ACM SIGMOD
PODS 2022
Philadelphia, PA, USA

Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory

Qing Wang, Youyou Lu, Jiwu Shu

Tsinghua University

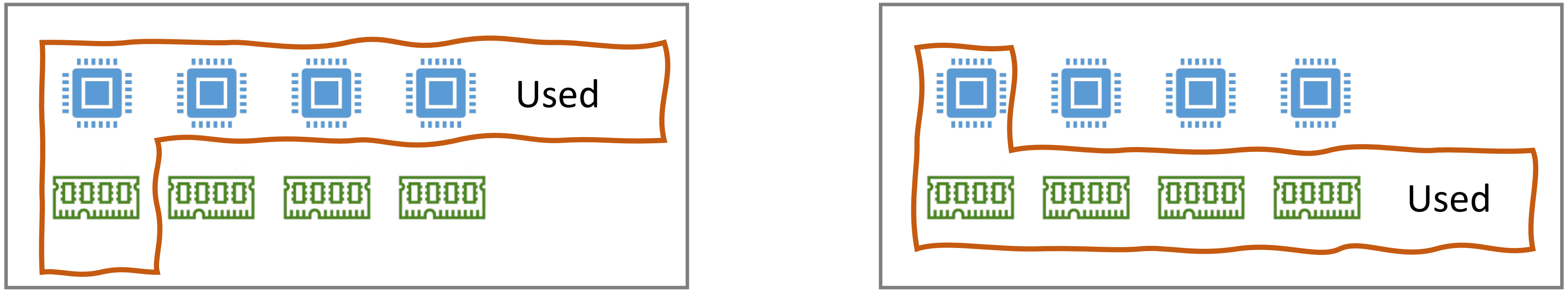
Memory Disaggregation (1)

Problem: low memory utilization in datacenters

- < 65% in Google, Alibaba, and Snowflake[1,2,3]

Root Cause: imbalanced memory usages across servers

- Some servers are CPU-bound, but some are memory-bound
- Cannot use memory beyond a local server



Server 1 (CPU-bound) Server 2 (memory-bound)

[1] Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces (IWQoS'19)
[2] Borg: the Next Generation (EuroSys'20)
[3] Memtrade: A Disaggregated-Memory Marketplace for Public Clouds (arXiv'21)

Memory Disaggregation (2)

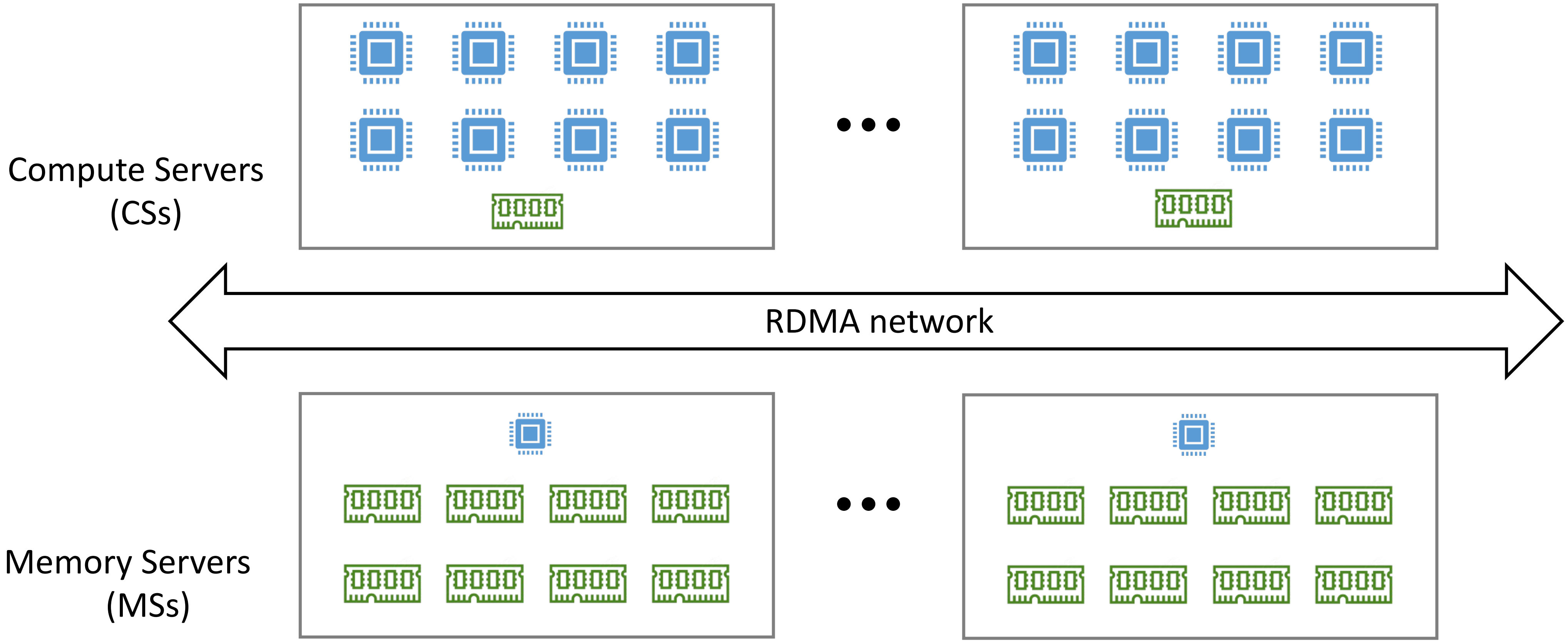
Memory Disaggregation

- ❖ Physically separate CPU and memory into network-attached components

Memory Disaggregation (2)

Memory Disaggregation

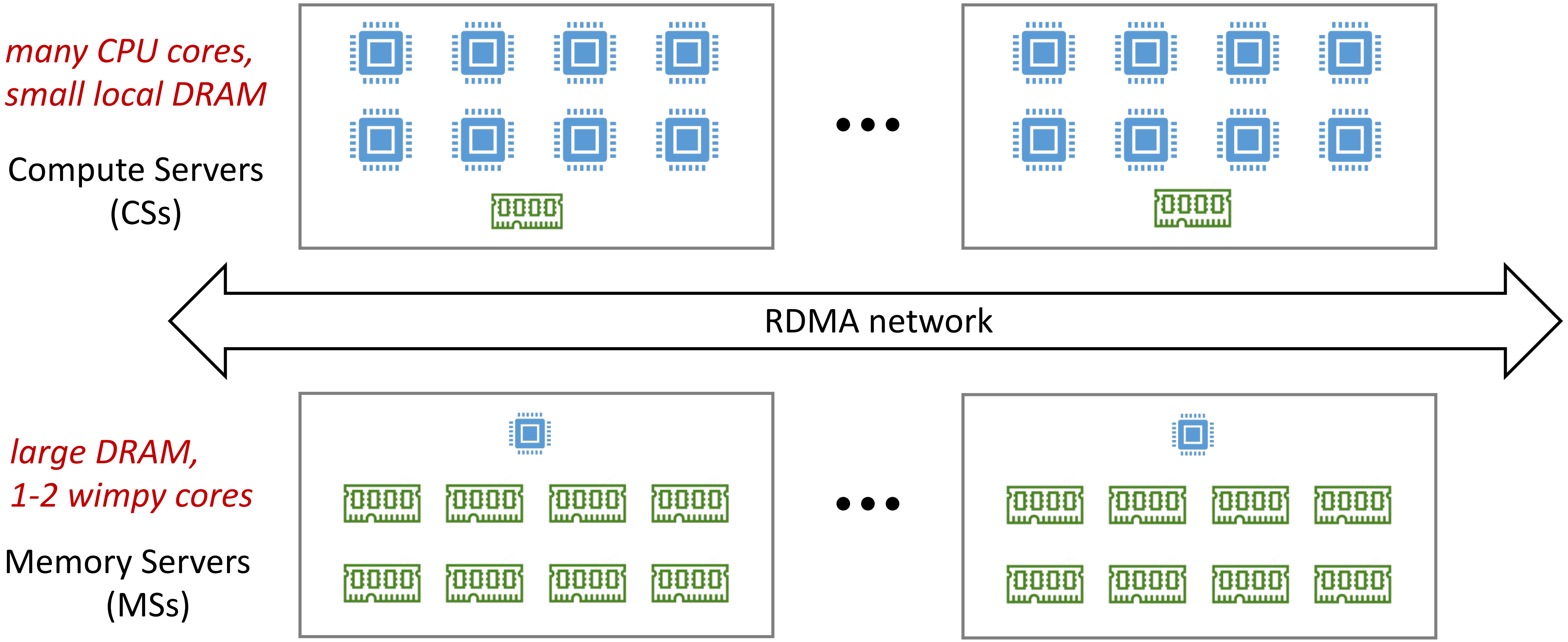
- ❖ Physically separate CPU and memory into network-attached components



Memory Disaggregation (2)

Memory Disaggregation

- ❖ Physically separate CPU and memory into network-attached components



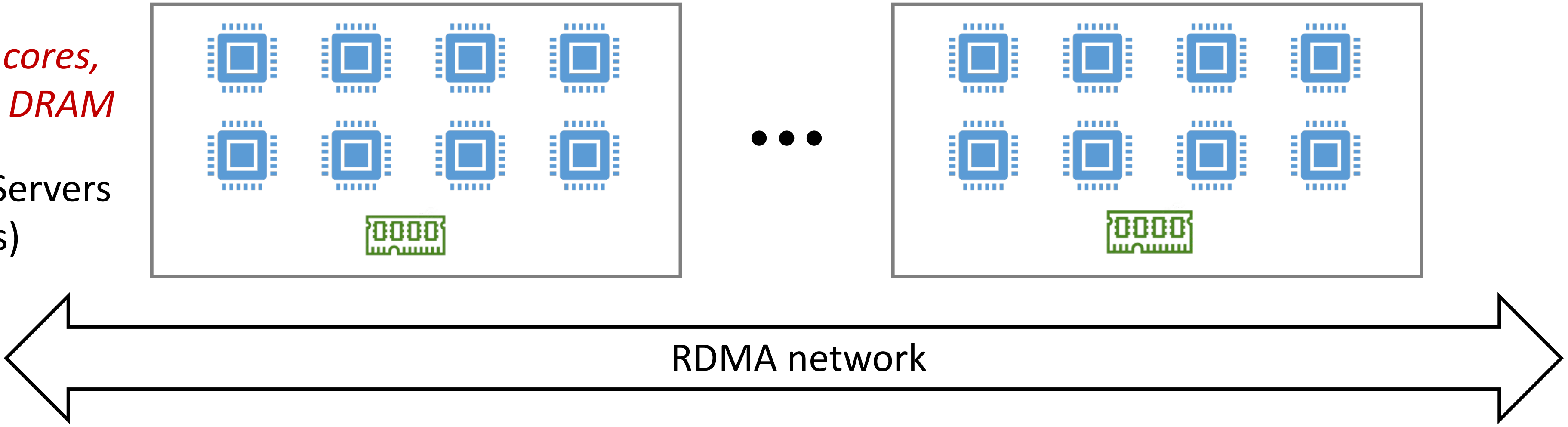
Memory Disaggregation (2)

Memory Disaggregation

- ❖ Physically separate CPU and memory into network-attached components

many CPU cores,
small local DRAM

Compute Servers
(CSs)



larg
1-2

Mem

Benefits:

- ✓ Independently scaling memory and CPU
- ✓ Flexibly assembling resources for apps
- ✓ Efficiently sharing memory between apps

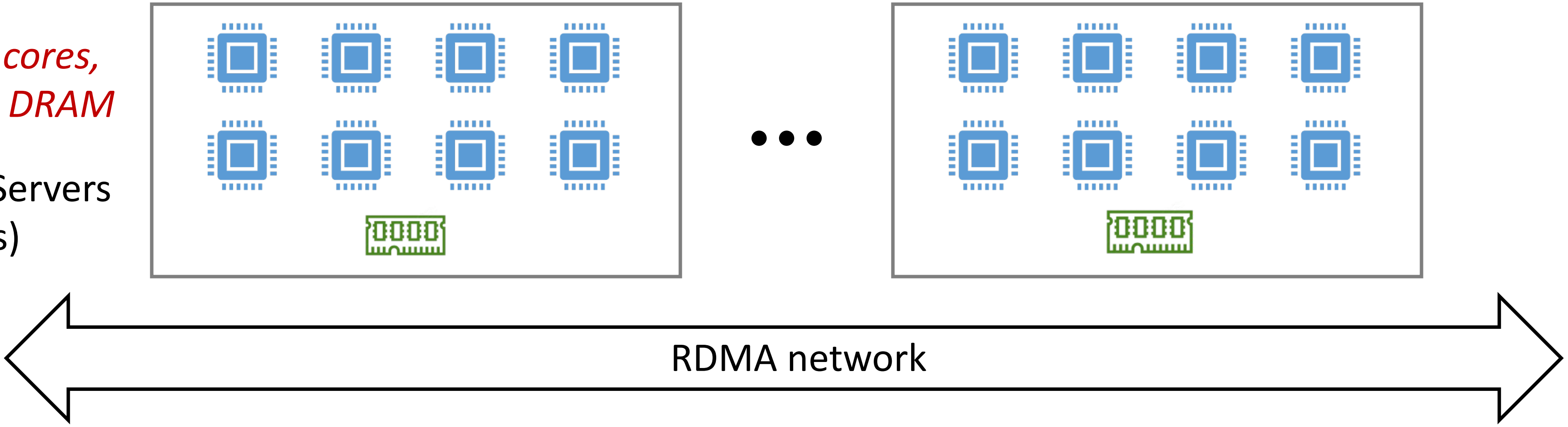
Memory Disaggregation (2)

