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

presentation by Michał Naruniec

Based on paper by

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

From SOSP '17

Agenda

1. **Introduction**
2. Background: RAMCloud
3. Solution: Rocksteady
4. Evaluation

(Live) Migration

- Scaling up
- Scaling down
- Rebalancing

The better the migration process, the more often we can afford to use it.

What is a GOOD migration?

2 contradictory goals:

- minimize the impact on the system -> **go slow**?
- quickly go into effect; react to skew shifts, load spikes -> **go fast**?

In-memory stores migrations

- **Low-latency access times**
  - RAMCloud responds in tens of microseconds for 99.9th percentile
  - millisecond differences at scale could hurt client-observed performance
- **Growing DRAM storage**
  - > 512GiB per server
  - few hours of migration, unusable for scaling
- **High bandwidth networking**
  - > 200 Gbps InfiniBand networks
  - network stacks and NICs can struggle
  - won't be able to saturate the network without massive parallelism and kernel/CPU bypassing

Design goals for Rocksteady

- **Pauseless**
- **Lazy partitioning**
  - deferring decisions until migration time, avoiding constraints
  - pre-partitioning could hart RAMCloud's internal log cleaning
- **Low impact with minimum headroom**
  - keep latency reasonable
  - utilize resources as efficiently as possible

Key ideas for meeting the goals

- Adaptive parallel replay
- Exploiting workload skew to create source-side headroom
- Smart, RDD-inspired fault-tolerance
- Optimization for modern NICs

