香港中文大學
The Chinese University of Hong Kong

ARISE
Automated Reliable Intelligent Software Engineering

Efficient Data Structures and Algorithms for Practical Resource Disaggregated Data Centers

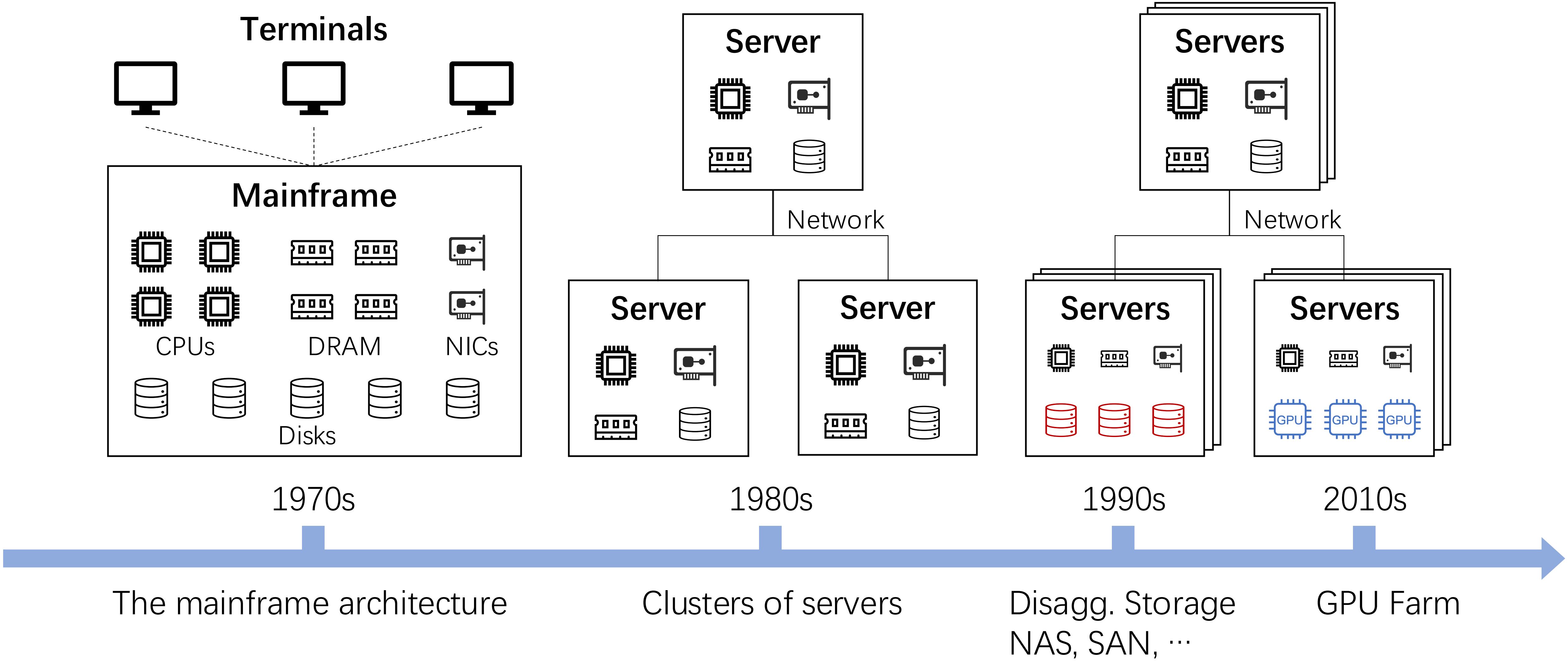
Jiacheng Shen

Ph.D. Oral Defense

Supervisor: Prof. Michael R. Lyu

18 June 2024

Data centers are heading towards disaggregation



Data centers are heading towards disaggregation

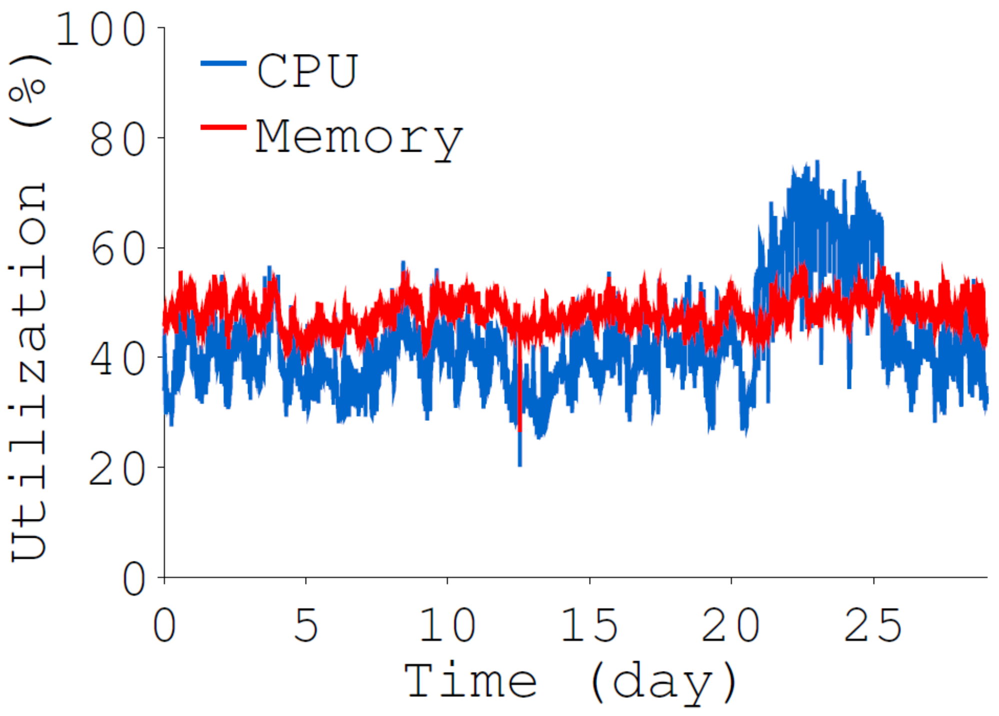
Lessons from History

Resource efficiency is a key motivation for all these successful disaggregations!

