

Towards Practical Low-Latency Key-Value Stores

Chinmay Kulkarni
University of Utah

Key-Value Stores are Used Everywhere!



Key-Value Stores are Used Everywhere!



Scaling Memcache at Facebook

Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li,
Ryan McElroy, Mike Pałeczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung,
Venkateshwaran Venkataramani

{rajeshn,hans}@fb.com, {sgrimm, marc}@facebook.com, {herman, hcli, rm, mpal, dpeek, ps, dstaff, ttung, veeve}@fb.com

Facebook Inc.

Abstract: Memcached is a well known, simple, in-memory caching solution. This paper describes how Facebook leverages memcached as a building block to construct and scale a distributed key-value store that supports the world's largest social network. Our system handles billions of requests per second and holds trillions of items to deliver a rich experience for over a bil-

however, web pages routinely fetch thousands of key-value pairs from memcached servers.

One of our goals is to present the important themes that emerge at different scales of our deployment. While qualities like performance, efficiency, fault-tolerance, and consistency are important at all scales, our experience indicates that at specific sizes some qualities re-

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati,
Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall
and Werner Vogels

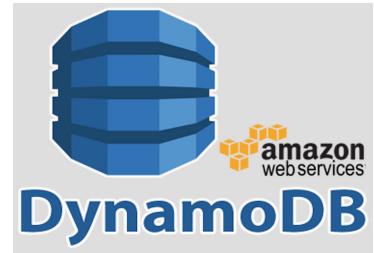
Amazon.com

ABSTRACT

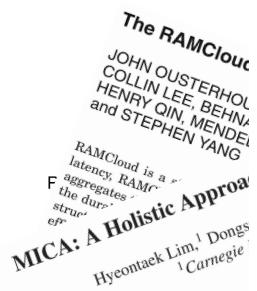
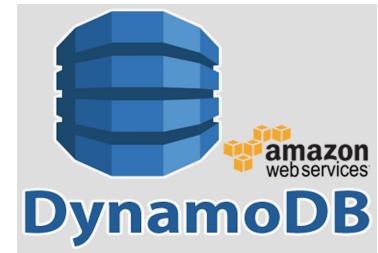
Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Key-Value Stores are Used Everywhere!



Key-Value Stores are Used Everywhere!



Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache

Xingda Wei, Rong Chen, Haibo Chen

Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University



Abstract

MICa is a scalable in-memory key-value store with over 65.6 to 76.9 million key-value pairs using a single general-purpose memory system, while providing consistent reads and writes with low latency. MICa is built on top of the RAMCloud system, which provides a variety of mixed read and write operations on a distributed store. The system has been deployed in various cloud environments and has been used by many applications, such as Redis, Memcached, and MySQL. The system has been evaluated on several benchmarks, including Redis, Memcached, and MySQL, and has shown significant performance improvements over traditional memory systems.

increase of clients [31]. RDMA (Remote Direct Memory Access) has gained considerable interests in network-attached in-memory key-value stores. However, traversing the remote tree-based index in ordered stores with RDMA becomes a critical obstacle, causing a significant performance degradation [31].

with lot of programming and memory management overhead. To address this issue, we propose a novel design and implementation of a system that exploits RDMA to run distributed transactions on a modern hardware transactional memory (HTM). The features like atomicity, consistency and isolation (ACI) make it very promising for distributed systems.

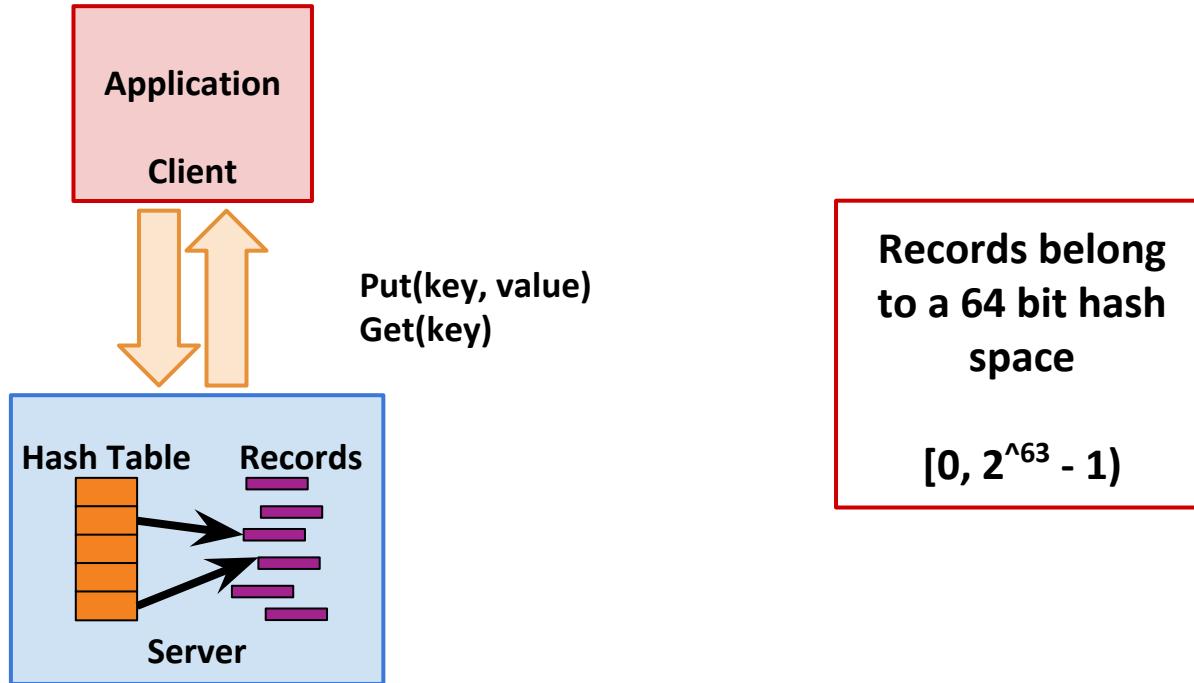
El Castro
Store
Processing using RDMA and HTM
zhe Chen, Rong Chen, Haibo Chen
Scalable Computing and Systems
Systems, Shanghai Jiao Tong University

than in a batched writes + high latencies shall be avoided

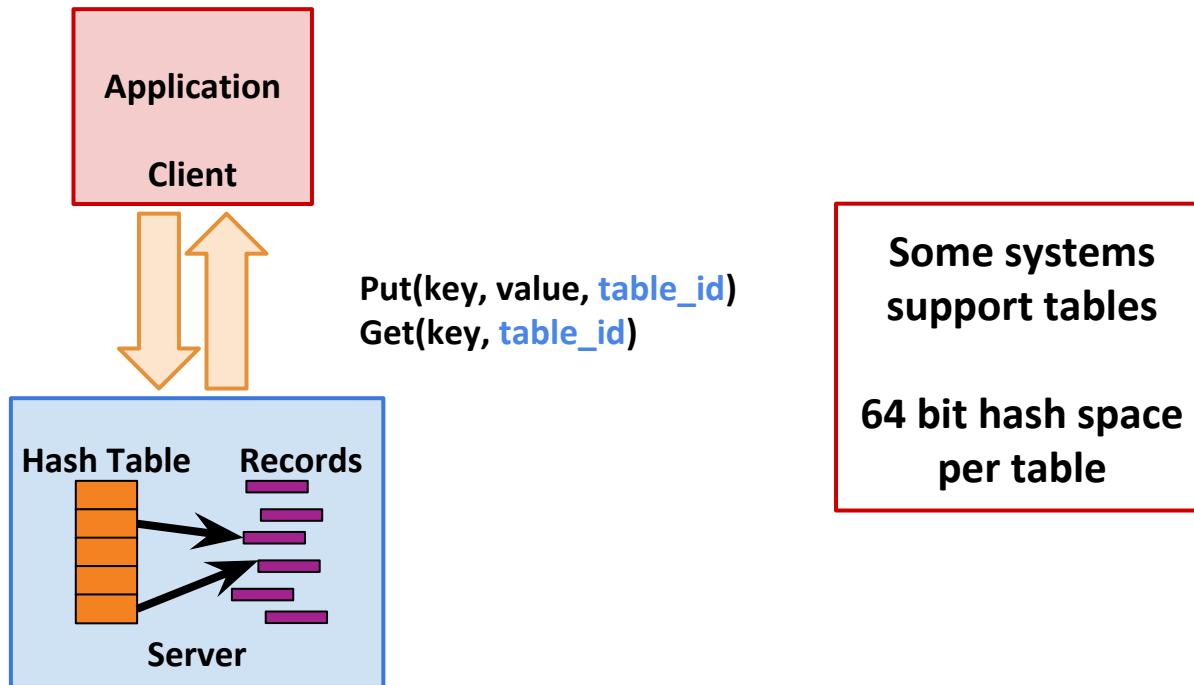
build a transaction processing system that is at least one order of magnitude faster than state-of-the-art systems without using such features. To answer this question, this paper presents the design and implementation of a system that exploits RDMA to run distributed transactions on a modern hardware transactional memory (HTM). The features like atomicity, consistency and isolation (ACI) make it very promising for distributed systems.

with lot of programming and memory management overhead. To address this issue, we propose a novel design and implementation of a system that exploits RDMA to run distributed transactions on a modern hardware transactional memory (HTM). The features like atomicity, consistency and isolation (ACI) make it very promising for distributed systems.

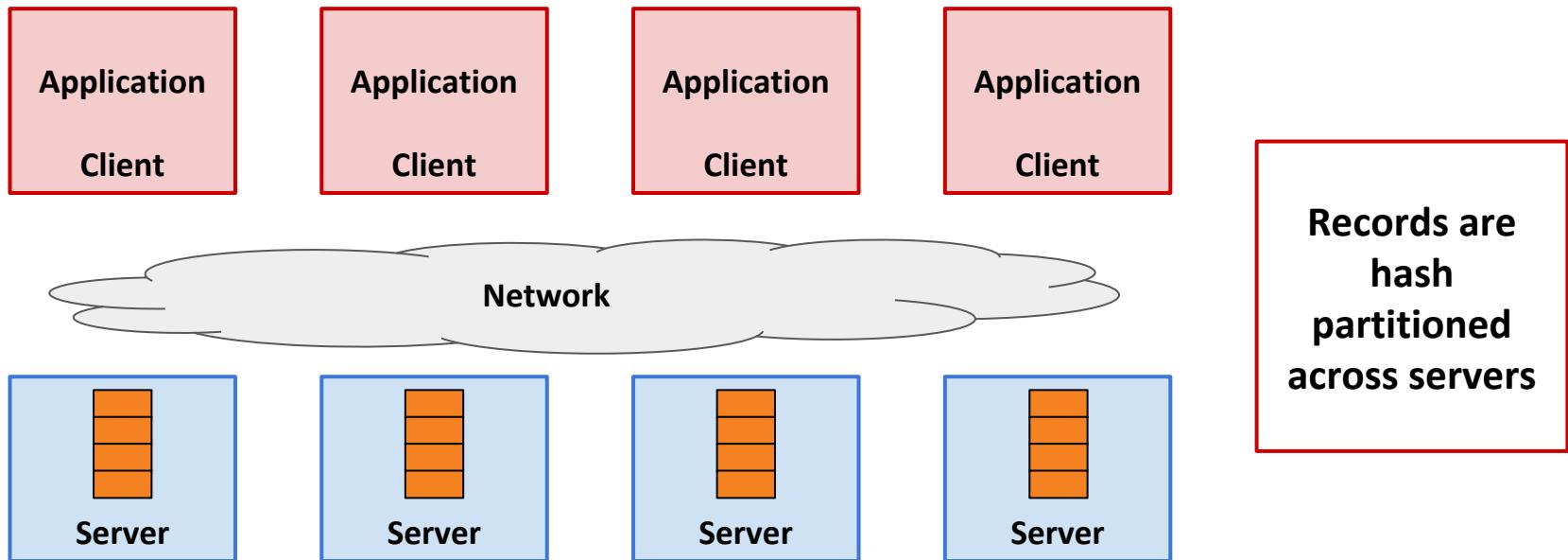
Key-Value Stores: Typical Deployment



Key-Value Stores: Typical Deployment

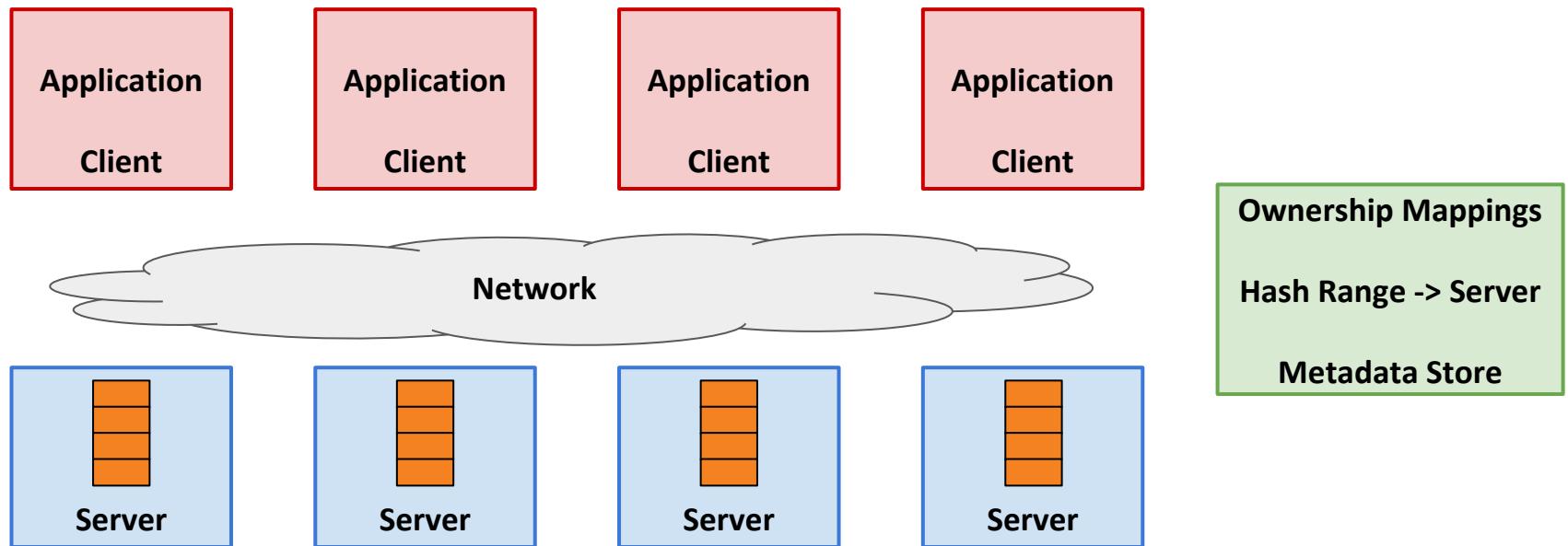


Key-Value Stores: Typical Deployment



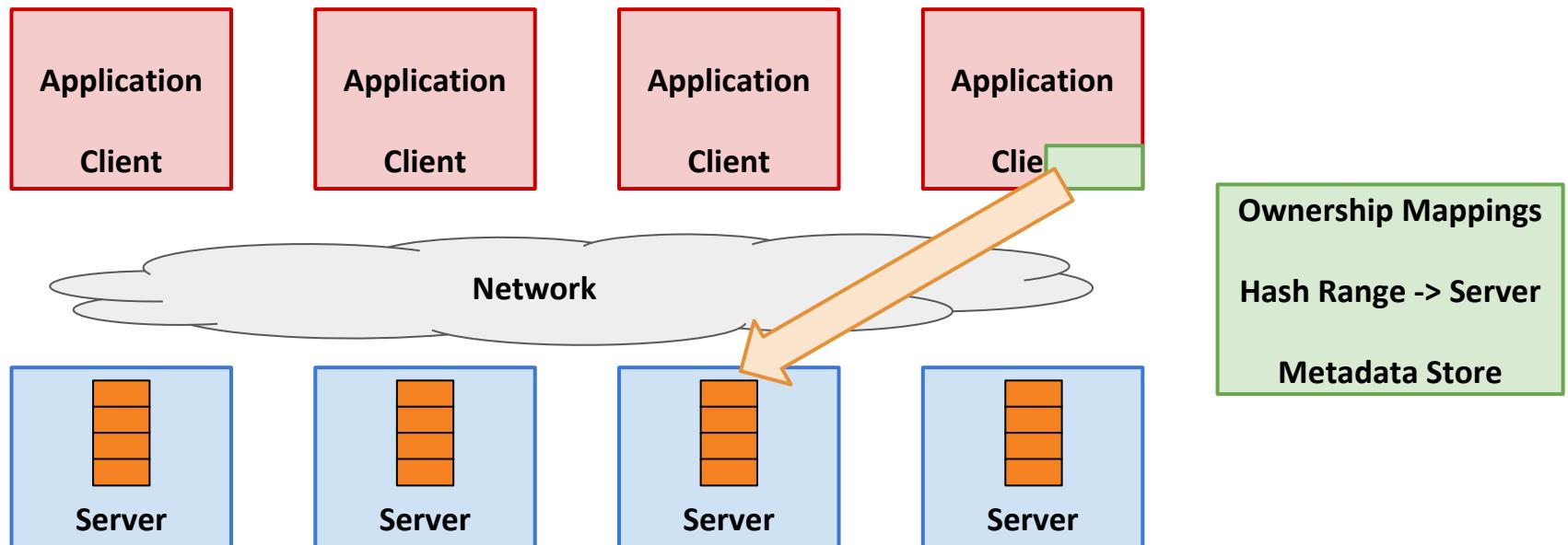
Disaggregate Servers from Clients over Network, Scale-out each Independently

Key-Value Stores: Typical Deployment



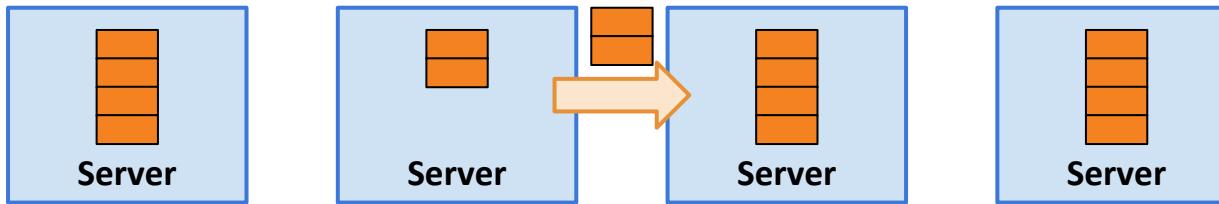
Servers **own** partitions: order requests, provide durability and crash recovery

Key-Value Stores: Typical Deployment



Clients cache mappings, route requests to correct owner

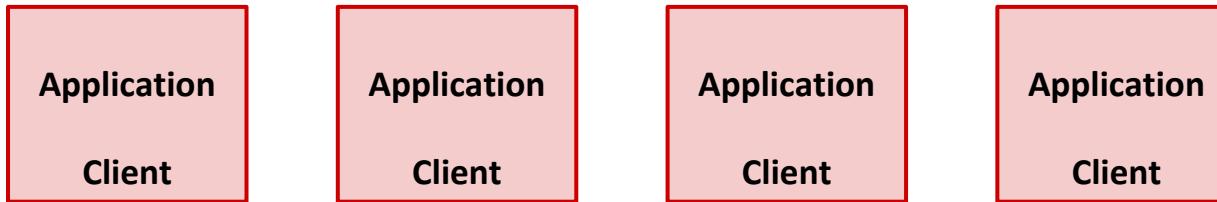
Key-Value Stores: Typical Deployment



Servers often need to horizontally re-distribute load and data (Slicer, OSDI'16)

- Workloads change; access patterns, load imbalances etc
- “Planned failures”; server maintenance, software updates etc

Key-Value Stores: Typical Deployment



`Put(key, value)` , `Get(key)`, `assoc_range(key)`, `assoc_get(key)`

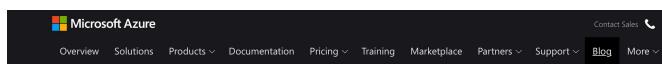
Some applications need rich data model/vertical interface (Facebook Tao, ATC'13)

- Get friend list (`assoc_get()`)
- Get friends of friends (`assoc_range()`)

New Wave of Key-Value Stores

Use Kernel-Bypass to leverage High Speed Networks

- Hardware accelerated transport. Ex: Mellanox Infiniband, Project Catapult, SmartNICs etc.
- Transport integrated into application. Ex: DPDK, RDMA etc.
- MICA(NSDI'14) , FaRM (NSDI'14, SOSP'15), KVDirect (SOSP'17) etc



[Blog](#) / Announcements

Availability of Linux RDMA on Microsoft Azure

Posted on July 9, 2015

Tejas Karmarkar, Principal Program Manager, Azure Compute

We are excited to announce availability of Linux RDMA on Microsoft Azure. With this release, we mark a new milestone in our cloud journey and in our vision to make HPC and Big Compute more accessible and cost-effective for a broader set of users. Linux RDMA makes high speed low latency networking accessible to the engineering and scientific community across the globe, helping them solve complex problems with the applications they use today. Remote Direct Memory Access, or RDMA, is a technology that provides a low-latency network connection between processing running on two servers, or virtual machines in Azure. This technology is essential for engineering simulations and other compute applications that are too large to fit in the memory of a single machine. The A8 and A9 VM sizes in Azure use the InfiniBand network to provide RDMA virtualized through Hyper-V with near "bare metal" performance of less than 3 microsecond latency and greater than 3.5 Gbps bandwidth. The current release of Azure Linux RDMA supports SUSE Linux Enterprise Server 12 (SLES12). We will continue to work with other Linux distributions and will have more to say about other supported distributions in near future. A SLES 12 image with completely integrated RDMA drivers



CONNECTX INFINIBAND ADAPTERS
Enhancing the top Supercomputers and Clouds with HDR 200Gb/s Speed

Overview

Leveraging faster speeds and innovative In-Network Computing, Mellanox InfiniBand smart adapters achieve extreme performance and scale, lowering cost per operation and increasing ROI for high performance computing, machine learning, advanced storage, clustered databases, low-latency embedded I/O applications, and more.



Low-Latency (10s of microseconds per request), High throughput (Million reqs/s)

New Wave of (Impractical) Key-Value Stores

Stripped down design to maximize normal case performance:

- Reconfiguration? Rich Data models? Multi-tenancy?
- Supporting the above can cut into normal-case performance. Ex: Record ownership checks

Fast key-value stores: An idea whose time has come and gone

Atul Adya, Robert Grandl, Daniel Myers
Google

Henry Qin
Stanford University

Abstract

Remote, in-memory key-value (RINK) stores such as Memcached [6] and Redis [7] are widely used in industry and are an active area of academic research. Coupled with stateless application servers to execute business logic and a database-like system to provide persistent storage, they form a core component of nonular data center service architectures. We

First, they may provide a cache over a storage system to enable faster retrieval of persistent state. Second, they may store short-lived data, such as per-session state, that does not warrant persistence [25].

These stores enable services to be deployed using *stateless application servers* [16], which maintain only per-request state; all other state resides in persistent storage or in a

HotOS 2019

My thesis: Mechanisms for the above without compromising on performance

Thesis Contributions

Fast Data Migration [[Rocksteady, SOSP'17](#)]:

- Enables quick reconfiguration in response to load imbalances, changes in access patterns etc

Low-cost Coordination [[Shadowfax, VLDB'21 \(Revision\)](#)]:

- Enables high throughput and elasticity on cloud VMs without stalls and cross-core coordination

Extensibility and Multi-tenancy [[Splinter, OSDI'18](#)]:

- Enables diverse data models by allowing tenants to push functions to the store at runtime

Thesis Statement

Low-latency stores adopt stripped down designs that optimize for the normal case, trading off features that would make them more practical.

My thesis shows that this trade off is unnecessary.

Carefully leveraging and extending abstractions for scheduling, data sharing, lock-freedom, and isolation will yield feature-rich systems that retain their performance.

