Rocksteady: Fast Migration for Low-latency In-memory Storage

Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman
University of Utah

ABSTRACT

Scalable in-memory key-value stores provide low-latency access times of a few microseconds and perform millions of operations per second per server. With all data in memory, these systems should provide a high level of reconfigurability. Ideally, they should scale up, scale down, and rebalance load more rapidly and flexibly than disk-based systems. Rapid reconfiguration is especially important in these systems since a) DRAM is expensive and b) they are the last defense against highly dynamic workloads that suffer from hot spots, skew, and unpredictable load. However, so far, work on in-memory key-value stores has generally focused on performance and availability, leaving reconfiguration as a secondary concern.

We present Rocksteady, a live migration technique for the RAMCloud scale-out in-memory key-value store. It balances three competing goals: it migrates data quickly, it minimizes response time impact, and it allows arbitrary, fine-grained splits. Rocksteady migrates 758 MB/s between servers under high load while maintaining a median and 99.9th percentile latency of less than 40 and 250 µs, respectively, for concurrent operations without pauses, downtime, or risk to durability (compared to 6 and 45 µs during normal operation). To do this, it relies on pipelined and parallel replay and a lineage-like approach to fault-tolerance to defer re-replication costs during migration. Rocksteady allows RAMCloud to defer all repartitioning work until the moment of migration, giving it precise and timely control for load balancing.

ACM Reference Format:
Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. 2017. Rocksteady: Fast Migration for Low-latency In-memory Storage. In *Proceedings of SOSP '17*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3132747.3132784

SOSP '17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5085-3/17/10…$15.00
https://doi.org/10.1145/3132747.3132784

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 [11, 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 [8, 30]. Several works address the general problem of online (or *live*) data migration for scale-out stores [5, 8–10, 13, 14, 41], 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 [10]. 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 [28] 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 (§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 [10, 11]. 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 predefined splits. Second, DRAM-based storage is expensive; during normal operation, RAMCloud's log cleaner [37] 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.

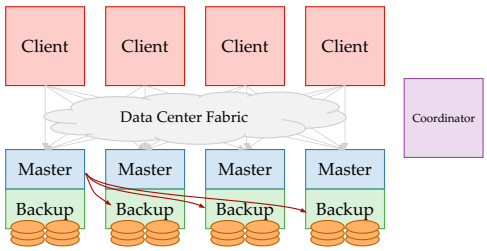

Figure 1: The RAMCloud architecture. Clients issue remote operations to RAMCloud storage servers. Each server contains a master and a backup. The master component exports the DRAM of the server as a large key-value store. The backup accepts updates from other masters and records state on disk used for recovering crashed masters. A central coordinator manages the server pool and maps data to masters.

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 [45]; 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) 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 (§2.3). We describe Rocksteady's full design (§3) including its fault tolerance strategy, and we evaluate its performance with emphasis on migration speed and tail latency impact (§4). 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.

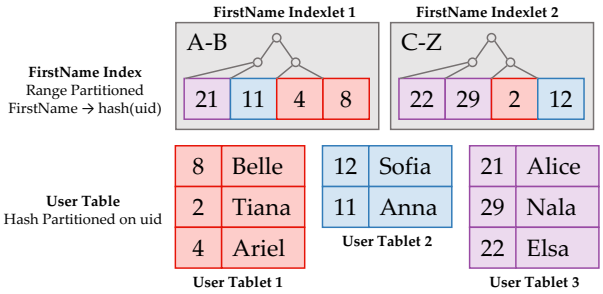

Figure 2: Index partitioning. Records are stored in unordered tables that can be split into tablets on different servers, partitioned on primary key hash. Indexes can be range partitioned into indexlets; indexes only contain primary key hashes. Range scans require first fetching a list of hashes from 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.

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 (§4).

Each server (Figure 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 [37].

RAMCloud's design and data model tightly intertwine with load balancing and 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 [22]. 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). 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].

2.1 Why Load Balance?

All scale-out stores need some way to distribute load. Most systems today use some form of consistent hashing [10, 39, 42]. 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 two-phase commit [4, 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 3 explores this effect. It consists of a microbenchmark run with 7 servers and 14 client machines. Clients issue back-to-back multiget operations evenly across the cluster, each for 7 keys at a time. In the experiment, clients vary which keys they request in each multiget to vary how many servers they must issue parallel requests to, but all servers still handle the same number of requests as each other. At Spread 1, all of the keys for a specific multiget come from one server. At Spread 2, 6 keys per multiget come from one server, and the 7^{th} key comes from another server. At Spread 7, each of the 7 keys in the multiget is serviced by a different server.