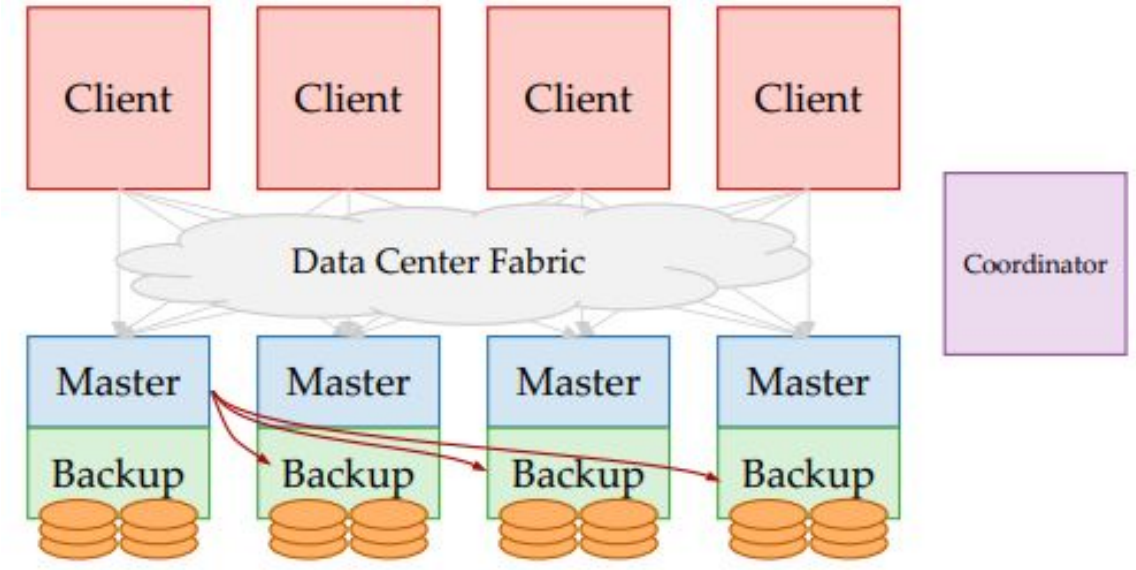
Agenda

1. Introduction
2. **Background: RAMCloud**
3. Solution: Rocksteady
4. Evaluation

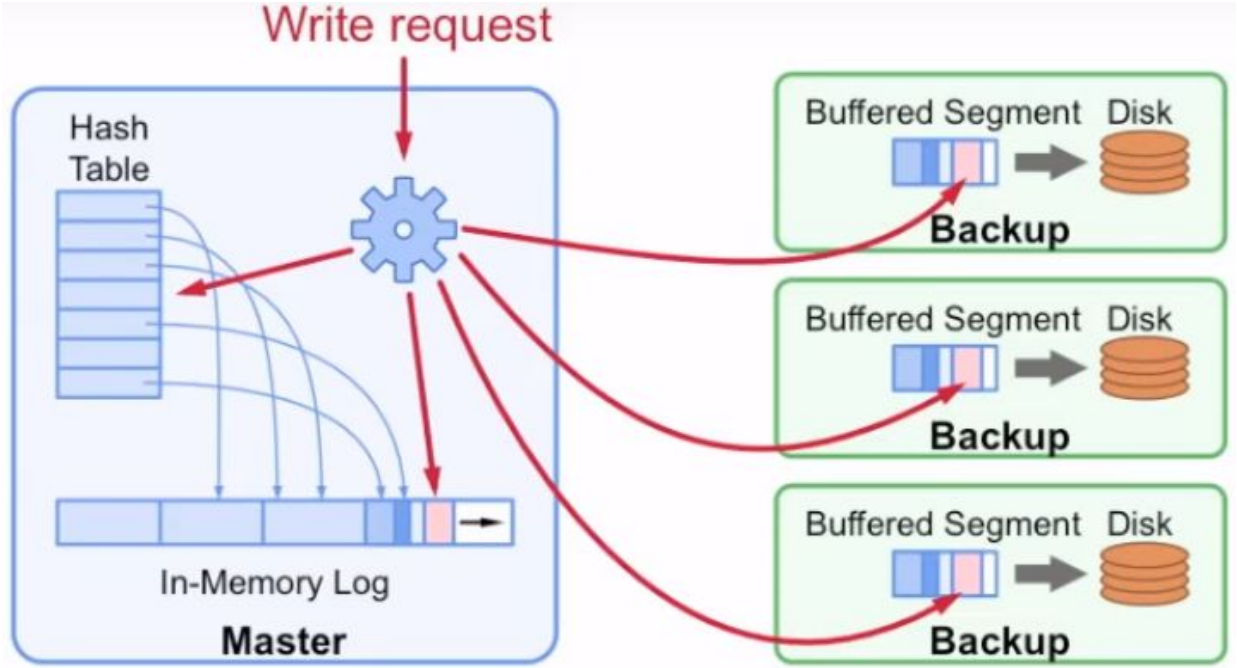
RAMCloud

- key-value store, multiple tables (dividable into tablets)
- can scale across thousands of servers
- all data always in DRAM
- focused on:
  - low median and tail latency (median 6 µs, 45 µs for 99.9th percentile)
  - durability
  - availability
  - effective usage of resources (**DRAM is expensive**, 80-90% utilization)
- no DRAM replication, only asynchronous (but durable) non-volatile replication
- availability by rapid recovery (1-2 seconds for reviving 60GiB server)

RAMCloud's architecture



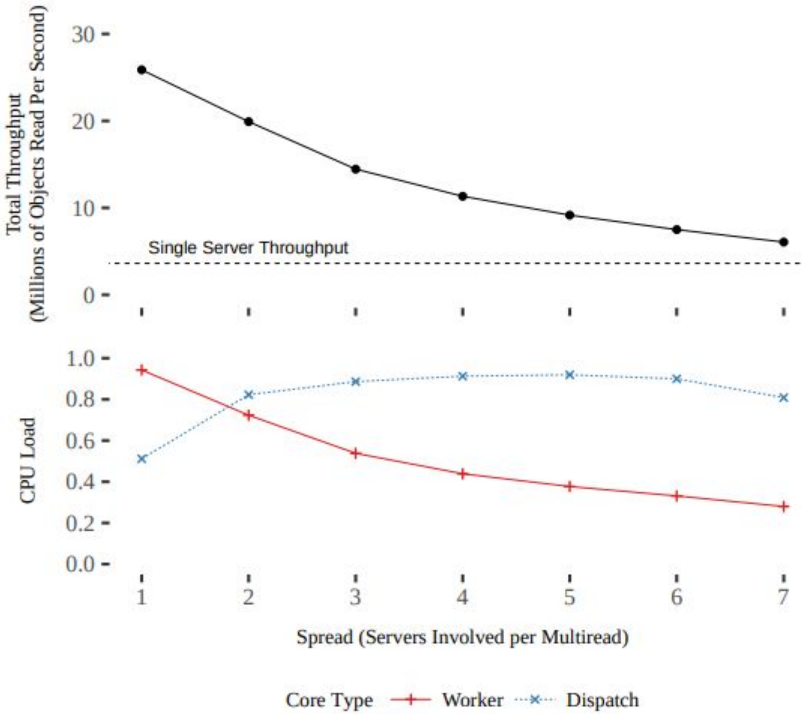
RAMCloud's durability



Other RAMCloud features

- core-bound dispatcher and workers without preemption
- multi-reads/writes
- transactions
- secondary indexes
  (partitionable into indexlets, independent from tables' locations)
- tablet profiling