Fast Data Migration

Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. Rocksteady: Fast Migration for Low-latency In-memory Storage. In *Proceedings of the Twenty-Sixth ACM Symposium on Operating Systems Principles, SOSP'17*

Chinmay Kulkarni, Aniraj Kesavan, Robert Ricci, and Ryan Stutsman. Beyond Simple Request Processing with RAMCloud. *IEEE Data Engineering Bulletin, IEEE DEB 40(1) (March 2017)*

Rocksteady Introduction

Distributed low-latency in-memory key-value stores are emerging

- Predictable response times ~**10 µs median, ~60 µs 99.9th-tile**

Problem: Must migrate data between servers

- Quickly respond to hot spots, skew shifts, load spikes
- Minimize performance impact of migration

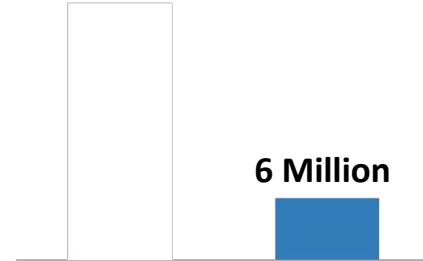
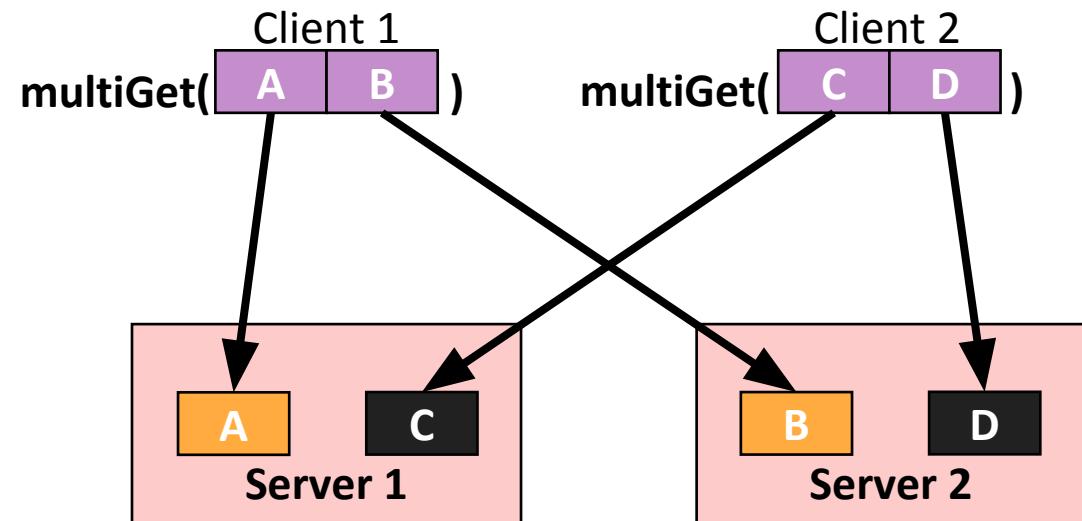
Solution: Fast data migration with low impact

- Early ownership transfer of data, leverage workload skew
- Low priority, parallel and adaptive migration

Result: Migration protocol for RAMCloud in-memory key-value store

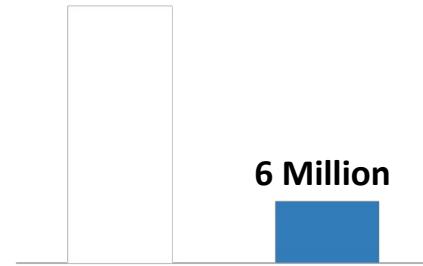
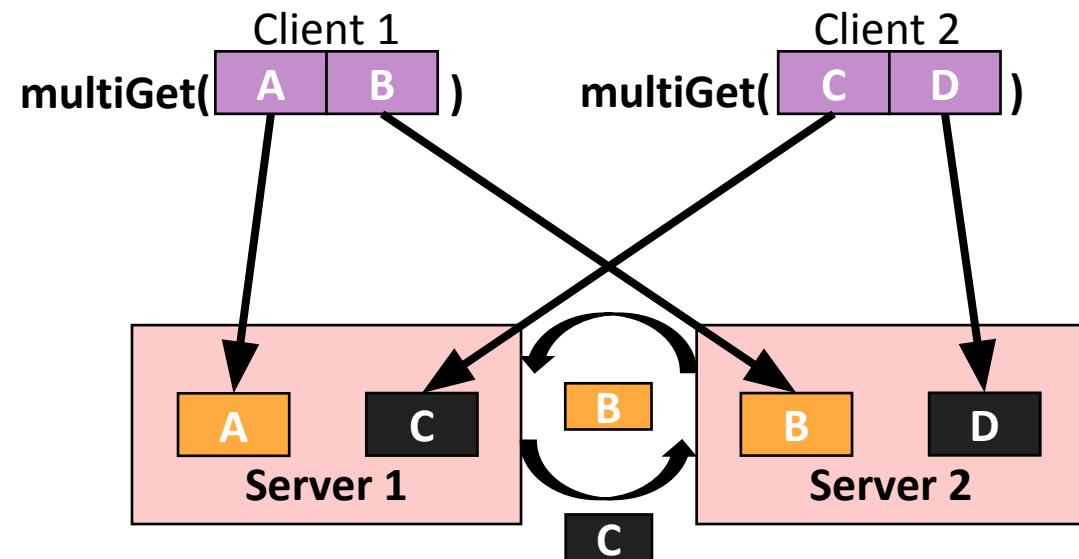
- Migrates **28%** faster than the baseline

Why Migrate Data?

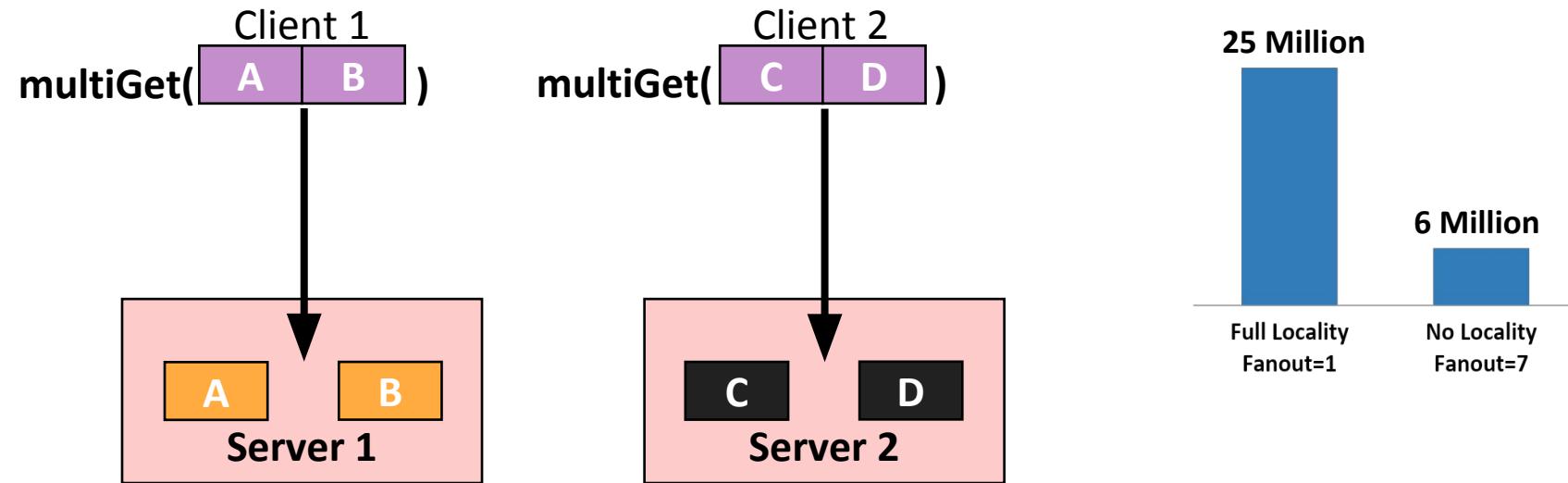


Poor spatial locality → High `multiGet()` fan-out → More RPCs

Migrate To Improve Spatial Locality



Spatial Locality Improves Throughput



Better spatial locality → Fewer RPCs → Higher throughput
Benefits multiGet(), range scans

Performance Goals For Migration

Migrate data fast

- Workloads dynamic → Respond quickly

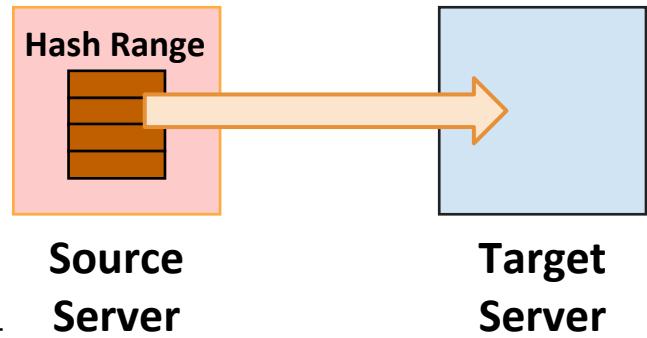
Maintain low access latency

- 10 µsec median latency → System extremely sensitive
- Tail latency matters at scale → Even more sensitive

Rocksteady: Design

Instantaneous ownership transfer

- Immediate load reduction at overloaded source
- Creates “headroom” for migration work



Leverage skew to rapidly migrate hot data

- Target comes up to speed with little data movement

Adaptive parallel, pipelined at source and target

- All cores avoid stalls, but yield to client-facing operations

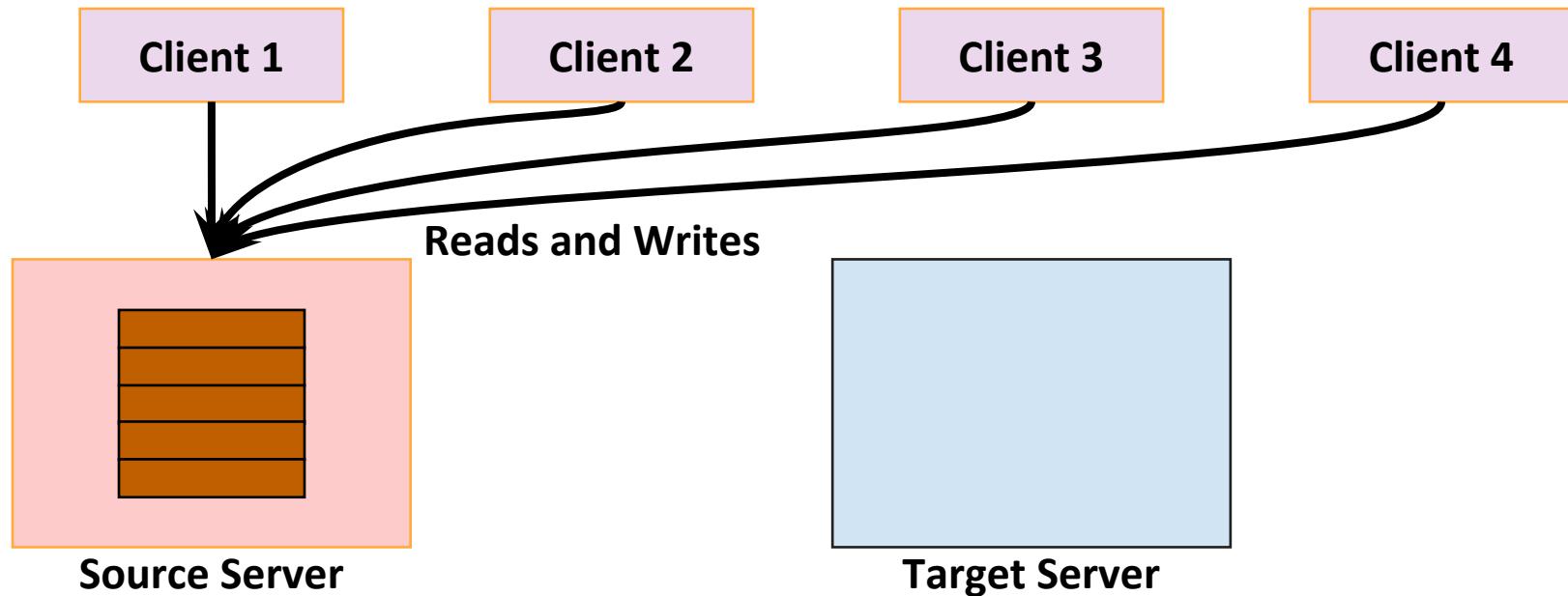
Safely defer replication at target

- Eliminates replication bottleneck and contention

Rocksteady: Early Ownership Transfer

Problem: Loaded source can bottleneck migration

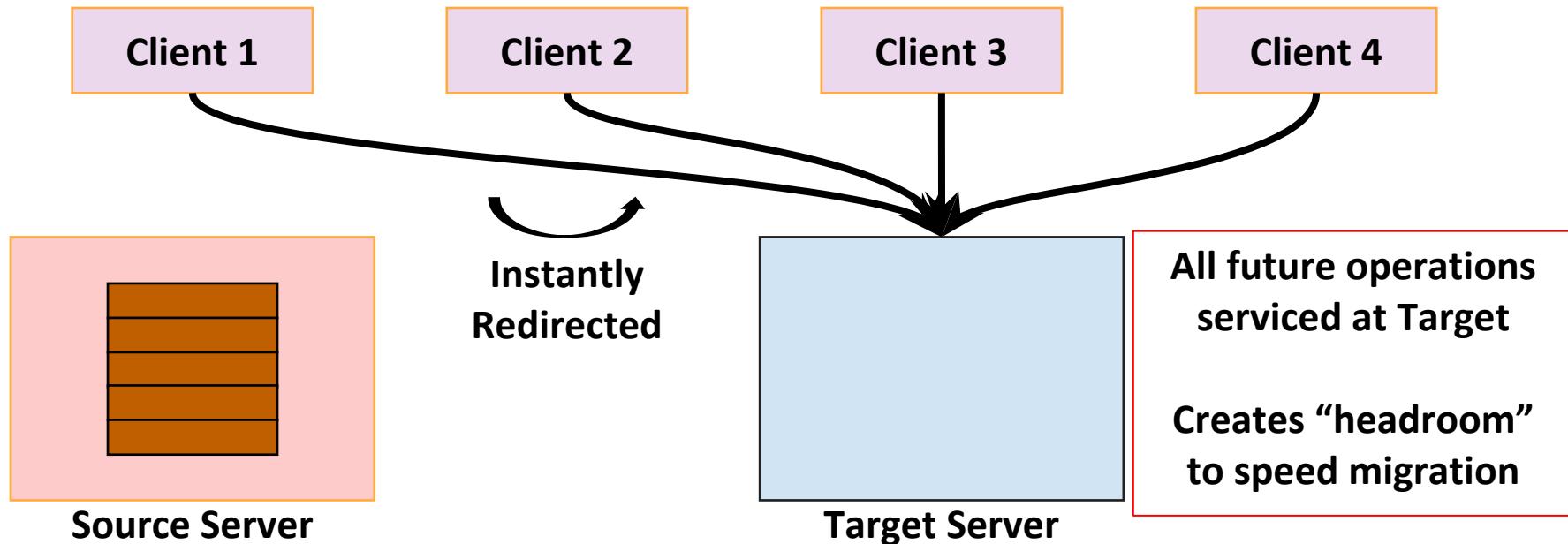
Solution: Instantly shift ownership and all load to target before shifting data



Rocksteady: Early Ownership Transfer

Problem: Loaded source can bottleneck migration

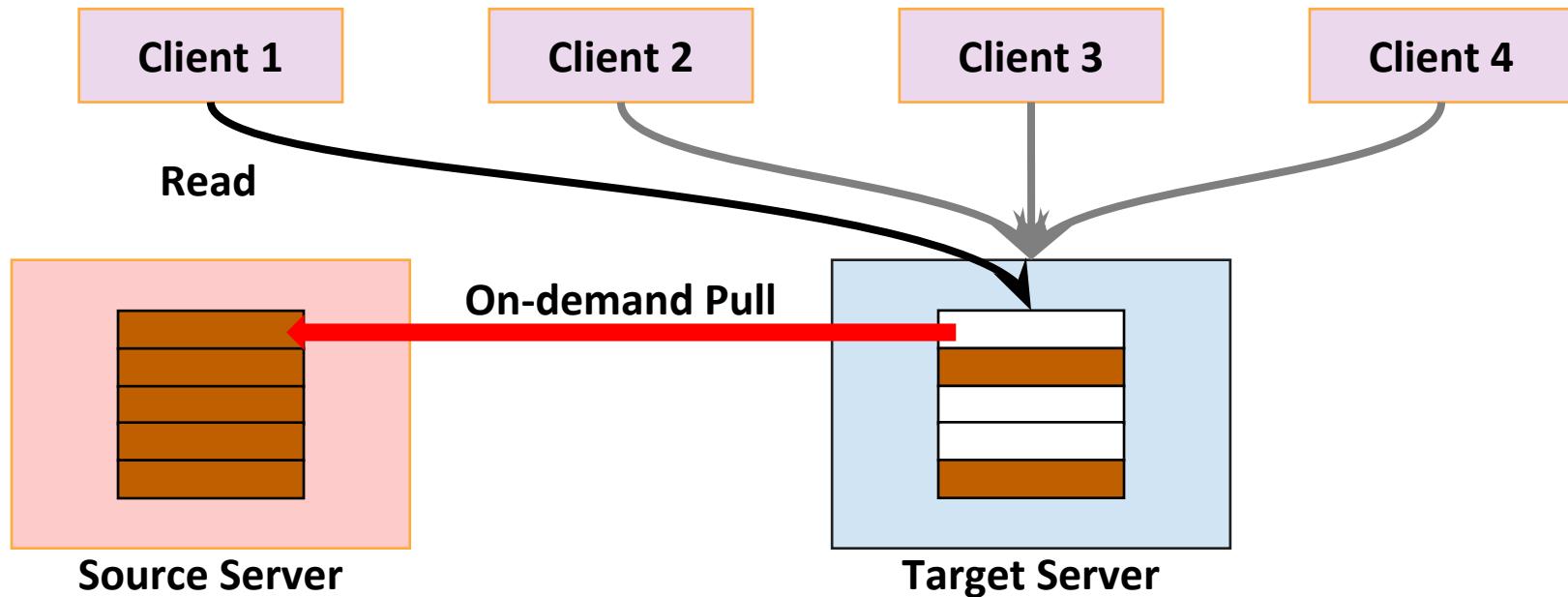
Solution: Instantly shift ownership and all load to target before shifting data



Rocksteady: Leverage Skew

Problem: Data has not arrived at source yet

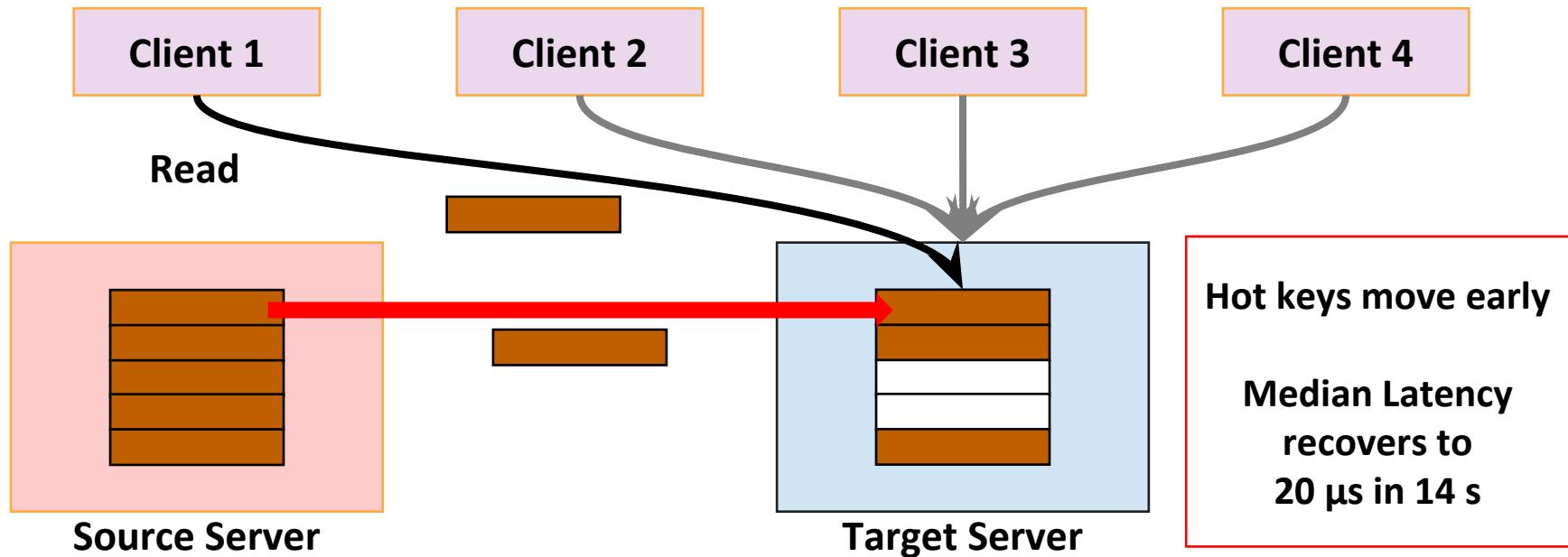
Solution: On demand migration of unavailable data



Rocksteady: Leverage Skew

Problem: Data has not arrived at source yet

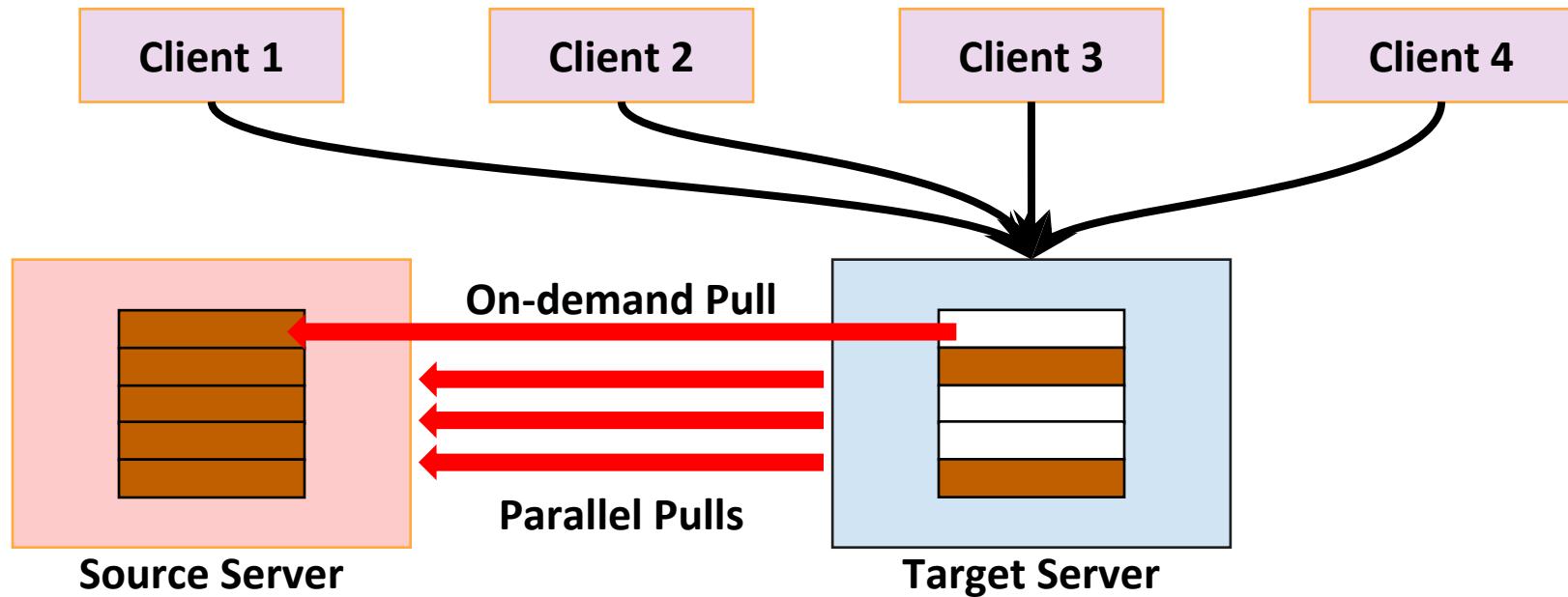
Solution: On demand migration of unavailable data



Rocksteady: Adaptive and Parallel

Problem: Old single-threaded protocol limited to 130 MB/s

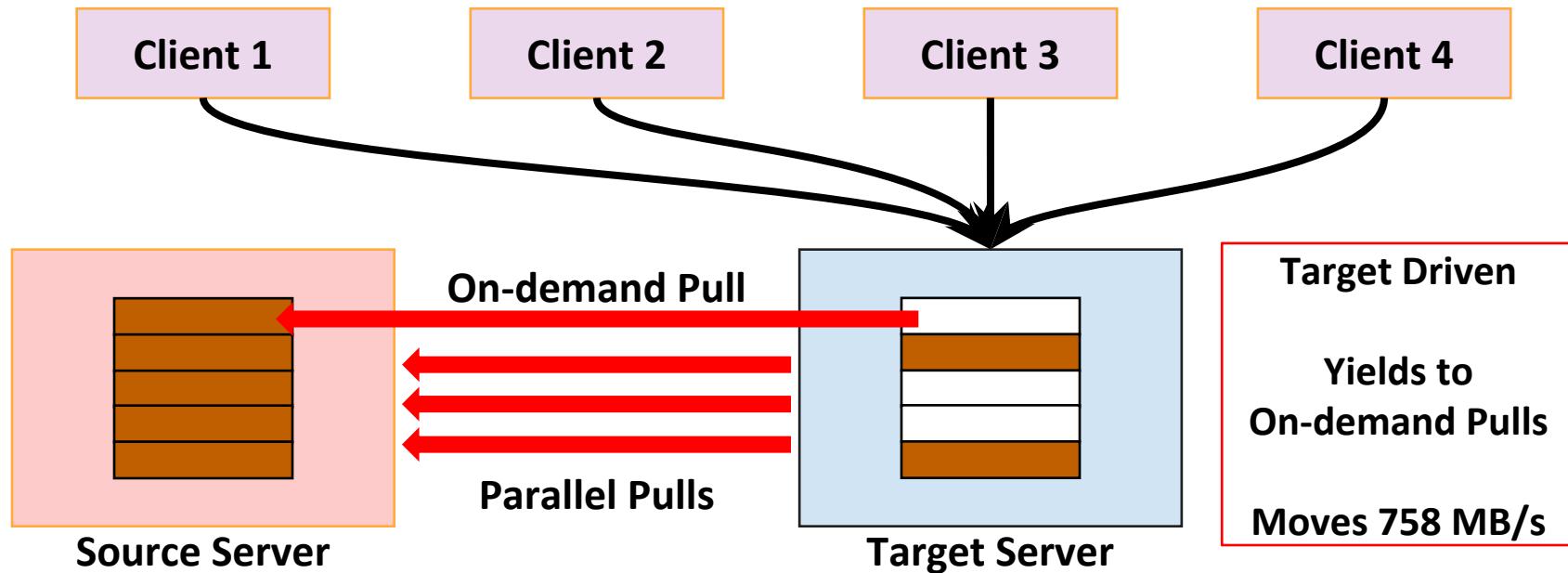
Solution: Pipelined and parallel at source and target