Memory Disaggregation

❖ Physically separate CPU and memory into network-attached components

many CPU cores,
small local DRAM

Compute Servers
(CSs)



larg
1-2

Mem

Benefits:

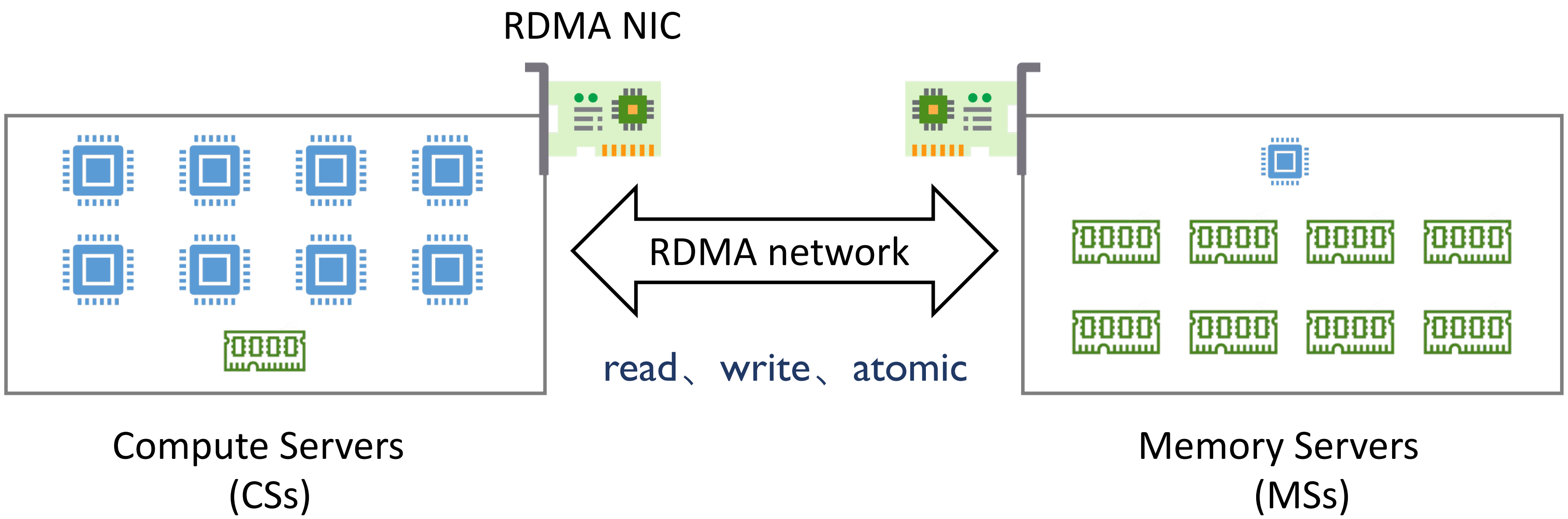
✓ Independently scaling memory and CPU
✓ Flexibly assembling resources for apps
✓ Efficiently sharing memory between apps

→ **high memory utilization**

Memory Disaggregation (3)

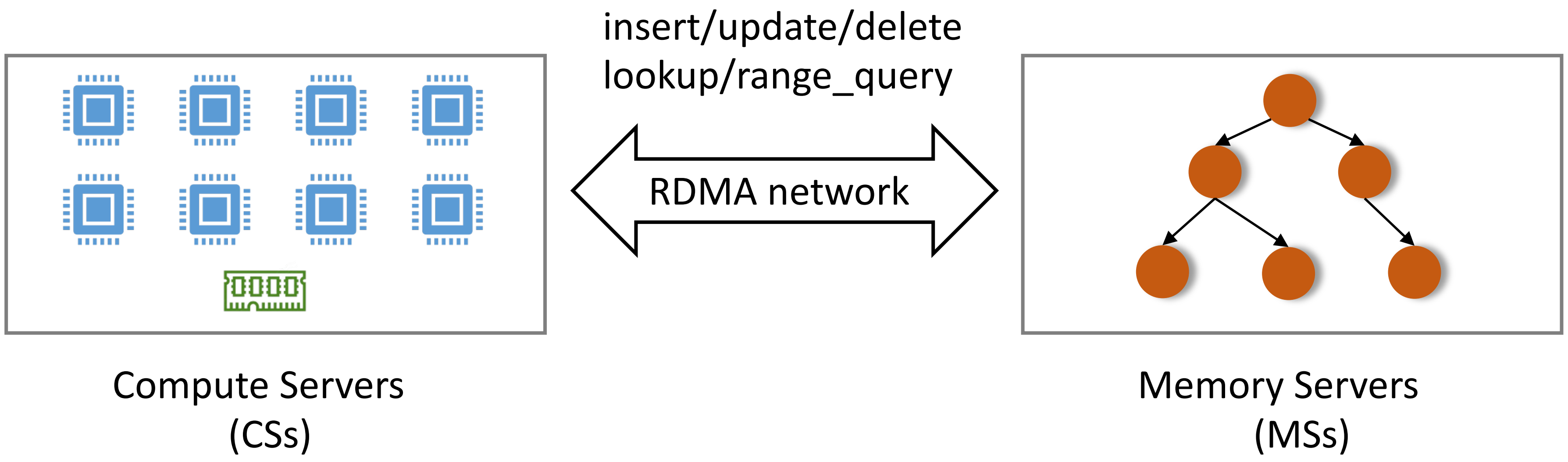
Key Enabler: Remote Direct Memory Access (RDMA)

- ❖ High bandwidth: 100/200/400Gbps
- ❖ Low latency: RTT < 2us
- ❖ Directly access remote memory: read、write、atomic (e.g., cas)



Tree Indexes on Disaggregated Memory (1)

In this work, we explore how to design a high-performance **tree index** on **disaggregated memory (DM)**



Tree Indexes on Disaggregated Memory (2)

Reexamine Existing RDMA-based Tree Indexes

1. Using RPC to handle index write operations (i.e., insert/update/delete)

   EXAMPLE Cell [ATC'16], FaRM-Tree [SIGMOD'19]

   Issue: Cannot be deployed on DM — near-zero computation power at memory-side

Tree Indexes on Disaggregated Memory (2)

Reexamine Existing RDMA-based Tree Indexes

1. Using RPC to handle index write operations (i.e., insert/update/delete)

   EXAMPLE Cell [ATC'16], FaRM-Tree [SIGMOD'19]

   Issue: Cannot be deployed on DM — near-zero computation power at memory-side

2. One-sided approach: leveraging RDMA read/write/atomic for all index ops

   EXAMPLE FG [SIGMOD'19]

   Issue: Low write performance

Tree Indexes on Disaggregated Memory (2)

Reexamine Existing RDMA-based Tree Indexes

1. Using RPC to handle index write operations (i.e., insert/update/delete)

   EXAMPLE Cell [ATC'16], FaRM-Tree [SIGMOD'19]

   Issue: Cannot be deployed on DM — near-zero computation power at memory-side

2. One-sided approach: leveraging RDMA read/write/atomic for all index ops

   EXAMPLE FG [SIGMOD'19]

   Issue: Low write performance

FG (Zipf 0.99)	Throughput (Mops)	50th Lat. (us)	99th Lat. (us)
5% Write	31.8	4.9	14.9
50% Write	0.34	10	19890

Tree Indexes on Disaggregated Memory (2)

Reexamine Existing RDMA-based Tree Indexes

1. Using RPC to handle index write operations (i.e., insert/update/delete)

   EXAMPLE Cell [ATC'16], FaRM-Tree [SIGMOD'19]

   Issue: Cannot be deployed on DM — near-zero computation power at memory-side

2. One-sided approach: leveraging RDMA read/write/atomic for all index ops

   EXAMPLE FG [SIGMOD'19]

   Issue: Low write performance

FG (Zipf 0.99)	Throughput (Mops)	50th Lat. (us)	99th Lat. (us)
5% Write	31.8	4.9	14.9
50% Write	0.34	10	19890

Low throughput & High latency
w/ 8 MSs and 8 CSs

Tree Indexes on Disaggregated Memory (2)

Reexamine Existing RDMA-based Tree Indexes

1. Using RPC to handle index write operations (i.e., insert/update/delete)

   EXAMPLE Cell [ATC'16], FaRM-Tree [SIGMOD'19]

   Issue: Cannot be deployed on DM — near-zero computation power at memory-side

2. One-sided approach: leveraging RDMA read/write/atomic for all index ops

   EXAMPLE FG [SIGMOD'19]

   Issue: Low write performance

FG (Zipf 0.99)	Throughput (Mops)	50th Lat. (us)	99th Lat. (us)
5% Write	31.8	4.9	14.9
50% Write	0.34	10	19890

Low throughput & High latency w/ 8 MSs and 8 CSs

3. Hardware modification or SmartNICs for offloading index ops

   EXAMPLE HT-Tree [HotOS'19]

   Issue: High TCO (total cost of ownership)

Our Goal

Our Goal: building a tree index on disaggregated memory
that can deliver high performance (for both read/write ops)
with commodity RDMA NICs

Why One-sided Approach is Slow ? (1)

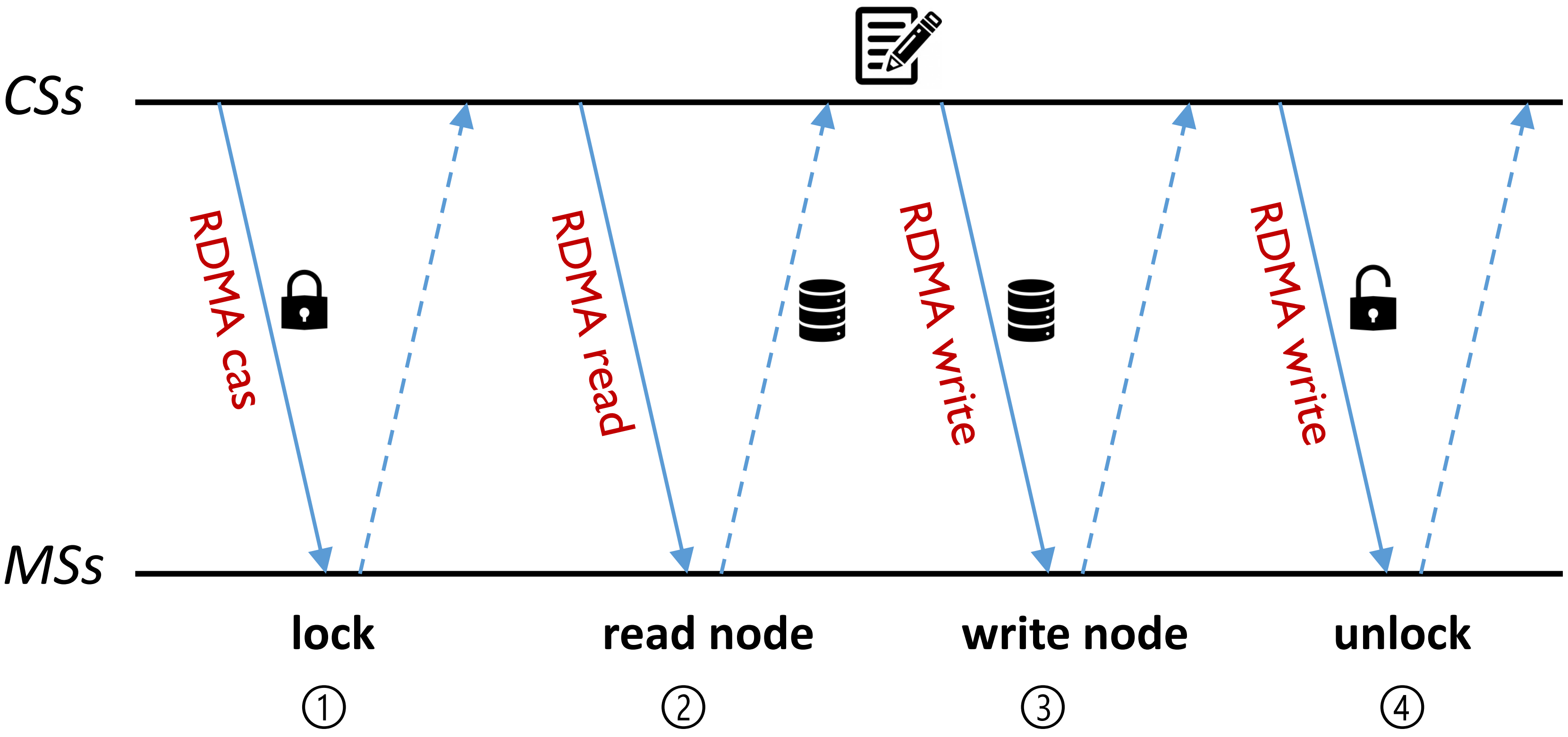
(1) Excessive Round Trips

cas = compare and swap ; faa = fetch and add

Why One-sided Approach is Slow ? (1)