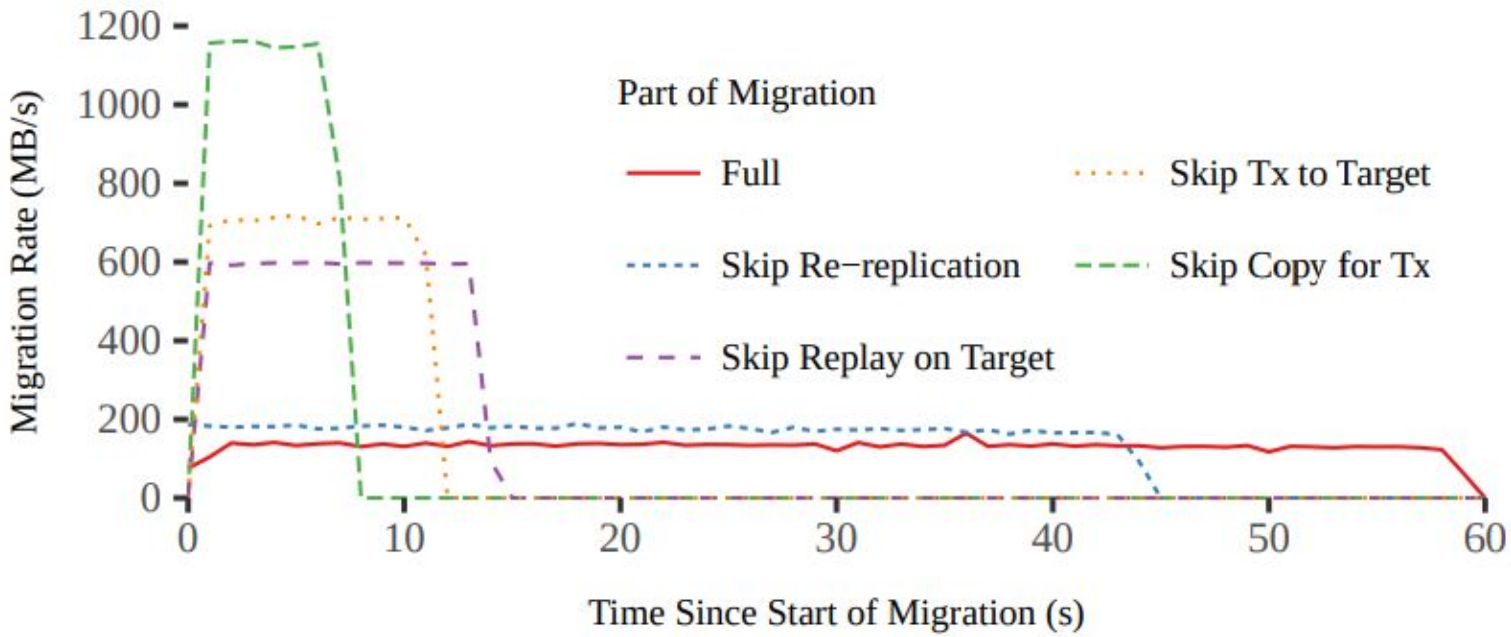
NB. multi-read locality



RAMCloud's built-in migration

1. Iterate over entries in source's log and copy relevant ones to staging buffers.
2. Send them to target.
3. Logically replay logs on the target and re-replicate.
4. After everything is transferred, pass tablet ownership to target.

