].

2.2.1 Why Load Balance?

All scale-out stores need some way to distribute load. Most systems today use some form of consistent hashing [26, 103, 110]. Consistent hashing is simple, keeps the key-to-server mapping compact, supports reconfiguration, and spreads load fairly well even as workloads change. However, its load spreading is also its drawback; even in-memory stores benefit significantly from exploiting access locality.

In RAMCloud, for example, co-locating access-correlated keys benefits multiget/multiput operations, transactions, and range queries. Transactions can benefit greatly if all affected keys can be co-located, since synchronous, remote coordination for

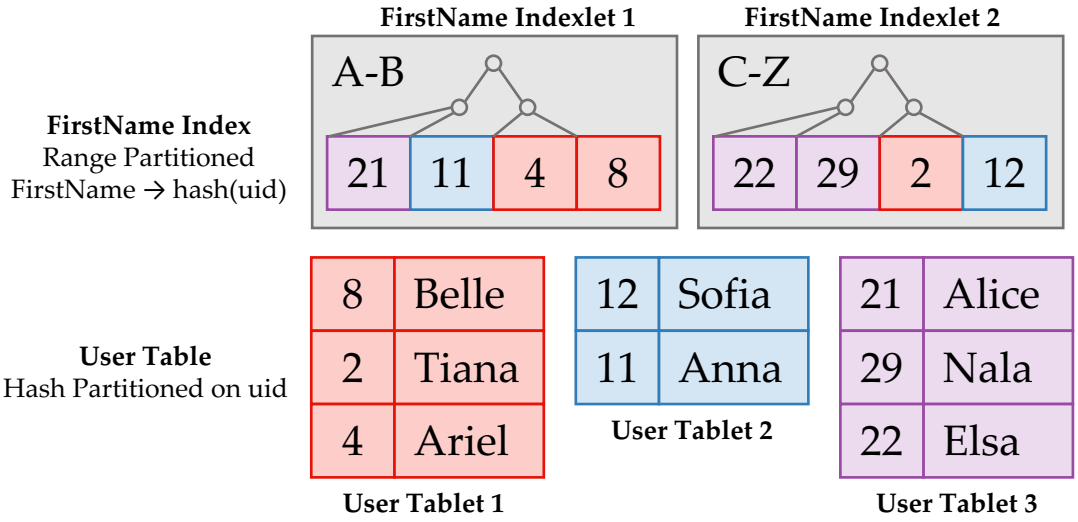


Figure 2.2: Index partitioning. Records are stored in unordered tables that can be split into tablets on different servers, partitioned on primary key hash. Indexes can be range partitioned into indexlets; indexes only contain primary key hashes. Range scans require first fetching a list of hashes from an indexlet, then multigets for those hashes to the tablet servers to fetch the actual records. A lookup or scan operation is (usually) handled by one server, but tables and their indexes can be split and scaled independently.

two-phase commit [2, 62] can be avoided. Multi-operations and range queries benefit in a more subtle but still important way. If all requested values live together on the same machine, a client can issue a single remote procedure call (RPC) to a single server to get the values. If the values are divided among multiple machines, the client must issue requests to all of the involved machines in parallel. From the client's perspective, the response latency is similar, but the overall load induced on the cluster is significantly different.

Figure 2.3 explores this effect. It consists of a microbenchmark run with 7 servers and 14 client machines. Clients issue back-to-back multiget operations evenly across the cluster, each for 7 keys at a time. In the experiment, clients vary which keys they request in each multiget to vary how many servers they must issue parallel requests to, but all servers still handle the same number of requests as each other. At Spread 1, all of the keys for a specific multiget come from one server. At Spread 2, 6 keys per multiget come from one server, and the 7^{th} key comes from another server. At Spread 7, each of the 7 keys in the multiget is serviced by a different server.

When multigets involve two servers rather than one, the cluster-wide throughput drops 23% even though no resources have been removed from the cluster. The bottom

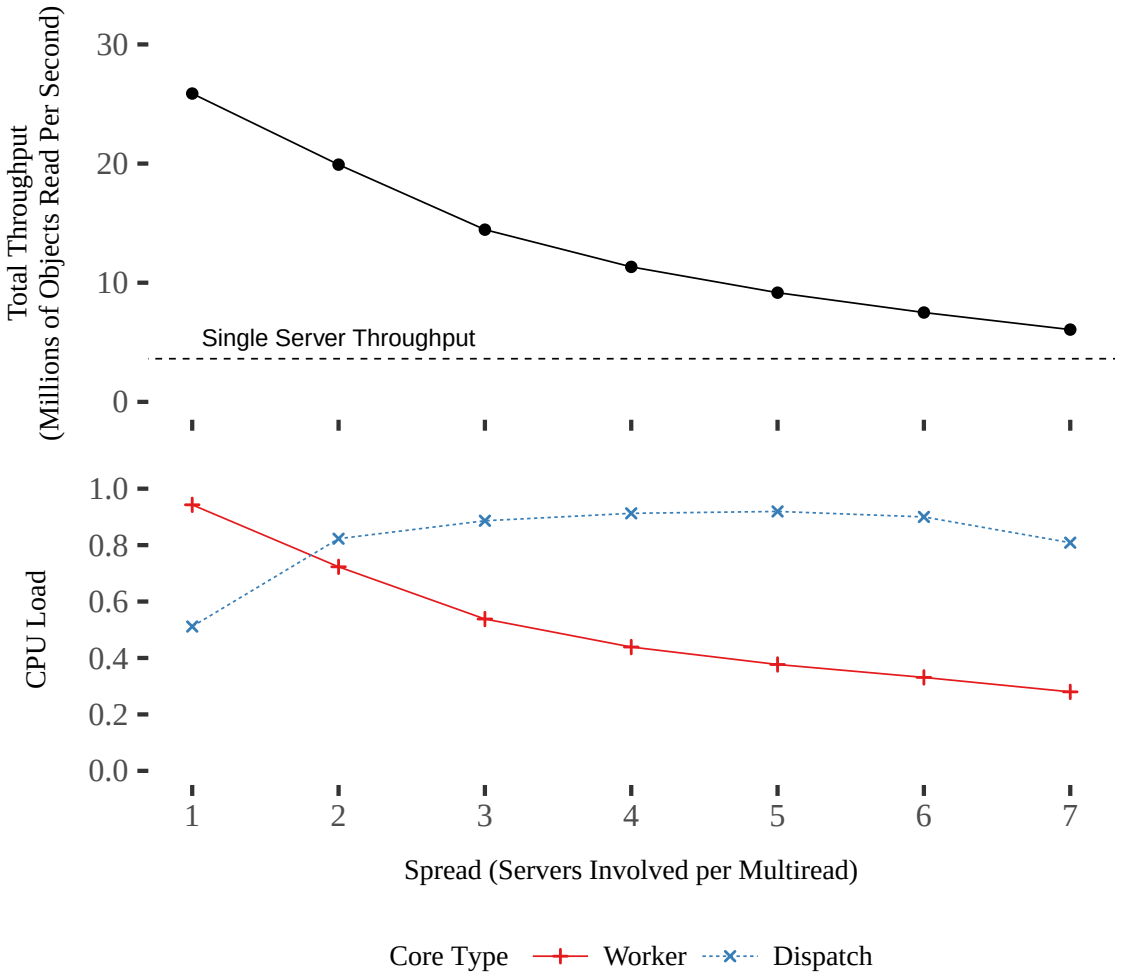


Figure 2.3: Throughput and CPU load impact of access locality. When multigets always fetch data from a single server (Spread 1) throughput is high and worker cores operate in parallel. When each multiget must fetch keys from many machines (Spread 7) throughput suffers as each server becomes bottlenecked on dispatching requests.

half of the figure shows the reason. Each server has a single *dispatch* core that polls the network device for incoming messages and hands off requests to idle *worker* cores. With high locality, the cluster is only limited by how quickly worker cores can execute requests. When each multiget results in requests to two servers, the dispatch core load doubles and saturates, leaving the workers idle. The dotted line shows the throughput of a single server. When each multiget must fetch data from all 7 servers, the aggregate performance of the entire cluster barely outperforms a single machine.

Overall, the experiment shows that, even for small clusters, minimizing tablet splits and maximizing locality has a big benefit, in this case up to 4.3×. Our findings echo recent

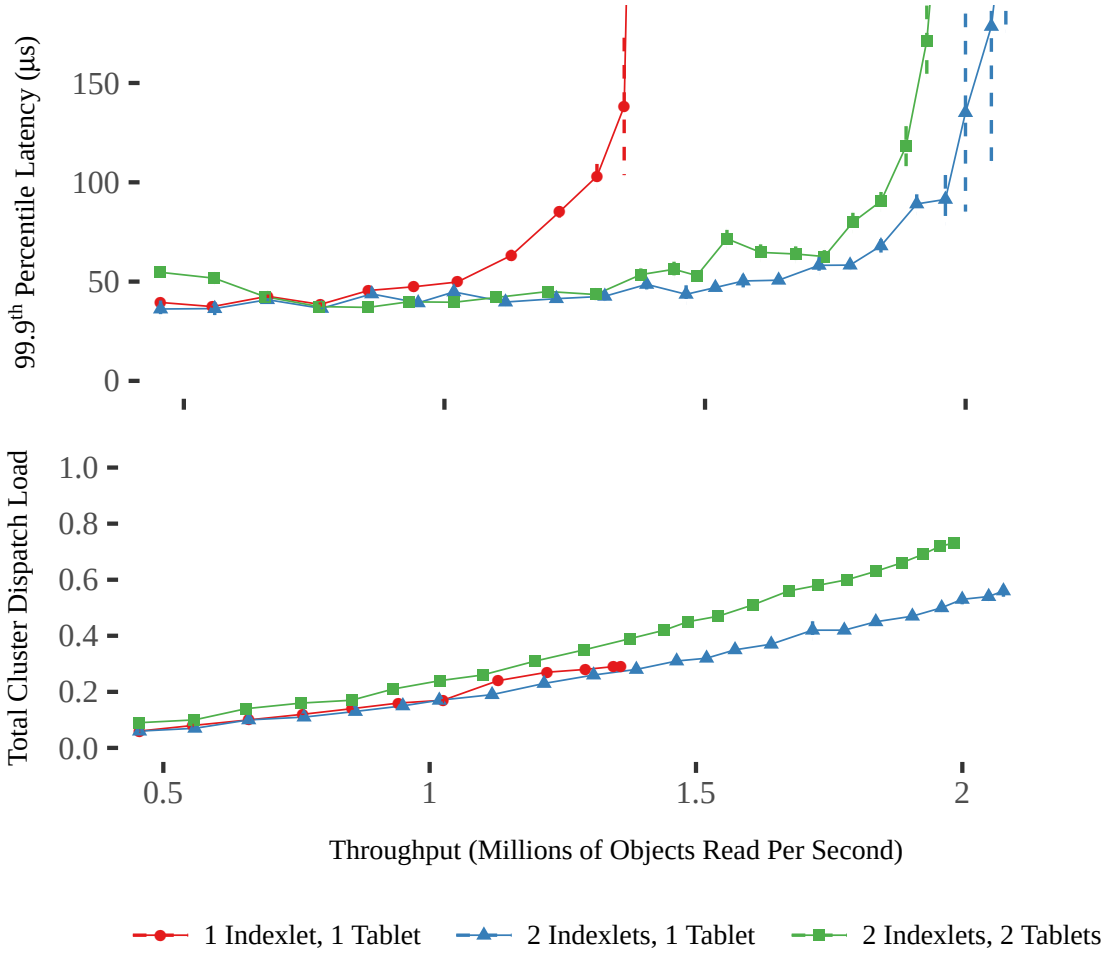


Figure 2.4: Index scaling as a function of read throughput. Points represent the median over 5 runs, bars show standard error. Spreading the backing table across two servers increases total dispatch load and the 99.9th percentile access latency for a given throughput when compared to leaving it on a single server.

trends in scale-out stores that replicate to minimize multiget "fan out" [79] or give users explicit control over data placement to exploit locality [22].

Imbalance has a similar effect on another common case: indexes. Index ranges are especially prone to hotspots, skew shifts, and load increases that require splits and migration. Figure 2.4 explores this sensitivity on a cluster with a single table and secondary index. The table contains one million 100 B records each with a 30 B primary and a 30 B secondary key. Clients issue short 4-record scans over the index with the start key chosen from a Zipfian distribution with skew θ = 0.5. Figure 2.4 shows the impact of varying offered client load on the 99.9th percentile scan latency.

For a target throughput of 1 million objects per second, it is sufficient (for a 99.9th percentile access latency of 100 µs) and most efficient (dispatch load is lower) to have the index and table on one server each, but this breaks down as load increases. At higher loads, 99.9th percentile latency spikes and more servers are needed to bound tail latency. Splitting the index over two servers improves throughput and restores low access latency.

However, efficiently spreading the load is not straightforward. Indexes are range partitioned, so any single scan operation is likely to return hashes using a single indexlet. Tables are hash partitioned, so fetching the actual records will likely result in an RPC to many backing tablets. As a result, adding tablets for a table might increase throughput, but it also increases dispatch core load since, cluster-wide, it requires more RPCs for the same work.

Figure 2.4 shows that neither minimizing nor maximizing the number of servers for the indexed table is the best under high load. Leaving the backing table on one server and spreading the index over two servers increases throughput at 100 µs 99.9th percentile access latency by 54% from 1.3 to 2.0 million objects per second. Splitting both the backing table and the index over two servers each gives 6.3% worse throughput *and* increases load by 26%.

Overall, reconfiguration is essential to meet SLAs (service level agreements), to provide peak throughput, and to minimize load as workloads grow, shrink, and change. Spreading load evenly is a non-goal inasmuch as SLAs are met; approaches like consistent hashing can throw away (sometimes large factor) gains from exploiting locality.

2.2.2 The Need for (Migration) Speed

Data migration speed dictates how fast a cluster can adapt to changing workloads. Even if workload shifts are known in advance (like diurnal patterns), if reconfiguration takes hours, then scaling up and down to save energy or to do other work becomes impossible. Making things harder, recent per-server DRAM capacity growth has been about 33-50% per year [39], meaning each server hosts more and more data that may need to move when the cluster is reconfigured.

A second hardware trend is encouraging; per-host network bandwidth has kept up with DRAM growth in recent years [42], so hardware itself doesn't limit fast migration. For

example, an unrealistically large migration that evacuates half of the data from a 512 GB storage server could complete in less than a minute at line rate (5 GB/s or more).

Unfortunately, state-of-the-art migration techniques haven't kept up with network improvements. They move data at a few megabytes per second in order to minimize impact on ongoing transactions, and they focus on preserving transaction latencies on the order of tens of milliseconds [31]. Ignoring latency, these systems would still take more than 16 hours to migrate 256 GB of data, and small migrations of 10 GB would still take more than half an hour. Furthermore, modern in-memory systems deliver access latencies more than 1,000× lower: in the range of 5 to 50 µs for small accesses, transactions, or secondary index lookups/scans. If the network isn't a bottleneck for migration, then what is?

2.2.3 Barriers to Fast Migration

RAMCloud has a simple, pre-existing mechanism that allows tables to be split and migrated between servers. During normal operation each server stores all records in an in-memory log. The log is incrementally cleaned; it is never checkpointed, and a full copy of it always remains in memory. To migrate a tablet, the source iterates over all of the entries in its in-memory log and copies the values that are being migrated into staging buffers for transmission to the target. The target receives these buffers and performs a form of logical replay as it would during recovery. It copies the received records into its own log, re-replicates them, and it updates its in-memory hash table, which serves as its primary key index. Only after all of the records have been transferred is tablet ownership switched from the source to the target.

This basic mechanism is faster than most approaches, but it is still orders of magnitude slower than what hardware can support. Figure 2.5 breaks down its bottlenecks. The experiment shows the effective migration throughput between a single loaded source and unloaded target server during the migration of 7 GB of data. All of the servers are interconnected via 40 Gbps (5 GB/s) links to a single switch.

The "Full" line shows migration speed when the whole migration protocol is used. The source scans its log and sends records that need to be migrated; the target replays the received records into its log and re-replicates them on backups. In steady state, migration

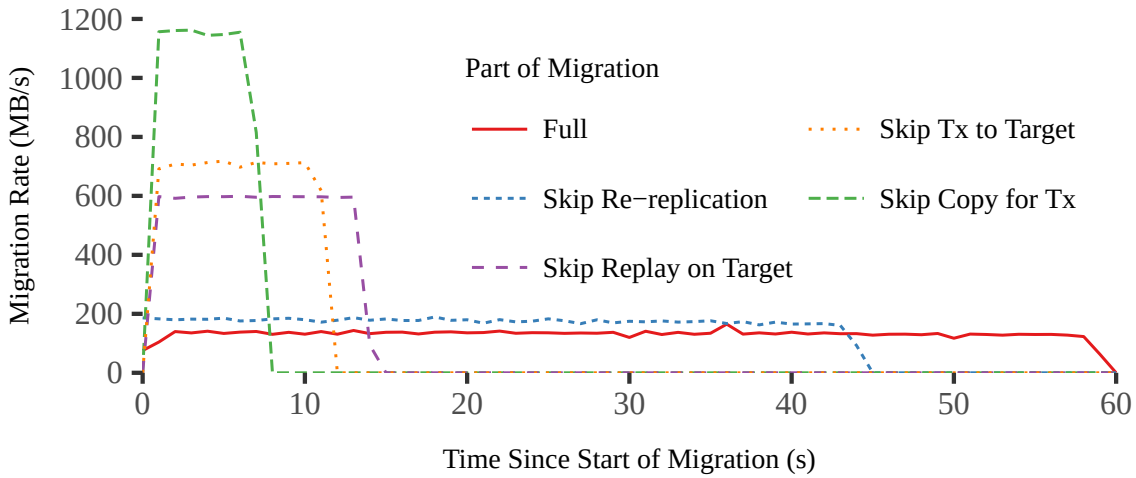


Figure 2.5: Bottlenecks using log replay for migration. Target side bottlenecks include logical replay and re-replication. Copying records into staging buffers at the source has a significant impact on migration rate.

transfers about 130 MB/s.

In "Skip Re-Replication" the target skips backing up the received data in its replicated log. This is unsafe, since the target might accept updates to the table after it has received all data from the source. If the target crashes, its recovery log would be missing the data received from the source, so the received table would be recovered to an inconsistent state. Even so, migration only reaches 180 MB/s. This shows that the logical replay used to update the hash table on the target is a key bottleneck in migration.

"Skip Replay on Target" does the full source-side processing of migration and transmits the data to the target, but the target skips replay and replication. This raises migration performance to 600 MB/s, more than a 3× increase in migration rate. Even so, it shows that the source side is also an impediment to fast migration. The hosts can communicate at 5 GB/s, so the link is still only about 10% utilized. Also, at this speed re-replication becomes a problem; RAMCloud's existing log replication mechanism bottlenecks at around 380 MB/s on our cluster.

Finally, "Skip Tx to Target" performs all source-side processing and skips transmitting the data to the target, and "Skip Copy for Tx" only identifies objects that need to be migrated and skips all further work. Overall, copying the identified objects into staging buffers to be posted to the transport layer (drop from 1,150 MB/s to 710 MB/s) has a bigger impact than the actual transmission itself (drop from 710 MB/s to 600 MB/s).

2.2.4 Requirements for a New Design

These bottlenecks give the design criteria for Rocksteady.

No Synchronous Re-replication. Waiting for data to be re-replicated by the target server wastes CPU cycles on the target waiting for responses from backups, and it burns memory bandwidth. Rocksteady's approach is inspired by lineage [119]; a target server takes a temporary dependency on the source's log data to safely eliminate log replication from the migration fast path (§2.3.4).

Immediate Transfer of Ownership. RAMCloud's migration takes minutes or hours during which no load can be shifted away from the source because the target cannot safely take ownership until all the data has been re-replicated. Rocksteady immediately and safely shifts ownership from the source to the target (§2.3).

Parallelism on Both the Target and Source. Log replay needn't be single threaded. A target is likely to be under-loaded, so parallel replay makes sense. Rocksteady's parallel replay can incorporate log records at the target at more than 3 GB/s (§2.4.5). Similarly, source-side migration operations should be pipelined and parallel. Parallelism on both ends requires care to avoid contention.

Load-Adaptive Replay. Rocksteady's migration manager minimizes impact on normal request processing with fine-grained low-priority tasks [63, 84]. Rocksteady also incorporates into RAMCloud's transport layer to minimize jitter caused by background migration transfers (§2.3.1).

2.3 Rocksteady Design

In order to keep its goal of fast migration that retains 99.9^{th} percentile access latencies of a few hundred microseconds, Rocksteady is fully asynchronous at both the migration source and target; it uses modern kernel-bypass and scatter/gather DMA for zero-copy data transfer when supported; and it uses pipelining and adaptive parallelism at both the source and target to speed transfer while yielding to normal-case request processing.

Migration in Rocksteady is driven by the target, which pulls records from the source. This places most of the complexity, work, and state on the target, and it eliminates the bottleneck of synchronous replication (§2.2.3). In most migration scenarios, the source of the records is in a state of overload or near-overload, so we must avoid giving it more work

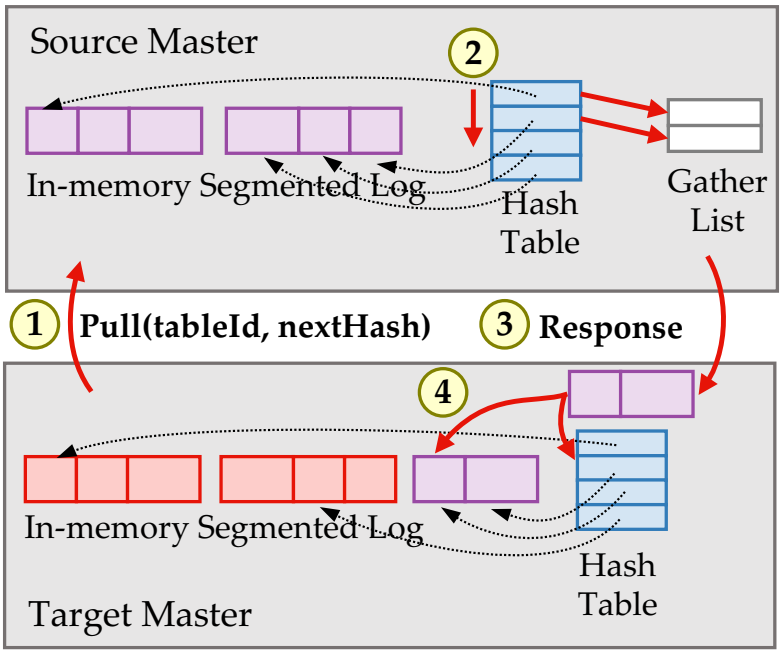


Figure 2.6: Overview of Rocksteady `Pulls`. A `Pull` RPC issued by the target iterates down a portion of the source's hash table and returns a batch of records. This batch is then logically replayed by the target into it's in-memory log and hash table.

to do. The second advantage of this arrangement is that it meets our goal of immediate transfer of record ownership. As soon as migration begins, the source only serves a request for each of the affected records at most once more. This makes the load-shedding effects of migration immediate. Finally, target-driven migration allows both the source and the target to control the migration rate, fitting with our need for load-adaptive migration and making sure that cores are never idle unless migration must be throttled to meet SLAs.

The heart of Rocksteady's fast migration is its pipelined and parallelized record transfer. Figure 2.6 gives an overview of this transfer. In the steady state of migration, the target sends pipelined asynchronous `Pull` RPCs to the source to fetch batches of records (①). The source iterates down its hash table to find records for transmission (②); it posts the record addresses to the transport layer, which transmits the records directly from the source log via DMA if the underlying hardware supports it (③). Whenever cores are available, the target schedules the replay of the records from any `Pulls` that have completed. The replay process incorporates the records into the target's in-memory log and links the records into the target's hash table (④).

Migration is initiated by a client: it does so by first splitting a tablet, then issuing a

`MigrateTablet` RPC to the target to start migration. Rocksteady immediately transfers ownership of the tablet's records to the target, which begins handling all requests for them. Writes can be serviced immediately; reads can be serviced only after the records requested have been migrated from the source. If the target receives a request for a record that it does not yet have, the target issues a `PriorityPull` RPC to the source to fetch it and tells the client to retry the operation after randomly waiting a few tens of microseconds. `PriorityPull` responses are processed identically to `Pulls`, but they fetch specific records and the source and target prioritize them over bulk `Pulls`.

This approach to `PriorityPulls` favors immediate load reduction at the source. It is especially effective if access patterns are skewed, since a small set of records constitutes much of the load: in this case, the source sends one copy of the "hot" records to the target early in the migration, then it does not need to serve any more requests for those records. In fact, `PriorityPulls` can actually accelerate migration. At the start of migration, they help to quickly create the headroom needed on the overloaded source to speed parallel background `Pulls` and help hide `Pull` costs.

Sources keep no migration state, and their migrating tablets are immutable. All the source needs to keep track of is the fact that the tablet is being migrated: if it receives a client request for a record that is in a migrating tablet, it returns a status indicating that it no longer owns the tablet, causing the client to re-fetch the tablet mapping from the coordinator.

2.3.1 Task Scheduling, Parallelism, and QoS

The goal of scheduling within Rocksteady is to keep cores on the target as busy as possible without overloading cores on the source, where overload would result in SLA violations.

To understand Rocksteady's approach to parallelism and pipelining, it is important to understand scheduling in RAMCloud. RAMCloud uses a threading model that avoids preemption: in order to dispatch requests within a few microseconds, it cannot afford the disruption of context switches [83]. One core handles dispatch; it polls the network for messages, and it assigns tasks to worker cores or queues them if no workers are idle. Each core runs one thread, and running tasks are never preempted (which would require

a context-switch mechanism). Priorities are handled in the following fashion: if there is an available idle worker core when a task arrives, the task is run immediately. If no cores are available, the task is placed in a queue corresponding to its priority. When a worker becomes available, if there are any queued tasks, it is assigned a task from the front of the highest-priority queue with any entries.

RAMCloud's dispatch/worker model gives four benefits for migration. First, migration blends in with background system tasks like garbage collection and (re-)replication. Second, Rocksteady can adapt to system load ensuring minimal disruption to normal request processing while migrating data as fast as possible. Third, since the source and target are decoupled, workers on the source can always be busy collecting data for migration, while workers on the target can always make progress by replaying earlier responses. Finally, Rocksteady makes no assumptions of locality or affinity; a migration related task can be dispatched to *any* worker, so any idle capacity on either end can be put to use.

2.3.1.1 Source-side Pipelined and Parallel Pulls

The source's only task during migration is to respond to `Pull` and `PriorityPull` messages with sufficient parallelism to keep the target busy. While concurrency would seem simple to handle, there is one challenge that complicates the design. A single `Pull` can't request a fixed range of keys, since the target does not know ahead of time how many keys within that range will exist in the tablet. A `Pull` of a fixed range of keys could contain too many records to return in a single response, which would violate the scheduling requirement for short tasks. Or, it could contain no records at all, which would result in `Pull`s that are pure overhead. `Pull` must be efficient regardless of whether the tablet is sparse or dense. One solution is for each `Pull` to return a fixed amount of data. The amount can be chosen to be small enough to avoid occupying source worker cores for long periods, but large enough to amortize the fixed cost of RPC dispatch.