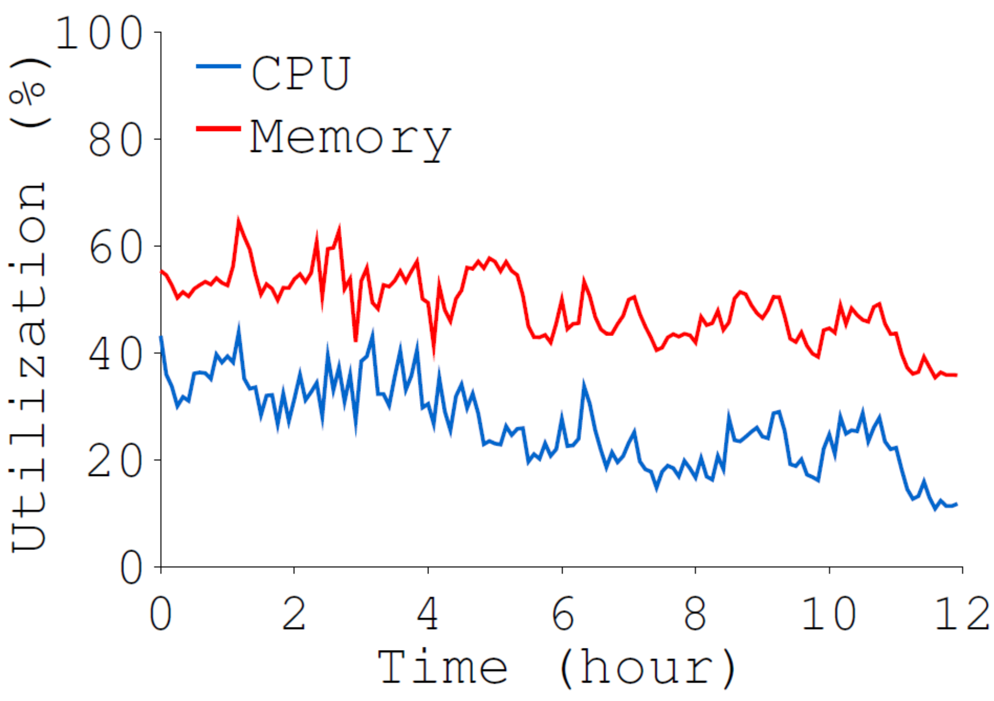
1970s	1980s	1990s	2010s
The mainframe architecture	Clusters of servers	Disagg. Storage NAS, SAN, …	GPU Farm

Data centers still suffer from resource inefficiency

Root cause: **resource coupling** on monolithic servers



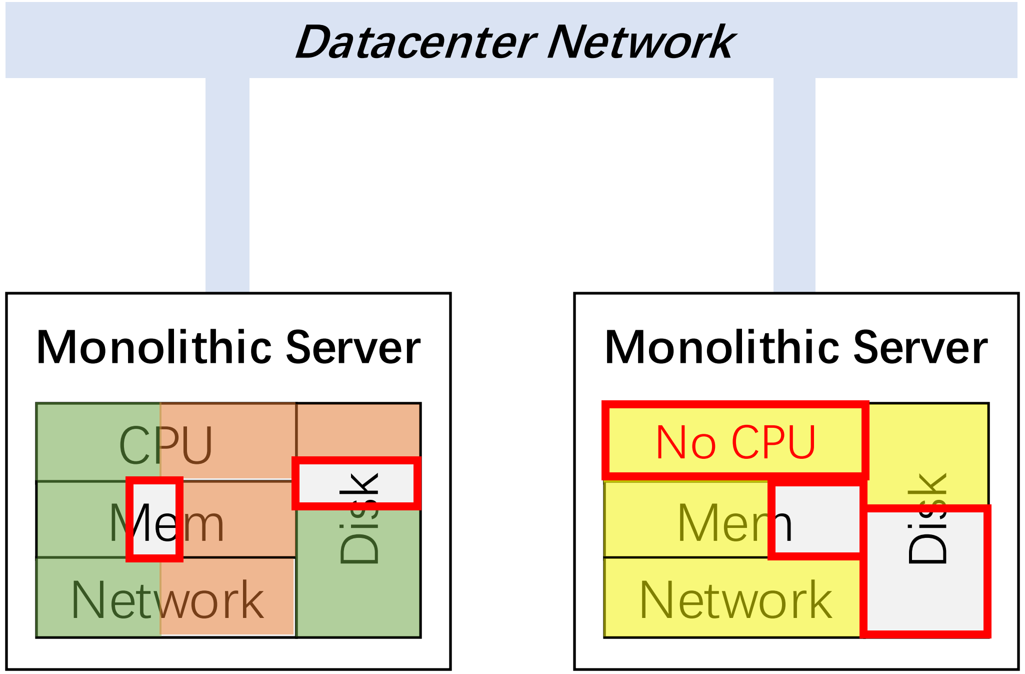
Google Cluster[1]



Alibaba Cluster[1]

[1] Yizhou Shan et al. "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation." OSDI 2018.