Rocksteady: Adaptive and Parallel

Problem: Old single-threaded protocol limited to 130 MB/s

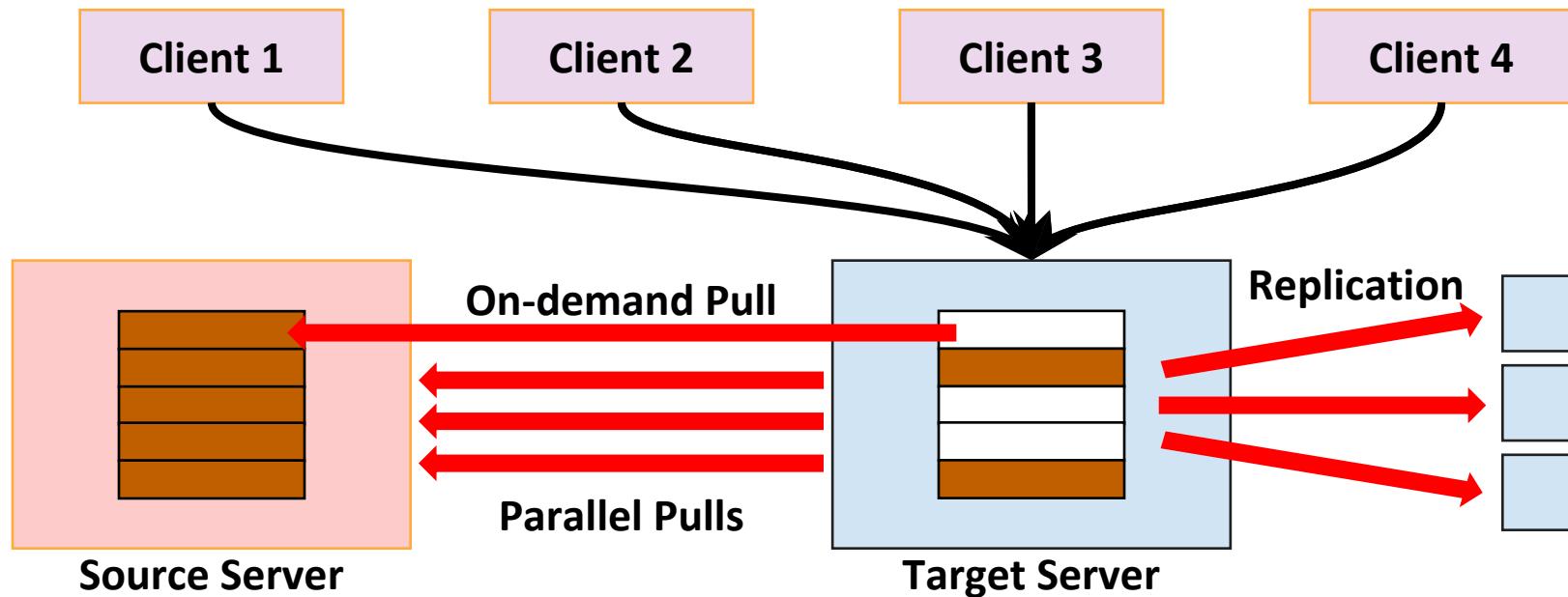
Solution: Pipelined and parallel at source and target



Rocksteady: Eliminate Sync Replication

Problem: Synchronous replication bottleneck at target

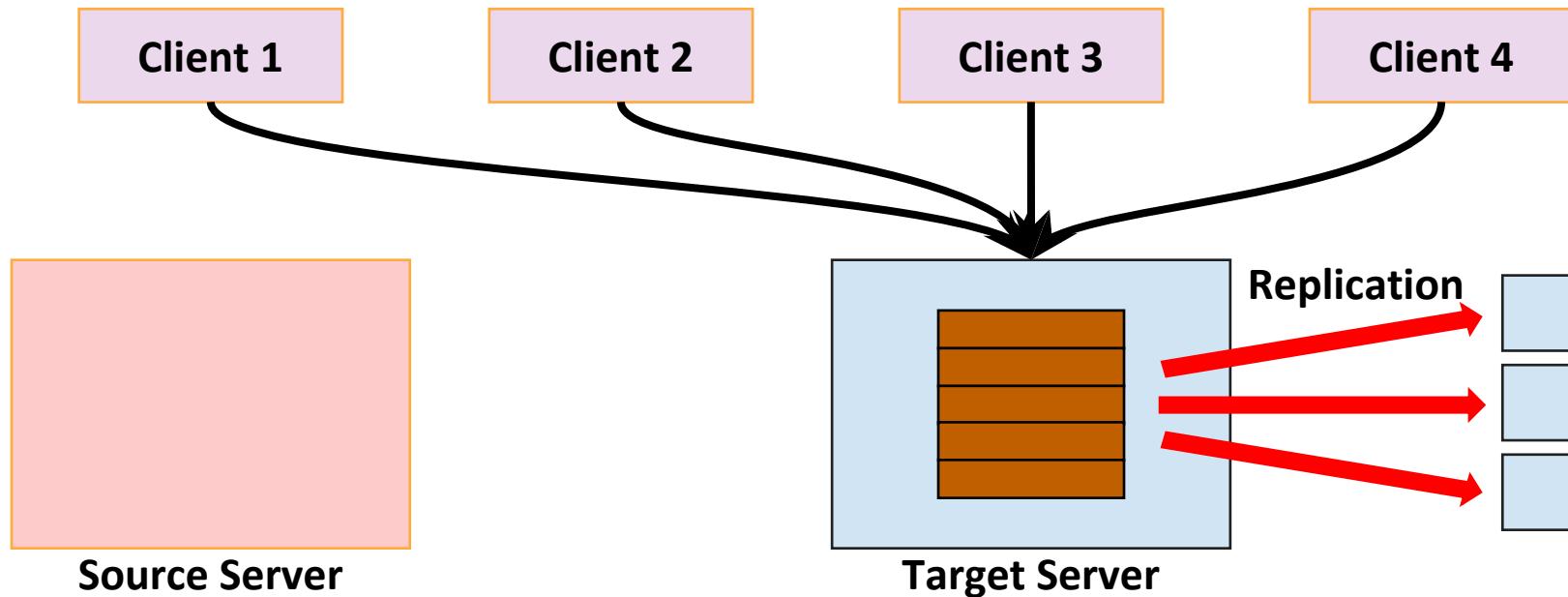
Solution: Safely defer replication until after migration



Rocksteady: Eliminate Sync Replication

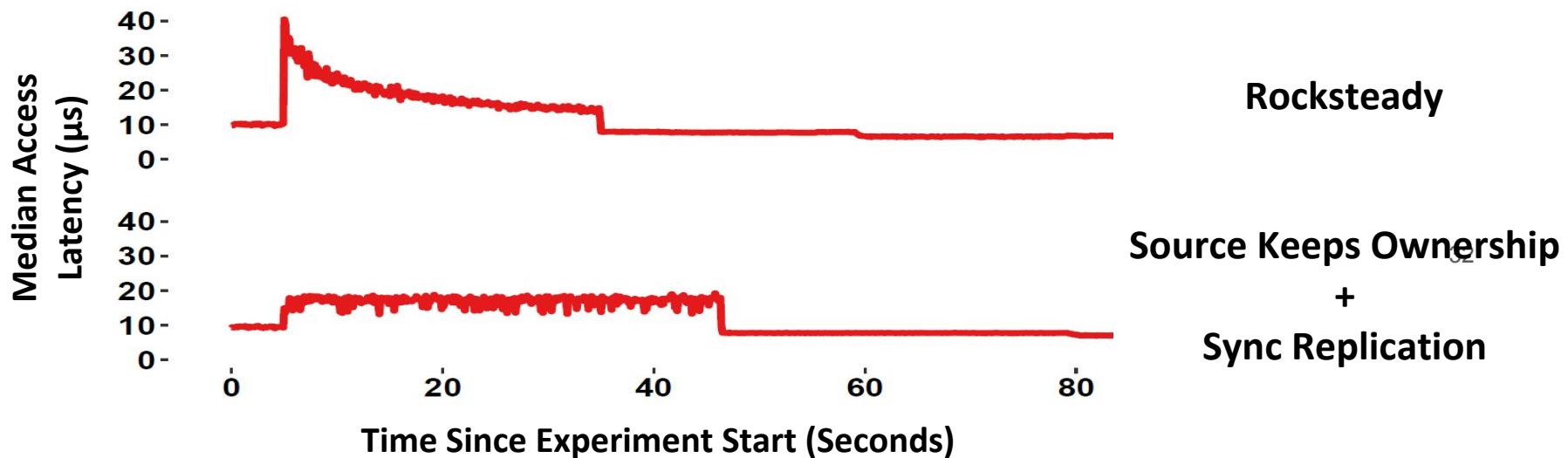
Problem: Synchronous replication bottleneck at target

Solution: Safely defer replication until after migration



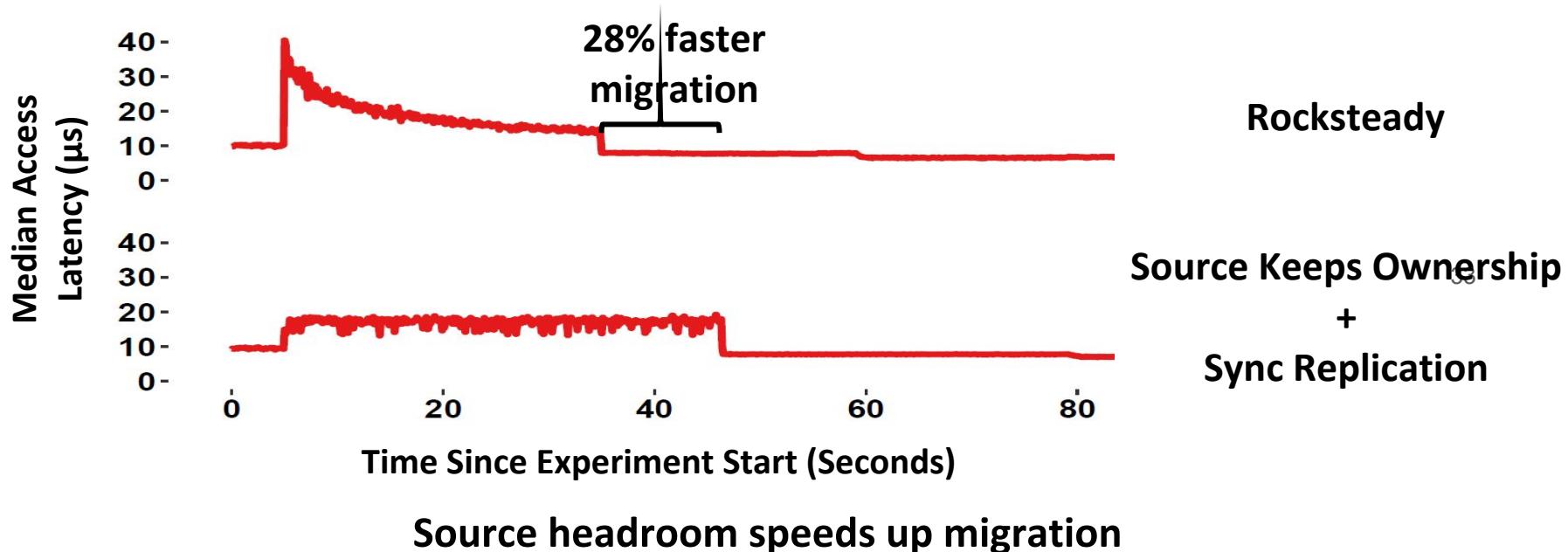
Performance of Rocksteady

YCSB-B, 300 Million objects (30 B key, 100 B value), migrate half



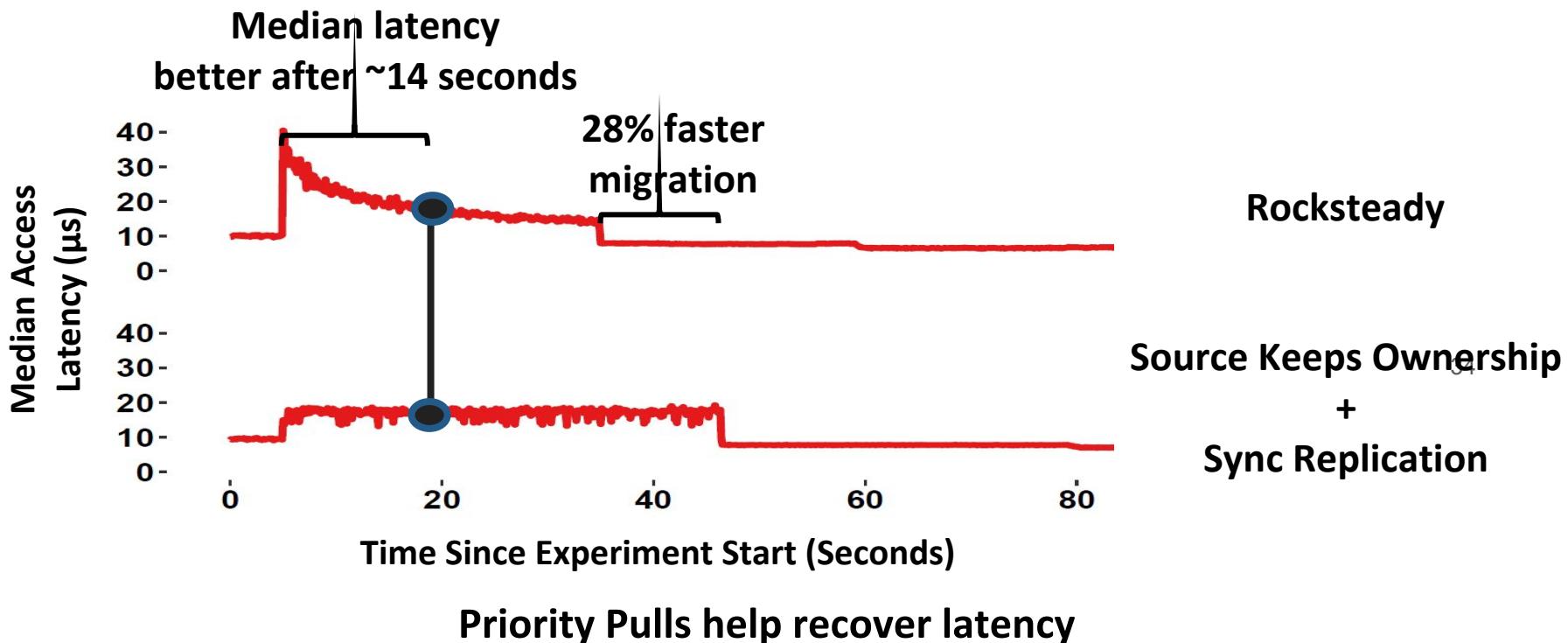
Performance of Rocksteady

YCSB-B, 300 Million objects (30 B key, 100 B value), migrate half



Performance of Rocksteady

YCSB-B, 300 Million objects (30 B key, 100 B value), migrate half



Low-Cost Coordination

Chinmay Kulkarni, Badrish Chandramouli, and Ryan Stutsman. Achieving High Throughput and Elasticity in a Larger-than-Memory Store. Under Revision at VLDB 2021

Shadowfax Introduction

Multi-core optimized single-node key-value stores are emerging

- **Very high throughput ~100 Million events/sec/server.** Ex: FASTER (SIGMOD'18)

Problem: Retain high throughput in the public cloud

- **Avoid dispatch and network bottlenecks** → Saturate servers within hash index
- Support reconfiguration/migration while preserving throughput

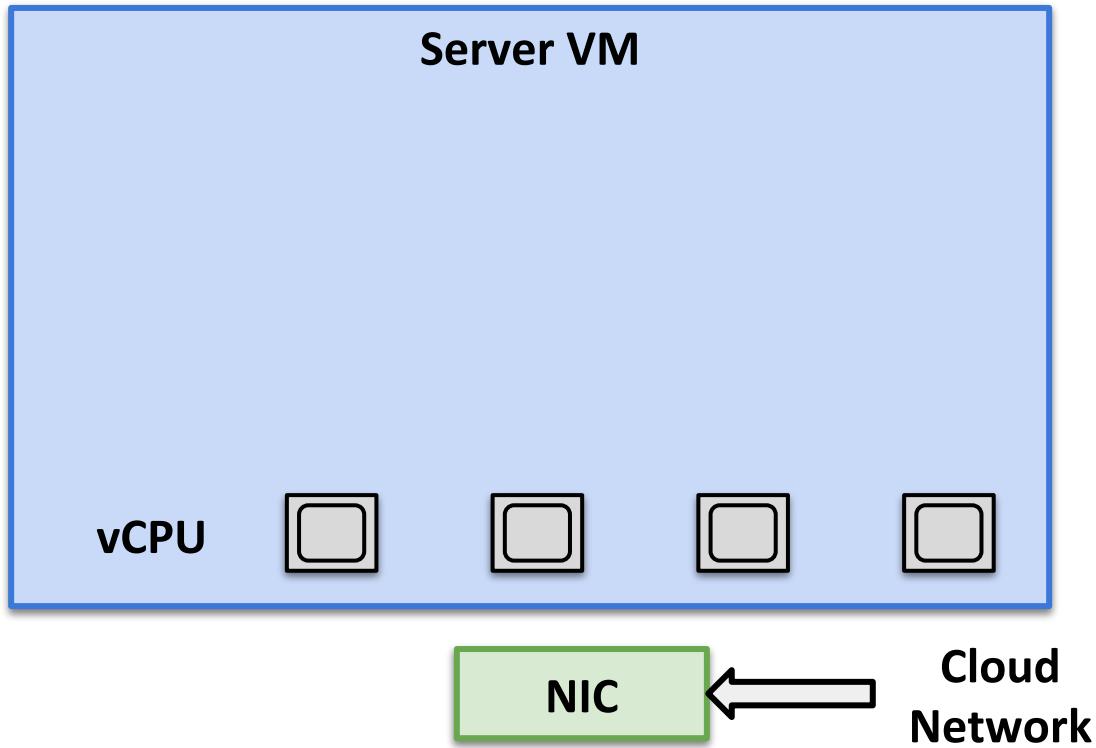
Solution: Partitioned Dispatch, Asynchronous Global Cuts

- Cores share FASTER index, dispatch layer is partitioned → **retain high throughput**
- Cores avoid coordination during migration → **maintain high throughput**

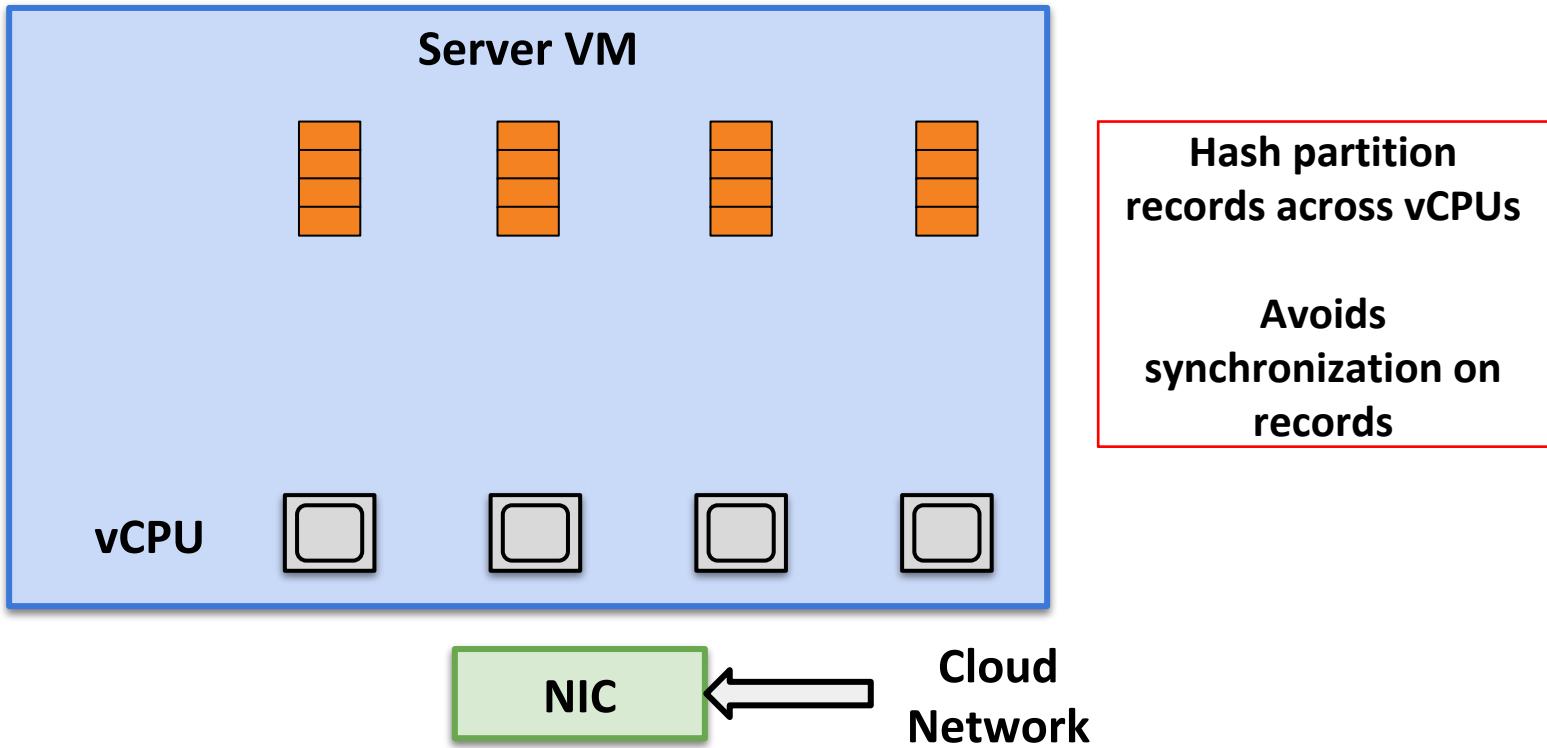
Shadowfax: High Throughput, Elastic, Larger-than-Memory key-value store

- **100 Million events/sec/VM on Azure, 80 Million events/sec/VM** during migration
- Spans DRAM, SSD and Cloud blob storage

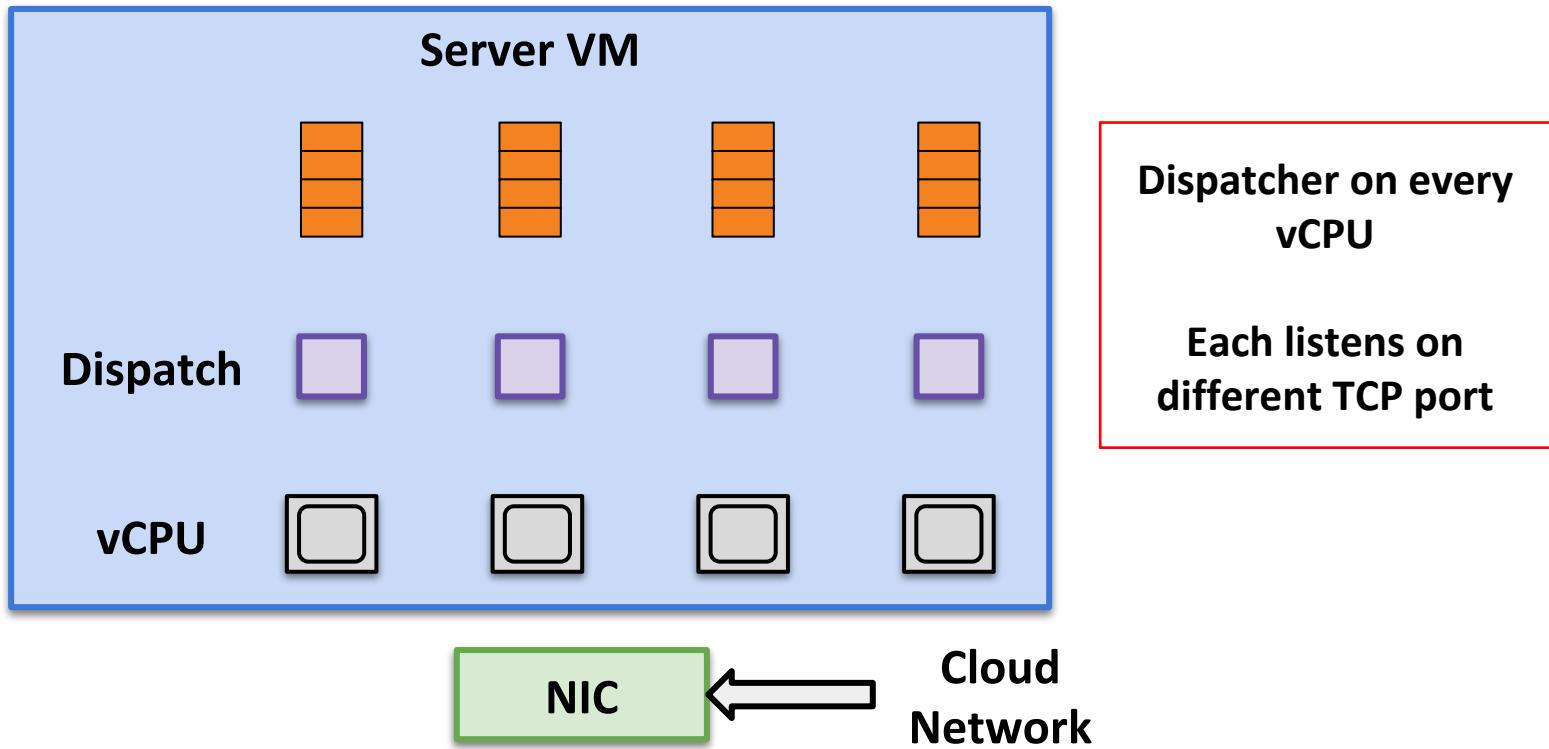
Seastar: State-of-the-art Cloud Key-Value Store