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

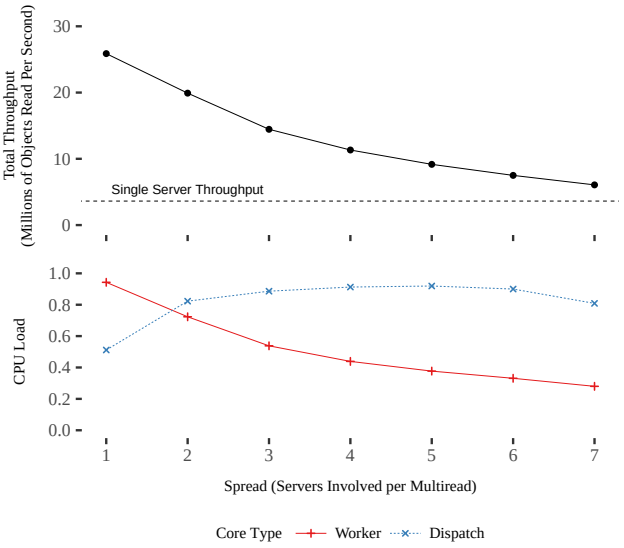

Figure 3: Throughput and CPU load impact of access locality. When multigets always fetch data from a single server (Spread 1) throughput is high and worker cores operate in parallel. When each multiget must fetch keys from many machines (Spread 7) throughput suffers as each server becomes bottlenecked on dispatching requests.

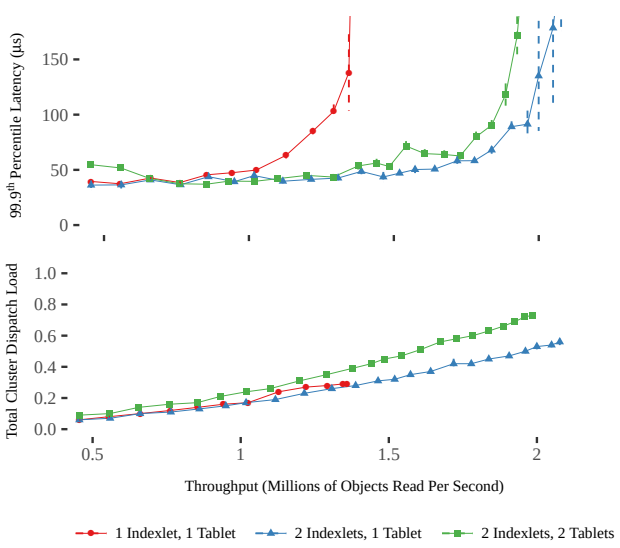

Figure 4: Index scaling as a function of read throughput. Points represent the median over 5 runs, bars show standard error. Spreading the backing table across two servers increases total dispatch load and the 99.9th percentile access latency for a given throughput when compared to leaving it on a single server.

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 [8].

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

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

2.2 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 [15], 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 [16], 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 [13]. 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?

2.3 Barriers to Fast Migration

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

This basic mechanism is faster than most approaches, but it is still orders of magnitude slower than what hardware can support. Figure 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.

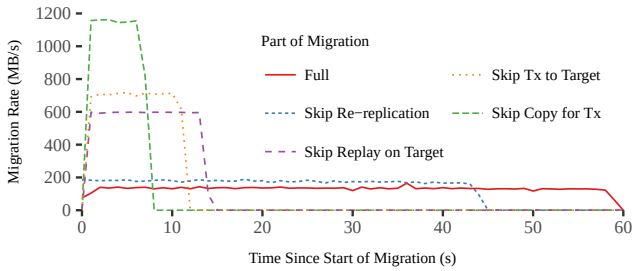

Figure 5: Bottlenecks using log replay for migration. Target side bottlenecks include logical replay and re-replication. Copying records into staging buffers at the source has a significant impact on migration rate.

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).

2.4 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

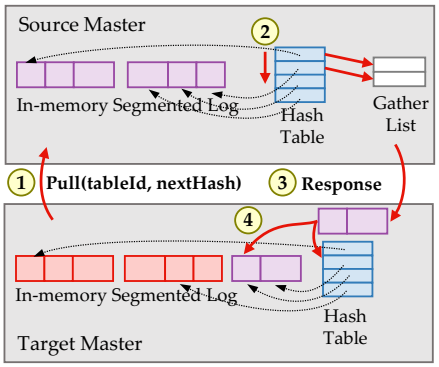

Figure 6: Overview of Rocksteady `Pulls`. A `Pull` RPC issued by the target iterates down a portion of the source's hash table and returns a batch of records. This batch is then logically replayed by the target into it's in-memory log and hash table.

target waiting for responses from backups, and it burns memory bandwidth. Rocksteady's approach is inspired by lineage [45]; a target server takes a temporary dependency on the source's log data to safely eliminate log replication from the migration fast path (§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 (§3).