Data centers still suffer from resource inefficiency

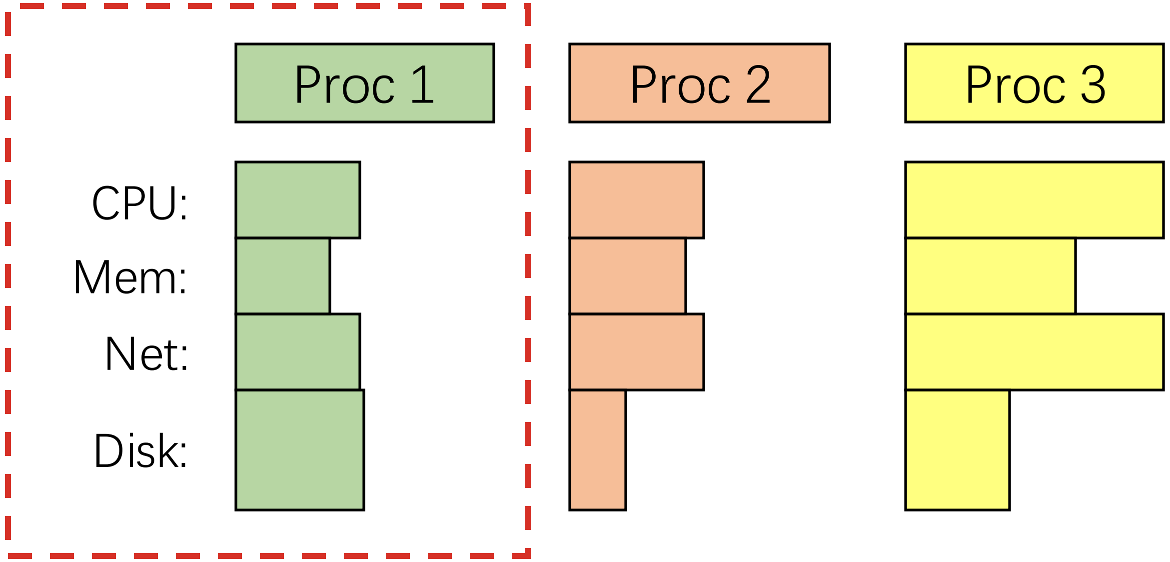


Fragmentation

Stranding

☹ ***Resource coupling***

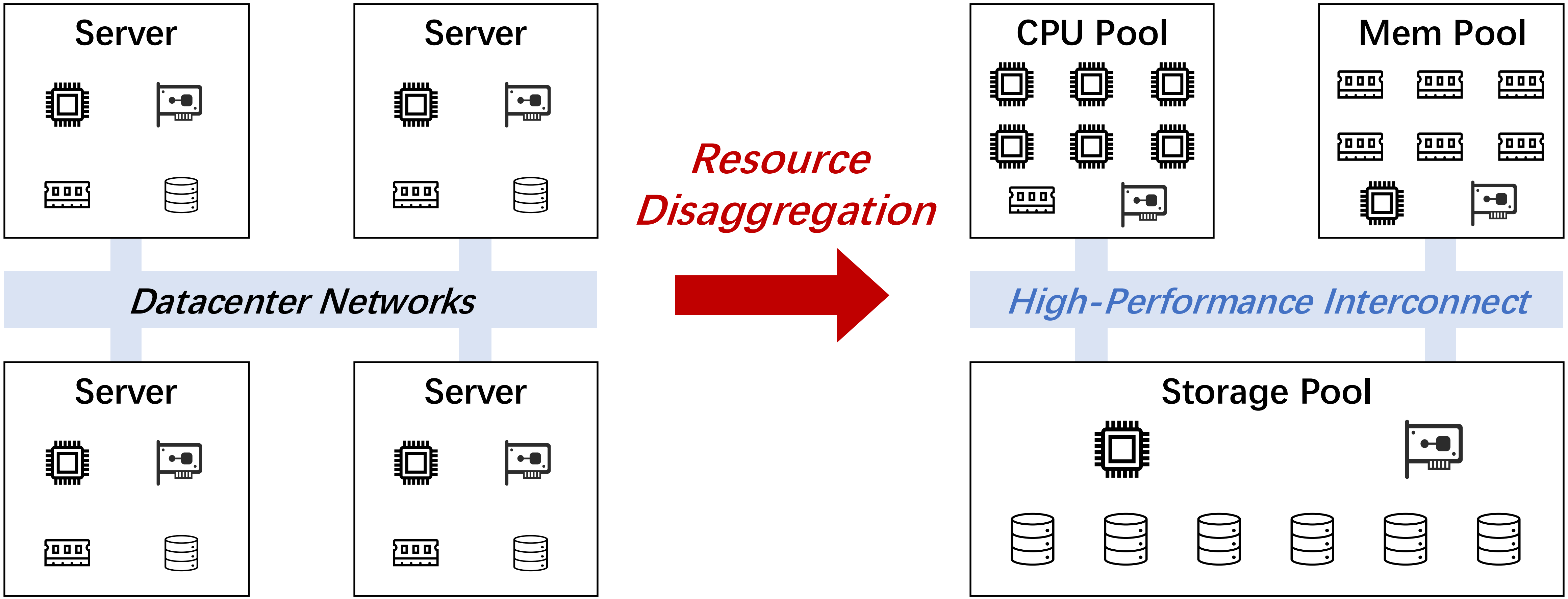
☹ **Multi-dimensional bin packing**



Must be allocated from a single server

Resource-Disaggregated Data Centers

Can we decouple resources from monolithic servers?



Resource-Disaggregated Data Centers

Can we decouple resources from monolithic servers?

☺ ***Resource efficiency***

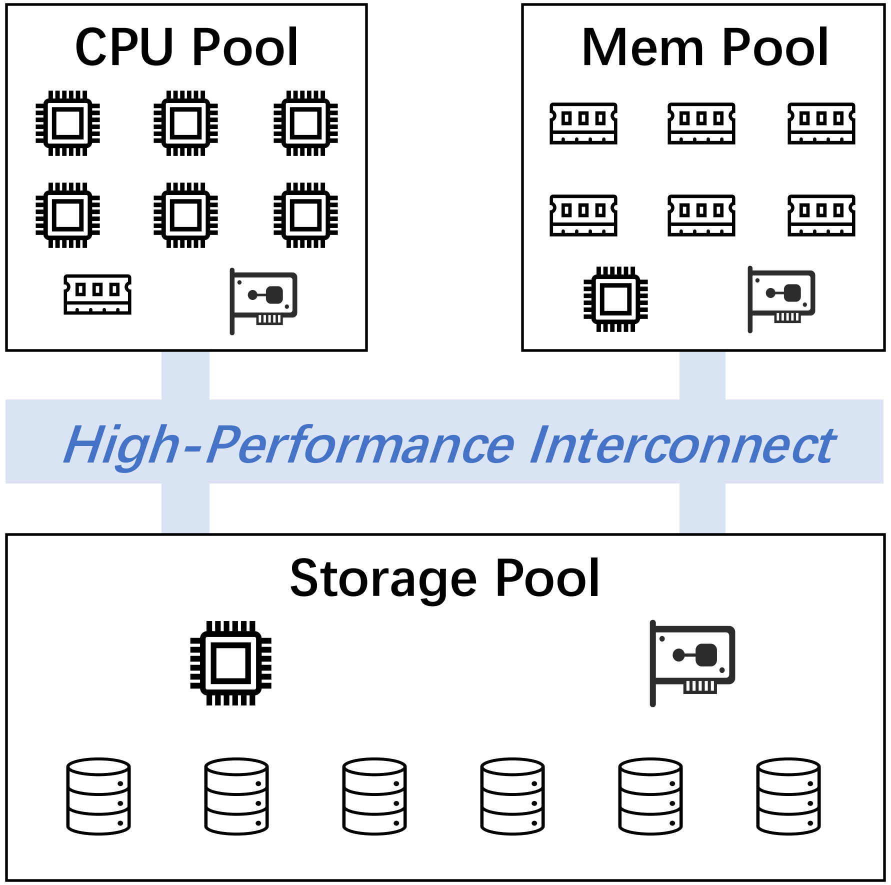
- ➢ Resources can be allocated flexibly
- ➢ Mitigate fragmentation & eliminate stranding