RAMCloud's migration analysis



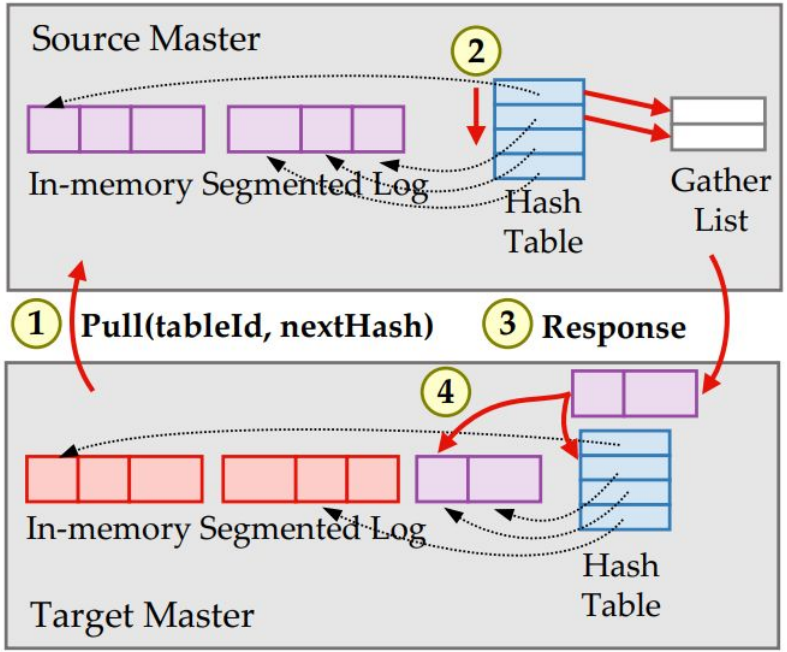
Agenda

1. Introduction
2. Background: RAMCloud
3. **Solution: Rocksteady**
4. Evaluation

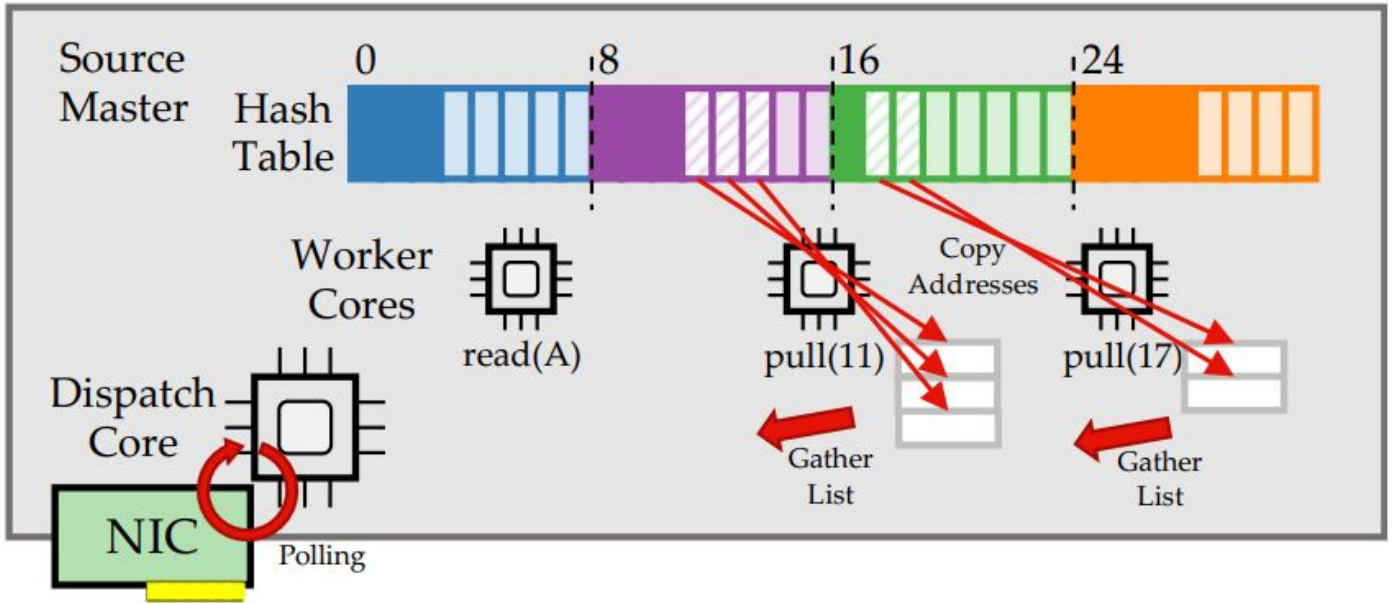
Rocksteady design criteria

- **no synchronous re-replication**
  - burns memory bandwidth
  - wastes CPU cycles for waiting for backups
- **immediate ownership transfer**
  - immediately shifts load
  - PriorityPulls
- **parallelism and pipelining on both target and source**
  - using scatter/gather RDMA for zero-copy transfers
- **load-adaptive replay**