Parallelism on Both the Target and Source. Log replay needn't be single threaded. A target is likely to be underloaded, so parallel replay makes sense. Rocksteady's parallel replay can incorporate log records at the target at more than 3 GB/s (§4.5). Similarly, source-side migration operations should be pipelined and parallel. Parallelism on both ends requires care to avoid contention.

Load-Adaptive Replay. Rocksteady's migration manager minimizes impact on normal request processing with fine-grained low-priority tasks [25, 34]. Rocksteady also incorporates into RAMCloud's transport layer to minimize jitter caused by background migration transfers (§3.1).

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 (§2.3). 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 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.

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.

3.1.1 Source-side Pipelined and Parallel Pulls. The source's only task during migration is to respond to `Pull` and `PriorityPull` messages with sufficient parallelism to keep the target busy. While concurrency would seem simple to handle, there is one challenge that complicates the design. A single `Pull` can't request a fixed range of keys, since the target does not know ahead of time how many keys within that range will exist in the tablet. A `Pull` of a fixed range of keys could contain too many records to return in a single response, which would violate the scheduling requirement for short tasks. Or, it could contain no records at all, which would result in `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.

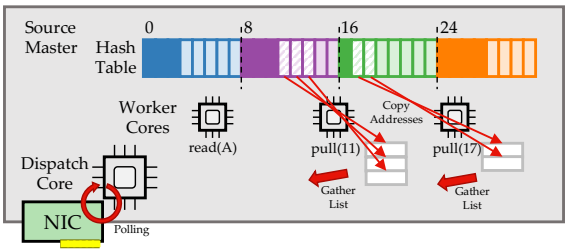

Figure 7: Source pull handling. `Pulls` work concurrently over disjoint regions of the source's hash table, avoiding synchronization, and return a fixed amount of data (20 KB, for example) to the target. Any worker core can service a `Pull` on any region, and all cores prioritize normal case requests over `Pulls`.

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 7 shows how this works. Since round-trip delay is similar to source pull processing time, a small constant factor more partitions than worker cores is sufficient for the target to keep any number of source workers running fully-utilized.

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

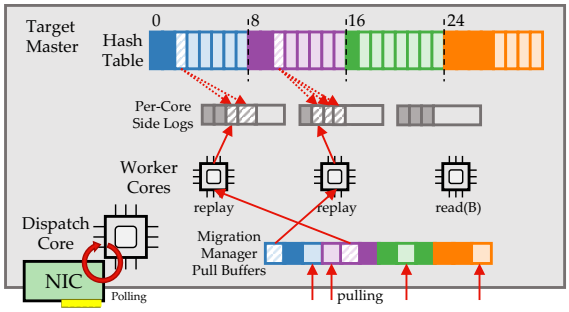

Figure 8: Target pull management and replay. There is one Pull outstanding per source partition. Pulled records are replayed at lower priority than normal requests. Each worker places records into a separate side log to avoid contention. Any worker core can service a replay on any partition.

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 [40]; 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 (§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 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.

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 (§4.5), but parallel replay where cores share a common log would also break down due to contention. Eliminating contention is key for fast migration.

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

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 [2]. Transition to zero-copy will reduce memory bandwidth consumption [23], but source-side memory bandwidth is not saturated during migration.

3.3 Priority Pulls

`PriorityPull`s work similarly to normal `Pull`s but are triggered on-demand by incoming client requests. A `PriorityPull` targets specific key hashes, so it doesn't require the coordination that `Pull`s do through partitioning. The key consideration for `PriorityPull`s 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. `PriorityPull`s 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 `PriorityPull`s 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 `PriorityPull`s 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 `PriorityPull`s 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.

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.

4 EVALUATION

To evaluate Rocksteady, we focused on five key questions:

How fast can Rocksteady go and meet tight SLAs? §4.2 shows Rocksteady can sustain migration at 758 MB/s with 99.9th percentile access latency of less than 250 µs.

Does lineage accelerate migration? Lineage and deferred log replication allow Rocksteady to migrate data 1.4× faster than synchronous re-replication, while shifting load from the source to the target more quickly (§4.2).

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

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

What is the impact at the source and target? §4.3 shows that regardless of workload skew, Rocksteady migrations cause almost no increase in source dispatch load, which is the source's most scarce resource for typical read-heavy workloads. Background `Pull`s add about 45% worker CPU utilization on the source, and Rocksteady effectively equalizes CPU load on the source and target. Dispatch load due to the migration manager on the target is minimal.

Are asynchronous batched priority pulls effective? §4.4 shows that asynchronous priority pulls are essential in two ways. First, synchronous priority pulls would increase both dispatch and worker load during migration due to the increased number of RPCs to the source and the wasted effort waiting for `PriorityPull` responses. Second, asynchronous batched `PriorityPull`s reduce load at the source fast enough to help hide the extra load due to background `Pull`s on the source, which is key to Rocksteady's fast transfer.

