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

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

CHAPTERS

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 [10, 28, 33]. 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 [9]. 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 [22]. 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 [18, 26, 28, 35]. 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 [27, 30] etc. Storage systems, on the other hand, have become faster with the help of kernel-bypass [11, 33], 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 [23]. 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* [22], 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* [21], 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* [23], 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 [33], 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 [4] 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 [5]. 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 [17, 28, 38, 40], 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 [37] 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 [3] 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 [10] 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 [10, 33]. 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 [6, 30]. Several works address the general problem of online (or *live*) data migration for scale-out stores [2, 6, 7, 9, 12, 13, 42], 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 [9]. 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 [29] 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 [32], 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 [9, 10]. 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 [36] 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 [44]; 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 (§2.2) 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 (§**??**). Compared to prior approaches, Rocksteady transfers data an order of magnitude faster (> 750 MB/s) with median and tail latencies 1,000× 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 [33] 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 (§**??**).

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 [31].

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 [32], 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 [36].

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

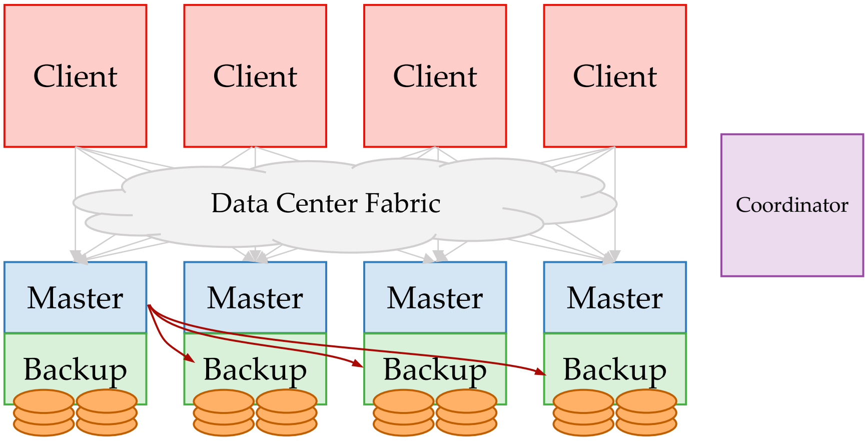

Figure 2.1: The RAMCloud architecture. Clients issue remote operations to RAMCloud storage servers. Each server contains a master and a backup. The master component exports the DRAM of the server as a large key-value store. The backup accepts updates from other masters and records state on disk used 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 [19]. 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 [24].

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 [9, 39, 43]. 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

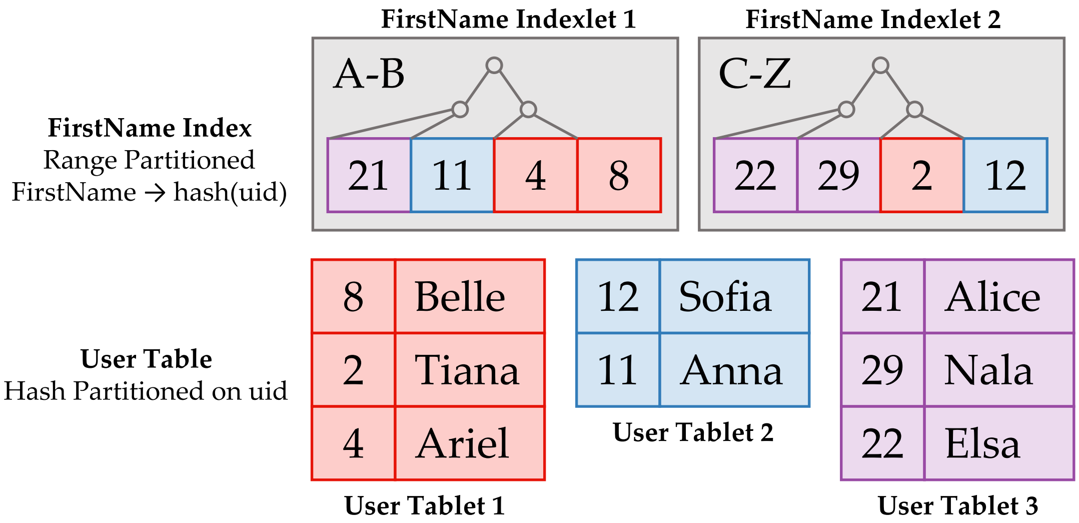

Figure 2.2: Index partitioning. Records are stored in unordered tables that can be split into tablets on different servers, partitioned on primary key hash. Indexes can be range partitioned into indexlets; indexes only contain primary key hashes. Range scans require first fetching a list of hashes from 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 [1, 24] 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 7th 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