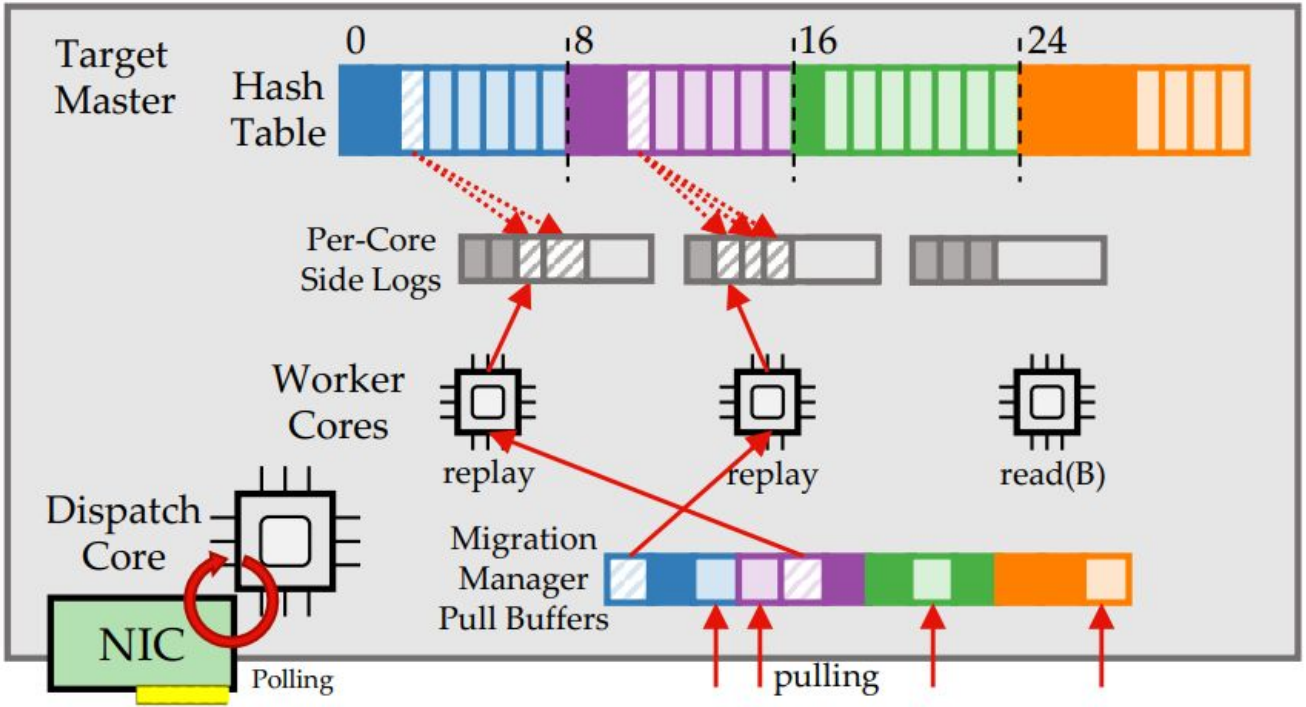
Migration loop



Source side


Source
Master
Hash
Table
0
8
16
24
Worker
Cores
read(A)
pull(11)
pull(17)
Copy
Addresses
Dispatch
Core
NIC
Polling
Gather
List
Gather
List

Target side


Target
Master
Hash
Table
0
8
16
24
Per-Core
Side Logs
Worker
Cores
replay
replay
read(B)
Dispatch
Core
NIC
Polling
Migration
Manager
Pull Buffers
pulling

Priority pulls

- highest priority
- asynchronous, client is told to retry after some time
- sent batched, no duplication

Safe, lazy replication

- client writes in the target are replicated normally
- migrated data is replicated at the end
- coordinator adds target performed writes as recovery dependency for source
- if something happens during migration, ownership goes back to source, which needs to restore target's changes

Agenda

1. Introduction
2. Background: RAMCloud
3. Solution: Rocksteady
4. **Evaluation**

Evaluation

- How fast and non-intrusive is it?
- Does lazy replication accelerate migration?
- What is the impact on source and target?
- Are asynchronous batched priority pulls effective?
- What limits migration?

Testbed

- 24 server CloudLab testbed; 1 coordinator, 8 clients, 15 masters
- at each node 1 dispatch core, 12 workers
- userspace Ethernet-based networking on dispatch core
- clients ran YCSB-B load (95% reads, 5% writes, keys chosen according to a Zipfian distribution with $\theta = 0.99$)
- 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
- every server kept at around 80% dispatch load
- Rocksteady settings: 8 parallelism units, each Pull returning 20 KB