However, this approach hurts concurrency: each new pull needs state recording which record was the last pulled, so that the pull can continue from where it left off. The target could remember the last key it received from the previous pull and use that as the starting point for the next pull, but this would prevent it from pipelining its pulls. It would have

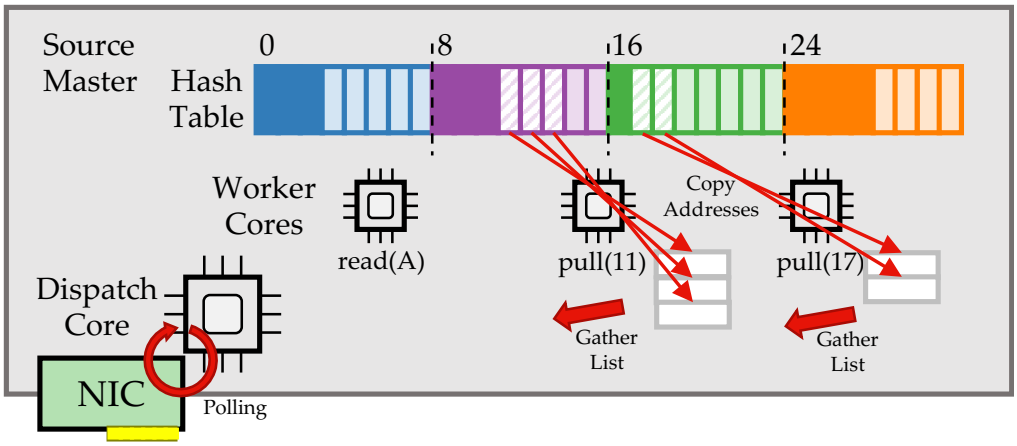


Figure 2.7: Source pull handling. `Pulls` work concurrently over disjoint regions of the source's hash table, avoiding synchronization, and return a fixed amount of data (20 KB, for example) to the target. Any worker core can service a `Pull` on any region, and all cores prioritize normal case requests over `Pulls`.

to wait for one to fully complete before it could issue the next, making network round trip latency into a major bottleneck. Alternately, the source could track the last key returned for each pull, but this has the same problem. Neither approach allows parallel `Pull` processing on the source, which is key for fast migration.

To solve this, the target logically *partitions* the source's key hash space and only issues concurrent `Pulls` if they are for disjoint regions of the source's key hash space (and, consequently, disjoint regions of the source's hash table). Figure 3.7 shows how this works. Since round-trip delay is similar to source pull processing time, a small constant factor more partitions than worker cores is sufficient for the target to keep any number of source workers running fully-utilized.

The source attempts to meet its SLA requirements by prioritizing regular client reads and writes over `Pull` processing: the source can essentially treat migration as a background task and prevent it from interfering with foreground tasks. It is worth noting that the source's foreground load typically drops immediately when migration starts, since Rocksteady has moved ownership of the (likely hot) migrating records to the target already; this leaves capacity on the source that is available for the background migration task. `PriorityPulls` are given priority over client traffic, since they represent the target servicing a client request of its own.

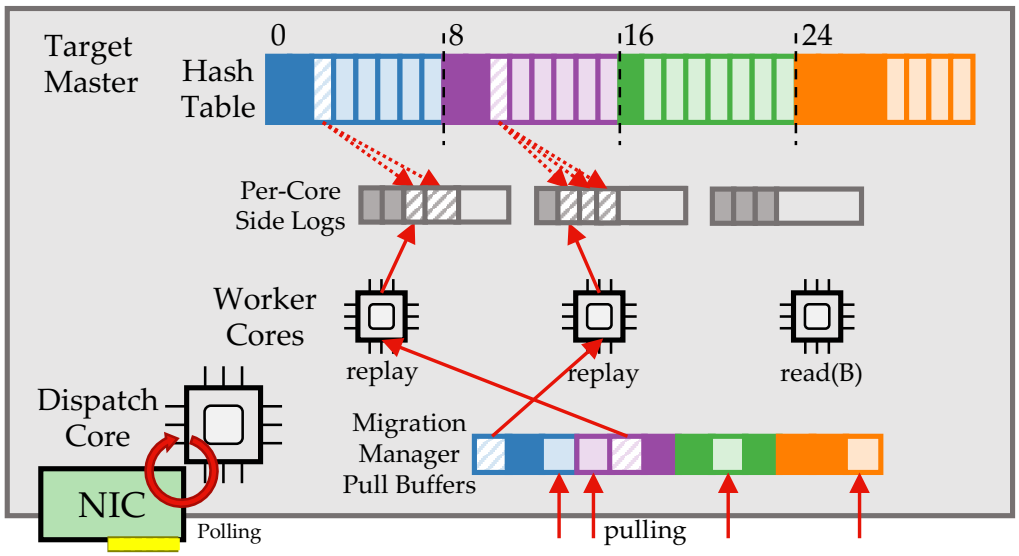


Figure 2.8: Target pull management and replay. There is one `Pull` outstanding per source partition. Pulled records are replayed at lower priority than normal requests. Each worker places records into a separate side log to avoid contention. Any worker core can service a replay on any partition.

2.3.1.2 Target-side Pull Management

Since the source is stateless, a *migration manager* at the target tracks all progress and coordinates the entire migration. The migration manager runs as an asynchronous continuation on the target's dispatch core [106]; it starts `Pulls`, checks for their completion, and enqueues tasks that replay (locally process) records for `Pulls` that have completed.

At migration start, the manager logically divides the source server's key hash space into partitions (§2.3.1.1). Then, it asynchronously issues `Pull` requests to the source, each belonging to a different partition of the source hash space. As `Pulls` complete, it pushes the records to idle workers, and it issues a new `Pull`. If all workers on the target are busy, then no new `Pull` is issued, which has the effect of acting as built-in flow control for the target node. In that case, new `Pulls` are issued when workers become free and begin to process records from already completed `Pulls`.

Records from completed pull requests are replayed in parallel into the target's hash table on idle worker cores. Pull requests from distinct partitions of the hash table naturally correspond to different partitions of the target's hash table as well, which mitigates contention between parallel replay tasks. Figure 3.8 shows how the migration manager "scoreboards" `Pull` RPCs from different hash table partitions and hands responses over to

idle worker cores. Depending on the target server's load, the manager naturally adapts the number of in-progress `Pull` RPCs, as well as the number of in-progress replay tasks.

Besides parallelizing `Pulls`, performing work at the granularity of distinct hash table partitions also hides network latency by allowing Rocksteady to pipeline RPCs. Whenever a `Pull` RPC completes, the migration manager first issues a new, asynchronous `Pull` RPC for the next chunk of records on the same partition; having a small number of independent partitions is sufficient to completely overlap network delay with source-side `Pull` processing.

2.3.1.3 Parallel Replay

Replaying a `Pull` response primarily consists of incorporating the records into the master's in-memory log and inserting references to the records in the master's hash table. Using a single core for replay would limit migration to a few hundred megabytes per second (§2.4.5), but parallel replay where cores share a common log would also break down due to contention. Eliminating contention is key for fast migration.

Rocksteady does this by using per-core *side logs* off of the target's main log. Each side log consists of independent *segments* of records; each core can replay records into its side log segments without interference. At the end of migration, each side log's segments are lazily replicated, and then the side log is *committed* into the main log by appending a small metadata record to the main log. RAMCloud's log cleaner needs accurate log statistics to be effective; side logs also avoid contention on statistics counters by accumulating information locally and only updating the global log statistics when they are committed to the main log.

2.3.2 Exploiting Modern NICs

All data transfer in Rocksteady takes place through RAMCloud's RPC layer allowing the protocol to be both transport and hardware agnostic. Target initiated one-sided RDMA reads may seem to promise fast transfers without the source's involvement, but they break down because the records under migration are scattered across the source's in-memory log. RDMA reads do support scatter/gather DMA, but reads can only fetch a single contiguous chunk of memory from the remote server. That is, a single RDMA read *scatters* the fetched value locally; it cannot *gather* multiple remote locations with a

single request. As a result, an RDMA read initiated by the target could only return a single data record per operation unless the source pre-aggregated all records for migration beforehand, which would undo the zero-copy benefits of RDMA. Additionally, one-sided RDMA would require the target to be aware of the structure and memory addresses of the source's log. This would complicate synchronization, for example, with RAMCloud's log cleaner. Epoch-based protection can help (normal-case RPC operations like read and write synchronize with the local log cleaner this way), but extending epoch protection across machines would couple the source and target more tightly.

Rocksteady never uses one-sided RDMA, but it uses scatter/gather DMA [83] when supported by the transport and the NIC to transfer records from the source without intervening copies. Rocksteady's implementation always operates on references to the records rather than making copies to avoid all unnecessary overhead.

All experiments in this paper were run with a DPDK driver that currently copies all data into transmit buffers. This creates one more copy of records than strictly necessary on the source. This limitation is not fundamental; we are in the process of changing RAMCloud's DPDK support to eliminate the copy. Rocksteady run on Reliable Connected Infiniband with zero-copy shows similar results. This is in large part because Intel's DDIO support means that the final DMA copy from Ethernet frame buffers is from the CPU cache [24]. Transition to zero-copy will reduce memory bandwidth consumption [54], but source-side memory bandwidth is not saturated during migration.

2.3.3 Priority Pulls

`PriorityPulls` work similarly to normal `Pulls` but are triggered on-demand by incoming client requests. A `PriorityPull` targets specific key hashes, so it doesn't require the coordination that `Pulls` do through partitioning. The key consideration for `PriorityPulls` is how to manage waiting clients and worker cores. A simple approach is for the target to issue a synchronous `PriorityPull` to the source when servicing a client `read` RPC for a key that hasn't been moved yet. However, this would slow migration and hurt client-observed latency and throughput. `PriorityPulls` take several microseconds to complete, so stalling a worker core on the target to wait for the response takes cores away from migration and normal request processing. Thread context switch also isn't an option

since the delay is just a few microseconds, and context switch overhead would dominate. Individual, synchronous `PriorityPulls` would also initially result in many (possibly duplicate) requests being forwarded to the source, delaying source load reduction.

Rocksteady solves this in two ways. First, the target issues `PriorityPulls` asynchronously and then immediately returns a response to the client telling it to retry the `read` after the time when the target expects it will have the value. This frees up the worker core at the target to process requests for other keys or to replay `Pull` responses. Second, the target *batches* the hashes of client-requested keys that have not yet arrived, and it requests the batch of records with a single `PriorityPull`. While a `PriorityPull` is in flight, the target accumulates new key hashes of newly requested keys, and it issues them when the first `PriorityPull` completes. De-duplication ensures that `PriorityPulls` never request the same key hash from the source twice. If the hash for a new request was part of an already in-flight `PriorityPull` or if it is in the next batch accumulating at the target, it is discarded. Batching is key to shedding source load quickly since it ensures that the source never serves a request for a key more than once after migration starts, and it limits the number of small requests that the source has to handle.

2.3.4 Lineage for Safe, Lazy Re-replication

Avoiding synchronous re-replication of migrated data creates a challenge for fault tolerance if tablet ownership is transferred to the target at the start of migration. If the target crashes in the middle of a migration, then neither the source nor the target would have all of the records needed to recover correctly; the target may have serviced writes for some of the records under migration, since ownership is transferred immediately at the start of migration. This also means that neither the distributed recovery log of the source nor the target contain all the information needed for a correct recovery. Rocksteady takes a unique approach to solving this problem that relies on RAMCloud's distributed fast recovery, which can restore a crashed server's records back into memory in 1 to 2 seconds.

To avoid synchronous re-replication of all of the records as they are transmitted from the source to the target, the migration manager registers a dependency of the source server on the tail of the target's recovery log at the cluster coordinator. The target must already contact the coordinator to notify it of the ownership transfer, so this adds no additional

CPU	2×Xeon E5-2650v2 2.6 GHz, 16 cores in total after disabling hyperthreading
RAM	64 GB 1.86 GHz DDR3
NIC	Mellanox FDR CX3 Single port (40 Gbps)
Switch	36 port Mellanox SX6036G (in Ethernet mode)
OS	Ubuntu 15.04, Linux 3.19.0-16, DPDK 16.11, MLX4 PMD, 1×1 GB Hugepage

Table 2.1: Experimental cluster configuration. The evaluation was carried out on a 24 node c6220 cluster on CloudLab. Hyperthreading was disabled on all nodes. Of the 24 nodes, 1 ran the coordinator, 8 ran one client each, and the rest ran RAMCloud servers.

overhead. The dependency is recorded in the coordinator's tablet metadata for the source, and it consists of two integers: one indicating which master's log it depends on (the target's), and another indicating the offset into the log where the dependency starts. Once migration has completed and all sidelogs have been committed, the target contacts the coordinator requesting that the dependency be dropped.

If either the source or the target crashes during migration, Rocksteady transfers ownership of the data back to the source. To ensure the source has all of the target's updates, the coordinator induces a recovery of the source server which logically forces replay of the target's recovery log tail along with the source's recovery log. This approach keeps things simple by reusing the recovery mechanism at the expense of extra recovery effort (twice as much as for a normal recovery) in the rare case that a machine actively involved in migration crashes. Extending RAMCloud's recovery to allow recovery from multiple logs is straightforward.

2.4 Evaluation

To evaluate Rocksteady, we focused on five key questions:

How fast can Rocksteady go and meet tight SLAs? §2.4.2 shows Rocksteady can sustain migration at 758 MB/s with 99.9^{th} percentile access latency of less than 250 µs.

Does lineage accelerate migration?

Lineage and deferred log replication allow Rocksteady to migrate data 1.4× faster than synchronous re-replication, while shifting load from the source to the target more quickly (§2.4.2).

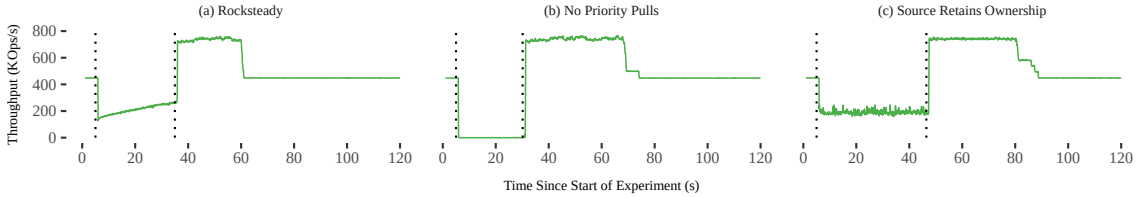


Figure 2.9: Running total YCSB-B throughput for (a) Rocksteady, (b) Rocksteady with no `PriorityPulls`, and (c) when ownership is left at the source throughout the migration. Dotted lines demarcate migration start and end.

What is the impact at the source and target? §2.4.3

shows that regardless of workload skew, Rocksteady migrations cause almost no increase in source dispatch load, which is the source's most scarce resource for typical read-heavy workloads. Background `Pulls` add about 45% worker CPU utilization on the source, and Rocksteady effectively equalizes CPU load on the source and target. Dispatch load due to the migration manager on the target is minimal.

Are asynchronous batched priority pulls effective? §2.4.4

shows that asynchronous priority pulls are essential in two ways. First, synchronous priority pulls would increase both dispatch and worker load during migration due to the increased number of RPCs to the source and the wasted effort waiting for `PriorityPull` responses. Second, asynchronous batched `PriorityPulls` reduce load at the source fast enough to help hide the extra load due to background `Pulls` on the source, which is key to Rocksteady's fast transfer.

What limits migration? §2.4.5 shows that the source and target can send/consume small records at 5.7 GB/s and 3 GB/s, respectively; for small records target replay limits migration more than networking (5 GB/s today). Target worker cores spend 1.8 to 2.4× more cycles processing records during migration than source worker cores.

2.4.1 Experimental Setup

All evaluation was done on a 24 server Dell c6220 cluster on the CloudLab testbed [93] (Table 2.1). RAMCloud is transport agnostic; it offers RPC over several hardware and transport protocol combinations. For these experiments, servers were interconnected with 40 Gbps Ethernet and Mellanox ConnectX-3 cards; hosts used DPDK [27] and the `mlx4`

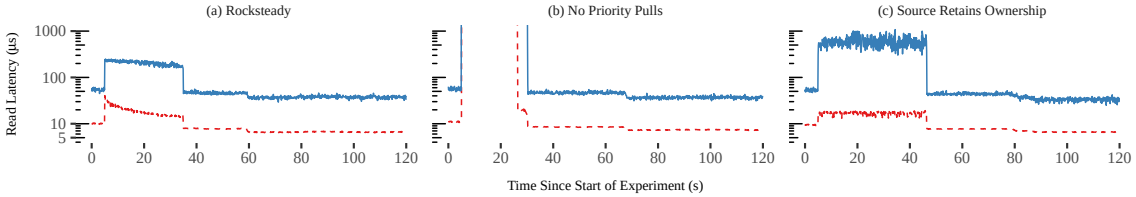


Figure 2.10: Running median (dashed line) and 99.9th percentile (solid line) client-observed access latency on YCSB-B for (a) Rocksteady, (b) Rocksteady with no `PriorityPulls`, and (c) when ownership is left at the source throughout.

poll-mode driver for kernel-bypass support. Each RAMCloud server used one core solely as a dispatch core to manage the network; it used 12 additional cores as workers to process requests; the remaining three cores helped prevent interference from background threads. The dispatch core runs a user-level reliable transport protocol on top of Ethernet that provides flow control, retransmission, etc. without the overhead of relying on the kernel TCP stack.

To evaluate migration under load, 8 client machines run the YCSB-B [18] workload (95% reads, 5% writes, keys chosen according to a Zipfian distribution with $\theta = 0.99$), which accesses a table on the source server. The table consists of 300 million 100 B record payloads with 30 B primary keys constituting 27.9 GB of record data consuming 44.4 GB of in-memory log on the source. Clients offer a nearly open load to the cluster sufficient to keep a single server at 80% (dispatch) load. While the YCSB load is running, a migration is triggered that live migrates half of the records from the source to the target.

Rocksteady was configured to partition the source's key hash space into 8 parts, with each `Pull` returning 20 KB of data. `Pulls` were configured to have the lowest priority in the system. `PriorityPulls` returned a batch of at most 16 records from the source and were configured to have the highest priority in the system. The version of Rocksteady used for the evaluation can be accessed online on github at `https://github.com/utah-scs/RAMCloud/tree/rocksteady-sosp2017`.

2.4.2 Migration Impact and Ownership

Figures 2.9 and 2.10 (a) show Rocksteady's impact from the perspective of the YCSB clients. Migration takes 30 s and transfers at 758 MB/s. Throughput drops when ownership is transferred at the start of migration, since the clients must wait for records to

arrive at the target. As records are being transferred, 99.9th percentile end-to-end response times start at 250 µs and taper back down to 183 µs as hot records from `PriorityPulls` arrive at the target. After migration, median response times drop from 10.1 µs to 6.7 µs, since each server's dispatch is under less load. Likewise, after migration moves enough records, throughput briefly exceeds the before-migration throughput, since client load is open and some requests are backlogged.

Figures 2.9 and 2.10 (b) show `PriorityPulls` are essential to Rocksteady's design. Without `PriorityPulls`, client requests for a record cannot complete until they are moved by the `Pulls`, resulting in requests that cannot complete until migration is done. Only a small fraction of requests complete while the migration is ongoing, and throughput is elevated after migration for a longer period. In practice, this would result in timeouts for client operations. Migration speed is 19% faster (904 MB/s) without `PriorityPulls` enabled.

Instead of transferring ownership to the target at the start of migration, another option is to leave ownership at the source during migration while synchronously re-replicating migrated data at the target. Figures 2.9 and 2.10 (c) explore this approach. The main drawback is that it cannot take advantage of the extra resources that the target provides. Similar to the case above, source throughput decreases under migration load, and clients eventually fall behind. For long migrations, this can lead to client timeouts in a fully open load, since throughput would drop below offered load for the duration of migration. Additionally, migration suffers a 27.7% slowdown (758 MB/s down to 549 MB/s), and the impact on the 99.9th percentile access latency is worse than the full Rocksteady protocol because of the re-replication load generated by the target interfering with the replication load generated by writes at the source. For larger RAMCloud clusters, such interference will not be an issue, and one would expect the 99.9th percentile to be similar to Rocksteady.

2.4.3 Load Impact

Figure 2.11 (a) shows Rocksteady immediately equalizes dispatch load on the source and target. Worker and dispatch load on the target jumps immediately when migration starts, offloading the source. Clients refresh their stale tablet mappings after migration starts. Dispatch is immediately equalized because a) exactly half of the table ownership

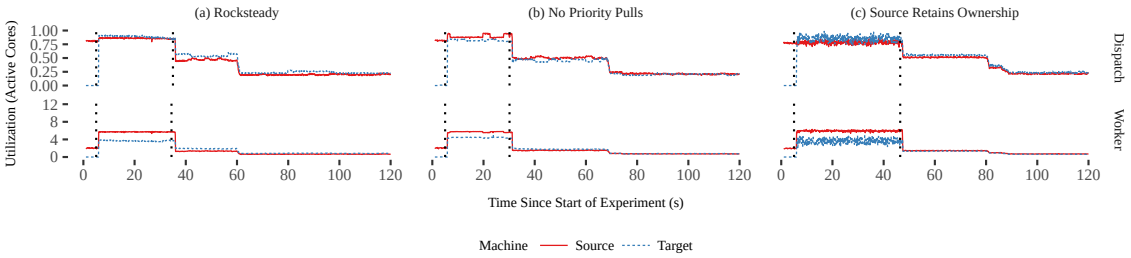


Figure 2.11: Dispatch core and worker core utilization on both source and target for (a) Rocksteady, (b) Rocksteady with no `PriorityPulls`, and (c) when ownership is left at the source throughout the migration.

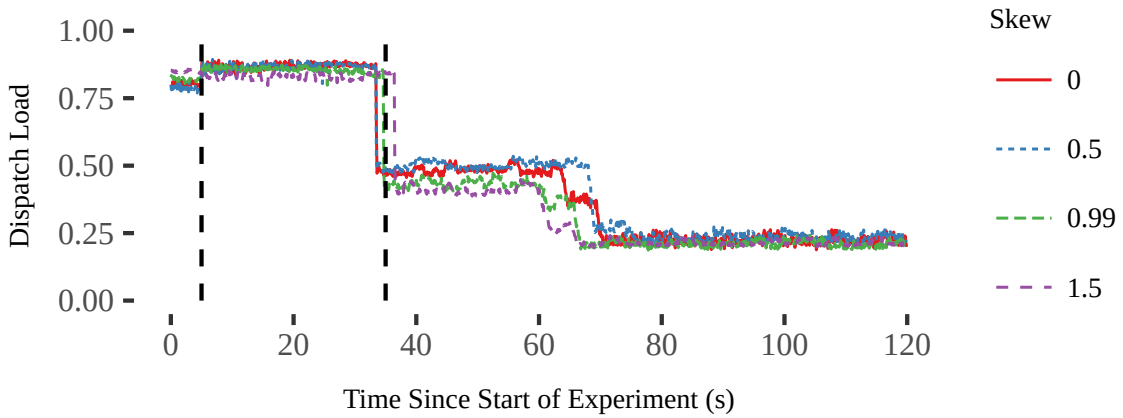


Figure 2.12: Impact of workload access skew on source-side dispatch load. Batched `PriorityPulls` hide the extra dispatch load of background `Pulls` regardless of access skew.

has been shifted to the target, and b) the migration manager is asynchronous and requires little CPU.

A key goal of Rocksteady is to shift load quickly from the source to the target. Most workloads exhibit some skew, but the extent of that skew impacts Rocksteady's ability to shift load quickly. Figure 2.12 examines the extent to which Rocksteady's effectiveness at reducing client load is skew dependent. With no skew (uniform access, skew θ = 0) `PriorityPulls` are sufficient to maintain client access to the tablet, but low request locality means the full load transfer only proceeds as quickly as the background pulls can transfer records. Overall, the results are promising when considering the source's dispatch load, which is its most scarce resource for typical read-heavy workloads. Regardless of workload skew, source-side dispatch load remains relatively flat from the time migration starts until it completes. This means that Rocksteady's eager ownership transfer enabled by batched `PriorityPulls` makes up for any extra dispatch load the `Pulls` place on the

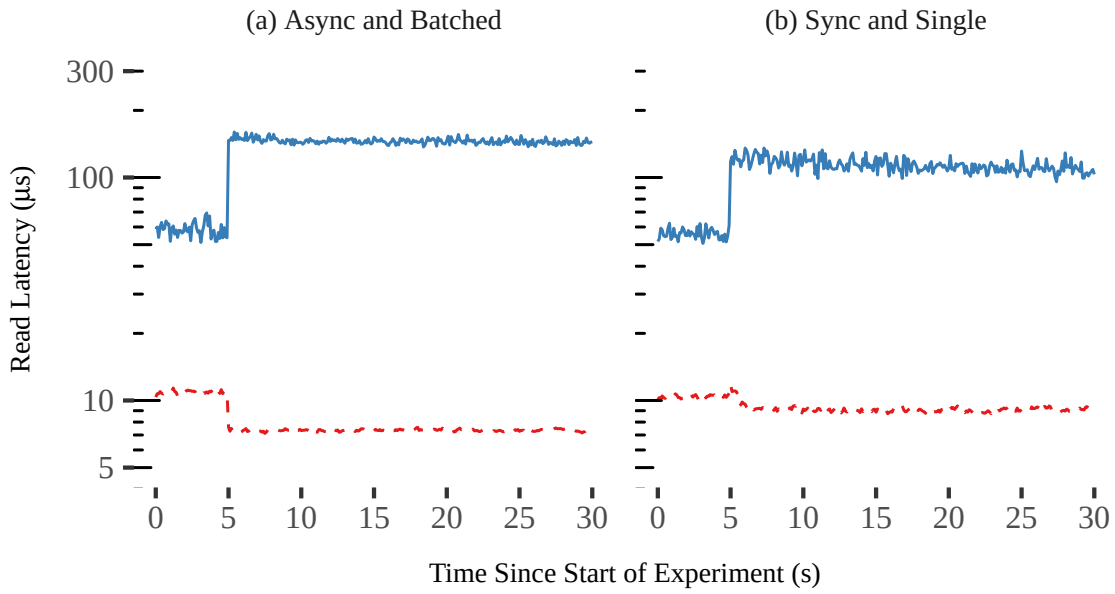


Figure 2.13: Median (dashed line) and 99.9th percentile (solid line) access latency without background `Pulls`. Async batched `PriorityPulls` restore median latency almost immediately compared to sync `PriorityPulls`.

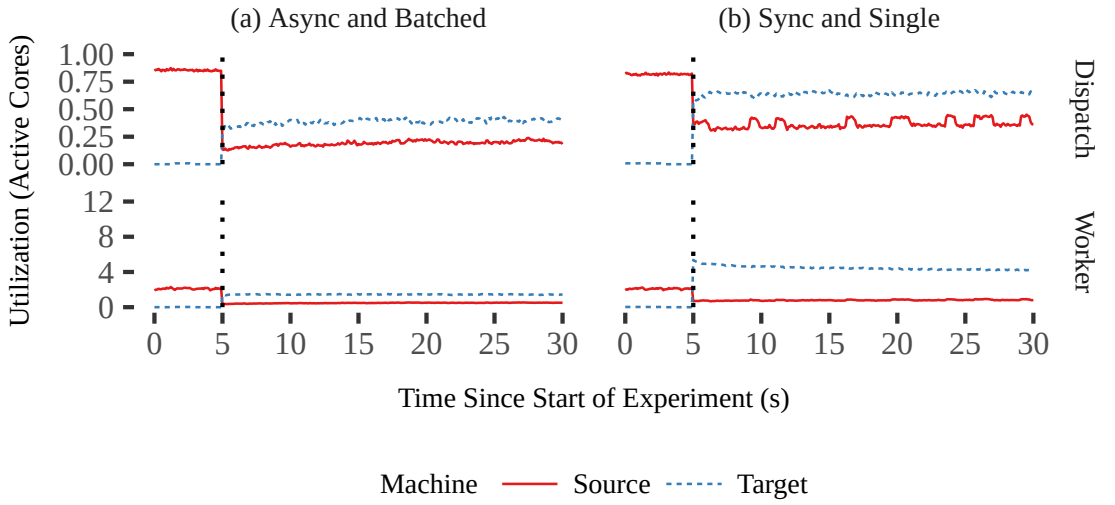


Figure 2.14: CPU Load with no background `Pulls`. Asynchronous batched `PriorityPulls` improve dispatch and worker utilization at both the source and target compared to synchronous `Pulls` that stall target worker cores.

source regardless of the skew.