What limits migration? §4.5 shows that the source and target can send/consume small records at 5.7 GB/s and 3 GB/s, respectively; for small records target replay limits migration more than networking (5 GB/s today). Target worker cores spend 1.8 to 2.4× more cycles processing records during migration than source worker cores.

4.1 Experimental Setup

All evaluation was done on a 24 server Dell c6220 cluster on the CloudLab testbed [36] (Table 1). RAMCloud is transport agnostic; it offers RPC over several hardware and transport protocol combinations. For these experiments, servers were interconnected with 40 Gbps Ethernet and Mellanox ConnectX-3 cards; hosts used DPDK [1] and the `mlx4` poll-mode driver for kernel-bypass support. Each RAMCloud server used one core solely as a dispatch core to manage the network; it used 12 additional cores as workers to process requests; the remaining three cores helped prevent interference from background threads. The dispatch core runs a user-level reliable transport protocol on top of Ethernet that provides flow control, retransmission, etc. without the overhead of relying on the kernel TCP stack.

To evaluate migration under load, 8 client machines run the YCSB-B [7] workload (95% reads, 5% writes, keys chosen according to a Zipfian distribution with $\theta = 0.99$), which accesses a table on the source server. The table consists of 300 million 100 B record payloads with 30 B primary keys constituting 27.9 GB of record data consuming 44.4 GB of in-memory log on the source. Clients offer a nearly open load to the cluster sufficient to keep a single server at 80% (dispatch) load. While the YCSB load is running, a migration is triggered that live migrates half of the records from the source to the target.

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

4.2 Migration Impact and Ownership

Figures 9 and 10 (a) show Rocksteady's impact from the perspective of the YCSB clients. Migration takes 30 s and transfers at 758 MB/s. Throughput drops when ownership is transferred at the start of migration, since the clients must wait for records to arrive at the target. As records are being transferred, 99.9th percentile end-to-end response times start at 250 µs and taper back down to 183 µs as hot records from `PriorityPull`s arrive at the target. After migration, median response times drop from 10.1 µs to 6.7 µs, since each server's dispatch is under less load. Likewise, after migration moves enough records, throughput briefly exceeds the before-migration throughput, since client load is open and some requests are backlogged.

Figures 9 and 10 (b) show `PriorityPull`s are essential to Rocksteady's design. Without `PriorityPull`s, client requests for a record cannot complete until they are moved by the `Pull`s, resulting in requests that cannot complete until migration is done. Only a small fraction of requests complete while the migration is ongoing, and throughput is elevated after migration for a longer period. In practice, this would result in timeouts for client operations. Migration speed is 19% faster (904 MB/s) without `PriorityPull`s enabled.

Instead of transferring ownership to the target at the start of migration, another option is to leave ownership at the source during migration while synchronously re-replicating

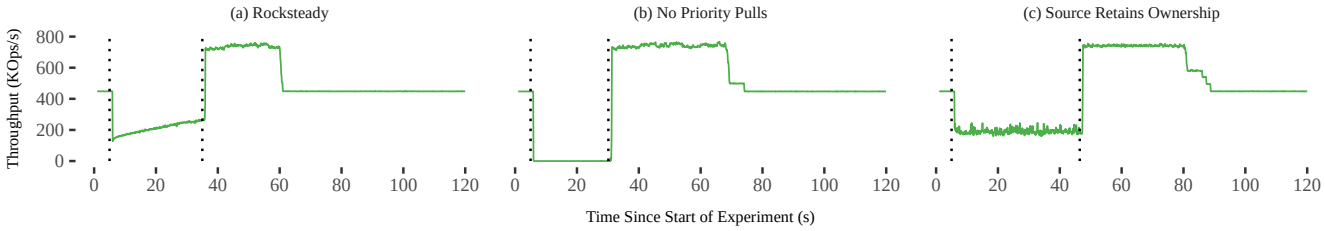

Figure 9: Running total YCSB-B throughput for (a) Rocksteady, (b) Rocksteady with no `PriorityPulls`, and (c) when ownership is left at the source throughout the migration. Dotted lines demarcate migration start and end.