Testbed

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

Throughput

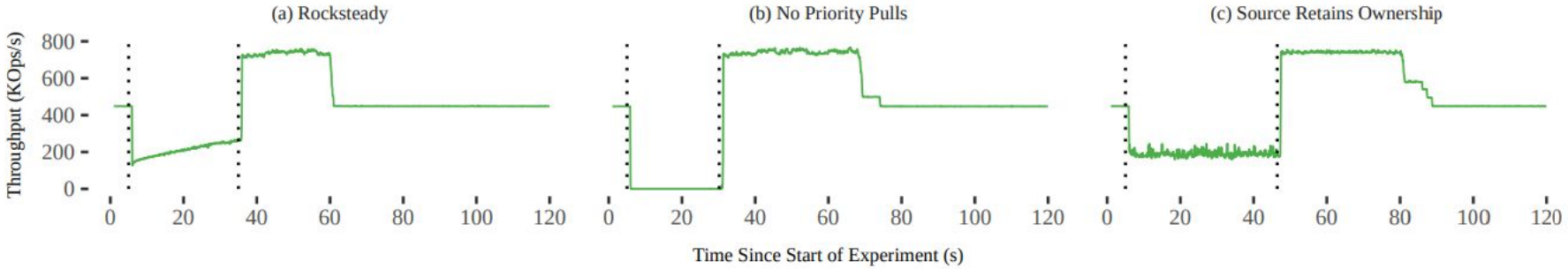


Figure 9: Running total YCSB-B throughput for (a) Rocksteady, (b) Rocksteady with no `PriorityPulls`, and (c) when ownership is left at the source throughout the migration. Dotted lines demarcate migration start and end.

Latency

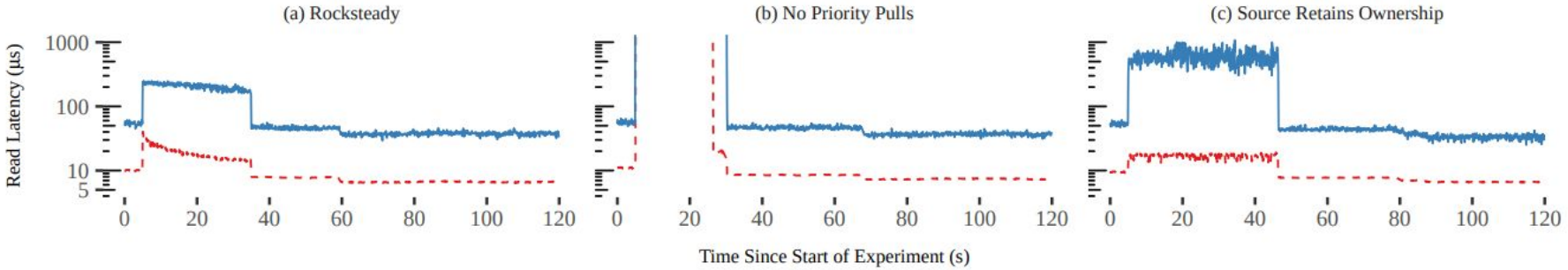


Figure 10: Running median (dashed line) and 99.9th percentile (solid line) client-observed access latency on YCSB-B for (a) Rocksteady, (b) Rocksteady with no `PriorityPulls`, and (c) when ownership is left at the source throughout.

CPU utilization

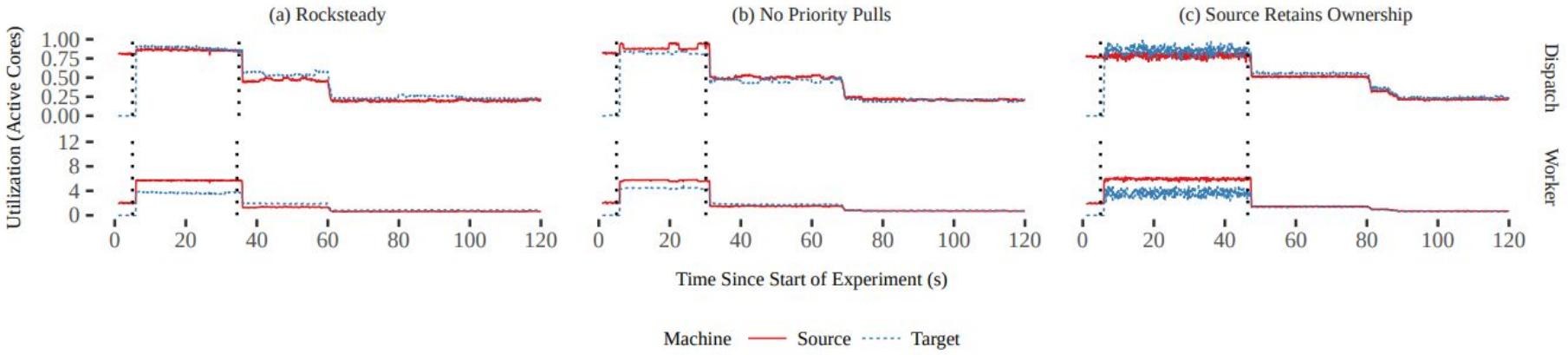


Figure 11: Dispatch core and worker core utilization on both source and target for (a) Rocksteady, (b) Rocksteady with no `PriorityPulls`, and (c) when ownership is left at the source throughout the migration.