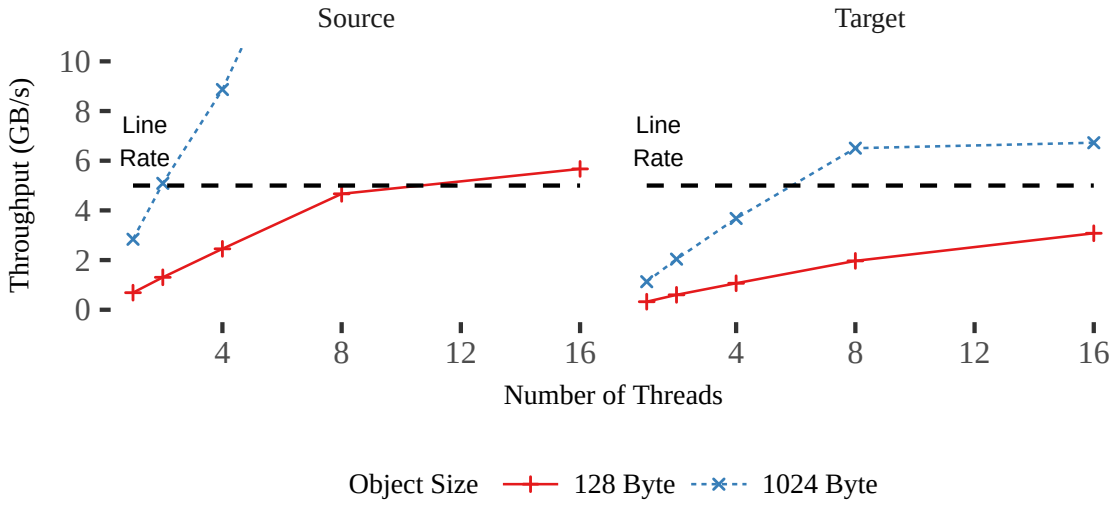


Figure 2.15: Source and target parallel migration scalability. Source side pull logic can process small 128 B objects at 5.7 GB/s. Target side replay logic can process small 128 B objects at 3 GB/s. For larger objects, neither side limits migration.

2.4.4 Asynchronous Batched Priority Pulls

Figures 2.13 and 2.14 compare asynchronous batched `PriorityPulls` with the naïve, synchronous approach when background `Pulls` are disabled. The asynchronous approach doesn't help tail latency: 99.9th latency stays consistent at 160 µs for the rest of the experiment, but median access latency drops to 7.4 µs immediately. On the other hand, the synchronous approach results in median latency jitter, primarily due to workers at the target waiting for `PriorityPulls` to return, which can be seen in the increased worker utilization at the target (Figure 2.14b). However, 99.9th percentile access latency is lower than the asynchronous approach since pull responses are sent to waiting clients immediately.

`PriorityPulls` are critical to the goal of rapidly shifting load away from the source. The headroom thus obtained can be used to service `Pulls` at the source thereby allowing the migration to go as fast as possible. At the same time, `PriorityPulls` help maintain tail latencies by fetching client requested data on-demand.

2.4.5 Pull and Replay Scalability

Parallel and pipelined pulls and replay are key to migration speed that interleaves with normal case request processing. The microbenchmark shown in Figure 2.15 explores the

scalability of the source and target pull processing logic. In the experiment, the source and target pull/replay logic was run in isolation on large batches of records to stress contention and to determine the upper bound on migration speed at both ends independently.

Overall, both the source and target can process pulls and replays in parallel with little contention. In initial experiments, performance was limited when the target replayed records into a single, shared in-memory log, but per-worker side logs remedy this. Small 128 B records (like those used in the evaluation) are challenging. They require computing hashes and checksums over many small log entries on both the source and target. On the target, they also require many probes into the hash table to insert references, which induces many costly cache misses. Even so, the source and target can migrate 5.7 GB/s and 3 GB/s respectively. The source outpaces target replay by 1.8 to 2.4× on the same number of cores, so migration stresses the target more than the source. This works well for scaling out, since the source is likely to be under an existing load that is being redistributed to a less loaded target. For larger record sizes, pull/replay logic doesn't limit migration.

2.5 Discussion

Some of the most broadly applicable lessons from Rocksteady are on the interplay of partitioning, dispatch, and synchronization. Recent works have often partitioned operations [51, 92] or sometimes just mutating operations [68] to reduce locking and contention. Systems that strictly partition work (even just writes) are likely to have to reconfigure more often under skew. Their access latencies also suffer, since migration must be interleaved with normal execution. RAMCloud's dispatch can be a bottleneck, but it can also redirect any idle CPU resources on few microseconds timescale, which is key to Rocksteady's adaptive parallel replay and tight SLAs. Hardware-assisted [52], client-assisted [68], and parallel dispatch help mitigate bottlenecks and delay the need for migration, but none of these can eliminate the need for cross-machine rebalancing or the need to overlap normal execution and migration. Optimizing for normal-case, steady-state request processing can make inevitable background system tasks more costly. Designers of in-memory systems must carefully navigate partitioning, dispatch, and locking trade-offs when planning for heavy rebalancing operations, like migration.

Rocksteady's safe deferred re-replication can also be applied to other systems. For

example, H-Store with the Squall [31] migration system could exploit the same idea to improve migration throughput and access distribution impact. Squall could take a temporary dependency on source data and backups to avoid synchronous re-replication at the target; this would have a significant impact since re-replication blocks execution on the whole target partition in Squall.

2.5.1 Going Even Faster

Rocksteady can migrate hundreds of megabytes per second with tight response latency, but it still only uses a small fraction of the bandwidth provided by modern networks. While its approach and its implementation can be tuned for some gains, it is unlikely that simple changes would result in the order-of-magnitude speed up that would be needed to saturate the network.

To achieve such gains without destroying normal case request processing, migration might be limited to merely transferring large, opaque memory regions between hosts, with little-to-no packaging or replay work on either end. This would require the source to keep state strictly physically partitioned in fine enough units for it to satisfy all possible future splits. FaRM's data layout for example, meets these properties [30].

Physically partitioning groups of records on key or key hash would constrain RAMCloud's log structured memory cleaning. The cleaner minimizes cleaning CPU and memory bandwidth load by physically colocating records that are likely to have a similar lifetime [94]. With physical partitioning constraints, the cleaner wouldn't be able to globally optimize hot/cold separation of objects. Investigating the cleaner's sensitivity to such partitioning could be an interesting direction, particularly since it might be able to assist in the process of physically partitioning records.

Even if records were partitioned and could be moved at line rate, it is possible that RAMCloud would need network-level support in order to avoid interference between large, fast migration transfers and fine-grained normal-case requests.

Beyond improvements in dispatch scalability, other improvements to RAMCloud's concurrency model could also have a significant impact on Rocksteady. Today, RAMCloud processes requests on workers that use standard kernel threads. Coroutines or cooperative user-level threading could both improve response distributions and efficiency [49]. If

`Pull` and replay operations could afford frequent yields to RAMCloud's dispatch, heavy operations would have less impact on normal case request processing. Replay and `Pull` operations could be coarser as well, resulting in less requests and lower dispatch overheads. This could allow Rocksteady to transfer data even more quickly with the same SLAs.

2.6 Related Work

Amazon's Dynamo [26] is a highly-available distributed key-value store that pushed for focus on 99.9th percentile access latency, though Rocksteady pushes for tail latency nearly 1,000× lower even while migrating. Dynamo supported strong SLAs and reconfiguration through a very different approach that took advantage of pre-partitioning records inside each server, replication, and weak consistency. DRAM is expensive, so Rocksteady must not rely on in-memory replication or internal pre-partitioning of records.

Distributed database live migration has received a great deal of attention, particularly for multi-tenant cloud databases. Rocksteady uses many ideas from prior work like pacing migration [6], eager transfer of ownership [31, 32], and combining on-demand and background migration [31, 32, 97]. Others have explored holding ownership at the source and "catching up" the target through delta records or recovery log data [23], similar to RAMCloud's original migration.

Squall [31, 109] is a state-of-the-art live migration system for the H-Store [51] scale-out shared-nothing database. It offloads the source quickly by breaking requested tuples out into separate units and migrating them on-demand. Under skewed loads, hot tuples move quickly and background transfers are paced to try to minimize disruption. Rocksteady uses Squall's combined background/tuple-level reactive pull, but it extends the approach to RAMCloud's more flexible parallelism model. H-Store's strict serial execution makes synchronizing with migration expensive; the execution of migration operations on a partition are interlocked between the source and target and block normal requests. That is, each pull from a target core can only be serviced by a specific source core, and pulls and replays must operate in isolation on a partition. Requests cannot be processed for keys that are being pulled (or for *any* key in a partition where a pull is ongoing). Target cores also spin waiting for pull responses hurting normal request access latency and throughput as

well as migration speed. Compared to all prior approaches, Rocksteady transfers data an order of magnitude faster with tail latencies 1,000× lower; in general, Rocksteady's ability to use *any* available core for *any* operation is key for both tail latency and migration speed.

Rocksteady builds on recent work on recovery and dispatching for in-memory storage that relies on kernel-bypass networking [8, 47–49, 67, 68, 76, 87]. RAMCloud's recovery is a form of distributed migration [81], but it is disruptive since it uses the resources of the entire cluster to reload contents of a crashed server as fast as possible. FaRM [29, 30] relies on in-memory triplication for redundancy, but it must re-replicate lost partitions when a server fails. It paces recovery to a few hundred megabytes per second per server in order to minimize performance impact. Similarly, DrTM-B [115] minimizes the impact of reconfiguration by relying on in-memory replicas. However, replicas can become overloaded too, so data is migrated using parallel RDMA reads. One key aspect of FaRM is that partitions are physical: a lost partition is an opaque region of memory, so most of the overhead of re-replication is network transfer. RAMCloud migration is more complex, since the source and target don't share a common partitioning or physical memory layout.

Rocksteady's fast parallel packaging and replay is similar to Silo's single-server parallel recovery [112, 120]. Silo partitions recovery logs across cores during record and during replay. Rocksteady's replay doesn't require any particular order; any core can replay any portion of records, which helps Rocksteady hit SLAs. In Silo the database is also naturally offline during replay, and recovery can consume all of the resources of the machine. Silo's parallel replay is state-of-the-art, but Rocksteady's parallel replay outperforms it on far fewer cores. This may be because Silo must reconstruct a tree-like index rather than a flat hash table and filesystem I/O may induce more overhead than a NIC using kernel-bypass.

2.7 Conclusion

Low-latency in-memory stores are designed to tolerate the heaviest request loads, but if they are too stripped down they cannot deal with complex higher-level operations like reacting to workload changes, skew shifts, and load spikes. Rocksteady is a migration protocol for in-memory key-value stores that avoids the need for and overhead of in-advance state partitioning; it eliminates replication overhead from the migration fast path; it exploits parallelism; and it exploits modern NIC hardware. Rocksteady has a

"pay-as-you-go" approach that helps avoid overloading the source during migration using asynchronous batched on-demand pulls to shift load away from the source as parallel background transfers proceed. In all, Rocksteady can move the entire DRAM of a modern data center machine in a few minutes while retaining 99.9th percentile tail latency of less than 250 µs.

CHAPTER 3

LOW COST COORDINATION

3.1 Introduction

Millions of sensors, mobile applications, users, and machines now continuously generate billions of events. These events are processed by streaming engines [15, 102] and ingested and aggregated by state management systems (Figure 3.1). Real-time queries are issued against this ingested data to train and update models for prediction, to analyze user behavior, or to generate device crash reports, etc. Hence, these state management systems are a focal point for massive numbers of events and queries over aggregated information about them.

Recently, this has led to specialized KVSs that can ingest and index these events at high rates – 100 million operations (Mops) per second (s) per machine – by exploiting many-core hardware [16, 117]. These systems are efficient if events are generated on the same machine as the KVS, but, in practice, events need to be aggregated from a wide and distributed set of data sources. Hence, fast indexing schemes alone only solve part of the problem. To be practical and cost-effective, a complete system for aggregating these events must ingest events over the network, must scale across machines as well as cores, and must be elastic (by provisioning and reconfiguring over inexpensive cloud resources as workloads change).

The only existing KVSs that provide similar performance [52, 65, 68, 88] rely on application-specific hardware acceleration, making them impossible to deploy on today's cloud platforms. Furthermore, these systems only store data in DRAM, and they do not scale across machines; adding support to do so without cutting into normal-case performance is not straightforward. For example, many of them statically partition

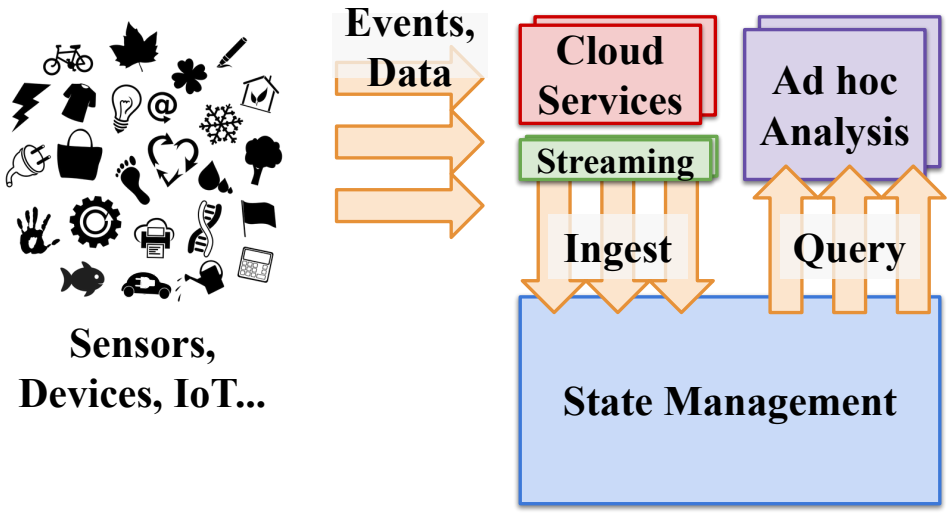


Figure 3.1: A typical data processing pipeline. Services receive and process raw events. A state management system ingests processed events and serves offline queries against them.

records across cores to eliminate cross-core synchronization. This optimizes normal-case performance, but it makes concurrent operations like migration and scale out impossible; transferring record data and ownership between machines and cores requires a stop-the-world approach due to these systems' lack of fine-grained synchronization.

Achieving this level of performance while fulfilling all of these requirements on commodity cloud platforms requires solving two key challenges simultaneously. First, workloads change over time and cloud VMs fail, so systems must tolerate failure and reconfiguration. Doing this without hurting normal-case performance at 100 Mops/s is hard, since even a single extra server-side cache miss to check key ownership or reconfiguration status would cut throughput by tens-of-millions of operations per second. Second, the high CPU cost of processing incoming network packets easily dominates in these workloads, especially since, historically, cloud networking stacks have not been designed for high data rates and high efficiency. We show this is changing; by careful design of each server's data path, cloud applications can exploit transparent hardware acceleration and offloading offered by cloud providers to process more than 100 Mops/s per cloud virtual machine (VM).

We present *Shadowfax*, a new distributed KVS that transparently spans DRAM, SSDs, and cloud blob storage while serving 130 Mops/s/VM over commodity Azure

VMs [19] using conventional Linux TCP. Beyond high single-VM performance, its unique approach to distributed reconfiguration avoids any server-side key ownership checks and any cross-core coordination during normal operation and data migration both in its indexing and network interactions. Hence, it can shift load in 17 s to improve cluster throughput by 10 Mops/s with little disruption. Compared to the state-of-the-art, it has 8× better throughput (than Seastar+memcached [98]) and scales out 6× faster (than Rocksteady [58]).

In this paper, we describe and evaluate three key pieces of Shadowfax that eliminate coordination throughout the client- and server-side by eliminating cross-request and cross-core coordination:

Low-cost Coordination via Global Cuts: In contrast to totally-ordered or stop-the-world approaches used by most systems, cores in Shadowfax avoid stalling to synchronize with one another, even when triggering complex operations like scale-out, which require defining clear before/after points in time among concurrent operations. Instead, each core participating in these operations – both at clients and servers – independently decides a point in an *asynchronous global cut* that defines a boundary between operation sequences in these complex operations. In this paper, we extend asynchronous cuts from cores within one process [16, 89] to servers and clients in a cluster, and we show how they coordinate server and client threads (through partitioned sessions) by detailing their role in Shadowfax's low-coordination data migration and reconfiguration protocol.

End-to-end Asynchronous Clients: All requests from a client on one machine to Shadowfax are asynchronous with respect to one another all the way throughout Shadowfax's client- and server-side network submission/completion paths and servers' indexing and (SSD and cloud storage) I/O paths. This avoids all client- and server-side stalls due to head-of-line blocking, ensuring that clients can always continue to generate requests and servers can always continue to process them. In turn, clients naturally batch requests, improving server-side high throughput especially under high load. This batching also suits hardware accelerated network offloads available in cloud platforms today further lowering CPU load and

improving throughput. Hence, despite batching, requests complete in less than 40 µs to 1.3 ms at more than 120 Mops/s/VM, depending on which transport and hardware acceleration is chosen.

Partitioned Sessions, Shared Data: Asynchronous requests eliminate blocking *between requests* within a client, but maintaining high throughput also requires minimizing coordination costs *between cores* at clients and servers. Instead of partitioning data among cores to avoid synchronization on record accesses [51, 68, 98, 105], Shadowfax partitions network sessions across cores; its lock-free hash index and log-structured record heap are shared among all cores. This risks contention when some records are hot and frequently mutated, but this is more than offset by the fact that no software-level inter-core request forwarding or routing is needed within server VMs.

The rest of the paper is organized as follows. We provide background on the FASTER key-value store and its use of epochs within a machine (§3.2). Next, we overview Shadowfax's design, including partitioned client sessions with global cuts and how they enable reconfiguration (§4.3). We then provide details on our parallel non-blocking migration and scale-out techniques (§3.4). Next, we evaluate Shadowfax in detail against other state-of-the-art shared-nothing approaches (§2.4), showing that by eliminating record ownership checks and cross-core communication for routing requests it improves per-machine throughput by 8.5× on commodity cloud VMs. We also show it retains high throughput during migrations and scaled it to a cluster that ingests and indexes 400 Mops/s in total, which, to the best of our knowledge, is the highest reported throughput for a distributed KVS till date. We finally cover related work (§4.6) and conclude the paper (§**??**).

3.2 Background on FASTER

Shadowfax is built over the `FASTER` single-node KVS, which it relies on for hash indexing and record storage. Here, we describe some key aspects of `FASTER`, since Shadowfax's design integrates with it and builds on its mechanisms. More details about `FASTER` itself can be found elsewhere [16, 89]. Specifically, Shadowfax extends `FASTER`'s asynchronous cuts, which help avoid coordination, and its `HybridLog`, which transparently spans DRAM and SSD.

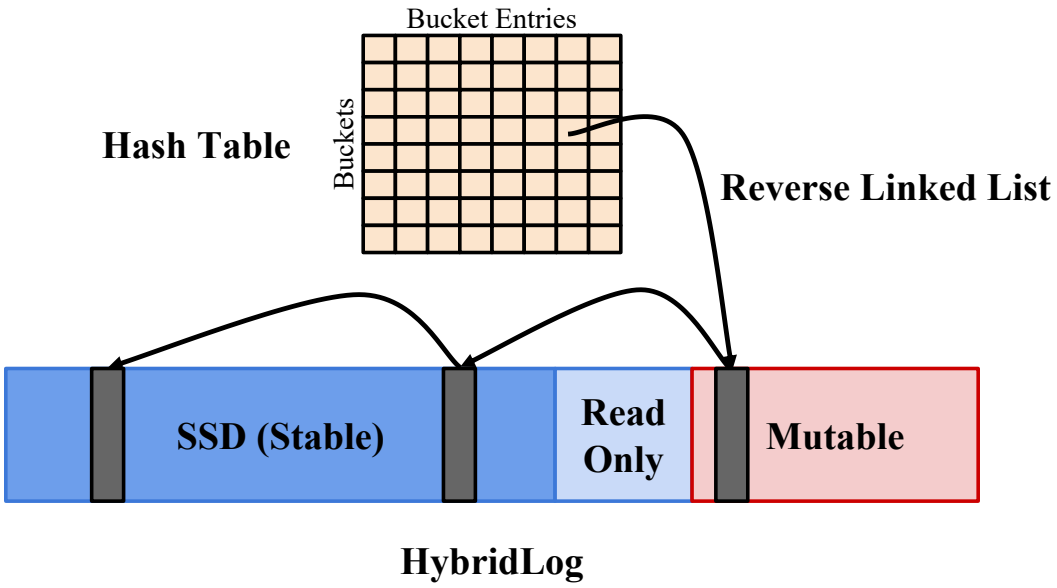


Figure 3.2: FASTER's HybridLog allocator spans memory and local SSD. The portion in memory contains a mutable region that acts as a cache and a read-only region. FASTER's hash table points to a reverse linked list of records on the HybridLog.

In most ways, FASTER works like most durable hash table libraries. It includes a lock-free hash table divided into cacheline-sized buckets (Figure 3.2). Each 8 byte bucket entry contains a pointer to a record whose key hashes to that bucket. Each record points to another record, forming a linked list of records with common significant key hash bits. Each bucket entry contains additional bits from the associated records' key hash, increasing hashing resolution and disambiguating what records the bucket entry points to without extra cache misses and without full key comparisons. Each record pointed to by the hash table is stored in the HybridLog.

FASTER clients can use it like any other library, but a common pattern is to pin one client application thread per CPU core to eliminate scheduler overheads. Each client thread calls read or read-modify-write operations on keys in FASTER. FASTER's cache-conscious design and lock-freedom are key in its ability to perform more than 100 Mops/s on a single multicore machine.

3.2.1 HybridLog Allocator

FASTER allocates and stores all records in its HybridLog, which spans memory and SSD (Figure 3.2). The HybridLog combines in-place updates (for records in memory) and

log-structured organization (for records on SSD), and provides lock-free access to records.

The portion of the HybridLog's address space on SSD forms the stable region. It contains cold records that have not been recently updated. The portion in memory is composed of two regions: a (larger) mutable region and a (smaller) read-only region. Records in the mutable region can be modified in-place with appropriate synchronization that is chosen by the application using FASTER (for example, atomic operations, locks, or validation). This region acts as a cache for recently updated records and avoids expensive per-update allocations.

The read-only region mostly contains records that are being asynchronously written to SSD. These records cannot be updated in place, since they must remain stable during I/O. The read-only region represents records that are becoming cold, and it acts as a second-chance cache. FASTER uses a read-copy-update to modify records in this region: the updated record version is appended to the mutable region, and the hash table is updated to point to it. This helps provide good cache hit rates without fine-grained metadata.

Each record entry in FASTER's hash table points to a reverse linked list of records on the HybridLog, allowing it to maintain a compact hash table for *larger-than-memory* datasets that span storage media. Note, that a consequence of this is that hash table lookups in FASTER may need to traverse chains of records that span from memory onto SSD. Section 3.4.2 describes how Shadowfax extends HybridLog so that it also spans shared cloud storage and how this accelerates the completion of scale out and data migration.

3.2.2 Asynchronous Cuts

Lock-freedom makes FASTER fast, but it creates challenges for synchronization and memory safety. Updated versions of records may be installed in its hash table, even as old versions of that record are still being read by other threads. This is a common problem in all lock-free, RCU-like schemes [73]. To solve this, FASTER uses an epoch-based memory-protection scheme [60]. All threads calling into FASTER are registered with an epoch manager that tracks when threads begin and end access to FASTER's internal structures. When a page is evicted to SSD, the epoch-based scheme ensures that the memory is not reused while any thread could still be accessing it. The full details of this scheme are beyond the scope of this paper.

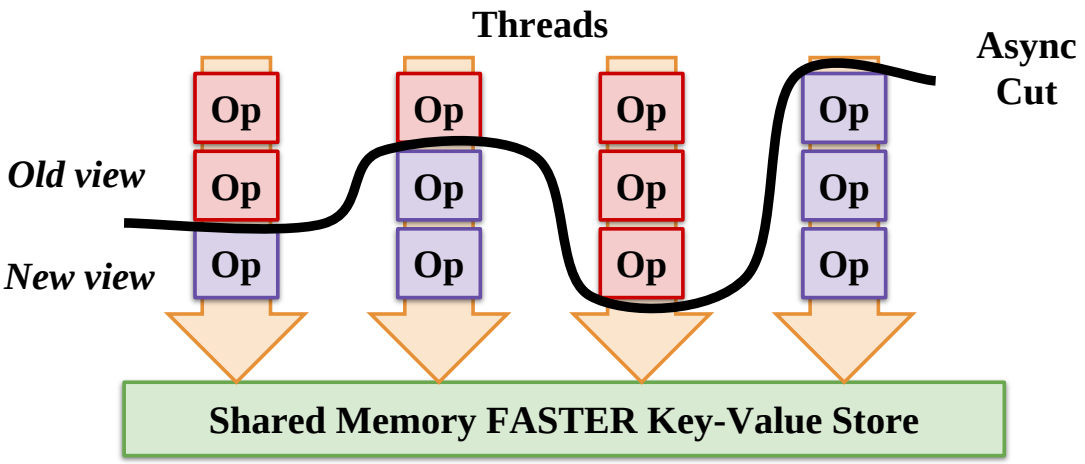


Figure 3.3: View changes of shared state in `FASTER` take place over an asynchronous cut using epochs. Process-global state is updated first; when every thread has observed the update a post-change function is triggered.

Critically, this epoch-based scheme also plays a key role in coordinating information across threads lazily without inducing stalls. During complex, process-wide events (such as page eviction and checkpointing), threads lazily coordinate by registering callback actions that are eventually executed once each thread synchronizes some local state with an updated process-global value. The same mechanism can also be used to trigger a function only once all threads are guaranteed to have updated their local state from some process-global state. In effect, this allows trigger actions that are guaranteed to take effect only after all threads agree on and have each locally observed some transition in process-global state. This can be used to create a process-wide *asynchronous cut*, where events such as process state transitions are realized asynchronously and lazily over a set of independent thread-local state transitions.

For instance, consider the read-offset address that demarcates read-only records from mutable records on the `HybridLog` (Section 3.2.1). When this address is updated, each thread may notice the update at different points in time, depending on when they refresh their epoch. Eventually, when all threads have observed the update, the records between the old and new read-offsets have become read-only, and a function is triggered to write the pages to disk. Using the same mechanism, addresses for which threads do not yet agree on the mutability status can be handled efficiently. Figure 3.3 shows this process in action.

FASTER's epoch protection works within a single shared memory process on one machine. Section 3.3.2.1 shows how Shadowfax extends the notion of cuts to apply globally *across machines* – with the assistance of client threads – to safely move ownership of records between servers while preserving throughput.