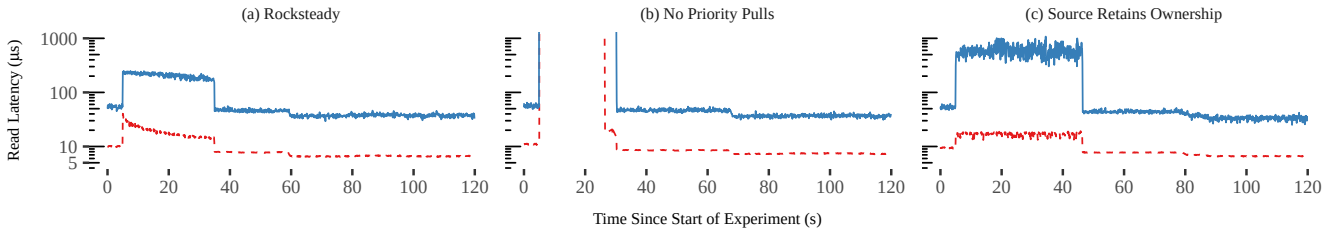

Figure 10: Running median (dashed line) and 99.9th percentile (solid line) client-observed access latency on YCSB-B for (a) Rocksteady, (b) Rocksteady with no `PriorityPulls`, and (c) when ownership is left at the source throughout.

migrated data at the target. Figures 9 and 10 (c) explore this approach. The main drawback is that it cannot take advantage of the extra resources that the target provides. Similar to the case above, source throughput decreases under migration load, and clients eventually fall behind. For long migrations, this can lead to client timeouts in a fully open load, since throughput would drop below offered load for the duration of migration. Additionally, migration suffers a 27.7% slowdown (758 MB/s down to 549 MB/s), and the impact on the 99.9th percentile access latency is worse than the full Rocksteady protocol because of the re-replication load generated by the target interfering with the replication load generated by writes at the source. For larger RAMCloud clusters, such interference will not be an issue, and one would expect the 99.9th percentile to be similar to Rocksteady.

4.3 Load Impact

Figure 11 (a) shows Rocksteady immediately equalizes dispatch load on the source and target. Worker and dispatch load on the target jumps immediately when migration starts, offloading the source. Clients refresh their stale tablet mappings after migration starts. Dispatch is immediately equalized because a) exactly half of the table ownership has been shifted to the target, and b) the migration manager is asynchronous and requires little CPU.

A key goal of Rocksteady is to shift load quickly from the source to the target. Most workloads exhibit some skew, but the extent of that skew impacts Rocksteady's ability to shift load quickly. Figure 12 examines the extent to which Rocksteady's effectiveness at reducing client load is skew dependent. With no skew (uniform access, skew $\theta = 0$) `PriorityPulls` are sufficient to maintain client access to the tablet, but low request locality means the full load transfer only proceeds as quickly as the background pulls can transfer records. Overall, the results are promising when considering the source's dispatch load, which is its most scarce resource for typical read-heavy workloads. Regardless of workload skew, source-side dispatch load remains relatively flat from the time migration starts until it completes. This means that Rocksteady's eager ownership transfer enabled by batched `PriorityPulls` makes up for any extra dispatch load the `Pulls` place on the source regardless of the skew.

4.4 Asynchronous Batched Priority Pulls

Figures 13 and 14 compare asynchronous batched `PriorityPulls` with the naïve, synchronous approach when background `Pulls` are disabled. The asynchronous approach doesn't help tail latency: 99.9th latency stays consistent at 160 µs for the rest of the experiment, but median access latency drops to 7.4 µs immediately. On the other hand, the synchronous approach results in median latency jitter, primarily due to workers at the target waiting for `PriorityPulls` to return, which can be seen in the increased worker utilization at

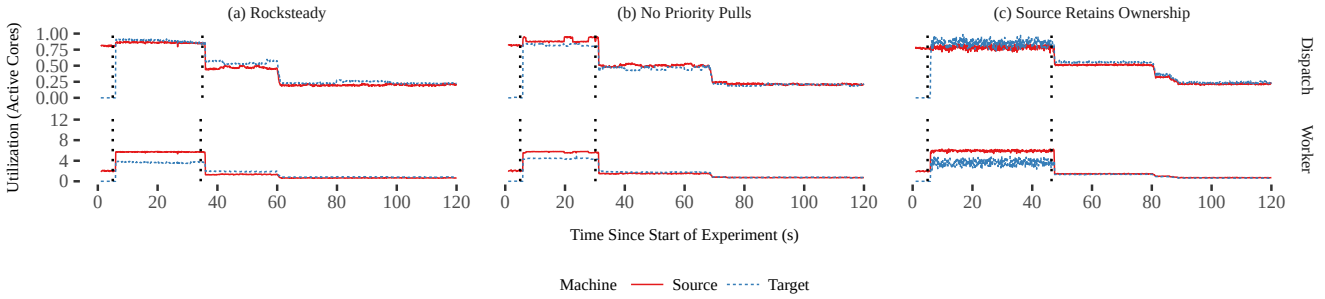

Figure 11: Dispatch core and worker core utilization on both source and target for (a) Rocksteady, (b) Rocksteady with no `PriorityPulls`, and (c) when ownership is left at the source throughout the migration.