Skew impact on CPU utilization

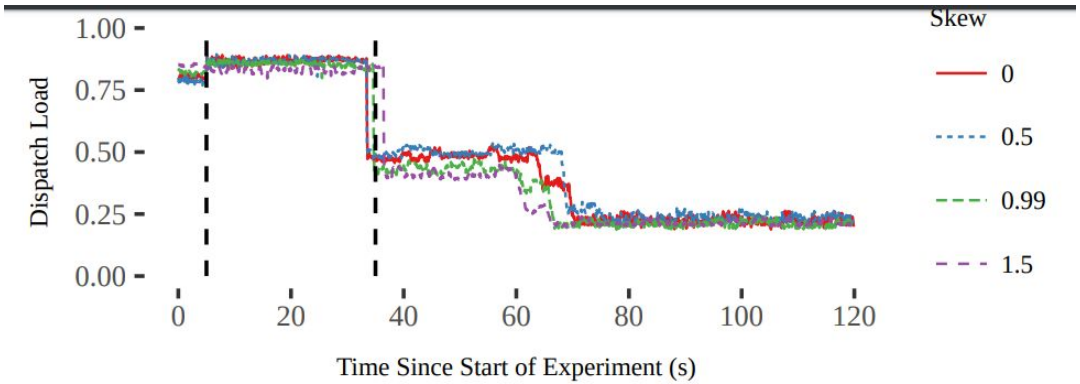


Figure 12: Impact of workload access skew on source-side dispatch load. Batched `PriorityPulls` hide the extra dispatch load of background `Pulls` regardless of access skew.

Async priority pulls latency

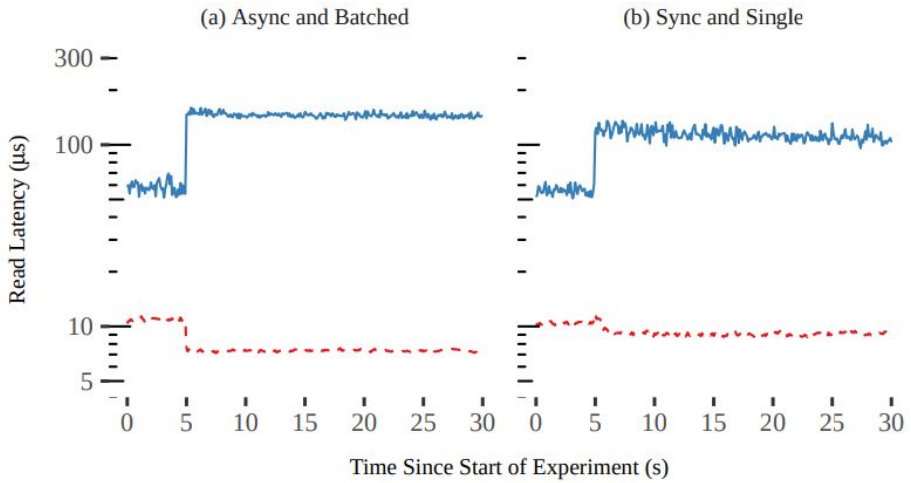


Figure 13: Median (dashed line) and 99.9th percentile (solid line) access latency without background `Pulls`. Async batched `PriorityPulls` restore median latency almost immediately compared to sync `PriorityPulls`.

Async priority pulls CPU utilization

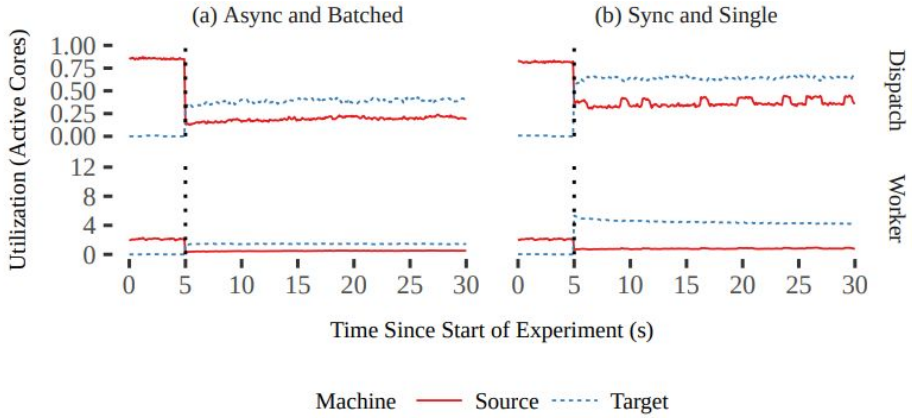


Figure 14: CPU Load with no background `Pulls`. Asynchronous batched `PriorityPulls` improve dispatch and worker utilization at both the source and target compared to synchronous `Pulls` that stall target worker cores.