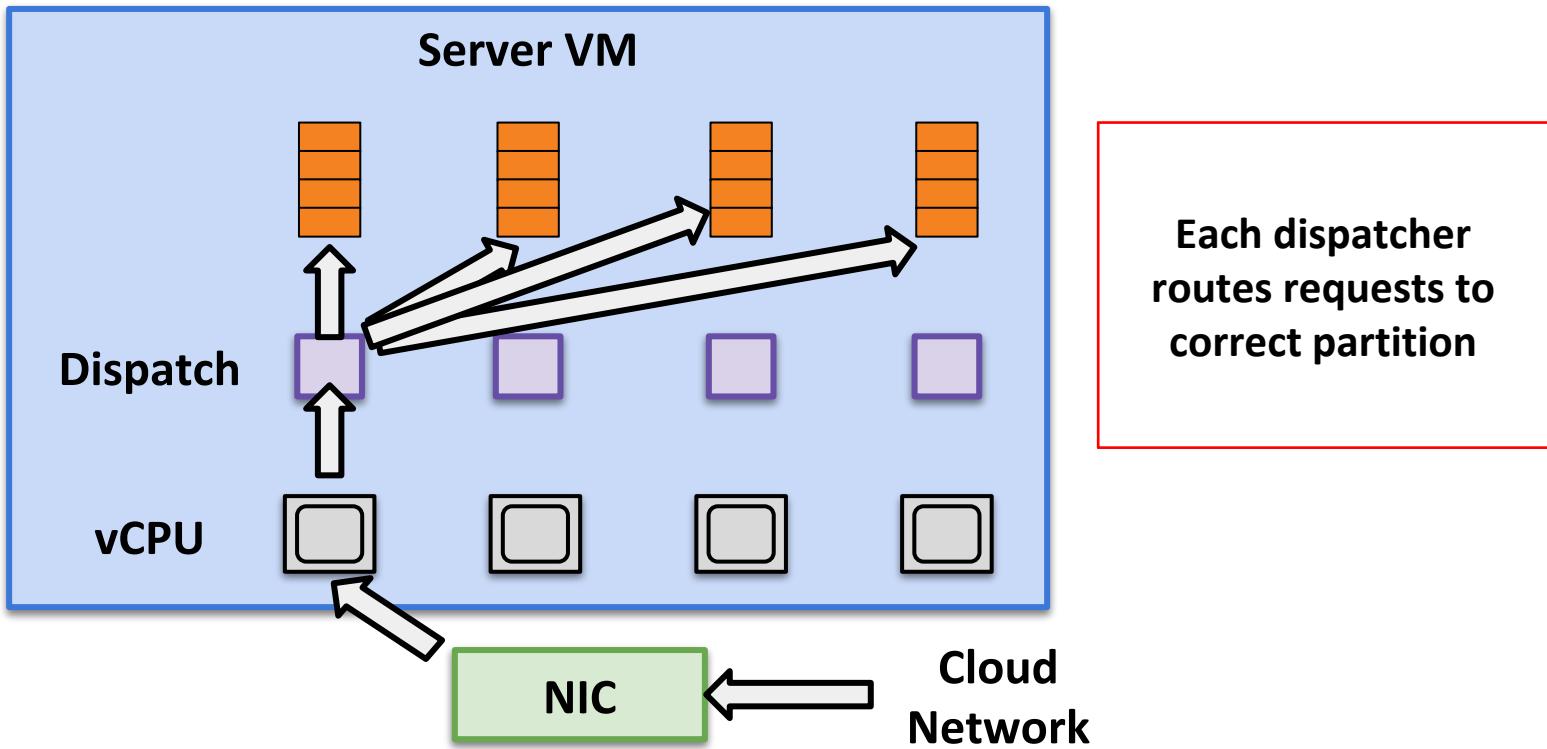
Seastar: State-of-the-art Cloud Key-Value Store



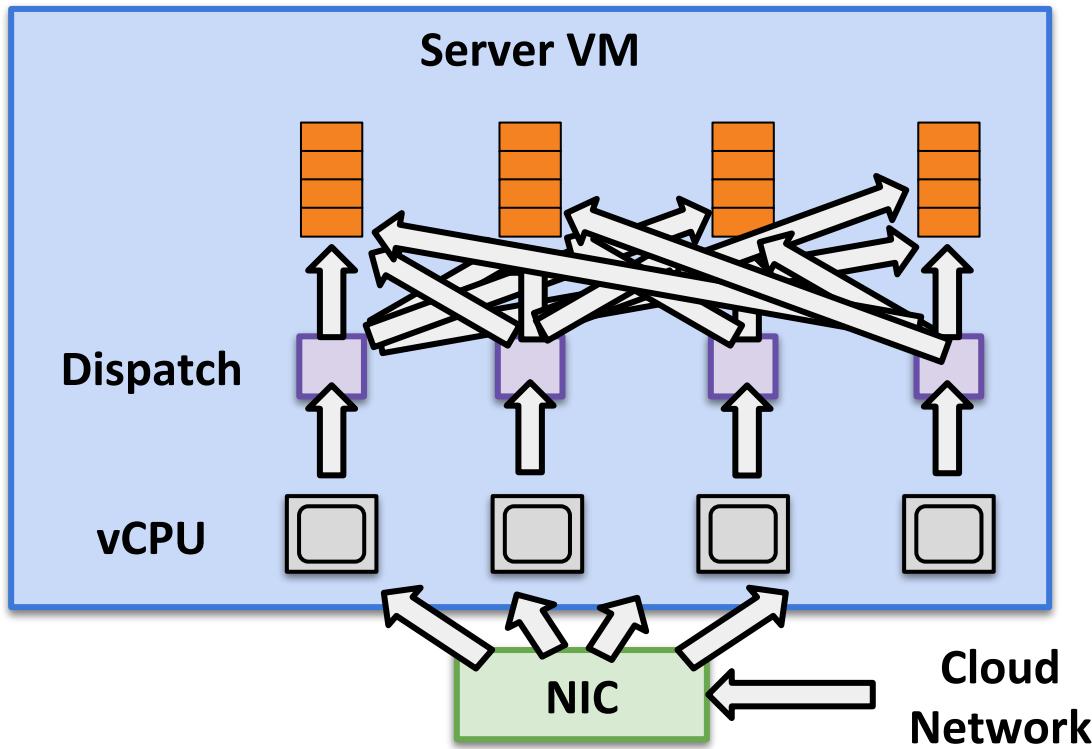
Seastar: State-of-the-art Cloud Key-Value Store



Seastar: State-of-the-art Cloud Key-Value Store

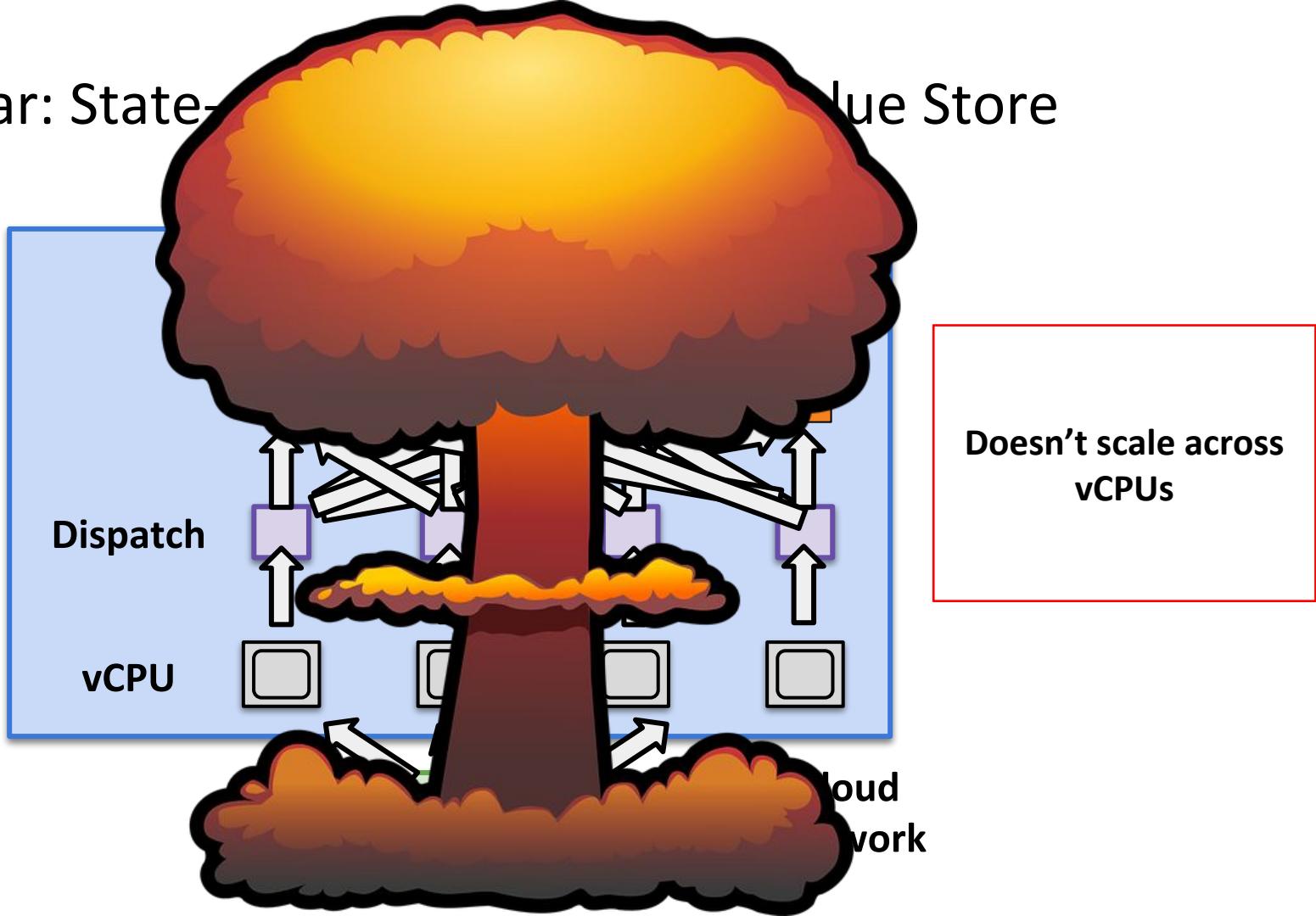


Seastar: State-of-the-art Cloud Key-Value Store

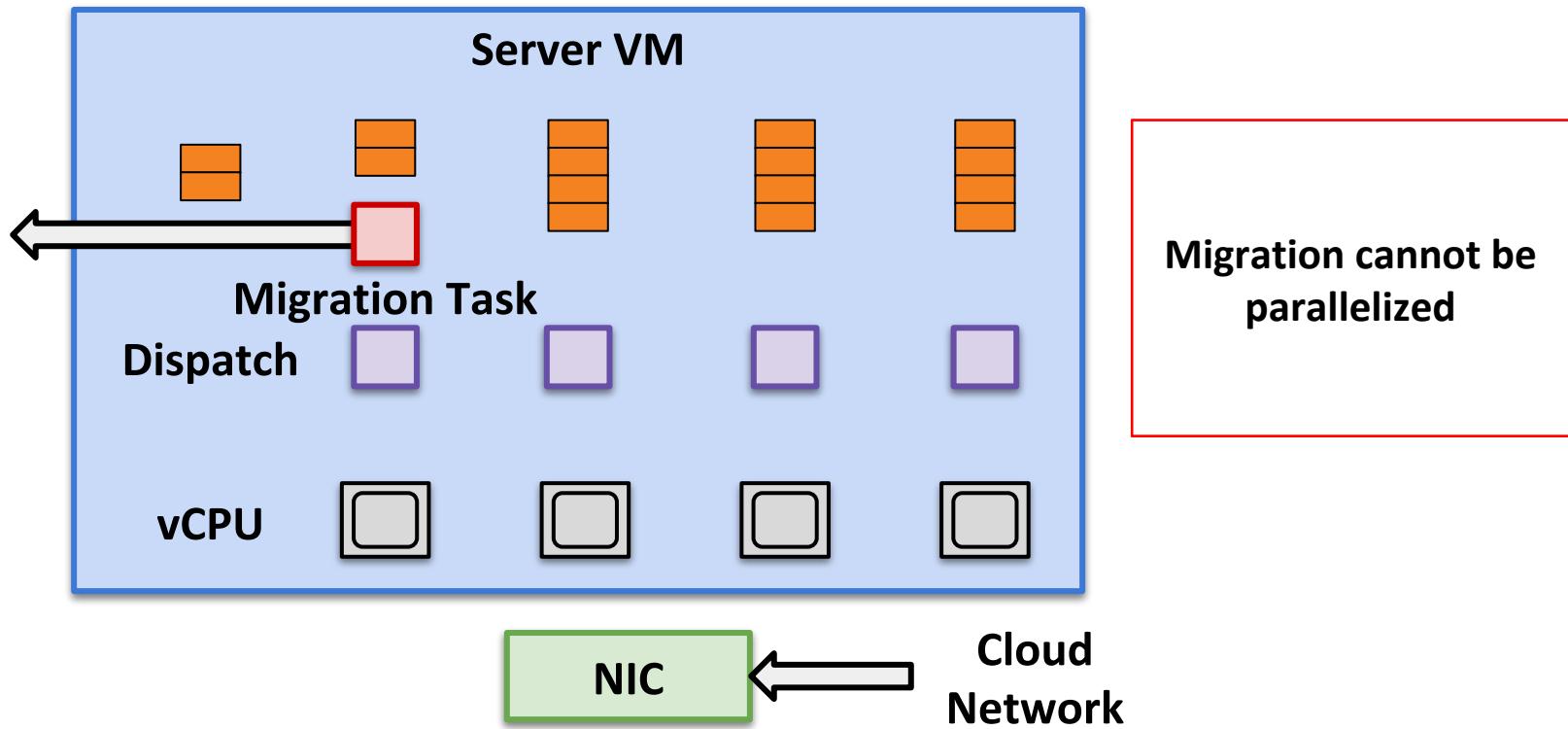


Each dispatcher routes requests to correct partition

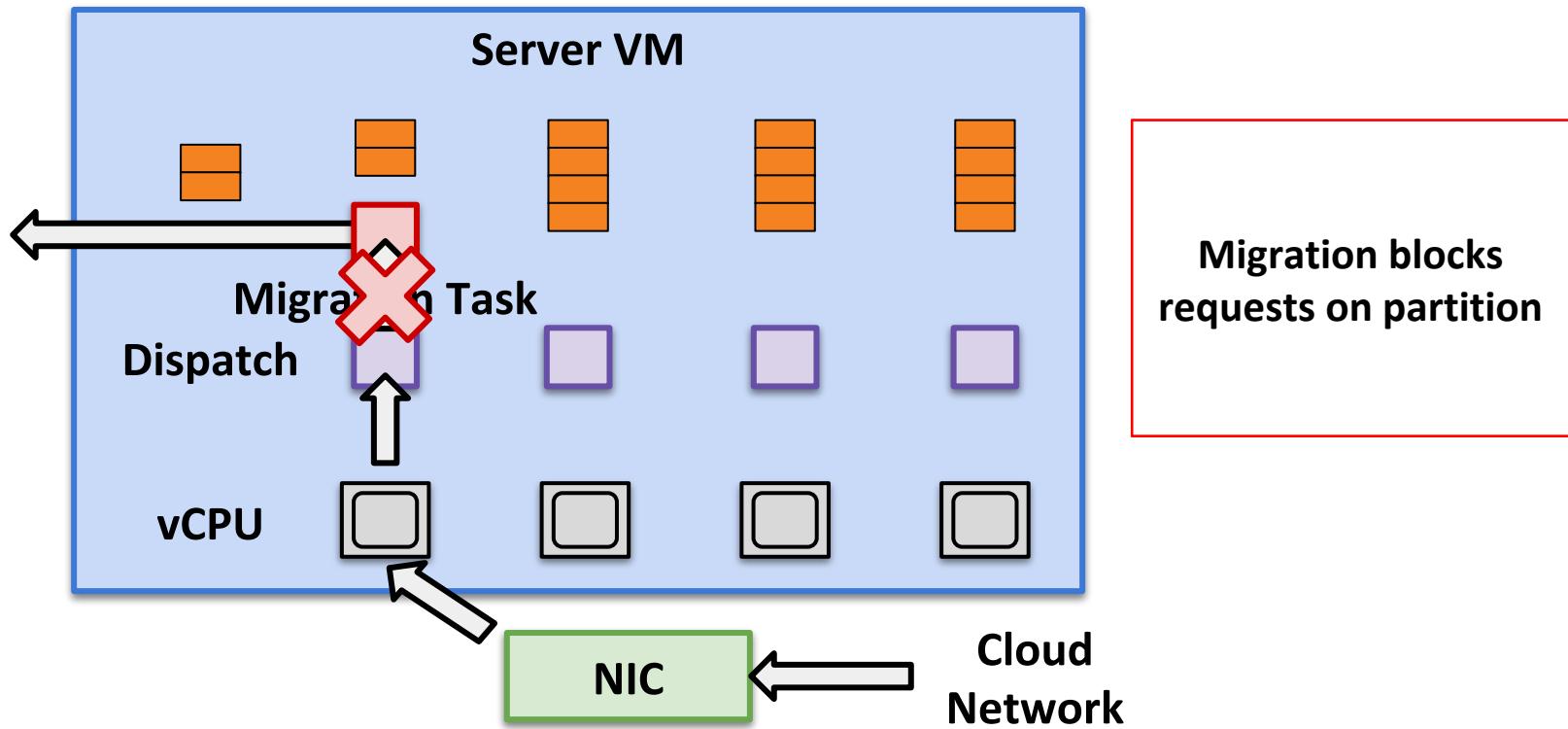
Seastar: State- Value Store



Seastar: State-of-the-art Cloud Key-Value Store



Seastar: State-of-the-art Cloud Key-Value Store



Shadowfax: Design

Partitioned Dispatch Shared Data

- vCPUs share lock-free index → record synchronization handled by cache coherence
- Migration can be parallelized, does not block requests on migrating hash range

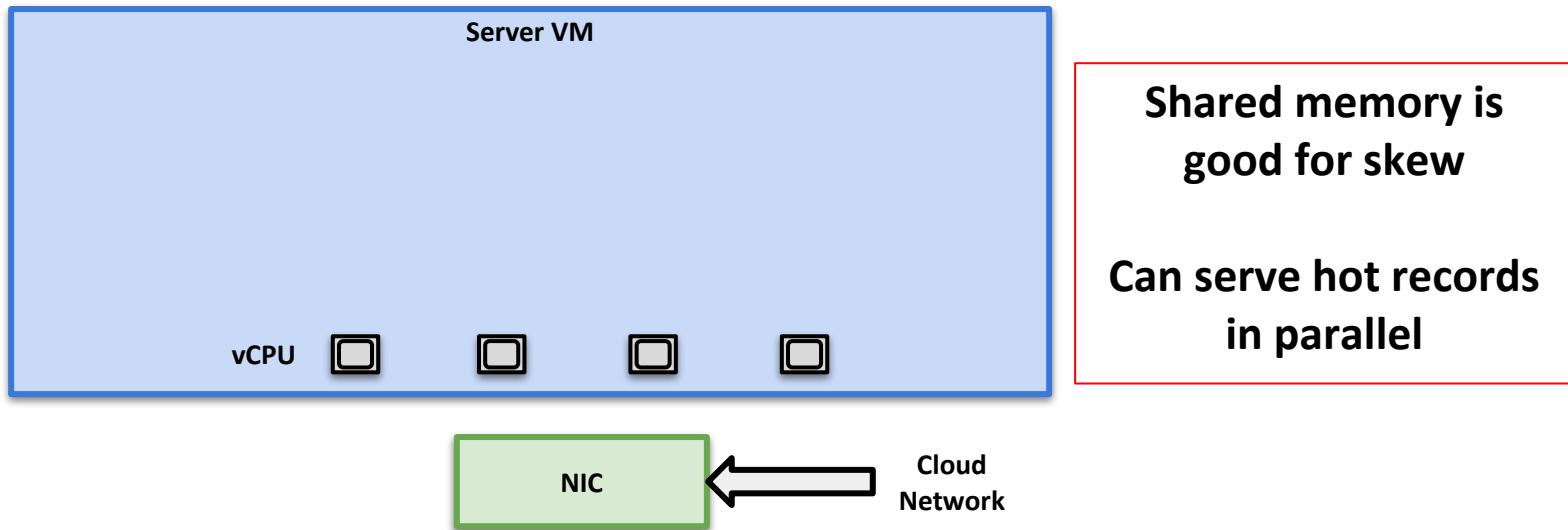
Asynchronous Global Cuts

- Per server view numbers represent hash ranges owned by server
- Cores avoid coordination during migration ownership changes

Shadowfax: Partitioned Dispatch Shared Data

Problem: Avoid cross-core coordination at server

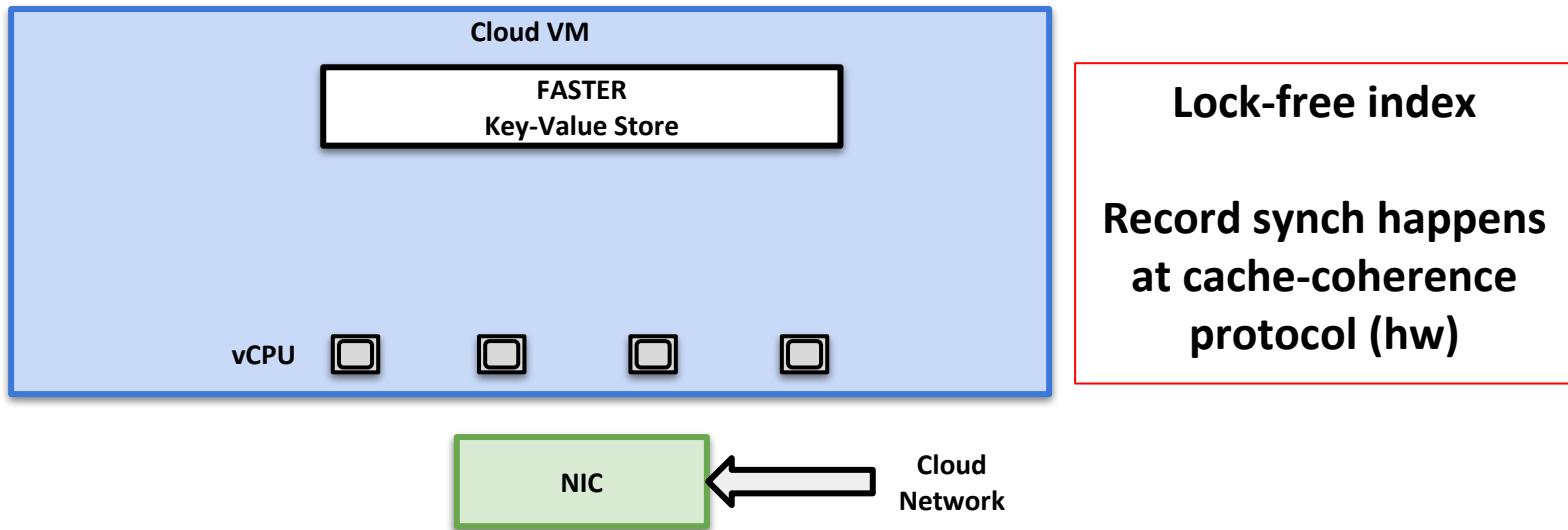
Solution: Offload all coordination to hardware cache-coherence protocol



Shadowfax: Partitioned Sessions Shared Data

Problem: Avoid cross-core coordination at server

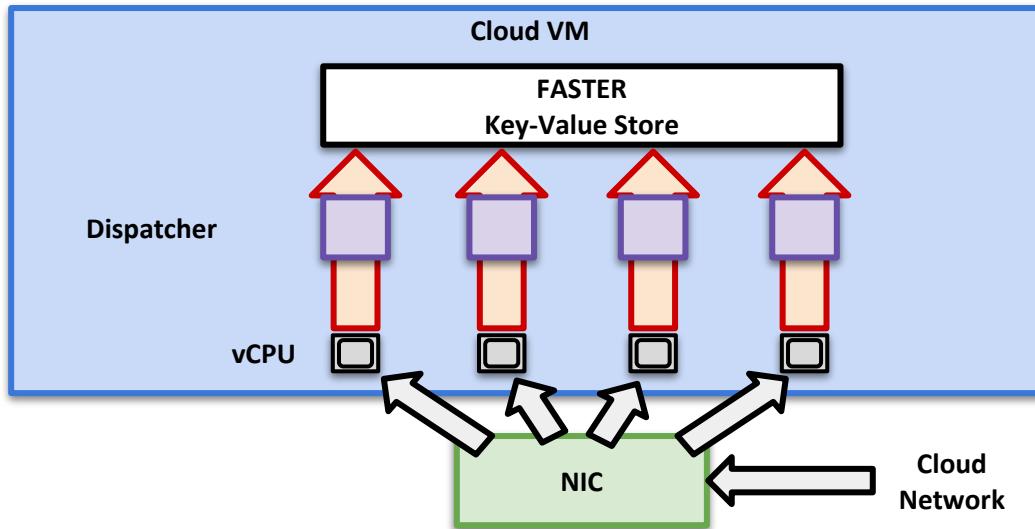
Solution: Offload all coordination to hardware cache-coherence protocol