☺ ***Improved scalability***

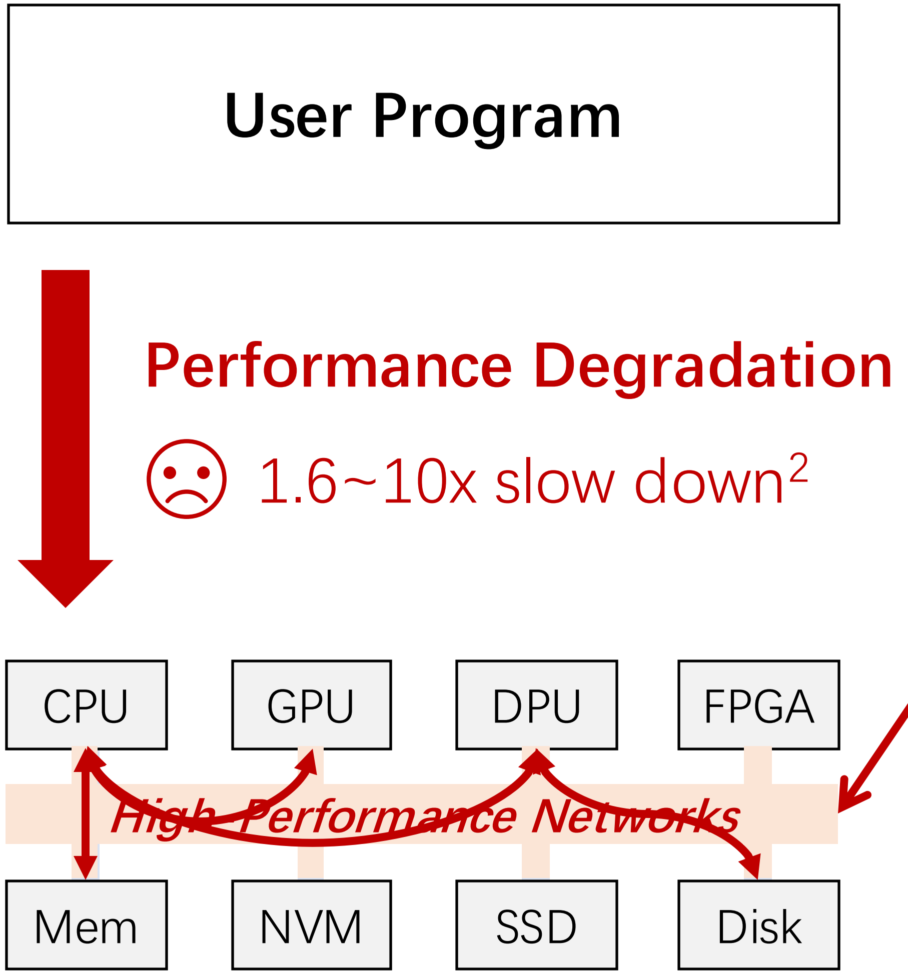
- ➢ Resources can be scaled independently

☺ ***Fine-grained failure domain***

- ➢ Hardware failures are isolated from each other



Problem with resource disaggregation



Existing data structures and algorithms become ***unsuitable***

Larger physical distance leads to **higher communication overhead**

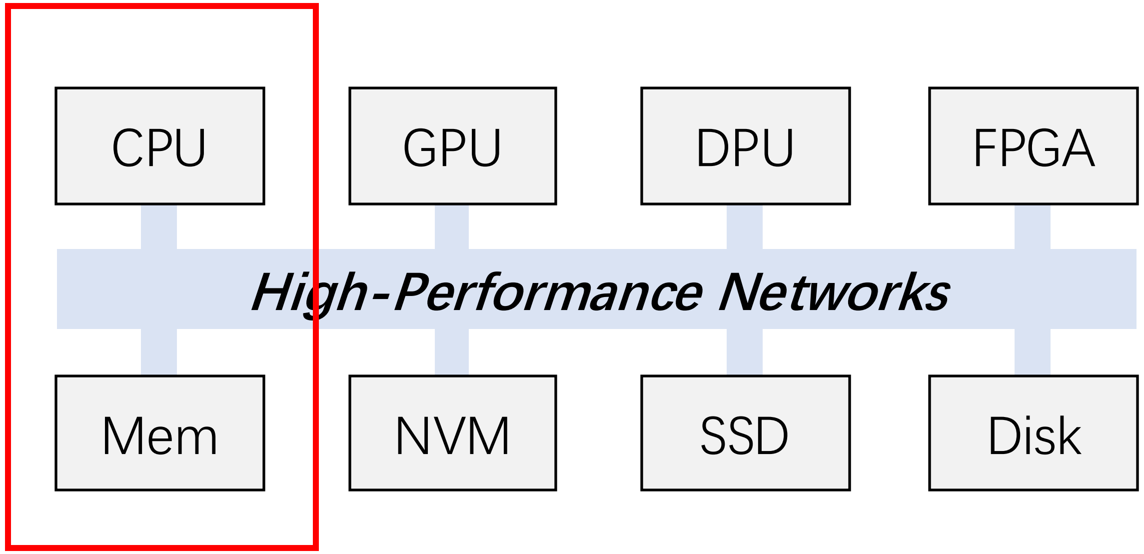
Mitigating the performance degradation is a ***systems problem***

[2] Qizhen Zhang et al. “Understanding the effect of data center resource disaggregation on production DMBSs.” VLDB 2020.

My work during my Ph.D. study

Bottom-Up Approach
Design *data structures and algorithms* to compose
high-performance disaggregated systems