3.3 Shadowfax Design

Shadowfax is a distributed key-value store. Each server in the system stores records inside an instance of `FASTER`, and clients issue requests for these records over the network. These requests can be of three types: *reads* that return a record's value, *upserts* that blindly update a record's value, and *read-modify-writes* that first read a record's value and then update a particular field within it. Within a server, records are allocated on `FASTER`'s `HybridLog`, whose stable region is extended by Shadowfax to also span a shared remote storage tier in addition to main memory and local SSD.

Each server runs one thread per core, and it shares its `FASTER` instance among all threads. Threads on remote clients directly establish a network *session* with one server thread on the machine that owns the record being accessed (§3.3.1.1). Sessions are the key to retaining `FASTER`'s throughput over the network: they allow clients to issue asynchronous requests; they batch requests to improve server-side throughput and avoid head-of-line blocking; and they avoid software-level inter-core request dispatching.

Shadowfax uses hash partitioning to divide records among servers. The set of hash ranges owned by a server at a given logical point of time is associated with a per-server strictly increasing *view number*. A fault-tolerant, external metadata store (e.g. ZooKeeper [41]) durably maintains these view numbers along with mappings from hash ranges to servers and vice versa. View numbers serve two key purposes in Shadowfax. First, they help minimize the impact of record ownership checks at servers, helping them retain `FASTER`'s performance. Second, they allow the system to make lazy and asynchronous progress through record ownership changes (§3.3.2).

Sessions and low-coordination global cuts via views play a key role in Shadowfax's reconfiguration, data migration, and scale out. Its scale out protocol migrates hash ranges from a *source* server to a *target* server and is designed to minimize migration's impact to throughput. The protocol uses a view change to transfer ownership of the hash range from

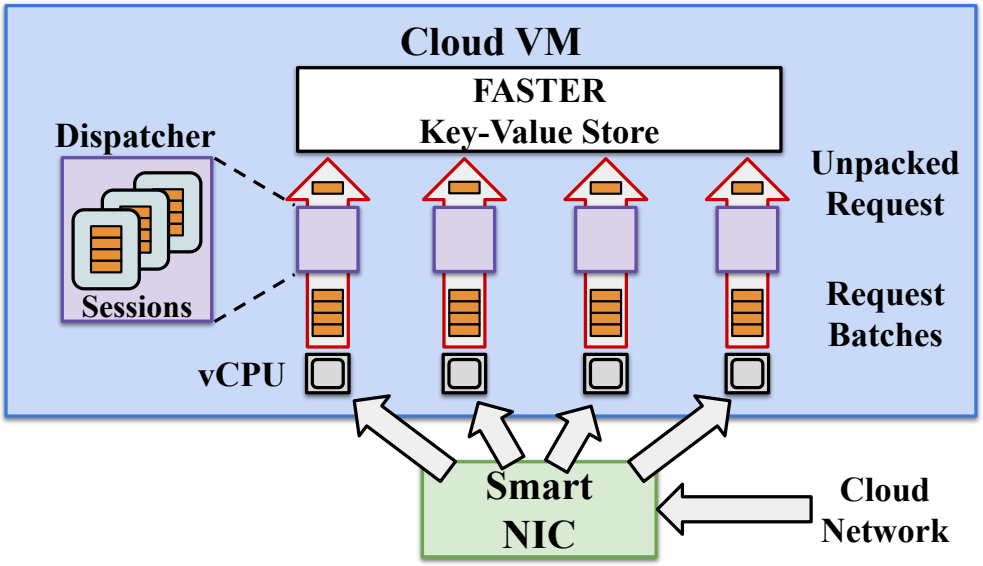


Figure 3.4: Each server thread receives batches of requests from sessions and processes them via a shared, per-machine `FASTER` instance. Results are returned over the network by the same thread, avoiding cross-thread coordination.

the source to the target along with a small set of recently accessed records. This allows the target to immediately start serving requests for these records and helps maintain high throughput during scale out. Since views are per-server, this also ensures that multiple migrations between disjoint sets of machines can take place simultaneously. Next, threads on the source work in parallel to collect records from `FASTER` and transmit them over sessions to the target. Similarly, threads on the target work in parallel to receive these records and insert them into its `FASTER` instance. This parallel approach helps migrate records quickly, reducing the duration of scale out's impact on throughput. Scale out completes once all records have been moved to the target.

3.3.1 Partitioned Dispatch & Sessions

Shadowfax's network request dispatching mechanism and client library need to be capable of saturating servers inside `FASTER`. One option would be to maintain a `FASTER` instance per server thread, partitioning records across them to avoid cache coherence costs. However, this would create a routing problem at the server; requests picked up from the network would need to be routed to the correct thread. This would require cross-thread coordination, hurting throughput and scalability. Clients could be made responsible for routing requests to the correct server thread, but this would require every client thread to

open a connection to every server thread and would not scale. To avoid this, client threads could partition and shuffle requests between themselves to directly transmit requests to the correct server thread, but this would require cross-thread coordination at the client which would also not scale well.

Using a connectionless transport like UDP could make client side routing feasible without introducing cross-thread coordination [68, 79]. However, the system would lose its ability to perform congestion control and flow control, or tolerate packet loss, which are basic requirements for running a networked storage system.

Shadowfax avoids cross-thread coordination by sharing a single instance of `FASTER` between server threads. `FASTER` defers cross-core communication to hardware cache coherence on the accessed records themselves, cleanly partitioning the rest of the system (Figure 3.4). Each server runs a pinned thread on each vCPU inside a cloud VM. Each server thread runs a continuous loop that does two things. First, it polls the network for new incoming connections. Next, it polls existing connections for requests, and it unpacks these requests, calling into `FASTER` to handle each of them. After requests are executed, the returned results are transmitted back over the session they were received on. Since `FASTER` is shared, neither requests nor results are ever passed across server threads.

3.3.1.1 Client Sessions

Shadowfax's partitioned dispatch/shared data approach also extends to clients. Since they don't need to route requests to specific server threads, they can reduce connection state while avoiding cross-thread coordination.

However, clients must also avoid stalling due to network delay in order to saturate servers. To do this, each client thread is pinned to a different vCPU of a cloud VM, and it issues asynchronous requests against an instance of Shadowfax's client library (Figure 3.5). The library pipelines batches of these requests to servers.

The client library achieves this through *sessions*. When the library receives a request, it first checks if it has a connection to the server that owns the corresponding record. If it does not, it looks up a cached copy of ownership mappings (periodically refreshed from the metadata store), establishes a connection to a thread on the server that owns the record, and associates a new *session* with the connection. Next, it buffers the request

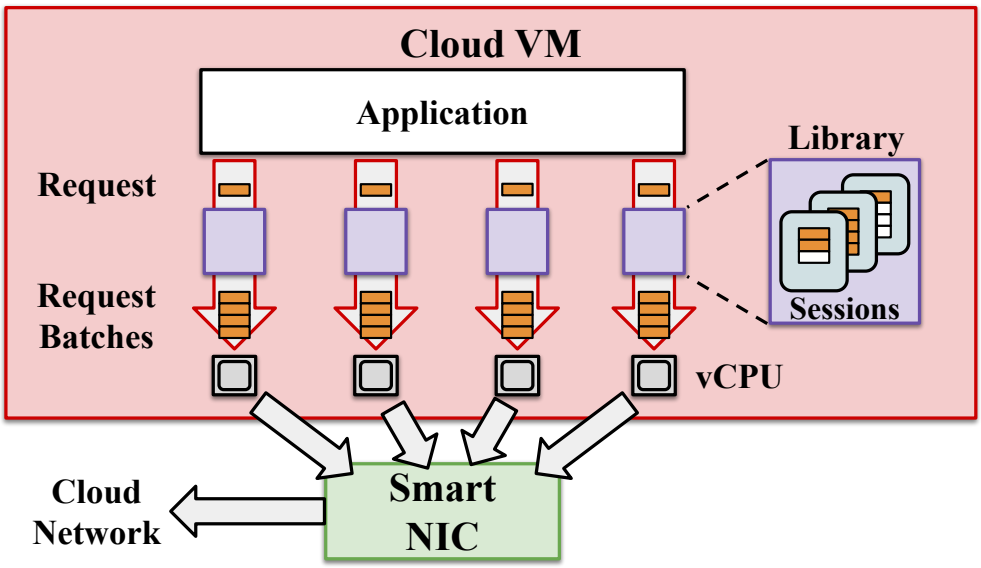


Figure 3.5: Client threads partition requests into per-session transmit buffers along with a callback. Batches of asynchronous requests are kept pipelined to the server, keeping both the client and the server busy.

inside the session, enqueues a completion callback for the request inside the session, and returns. This allows the client thread to continue issuing requests without blocking. Once enough requests have been buffered inside a session, the library sends them out in a batch to the server thread. On receiving a batch of results from the server, the library dequeues callbacks and executes them to complete the corresponding requests.

Sessions are fully pipelined, so multiple batches of requests can be sent to a server thread without waiting for responses. This also means that a client thread can continue issuing asynchronous requests into session buffers while waiting for results. This pipelined approach hides network delays and helps saturate servers. It also helps keep request batch sizes small, which is good for latency.

3.3.1.2 Exploiting Cloud Network Acceleration

The cloud network has traditionally not been designed for high data rates and efficiency. The high CPU cost of processing packets over this network can easily prevent servers and clients from retaining `FASTER`'s throughput. However, this is beginning to change; many cloud providers are now transparently offloading parts of their networking stack onto SmartNIC FPGAs to reduce this cost. Shadowfax's design interplays well with this acceleration; batched requests avoid high per-packet overheads and its reduced

connection count avoids the performance collapse some systems experience [29].

Since threads do not communicate or synchronize, all CPU cycles recovered from offloading the network stack can be used for executing requests at the server and issuing them from the client. This allows Shadowfax to retain `FASTER`'s high throughput using the Linux kernel's TCP stack on cloud networks, avoiding dependence on kernel-bypass or RDMA.

3.3.2 Record Ownership

To support distributed operations such as scale out and crash recovery, Shadowfax must be able to move ownership of records between servers at runtime. This creates a problem during normal operation: a client might send out a batch of requests to a server after referring to its cache of ownership mappings. By the time the server receives the batch, it might have lost ownership of some of the requested records in the batch (e.g. due to scale out). To solve this, the server must validate that it still owns the records requested in the batch before it processes the batched requests. This can hurt normal case throughput if each request in the batch must be cross-checked against a set of hash ranges owned by the server.

Shadowfax solves this by associating the set of hash ranges owned by a server with a per-server strictly-increasing *view number*. All request batches are tagged with a view number, letting servers quickly assess whether the batch only includes requests for records that it currently owns. When a server's set of owned ranges changes, its view number is advanced. Each server's latest view number is durably stored along with a list of the hash ranges it owns in the metadata store.

When a client connects to a server, it caches a copy of the server's latest view (a view number and its associated hash ranges) inside the session. Every batch of requests sent on that session is tagged with this view number, and clients only put requests for keys into batches that were owned by that server in that view number. Upon receiving a batch, the server always checks its current view number against the view number tagged on the batch. If they match, then the server and client agree about what hash ranges are owned by the server, ensuring the batch is safe to process without further key or hash range checks on each request. If they don't match, then either the client or the server has

out-of-date information about what hash ranges the server owns. In this case, the server rejects the batch and refreshes its view from the metadata server. Upon receiving this batch rejection, the client refreshes its view information from the metadata server and then reissues requests from the rejected batch.

In essence, view numbers offload expensive hash range checks on each requested key to clients, reducing load at servers. For a server that owns P hash ranges accepting R requests in batches of size B, views reduce the cost of ownership checks from $O(R \log P)$ to $O(R/B)$. Even more crucially, since it is a single integer comparison per batch, it ensures we never take a cache miss to perform record ownership checks, which would be prohibitive at 100 Mops/s. Hence, views are key in supporting dynamic movement of ownership between servers while preserving normal case throughput.

3.3.2.1 Ownership Transfer

When ownership of a hash range needs to be transferred to or away from a server, its ownership mappings are first atomically updated at the metadata store. This increments its view number and adds or removes the hash range from its mapping. Servers and clients observe this view change either when they refresh their local caches of views and ownership mappings (via an epoch action) or when they communicate with a machine that has already observed this change.

When a server involved in the transfer observes that its view has changed, it must move into the new view. However, this step is not straightforward; keeping with Shadowfax's design principle, it must be achieved without stalling server threads. Within the server, this view change is propagated asynchronously across threads via an epoch action (Figure 3.6). Threads each mark a point in their sequence of operations, collectively creating an async cut among all of the operations on all of the threads at the server (§3.2.2). This cut unambiguously ensures no two servers concurrently serve operations on an overlapping hash range. This approach is free of synchronous coordination, helping maintain high throughput.

The server might be connected to clients still using the old view; it must also propagate the view change to clients in a similar way without stalling client threads. Sessions help Shadowfax achieve this. When a server thread moves into a new view, view validations on

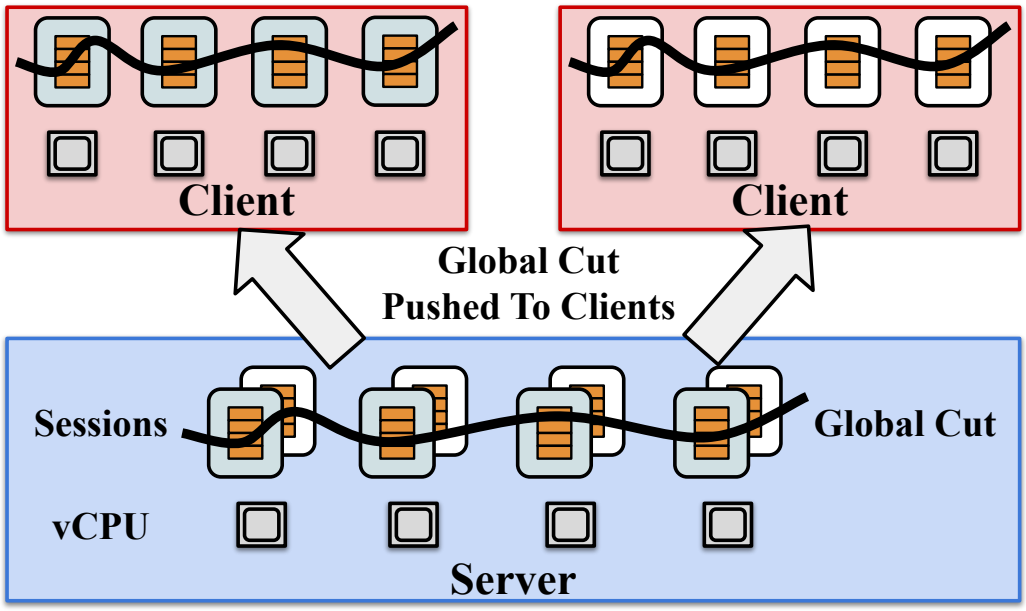


Figure 3.6: Ownership transfer. A view change is asynchronously propagated within a server, defining a cut across server threads. Then, the server extends this into a global cut covering all its connected client sessions. This defines a global view boundary among all operations while avoiding cross-core coordination both at servers *and* clients.

request batches received over sessions with clients still in an older view are rejected. On receiving a rejected batch over a session, each client thread first independently updates its thread local cache of ownership mappings and views. Next, the thread marks the point in the sessions' sequence of operations after which batches were rejected by the server (since there can be multiple such batches because of pipelining, this has to be the earliest such point). Collectively, these points help create an implicit async cut across threads within a client. Thus, clients avoid cross-thread coordination when observing an ownership change. Each client connected to the server creates its async cut independently, resulting in a cluster wide *asynchronous global cut* for ownership transfer.

Once it has observed ownership transfer, each client thread must reissue requests that were rejected by the previous owner. It does so by *shuffling* these requests between its sessions to the previous and new owners of the transferred hash range. First, they are marked invalid within the previous session's buffer. Next, they are (re)buffered into the correct session based on the updated ownership mappings.

3.4 Scale-Out and Hash Migration

Sessions and view numbers help retain high throughput over the network, but only upto a certain point. Beyond high single node throughput, Shadowfax must also scale-out to multiple servers, and it must retain FASTER's throughput on each of these servers.

In a distributed setting, partitioning becomes critical to performance; it is well known that pre-partitioning records between servers results in load imbalances, which significantly hurts throughput [1, 26]. Therefore, in order to meet its performance goals, Shadowfax must be capable of dynamically migrating arbitrary, fine-grained splits of its hash space between servers (new or existing) in response to load imbalances.

For migration to be practical, it must have low impact on throughput, and it must be fast. Partitioned sessions help achieve the latter; they allow servers to collect, transmit, receive and insert migrated records without cross-thread coordination. Since FASTER is shared, server threads can also easily interleave migration work with request processing. Views help too; a server can own many fine-grained splits and still serve 100 Mops/s during normal operation.

Achieving low impact during migration is harder. Ownership of records must be safely moved between servers, requests must be correctly executed on these records, and progress must be tracked. All of these could potentially introduce serial bottlenecks, and must hence be carefully performed in an asynchronous, low-coordination way. Shadowfax's migration protocol uses global cuts to proceed in asynchronous phases that transfer hash range ownership before migrating records, as described next.

3.4.1 Migration Protocol

Shadowfax migrates hash ranges from a *source* to a *target* server. Migration is implemented as a state machine on the source and target. Both servers transition through migration phases on global cuts, created in the same non-blocking, low-coordination way described in §3.2.2. First, each thread enters into a phase at a point in the sequence of requests that it is processing that it chooses (a point that makes up part of the global cut for the transition into that phase), and then it starts performing the work of that phase. Once all threads have entered into the phase and have completed all work relating to it, the server transitions to the next phase.

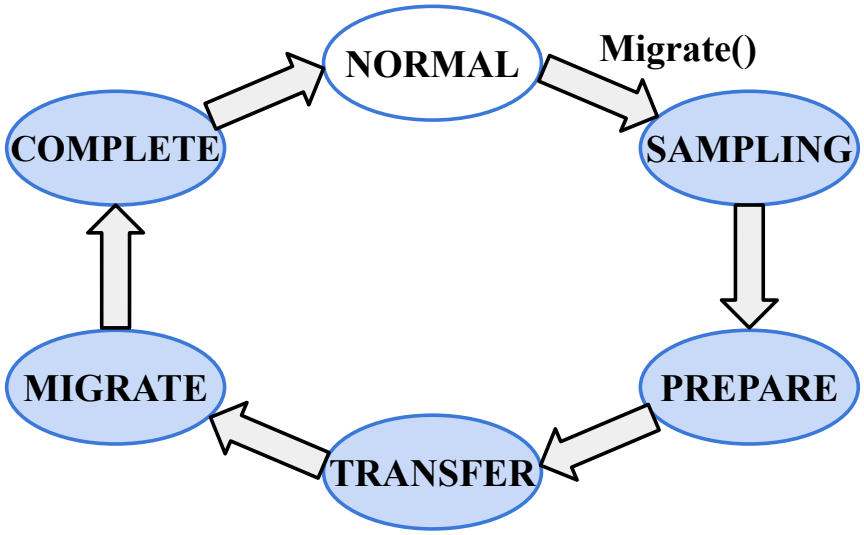


Figure 3.7: Migration state machine on the source. This state machine is responsible for moving the source into the new view, for sampling and shipping hot records to the target, and for migrating all records in the hash range to the target.

Migration is driven by the source as we outline below (Figure 3.7):

Sampling: Initiated by receiving a `Migrate()` RPC from a client, whereupon the source

1. atomically remaps ownership of hash ranges from the source to the target, increments the source's and target's view numbers, and registers a dependency between the source and target (for crash recovery, §3.4.4) within the metadata store;

2. begins sampling hot records by forcing all accessed records to be copied to the `HybridLog` tail.

Since the records are not yet at the target and a migration is in progress, both the source and the target continue to temporarily operate in the old ownership view; at this point the source is still servicing requests for records in the migrating ranges. To ensure that sampled records only get copied once, the source only copies records whose address is lower than the `HybridLog` tail address at the start of this phase.

Prepare: Initiated after all source threads have completed the Sampling phase. The source

sends a `PrepForTransfer()` RPC to the target asynchronously, transitioning the target to its own Target-Prepare phase. The Target-Prepare phase tells the target that ownership transfer is imminent. The target temporarily pends requests in the migrating hash ranges (since some clients may discover the new views) and services them after the source indicates that it has stopped servicing requests in the old view.

Transfer: Initiated after all source threads have completed the Prepare phase. The source moves into its new view and stops servicing requests on the migrating hash ranges. When all server threads are in the new view, it sends out a `TransferedOwnership()` RPC to the target asynchronously, which also includes the hot records sampled in the Sampling phase. This moves the target into its Target-Receive phase, whereupon it inserts the sampled records into its `FASTER` instance and then begins servicing requests for the migrating hash ranges. This also triggers the target to service any requests pending from the Target-Prepare phase.

Migrate: Initiated after all source threads have completed the Transfer phase. The source uses thread-local sessions to send records in the migrating hash ranges to the target. Threads interleave processing normal requests with sending batches of migrating records collected from the source's hash table to the target. Each thread works on independent, non-overlapping hash table regions, avoiding contention.

Complete: Initiated after all source threads have completed the Migrate phase. The source sends a `CompleteMigration()` RPC asynchronously, moving the target to the Target-Complete phase. Then, the source sets a flag in the metadata store indicating that its role in migration is complete, and it returns to normal operation.

The target is mostly passive during migration; most of its phase changes are triggered by source RPCs (Figure 3.8). Requests for a record may arrive after a `TransferredOwnership()` RPC is received by the target, but before the source has sent that record. The target marks these requests pending, and it processes them when it receives the corresponding record.

When the target receives the `CompleteMigration()` RPC, it also sets a flag at the metadata store indicating that its role in the migration is complete, and it returns to normal operation.

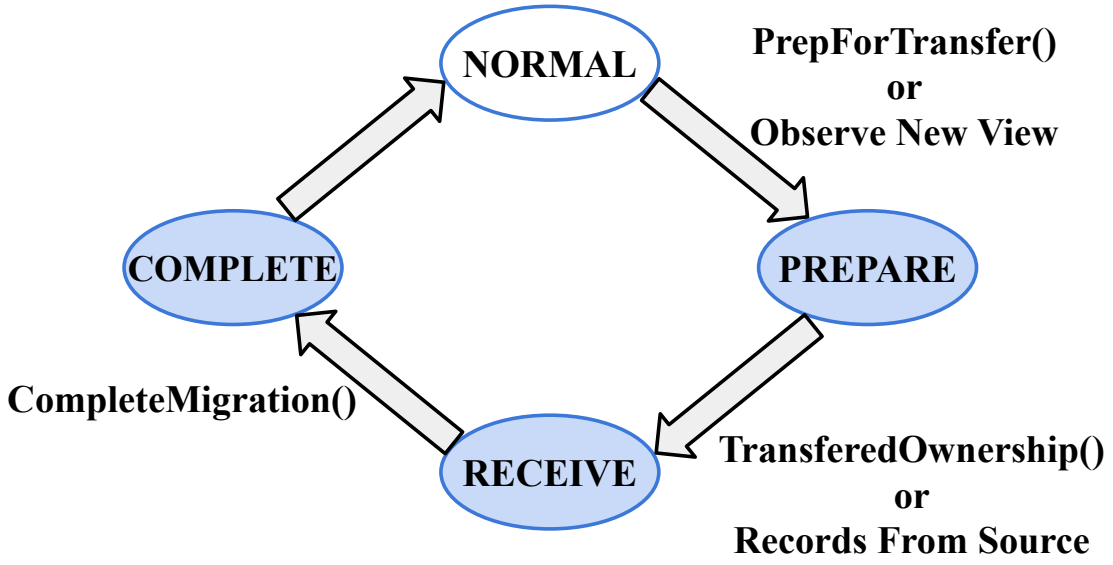


Figure 3.8: Migration state machine on the target. It is responsible for moving the target into the new view, safely executing requests on the migrating hash range, and receiving records from the source

Migration has succeeded once both servers have set their respective flags at the metadata store. A cluster management thread will have to periodically check these flags; on finding both set, it deletes the dependency at the metadata store to complete migration.

Shadowfax maintains high throughput during scale up via low-coordination, non-blocking epoch actions and purely asynchronous inter-machine communication. The source prioritizes request processing, making progress in between request batches. Its state machine transitions are independent of the target; all migration RPCs and checkpoints are asynchronous. The target prioritizes request processing in the same way. Early ownership transfer, sampled records, and pending operations let the target start servicing requests on moved ranges quickly, improving throughput recovery. Sessions let the source collect and asynchronously transmit records in parallel while the target receives and inserts them in parallel.

3.4.2 Leveraging Shared Storage for Decoupling

Migration cannot complete until all records have been moved to the target, so Shadowfax must ensure that this happens quickly. However, `FASTER`'s larger-than-memory index makes this challenging: entries in its hash table point to linked lists of

records, which can span onto local SSD. Performing I/O (sequential or random) to migrate these records can slow migration and hurt throughput.

Shadowfax's shared remote tier helps solve this problem. Records on local SSD are always eventually flushed to this tier, so migration can avoid accessing them. When the source encounters an address for a record in a list that is on the SSD, it sends an *indirection record* to the target that indicates this record's location in the shared tier. This indirection record contains the next address in the list, an identifier for the source's log, the hash range being migrated, and the hash entry that pointed to the list. The target inserts these records into its hash table using the hash entry contained in the record. Overall, these fine-grained inter-log dependencies represented by indirection records accelerate migration completion by eliminating all I/O that would otherwise be needed to consolidate records and transmit them to the target.

During normal operation, if the target encounters an indirection record when processing a request and the request's key falls in the hash range contained in the record, the target asynchronously retrieves the actual record from the shared tier using the contained address and log identifier, inserts it into its hash table, and then completes the request.

3.4.3 Cleaning Up Indirection Records

Migrations can accumulate indirection records between server logs for records that are never accessed (Figure 3.9). On scaling up (①) by moving a hash range from Log 0 to Log 2, Log 2 contains indirection records that point to Log 0 on the shared tier. Dependencies are also created during scale down (②) when records on Log 1 are migrated to Log 2. These dependencies must eventually be cleaned up.

Shadowfax must already periodically do log compaction to eliminate stale versions of records from its shared tier; resolving and removing indirection records can be piggybacked on this process to eliminate overheads for cleaning them (③). When compacting its log, if a server encounters a record belonging to a hash range it no longer owns, the server transmits the record to the current owner. On receiving such a record, the owner first looks up the key. If it encounters an indirection record while doing so and the key falls in the contained hash range, then it means that the key was not retrieved from the

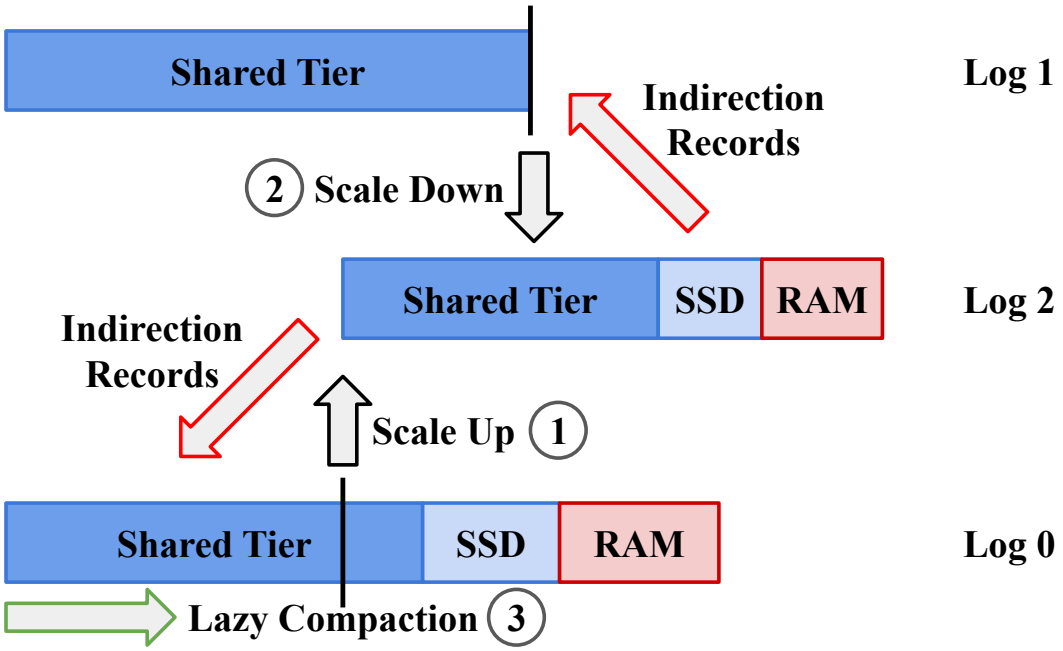


Figure 3.9: Indirection records create fine-grained data dependencies between logs. These dependencies are cleaned up lazily during log compaction.

shared tier after migration. In this case, the server inserts the received record; otherwise, it discards the record.