(1) Excessive Round Trips

Four round trips when modifying a tree node (FG [SIGMOD'19])

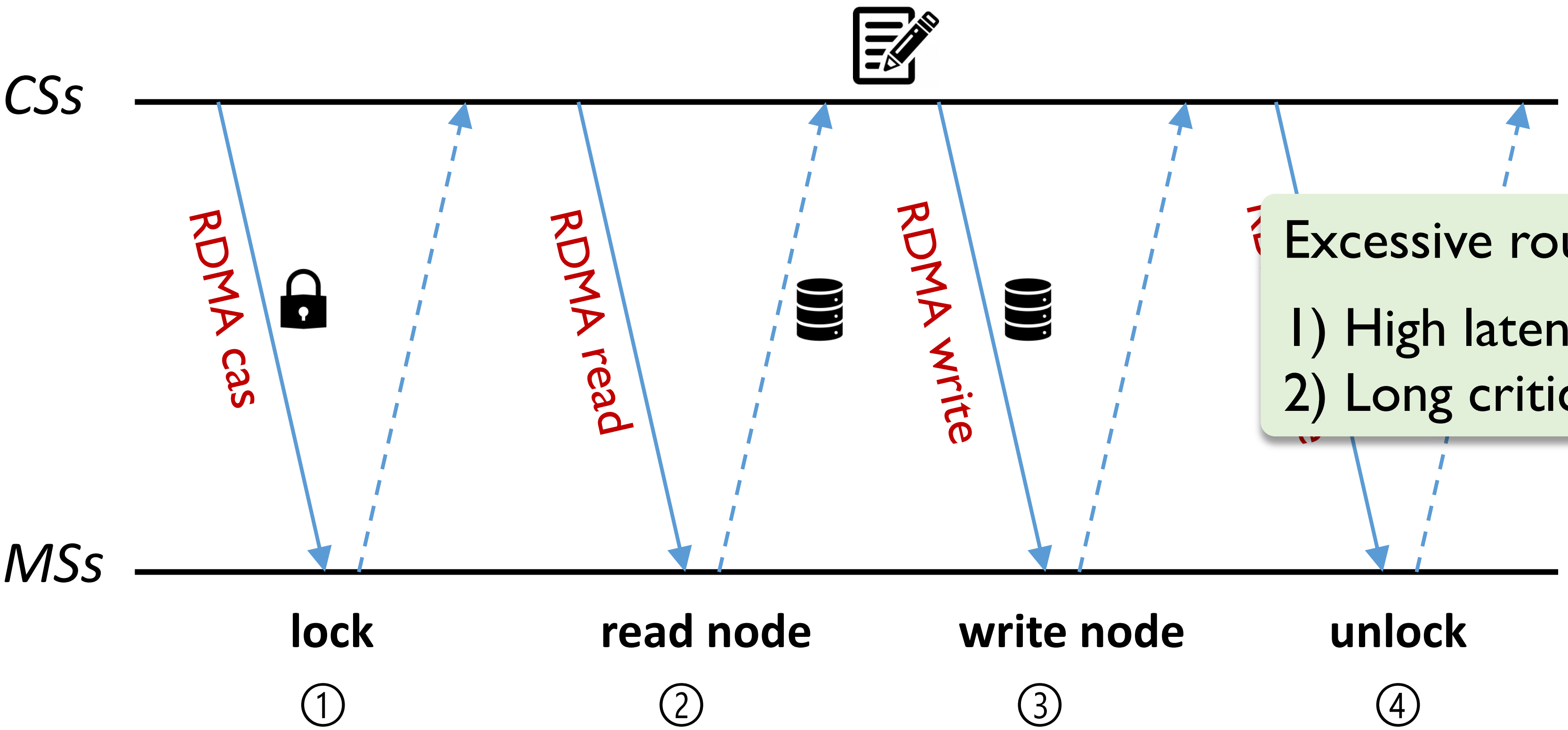


cas = compare and swap ; faa = fetch and add

Why One-sided Approach is Slow ? (1)

(1) Excessive Round Trips

Four round trips when modifying a tree node (FG [SIGMOD'19])



Excessive round trips harm performance:

1) High latency of single write op
2) Long critical path, blocking conflicting ops

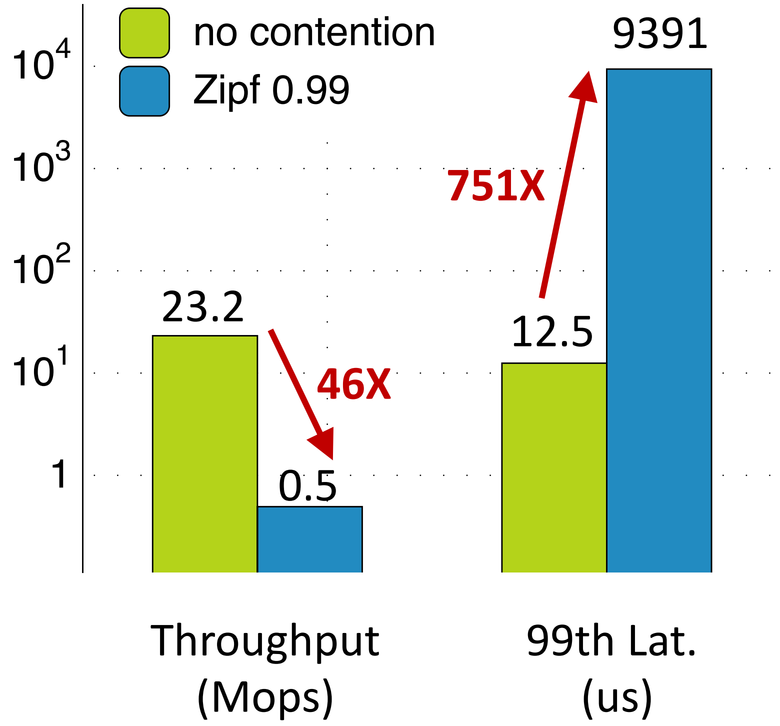
cas = compare and swap ; faa = fetch and add

Why One-sided Approach is Slow ? (2)

(2) Slow Synchronization Primitives — RDMA lock

Why One-sided Approach is Slow ? (2)

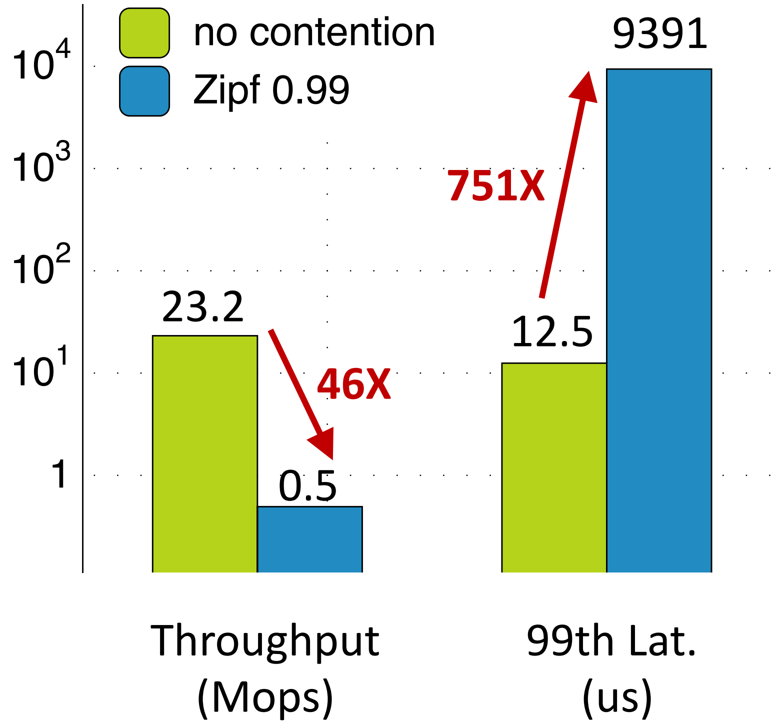
(2) Slow Synchronization Primitives — RDMA lock



154 threads acquire/release 10240 locks in an MS,
RDMA cas for lock acquisition and faa for release (FG)

Why One-sided Approach is Slow ? (2)

(2) Slow Synchronization Primitives — RDMA lock

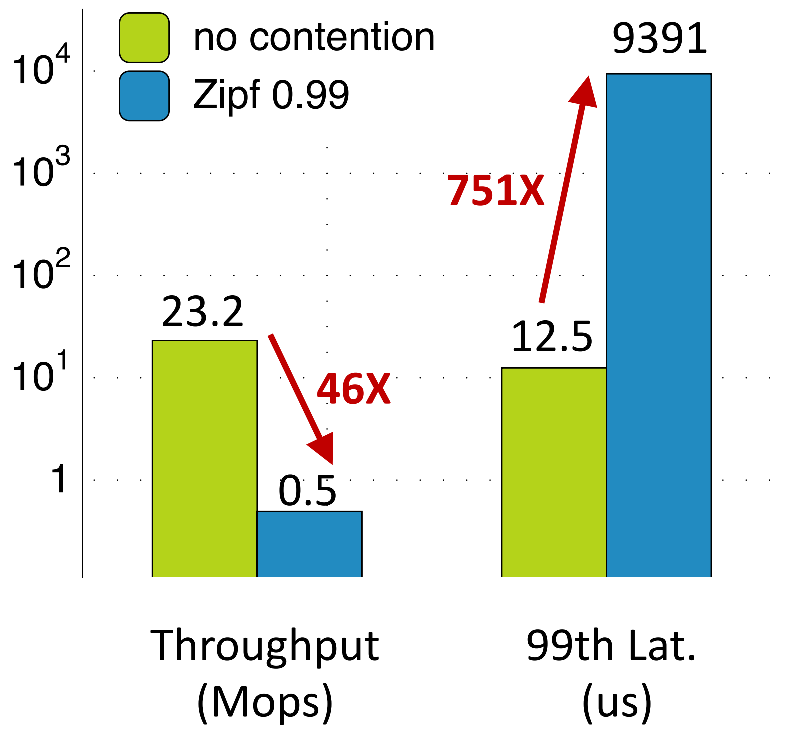


154 threads acquire/release 10240 locks in an MS,
RDM e (FG)

Performance of lock collapses when contention appears

Why One-sided Approach is Slow ? (2)

(2) Slow Synchronization Primitives — RDMA lock

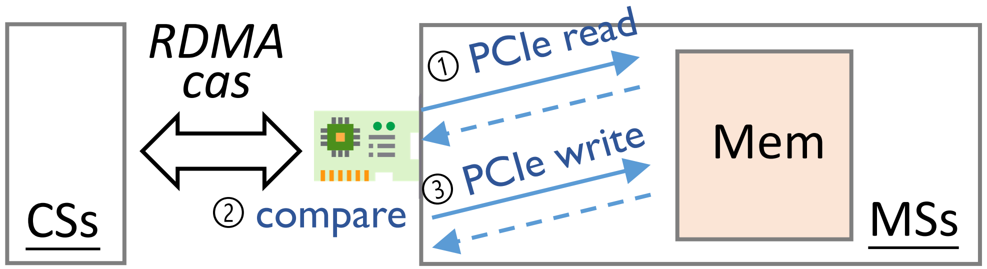


154 threads acquire/release 10240 locks in an MS,
RDM ... e (FG)

Performance of lock collapses when contention appears

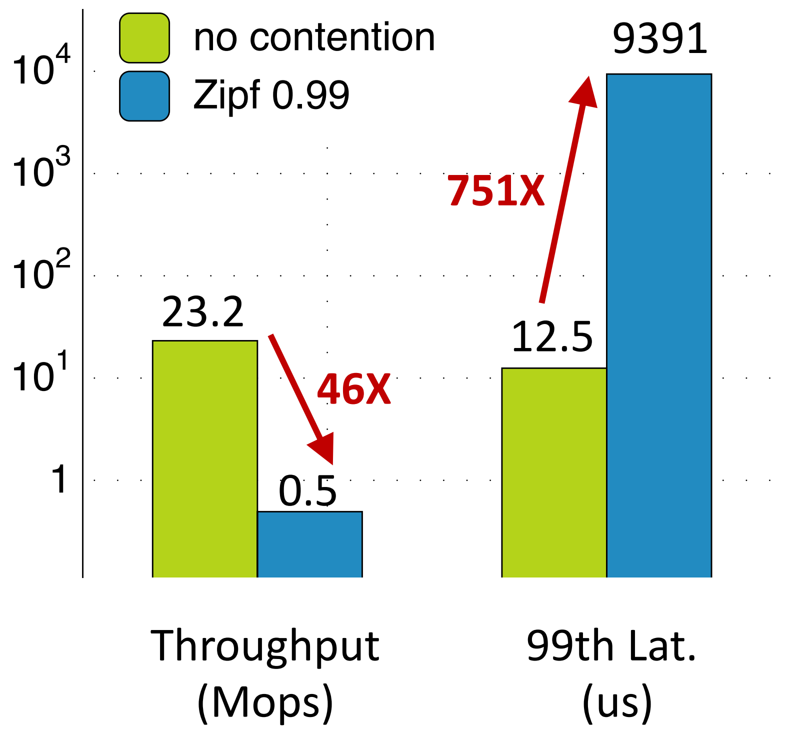
1. Expensive in-NIC concurrency control

- NICs serialize atomic verbs w/ 2-PCIe-txn critical path



Why One-sided Approach is Slow ? (2)