Shadowfax: Partitioned Sessions Shared Data

Problem: Multi-core request processing requires cross-core coordination

Solution: Offload all coordination to hardware cache-coherence protocol



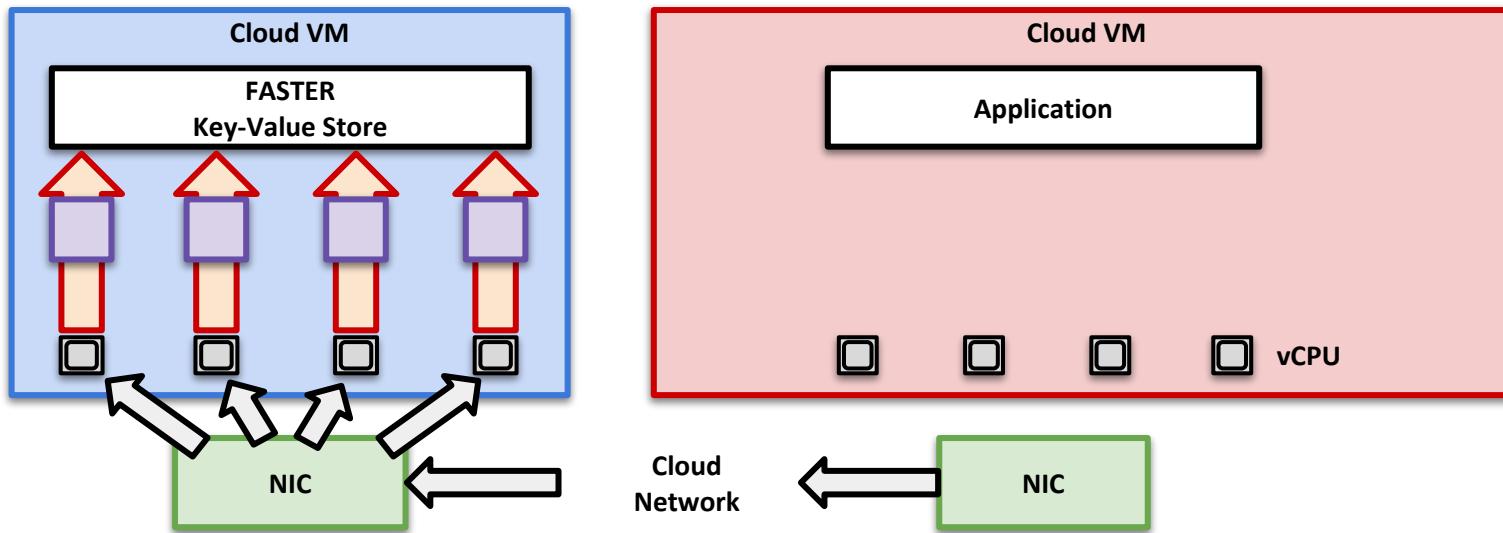
Partition dispatch

Each dispatcher listens
on separate TCP port

Shadowfax: Partitioned Sessions Shared Data

Problem: Keep servers saturated at index, not network or dispatch

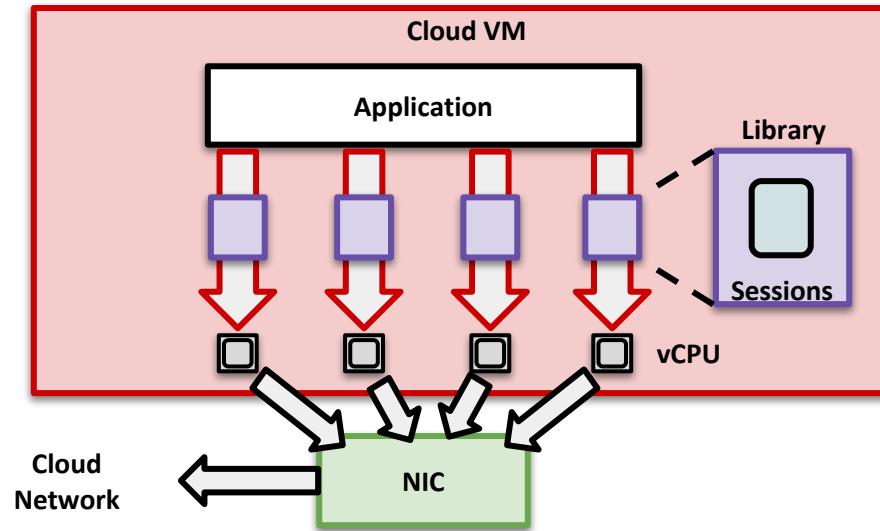
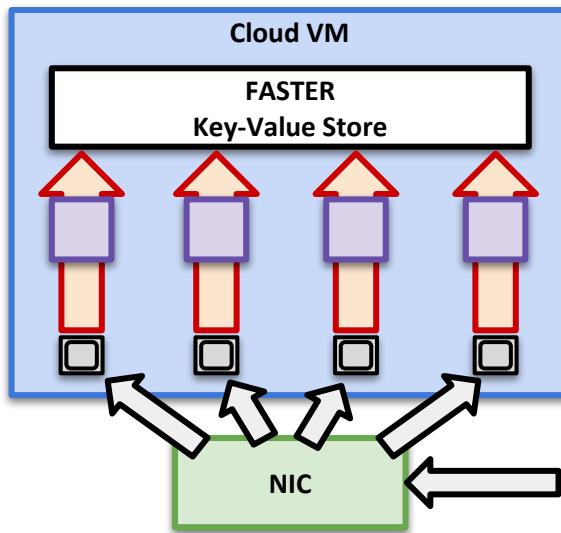
Solution: Asynchronous client library, transparent network acceleration



Shadowfax: Partitioned Sessions Shared Data

Problem: Keep servers saturated at index, not network or dispatch

Solution: Asynchronous client library, transparent network acceleration

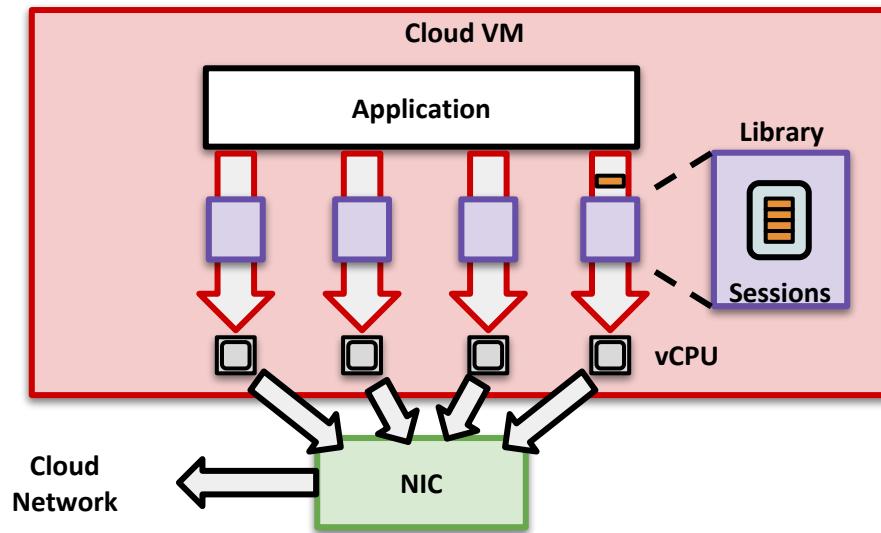
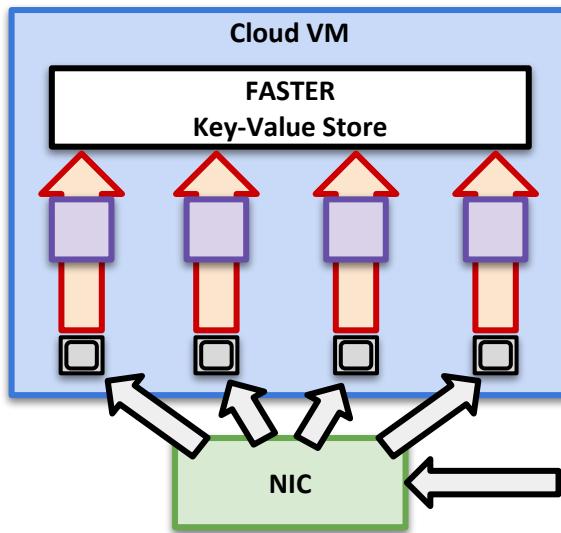


Client cores open “sessions” to server cores

Shadowfax: Partitioned Sessions Shared Data

Problem: Keep servers saturated at index, not network or dispatch

Solution: Asynchronous client library, transparent network acceleration



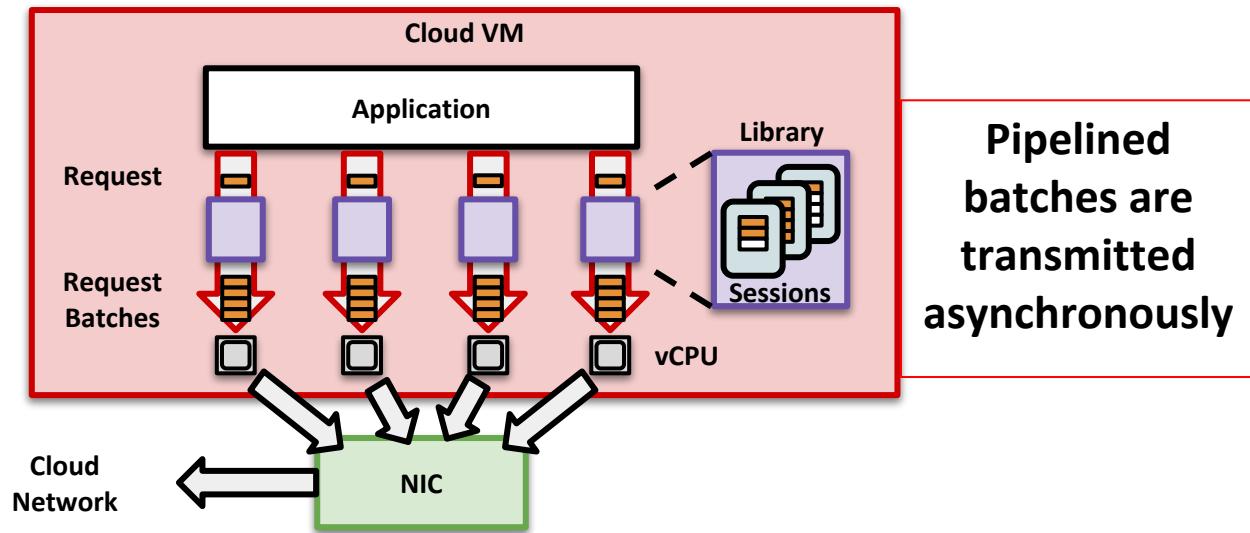
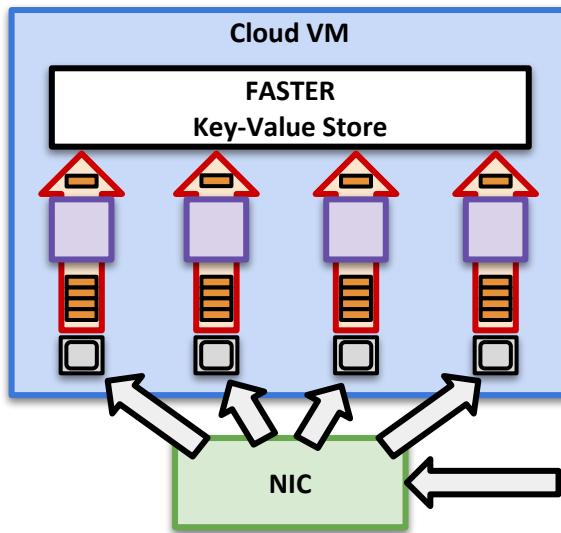
Requests are
async

Enqueued
inside
session
buffers

Shadowfax: Partitioned Sessions Shared Data

Problem: Keep servers saturated at index, not network or dispatch

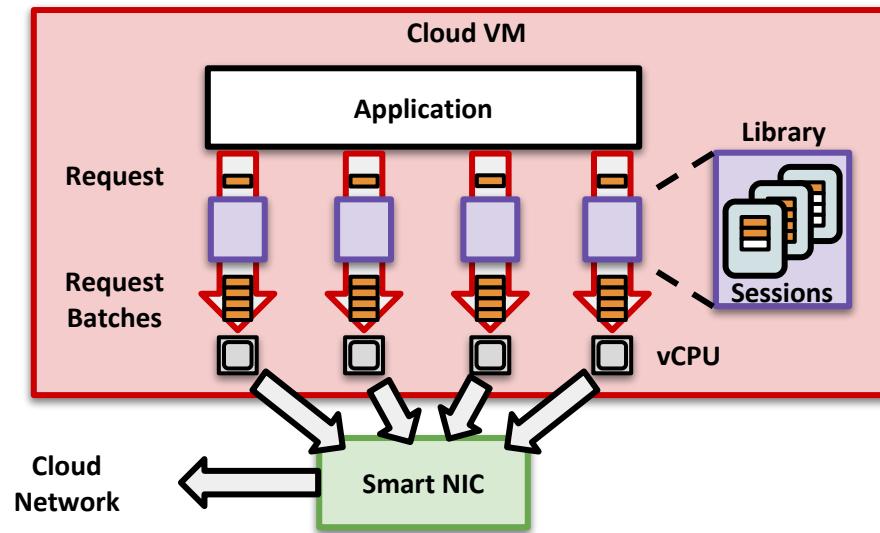
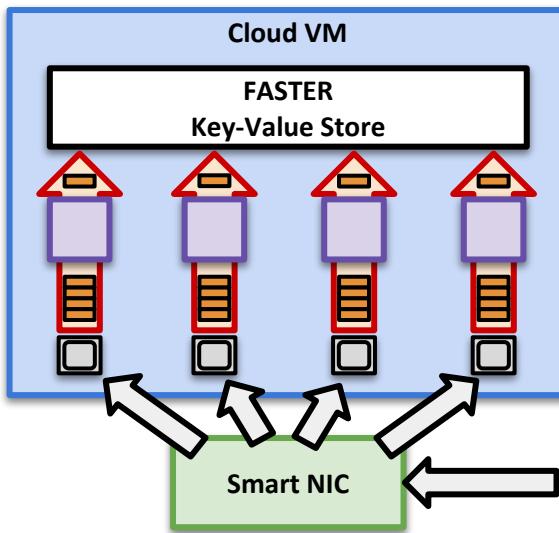
Solution: Asynchronous client library, transparent network acceleration



Shadowfax: Partitioned Sessions Shared Data

Problem: Keep servers saturated at index, not network or dispatch

Solution: Asynchronous client library, transparent network acceleration

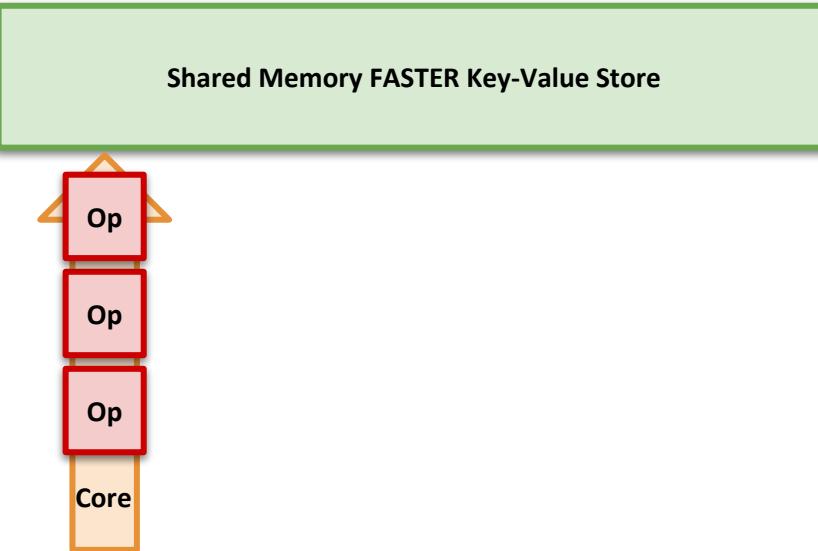


Hardware accelerated transport (TCP & Infrc)

Shadowfax: Asynchronous Global Cuts

Problem: Avoid cross-core coordination during migration

Solution: Server and client cores observe ownership change independently



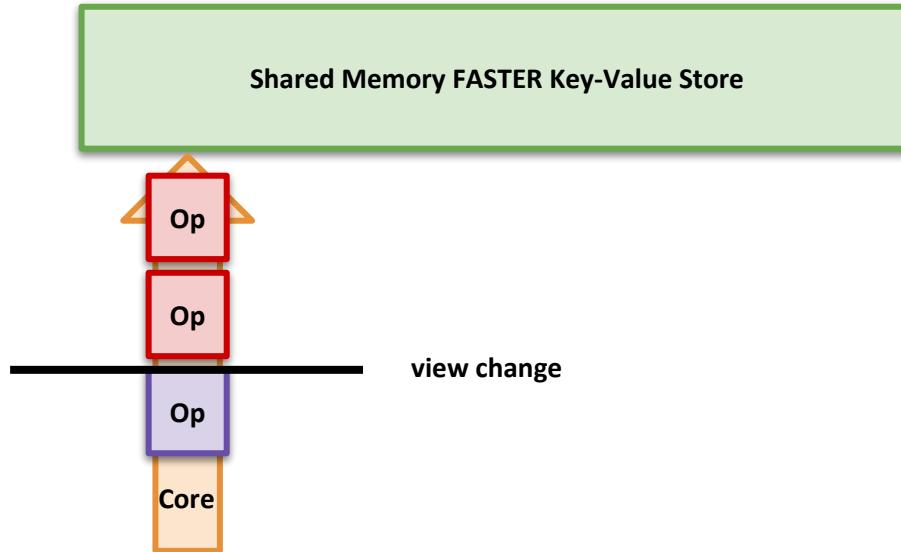
Migration
changes server's
ownership

“View” change

Shadowfax: Asynchronous Global Cuts

Problem: Avoid cross-core coordination during migration

Solution: Server and client cores observe ownership change independently

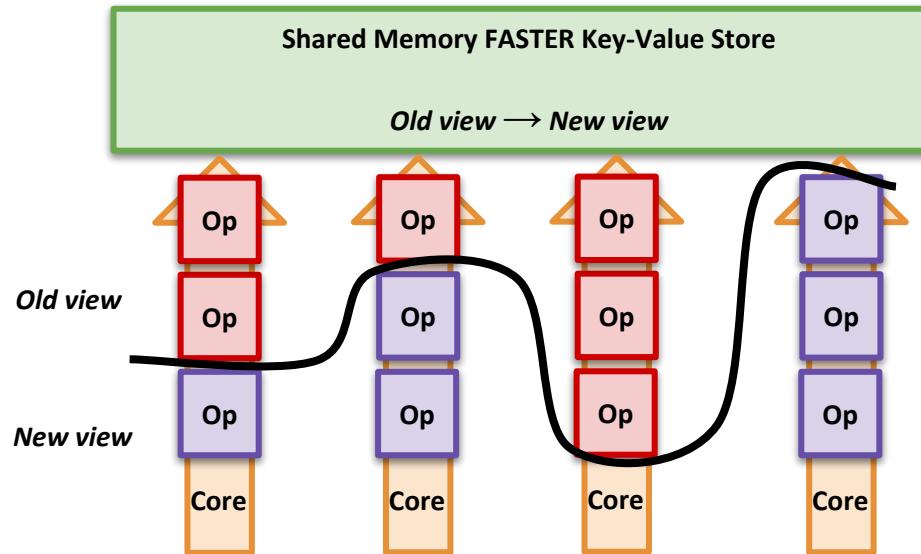


Operations after
view change
must be
retransmitted to
new owner

Shadowfax: Asynchronous Global Cuts

Problem: Avoid cross-core coordination during migration

Solution: Server and client cores observe ownership change independently



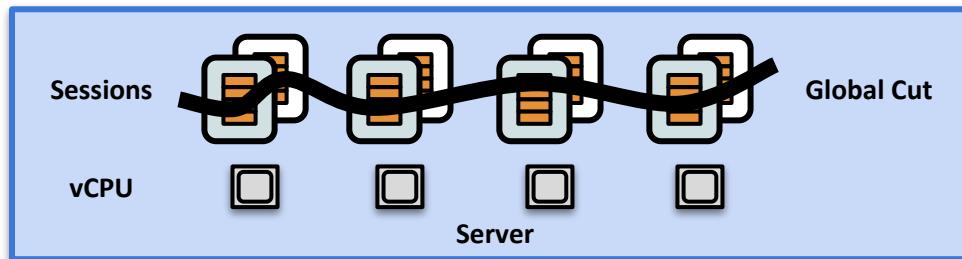
Each core observes change independently

“Async cut” avoids coordination within server

Shadowfax: Asynchronous Global Cuts

Problem: Avoid cross-core coordination during migration

Solution: Server and client cores observe ownership change independently

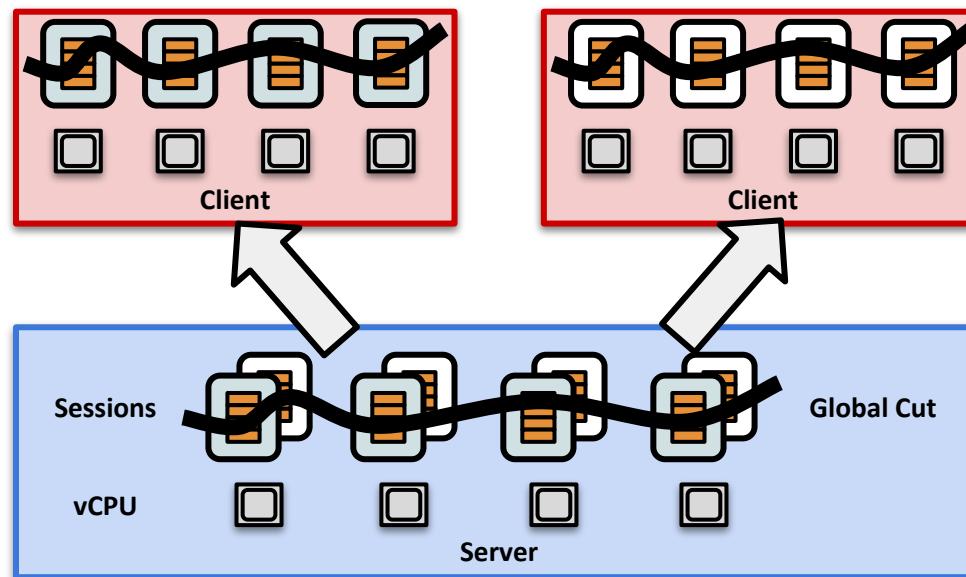


Async cut becomes "*global cut*" across sessions

Shadowfax: Asynchronous Global Cuts

Problem: Avoid cross-core coordination during migration

Solution: Server and client cores observe ownership change independently



Global cut
pushed to clients

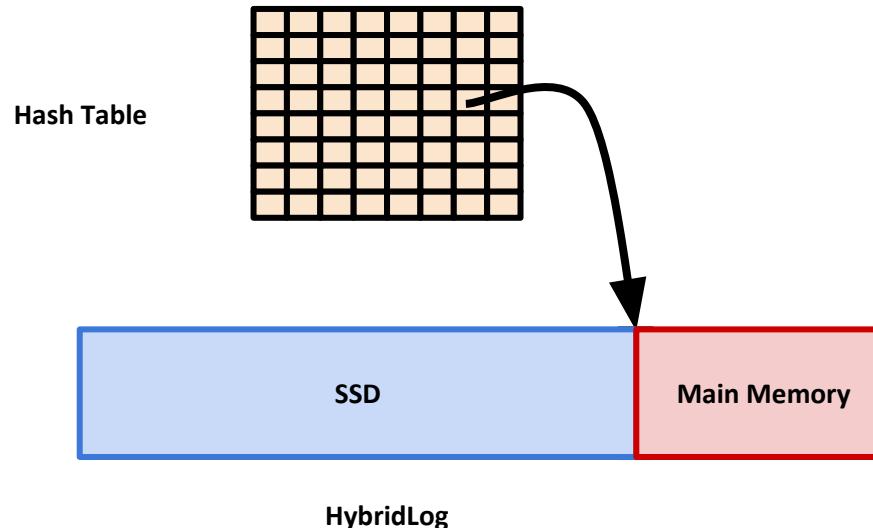
Avoids
coordination at
clients

Clients reissue
operations

Shadowfax: Indirection Records

Problem: Migrating records on SSD can slow down reconfiguration

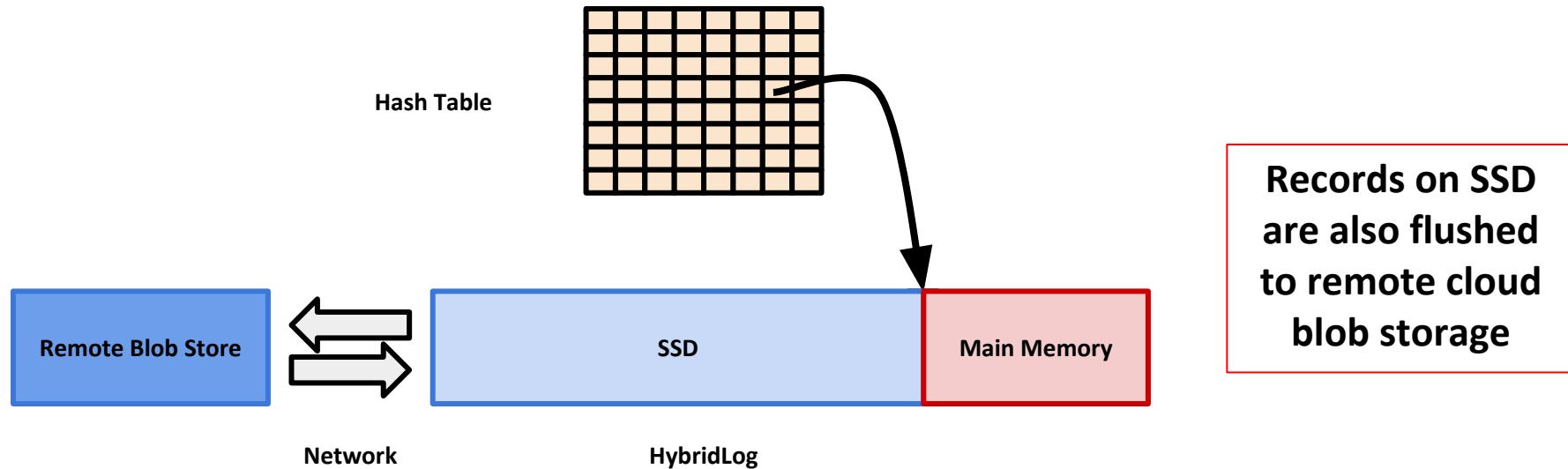
Solution: Use shared remote tier to restrict migration to main memory