Barring normal case request processing, this lazy approach ensures that records not in main memory are accessed only once, during the sequential I/O of compaction, which has to be done anyway. It is also deadlock-free: two servers might have indirection records pointing to each others' log, but the resulting dependencies are cleaned up independently.

3.4.4 Fault Tolerance

Migrations in Shadowfax can be easily made fault tolerant. During their respective `Complete` phases in the protocol, the source and target would first have to take a checkpoint before setting their flags at the metadata store. This would make the migration durable; if either machine crashes hereafter, it can be independently recovered from a checkpoint containing the effects of the migration.

If either server crashes during, recovery must involve both, which is why the metadata store tracks the dependency between them. This is because of early ownership transfer; during migration, the target services operations on the migrating ranges, but many records belonging to it may still be on the source. When recovering a server, if Shadowfax finds

a migration dependency involving the server without both completion flags set, it cancels the migration by setting a cancellation flag in the metadata server. Then, it transfers ownership of hash ranges back to the source (incrementing the source and target's view), restores both machines using their pre-migration checkpoints, and recovers requests on hash ranges that were issued during migration at the source.

This cancellation procedure ensures migration is deadlock-free. If either server fails to make progress through the protocol in a timely manner, the migration can always be cancelled by any party, and both servers can be rolled back. No server can stall migration completion indefinitely.

A full description and evaluation of these mechanisms is beyond the scope of this paper, and we leave it to future work.

CHAPTER 4

EXTENSIBILITY AND MULTI-TENANCY

4.1 Introduction

Today's model of separated compute and storage is reaching its limits. Fast, kernel-bypass networking has yielded key-value stores that perform millions of requests per second per machine with microseconds of latency [29, 49, 68, 83, 116]. These systems gain much of their speed by being simple, allowing only lookups and updates. However, this simplicity results in inefficient data movement between storage and compute and costly client-side stalls [7, 76]. To efficiently exploit these new stores, applications will be under increasing pressure to push compute to them, but the granularity at which they can do so is a concern. At microsecond timescales, even small costs for isolation, containerization, or request dispatching dominate, placing practical limits on the granularity of functions that applications can offload to storage.

We resolve this tension in *Splinter*, a multi-tenant in-memory key-value store with a new approach to pushing compute to storage servers. Splinter preserves the low remote access latency (9 µs) and high throughput (3.5 Mops/s) of in-memory storage while adding native-code runtime *extensions* and the dense *multi-tenancy* (thousands of tenants) needed in modern data centers. Tenants send arbitrary type- and memory-safe extension code to stores at runtime, adding new operations, data types, or storage personalities. These extensions are exposed so tenants can remotely invoke them to perform operations on their data. Splinter's lightweight isolation lets thousands of untrusted tenants safely share storage and compute, giving them access to as much or as little storage as they need.

Splinter's design springs from the intersection of three trends: *in-memory storage with low-latency networking*, which is driving down the practical limits of request granularity;

massive multi-tenancy driven by the cloud and the efficiency gains of consolidation; and *serverless computing*, which is already training developers to write stateless, decomposed application logic that can run anywhere in order to gain agility, scalability, and ease of provisioning.

Together, these trends drive Splinter's key design goals:

No-cost Isolation. Since extensions come from untrusted tenants, they must be isolated from one another. Hardware-based isolation is too expensive at microsecond time scales; even a simple page table switch would significantly impact response time and throughput.

Zero-copy Storage Interface. Extensions interact with stored data through a well-defined interface that serves as a trust boundary. For fine-grained requests, it must be lightweight in terms of transfer of control and in terms of data movement. This effectively requires extensions to be able to directly operate on tenant data *in situ* in the store, while maintaining protection and preventing data races with each other and the storage engine.

Lightweight Scheduling for Heterogeneous Tasks.
Extensions are likely to be heterogeneous. Some extensions might involve simple point lookups of data or constructing small indexes; others might involve expensive computation or more data. Preemptive scheduling involves costly context switches, so Splinter must avoid preemption in the normal case, yet maintain it as an option to contain poorly-behaving extensions. It must also be able to support high quality of service under heavy skew, both in terms of the tenants issuing requests at different rates and extensions that take different amounts of time to complete.

Adaptive Multi-core Request Routing. With multiple tenants sharing a single machine, synchronization over tenant state can become a bottleneck. To minimize contention, tenants maintain locality by routing requests to preferred cores on Splinter servers. We can't, however, use a hard partitioning, as we don't want high skew to create hotspots and underused cores [90]. Routing decisions can't get in the way of fast dispatch of requests [8].

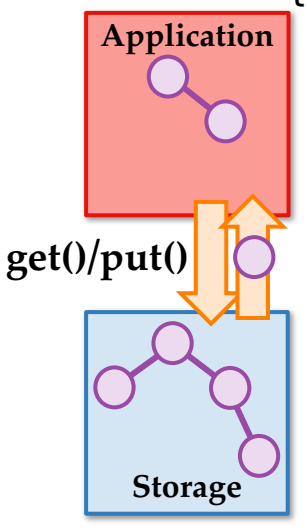
These goals give rise to Splinter's design. Developers write type-safe, memory-safe extensions in Rust [96] that they push to Splinter servers. Exploiting type-safety for lightweight isolation isn't new; SPIN [9] allowed applications to safely and dynamically load extensions into its kernel by relying on language-enforced isolation. Similarly, NetBricks [85] applied Rust's safety properties to dataplane packet processing to provide memory safety between sets of compile-time-known domains comprising network function chains. Splinter combines these approaches and applies them in a new and challenging domain. Language-enforced isolation with native performance and without garbage collection overheads is well-suited to low-latency data-intensive services like in-memory stores — particularly, when functionality must be added and removed at runtime by large numbers of fine-grained protection domains.

Splinter's approach allows it to scale to support thousands of tenants per machine, while processing more than 3.5 million tenant-provided extension invocations per second with a median response time of less than 9 µs. We describe our prototype of the Splinter key-value store and its extension and isolation model. We evaluate it on commodity hardware and show that a simple 800 line extension imbues Splinter with the functionality of Facebook's TAO [11]. On a single store, the extension can perform 3.2 million social graph operations per second with 30 µs average response times, making it competitive with the fastest known implementation [29].

4.2 Motivation

Splinter's key motivation is the desire to support complex data models and operations over large structures in a fast kernel-bypass stores. Existing in-memory stores trade data model for performance by providing a simple key-value interface that only supports `get` and `put`. Many real applications organize their data as trees, graphs, matrices, or vectors. Performing operations like aggregation or tree traversal with a key-value interface often requires multiple `gets`. Applications are usually *disaggregated* into a storage and compute tier, so these extra `gets` move data over the network and induce stalls for each request.

Figure 4.1 illustrates this problem with a storage client that traverses data logically organized as a tree. The client must first issue a `get` to retrieve the tree's root node. Next, it must perform a comparison and move down the tree by issuing another `get`. It must




```
fn find_in_tree(n: &Node, key: u64)
  -> Option<Value>
{
  if n.key == key { // Found correct value
    Some(n.value)
  } else {
    // Traverse left or right
    let next = if key < n.key { n.left }
               else { n.right };
    if let Some(next) = next {
        // Fetch each node from storage
        find_in_tree(get(next), key)
    } else {
        None // Break if dead end
    }
  }
}
```

Figure 4.1: Tree traversal using `get()` operations over a key-value store. Each step requires a lookup at the storage layer, which is latency-bound and expensive for deep traversals. If multi-tenant stores could be safely extended this function could avoid remote access stalls and request processing costs.

repeat this for every step of the traversal. Each `get` incurs a round trip that fetches a single node from storage; since the control flow is dependent on the data fetched, the client can only issue one request at a time. The number of round trips needed is proportional to the tree's depth, and a significant portion of the tree gets moved over the network. Even with modern low-latency networking, latency still dominates the client's performance: network transmission and processing takes tens of microseconds while the actual comparisons take less than a microsecond [83].

One solution is to customize the storage tier of each application to support specialized data types. However, to improve efficiency and utilization, storage tiers are usually deployed as multi-tenant services [17, 26], so they cannot be customized for every possible data structure. SQL could be used at the storage tier, but SQL is known to be a poor fit for data types like graphs and matrices, does not support abstract data types, and is too expensive at microsecond timescales. Instead, Splinter takes a different approach; it allows applications to push small pieces of native compute (extensions) to stores at runtime.

Xeon Architecture	Context switch delay (µs)	
	Pre KPTI	KPTI
D-1548, Broadwell	1.60	2.40
E5 2450, Sandy bridge	1.50	2.48
Gold 6142, Skylake	1.40	2.16

Table 4.1: Context switch overhead for different Intel Xeon architectures as measured on CloudLab. Each number represents the median of a million samples. Based on these measurements, we chose 2.16 µs and 1.40 µs for the context switch overhead with and without KPTI in our simulations.

These extensions can implement richer data types and operators, avoiding extra round trips and reducing data movement.

4.2.1 The Need for Lightweight Isolation

Multi-tenancy at the storage layer makes running extensions challenging; a tenant cannot be allowed to access memory it does not own, starve others for resources, or crash the system. The major challenge is that, at microsecond timescales, context switches and data copying across isolation boundaries significantly hurt performance.

To quantify the overhead of hardware isolation, we simulated an 8-core multi-tenant store that isolates extensions using processes while varying the numbers of tenants making requests to it. Simulated requests consume 1.5 µs of compute at the store; this is based on our benchmarks of simple unisolated operations on Splinter (§4.5.2); our numbers are similar to those reported by others' kernel-bypass stores [83]. Different context switch costs are simulated to show the overheads of hardware-based isolation of tenant code. The simulation only accounts for context switch costs; copying data across hardware isolation boundaries has also been shown to have significant performance costs [85]. Nearly all extensions will access data, which will force data copying when using hardware isolation and hurt throughput further. Based on measurements we made on different processor microarchitectures (Table 4.1), we simulate 1.40 µs of overhead for a basic context switch and 2.16 µs for a KPTI [20] protected kernel (which mitigates attacks that can leak the contents of protected memory [69]). The request pattern is uniform; all tenants make the same number of requests. The results are similar with skew. The simulator is also optimistic; whenever a request is made and an idle core is available at the store that last processed a request from the same tenant, the isolation cost is assumed to be zero.

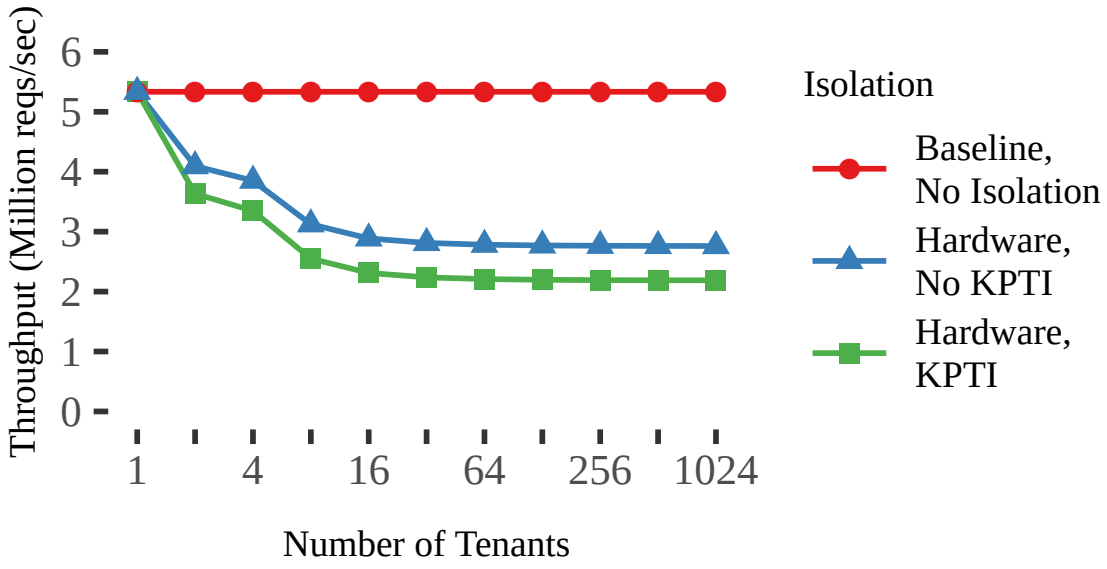


Figure 4.2: Simulated throughput versus the number of tenants. With hardware isolation, even modestly increasing the number of tenants to 16 (just twice the number of cores) leads to a significant drop in throughput. "No isolation" represents an upper bound where isolation costs are zero.

Figure 4.2 presents simulated throughput at different tenant densities. The baseline represents an upper bound where extensions are run un-isolated at the storage system. The simulations show that throughput with hardware isolation (irrespective of KPTI) is significantly lower than the baseline. Even at just 16 tenants, context switch costs alone cut server throughput by a factor of 1.8.

Overall, for these types of fast stores, hardware isolation limits performance and tenant density. The challenges that we face in Splinter, and our design goals, stem from the need to (nearly) eliminate trust boundary crossing costs, to keep data movement across trust boundaries low, and to perform efficient fine-grained task scheduling.

4.3 Splinter Design

Each Splinter server works as an in-memory key-value store (Figure 4.3). Like most key-value stores, tenants can directly `get` and `put` values, but they can also customize the store at runtime by installing safe Rust-based extensions (shared libraries mapped into the store's address space) (Figure 4.3 ①). These extensions can define new operations on the tenant's data, including extensions that stitch together new data models in terms of the store's low-level get/put interface. Each tenant-provided extension is exported over the

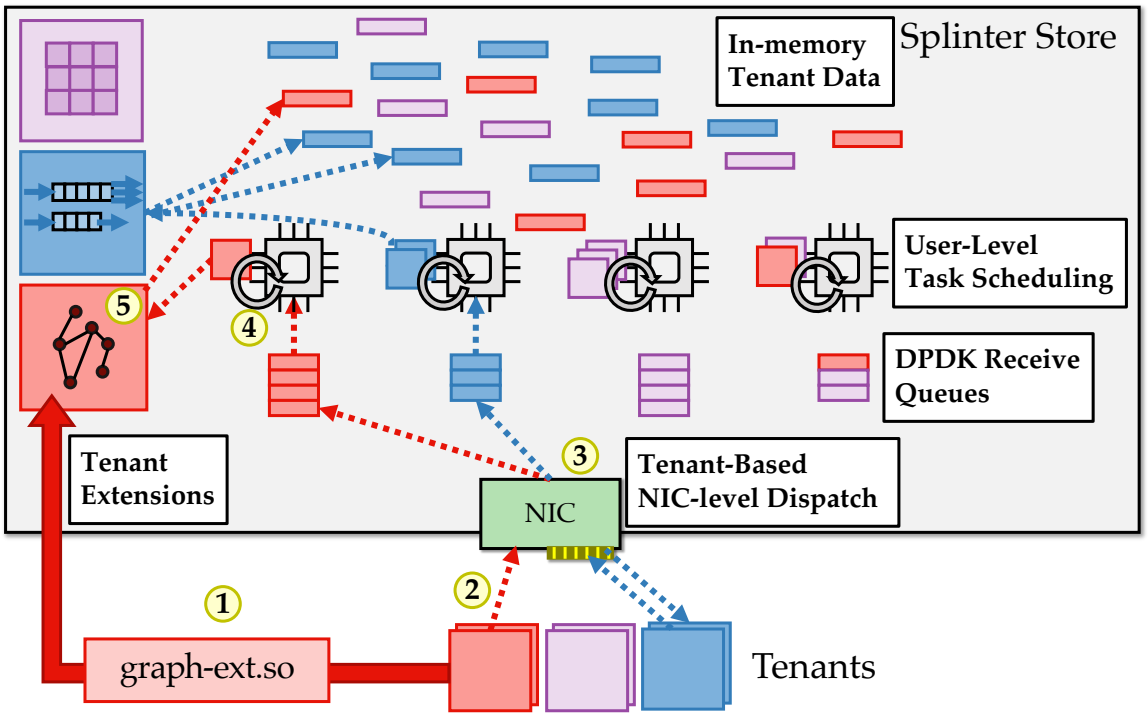


Figure 4.3: Overview of Splinter. Tenant data is stored in memory, and tenants can invoke extensions they have installed in the store (①). Extensions are type safe, but compile to native code. The NIC uses kernel bypass for low latency (②) and assists in dispatch by routing tenant requests to cores (③). Each core runs a single *worker* kernel thread that uses a user-level task scheduler to interleave the execution of tenant requests (④).

network, so a tenant can remotely invoke the procedures it has installed into the store.

Tenants send requests to a Splinter store over the network using kernel bypass (②). Splinter currently only supports a simple, custom UDP-based RPC protocol, though other optimized transports may provide similar performance [50]. Each tenant's requests are steered to a specific receive queue by the network card, improving locality (③). Each receive queue is paired with a single kernel thread (or *worker*) that is pinned to a specific core. Each worker pulls requests from its receive queue and creates a user-level task for the requested operation. Tasks provide an accounting context for resources consumed while executing the operation, the storage needed to suspend/resume the operation, and a unit of scheduling. Each worker has a task queue of new and suspended tasks, and it schedules across them to make progress in processing the operations (④). Scheduling is cooperative; as tasks yield and are resumed, they store/restore their state, so when a worker schedules a task no stack switch is performed. As tasks execute user-provided logic, they interact with the store through a `get`/`put` interface similar to the one exposed remotely (⑤); the

key difference is that the functions exposed to extensions take and return references rather than forcing copies (Table 4.2).

Beyond fast kernel-bypass network request processing, Splinter's speed depends on exploiting the Rust compiler in two key ways: first, to enable low-cost isolation and, second, to enable low-cost task switching. The two are intertwined. Splinter uses stackless generators to suspend and resume running extensions, which require compiler support. That is, the Rust compiler analyzes extension code, determines the state that needs to be held across extension cooperative-yield/resume boundaries, and generates the code to suspend and resume extension operations. No separate stack is needed, and the code needed to yield/resume is transparent to the extension.

These lightweight tasks are key, but Splinter's careful attention to object lifetimes, ownership, and memory safety make them effective, since otherwise full context switch would be needed between tasks for isolation. A key challenge in Splinter is ensuring its fine-grained tasks from different trust domains—compiled to native code, and mapped directly into the store's memory—remain low-overhead while still operating within Rust's static safety checks. Low-overhead trust boundary crossings are essential to Splinter's design; they enable easy and inexpensive task switching, dispatch (§4.3.3), and work stealing (§4.3.4), which keep response latency low and CPU utilization high across all the cores of the store.

Another key challenge is that extension invocations introduce more irregularity into request processing than a simple get/put interface. By avoiding hardware context switches, Splinter keeps task switch costs down to about 11 nanoseconds, but the difficult tradeoff is that this forces it to handle these variable workloads without traditional preemptive scheduling. At the same time, it cannot use fully cooperative scheduling, since the store does not trust tenants to supply well-behaved extensions. Splinter's per-worker task scheduler resolves this tension by multiplexing long-running and short-running tasks to build mostly-cooperative scheduling. This is backed up by having an extra thread that acts as a watchdog for the others to support preemption when needed.

4.3.1 Compiling and Restricting Extensions

The Splinter store cannot directly load native code provided by tenants. Code must be compiled and type checked to ensure its safety before it can be loaded into a store, and extensions face some extra restrictions that must be enforced at compile time. The compiler is trusted and must be run by the storage provider. Tenants must not be able to tamper with the emitted extension, so it must be loaded directly into the store by the provider or the provider must ensure its integrity in transit between the trusted compiler and the store. Aside from Rust's standard type and lifetime checks (§4.3.1.2), Splinter extensions have the following static restrictions:

No Unsafe Code. Unsafe code could skip compiler checks resulting in memory unsafety. So, our wrapper over `rustc` disallows unsafe code in extensions (§4.3.1.3).

Module Whitelist. Code from external dependencies could include unsafe code, and that unsafe code shouldn't be incorporated into untrusted extensions unless it is trusted. Even beyond memory safety, such unsafe blocks could, for example, make syscalls. So, our wrapper restricts external dependencies to modules that are re-exported by a Splinter library that includes many standard functions and types. This restriction applies to the standard library (`std`) as well: the wrapper only exposes whitelisted `std` functionality to extensions.

These checks combine with three other runtime guarantees to ensure isolation: the store only accepts or provides references to insert/fetch a value under a key if the same tenant owns both the extension and the key (§4.3.2); it prevents uncooperative extensions from dominating CPU time and stack, heap, or record memory (§4.3.3); and it catches panics (runtime exceptions) and stack overflows that occur while executing an extension operation (§4.3.3). Next, we describe what guarantees this gives the storage provider and its tenants; the runtime checks are described later along with details about the execution model.

4.3.1.1 Trust Model

There are two stakeholders for a Splinter store: the storage provider and storage tenants. Splinter should protect tenants from each other and the provider from the tenants.

Tenant misbehavior could be unintentional, in the form of bugs or unexpectedly high application load, or it could be malicious, in the form of tenants attempting to read others' data, deny service, or use an unfair fraction of resources. We consider threats from "within" the store; threats from "without" such as an attacker gaining root access to the machine by exploiting other services running on it should be dealt with using standard security best practices.

Aside from providing good quality of service to tenants, service providers have one key concern: protecting the secrecy and integrity of tenants' data. Extensions don't share state with one another, and Splinter provides no means for inter-extension communication. So, no complex sharing policies are needed; Splinter's only goal is extension isolation. Rust references act as capabilities; they ensure that extensions cannot fabricate arbitrary references to storage state or to other tenants' state (§4.3.1.2).

Like any database, Splinter's Trusted Computing Base (TCB) includes the libraries, compilers, hardware, etc. on which it is built; while this code is not directly exposed to tenants, vulnerabilities in it can still lead to exploits. Dependencies include LLVM [61], the CPU, the network card (NIC) and its kernel-bypass libraries (DPDK [27]).

Splinter's design provides a larger attack surface relative to other databases in some ways, but decreases the attack surface in others. Because it allows execution of tenant code, Splinter's safety depends on the soundness of Rust's type system, which is not proven. While some soundness issues in the compiler have been found [45], progress is being made in proof efforts [46], and Splinter automatically benefits from such progress. If extensions cannot violate Rust's safe types, the remaining avenue for attack is unsafe code in the system; extensions cannot supply unsafe code, but they can indirectly call it in the interfaces and libraries that Splinter explicitly exposes to extensions. On the plus side, extensions *must* break one of these layers of protection before they can attack other code: they do not have direct access to system libraries, system calls, etc. and can only gain it by breaking out of Rust's safe environment.

Splinter decreases the attack surface with respect to the virtual memory system – both hardware and kernel components. Because it doesn't rely on virtual address translation for isolation, recent Meltdown speculation attacks don't affect its design [69]; however, Spectre-based speculation attacks do affect Splinter [55, 56]. Like any system that runs

untrusted code or operates on untrusted inputs, Splinter would require special steps to mitigate these side channels. It already limits them in part because it doesn't provide explicit timing functions to extensions. Full protection will require compiler support [14], hardened storage interfaces (like the Linux kernel [21]), and hardened libraries for extensions. The measurements in this paper do not include these mitigations.

4.3.1.2 Memory Safety

Rust's memory safety (and data race freedom) is guaranteed through a strong notion of *ownership* that lets the `rustc` compiler reason statically about the lifetime of each object and any references to it. The compiler's *borrow checker* statically tracks where objects and references are created and destroyed. It ensures that the lifetime of a reference (initially determined by its binding's scope) is subsumed by the lifetime of its referent. Rust separates immutable and mutable references; an immutable reference is a reference that when held restricts access to the underlying object to be read-only. The compiler disallows multiple references (of either type) to an object while a mutable reference exists, which prevents data races.

Often, the lifetime of an object cannot be restricted to a single, static scope. This is especially true in a server that processes requests across threads, where the lifetime of many objects (RPC buffers, extension runtime state) is defined by request/response. Rust provides various accommodations for this, such as moving ownership between bindings and runtime reference counting that is safe but implemented in unsafe Rust. Splinter efficiently handles these issues while working within `rustc`'s static safety checks (§4.3.2.2). Unlike C/C++ pointers, Rust references cannot be fabricated or manipulated with arithmetic; they always refer to a valid, live object. Rust supports pointers but their use is restricted for safety.

4.3.1.3 Restricting Unsafe Rust

An important extra restriction that Splinter imposes beyond Rust is that extension code must be free from *unsafe* Rust, a superset of the language that allows operations that could violate its safety properties. For example, unsafe code can dereference pointers, perform unsafe casts, omit bounds checks, and implement low-level synchronization primitives. All unsafe code in Rust requires an `unsafe` block, which Splinter disallows in extension

Store Operations for Extensions
get(table: u64, key: &[u8]) → Option⟨ReadBuf⟩ Return view of current value stored under ⟨`table`, `key`⟩.
alloc(table: u64, key: &[u8], len: u64) → Option⟨WriteBuf⟩ Get buffer to be filled and then `put` under ⟨`table`, `key`⟩.
put(buf: WriteBuf) → bool Insert filled buffer allocated with `alloc`.
args() → &[u8] Return a slice to procedure args in request receive buffer.
resp(data: &[u8]) Append data to response packet buffer.

Table 4.2: Extensions interact with the store locally through an interface designed to avoid data copying.

code.

Extensions cannot implement unsafe code, but they can invoke it indirectly. This is often desired. For example, extensions execute some unsafe code when they ask the store to populate a response packet buffer. In some cases it is not desired. For example, file I/O can be induced through the Rust standard library. To prevent this, Splinter restricts extensions to use a subset of the standard library that doesn't include I/O or OS functionality.

Our experience has been that safe Rust combined with basic data structures from its standard library are sufficient to write even complex imperative extensions like Facebook's TAO [11]. In cases where unsafe code could provide a performance benefit, the store can provide that functionality if it is deemed safe to do so, since it is trusted and can include `unsafe` code (§4.3.2.3).

4.3.2 Store Extension Interface

The interface that extensions use on the server to interact with stored records is similar to the external, remote interface that clients use in any conventional key-value store (Table 4.2). The main differences are in careful organization to eliminate the need to copy data between buffers.

All persisted records are stored in a *table heap*. Keeping records in a identifiable region

will be essential to support replication, recovery, and garbage collection as Splinter's implementation evolves.

4.3.2.1 Storing Values

Extensions can `put()` data they receive over the network or new values that they produce into the store. When an extension invocation request is received from a tenant, the store invokes the indicated operation. Incoming data is in a packet buffer that is registered with the NIC. Those buffers cannot be used for long-term storage because the NIC must use them to receive new requests; data that must be preserved needs to be copied into the store.