(2) Slow Synchronization Primitives — RDMA lock

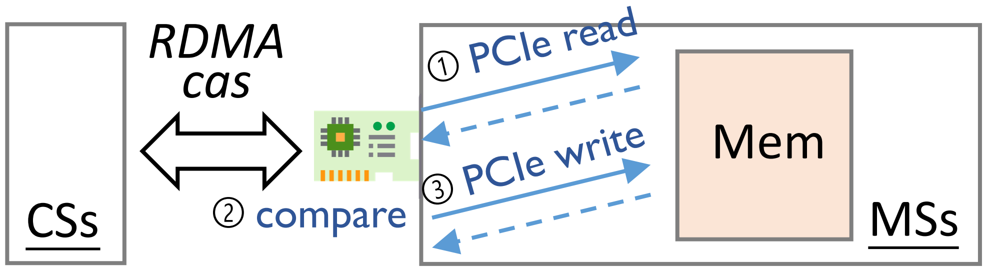


154 threads acquire/release 10240 locks in an MS,
RDM e (FG)

Performance of lock collapses when contention appears

1. Expensive in-NIC concurrency control

- NICs serialize atomic verbs w/ 2-PCIe-txn critical path

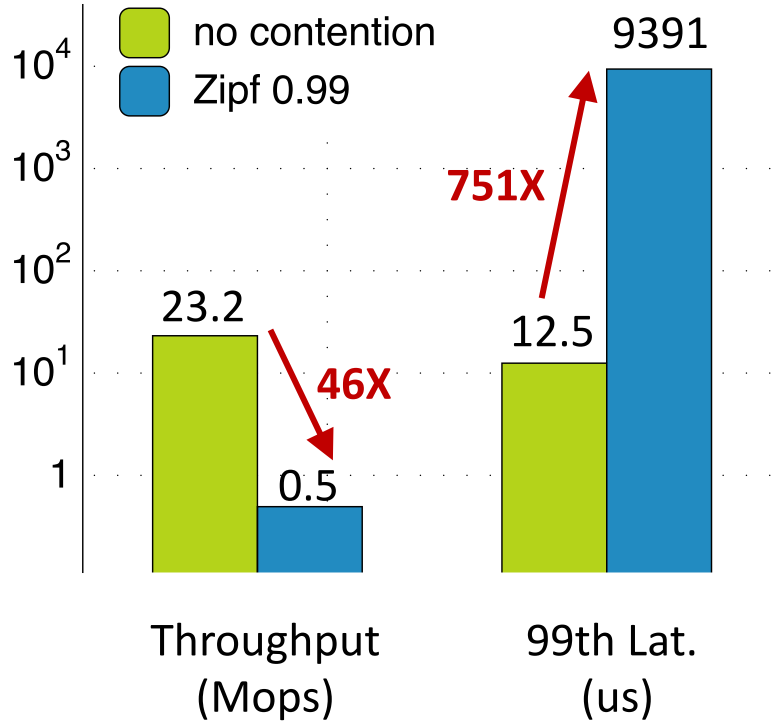


2. Unnecessary retries

- Lock retries consume limited RDMA throughput

Why One-sided Approach is Slow ? (2)

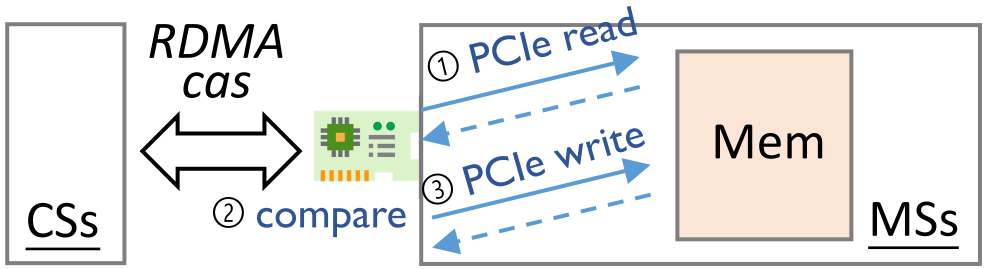
(2) Slow Synchronization Primitives — RDMA lock



154 threads acquire/release 10240 locks in an MS,
RDM … e (FG)

Performance of lock collapses when contention appears

1. Expensive in-NIC concurrency control
- NICs serialize atomic verbs w/ 2-PCIe-txn critical path



2. Unnecessary retries
- Lock retries consume limited RDMA throughput

3. Lacking Fairness
- Do not consider fairness, starving some clients and further inducing high tail latency

Why One-sided Approach is Slow ? (3)

(3) Write Amplification

Why One-sided Approach is Slow ? (3)

(3) Write Amplification

Lots of indexes use lock-free lookup to eliminate read locks:

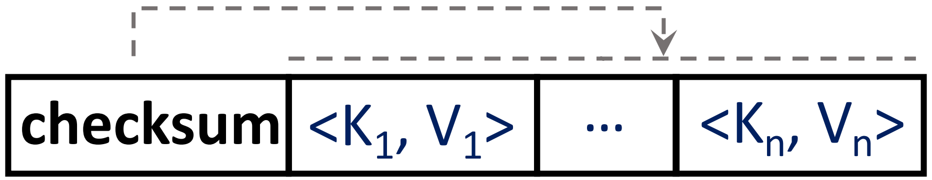
- Issue RDMA read to fetch tree node
- Detect inconsistent data due to concurrent writes via checksum or versions
- Retry if data is inconsistent

Why One-sided Approach is Slow ? (3)

(3) Write Amplification

Lots of indexes use lock-free lookup to eliminate read locks:

- Issue RDMA read to fetch tree node
- Detect inconsistent data due to concurrent writes via checksum or versions
- Retry if data is inconsistent



Checksum-based

Writer: modify entries, checksum = *crc*(node)
Reader: if checksum == *crc*(node) ?

Why One-sided Approach is Slow ? (3)

(3) Write Amplification

Lots of indexes use lock-free lookup to eliminate read locks:
- Issue RDMA read to fetch tree node
- Detect inconsistent data due to concurrent writes via checksum or versions
- Retry if data is inconsistent

checksum	$<K_1, V_1>$	…	$<K_n, V_n>$

Checksum-based

ver_a	$<K_1, V_1>$	…	$<K_n, V_n>$	**ver_b**

Version-based

Writer: modify entries, checksum = *crc*(node)
Reader: if checksum == *crc*(node) ?

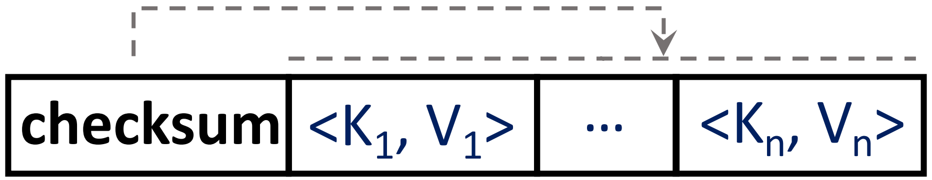
Writer: ver_a++, modify entries, ver_b++
Reader: if ver_a == ver_b ?

Why One-sided Approach is Slow ? (3)

(3) Write Amplification

Lots of indexes use lock-free lookup to eliminate read locks:

- Issue RDMA read to fetch tree node
- Detect inconsistent data due to concurrent writes via checksum or versions
- Retry if data is inconsistent



Checksum-based

Version-based

In these two mechanisms, writers must write back the whole tree node, even when modifying an individual KV entry, inducing write amplification

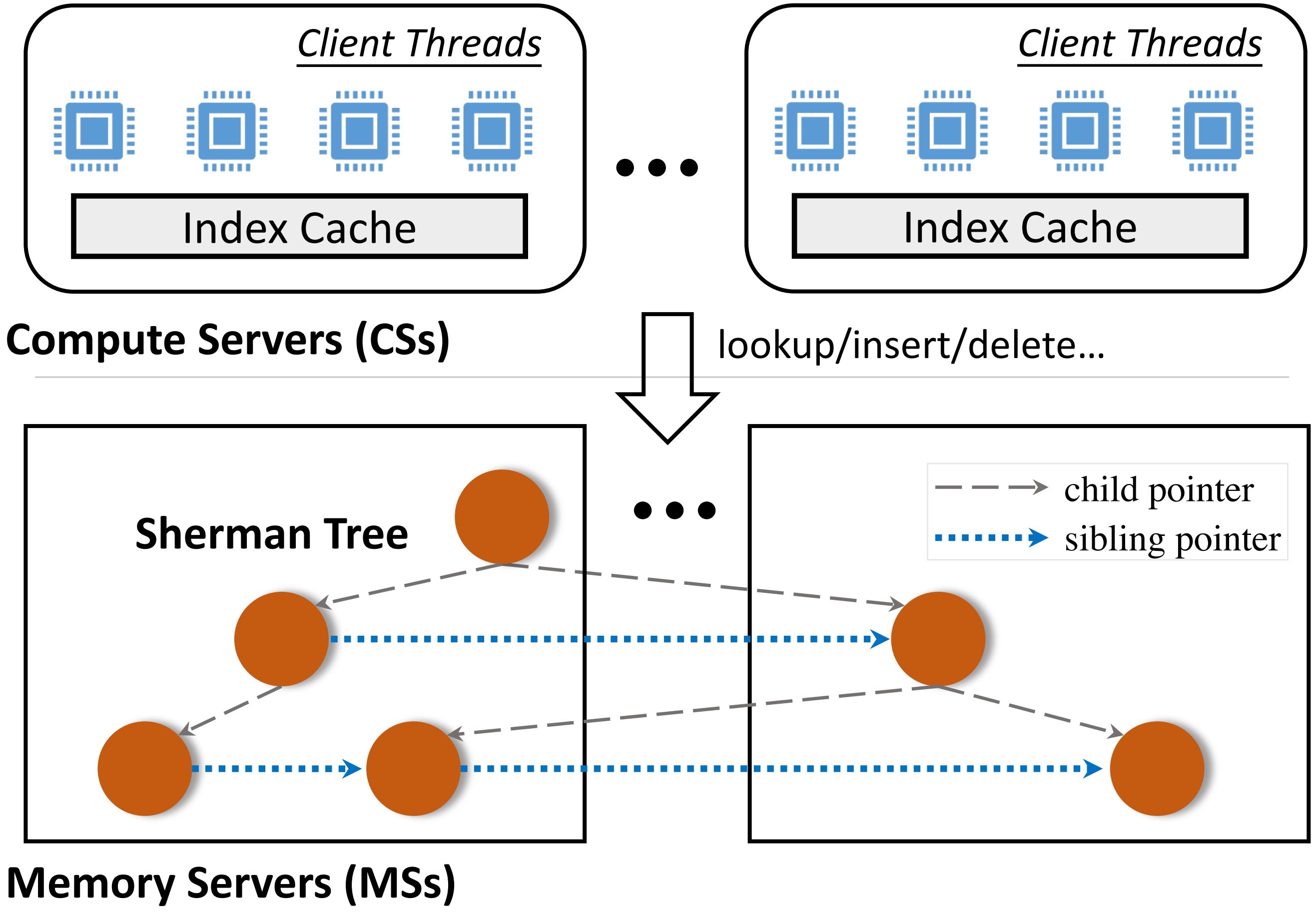
Outline

- Background & Motivation
- Sherman – A Write-Optimized B+Tree on Disaggregated Memory
- Evaluation
- Summary

Sherman Overview

Sherman is a B+Tree index on disaggregated memory

- B-link tree structure (sibling pointer)
- Tree nodes are across many MSs
- One-sided RDMA for *all* index ops
- Index cache at CSs
  - caching internal tree nodes
  - reducing remote accesses
- Concurrency control
  - write-write conflicts:
    - node-grained exclusive locks
  - read-write conflicts:
    - lock-free search w/ versions



Key Idea

Combining RDMA hardware features with

RDMA-friendly software techniques

Key Idea

Combining RDMA hardware features with
RDMA-friendly software techniques

Reducing round trips → Command combination

Key Idea

Combining RDMA hardware features with RDMA-friendly software techniques

Reducing round trips	➡	Command combination
Accelerating concurrent accesses	➡	Hierarchical on-chip lock

Key Idea

Combining RDMA hardware features with
RDMA-friendly software techniques

Reducing round trips	→	Command combination
Accelerating concurrent accesses	→	Hierarchical on-chip lock
Mitigating write amplification	→	Two-level version layout