Shadowfax: Indirection Records

Problem: Migrating records on SSD can slow down reconfiguration

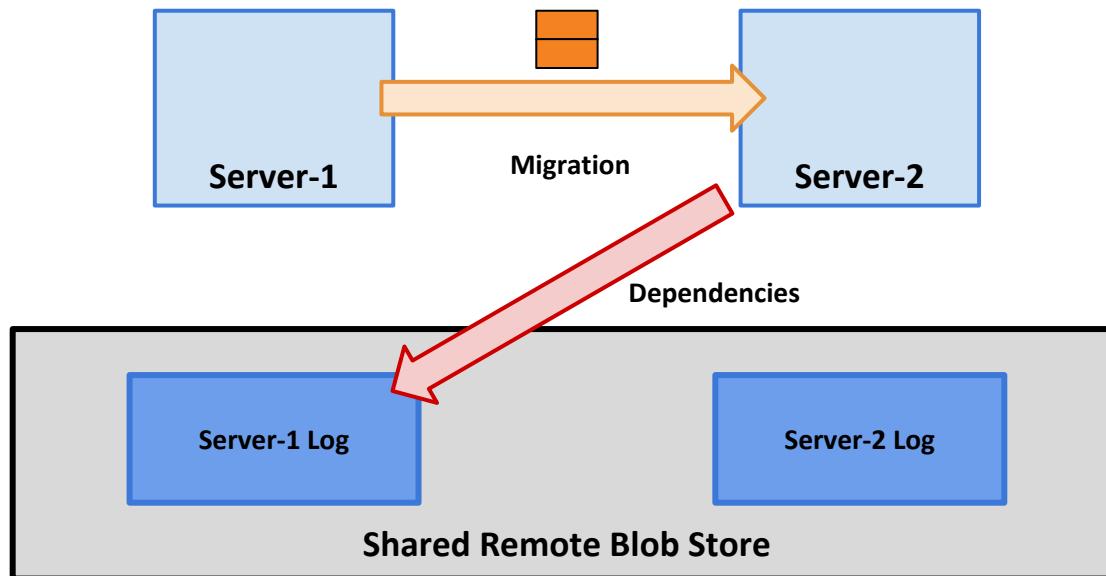
Solution: Use shared remote tier to restrict migration to main memory



Shadowfax: Indirection Records

Problem: Migrating records on SSD can slow down reconfiguration

Solution: Use shared remote tier to restrict migration to main memory

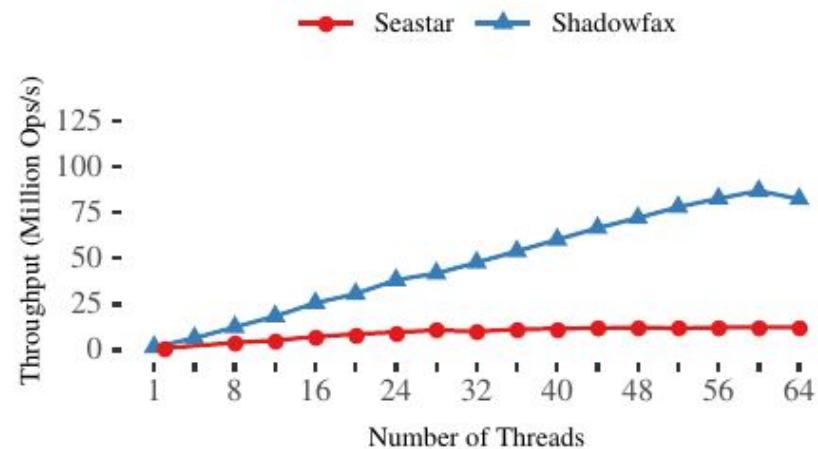
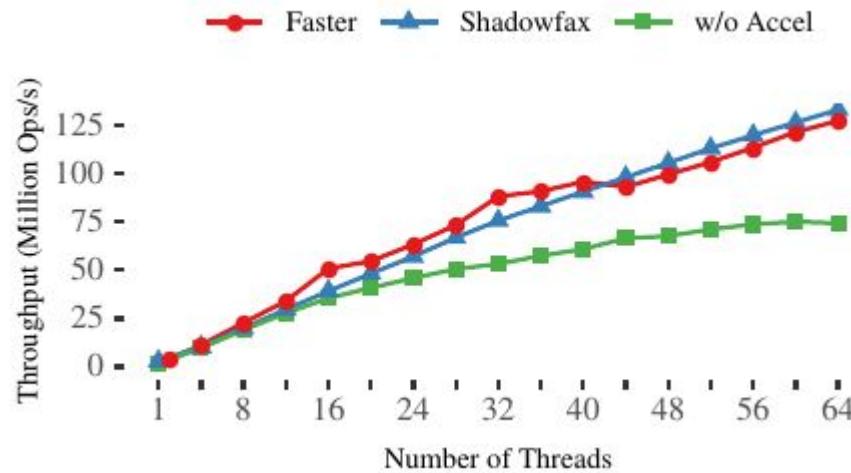


Migrate pointers
to shared layer

Lazily cleaned up

Performance of Shadowfax

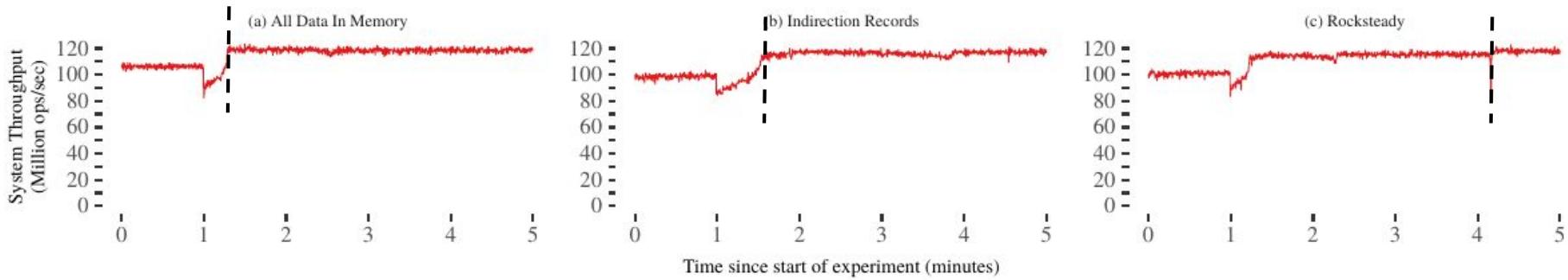
YCSB-F (Read-Modify-Writes), 250 Million objects



Saturates server at Index, 8.5x state-of-the-art

Performance of Shadowfax

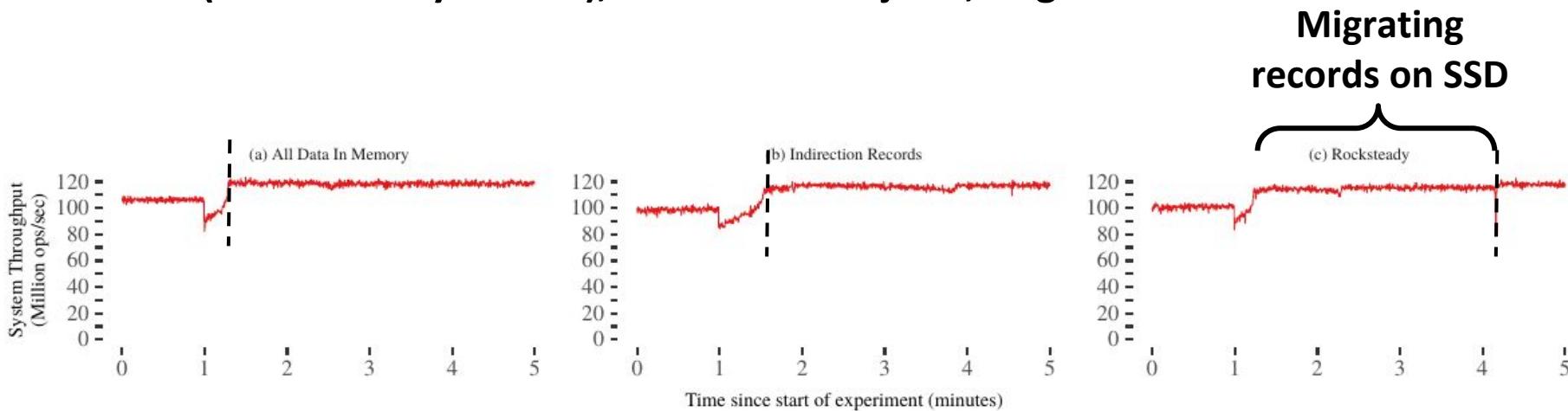
YCSB-F (Read-Modify-Writes), 250 Million objects, Migrate 10%



Throughput maintained > 80 Million events/second

Performance of Shadowfax

YCSB-F (Read-Modify-Writes), 250 Million objects, Migrate 10%



Indirection records speed up migration by 6x

Shadowfax: TODO

Analyze larger-than-memory aspect of system

- Throughput when data does not fit in main memory
- What limits performance? What should change to improve performance?

Extensibility and Multi-tenancy

Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. **Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage.** In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI'18*

Ankit Bhardwaj, **Chinmay Kulkarni, and Ryan Stutsman.** **Adaptive Placement for In-memory Storage Functions.** In *Proceedings of the 2020 USENIX Annual Technical Conference, ATC'20*

Introduction

Kernel-bypass key-value stores offer **< 10µs** latency, **> Mops/s** throughput

- Fast because they support just simple gets and puts?

Problem: Leverage performance → **share** between tenants

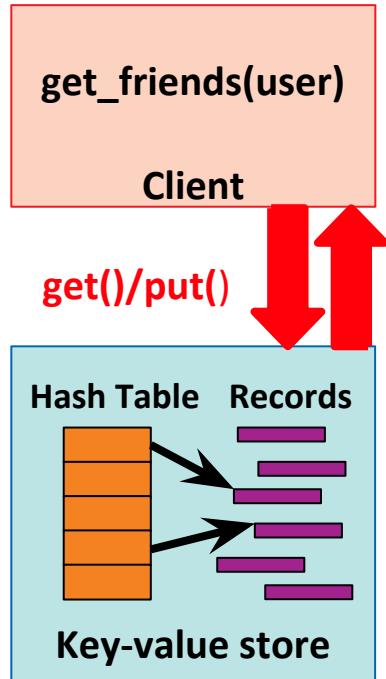
Problem: Apps require rich data models. Ex: Facebook's TAO

- Implement using gets & puts? → **Data movement, client stalls**
- Push code to key-value store? → **Isolation costs limit density**

Splinter: Multi-tenant key-value store that code can be pushed to

- Tenants push type- & memory-safe code written in **Rust** at runtime
- Aggregate workload improves by **2x**, Facebook's TAO by **400 Kops/s**

Richer Data Models Come At A Price



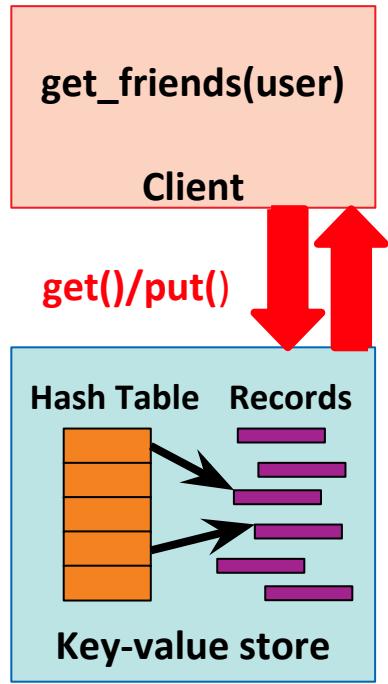
Apps require rich data models in addition to performance

- Ex: Social graphs, Decision trees etc

Key-value stores trade-off data model for performance

- Simple get()'s & put()'s over key-value pairs

Richer Data Models Come At A Price



Apps require rich data models in addition to performance

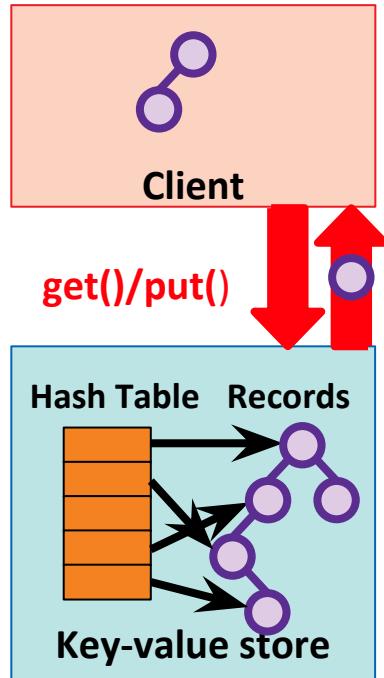
- Ex: Social graphs, Decision trees etc

Key-value stores trade-off data model for performance

- Simple get()'s & put()'s over key-value pairs

**Thinner data model → Better performance
But do applications benefit?**

Extra Round-Trips (RTTs) Hurt Latency & Utilization



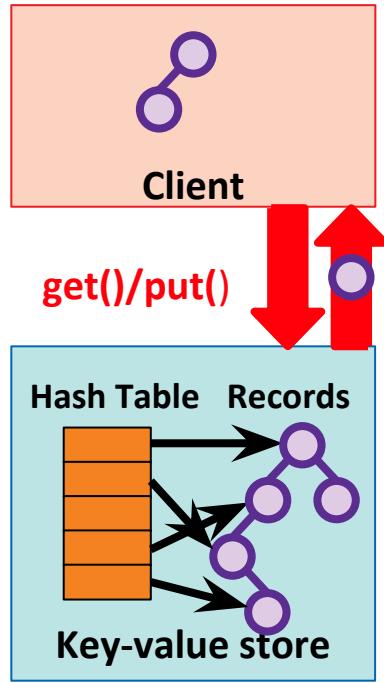
Example: Traverse tree with N nodes using gets

- One `get()` at each level of the tree → **$O(\log N)$ RTTs**
- Control flow depends on data → **Client stalls during get()**

Network RTTs, dispatch are the main bottleneck $\sim 10\mu s$

- $1.5\mu s$ inside the server

Extra Round-Trips (RTTs) Hurt Latency & Utilization



Example: Traverse tree with N nodes using gets

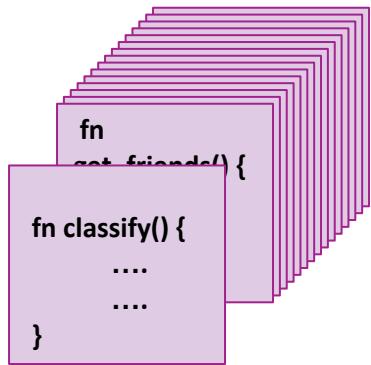
- One `get()` at each level of the tree → **O(log N) RTTs**
- Control flow depends on data → **Client stalls during get()**

Network RTTs, dispatch are the main bottleneck ~10µs

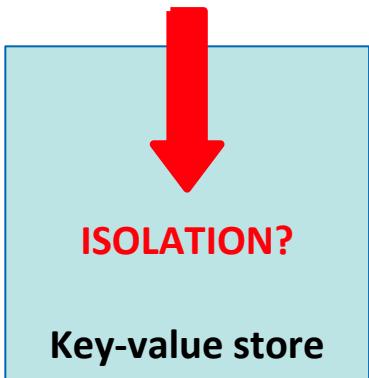
- 1.5µs inside the server

So push code to storage?

Why Not Push Compute To Storage?



RPC Processing Time $\sim 1.5\mu s$
Only native code will do



Context Switches $\sim 1.5\mu s$
Multi-tenancy → Need hardware isolation

What Do We Want From The Storage Layer?

Granularity of compute is steadily decreasing

Virtual machines → Containers → Lambdas

Extremely high tenant density

- Fine-grained resource allocation; 100s of CPU cycles, Kilobytes of memory

Allow tenants to extend data model at runtime

- Low overhead isolation between tenants & storage layer

Splinter: A Multi-Tenant Key-Value Store

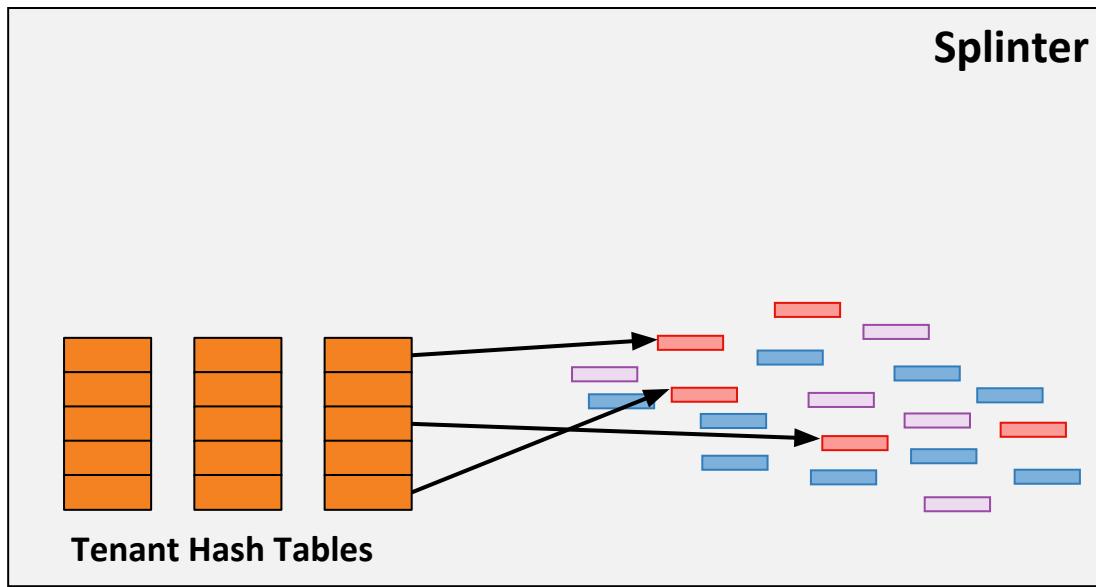
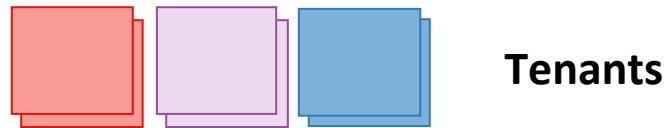
Tenants can install and invoke extensions at runtime

- Extensions written in **Rust**
- Rely on type and memory safety for isolation, avoids context switch

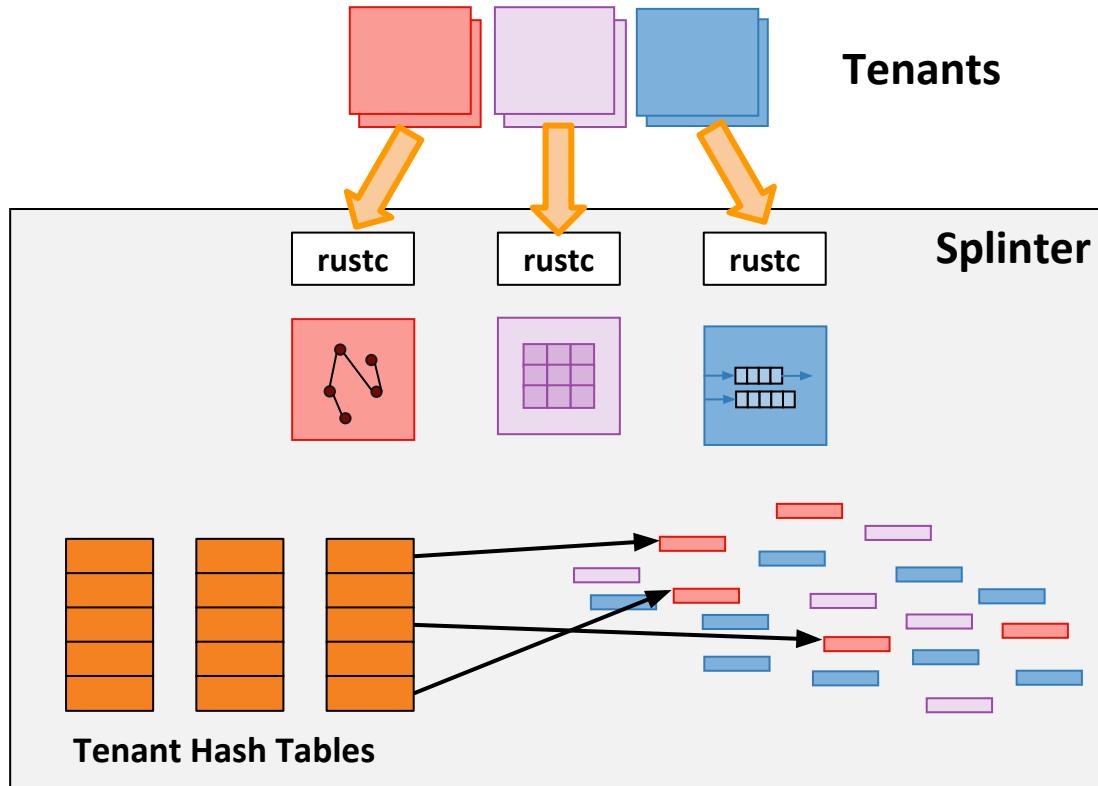
Implemented in ~9000 lines of Rust

- Supports two RPCs → **install(ext_name) & invoke(ext_name)**
- Also supports regular get() & put() RPCs → “**Native**” operations

1000 Foot View Of Splinter



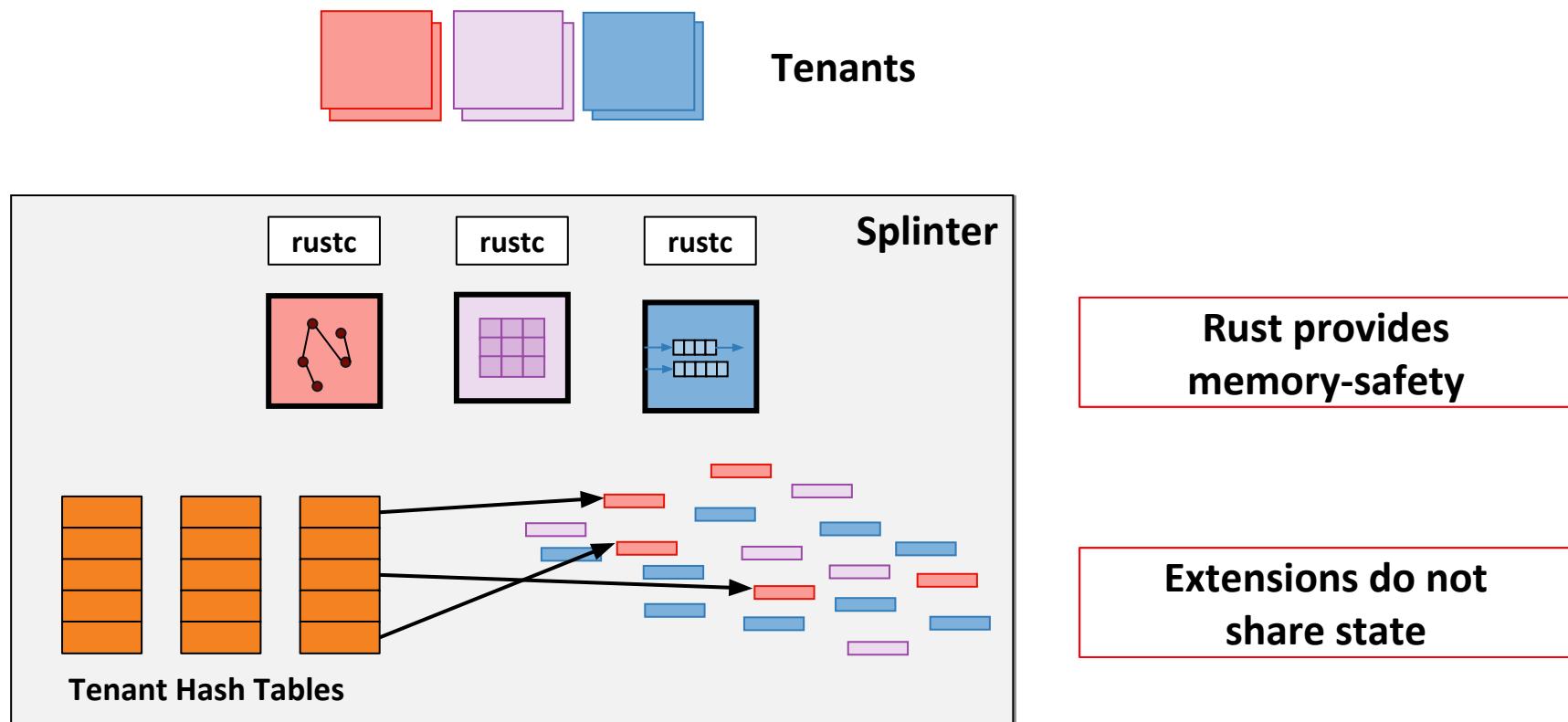
1000 Foot View Of Splinter



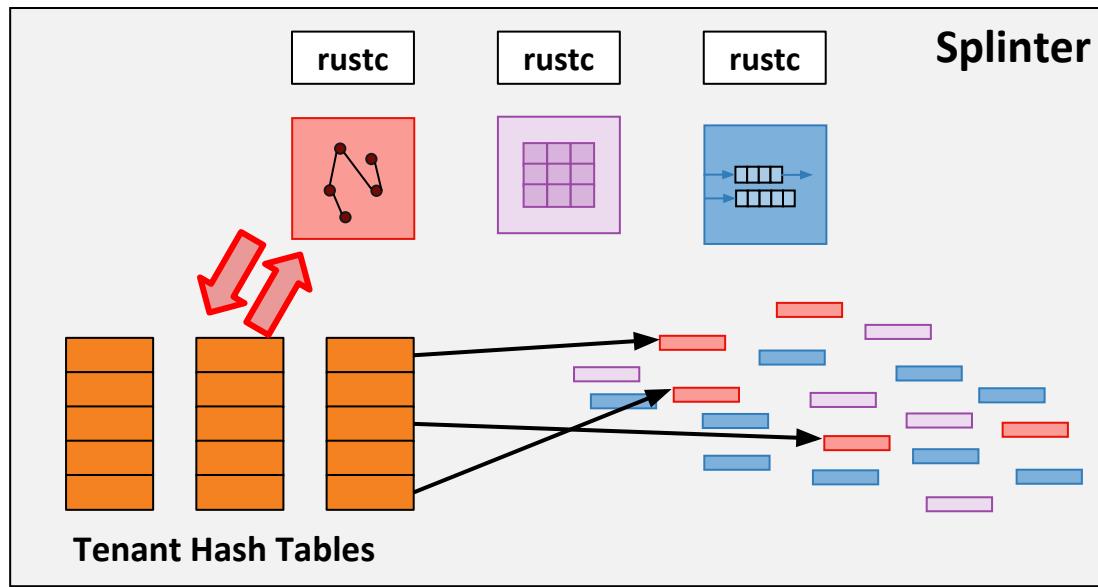
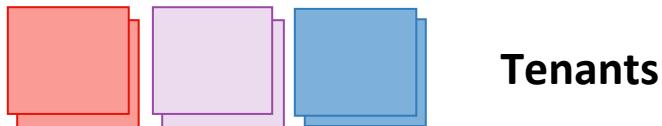
Tenants push extensions
written in Rust