Splinter tries to ensure that data can be moved from NIC buffers into the store with a single copy. This requires `put()` to be split into two steps. First, an extension calls `alloc(table, key, length)` to allocate a region in the table heap for a record. The extension receives a bounded slice (a view) to the underlying allocated memory. Then, it copies data from the request's receive buffer, unmarshalling as it does so, if needed. Extensions use `args()` to directly access data (by reference) in the receive buffer to perform this copy. An extension may produce its own data values as part of this process either from input arguments or together with values read from the store. Once the allocated region is properly populated, it is inserted into the table with `put()`, which takes ownership of the buffer and inserts it into a hash table.

Problems like use-after-free are prevented by Rust's borrow checker; extensions cannot hold references to a buffer once ownership is transferred to the store, eliminating the need for copying data into the store for safety. The receive packet buffer has the same guarantee. Rust's borrow checker ensures references to it cannot outlast the life of the RPC, eliminating the need to copy received arguments or data into the extension for safety.

Values stored by `put()` must be allocated from the table heap; extensions should not be able to pass arbitrary (heap or stack allocated) memory to `put()`. Splinter enforces this so that it can optimize record layout; keys and values can be forced into a single table heap allocation, which eases heap management and eliminates cache misses for hash table lookups. As a result, Splinter wraps allocations with a type (`WriteBuf`) that extensions cannot construct, ensuring they can only pass buffers acquired from `alloc()`. `WriteBuf`

```
fn aggregate(db: Rc<DB>) {
  let mut sum = 0u64;
  let mut status = SUCCESS;
  let key = &db.args()[..size_of::<u64>()];

  if let Some(key_lst) = db.get(TBL, key) {
    // Iterate KLEN sub-slices from key_lst
    for k in key_lst.read().chunks(KLEN) {
      if let Some(v) = db.get(TBL, k) {
        sum += v.read()[0] as u64;
      } else {
        status = INVALIDKEY;
        break;
      }
    }
  } else {
    status = INVALIDARG;
  }
  db.resp(pack(&status));
  db.resp(pack(&sum));
}
```

Figure 4.4: Example aggregate extension code. The extension takes a key as input (directly from a request receive buffer), looks it up in the store, and gets a reference to a value that contains a list of keys. It looks up each of those keys, it sums their values, and directly appends the result to a response buffer.

has a method to get a reference to the underlying buffer, so extensions can fill it.

4.3.2.2 Accessing Values

Extensions can interact with stored data in a similar way, requiring only one copy into a response buffer to return values from the store. When an extension procedure is invoked, it is also provided with a response buffer that can be incrementally filled via `resp()`. On each extension procedure invocation, the store pre-populates the response buffer's packet headers; extensions can only append their data after these headers. All response buffers are pre-registered with the NIC for transmission.

Extensions call `get(table, key)`, and they receive back a reference to the underlying portion of the table heap that contains the value associated with `key`. No copying is needed at this step; the store tracks this reference and prevents the table heap garbage collector from freeing the buffer while an extension has a live reference to the data. Since values

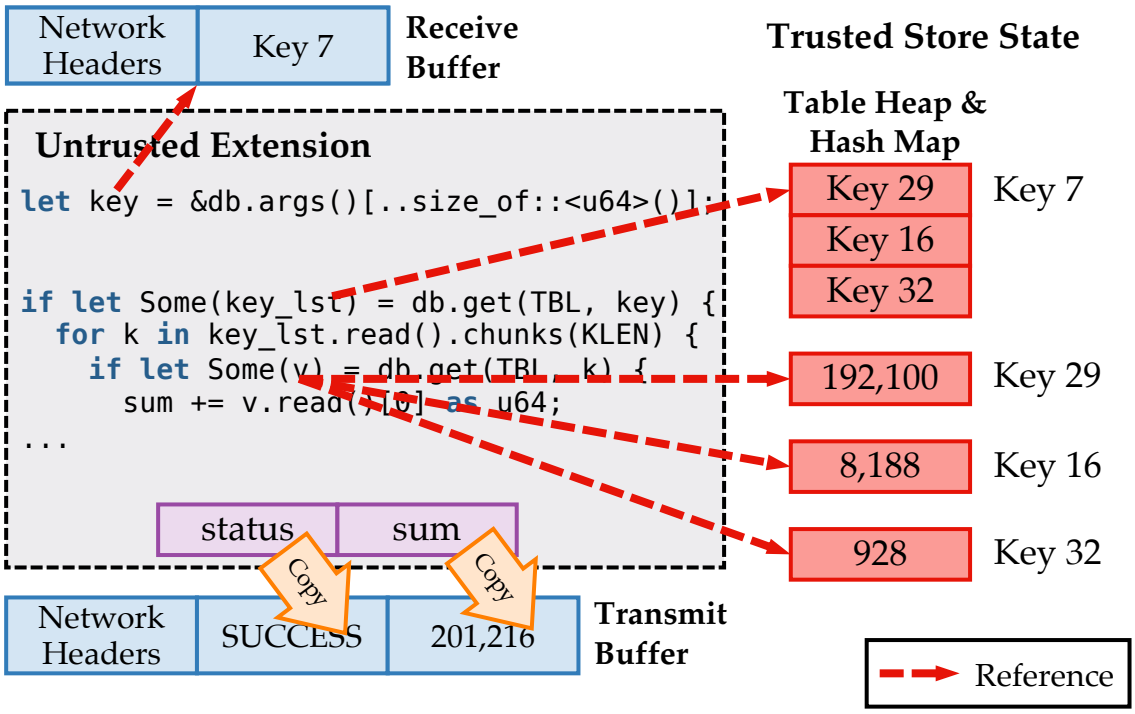


Figure 4.5: References during aggregation. All data accessed by the extension in Listing 4.4 is by reference whether that data is part of the arguments in the receive buffer or part of a record in the store. References work in reverse for the response; the extension passes references to data to the store, and the store copies that data into the response buffer.

are never updated in place, extensions see stable views of values. Extensions can compute over the value or many values concurrently (by calling `get()` multiple times), and they can copy portions of the data they observe or any results they compute directly into the response buffer. Once the extension procedure has populated the response buffer, Intel's DDIO [24] transmits the data directly from the L1 cache, which avoids the cost of memory access for DMA of stored data.

Listing 4.4 and Figure 4.5 show an example of how this works for a simple extension that sums up a set of values stored under keys that are listed as part of another stored value without any extra data copying. In Line 4, the extension obtains a reference to its transmit buffer to find which key it should look up in order to find a list of keys that will be aggregated over. Line 6 passes a reference to that same location to the store in order to obtain a reference to the value that contains the key list. In Line 8, still without copying, the extension iterates over that value in chunks equal to the length of the keys stored in the value. Each step of the iteration produces a reference that the extension uses to `get()`

references to values for each of the stored keys, one at a time (Line 9). Using each of those references, it extracts a field that it adds to `sum`, a local variable. Finally, the extension passes references to `status` and `sum` to append them to the response buffer. In all, data copying is only forced where it is needed, so the compiler has flexibility in optimizing extension code.

The store's `get()` call returns a `ReadBuf` rather than a plain slice (`&[u8]`) in order to satisfy Rust's borrow checker. Calling `get()` cannot return an immutable reference or slice to a stored value, because the borrow checker wouldn't be able to statically verify that the reference would always refer to a valid location. For example, the compiler couldn't be sure that the store wouldn't garbage collect the value while the reference still exists. Furthermore, extension invocations are generators, and they must yield regularly (§4.3.3). Yielding marks the end and start of a new static scope, so each time the generator is resumed, the calling scope could vary. Any obtained references to a stored value couldn't be held across yields, because the borrow checker wouldn't be able to verify that those references would still be valid on reentry.

The `ReadBuf` returned by `get()` solves this. It is a smart pointer that maintains a reference count to ensure the underlying stored object isn't disposed, and it allows the extension code to (re-)obtain a reference to the underlying object data. Once a `ReadBuf` is returned to a generator, it is stored within the generator's local state, so the generator owns this `ReadBuf`. Extensions cannot hold references between yields, but by working with the `ReadBuf` it can (transparently) re-obtain a reference to the data without performing another `get()`. Rust's `Arc` smart pointer does the same; `ReadBuf` hides its constructor from extensions and disallows duplication. This prevents extension code from creating `ReadBuf`s that persist beyond the life of a single request/response, which could otherwise hold back table heap garbage collection.

4.3.2.3 Avoiding Serialization and De-serialization

Allowing extensions to interact directly with receive buffers, transmit buffers, and table heap buffers eliminates copying for opaque data, but Rust's safety makes avoiding some copies harder. Extensions cannot perform unsafe operations, otherwise they could thwart Rust's memory safety guarantees. Unfortunately, this means safe Rust code cannot cast an

opaque byte array to/from different types to avoid the need to serialize/de-serialize data. For example, if `args()` returned an 8-byte slice an extension may desire to treat that slice data as a 64-bit unsigned value. Safe Rust disallows this.

For small arguments, extensions can convert between formats with arithmetic, but for richer data models, arguments, stored values, and responses will have more complex, structured formats. To accommodate this, Splinter's interface provides a mechanism for extension code to convert between byte slices and references to a small set of types. If a slice (`&[u8]`) is naturally aligned to the desired type, Splinter allows conversion to a reference of that type (`&T`), where `T` is limited to signed/unsigned integers and compound types built from them.

These casts are safe, but they are meaningless across architectures. As a result, they can only be used between a client and the store when they have the same underlying platform (e.g. x86-64). Similarly, they can only be used with extensions' `get/alloc/put` interface if all stores in the system (e.g. before/after recovery, source/destination for migration) have matching hardware platforms.

4.3.3 Cooperatively Scheduled Extensions

Splinter is designed to work well regardless of whether tenant-provided extensions are short and latency-sensitive or long-running and compute- or data-intensive. In fact, the best mix of tenants will mix these operations, keeping CPU, network, and in-memory storage better utilized than would be possible with a single, homogeneous workload. Even so, latency-sensitive operations can easily suffer under interference from heavier operations.

This means Splinter must multiplex execution of tenant extension invocations not only across cores but also within a core. Long-running procedures cannot be allowed to dominate CPUs, but preemptive multitasking is too costly even when page table switching can be avoided.

Rust's lightweight isolation is part of the solution, since calls across trust domains have little overhead. Splinter already relies on `rustc` for safety, but it can also rely on it to help minimize task switching costs. When a new request comes into the store, Splinter calls into the responsible extension to allocate a stackless coroutine (a generator) that closes over the

state needed to process the request. Generators support a `yield` statement that suspends execution and enables cooperative scheduling; extension code is expected to periodically call `yield` to allow other tasks to run. `rustc` produces generators specific to the extension, so the cost to create them and switch between them is low. Splinter invokes the created generator. Whenever it yields, Splinter's per-core task scheduler runs another generator task. Since yielding requires no costly hardware boundary crossing and no stack switch, it is fast and inexpensive to yield frequently.

Like other similar systems, to avoid jitter due to kernel thread context switches and migrations, Splinter runs the same number of worker threads as cores in the system (Figure 4.3), and each is pinned to a specific core. Generators are invoked on the worker's stack, avoiding a stack switch. Note that the compiler generates the structure to hold a suspended task's state across yields. Consequently, a worker's stack never concurrently contains state for different tenants (or even tasks); furthermore, whenever a task yields or completes, the worker's stack contains no extension state. This makes it easier to handle uncooperative extensions (§4.3.3.1) and load imbalance (§4.3.4).

4.3.3.1 Uncooperative and Misbehaving Extensions

All calls through the store interface include an implicit yield, so extensions can only dominate CPU time with infinite or compute-intensive loops. Nonetheless, such behavior can disrupt latency-sensitive tasks and constitute a denial-of-service attack in the limit.

To solve this, Splinter uses ideas from user-level threading for latency-sensitive services [91] and adapts them for untrusted code. An extra (mostly idle) thread acts as a watchdog. If a task on a core fails to yield for a few milliseconds, the watchdog remedies the situation. First, the worker thread on the core with the uncooperative task is re-pinned to a specific core that is shared among all misbehaving threads and low-priority background work that the store performs. Second, a new worker kernel thread is started and pinned to the idle core left behind after the misbehaving thread was re-pinned. Finally, the new worker steals the tasks remaining in the scheduler queue for the re-pinned worker and resumes execution for these tasks. Note, this is safe in part because all of the state of a suspended task is encapsulated. Tasks only have state on a worker's stack if they are running, so the misbehaving task is the only one the new worker cannot steal. Whenever

a misbehaving task finally yields, the scheduler on that worker realizes that it has been displaced, and the worker thread terminates along with the task.

Hence, misbehaving tasks don't block other requests, but they can still cause disruption. Creating and migrating kernel threads is expensive, so there must be a disincentive against forcing watchdog action. Tenants that run uncooperative tasks will experience poor quality of service, since they must share a core with other disruptive work. Furthermore, when a worker is re-pinned the watchdog also takes away access to its receive and transmit queues, so tenants cannot get responses from bad requests and, thus, benefit from their misbehavior. Even so, billing policies should ensure such behavior is unprofitable.

Aside from infinite loops, the store must also protect against other things that cannot be prevented with compile-time checks. For example, Rust doesn't have general exceptions, but extensions can raise exceptions with operations like division by zero that raise a `panic`. Splinter must "catch" these panics or they would terminate the worker, since panics unwind the call stack and worker threads call extension code on their own stack. Fortunately, Rust provides a mechanism to do this, and Splinter catches panics and converts them to an error response to the appropriate client. Stack overflows and violation of heap quotas are handled similarly.

4.3.4 Tenant Locality and Work Stealing

The Splinter store avoids any kind of centralized dispatch core to route requests to cores, since this can easily become a bottleneck [83]. At the same time, it needs to balance requests across cores, while still trying to exploit locality to avoid cross-core coordination overheads. To do this, clients route each tenant's requests to a particular core. This provides cache locality, it reduces contention, and it improves performance isolation. Splinter configures Flow Director [43] so that the NIC directly stores packets with a specific destination port number in a specific receive queue. Each receive queue is paired to a single task dispatcher owned by a worker thread (pinned to a core). As a result, tenants can steer requests to specific cores by placing their tenant id in the UDP destination port field.

However, this approach alone can leave cores idle under imbalance, and, as a multi-tenant store, it is important for the system to deliver good resource utilization. Whenever

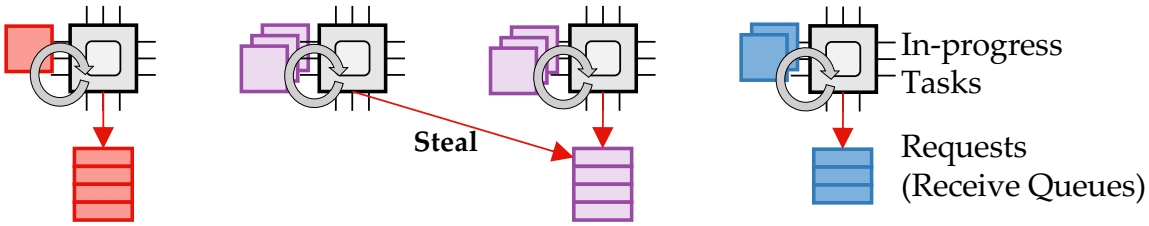


Figure 4.6: Dispatch tasks on each core steal requests from the receive queue of the core to their right whenever they have no requests in their own receive queue. As a result, work from overloaded cores get redistributed without generating high contention. Here, core 1's in-progress tasks were induced by requests stolen from core 2's queue.

the scheduler on a core has no incoming requests in its local receive queue, it attempts to steal requests from a neighbor's receive queue (Figure 4.6). Transmit queues aren't bound to specific (server-side) source ports, so the response can be sent directly from the core that stole the request. This simple form of soft affinity works well, and, since tasks are lightweight, it is also relatively easy for Splinter to take advantage of idle compute in the system without costly thread migration.

4.4 Implementation

The Splinter store is implemented in 7,500 lines of Rust. It uses the NetBricks network function virtualization framework [85] as a wrapper over the DPDK [27] packet processing framework. Splinter also includes 1,100 lines of Rust that provide the store interface to extensions. Extensions import it and compile against it. The store also imports the interface, since it defines how the store interacts with extensions to create a new generator for an invocation. The Splinter codebase is open sourced and freely available on github at the following link: `https://github.com/utah-scs/splinter`.

The store needn't be written in Rust, but doing so has advantages. It prevents data races and segmentation faults within the store, but it also lets the store use Rust's type system and lifetimes to ensure that mistakes aren't made with lifetimes of objects and references handed across trust boundaries, which an adversary could exploit.

4.5 Evaluation

We evaluated Splinter on five key questions:

1. What is Splinter's isolation overhead?

CPU	2×Xeon E5-2640v4 2.40 GHz 10 cores (20 hardware threads) per socket
RAM	1 TB 2400 MHz DDR4
NIC	Mellanox CX5, 40 Gbps Ethernet
OS	Ubuntu 16.04, Linux 4.4.0-116, DPDK 17.08, 16×1 GB Hugepages, Rust 1.28.0-nightly

Table 4.3: Experimental configuration. Evaluation used one machine as server and one as client. Only the NIC-local CPU socket was used on the server.

2. Does Splinter support high tenant densities?

3. How does Splinter perform under operations with heterogeneous runtimes?

4. Do representative extensions see latency and throughput benefits?

5. When does performing operations client-side outperform extension-based operations?

4.5.1 Experimental Setup

All evaluation was done on two machines consisting of one client and one storage server on the CloudLab testbed [93] (Table 4.3). Both used DPDK [27] over Ethernet using Mellanox NICs for kernel-bypass support. The server was configured to use only one processor socket; out of the ten hardware cores, eight were used for request processing, one was used for management and to detect misbehaving extensions, and the last one was used to hold all misbehaving extensions once detected.

To evaluate Splinter and its isolation costs under high load and density, the client ran a YCSB-B workload [18] (95% gets, 5% puts; keys were chosen from a Zipfian distribution with $\theta = 0.99$) that accessed tenant data on the storage server. Unless stated otherwise, the client simulates 1,024 total tenants. Tenant ids for each request were chosen from a Zipfian distribution with $\theta = 0.1$ (unless stated otherwise) to simulate some tenant skew. Each simulated tenant owns one data table consisting of 1 million 100 B record payloads with 30 B primary keys (totaling about 120 GB of stored data). The client always offered an open-loop load to the server.

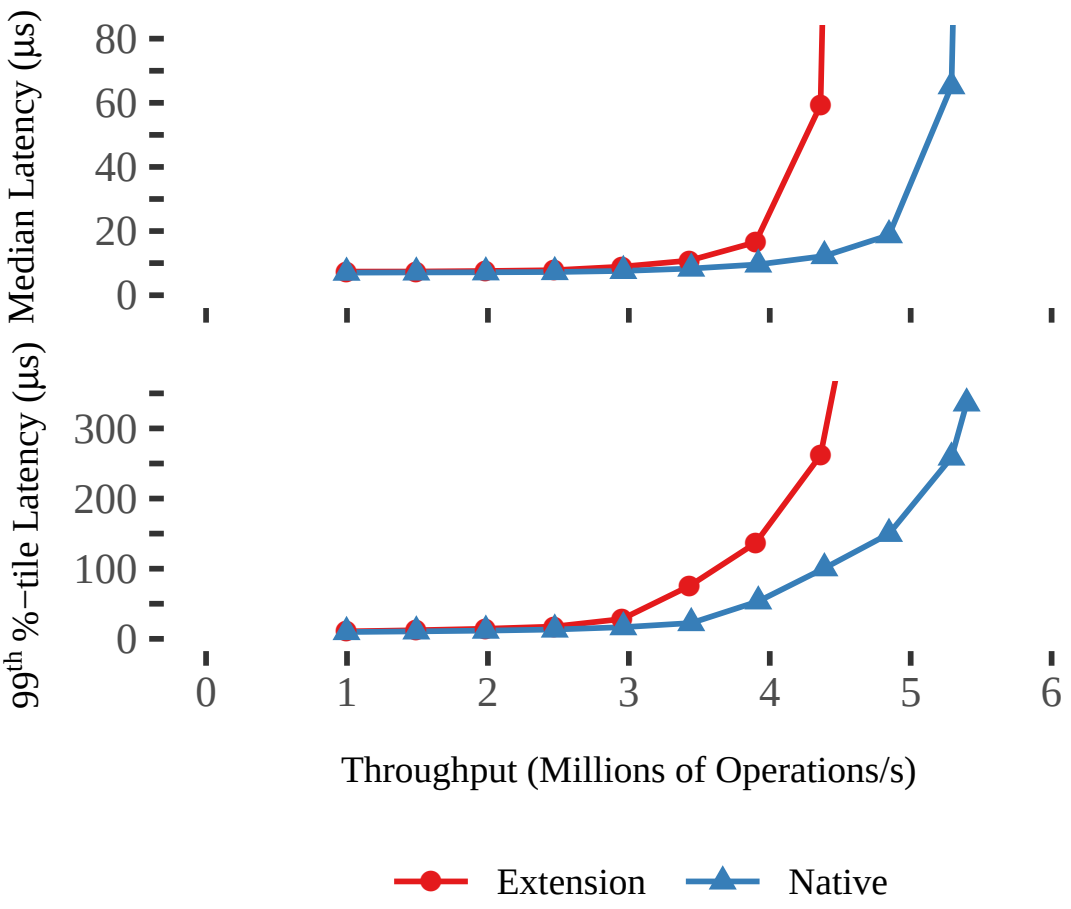


Figure 4.7: Comparison of YCSB-B performance using native and extension-based `get()` and `put()` operations at a tenant density of 1,024. When using extensions, the server saturates at 4.3 million operations per second. In comparison, native operations are about 23% more efficient, saturating at 5.3 million operations per second.

4.5.2 Isolation Overhead

Figure 4.7 compares the performance of YCSB-B under two different cases. In one case ("Native"), the Splinter store executes get and put operations like any other key-value store would; none of Splinter's extension functionality is used. This case sets an upper-bound for Splinter's performance. In the other case ("Extension"), that same get or put is executed as part of a tenant-provided and untrusted Splinter extension. This teases apart the isolation and dispatch costs for Splinter to run arbitrary tenant-provided logic. For offered loads of less than 3.5 million operations per second (Mops/s), median latency with and without isolation are nearly identical (about 9 µs).

Splinter extensions have some overhead, so the store saturates earlier when gets/puts

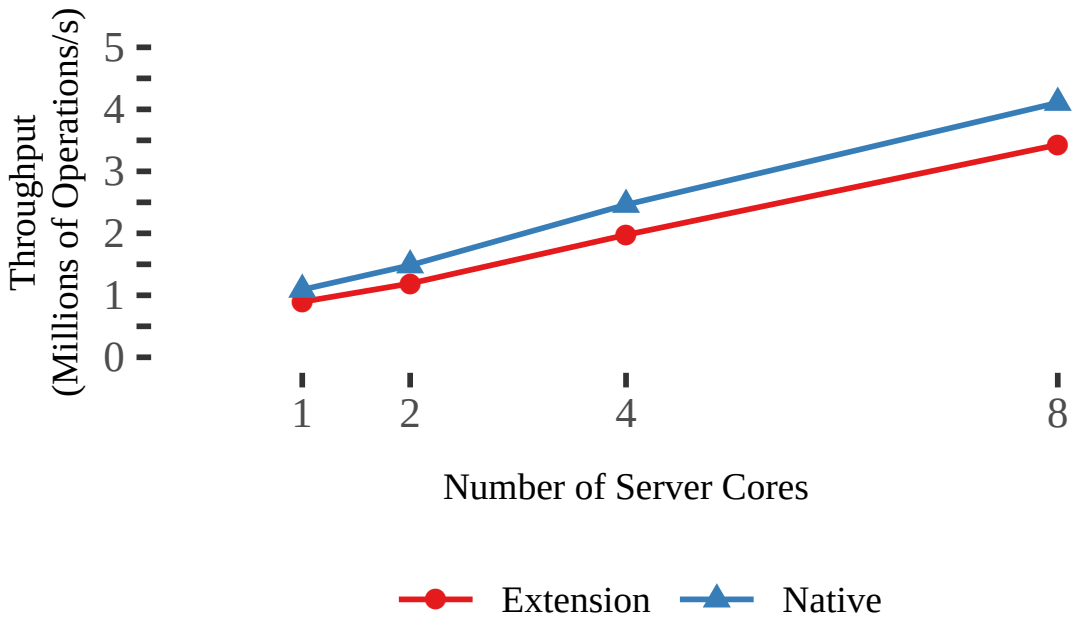


Figure 4.8: Storage server scalability at a tenant density of 1,024. Points represent throughput when YCSB-B latency crosses 10 µs. Isolation overhead is consistently lower than 20%.

are executed through extensions. With isolation, the median latency spikes above 4 Mops/s, reaching 59 µs at 4.3 Mops/s. Without isolation, this spike comes at 5.3 Mops/s. Tail latency (99^{th}-percentile) begins to show a difference at 3 Mops/s. On the whole, in this pessimal workload with extremely fine-grained operations all invoked as extensions, Splinter's isolation costs still only impact throughput of the store by about 19%. Compared to the 1.8× (simulated) penalty for hardware-based isolation in Figure 4.2, this is a significant improvement (a 1.2× penalty over native get/put).

Figure 4.8 compares YCSB-B scalability when the server is approaching saturation (median latency $>$ 10 µs) under the native and extension-based cases. Invoking get and put operations from extensions instead of directly has no impact on scalability; scalability is near linear in both scenarios. However, as pointed out above, it does affect throughput. At one core, throughput is reduced by 200 Kops/s (18%), while at eight cores, the reduction is 700 Kops/s (17%). This shows that, though extensions do increase the number of cycles each core spends processing requests, it doesn't come at the cost of significant increased coordination between the cores.

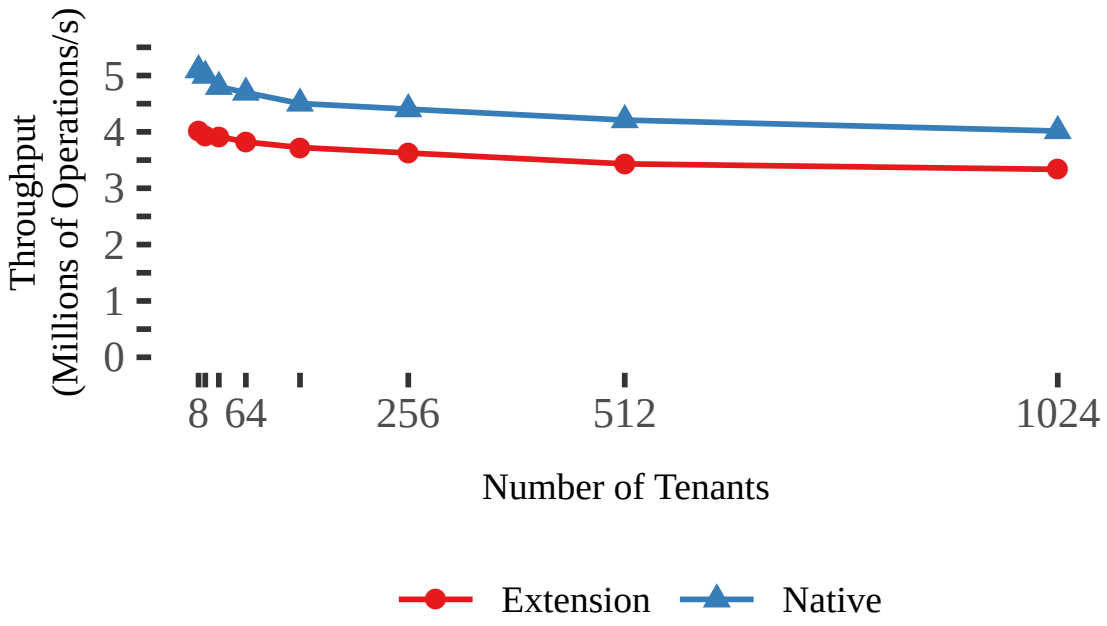


Figure 4.9: Scaling tenants. Points represent server throughput when YCSB-B latency crosses 10 µs. With isolation, increasing the number of tenants only impacts performance modestly; moving from 8 to 1,024 tenants reduces throughput by 700 Kops/s.