Command combination

Observation: RDMA write commands are executed in order at receivers

Command combination

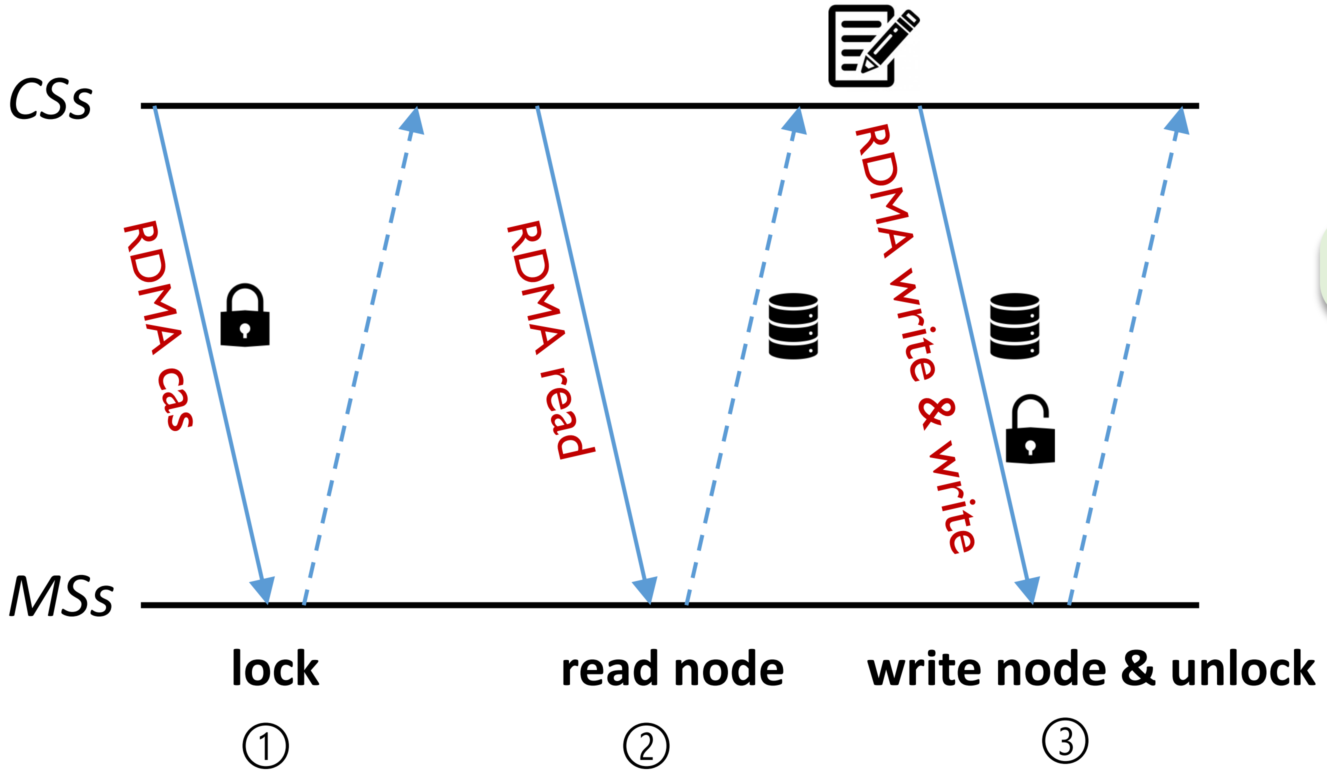
Observation: RDMA write commands are executed in order at receivers

Command combination: client threads issue dependent RDMA writes simultaneously

Command combination

Observation: RDMA write commands are executed in order at receivers

Command combination: client threads issue dependent RDMA writes simultaneously

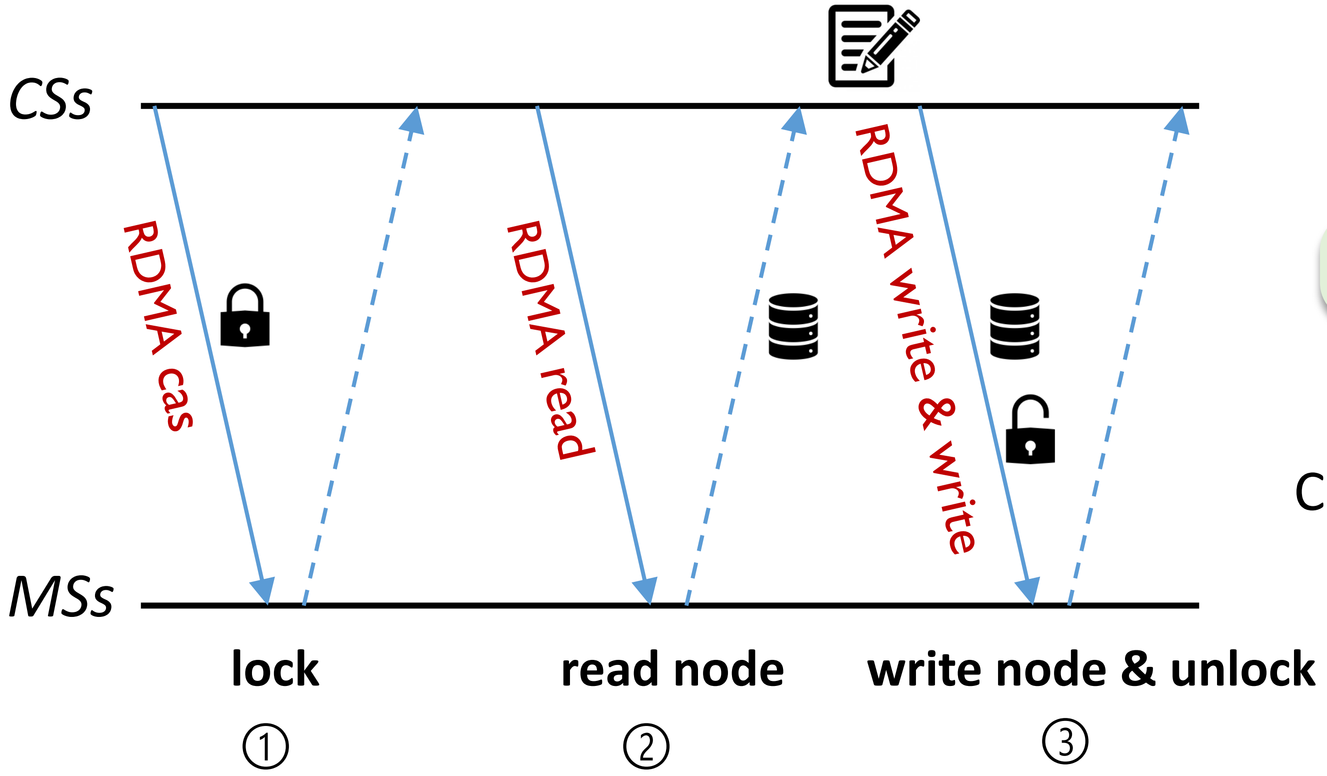


Combine write-back and lock release

Command combination

Observation: RDMA write commands are executed in order at receivers

Command combination: client threads issue dependent RDMA writes simultaneously



Combine write-back and lock release

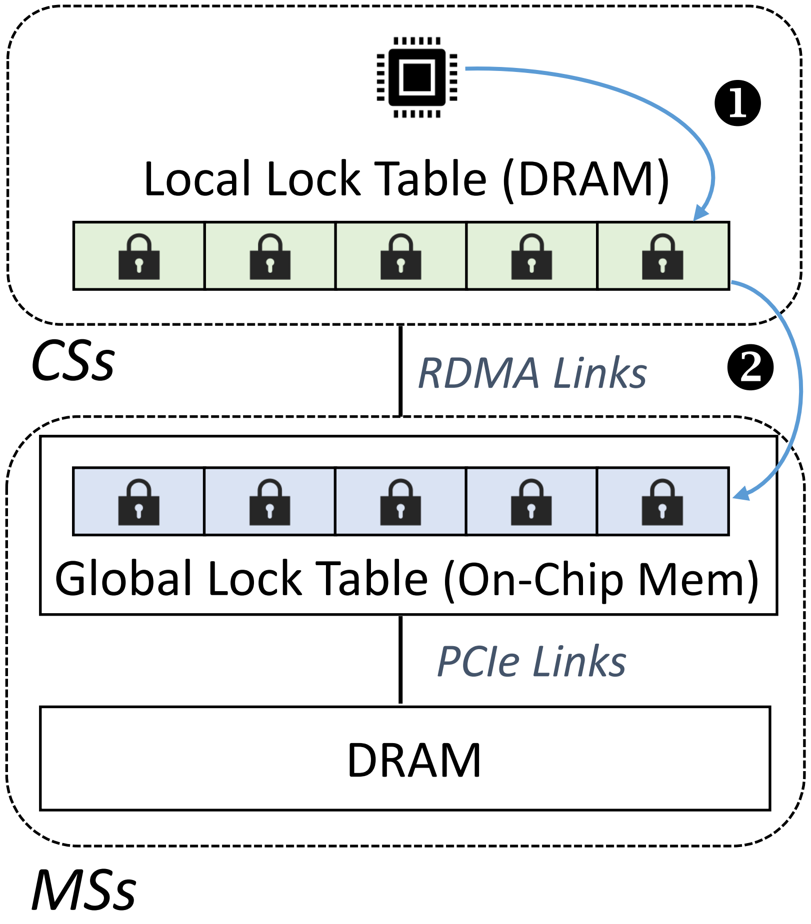
Checkout paper for other cases of combination

Hierarchical On-Chip Lock

Observation: RDMA NICs can expose on-chip memory (SRAM) for usages

Hierarchical On-Chip Lock

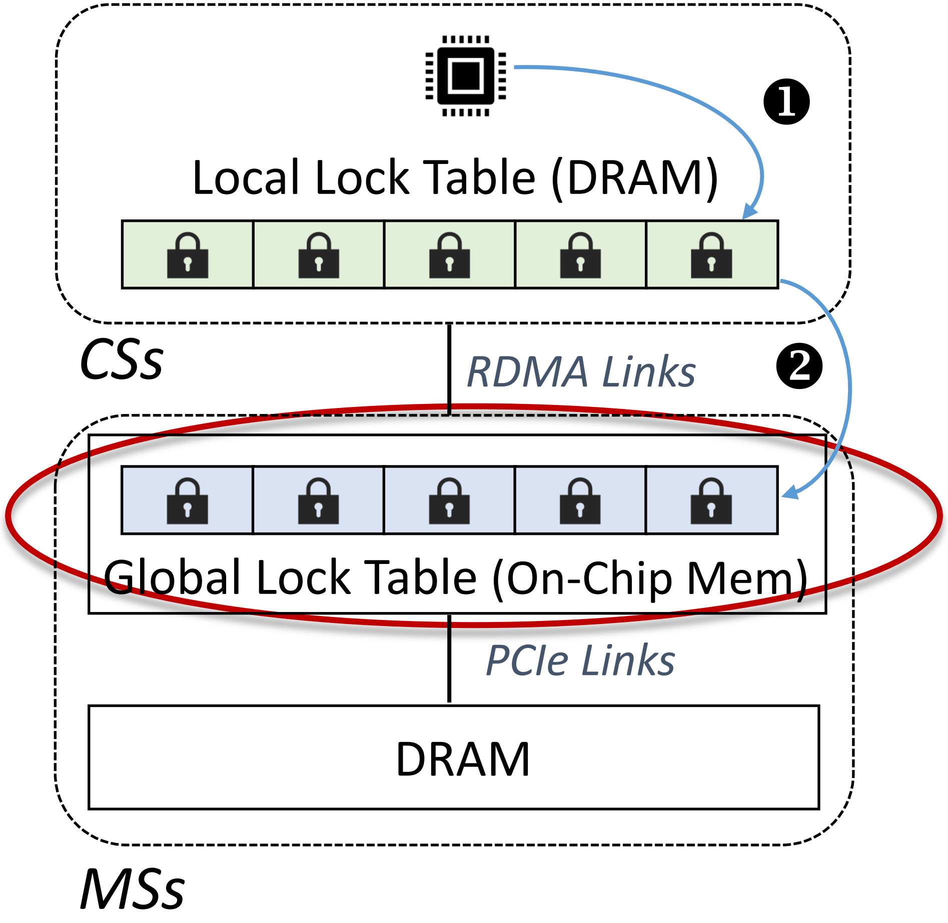
Observation: RDMA NICs can expose on-chip memory (SRAM) for usages



Hierarchical On-Chip Lock

Observation: RDMA NICs can expose on-chip memory (SRAM) for usages

❖ Store locks in on-chip mem of MSs' NICs
- an array called Global Lock Table (GLT)
- hash [addr of tree node] => position in GLT
- eliminate PCIe txn at MSs