Pulling scalability

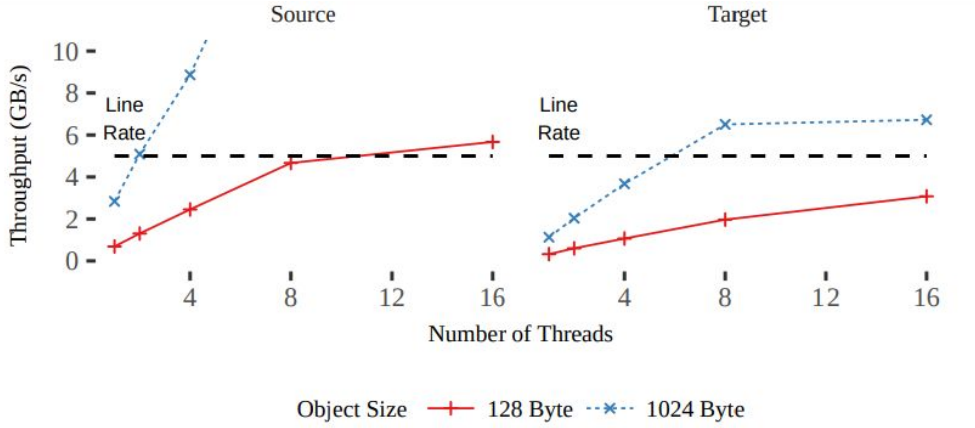


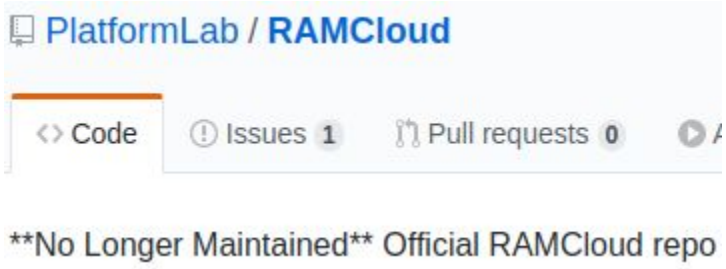
Figure 15: Source and target parallel migration scalability. Source side pull logic can process small 128 B objects at 5.7 GB/s. Target side replay logic can process small 128 B objects at 3 GB/s. For larger objects, neither side limits migration.

Conclusion

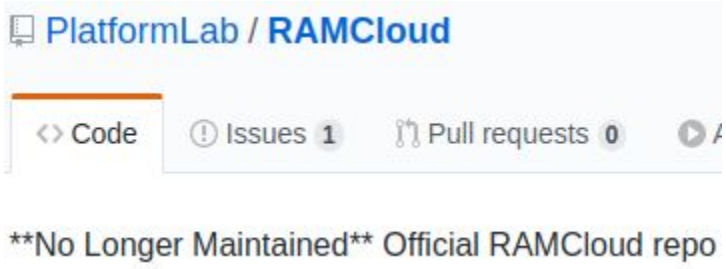
- 758 MB/s, ~30s migration
- almost no increase in source dispatch load
- for large records network limits migration
- 40/250 µs latency compared to normal 6/45 µs

The sad part

The sad part


PlatformLab / RAMCloud
Code
Issues 1
Pull requests 0
No Longer Maintained Official RAMCloud repo

The sad part



~~ramcloud-dev~~. **(As of 2019/10/7 support for RAMCloud has ended)**

It died so others could live

- **Dynamo** - pre-partitioning
- **Spanner**
- **FaRM and DrTM** - re-using in-memory redundancy
- **Squall** (for H-Store) - very similar, could use deferred re-replication

Thank you

References

1. Kulkarni et al. *Rocksteady: Fast Migration for Low-latency In-memory Storage* [with presentation] (2017)
2. Ousterhout - *RAMCloud: Scalable High-Performance Storage Entirely in DRAM* (presentation @ LinkedIn TechTalks, 2011)
3. Ousterhout et al. *The RAMCloud Storage System*. (2015)
4. Rumble et al. *Log-structured Memory for DRAM-based Storage*. (2014)
5. Ongaro et al. *Fast Crash Recovery in RAMCloud*. (2011)