4.5.3 Tenant Density

Figure 4.9 shows how varying the number of tenants sharing the store impacts its throughput. As in the prior experiments, tenants run YCSB-B under two cases: without isolation ("Native") and with isolation ("Extension"), so the experiment captures extension isolation overheads. The results show that Splinter can efficiently support high tenant densities with minimal overhead. With isolation, the throughput at 1,024 tenants is 3.3 Mops/s, only 700 Kops/s less than the throughput at 8 tenants. Additionally, the throughput with isolation is consistently within 22% of the throughput without isolation.

In practice, offered tenant load will be skewed, since some tenants are likely to have heavier workloads than others. This results in a few heavy workloads that must share the store with a long tail of many more passive ones. We ran an experiment to show that Splinter can handle this imbalance and that its work stealing and tenant locality help maintain Splinter's response times under high load.

Recall that Splinter routes requests for a tenant to a specific core, but cores steal work from each other to combat imbalance. To gauge the benefits of this approach, we compare it against a tenant-partitioned approach with no work stealing and an unpartitioned

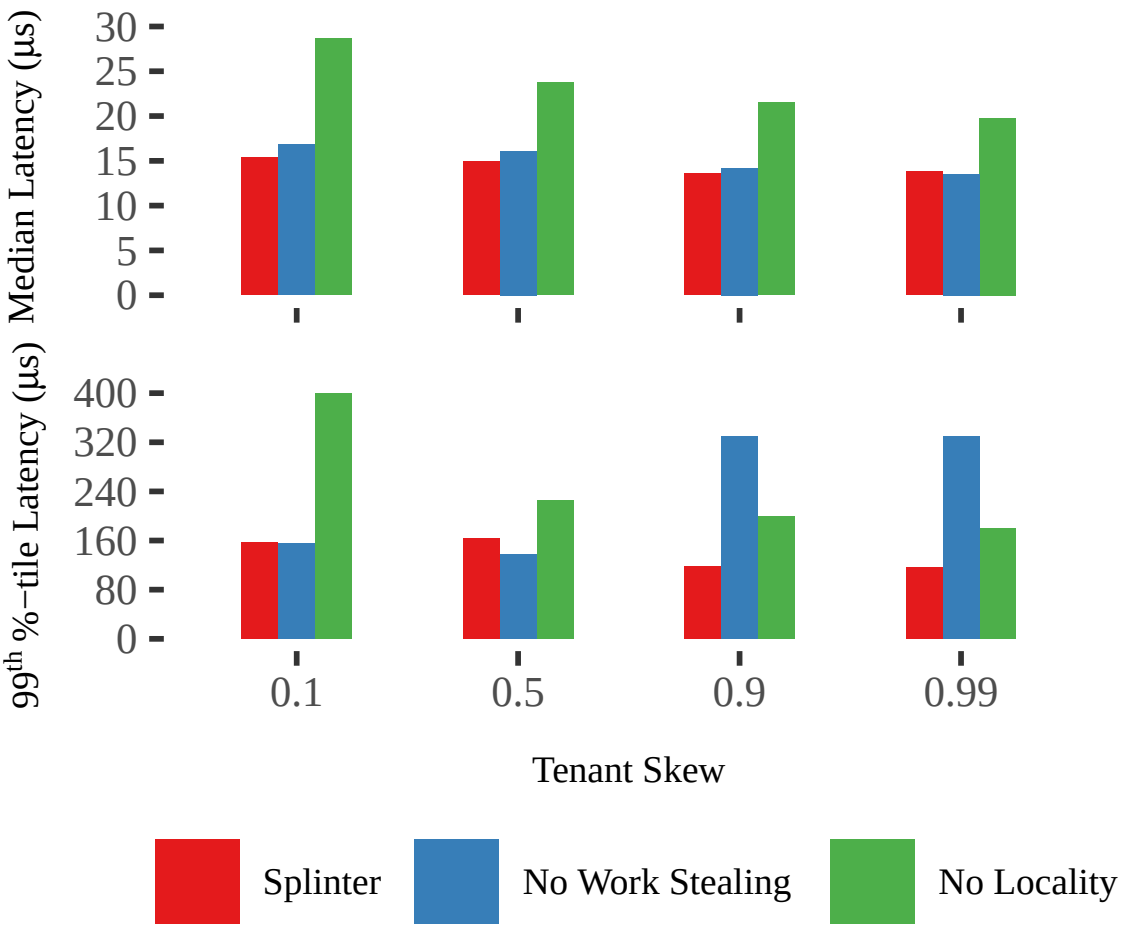


Figure 4.10: Latency with tenant skew. The server runs near saturation at 4 Mops/s in each case. Without work stealing, tail latency under high skew increases from 138 µs to 330 µs. Without tenant locality, median and tail latencies are affected.

approach that sprays requests over all cores in a tenant-oblivious fashion. We vary tenant skew, which affects all three approaches.

Figure 4.10 shows the results. These measurements are with an offered load of 4 Mops/s, keeping the store close to saturation. In each case, the store meets the offered load by running at 4 Mop/s. Without work stealing, Splinter's tail latency suffers by a factor of 2 under high tenant skew (0.9 and 0.99). In this case, partitioning helps throughput due to locality and reduced contention (as evidenced by its relatively consistent median response time), but queues become imbalanced hurting tail latency. The unpartitioned approach doesn't respond as significantly to tenant skew though it is slower overall, as expected. Unpartitioned execution results in 42% to 86% worse median latency with 38% to 155% worse tail latency.

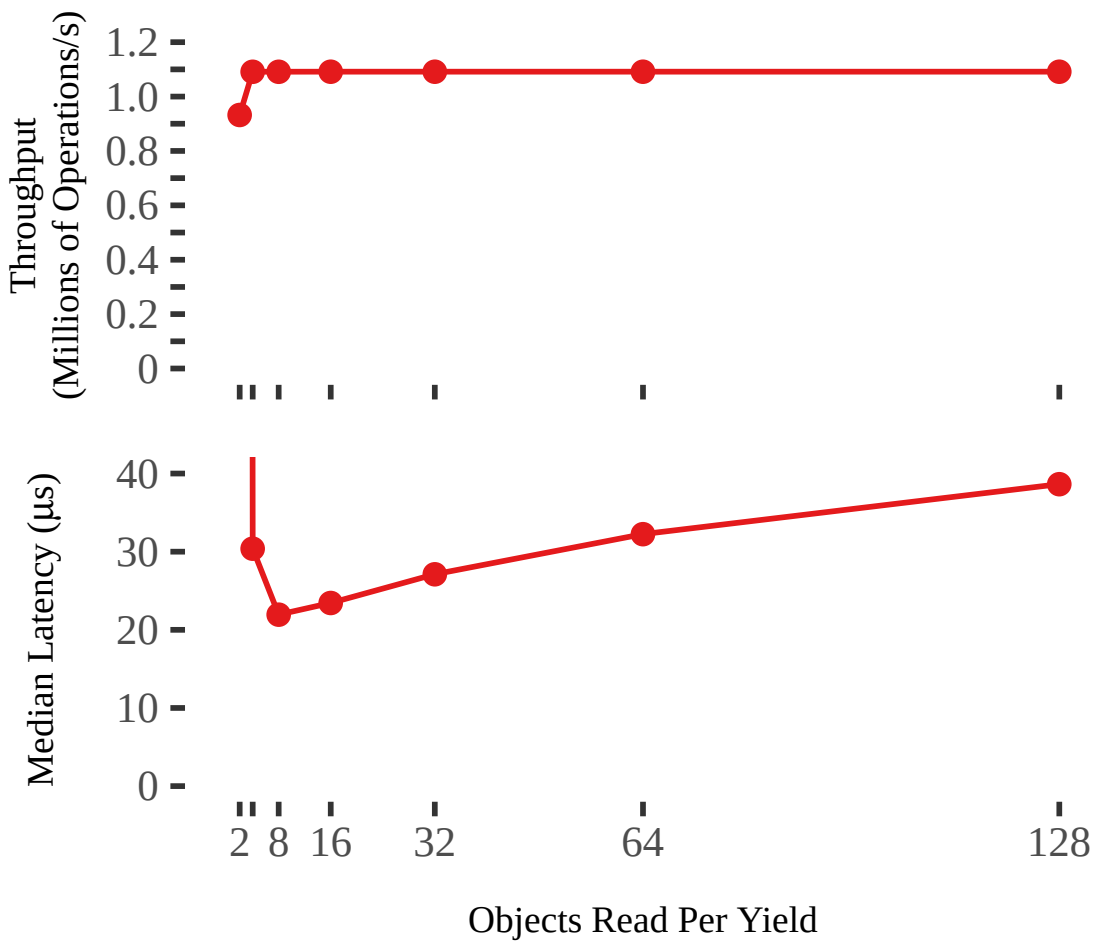


Figure 4.11: Performance with a small fraction (15%) of cooperative long running procedures that perform 128 `gets`. Yielding frequently can help improve median latency from 38 µs to 22 µs. However, yielding too frequently hurts median latency. The storage server was offered a constant load of 1.1 Mops/s.

4.5.4 Request Heterogeneity

Figure 4.11 investigates the impact of mixing short operations with cooperative longer-running operations. We configured our client so that 15% of extension operations performed 128 `gets` on the storage server. The rest of the requests invoked an extension that performed one `get`. We varied the number of `gets` made by the longer extension per yield (frequency). These measurements were made at an offered load of 1.1 Mops/s. Increasing the frequency of yields improves median latency of the smaller operations by 42% until a frequency of 8 `gets` per yield. Yields add some overhead, and yielding more frequently pushes the store to saturation in this case. As a result, all requests see increased response times. Extensions should yield frequently, but yielding too often is wasteful.

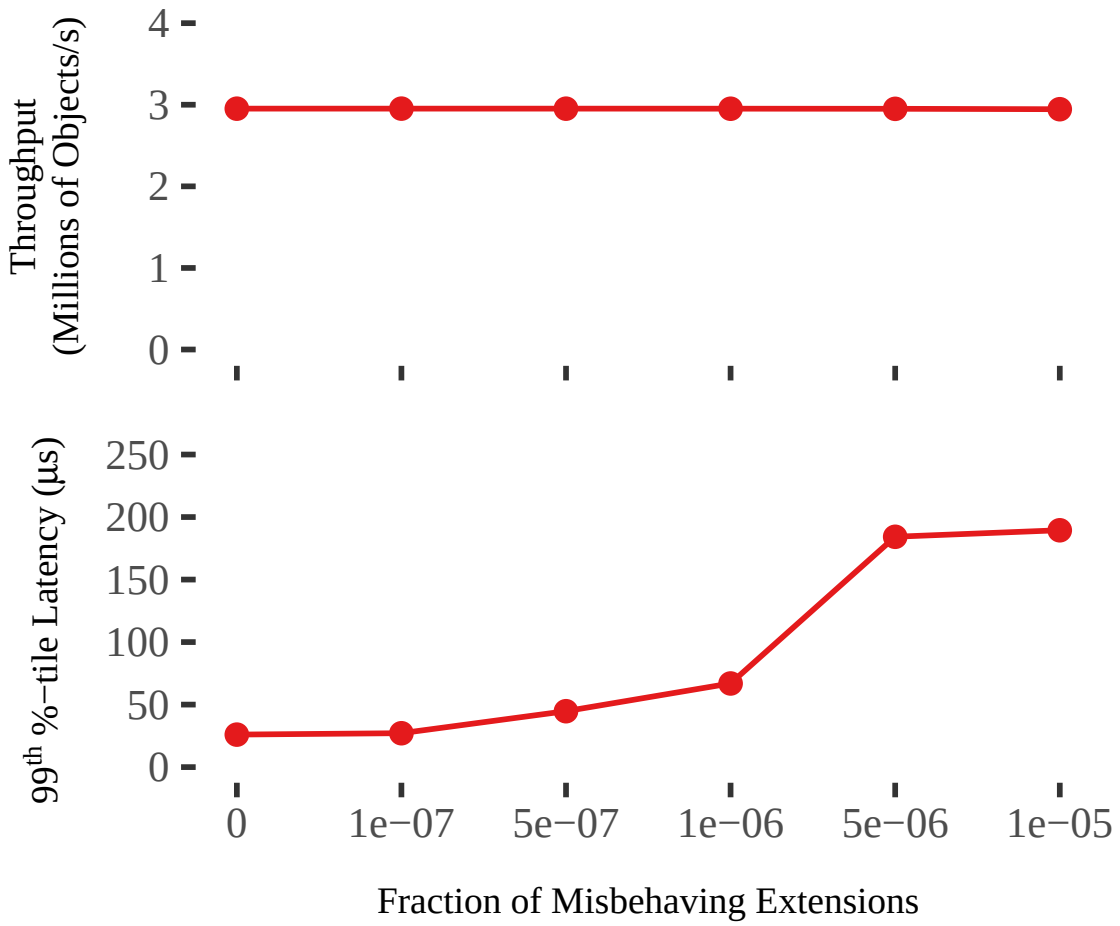


Figure 4.12: Impact of uncooperative requests on performance. System throughput stays constant at 3 Mops/s throughout. For fractions of uncooperative requests greater than 1 every million, tail latency is significantly affected (¿ 100 µs).

Splinter may be able to help with this in the future; Splinter could provide extensions with a yield that is ignored if called too quickly in succession, avoiding the full yield cost.

Figure 4.12 shows how uncooperative extensions impact system performance. Here, the client invoked a small fraction of extension operations that executed an infinite loop. The remaining fraction of requests invoked a small extension that performed a single `get`. Splinter performs well in the presence of misbehaving extensions. Throughput is steady at 3 Mops/s irrespective of the fraction of misbehaving requests. Median latency isn't shown, but it is steady as well. Tail latency suffers as more requests misbehave, though it is within 100 µs for fractions as high as one in a million requests.

Note that one in a million requests (1e-6) is harsh. The store can execute more than 4 Mop/s, so this represents a misbehaving invocation starting every quarter second; at 1e-5

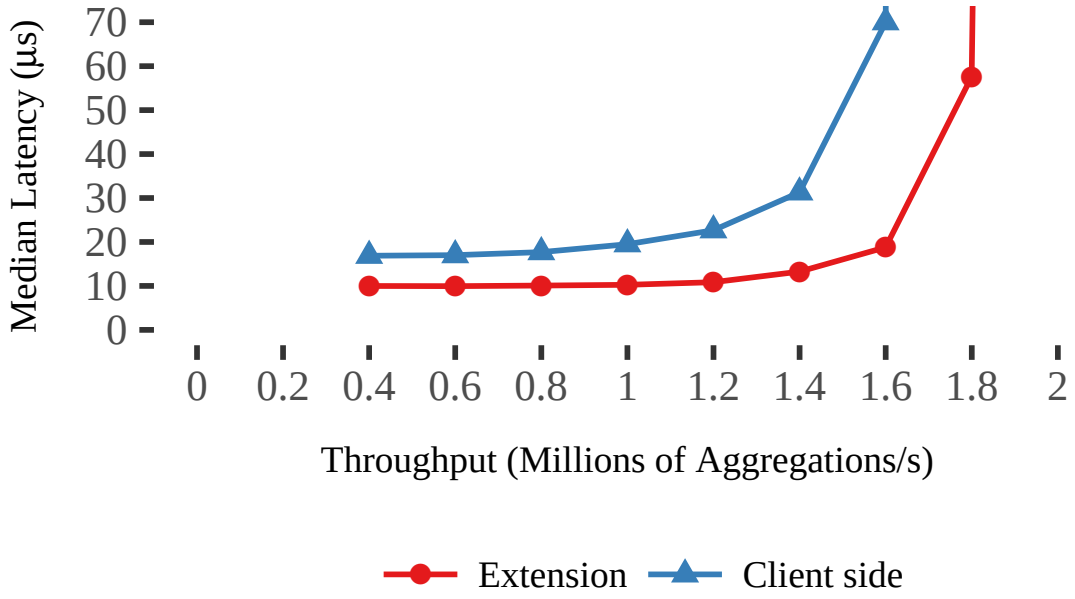


Figure 4.13: Aggregation throughput versus latency. Aggregations combine 4 records. Under low load, the median latency of a client-side implementation is 1.6× that of an extension-based implementation. Using an extension also improves saturating throughput from 1.2 M to 1.6 M aggregations per second.

misbehavior starts about once every 25 ms.

4.5.5 Aggregation Extension

Online data aggregation is a common task for applications. For example, a user might send a query demanding a movie studio's total earnings in the year 2017. With a key-value data model, this would require two round-trips to storage: one to fetch the list of movies made by the studio and one to fetch the box-office earnings of each of the movies. Splinter improves the user-facing and server-side performance of these types of queries by allowing applications to inexpensively embed their data model (studios and movies) and operations (total earnings aggregation) within storage.

Figure 4.13 compares a completely client-based and a Splinter extension-based implementation of such an aggregation over 4 records. Each of the store's 1,024 tenants owned a table with 300 K indirection lists pointing to 1.2 million records, totalling about 100 GB of stored data. The client-based implementation first performed a `get()` to retrieve an indirection list followed by a `multiget()` (a single RPC requesting values for multiple keys) to fetch all of the records indicated in the indirection list. The first

field from each of the returned objects is summed up into a single 64-bit result. The extension-based implementation invoked a Splinter extension called `aggregate()` with the same functionality as the client-based approach.

Pushing the aggregation from the client to the server has two key benefits. First, it improves performance from the client's perspective: the extension-based implementation reduces median latency by 38% (from 16 µs to 10 µs) under low load with larger gains under higher loads. This improvement is mainly due to a reduction in the number of round-trips; unlike the client-based extension, the `aggregate()` extension doesn't need to wait for the store to return an indirection list before it can start aggregation. Second, it improves performance from the server's perspective as well. Splinter's extension invocations are more expensive than plain `get()` operations (§4.5.2), but they eliminate some of the costly network and RPC processing. Hence, saturating throughput improves from 1.2 M to 1.6 M aggregations per second.

Note, this improvement comes in a challenging case for Splinter; at 40 Gbps, Splinter is never network limited. These results show that even if a store is CPU-limited, pushing compute to the store can still provide a throughput benefit, since it can mitigate request processing overheads. On slower networks, Splinter would provide more of a benefit since extensions can reduce network load.

Figure 4.14 shows the impact of the number of records aggregated on the saturating throughput of the extension-based and client-based implementation. In both approaches, increasing the number of records aggregated increases the work the store has to do per request (`aggregate()`/`multiget()`), and, hence, decreases the overall throughput of the system. However, if that work is simple (like summation) it is always better to aggregate at the store. The gain in saturating throughput of the extension-based aggregation is always more than 50%.

For compute-intensive operations, the extra CPU cost of running extensions at the store can outweigh the gains of fewer RPCs. Figure 4.15 explores this effect. After adding the first field of two records, each operation raises the result to the power `n` (with `n` 64-bit multiplications). Using an extension, increasing `n` above 2,000 slows the store and decreases saturating throughput from 1.8 M to 800 K aggregations per second. The client-side approach can hold throughput constant at 1.6 M aggregations per second; the

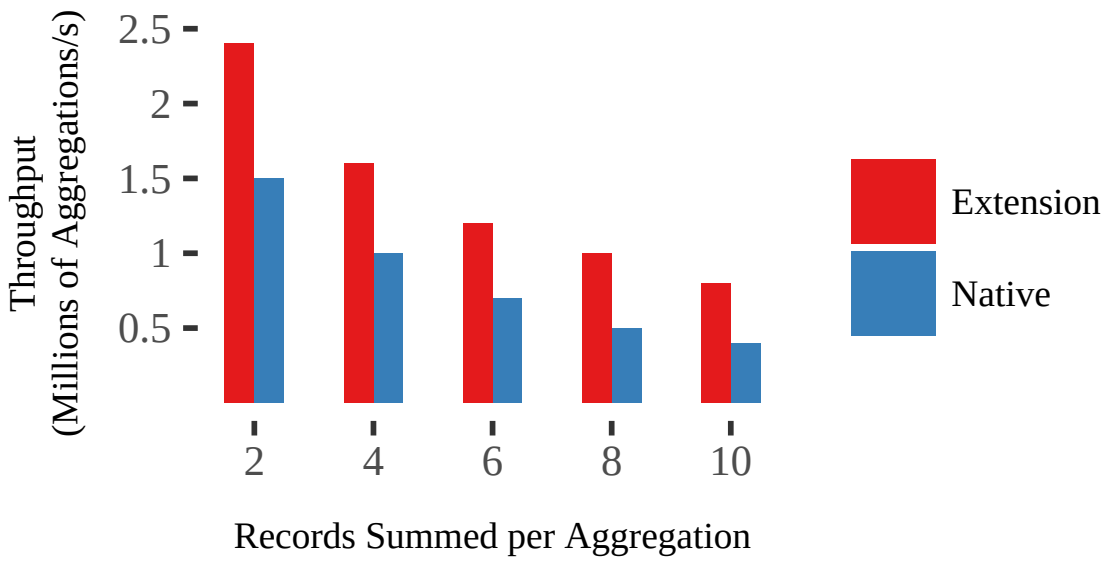


Figure 4.14: Saturating throughput of aggregation versus the number of aggregated records. The extension-based implementation outperforms the client-side implementation irrespective of the number of records aggregated. The gains are highest when aggregations are over two records (2.4 M versus 1.5 M aggregations per second).

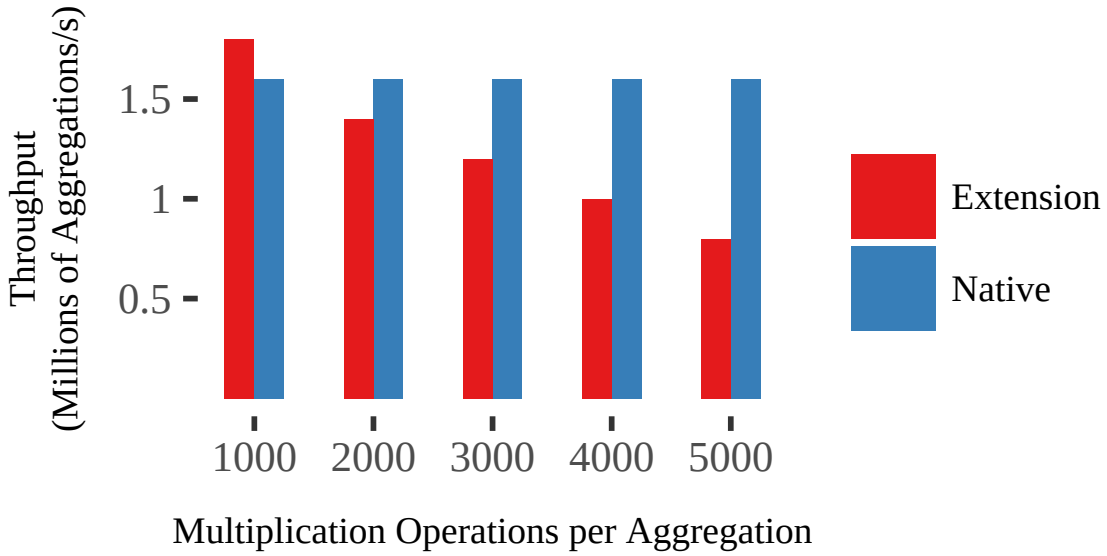


Figure 4.15: Saturating throughput of the aggregation extension versus the amount of compute per aggregation. After aggregating 2 records, each operation raised the result to the power `n`, implemented as `n` 64-bit multiplications (hence the x-axis). Increasing the order (`n`) increases server-side compute in the extension-based implementation, hurting throughput. At an order of 5000, the client-side approach is 2× faster.

client has enough idle CPU capacity to compute the result. This shows that extensions are ideal for operations with modest amounts of compute. For compute-intensive operations

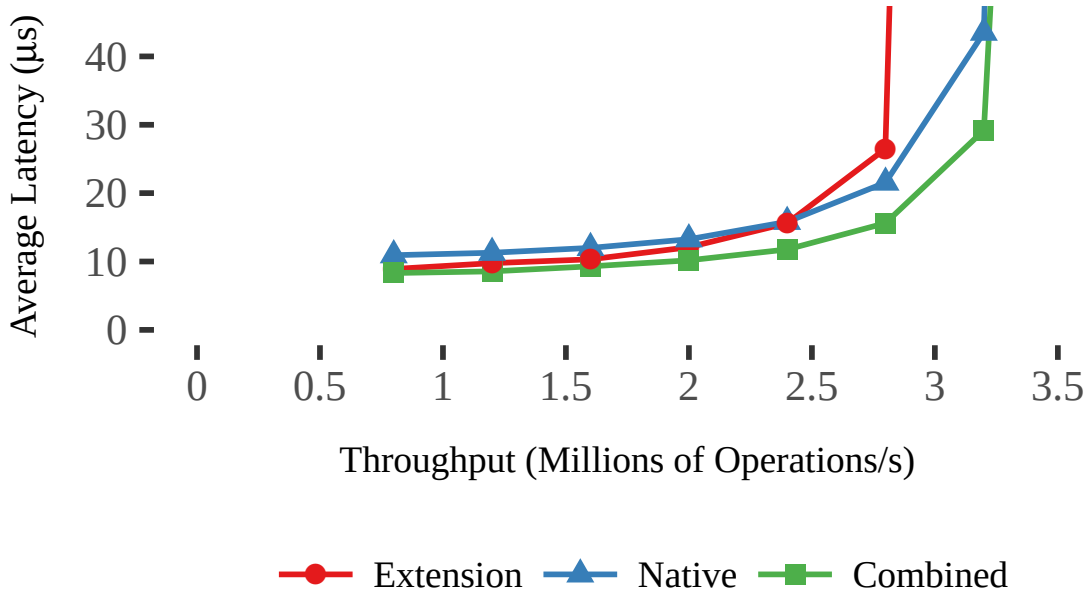


Figure 4.16: TAO extension throughput versus latency. With 60% `object_get` and 40% `assoc_range` operations, the TAO extension can reach 2.8 Mop/s before saturating with an average latency of 30 µs. By using native `get()` operations for `object_get`, the extension-based approach can outperform a purely client-side implementation by 400 Kop/s.

over data stored on high-load servers, clients should fetch data and perform operations locally.

4.5.6 TAO Extension

TAO [11] is a graph-oriented in-memory cache used at Facebook to hold objects from the social graph and associations between those objects. TAO is well-suited to Splinter. It is designed for interactive data, but it embeds knowledge about Facebook's workload to decrease round-trips to the store, which eliminates client-side stalls and improves server-side efficiency. We have implemented its simple operations as an 800-line Splinter extension.

Full details of TAO are beyond the scope of this paper, but the basics are simple. Aside from object put/get, TAO's *association lists* (e.g. `user1`'s "likes") allow one object to be associated to another via a typed, directed edges. For example, `user1`'s "likes" may be represented as an association list `(user1, likes)` → [`post1`, `post32`]. Association lists provide simple operations for adding, removing, and counting associations. Entries in association lists are timestamped, and range operations over association lists to fetch

subsets of them are common ("get the first 10 entries in the `(user1, likes)` association list").