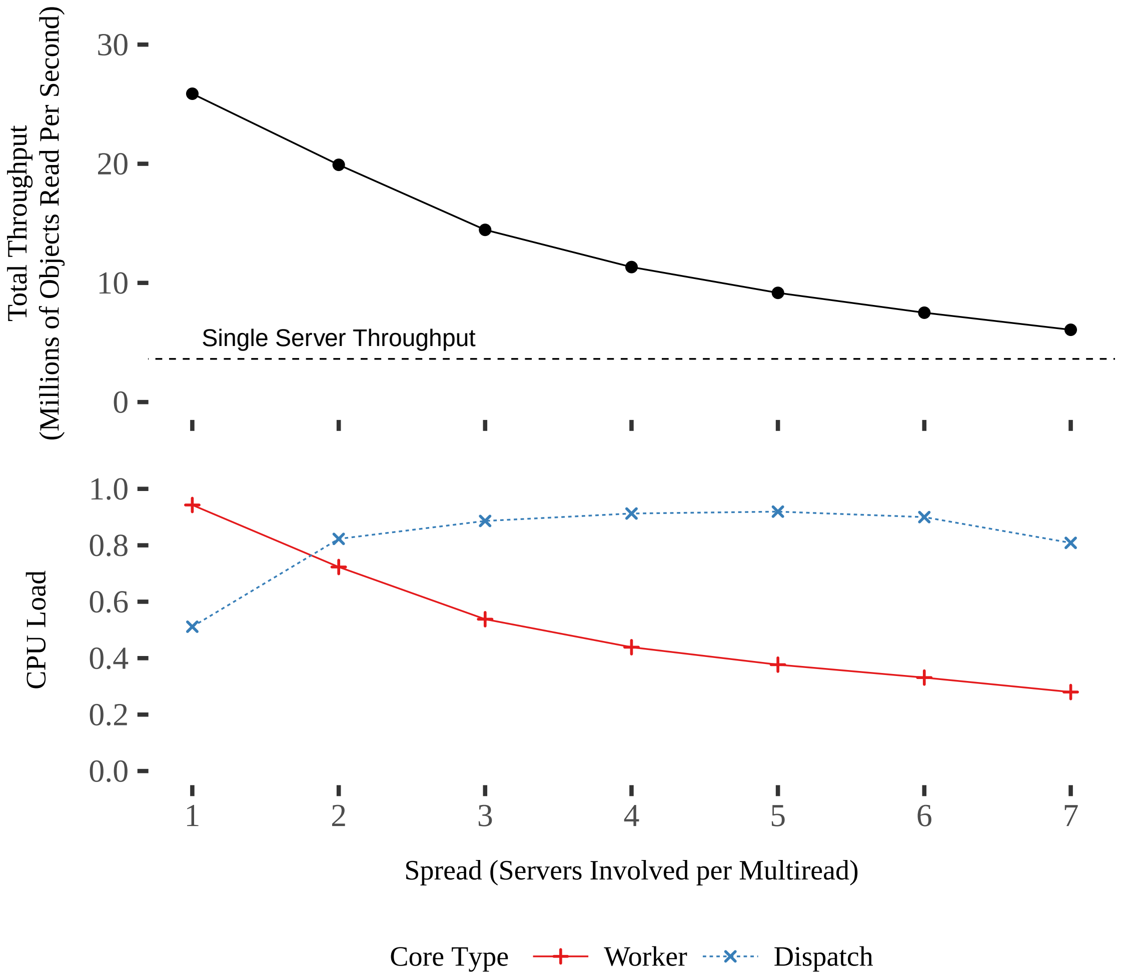

Figure 2.3: Throughput and CPU load impact of access locality. When multigets always fetch data from a single server (Spread 1) throughput is high and worker cores operate in parallel. When each multiget must fetch keys from many machines (Spread 7) throughput suffers as each server becomes bottlenecked 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 trends in scale-out stores that replicate to minimize multiget "fan out" [30] or give users explicit control over data placement to exploit locality [6].