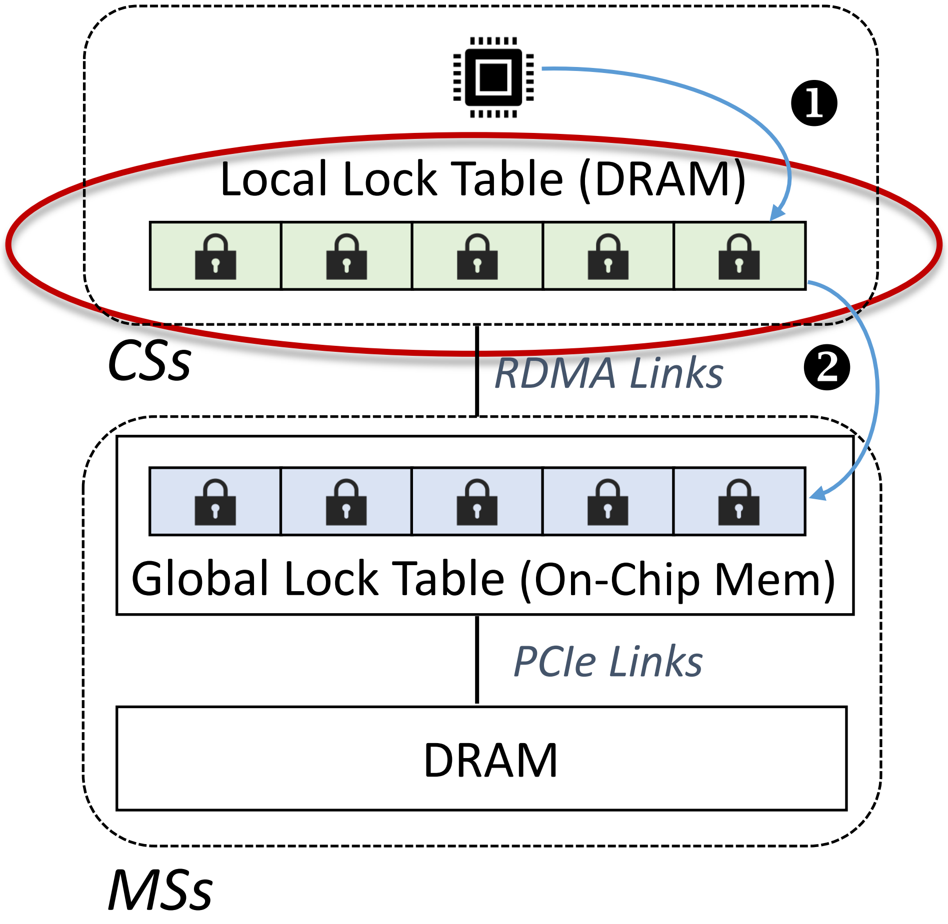


Hierarchical On-Chip Lock

Observation: RDMA NICs can expose on-chip memory (SRAM) for usages

- ❖ Store locks in on-chip mem of MSs' NICs
  - an array called Global Lock Table (GLT)
  - hash [addr of tree node] => position in GLT
  - eliminate PCIe txn at MSs

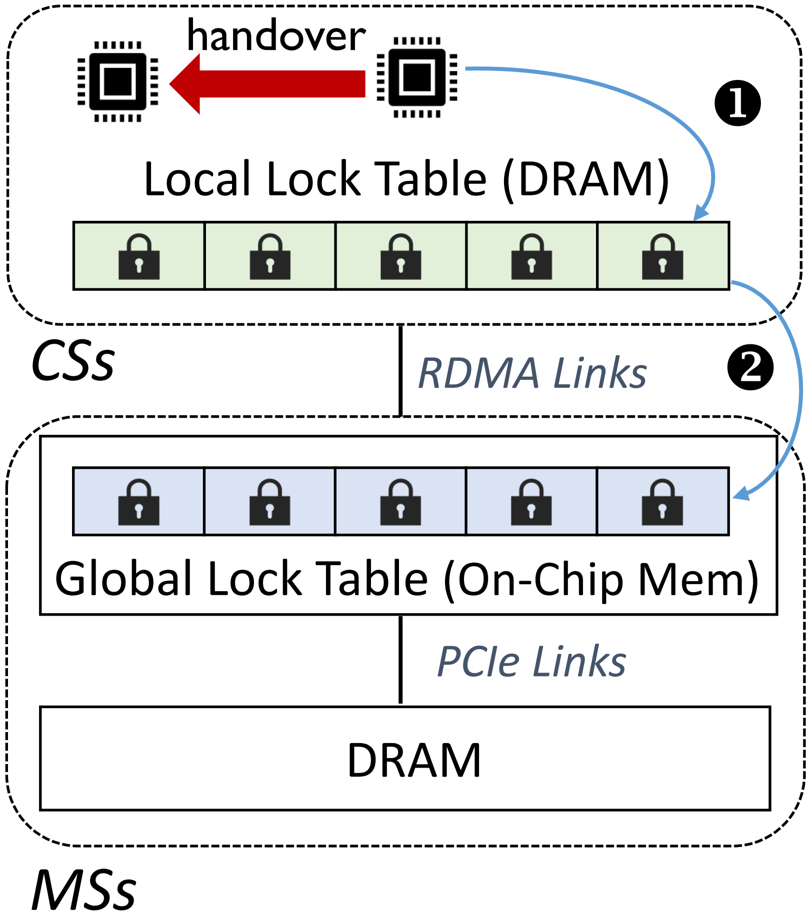
- ❖ Hierarchical structure
  - Maintain a mirror of GLT at each CS: Local Lock Table
  - first get local lock, then global one
  - avoid unnecessary across-network retries
  - bind a wait queue to each local lock, boosting fairness



Hierarchical On-Chip Lock

Observation: RDMA NICs can expose on-chip memory (SRAM) for usages

- ❖ Store locks in on-chip mem of MSs' NICs
  - an array called Global Lock Table (GLT)
  - hash [addr of tree node] => position in GLT
  - eliminate PCIe txn at MSs
- ❖ Hierarchical structure
  - Maintain a mirror of GLT at each CS: Local Lock Table
  - first get local lock, then global one
  - avoid unnecessary across-network retries
  - bind a wait queue to each local lock, boosting fairness
- ❖ Handover mechanism
  - Hand over a lock from one thread to another locally
  - reduce one round trip



Two-level version layout

Sherman tailors the B+Tree layout to mitigate write amplification

Two-level version layout

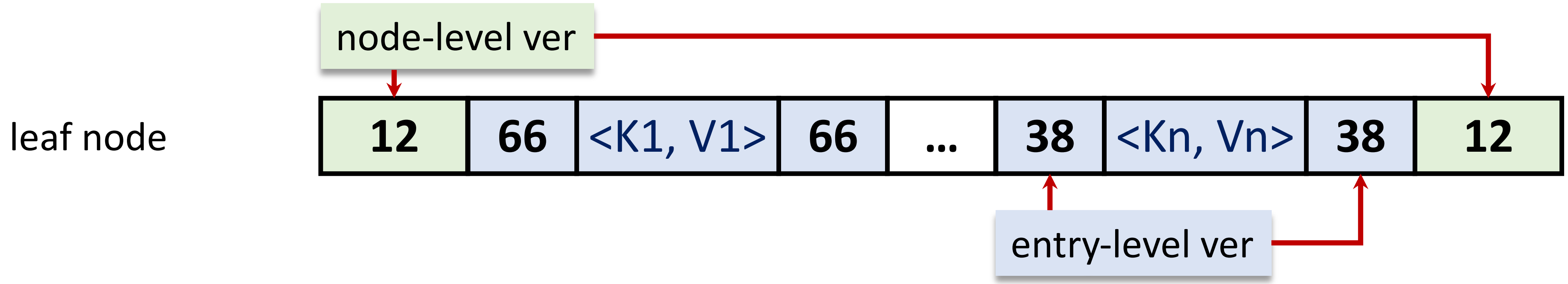
Sherman tailors the B+Tree layout to mitigate write amplification

- ❖ Make entries in leaf nodes unsorted
  - avoid shift operation on insert/delete

Two-level version layout

Sherman tailors the B+Tree layout to mitigate write amplification

- ❖ Make entries in leaf nodes unsorted
  - avoid shift operation on insert/delete
- ❖ Two-level version in leaf nodes
  - ❖ node-level version protects leaf nodes => increment when insert/update/delete KV
  - ❖ entry-level version protects KV entries => increment when nodes split/merge



Two-level version layout

Sherman tailors the B+Tree layout to mitigate write amplification

- Make entries in leaf nodes unsorted
  - avoid shift operation on insert/delete
- Two-level version in leaf nodes
  - node-level version protects leaf nodes => increment when insert/update/delete KV
  - entry-level version protects KV entries => increment when nodes split/merge

leaf node

12	**66**	<K1, V1>	**66**	**…**	**38**	<Kn, Vn>	**38**	**12**

write back entry (not node) via RDMA write

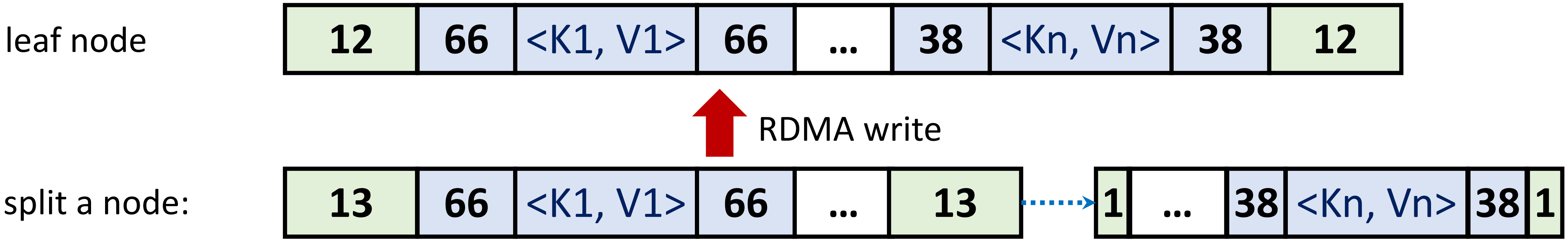
update a KV:

67	<K1, V1’>	**67**

Two-level version layout

Sherman tailors the B+Tree layout to mitigate write amplification

- ❖ Make entries in leaf nodes unsorted
  - avoid shift operation on insert/delete
- ❖ Two-level version in leaf nodes
  - ❖ node-level version protects leaf nodes => increment when insert/update/delete KV
  - ❖ entry-level version protects KV entries => increment when nodes split/merge



Two-level version layout

Sherman tailors the B+Tree layout to mitigate write amplification

- Make entries in leaf nodes unsorted
  - avoid shift operation on insert/delete
- Two-level version in leaf nodes
  - node-level version protects leaf nodes => increment when insert/update/delete KV
  - entry-level version protects KV entries => increment when nodes split/merge

leaf node	**12**	**66**	<K1, V1>	**66**	**...**	**38**	<Kn, Vn>	**38**	**12**

RDMA read

read a node:	**12**	**66**	<K1, V1>	**66**	**...**	**38**	<Kn, Vn>	**38**	**12**

Whether two node-level vers are equal ? two entry-level vers are equal ? If no, retry

Outline

- ❖ Background & Motivation
- ❖ Sherman – A Write-Optimized B+Tree on Disaggregated Memory
- ❖ Evaluation
- ❖ Summary

Experimental Setup

Hardware Platform

Machine * 8

CPU	2 Intel Xeon E5-2650 (12 core)
Mem	128GB DRAM
NIC	100Gbps Mellanox ConnectX-5 w/ 256KB on-chip memory
OS	CentOS 7.7 , Linux kernel 3.10.0

Experimental Setup

Hardware Platform

Machine * 8

CPU	2 Intel Xeon E5-2650 (12 core)
Mem	128GB DRAM
NIC	100Gbps Mellanox ConnectX-5 w/ 256KB on-chip memory
OS	CentOS 7.7 , Linux kernel 3.10.0

⇨

We emulate each machine as one MS and one CS

- MS: 64GB DRAM and 2 CPU cores
- CS: 1GB DRAM and 22 CPU cores

Experimental Setup

Hardware Platform

Machine * 8

CPU	2 Intel Xeon E5-2650 (12 core)
Mem	128GB DRAM
NIC	100Gbps Mellanox ConnectX-5 w/ 256KB on-chip memory
OS	CentOS 7.7 , Linux kernel 3.10.0

⇨ We emulate each machine as one MS and one CS

- MS: 64GB DRAM and 2 CPU cores
- CS: 1GB DRAM and 22 CPU cores

Compared System: FG [SIGMOD'19]

- One-sided RDMA for all index ops, so it can be deployed on DM
- RDMA locks for write-write conflicts; checksum for read-write conflicts
- We add CS-side index cache for FG, for fair comparison

Experimental Setup

Hardware Platform

Machine * 8

CPU	2 Intel Xeon E5-2650 (12 core)
Mem	128GB DRAM
NIC	100Gbps Mellanox ConnectX-5 w/ 256KB on-chip memory
OS	CentOS 7.7 , Linux kernel 3.10.0