Splinter
loads extensions
Into address-space

1000 Foot View Of Splinter



1000 Foot View Of Splinter



Extensions receive references to records

Each tenant sees a custom key-value store

Splinter: Design

Low cost isolation

- No forced data copies across trust boundary

Tenant Locality And Work Stealing

- Avoid cross-core coordination while avoiding hotspots

Lightweight Cooperative Scheduling

- Prevent long running extensions from starving short running ones

Splinter: Low Cost Isolation

Problem: No forced data copies across trust boundary

Solution: Ensure buffers outlast reference lifetime

```
aggregate() → u64 {  
    ...  
    ...  
}
```

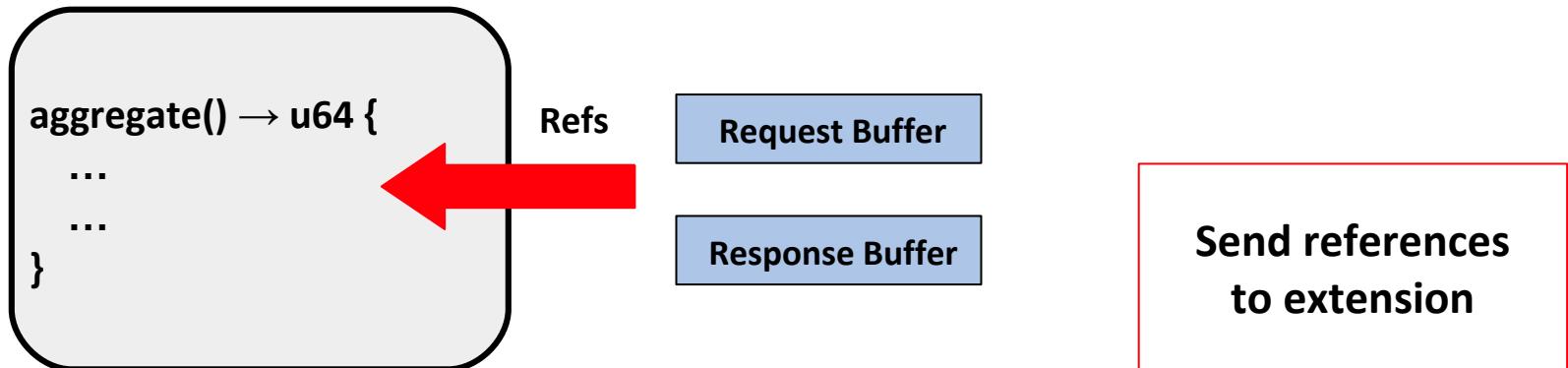
Request Buffer

Response Buffer

Splinter: Low Cost Isolation

Problem: No forced data copies across trust boundary

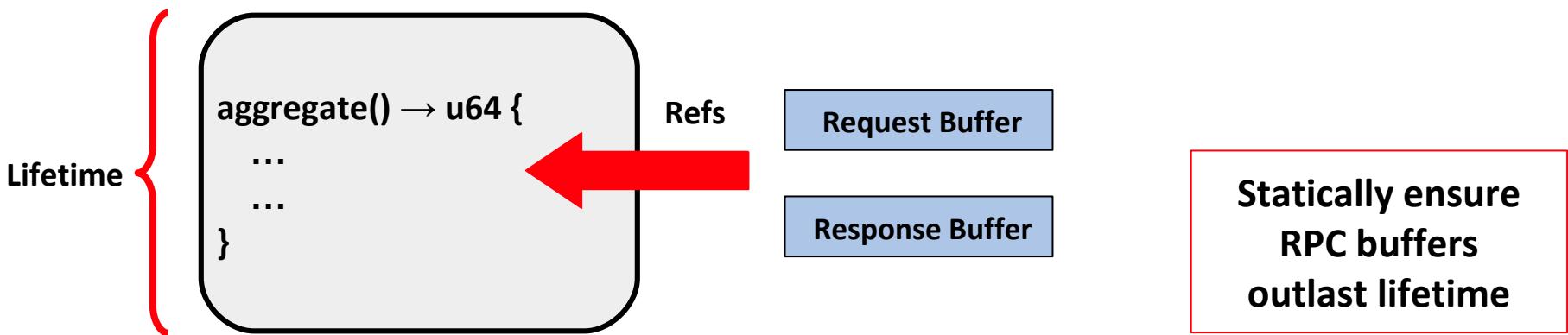
Solution: Ensure buffers outlast reference lifetime



Splinter: Low Cost Isolation

Problem: No forced data copies across trust boundary

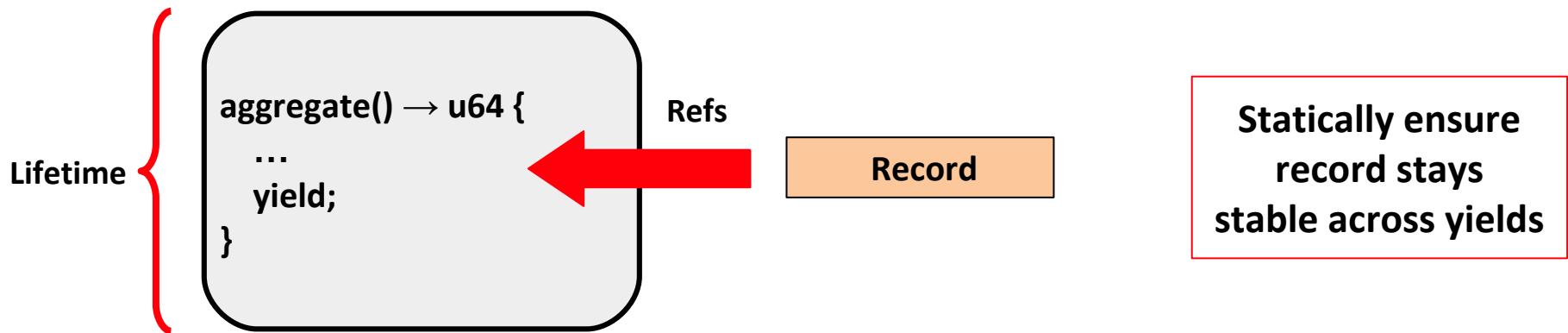
Solution: Ensure buffers outlast reference lifetime



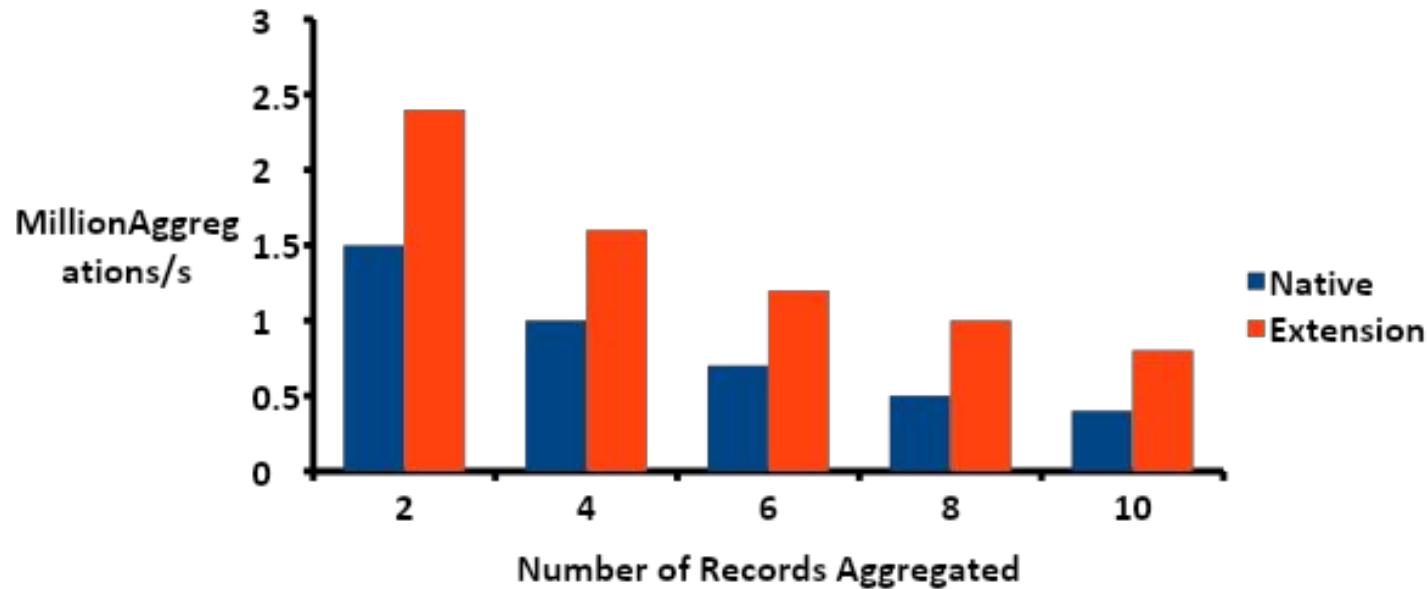
Splinter: Low Cost Isolation

Problem: No forced data copies across trust boundary

Solution: Ensure buffers outlast reference lifetime

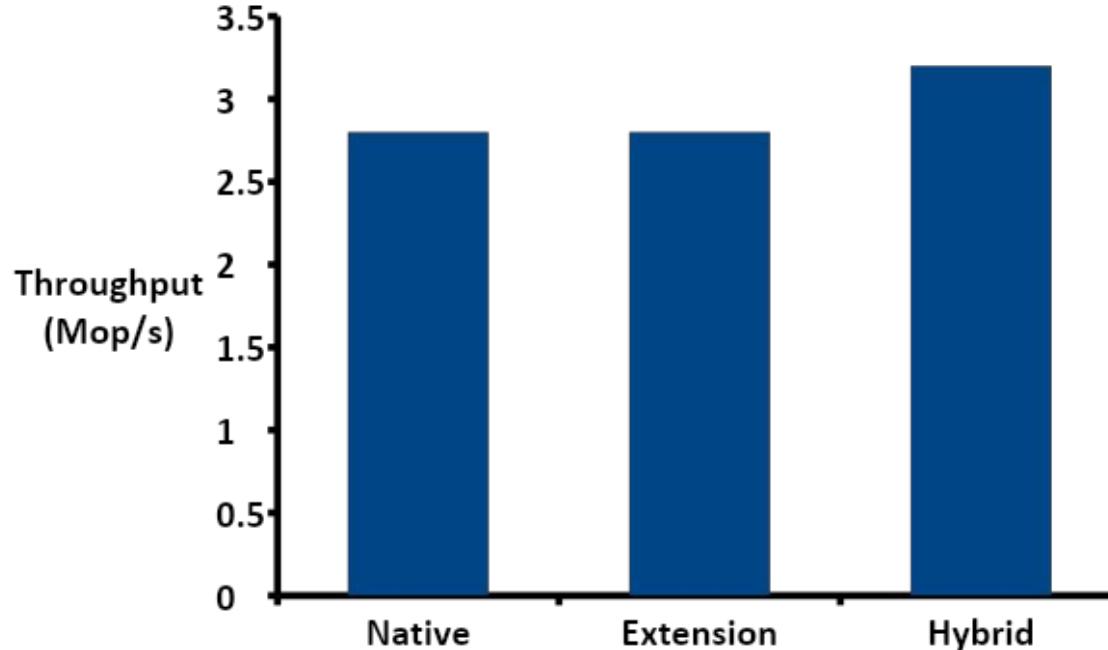


Simple Aggregation With Splinter



Extension Mode → Few RPCs, Less Data movement → Better Throughput

Pushing Facebook's TAO To Splinter



Hybrid → get() for point ops, extension for dependencies
Best of both worlds!

Adaptive Placement for In-Memory Storage Functions

Ankit Bhardwaj (Labmate) has built on Splinter

- For certain functions, client-side execution (gets & puts) is better

ASFP: Adaptive Storage Function Placement

- Decouples function placement from execution
- Dynamically models and decides where to execute functions
- Published at ATC'20 (I am the second author)

Thesis Contributions

Fast Data Migration [[Rocksteady, SOSP'17](#)]

Low-cost Coordination [[Shadowfax, VLDB'21, Revision](#)]

Extensibility and Multi-tenancy [[Splinter, OSDI'18](#)]

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

Thesis Contributions

Fast Data Migration [[Rocksteady, SOSP'17](#)]

Create headroom

Low-cost Coordination [[Shadowfax, VLDB'21](#), [Revision](#)]

On-demand
migration of hot
keys

Extensibility and Multi-tenancy [[Splinter, OSDI'18](#)]

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

Thesis Contributions

Fast Data Migration [[Rocksteady, SOSP'17](#)]

Partitioned Dispatch
Shared Data

Low-cost Coordination [[Shadowfax, VLDB'21, Revision](#)]

Asynchronous Global
Cuts

Extensibility and Multi-tenancy [[Splinter, OSDI'18](#)]

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

Thesis Contributions

Fast Data Migration [[Rocksteady, SOSP'17](#)]

Low-cost Isolation

Low-cost Coordination [[Shadowfax, VLDB'21, Revision](#)]

Co-op scheduling
Tenant Locality

Extensibility and Multi-tenancy [[Splinter, OSDI'18](#)]

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

Thesis Timeline

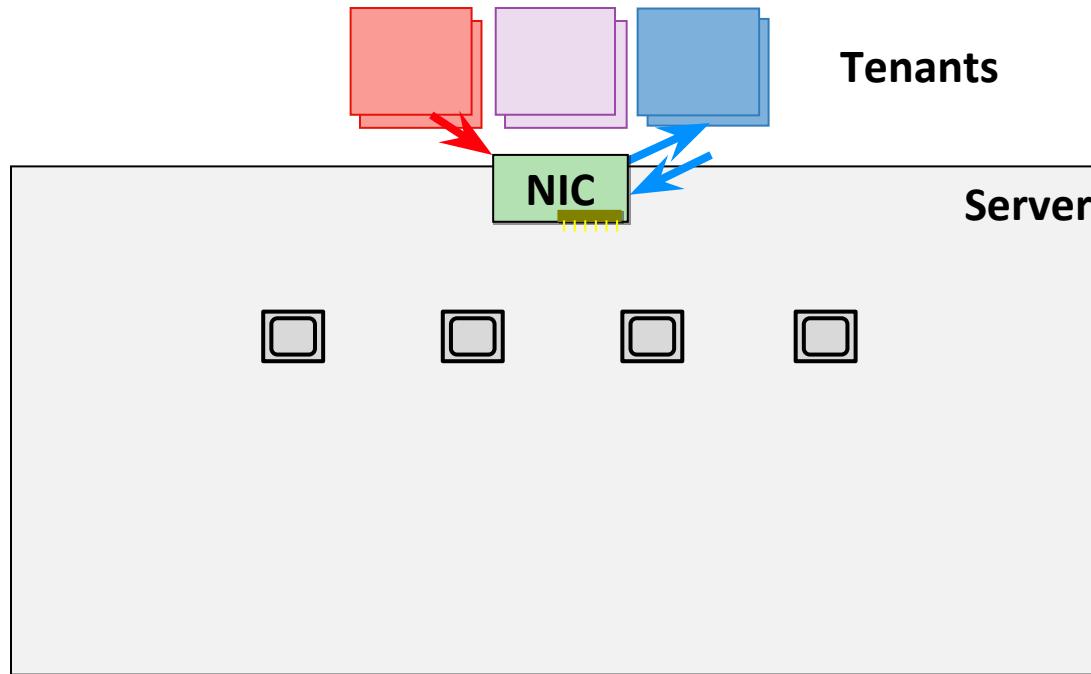
Task 1	Develop fast data migration protocol for low-latency stores	Feb 2017	Done
Task 2	Write up and submit results to SOSP 2017	Apr 2017	Done
Task 3	Develop mechanism for extensibility and multi-tenancy for low-latency stores	Mar 2018	Done
Task 4	Write up and submit results to OSDI 2018	May 2018	Done
Task 5	Develop mechanisms for low cost coordination in low-latency stores	Mar 2019	Done
Task 6	Develop mechanisms for scale-out with cloud storage in low-latency stores	Mar 2020	Done
Task 7	Write up and submit results to VLDB 2021	Sep 2020	Done
Task 8	Finish Writing Thesis Introduction	Nov 2020	Done
Task 9	Analyze normal-case store performance with cloud storage	Dec 2020	TBD
Task 10	Finish writing low cost coordination chapter	Jan 2021	TBD
Task 11	Finish writing Conclusion	Jan 2021	TBD
Task 12	Final thesis defense	Feb 2021	TBD

Backup Slides

Splinter: Tenant Locality And Work Stealing

Problem: Quickly dispatch requests to cores, avoid hotspots

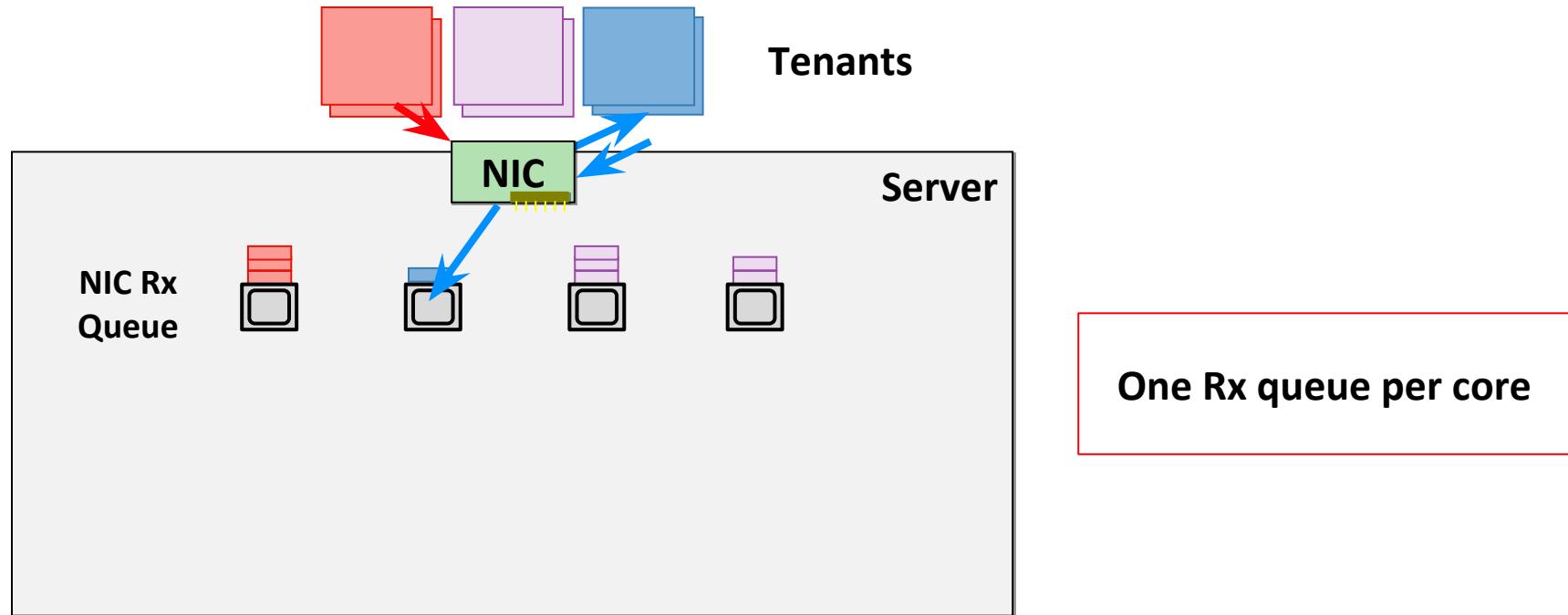
Solution: NIC routes tenants to cores, cores steal work



Splinter: Tenant Locality And Work Stealing

Problem: Quickly dispatch requests to cores, avoid hotspots

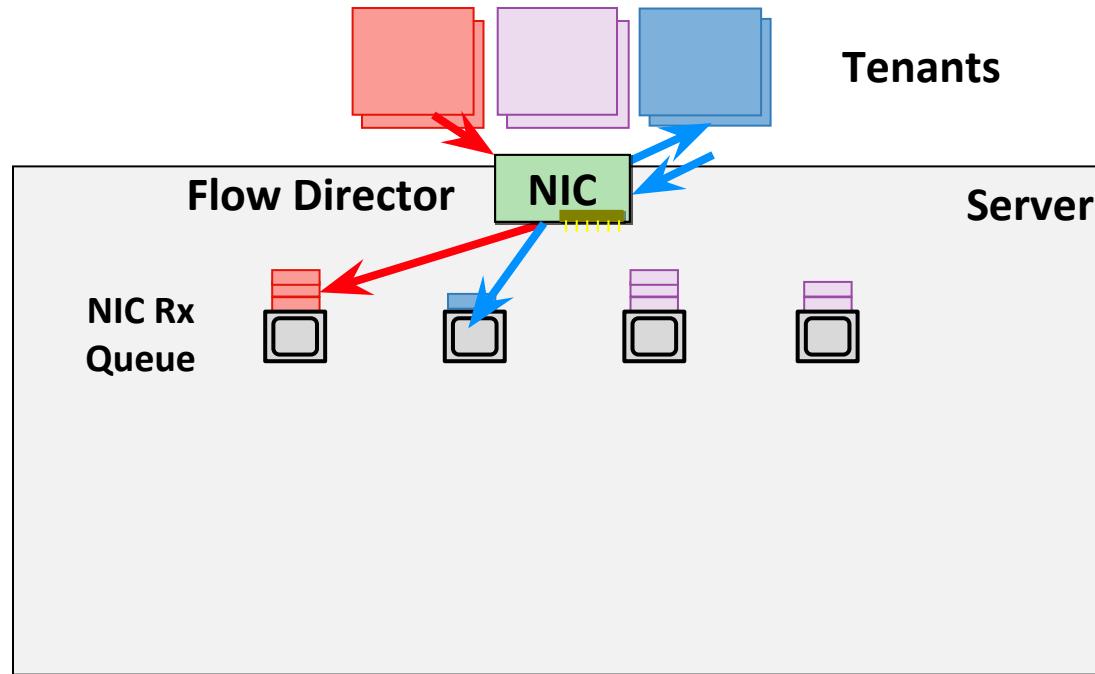
Solution: NIC routes tenants to cores, cores steal work



