MAKING LOW LATENCY STORES PRACTICAL AT CLOUD SCALE

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

CONTENTS

LIST OF FIGURES

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 [12, 25, 30]. 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 [11]. 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 [21]. 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 [19, 23, 25, 32]. 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 [24, 28] etc. Storage systems, on the other hand, have become faster with the help of kernel-bypass [13, 30], 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 [22]. 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, stipped 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* [21], 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* [20], 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* [22], 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 [30], 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 [7] 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 a high throughput of 130 Mops/s/VM over commodity Azure VMs [8]. 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 [18, 25, 36, 38], 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 [35] 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 [16] 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 [5] 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 [12] 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

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 5 µs or less [12, 30]. 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, this chapter presents *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. Measurements show that Rocksteady can improve the efficiency of clustered accesses and index operations by more than 4× (Figure 2.1): operations that are common in many real-world large-scale systems [9, 28]. Several works address the general problem of online (or *live*) data migration for scale-out stores [1, 9–11, 14, 15, 39], 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 5 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 [11]. For clients with high fan-out requests, even a millisecond of extra tail latency would destroy client-observed performance. Migration must have minimum

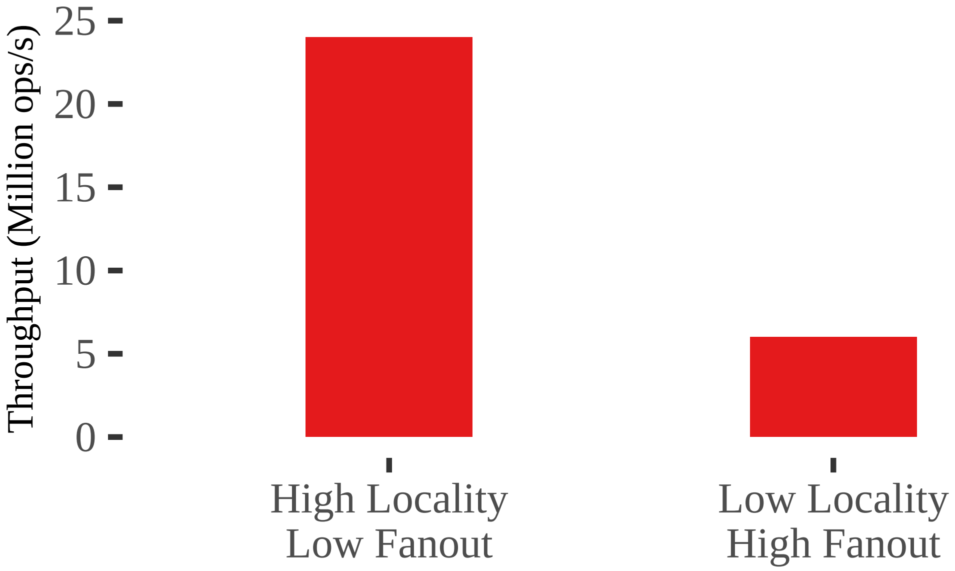

Figure 2.1: The impact of locality on cluster throughput. When locality is low due to high fanout, a cluster of RAMCloud servers performs 6 Million ops/s. On improving locality, by potentially migrating data to reduce fanout, throughput improves to 24 Million ops/s

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 [26] 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. 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 [29], 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 [11, 12]. 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 [34] 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 [42]; 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.

CHAPTER 3

LOW COST COORDINATION

Millions of sensors, mobile applications, users, and machines now continuously generate billions of events. These events are processed by streaming engines [6, 37] 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 key-value stores (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 [7, 41]. 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 [19, 23, 25, 32] 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;