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 $\theta = 0.5$. 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

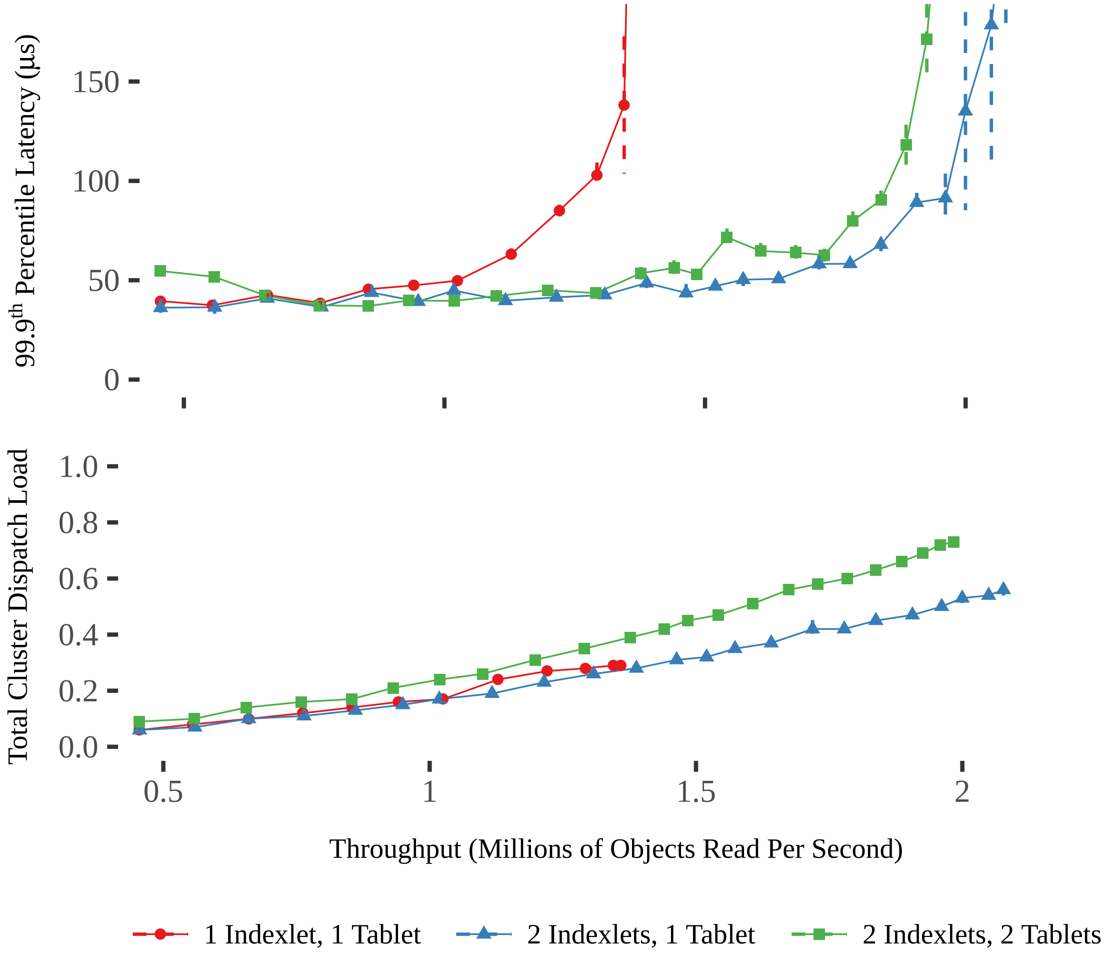

Figure 2.4: Index scaling as a function of read throughput. Points represent the median over 5 runs, bars show standard error. Spreading the backing table across two servers increases total dispatch load and the 99.9th percentile access latency for a given throughput when compared to leaving it on a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

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 [14], 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 [15], 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 [12]. 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