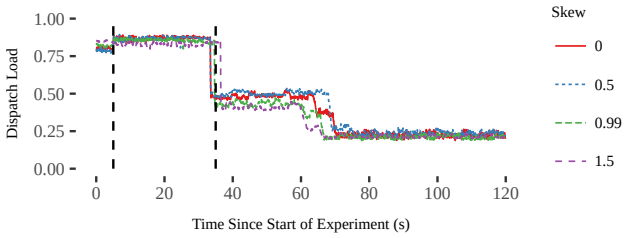

Figure 12: Impact of workload access skew on source-side dispatch load. Batched `PriorityPulls` hide the extra dispatch load of background `Pulls` regardless of access skew.

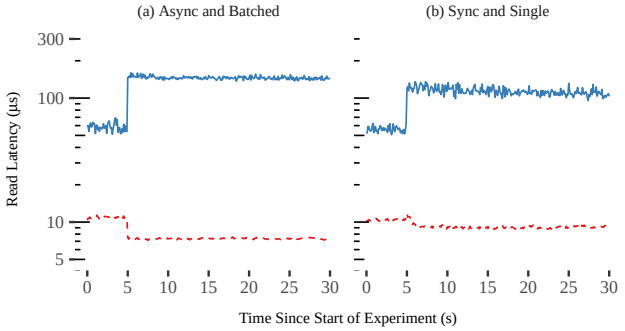

Figure 13: Median (dashed line) and 99.9th percentile (solid line) access latency without background `Pulls`. Async batched `PriorityPulls` restore median latency almost immediately compared to sync `PriorityPulls`.

the target (Figure 14b). However, 99.9th percentile access latency is lower than the asynchronous approach since pull responses are sent to waiting clients immediately.

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

4.5 Pull and Replay Scalability

Parallel and pipelined pulls and replay are key to migration speed that interleaves with normal case request processing. The microbenchmark shown in Figure 15 explores the scalability of the source and target pull processing logic. In the experiment, the source and target pull/replay logic was run in isolation on large batches of records to stress contention and to determine the upper bound on migration speed at both ends independently.

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

5 DISCUSSION

Some of the most broadly applicable lessons from Rocksteady are on the interplay of partitioning, dispatch, and synchronization. Recent works have often partitioned operations [3, 20] or sometimes just mutating operations [27] to

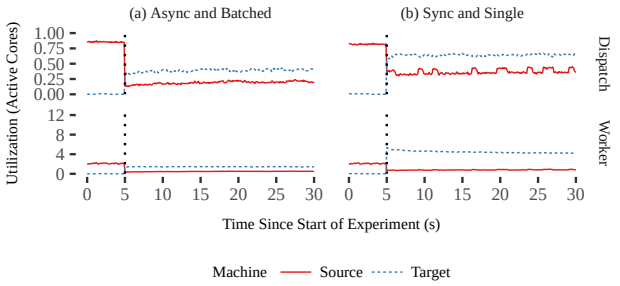

Figure 14: CPU Load with no background `Pull`s. Asynchronous batched `PriorityPull`s improve dispatch and worker utilization at both the source and target compared to synchronous `Pull`s that stall target worker cores.

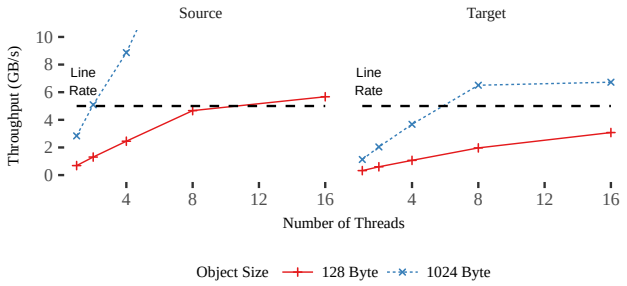

Figure 15: Source and target parallel migration scalability. Source side pull logic can process small 128 B objects at 5.7 GB/s. Target side replay logic can process small 128 B objects at 3 GB/s. For larger objects, neither side limits migration.

reduce locking and contention. Systems that strictly partition work (even just writes) are likely to have to reconfigure more often under skew. Their access latencies also suffer, since migration must be interleaved with normal execution. RAMCloud's dispatch can be a bottleneck, but it can also redirect any idle CPU resources on few microseconds timescale, which is key to Rocksteady's adaptive parallel replay and tight SLAs. Hardware-assisted [21], client-assisted [27], and parallel dispatch help mitigate bottlenecks and delay the need for migration, but none of these can eliminate the need for cross-machine rebalancing or the need to overlap normal execution and migration. Optimizing for normal-case, steady-state request processing can make inevitable background system tasks more costly. Designers of in-memory systems must carefully navigate partitioning, dispatch, and locking trade-offs when planning for heavy rebalancing operations, like migration.