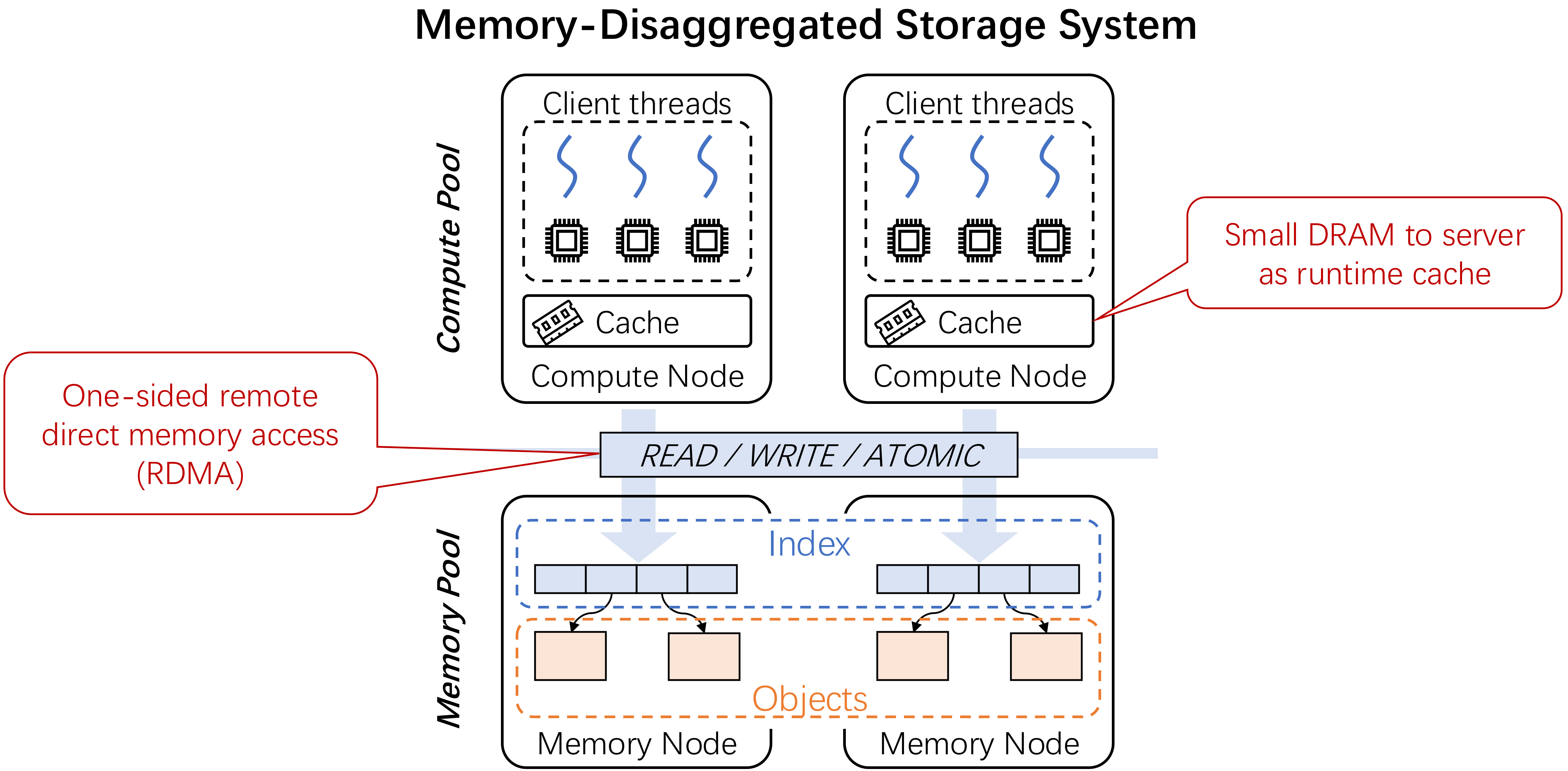
Disaggregated Memory



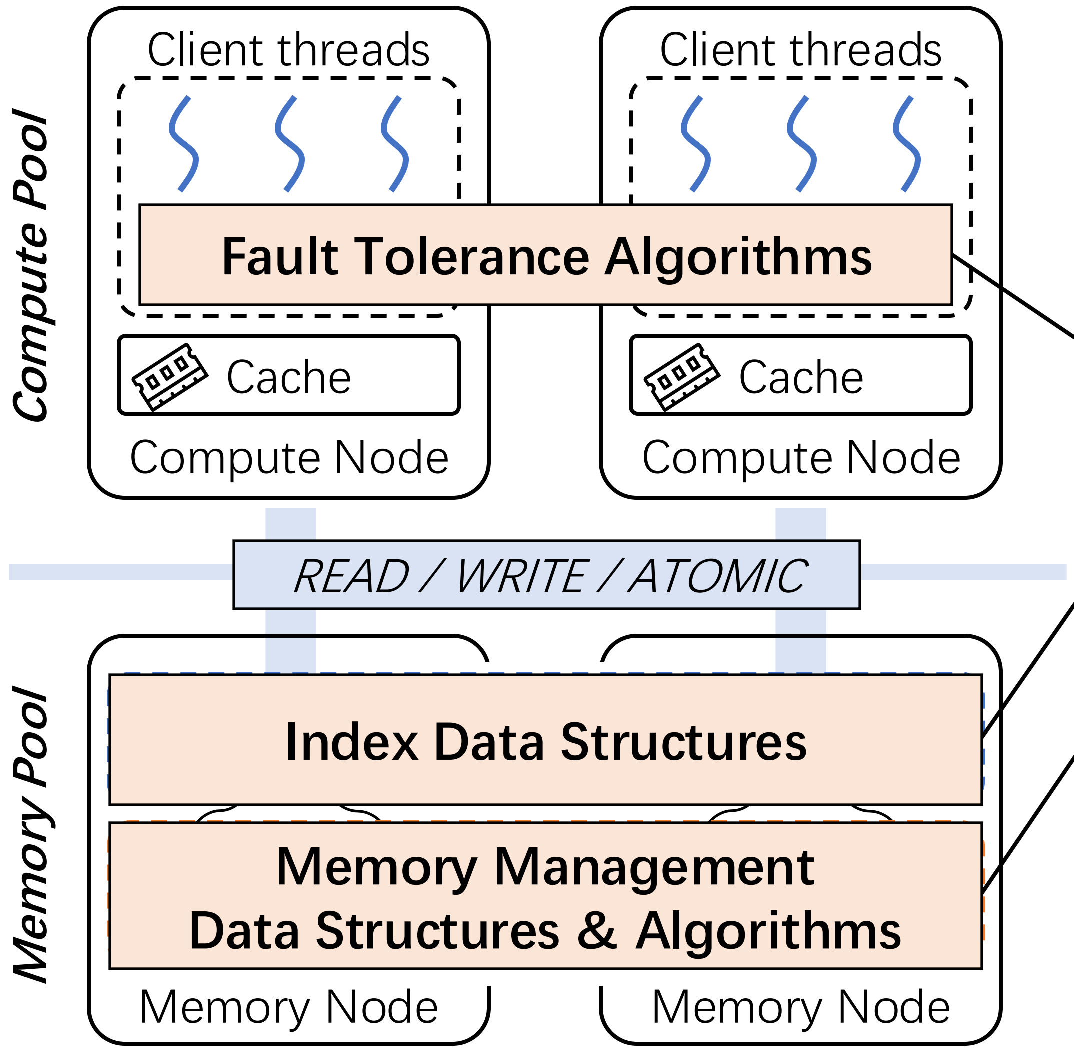
In-Memory Storage Systems

- Widely adopted in cloud data centers
- Contains many common data structures and algorithms