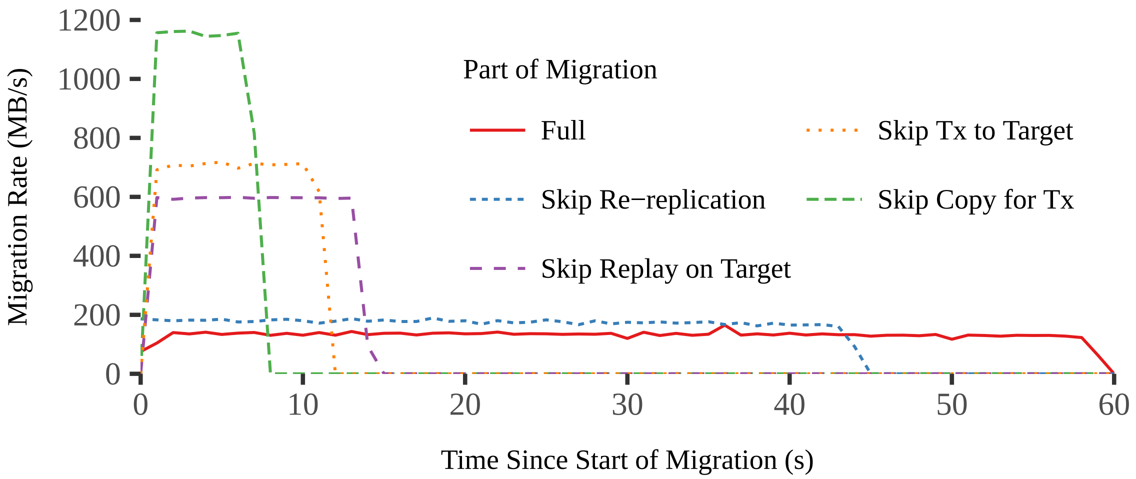

Figure 2.5: Bottlenecks using log replay for migration. Target side bottlenecks include logical replay and re-replication. Copying records into staging buffers at the source has a significant impact on migration rate.

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 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 [44]; 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 (§**??**). 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 [25, 34]. Rocksteady also incorporates into RAMCloud's transport layer to minimize jitter caused by background migration transfers (§2.3.1).

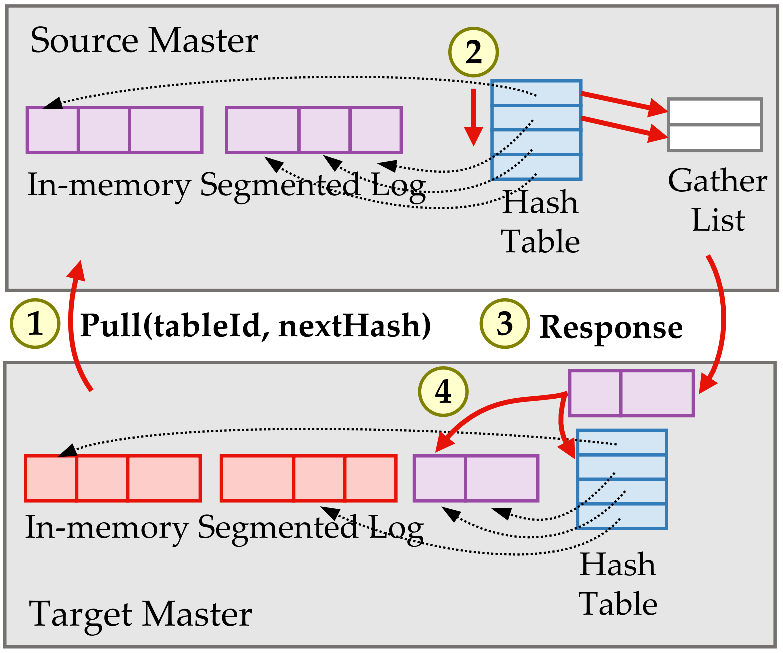

Figure 2.6: Overview of Rocksteady `Pulls`. A `Pull` RPC issued by the target iterates down a portion of the source's hash table and returns a batch of records. This batch is then logically replayed by the target into it's in-memory log and hash table.