Rocksteady's safe deferred re-replication can also be applied to other systems. For example, H-Store with the Squall [13] migration system could exploit the same idea to improve migration throughput and access distribution impact. Squall could take a temporary dependency on source data and backups to avoid synchronous re-replication at the target; this would have a significant impact since re-replication blocks execution on the whole target partition in Squall.

5.1 Going Even Faster

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

To achieve such gains without destroying normal case request processing, migration might be limited to merely transferring large, opaque memory regions between hosts, with little-to-no packaging or replay work on either end. This would require the source to keep state strictly physically partitioned in fine enough units for it to satisfy all possible future splits. FaRM's data layout for example, meets these properties [12].

Physically partitioning groups of records on key or key hash would constrain RAMCloud's log structured memory cleaning. The cleaner minimizes cleaning CPU and memory bandwidth load by physically colocating records that are likely to have a similar lifetime [37]. With physical partitioning constraints, the cleaner wouldn't be able to globally optimize hot/cold separation of objects. Investigating the cleaner's sensitivity to such partitioning could be an interesting direction, particularly since it might be able to assist in the process of physically partitioning records.

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

Beyond improvements in dispatch scalability, other improvements to RAMCloud's concurrency model could also have a significant impact on Rocksteady. Today, RAMCloud processes requests on workers that use standard kernel threads. Coroutines or cooperative user-level threading could both improve response distributions and efficiency [19]. If `Pull` and replay operations could afford frequent yields to RAMCloud's dispatch, heavy operations would have less impact on normal case request processing. Replay and `Pull` operations could be coarser as well, resulting in less requests and lower dispatch overheads. This could allow Rocksteady to transfer data even more quickly with the same SLAs.

6 RELATED WORK

Amazon's Dynamo [10] is a highly-available distributed key-value store that pushed for focus on 99.9th percentile access latency, though Rocksteady pushes for tail latency nearly 1,000× lower even while migrating. Dynamo supported strong SLAs and reconfiguration through a very different approach that took advantage of pre-partitioning records inside each server, replication, and weak consistency. DRAM is expensive, so Rocksteady must not rely on in-memory replication or internal pre-partitioning of records.

Distributed database live migration has received a great deal of attention, particularly for multi-tenant cloud databases. Rocksteady uses many ideas from prior work like pacing migration [5], eager transfer of ownership [13, 14], and combining on-demand and background migration [13, 14, 38]. Others have explored holding ownership at the source and "catching up" the target through delta records or recovery log data [9], similar to RAMCloud's original migration.

Squall [13, 41] is a state-of-the-art live migration system for the H-Store [20] scale-out shared-nothing database. It offloads the source quickly by breaking requested tuples out into separate units and migrating them on-demand. Under skewed loads, hot tuples move quickly and background transfers are paced to try to minimize disruption. Rocksteady uses Squall's combined background/tuple-level reactive pull, but it extends the approach to RAMCloud's more flexible parallelism model. H-Store's strict serial execution makes synchronizing with migration expensive; the execution of migration operations on a partition are interlocked between the source and target and block normal requests. That is, each pull from a target core can only be serviced by a specific source core, and pulls and replays must operate in isolation on a partition. Requests cannot be processed for keys that are being pulled (or for *any* key in a partition where a pull is ongoing). Target cores also spin waiting for pull responses hurting normal request access latency and throughput as well as migration speed. Compared to all prior approaches, Rocksteady transfers data an order of magnitude faster with tail latencies 1,000× lower; in general, Rocksteady's ability to use *any* available core for *any* operation is key for both tail latency and migration speed.

Rocksteady builds on recent work on recovery and dispatching for in-memory storage that relies on kernel-bypass networking [6, 17–19, 26, 27, 29, 35]. RAMCloud's recovery is a form of distributed migration [32], but it is disruptive since it uses the resources of the entire cluster to reload contents of a crashed server as fast as possible. FaRM [11, 12] relies on in-memory triplication for redundancy, but it must re-replicate lost partitions when a server fails. It paces recovery to a few hundred megabytes per second per server in order to minimize performance impact. Similarly, DrTM-B [44] minimizes the impact of reconfiguration by relying on in-memory replicas. However, replicas can become overloaded too, so data is migrated using parallel RDMA reads. One key aspect of FaRM is that partitions are physical: a lost partition is an opaque region of memory, so most of the overhead of re-replication is network transfer. RAMCloud migration is more complex, since the source and target don't share a common partitioning or physical memory layout.