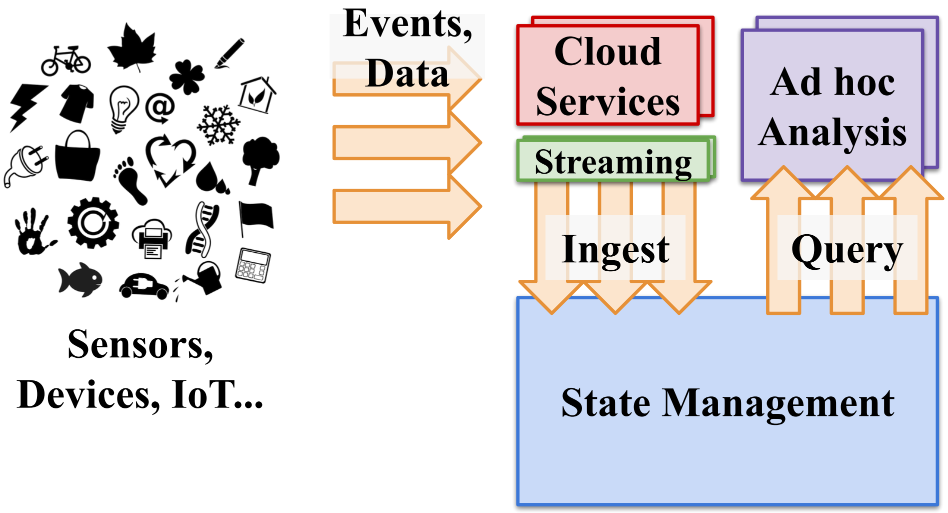

Figure 3.1: A typical data processing pipeline. Services receive and process raw events and data. A state management system ingests processed events and data, and serves offline queries against them.

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. However, 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).

Shadowfax is a new distributed KVS that transparently spans DRAM, SSDs, and cloud blob storage while serving 130 Mops/s/VM over commodity Azure VMs [8] 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 [36] and scales out 6× faster (than Rocksteady [21]).

Three key pieces of Shadowfax help 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. Shadowfax extends asynchronous cuts from cores within one process [7] to servers and clients in a cluster. This helps coordinate server and client threads (through partitioned sessions) 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 [18, 25, 36, 38], 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.

Evaluating Shadowfax in detail against other state-of-the-art shared-nothing approaches shows 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. It also retains high throughput during migrations and scales to a cluster that ingests and indexes 400 Mops/s in total, which is the highest reported throughput for a distributed KVS till date.

CHAPTER 4

EXTENSIBILITY AND MULTI-TENANCY

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 [12, 17, 25, 30, 40]. 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 [2, 27]. 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 (Figure 4.1).

This tension is resolved 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

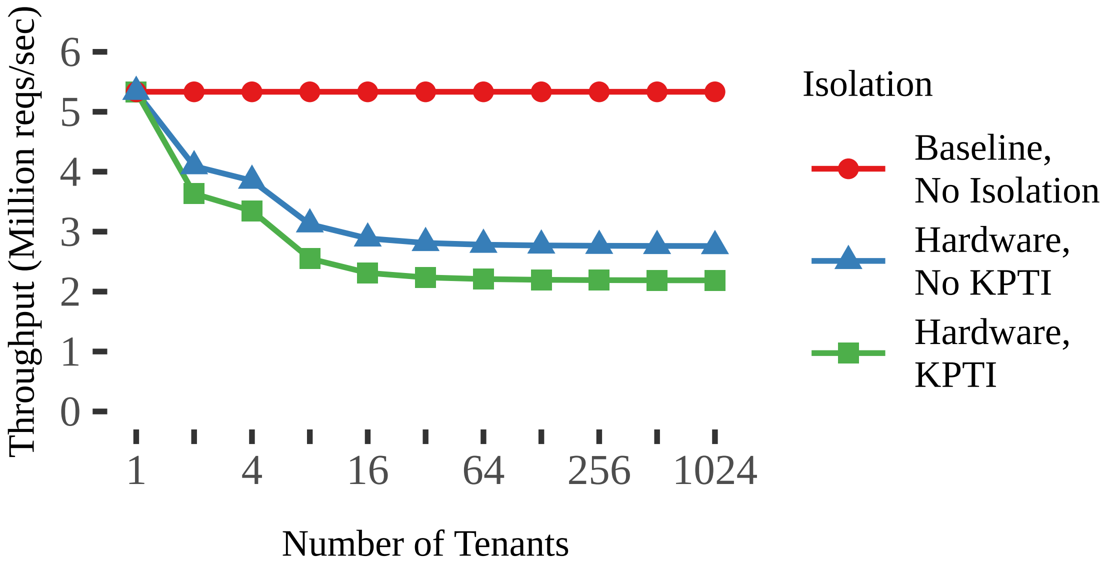

Figure 4.1: Simulated throughput of a system that isolates pushed code using hardware. The baseline represents the upper bound when there is no isolation. At high tenant density, isolation hurts throughput by nearly 2x.

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 [33]. Routing decisions can't get in the way of fast dispatch of requests [3].

These goals give rise to Splinter's design. Developers write type-safe, memory-safe extensions in Rust [35] that they push to Splinter servers. Exploiting type-safety for lightweight isolation isn't new; SPIN [4] allowed applications to safely and dynamically load extensions into its kernel by relying on language-enforced isolation. Similarly, NetBricks [31] 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.

REFERENCES

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] A. Kalia, M. Kaminsky, and D. G. Andersen, *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.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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).

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

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

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

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.