My work during my Ph.D. study



Thesis Contributions



Analysis

Guidelines to achieve high performance

Design

Efficient replication and logging algorithms

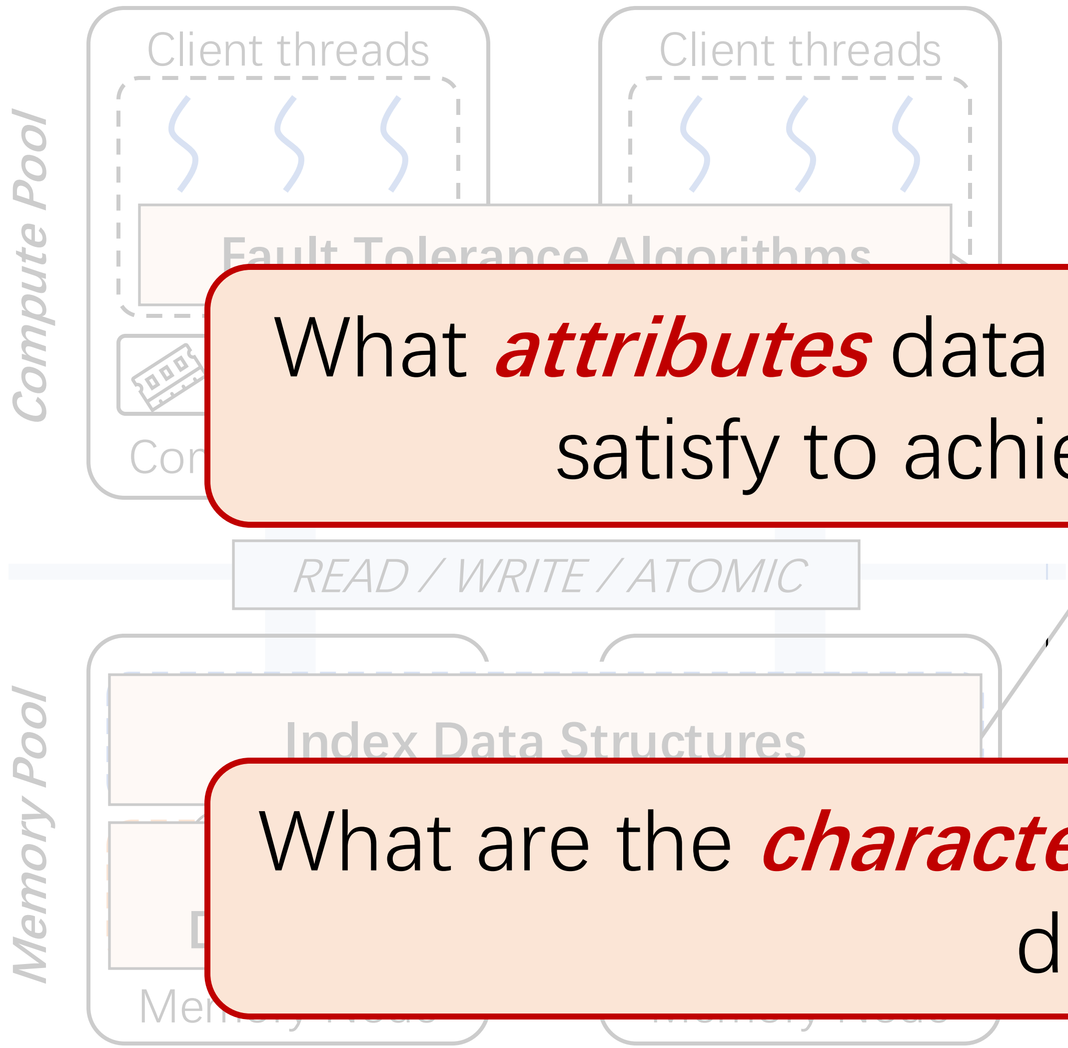
A high-performance range index data structure

Data structures and algorithms for memory allocation and cache replacement

Deployment

A production-level disaggregated caching service

Thesis Contributions



Analysis

Guidelines to achieve high performance

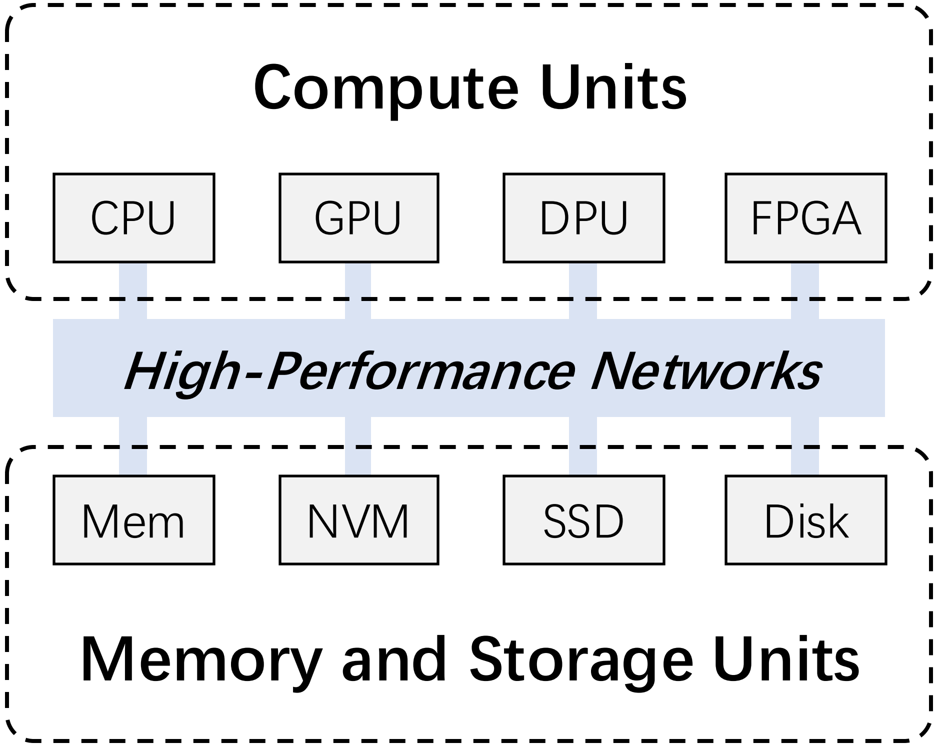
Data structures and algorithms for memory allocation and cache replacement

What ***attributes*** data structures and algorithms must satisfy to achieve good performance?

What are the ***characters*** of a resource-disaggregated data center?