Rocksteady's fast parallel packaging and replay is similar to Silo's single-server parallel recovery [43, 46]. Silo partitions recovery logs across cores during record and during replay. Rocksteady's replay doesn't require any particular order; any core can replay any portion of records, which helps Rocksteady hit SLAs. In Silo the database is also naturally offline during replay, and recovery can consume all of the resources of the machine. Silo's parallel replay is state-of-the-art, but Rocksteady's parallel replay outperforms it on far fewer cores. This may be because Silo must reconstruct a tree-like index rather than a flat hash table and filesystem I/O may induce more overhead than a NIC using kernel-bypass.

7 CONCLUSION

Low-latency in-memory stores are designed to tolerate the heaviest request loads, but if they are too stripped down they cannot deal with complex higher-level operations like reacting to workload changes, skew shifts, and load spikes. Rocksteady is a migration protocol for in-memory key-value stores that avoids the need for and overhead of in-advance state partitioning; it eliminates replication overhead from the migration fast path; it exploits parallelism; and it exploits modern NIC hardware. Rocksteady has a "pay-as-you-go" approach that helps avoid overloading the source during migration using asynchronous batched on-demand pulls to shift load away from the source as parallel background transfers proceed. In all, Rocksteady can move the entire DRAM of a modern data center machine in a few minutes while retaining 99.9th percentile tail latency of less than 250 µs.

ACKNOWLEDGMENTS

Thanks to the Stanford RAMCloud team for creating a great system to build upon. Thanks to Kirk Webb and the CloudLab team for help with our special networking requests. Thanks to the anonymous reviewers for their comments and to our shepherd, Dan Ports. This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1566175 and CNS-1338155. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work was also supported in part by Facebook and VMware.

REFERENCES

[1] Data Plane Development Kit. http://dpdk.org/. 4/10/2017.

[2] 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.

[3] Redis. http://redis.io/. 7/24/2015.

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

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

[6] Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 49–65.

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

[8] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. 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.

[9] Das, S., Nishimura, S., Agrawal, D., and El Abbadi, A. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration. *Proc. VLDB Endow. 4*, 8 (May 2011), 494–505.

[10] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Operating Systems Review 41*, 6 (Oct. 2007), 205–220.

[11] Dragojević, A., Narayanan, D., Castro, M., and Hodson, O. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 401–414.

[12] Dragojević, A., Narayanan, D., Nightingale, E. B., Renzelmann, M., Shamis, A., Badam, A., and Castro, M. No compromises: distributed transactions with consistency, availability, and performance . In *SOSP* (2015), pp. 85–100.

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

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

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

[16] IEEE. 802.3-2015 - IEEE Standard for Ethernet. https://standards.ieee.org/findstds/standard/802.3-2015.html.

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

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

[19] Kalia, A., Kaminsky, M., and Andersen, D. G. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, Nov. 2016), USENIX Association, pp. 185–201.

[20] Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. P. C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., and Abadi, D. J. H-store: A High-performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow. 1*, 2 (Aug. 2008), 1496–1499.

[21] Kaufmann, A., Peter, S., Sharma, N. K., Anderson, T., and Krishnamurthy, A. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2016), ASPLOS '16, ACM, pp. 67–81.

[22] Kejriwal, A., Gopalan, A., Gupta, A., Jia, Z., Yang, S., and Ousterhout, J. 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.

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

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

[25] Leis, V., Boncz, P., Kemper, A., and Neumann, T. 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* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 743–754.

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

[27] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 429–444.

[28] 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.

[29] Nelson, J., Holt, B., Myers, B., Briggs, P., Ceze, L., Kahan, S., and Oskin, M. 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.

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

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

[33] Ousterhout, J., Gopalan, A., Gupta, A., Kejriwal, A., Lee, C., Montazeri, B., Ongaro, D., Park, S. J., Qin, H., Rosenblum, M., Rumble, S., Stutsman, R., and Yang, S. The RAMCloud Storage System. *ACM Transactions on Computer Systems 33*, 3 (Aug. 2015), 7:1–7:55.

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

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

[36] Ricci, R., and Eide, E. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *; login: 39*, 6 (2014), 36–38.

[37] Rumble, S. M., Kejriwal, A., and Ousterhout, J. 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.

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

[39] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. 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] Stutsman, R., Lee, C., and Ousterhout, J. Experience with Rules-Based Programming for Distributed, Concurrent, Fault-Tolerant Code. In *USENIX ATC* (Santa Clara, CA, July 2015).

[41] Taft, R., Mansour, E., Serafini, M., Duggan, J., Elmore, A. J., Aboulnaga, A., Pavlo, A., and Stonebraker, M. E-store: Fine-grained Elastic Partitioning for Distributed Transaction Processing Systems. *Proc. VLDB Endow. 8*, 3 (Nov. 2014), 245–256.

[42] The Apache Software Foundation. Apache Cassandra. http://cassandra.apache.org/.

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

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

[45] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. 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.

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