Splinter: Tenant Locality And Work Stealing

Problem: Quickly dispatch requests to cores, avoid hotspots

Solution: NIC routes tenants to cores, cores steal work

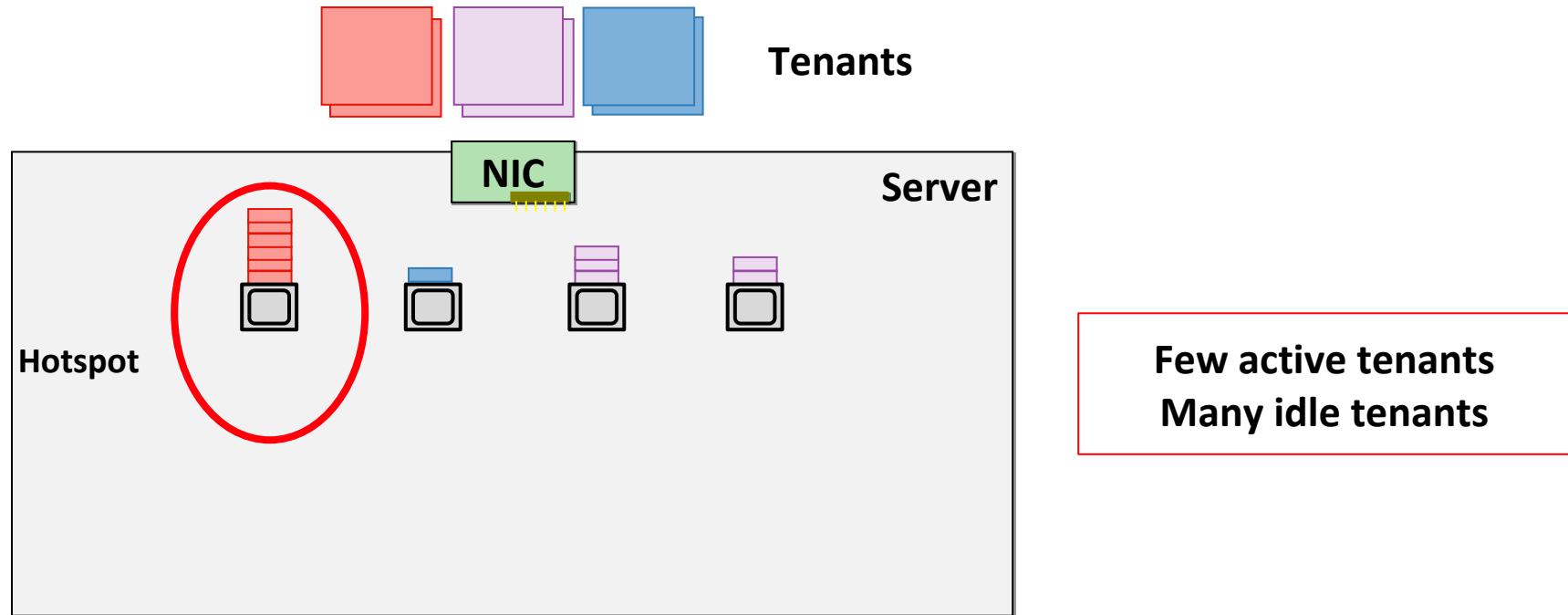


Maintain “Locality”
route tenant to queue

Splinter: Tenant Locality And Work Stealing

Problem: Quickly dispatch requests to cores, avoid hotspots

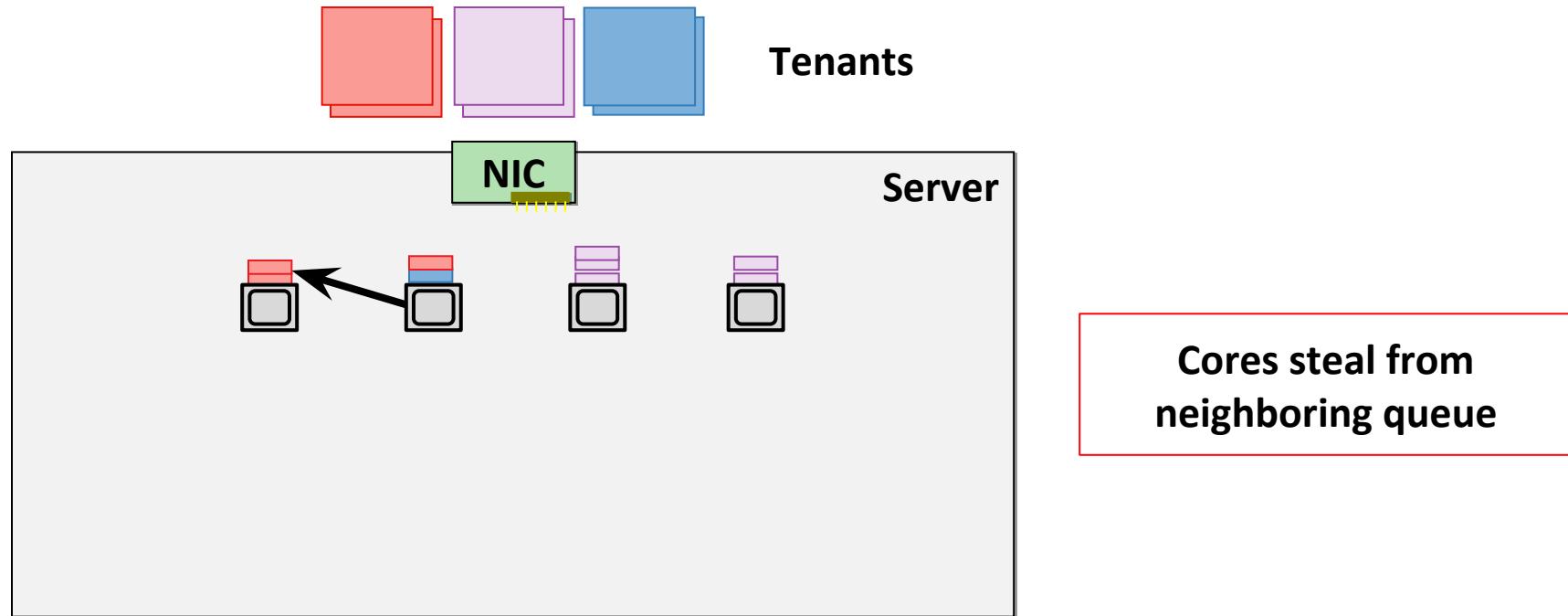
Solution: NIC routes tenants to cores, cores steal work



Splinter: Tenant Locality And Work Stealing

Problem: Quickly dispatch requests to cores, avoid hotspots

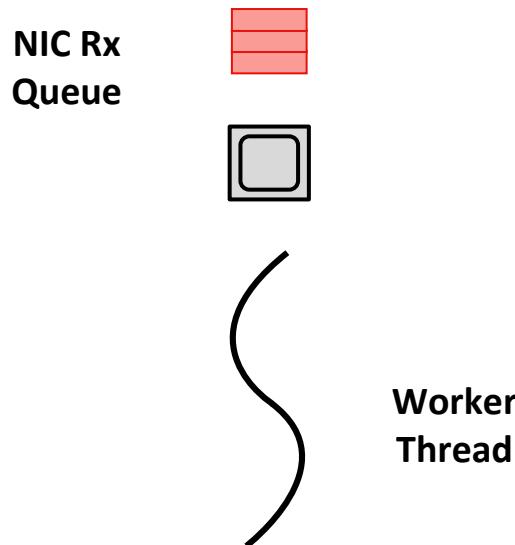
Solution: NIC routes tenants to cores, cores steal work



Splinter: Lightweight Cooperative Scheduling

Problem: Minimize trust boundary crossing cost

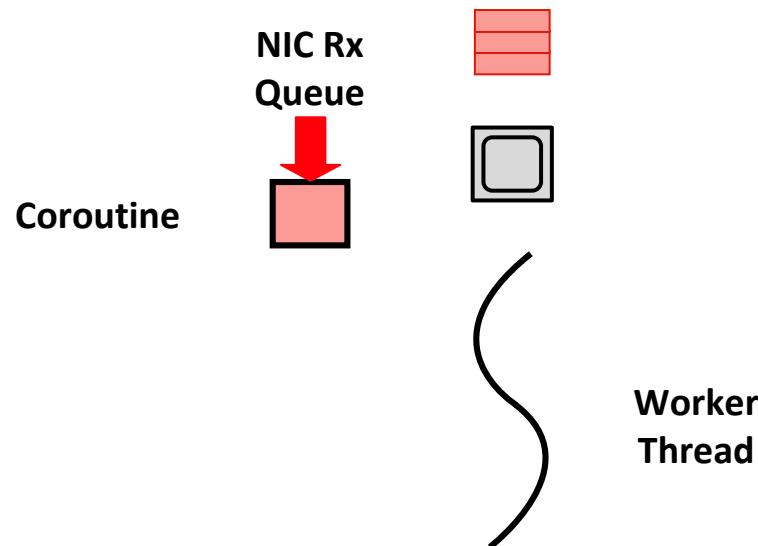
Solution: Run extensions in stackless coroutines



Splinter: Lightweight Cooperative Scheduling

Problem: Minimize trust boundary crossing cost

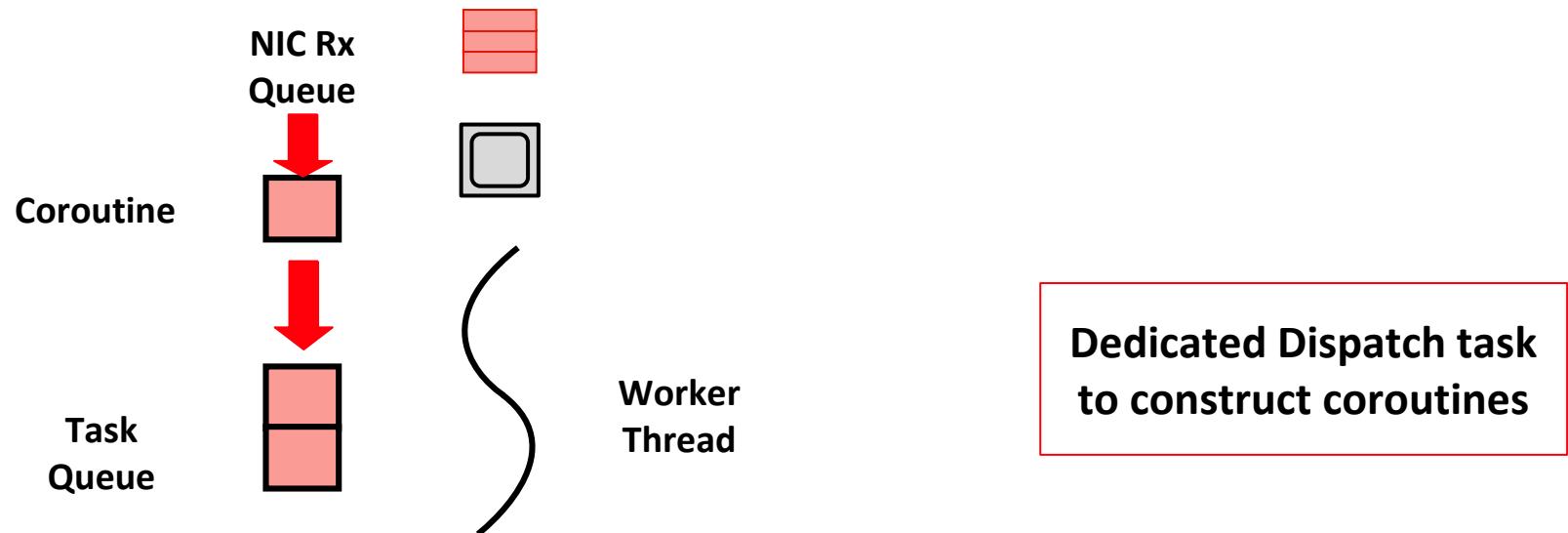
Solution: Run extensions in stackless coroutines



Splinter: Lightweight Cooperative Scheduling

Problem: Minimize trust boundary crossing cost

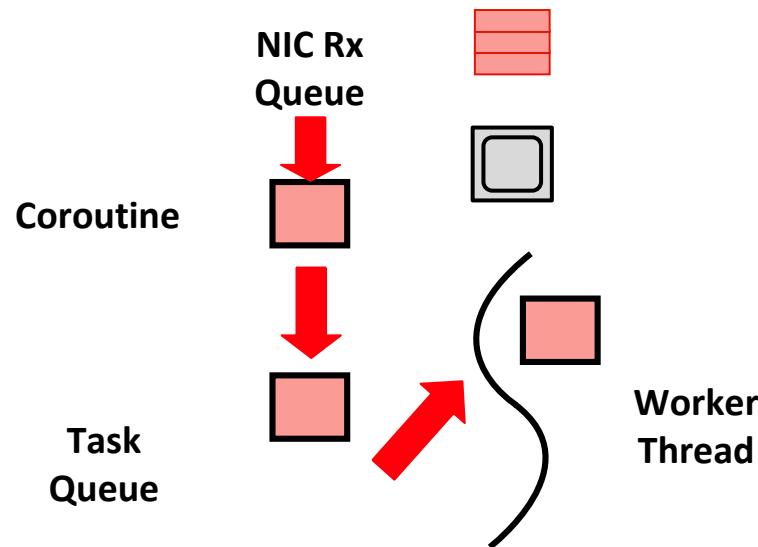
Solution: Run extensions in stackless coroutines



Splinter: Lightweight Cooperative Scheduling

Problem: Minimize trust boundary crossing cost

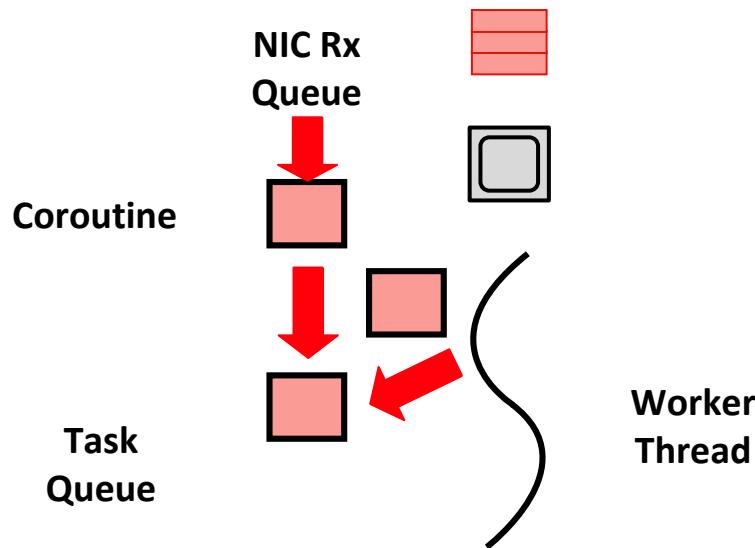
Solution: Run extensions in stackless coroutines



Splinter: Lightweight Cooperative Scheduling

Problem: Minimize trust boundary crossing cost

Solution: Run extensions in stackless coroutines



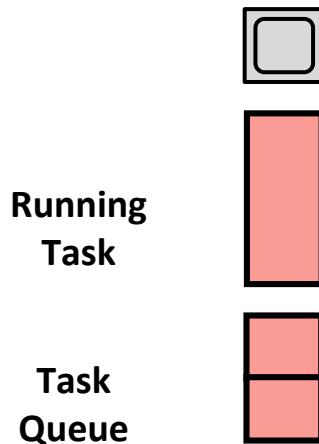
Task switch cost ~10ns

Run extension
until it returns

Splinter: Lightweight Cooperative Scheduling

Problem: Long running tasks starve shorter tasks, hurt latency

Solution: Extensions are cooperative, must yield frequently

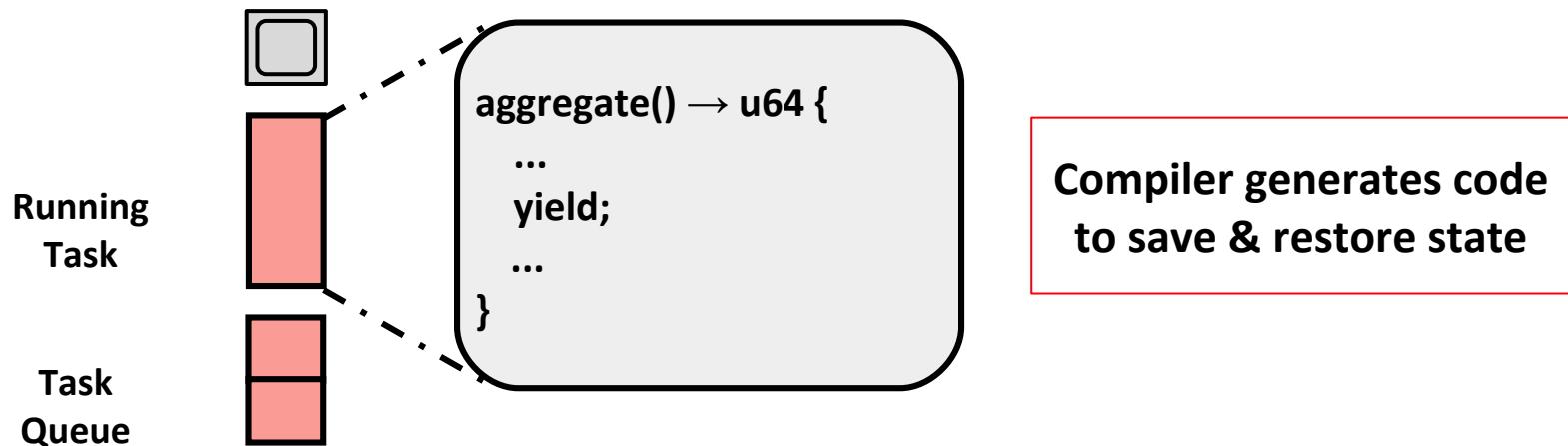


Preemption is too
expensive!

Splinter: Lightweight Cooperative Scheduling

Problem: Long running tasks starve shorter tasks, hurt latency

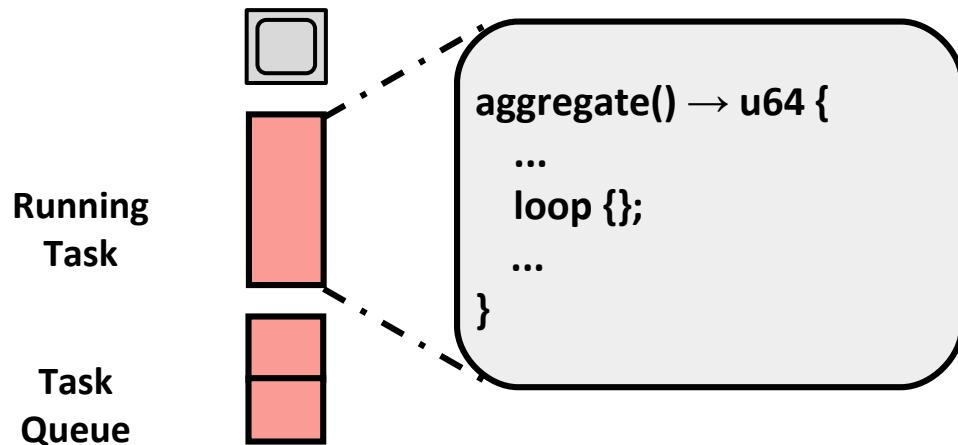
Solution: Extensions are cooperative, must yield frequently



Splinter: Lightweight Cooperative Scheduling

Problem: Uncooperative extensions

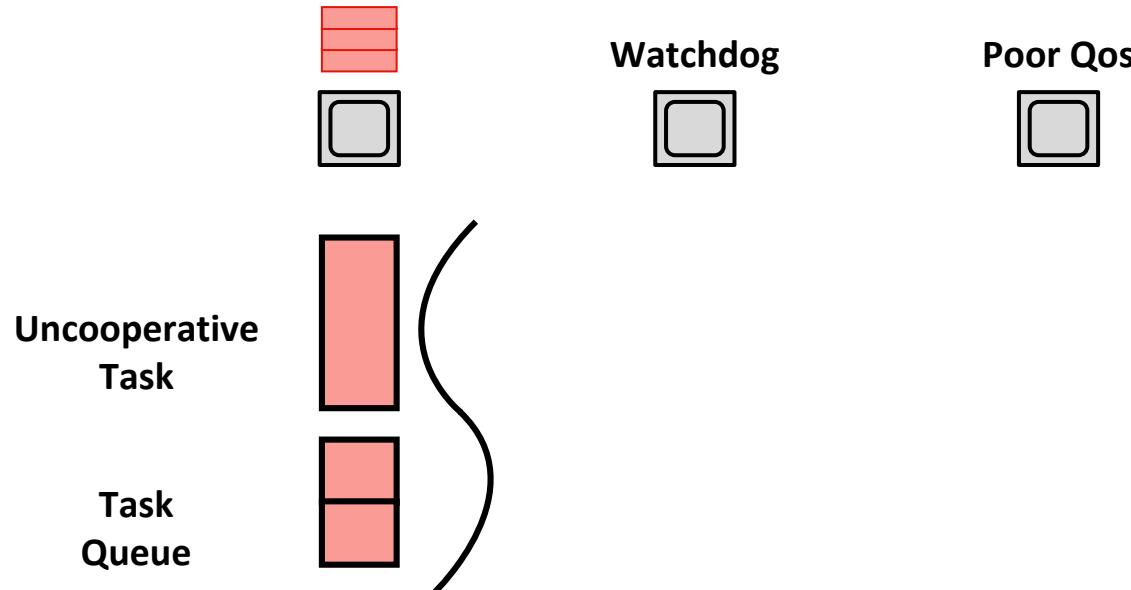
Solution: Trusted watchdog core



Splinter: Lightweight Cooperative Scheduling

Problem: Uncooperative extensions

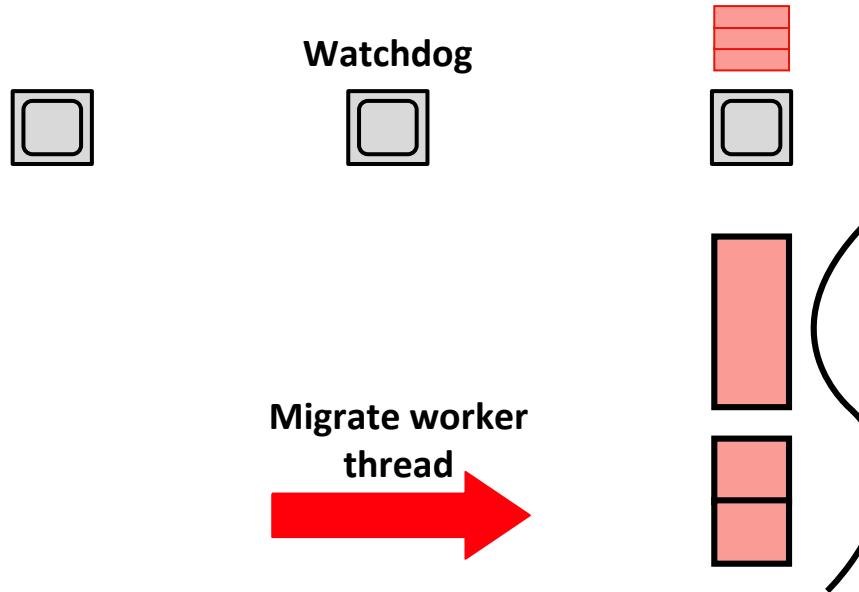
Solution: Trusted watchdog core



Splinter: Lightweight Cooperative Scheduling

Problem: Uncooperative extensions

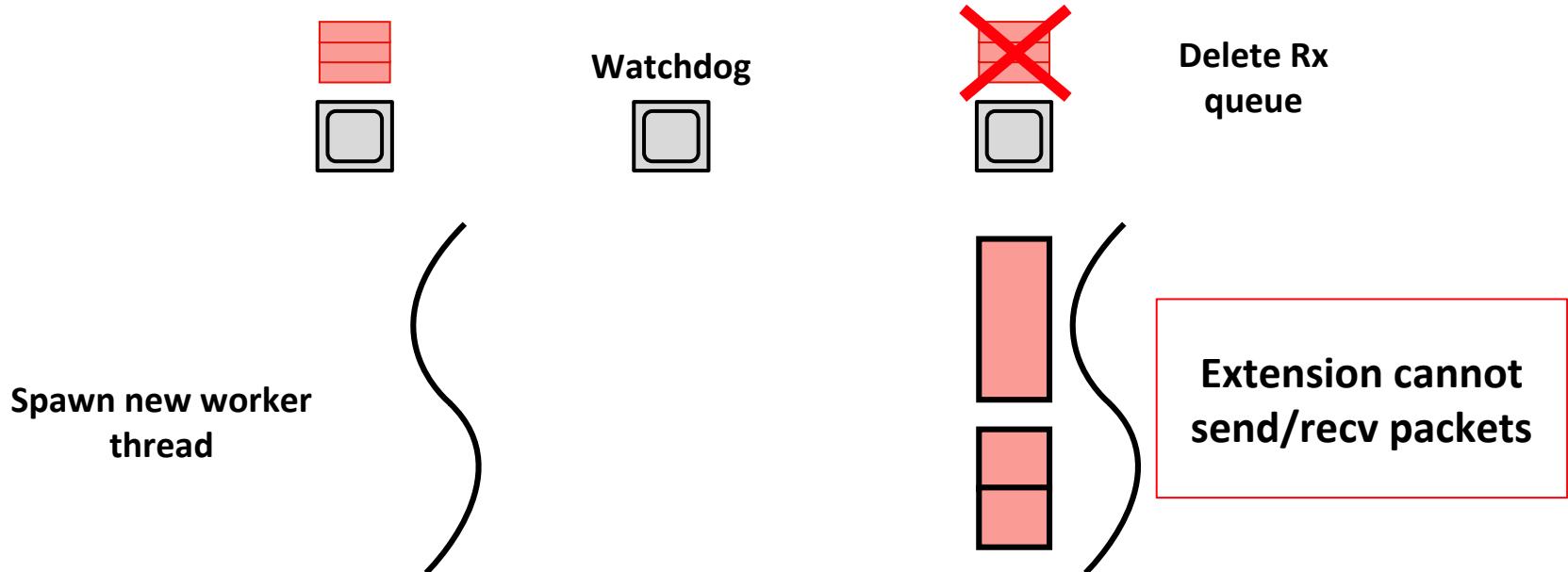
Solution: Trusted watchdog core



Splinter: Lightweight Cooperative Scheduling

Problem: Uncooperative extensions

Solution: Trusted watchdog core



Splinter: Lightweight Cooperative Scheduling

Problem: Uncooperative extensions

Solution: Trusted watchdog core