2.3 Rocksteady Design

In order to keep its goal of fast migration that retains 99.9th 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 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 [33]. 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

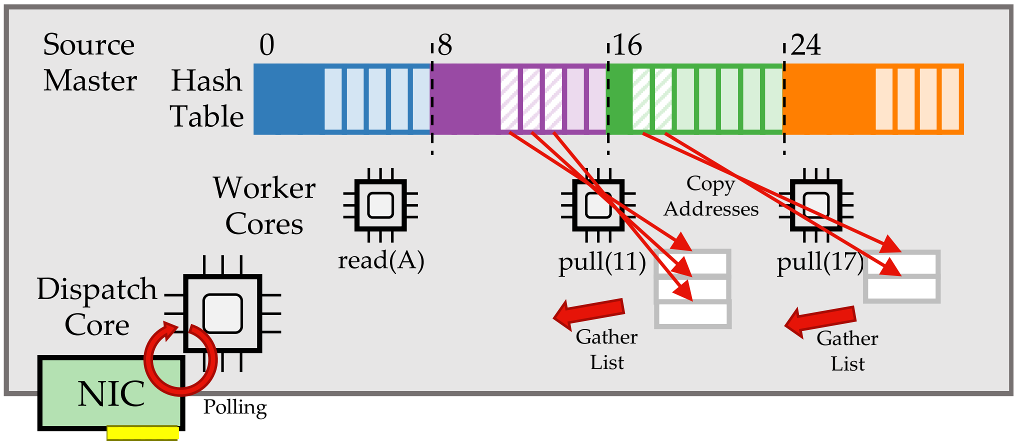

Figure 2.7: Source pull handling. `Pulls` work concurrently over disjoint regions of the source's hash table, avoiding synchronization, and return a fixed amount of data (20 KB, for example) to the target. Any worker core can service a `Pull` on any region, and all cores prioritize normal case requests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

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 `Pulls` 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 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 2.7 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

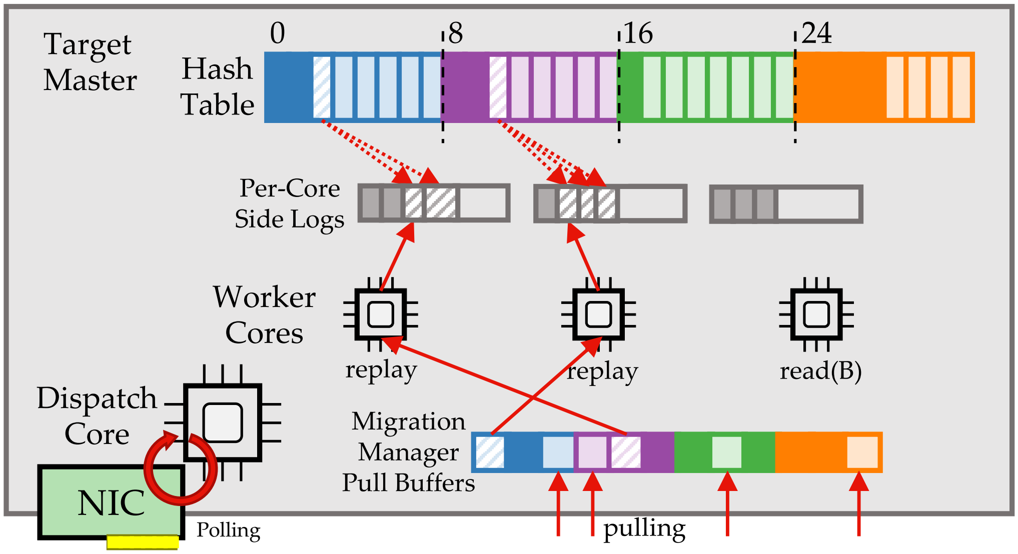

Figure 2.8: Target pull management and replay. There is one `Pull` outstanding per source partition. Pulled records are replayed at lower priority than normal requests. Each worker places records into a separate side log to avoid contention. Any worker core can service a replay on any partition.

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.

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 [41]; 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 2.8 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 (§**??**), 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 [33] 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 [8]. Transition to zero-copy will reduce memory bandwidth consumption [20], 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 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 but ongoing.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] *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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

[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] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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).

[34] 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.

[35] 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.

[36] 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.

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

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

[39] 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.

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

[41] 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.

[42] 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.

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

[44] 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.