⇨ We emulate each machine as one MS and one CS

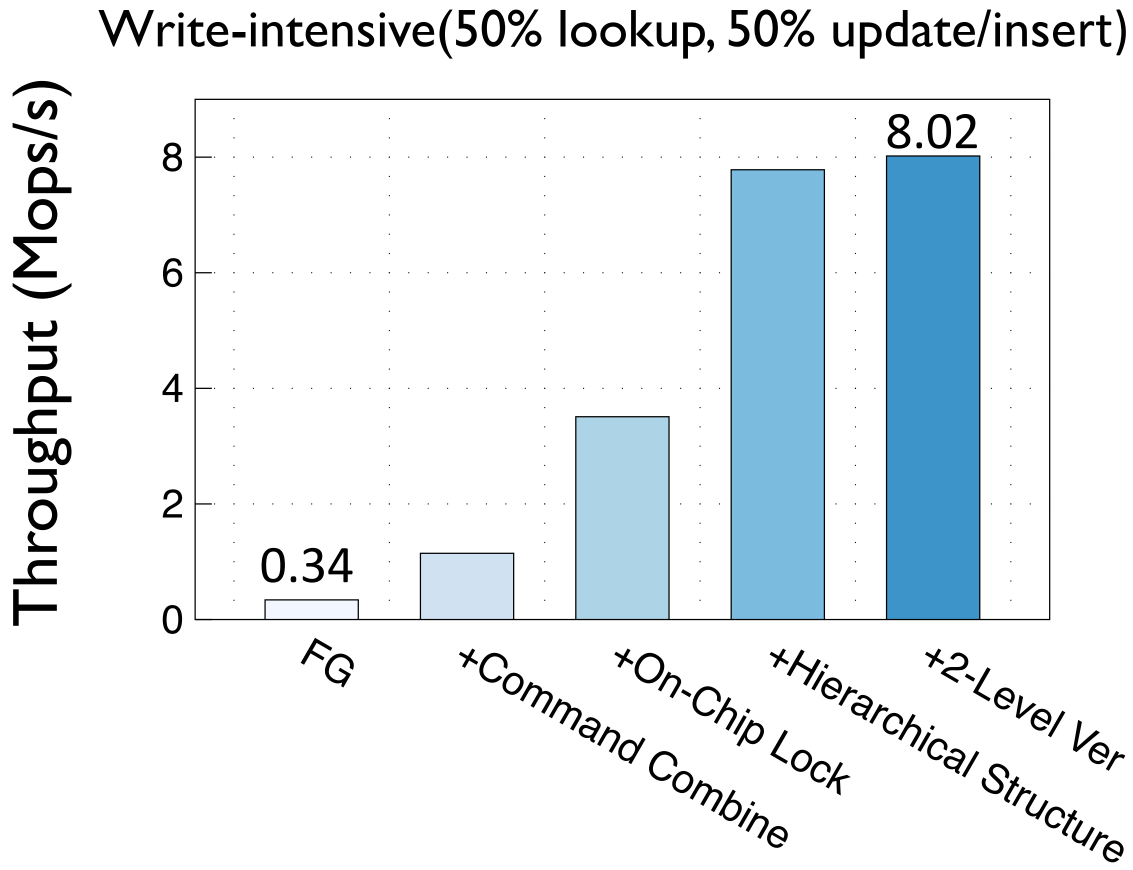
- MS: 64GB DRAM and 2 CPU cores
- CS: 1GB DRAM and 22 CPU cores

Compared System: FG [SIGMOD'19]

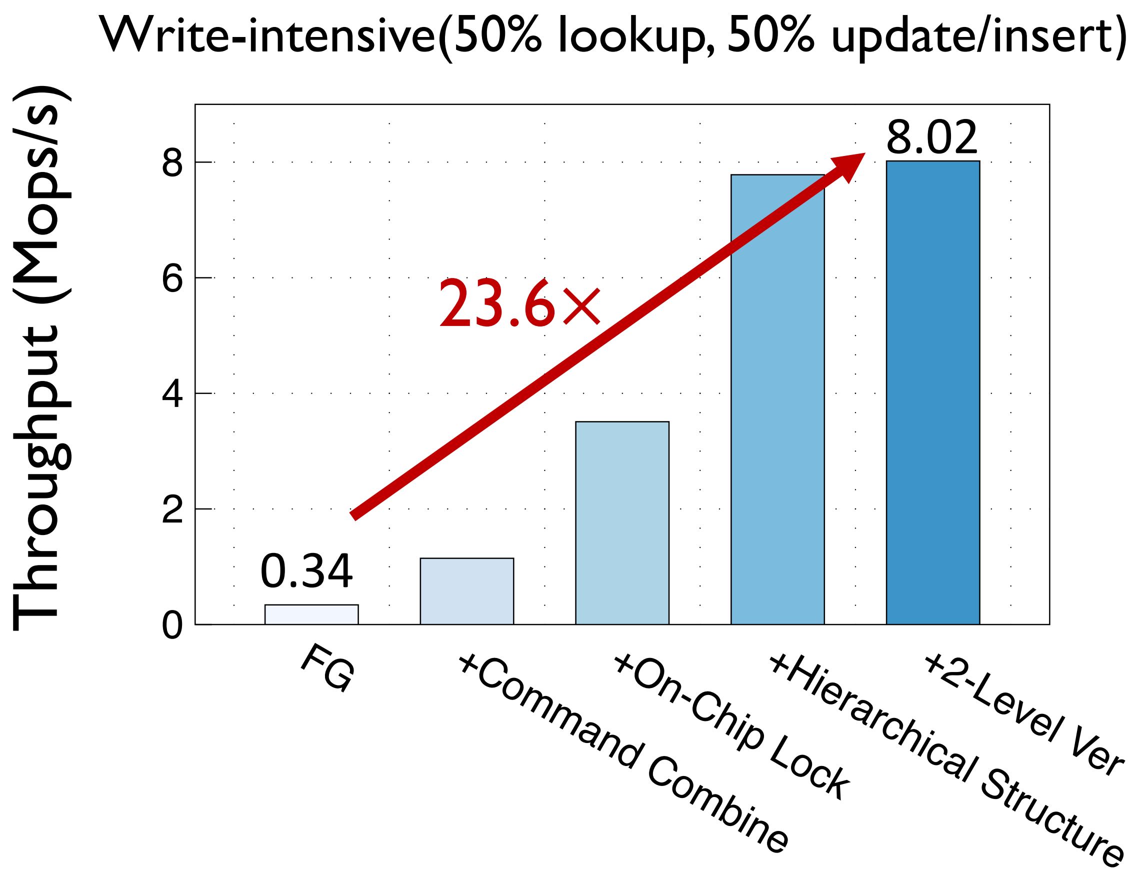
- One-sided RDMA for all index ops, so it can be deployed on DM
- RDMA locks for write-write conflicts; checksum for read-write conflicts
- We add CS-side index cache for FG, for fair comparison

Benchmark: YCSB, Zipfian 0.99; 8B key & 8B value, 1 billion KV; 1KB node; 500MB index cache

Throughput (176 client threads)

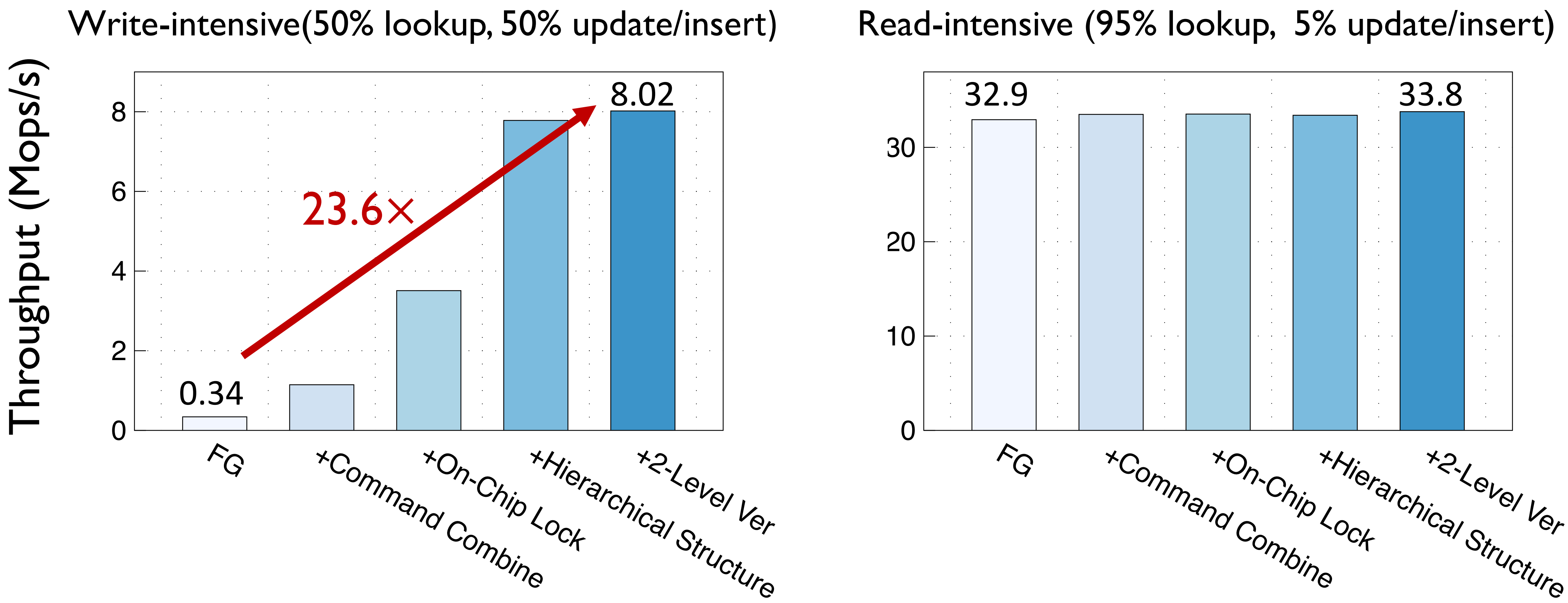


Throughput (176 client threads)



1. Sherman improves throughput significantly under write-intensive workloads
2. All techniques contribute to the high write efficiency

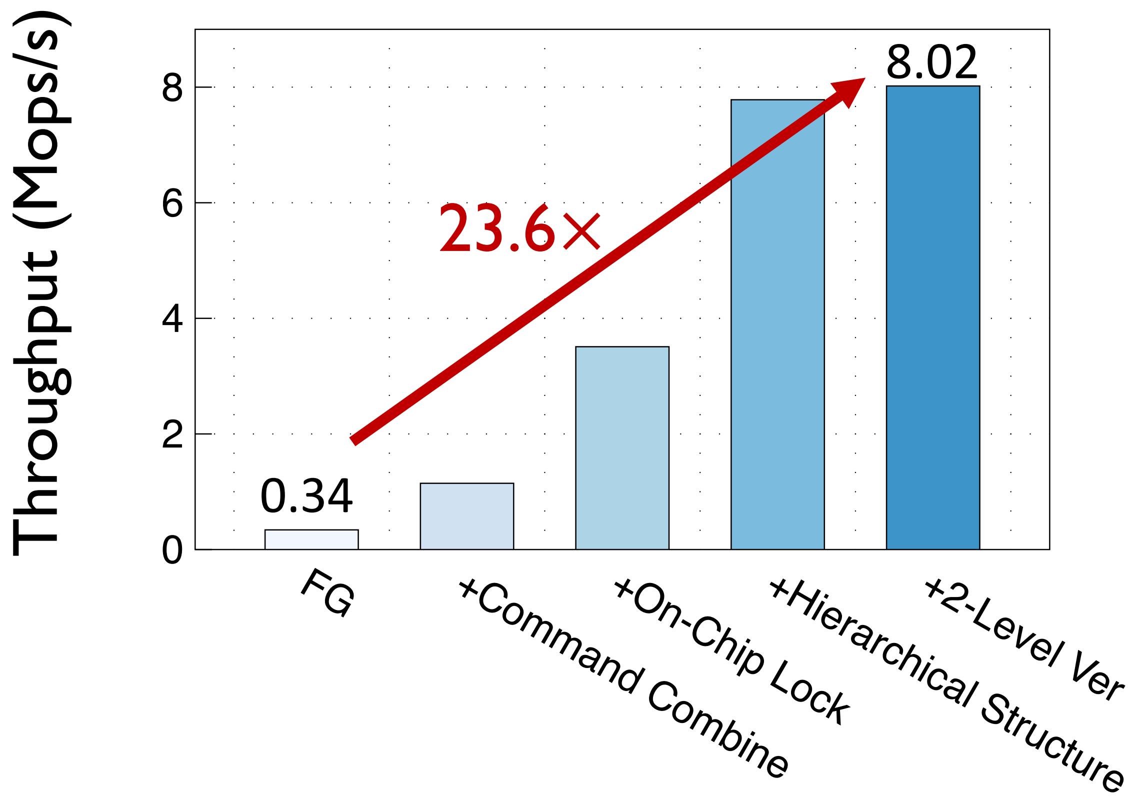
Throughput (176 client threads)