Figure 4.16 shows Splinter's performance under three different configurations: an extension-based approach (Extension), a client-based approach (Native), and a combined approach (Combined) that implemented `object_get` using native `get()` operations, and `assoc_range` using an extension. The workload was configured to issue a mix of 60% `object_get` and 40% `assoc_range` operations. We picked this ratio based on Facebook's reported TAO workload [11], which is dominated by reads (99.8%) mostly from these two operations. Each of the 1,024 tenants on the storage node owned a graph with half a million objects and two million edges (associations), totalling about 100 GB of stored data.

Since a significant fraction of requests are single round-trip `object_gets`, the client-based approach has a better saturating throughput than the extension-based approach. However, combining the two improves saturating throughput from 2.8 Mop/s to 3.2 Mop/s at a latency of 31 µs; the native `get()` helps eliminate the isolation overhead while executing an `object_get`, and the extension helps reduce the number of round-trips required by an `assoc_range`.

This makes Splinter competitive with FaRM's TAO implementation which is the fastest known implementation. Interestingly FaRM, takes the opposite approach of Splinter. On FaRM, TAO operations use multiple RDMA reads and careful object layout. FaRM reported 6.3 Mops/s (about 200 Kop/s/core) with a 41 µs average latency; Splinter performs about 400 Kops/s/core with lower latency. Differences in hardware and experimental setup likely account for some of the differences, but it shows Splinter's CPU-active server approach is competitive against FaRM's CPU-passive server approach. Furthermore, Splinter maintains a simple, remote procedure call interface, and the TAO extension enforces strong abstract data types. Splinter TAO clients have no knowledge of the internal layout of the stored data objects.

4.6 Related Work

Shipping computation to data and isolating untrusted code are well-studied, and Splinter builds on prior work. However, prior work does not address multi-tenancy at Splinter's granularity and number of tenants; further, no work addresses these issues with

its throughput and latency goals, which are far beyond most cloud storage systems.

Low-latency RDMA-based Storage Systems. Low-latency, high-throughput key-value stores are now thousands of times faster than conventional cloud storage by exploiting RDMA, kernel-bypass, and DRAM [29, 30, 47, 67, 68, 83]. These systems are well-understood for small, regular workloads, but their simple (get/put, read/write) interfaces make them easy to optimize internally at the expense of application efficiency, since they force clients to make many round trips to storage and to compute locally [28]. RDMA lowers CPU overhead for transmit, but it cannot make up for the fundamental inefficiency of moving large amounts of data over the wire; receivers must still perform the same computation on the data that a server could have. Splinter eliminates this waste, while still using efficient kernel-bypass networking. At 40 Gbps a Splinter store is never network bound, so combining Splinter's approach with (one- or two-sided) RDMA verbs could provide a benefit by freeing up additional compute on store servers.

4.6.1 Pushing Computation to Storage

MapReduce [25] and Spark [119] ship code to data sets, though latency is not a concern. Even when compute is shipped to a storage (HDFS [101]) node, data is still copied via interprocess communication. Untrusted extensions, like those in Splinter, could eliminate these overheads.

Some distributed systems and frameworks support composing internal storage abstractions to synthesize new services [3, 4, 12, 37, 71, 100]. Malacology [100] claims storage extensions have been popular in the Ceph distributed file system, showing that extensions are useful to developers. In these systems, extensions are trusted, so they don't work for cloud storage; Splinter is also focused on tight integration of fine-grained computation and storage rather than on coarse composition of software services. Comet [35] embedded sandboxed Lua extensions into a decentralized hash table to allow application-specific extensions to get/put behavior. Lua's entry/exit costs are low; it is unclear how the performance of its just-in-time (JIT) compiled runtime would compare to Splinter.

SQL. SQL may be the most widely used approach to ship computation to data, and it also supports use as a stored procedure language [75, 82]. In-memory databases have

placed pressure on performance, resulting in JIT compilation for SQL [34, 78]. With JIT, queries run fast, and calls back-and-forth between the database and user logic are inexpensive. SQL is type safe, so it is also easy to isolate. SQL's main drawback is that it is declarative. Often, this is a benefit, since it can use runtime information for optimization, but this also limits its generality. Implementing new functionality, new operators, or complex algorithms in SQL is difficult and inefficient. Some have extended SQL for specific domains, like graph processing [77], scientific computing [70, 86] and simulation [13], showing that SQL by itself is insufficient for many domains.

Native-code Extensions. The popular Redis [92] in-memory store supports native extensions. In FaRM [29, 30], an RDMA-based in-memory store, applications are written as native, storage-embedded functions that are statically compiled into the server. These systems don't allow extensions to be loaded at runtime, and application code is trusted so it does not work for multi-tenant cloud storage. Similarly, H-Store [51], VoltDB [105], and Hazelcast [38] are in-memory stores that support Java-based procedures, though none of them provide multi-tenancy.

4.6.2 Fault Isolation

Software-fault isolation (SFI) sandboxes untrusted code within a process (or OS kernel [44, 99, 108]) with low control transfer costs [10, 33, 36, 72, 118]. Both hardware isolation [107] and SFI [113] were applied to Postgres [104], which pioneered database extensions [111]. SFI still requires protected data to be copied in/out of extensions, since it relies on hardware paging or address masking that can only restrict access to contiguous memory regions.

Language-level approaches to kernel extension [9, 40] closely match Splinter's design and goals. SPIN let language-isolated extensions run as part of the kernel. It eliminated runtime overheads (aside from garbage collection), since extensions were compiled; it eliminated control transfer overheads, since it didn't require page table switching; and it eliminated copying between protection domains, since type-safe pointers worked as capabilities. Like Splinter, where tenants must write Rust code, a key downside of SPIN was that extensions had to be written in Modula-3, not C, so legacy code couldn't be used. Java also "sandboxed" applets using type-safety and specialized class loaders, which

supported inexpensive control transfer and data access between domains [114].

Using Rust for low-cost, zero-copy isolation has been used for inexpensive software fault isolation both generally [5] and for network packet processing pipelines [85]. Splinter builds on these ideas, bringing them to storage and moving beyond static domains to a runtime extensible service. Tock [64] is an embedded OS that decomposes its kernel into untrusted *capsules* by exploiting Rust's safety. Tock's capsules are similar to Splinter's extensions, but they don't protect against denial of service (infinite loops) and capsules are static – they can't be added to a running kernel. These also differ from Splinter in that they assume a small number of trust domains; they are targeted at software decomposition. Splinter targets dense multi-tenancy with no static bound on the number of trust domains.

4.7 Conclusion

In-memory storage can significantly accelerate data-intensive applications, including those that need fine-grained and real-time access to data. However, as Dennard scaling ends, future cloud storage must not only be faster but also more efficient. Splinter shows that soon legacy hardware isolation techniques will limit resource provisioning granularity in the cloud, but it also provides a way forward. Systems must evolve to support granular, low-overhead shipping of compute to storage, and lightweight isolation between small compute tasks. Splinter works toward that evolution by discarding hardware isolation in favor of static safety checks. As a result, it supports thousands of tenants that can all access data in tens of microseconds while customizing storage operations to their needs and while performing millions of remote operations on modern multicore machines.

REFERENCES

[1] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, R. Peon, L. Kai, A. Shraer, A. Merchant, and K. Lev-Ari, *Slicer: Auto-sharding for datacenter applications*, in Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16, Berkeley, CA, USA, 2016, USENIX Association, pp. 739–753.

[2] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, *Sinfonia: A New Paradigm for Building Scalable Distributed Systems*, in Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07, New York, NY, USA, 2007, ACM, pp. 159–174.

[3] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, and J. D. Davis, *CORFU: A Shared Log Design for Flash Clusters*, in Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI '12, San Jose, CA, 2012, USENIX Association, pp. 1–14.

[4] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck, *Tango: Distributed Data Structures Over a Shared Log*, in Proceedings of the 24th ACM Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, 2013, ACM, pp. 325–340.

[5] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamarić, and L. Ryzhyk, *System Programming in Rust: Beyond Safety*, in Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS '17, New York, NY, 2017, ACM, pp. 156–161.

[6] S. Barker, Y. Chi, H. J. Moon, H. Hacigümüş, and P. Shenoy, *"Cut Me Some Slack": Latency-aware Live Migration for Databases*, in Proceedings of the 15th International Conference on Extending Database Technology, EDBT '12, New York, NY, USA, 2012, ACM, pp. 432–443.

[7] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, *Attack of the Killer Microseconds*, Communications of the ACM, 60 (2017), pp. 48–54.

[8] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, *IX: A Protected Dataplane Operating System for High Throughput and Low Latency*, in Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, 2014, USENIX Association, pp. 49–65.

[9] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, *Extensibility, Safety and Performance in the SPIN Operating System*, in ACM SIGOPS Operating Systems Review, vol. 29, ACM, 1995, pp. 267–283.

[10] A. Bittau, P. Marchenko, M. Handley, and B. Karp, *Wedge: Splitting Applications into Reduced-Privilege Compartments*, in Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, NSDI '08, Berkeley, CA, 2008, USENIX Association, pp. 309–322.

[11] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, *TAO: Facebook's Distributed Data Store for the Social Graph*, in Proceedings of the 2013 USENIX Annual Technical Conference, USENIX ATC '13, San Jose, CA, 2013, USENIX Association, pp. 49–60.

[12] A. Brown, D. Oppenheimer, K. Keeton, R. Thomas, J. Kubiatowicz, and D. A. Patterson, *ISTORE: Introspective Storage for Data-Intensive Network Services*, in Proceedings of the The 7th Workshop on Hot Topics in Operating Systems, HotOS '99, Washington, DC, 1999, IEEE Computer Society, pp. 32–37.

[13] Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine, *Simulation of database-valued Markov chains using SimSQL*, in Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13, New York, NY, 2013, ACM, pp. 637–648.

[14] C. Carruth, *[SLH] Introduce a new pass to do Speculative Load Hardening to mitigate*. `http://reviews.llvm.org/rL336990`, 2018. Accessed: 2018-09-27.

[15] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing, *Trill: A high-performance incremental query processor for diverse analytics*, Proc. VLDB Endow., 8 (2014), pp. 401–412.

[16] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, *Faster: A concurrent key-value store with in-place updates*, in Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18, New York, NY, USA, 2018, ACM, pp. 275–290.

[17] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, *Bigtable: A Distributed Storage System for Structured Data*, ACM Transactions on Computer Systems (TOCS), 26 (2008), pp. 4:1–4:26.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, *Benchmarking Cloud Serving Systems with YCSB*, in Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC '10, New York, NY, USA, 2010, ACM, pp. 143–154.

[19] M. Copeland, J. Soh, A. Puca, M. Manning, and D. Gollob, *Microsoft Azure: Planning, Deploying, and Managing Your Data Center in the Cloud*, Apress, USA, 1st ed., 2015.

[20] J. Corbet, *KAISER: hiding the kernel from user space*. `http://lwn.net/Articles/738975/`. Accessed: 2018-09-27.

[21] ——, *Meltdown/Spectre mitigation for 4.15 and beyond*. `http://lwn.net/Articles/744287/`, 2018. Accessed: 2018-09-27.

[22] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, *Spanner: Google's globally-distributed database*, in 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), Hollywood, CA, Oct. 2012, USENIX Association, pp. 251–264.

[23] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, *Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration*, Proc. VLDB Endow., 4 (2011), pp. 494–505.

[24] *Intel®Data Direct I/O technology*. `http://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html`. Accessed: 10-19-2016.

[25] J. Dean and S. Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, in Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation, OSDI '04, Berkeley, CA, 2004, USENIX Association, pp. 10–10.

[26] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, *Dynamo: Amazon's Highly Available Key-value Store*, in Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07, New York, NY, 2007, ACM, pp. 205–220.

[27] *Data Plane Development Kit*. `http://dpdk.org/`. 4/10/2017.

[28] A. Dragojević, D. Narayanan, and M. Castro, *RDMA Reads: To Use or Not to Use?*, IEEE Data Engineering Bulletin, 40 (2017), pp. 3–14.

[29] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, *FaRM: Fast Remote Memory*, in Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI '14, Berkeley, CA, 2014, USENIX Association, pp. 401–414.

[30] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, *No compromises: Distributed transactions with consistency, availability, and performance*, in Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15, New York, NY, USA, 2015, Association for Computing Machinery, p. 54–70.

[31] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. El Abbadi, *Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases*, in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15, New York, NY, USA, 2015, ACM, pp. 299–313.

[32] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi, *Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms*, in Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD '11, New York, NY, USA, 2011, ACM, pp. 301–312.

[33] B. Ford and R. Cox, *Vx32: Lightweight User-level Sandboxing on the x86*, in Proceedings of the 2008 USENIX Annual Technical Conference, USENIX ATC '08, Berkeley, CA, 2008, USENIX Association, pp. 293–306.

[34] C. Freedman, E. Ismert, and P. Larson, *Compilation in the Microsoft SQL Server Hekaton Engine*, IEEE Data Engineering Bulletin, 37 (2014), pp. 22–30.

[35] R. Geambasu, A. A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy, *Comet: An Active Distributed Key-Value Store*, in Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI '10, Vancouver, BC, 2010, USENIX Association, pp. 323–336.

[36] Google LLC., *NaCl and PNaCl*. `http://developer.chrome.com/native-client/nacl-and-pnacl`. Accessed: 2018-09-27.

[37] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler, *Scalable, Distributed Data Structures for Internet Service Construction*, in Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation, OSDI '00, Berkeley, CA, 2000, USENIX Association.

[38] Hazelcast., *Hazelcast the Leading In-Memory Data Grid - Hazelcast.com*. `http://hazelcast.com/`. Accessed: 2018-09-27.

[39] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Elsevier, 2011.

[40] G. Hunt and J. Larus, *Singularity: Rethinking the Software Stack*, ACM SIGOPS Operating Systems Review, 41/2 (2007), pp. 37–49.

[41] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, *ZooKeeper: Wait-free Coordination for Internet-scale Systems*, in 2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010, USENIX Association, 2010.

[42] IEEE, *802.3-2015 - IEEE Standard for Ethernet*. `https://standards.ieee.org/findstds/standard/802.3-2015.html`.

[43] Intel Corporation., *Flow Director*. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf`. Accessed: 2018-09-27.

[44] C. Jacobsen, M. Khole, S. Spall, S. Bauer, and A. Burtsev, *Lightweight Capability Domains: Towards Decomposing the Linux Kernel*, SIGOPS Operating Systems Review, 49 (2016), pp. 44–50.

[45] R. Jung, *LLVM loop optimization can make safe programs crash #28728*. `http://github.com/rust-lang/rust/issues/28728`, 2018. Accessed: 2018-09-27.

[46] R. Jung, J. Jourdan, R. Krebbers, and D. Dreyer, *RustBelt: Securing the Foundations of the Rust Programming Language*, Proceedings of the ACM on Programming Languages, 2 (2018), pp. 66:1–66:34.

[47] A. Kalia, M. Kaminsky, and D. G. Andersen, *Using RDMA Efficiently for Key-value Services*, in Proceedings of the 2014 ACM Conference on SIGCOMM, SIGCOMM '14, New York, NY, USA, 2014, ACM, pp. 295–306.

[48] A. Kalia, M. Kaminsky, and D. G. Andersen, *Design Guidelines for High Performance RDMA Systems*, in 2016 USENIX Annual Technical Conference (USENIX ATC 16), Denver, CO, June 2016, USENIX Association, pp. 437–450.

[49] ——, *FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs*, in Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI '16, Savannah, GA, 2016, USENIX Association, pp. 185–201.

[50] ——, *Datacenter RPCs can be General and Fast*, CoRR, abs/1806.00680 (2018).

[51] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, *H-store: A High-performance, Distributed Main Memory Transaction Processing System*, Proc. VLDB Endow., 1 (2008), pp. 1496–1499.

[52] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy, *High performance packet processing with flexnic*, in Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, New York, NY, USA, 2016, Association for Computing Machinery, p. 67–81.

[53] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. Ousterhout, *SLIK: Scalable Low-Latency Indexes for a Key-Value Store*, in 2016 USENIX Annual Technical Conference (USENIX ATC 16), Denver, CO, June 2016, USENIX Association, pp. 57–70.

[54] A. Kesavan, R. Ricci, and R. Stutsman, *To Copy or Not to Copy: Making In-Memory Databases Fast on Modern NICs*, in 4th Workshop on In-memory Data Management, 2017.

[55] V. Kiriansky and C. Waldspurger, *Speculative Buffer Overflows: Attacks and Defenses*, CoRR, abs/1807.03757 (2018).

[56] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, *Spectre Attacks: Exploiting Speculative Execution*, in Proceedings of the 40th IEEE Symposium on Security and Privacy, S&P '19, 2019.

[57] C. Kulkarni, B. Chandramouli, and R. Stutsman, *Achieving high throughput and elasticity in a larger-than-memory store*, 2020.

[58] C. Kulkarni, A. Kesavan, T. Zhang, R. Ricci, and R. Stutsman, *Rocksteady: Fast Migration for Low-latency In-memory Storage*, in Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, New York, NY, 2017, ACM, pp. 390–405.

[59] C. Kulkarni, S. Moore, M. Naqvi, T. Zhang, R. Ricci, and R. Stutsman, *Splinter: Bare-metal extensions for multi-tenant low-latency storage*, in 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), Carlsbad, CA, Oct. 2018, USENIX Association, pp. 627–643.

[60] H. T. Kung and P. L. Lehman, *Concurrent manipulation of binary search trees*, ACM Trans. Database Syst., 5 (1980), p. 354–382.

[61] C. Lattner and V. Adve, *LLVM: A compilation framework for lifelong program analysis & transformation*, in Proceedings of the international symposium on Code generation

and optimization: feedback-directed and runtime optimization, CGO '04, IEEE, 2004, pp. 75–86.

[62] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. Ousterhout, *Implementing linearizability at large scale and low latency*, in Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15, New York, NY, USA, 2015, ACM, pp. 71–86.

[63] V. Leis, P. Boncz, A. Kemper, and T. Neumann, *Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age*, in Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14, New York, NY, USA, 2014, ACM, pp. 743–754.

[64] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis, *Multiprogramming a 64kB Computer Safely and Efficiently*, in Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, New York, NY, 2017, ACM, pp. 234–251.

[65] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. 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, New York, NY, USA, 2017, ACM, pp. 137–152.

[66] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, *Scaling distributed machine learning with the parameter server*, in 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), Broomfield, CO, Oct. 2014, USENIX Association, pp. 583–598.

[67] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey, *Architecting to achieve a billion requests per second throughput on a single key-value store server platform*, in Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA '15, New York, NY, USA, 2015, ACM, pp. 476–488.

[68] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, *MICA: A Holistic Approach to Fast In-memory Key-value Storage*, in Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI '14, Berkeley, CA, 2014, USENIX Association, pp. 429–444.

[69] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, *Meltdown: Reading kernel memory from user space*, in Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, 2018, USENIX Association, pp. 973–990.

[70] R. Maas, J. Hyrkas, O. Telford, M. Balazinska, A. Connolly, and B. Howe, *Gaussian Mixture Models Use-Case: In-Memory Analysis with Myria*, Third International Workshop on In-Memory Data Management and Analytics (IMDM'15), (2015).

[71] J. MacCormick, N. Murphy, M. Najork, C. A. Theth, and L. Zhou, *Boxwood: Abstractions As the Foundation for Storage Infrastructure*, in Proceedings of the 6th

USENIX Symposium on Opearting Systems Design and Implementation, vol. 6 of OSDI '04, Berkeley, CA, 2004, USENIX Association, pp. 8–8.

[72] S. McCamant and G. Morrisett, *Evaluating SFI for a CISC Architecture*, in Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15, USENIX-SS '06, Berkeley, CA, 2006, USENIX Association.

[73] P. E. McKenney and J. D. Slingwine, *Read-copy update: Using execution history to solve concurrency problems*, in Parallel and Distributed Computing and Systems, 1998, pp. 509–518.

[74] Mellanox Technologies, *Mellanox Announces 200Gb/s HDR InfiniBand Solutions Enabling Record Levels of Performance and Scalability*. `http://www.mellanox.com/page/press_release_item?id=1810`, 2016.

[75] Microsoft, Inc., *Transact-SQL Reference (Database Engine)*. `http://docs.microsoft.com/en-us/sql/t-sql/language-reference`. Accessed: 2018-09-27.

[76] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, *Latency-Tolerant Software Distributed Shared Memory*, in 2015 USENIX Annual Technical Conference, USENIX ATC '15, Santa Clara, CA, July 2015, USENIX Association, pp. 291–305.

[77] Neo4j, Inc., *Neo4j, the World's Leading Graph Database*. `http://neo4j.com/`. Accessed: 2018-09-27.

[78] T. Neumann, *Efficiently Compiling Efficient Query Plans for Modern Hardware*, Proceedings of the VLDB Endowment, 4 (2011), pp. 539–550.

[79] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, *Scaling memcache at facebook*, in 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), Lombard, IL, Apr. 2013, USENIX Association, pp. 385–398.

[80] D. Ongaro and J. Ousterhout, *In Search of an Understandable Consensus Algorithm*, in 2014 USENIX Annual Technical Conference (USENIX ATC 14), Philadelphia, PA, 2014, USENIX Association, pp. 305–319.

[81] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, *Fast Crash Recovery in RAMCloud*, in Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, ACM, 2011, pp. 29–41.

[82] Oracle, Inc., *Oracle Database 12c PL/SQL*. `http://www.oracle.com/technetwork/database/features/plsql/index.html`. Accessed: 2018-09-27.

[83] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, and et al., *The ramcloud storage system*, ACM Trans. Comput. Syst., 33 (2015).

[84] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, *Sparrow: Distributed, Low Latency Scheduling*, in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13, New York, NY, USA, 2013, ACM, pp. 69–84.

[85] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, *NetBricks: Taking the V out of NFV*, in Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI '16, Savannah, GA, 2016, USENIX Association, pp. 203–216.

[86] Paradigm4, Inc., *Paradigm4: Cretators of SciDB a computational Database*. `http://www.paradigm4.com/`. Accessed: 2018-09-27.

[87] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, *Arrakis: The operating system is the control plane*, in Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI'14), 2014.

[88] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson, *Floem: A programming system for nic-accelerated network applications*, in Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'18, USA, 2018, USENIX Association, p. 663–679.

[89] G. Prasaad, B. Chandramouli, and D. Kossmann, *Concurrent Prefix Recovery: Performing CPR on a Database*, in Proceedings of the 2019 International Conference on Management of Data, SIGMOD '19, New York, NY, USA, 2019, Association for Computing Machinery, p. 687–704.

[90] G. Prekas, M. Kogias, and E. Bugnion, *ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks*, in Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, Shanghai, China, 2017, ACM, pp. 325–341.

[91] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout, *Arachne: Core-Aware Thread Management*, in Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI '2018, Carlsbad, CA, 2018, USENIX Association.

[92] *Redis*. `http://redis.io/`. 7/24/2015.

[93] R. Ricci and E. Eide, *Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications*, ; login:, 39 (2014), pp. 36–38.

[94] S. M. Rumble, A. Kejriwal, and J. Ousterhout, *Log-structured Memory for DRAM-based Storage*, in Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14), Santa Clara, CA, 2014, USENIX, pp. 1–16.

[95] *The Rust Programming Language*. `http://www.rust-lang.org/en-US/`. Accessed: 2018-09-27.

[96] *The Rust Programming Language*. `http://www.rust-lang.org/en-US/`. Accessed: 2018-09-27.

[97] O. Schiller, N. Cipriani, and B. Mitschang, *ProRea: Live Database Migration for Multi-tenant RDBMS with Snapshot Isolation*, in Proceedings of the 16th International Conference on Extending Database Technology, EDBT '13, New York, NY, USA, 2013, ACM, pp. 53–64.

[98] *Seastar Framework*. `http://seastar.io`. Accessed: 4/22/2020.

[99] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith, *Dealing with Disaster: Surviving Misbehaved Kernel Extensions*, in Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation, OSDI '96, USENIX Association, 1996, pp. 213–227.

[100] M. A. Sevilla, N. Watkins, I. Jimenez, P. Alvaro, S. Finkelstein, J. LeFevre, and C. Maltzahn, *Malacology: A Programmable Storage System*, in Proceedings of the 12th European Conference on Computer Systems, Eurosys '17, ACM, 2017, pp. 175–190.

[101] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, *The Hadoop Distributed File System*, in 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), IEEE, 2010, pp. 1–10.

[102] *Spark Streaming*. https://spark.apache.org/streaming/.

[103] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, *Chord: A scalable peer-to-peer lookup service for internet applications*, in Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, 2001.

[104] M. Stonebraker and G. Kemnitz, *The POSTGRES Next Generation Database Management System*, Communications of the ACM, 34 (1991), pp. 78–92.

[105] M. Stonebraker and A. Weisberg, *The voltdb main memory DBMS*, IEEE Data Eng. Bull., 36 (2013), pp. 21–27.

[106] R. Stutsman, C. Lee, and J. Ousterhout, *Experience with Rules-Based Programming for Distributed, Concurrent, Fault-Tolerant Code*, in USENIX ATC, Santa Clara, CA, July 2015.

[107] M. Sullivan and M. Stonebraker, *Using Write Protected Data Structures to Improve Software Fault Tolerance in Highly Available Database Management Systems*, in Proceedings of the VLDB Endowment, VLDB '91, VLDB Endowment, 1991, pp. 171–180.

[108] M. M. Swift, B. N. Bershad, and H. M. Levy, *Improving the Reliability of Commodity Operating Systems*, in ACM SIGOPS Operating Systems Review, vol. 37, ACM, 2003, pp. 207–222.

[109] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker, *E-store: Fine-grained Elastic Partitioning for Distributed Transaction Processing Systems*, Proc. VLDB Endow., 8 (2014), pp. 245–256.

[110] The Apache Software Foundation, *Apache Cassandra*. `http://cassandra.apache.org/`.

[111] The PostgreSQL Global Development Group., *PostgreSQL: Documentation: 10: H.4. Extensions*. `http://www.postgresql.org/docs/10/static/external-extensions.html`. Accessed: 2018-09-27.

[112] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, *Speedy Transactions in Multicore In-memory Databases*, in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, ACM, 2013, pp. 18–32.

[113] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, *Efficient Software-based Fault Isolation*, in Proceedings of the 14th Symposium on Operating Systems Principles, SOSP '93, New York, NY, 1993, ACM, pp. 203–216.

[114] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten, *Extensible Security Architectures for Java*, SIGOPS Operating Systems Review, 31 (1997), pp. 116–128.

[115] X. Wei, S. Shen, R. Chen, and H. Chen, *Replication-driven live reconfiguration for fast distributed transaction processing*, in 2017 USENIX Annual Technical Conference (USENIX ATC 17), Santa Clara, CA, 2017, USENIX Association, pp. 335–347.

[116] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, *Fast In-memory Transaction Processing Using RDMA and HTM*, in Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15, New York, NY, 2015, ACM, pp. 87–104.

[117] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein, *Anna: A kvs for any scale*, in 2018 IEEE 34th International Conference on Data Engineering (ICDE), 2018, pp. 401–412.

[118] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, *Native Client: A Sandbox for Portable, Untrusted x86 Native Code*, in Proceedings of the 30th IEEE Symposium on Security and Privacy, S&P '09, IEEE, 2009, pp. 79–93.

[119] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*, in 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), San Jose, CA, 2012, USENIX, pp. 15–28.

[120] W. Zheng, S. Tu, E. Kohler, and B. Liskov, *Fast databases with fast durability and recovery through multicore parallelism*, in Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14, Berkeley, CA, USA, 2014, USENIX Association, pp. 465–477.