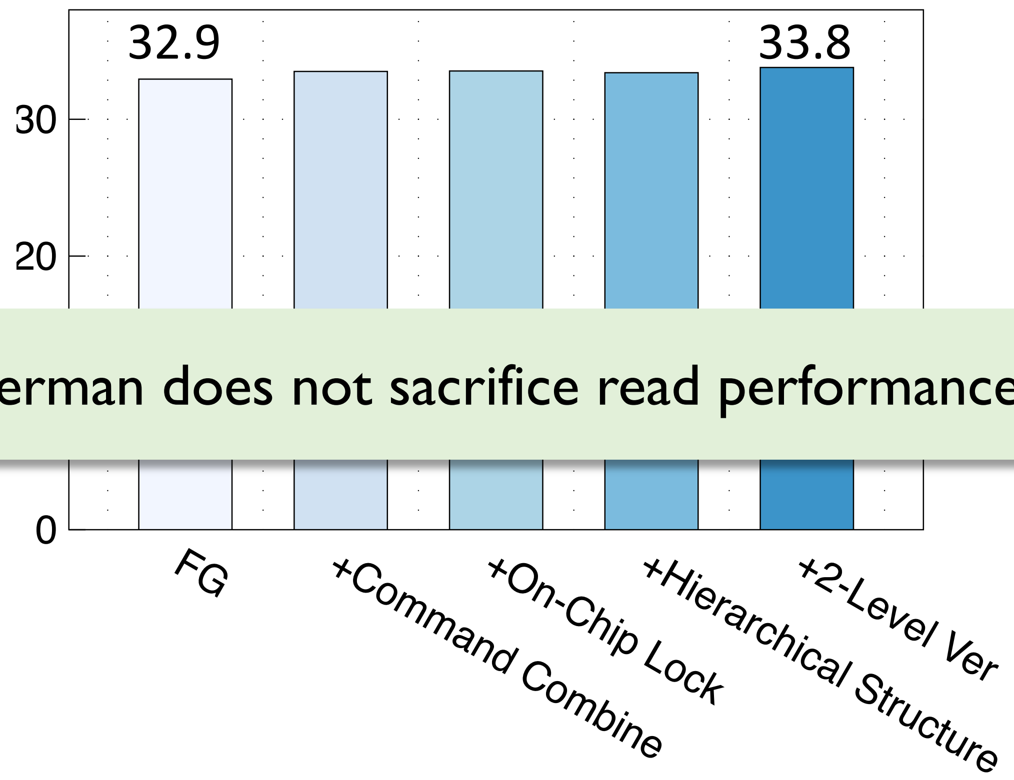
1. Sherman improves throughput significantly under write-intensive workloads
2. All techniques contribute to the high write efficiency

Throughput (176 client threads)

Write-intensive(50% lookup, 50% update/insert)



Read-intensive (95% lookup, 5% update/insert)

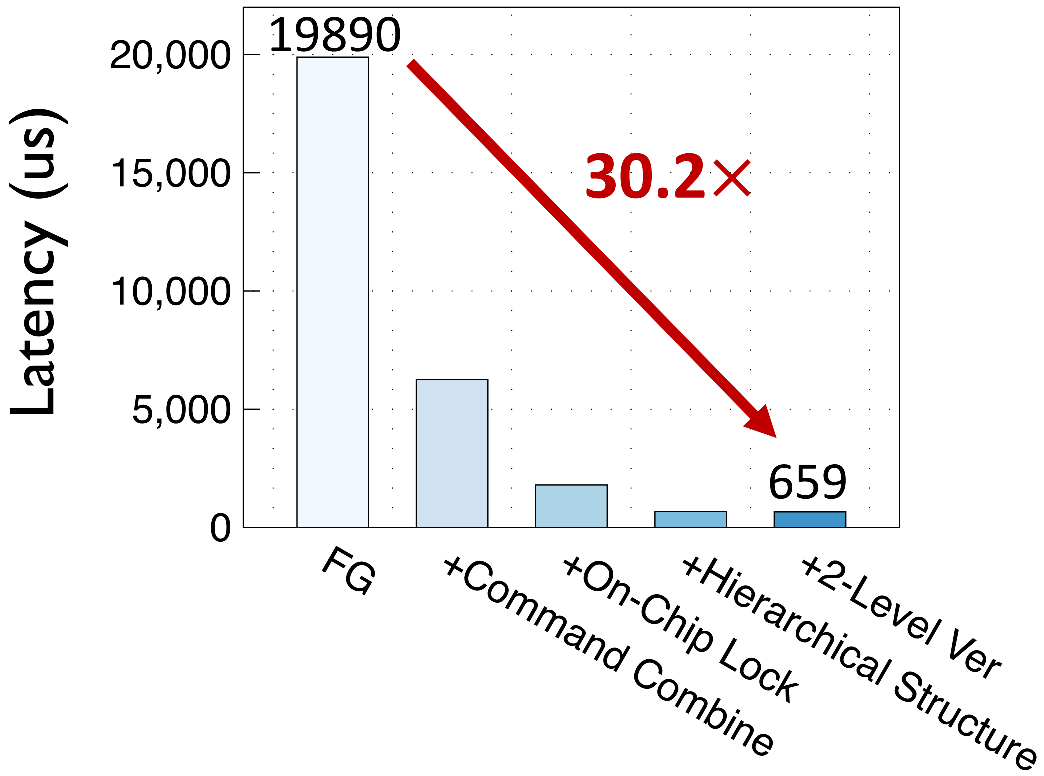


Sherman does not sacrifice read performance

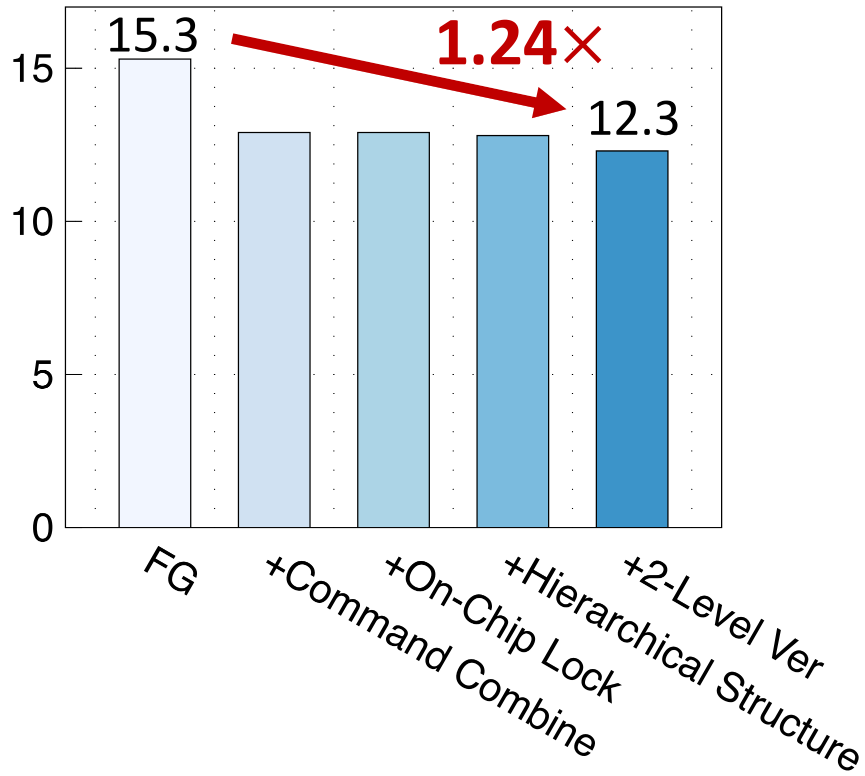
1. Sherman improves throughput significantly under write-intensive workloads
2. All techniques contribute to the high write efficiency

99th Percentile Latency (176 client threads)

Write-intensive(50% lookup, 50% update/insert)

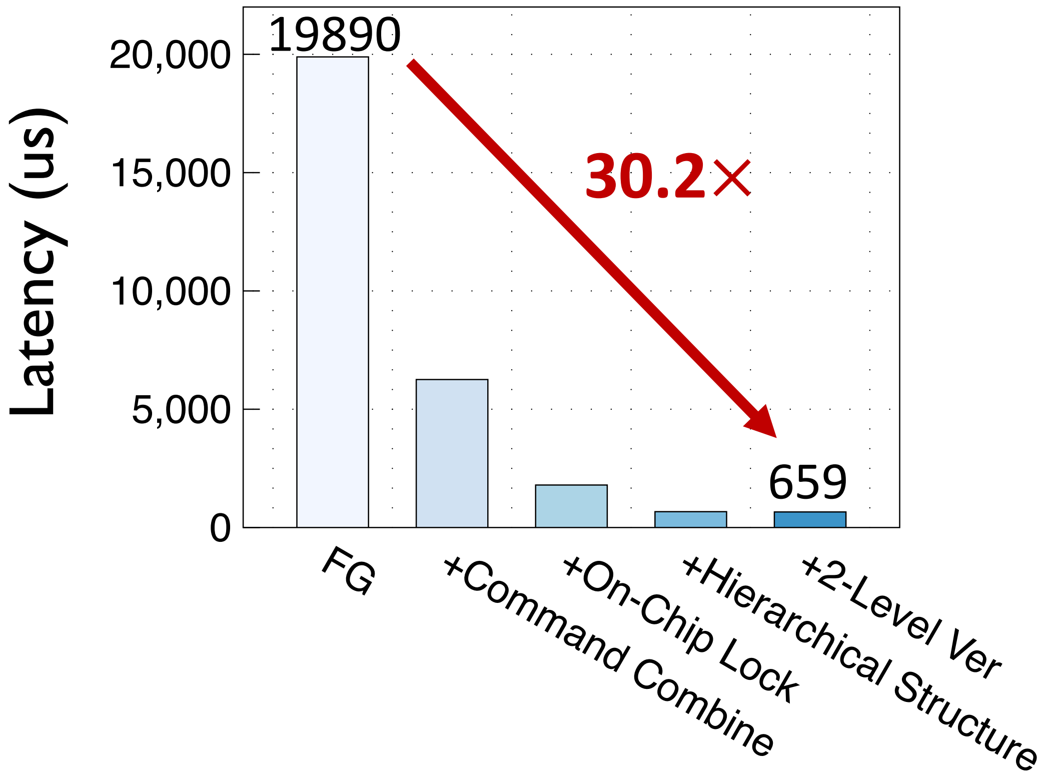


Read-intensive (95% lookup, 5% update/insert)

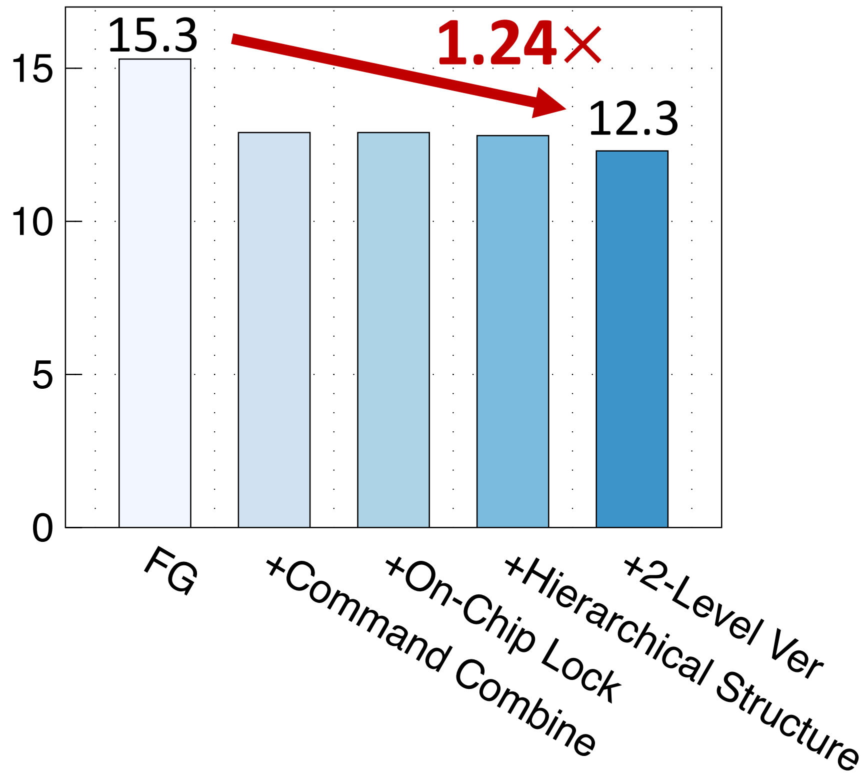


99th Percentile Latency (176 client threads)

Write-intensive(50% lookup, 50% update/insert)



Read-intensive (95% lookup, 5% update/insert)



Sherman lowers tail latency by reducing round trips and boosting concurrency efficiency

Outline

- Background & Motivation
- Sherman – A Write-Optimized B+Tree on Disaggregated Memory
- Evaluation
- Summary

Summary

- Goal
  - Building a fast tree index on disaggregated memory with commodity RDMA NICs

Summary

- Goal
  - Building a fast tree index on disaggregated memory with commodity RDMA NICs

- Key Idea
  - Combining RDMA hardware features with RDMA-friendly software techniques

Summary

- Goal
  - Building a fast tree index on disaggregated memory with commodity RDMA NICs
- Key Idea
  - Combining RDMA hardware features with RDMA-friendly software techniques
- Techniques in Sherman
  - Command combination – Reducing round trips
  - Hierarchical on-chip lock – Accelerating concurrent accesses
  - Two-level version layout – Mitigating write amplification

Summary

- Goal
  - Building a fast tree index on disaggregated memory with commodity RDMA NICs
- Key Idea
  - Combining RDMA hardware features with RDMA-friendly software techniques
- Techniques in Sherman
  - Command combination – Reducing round trips
  - Hierarchical on-chip lock – Accelerating concurrent accesses
  - Two-level version layout – Mitigating write amplification
- Results
  - Sherman improves throughput and 99th percentile latency by one order of magnitude on typical write-intensive workloads

Thanks & QA

Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory

Contact Information: q-wang18@mails.tsinghua.edu.cn