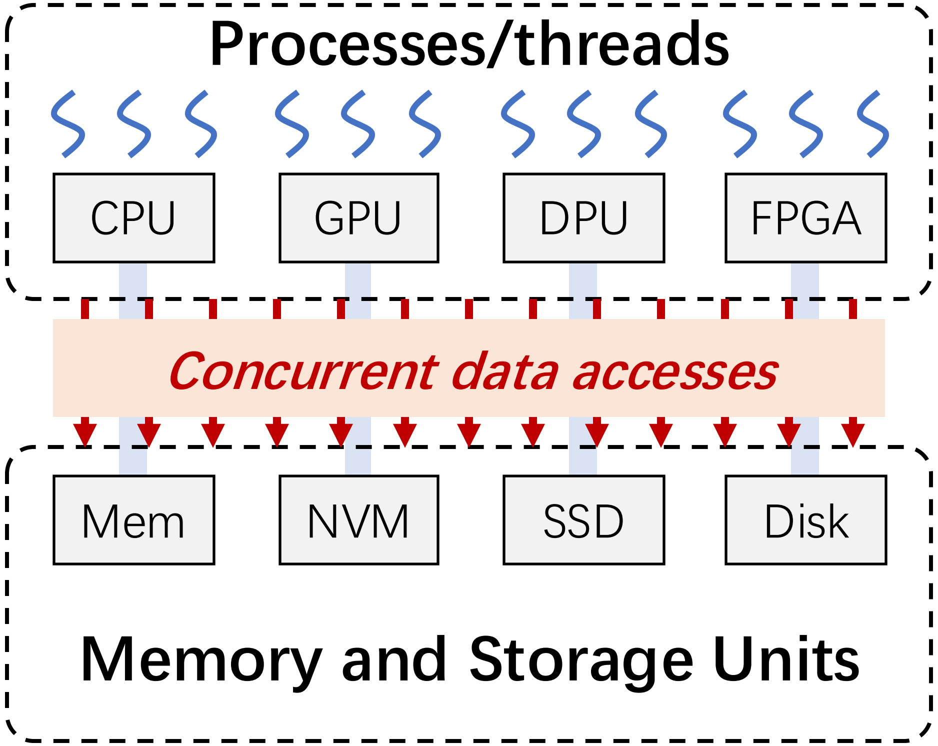
The disaggregated architecture is like…

A data center scale big computer



The disaggregated architecture is like…

A data center scale big computer

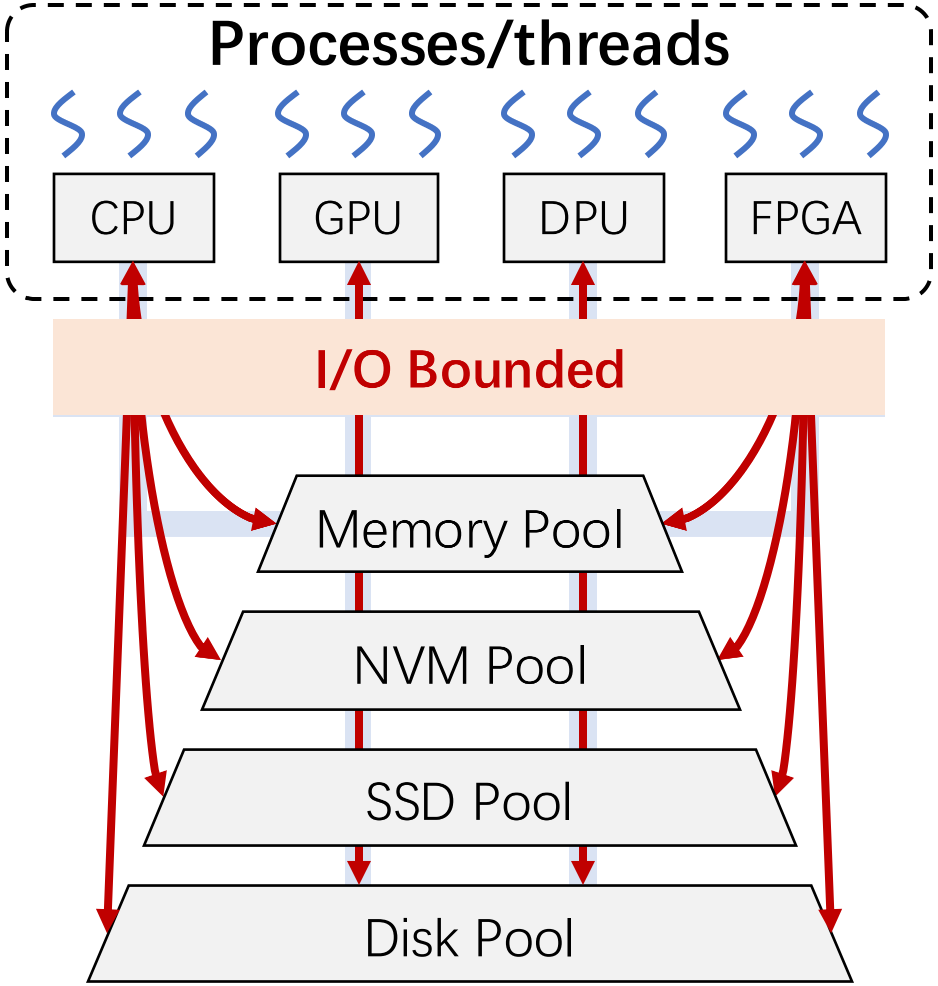


The compute unit:

- A large-scale parallel machine
- Need to optimize **concurrency**

The disaggregated architecture is like…

A data center scale big computer



The compute unit:

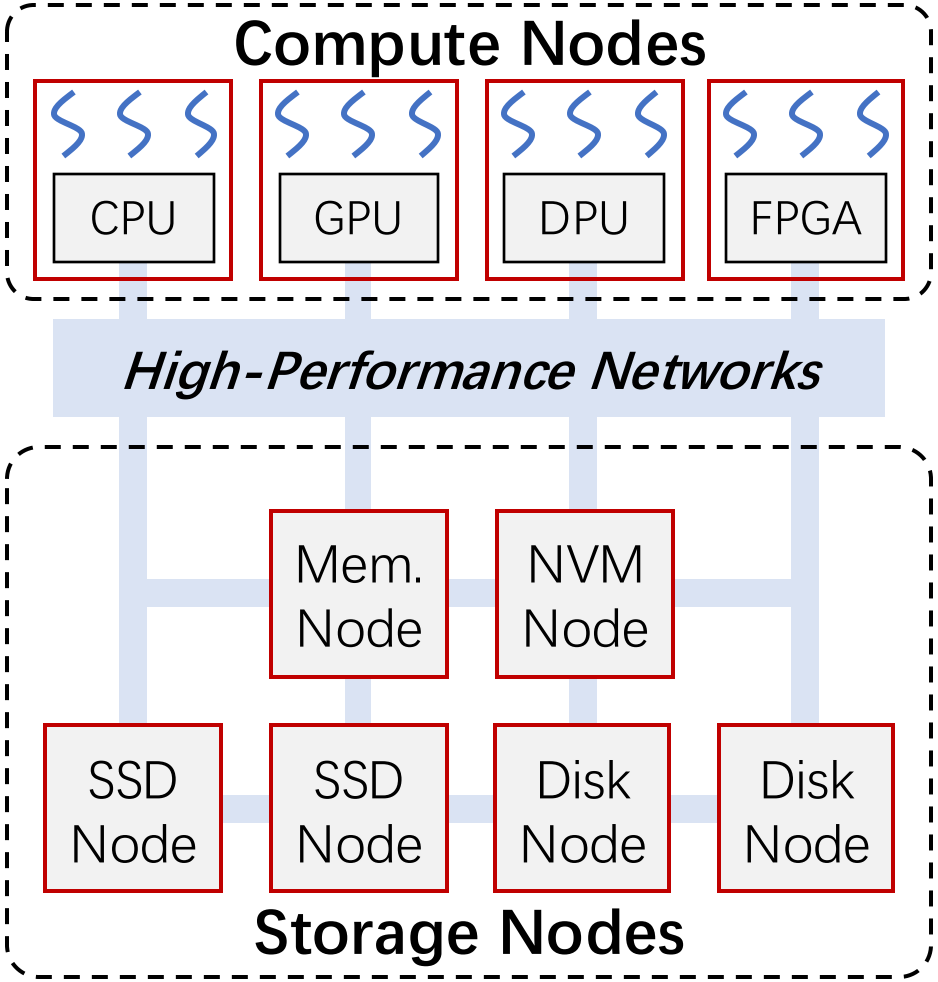
- A large-scale parallel machine
- Need to optimize **concurrency**

The storage unit:

- Yet another tiered memory system
- Need to optimize **I/O**

The disaggregated architecture is like…

A data center scale big computer



The compute unit:

- A large-scale parallel machine
- Need to optimize **concurrency**

The storage unit:

- Yet another tiered memory system
- Need to optimize **I/O**

The physical construction:

- An asymmetric distributed system
- Need to optimize **asymmetry**

Data structure and algorithms we now have

	Concurrency	I/O	Asymmetry
Parallel machines	🙂	☹️	☹️
Tiered memory systems	😑	🙂	☹️
Distributed systems	😑	😑	☹️

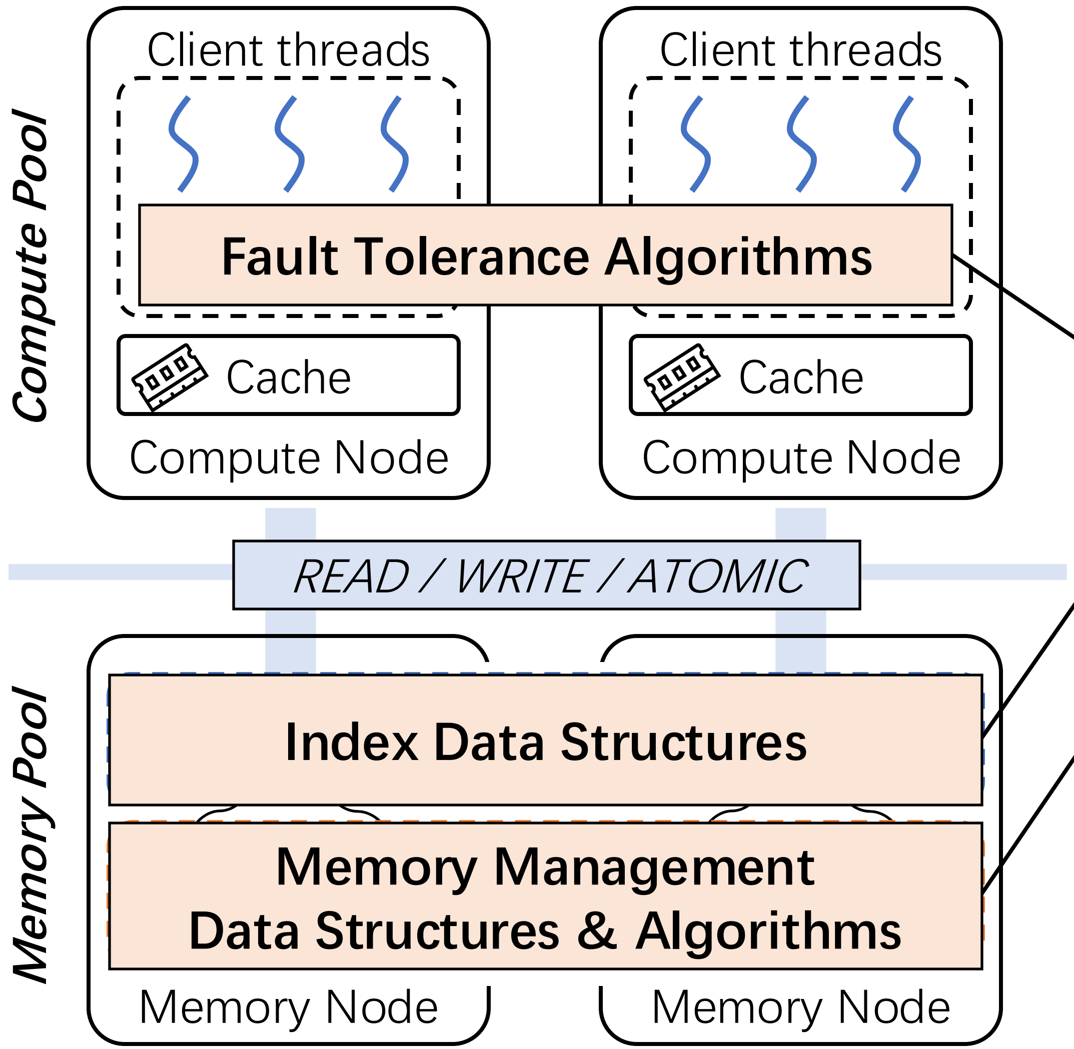
Data structure and algorithms we now have

What ***attributes*** data structures and algorithms must satisfy to achieve good performance?

Consider and optimize **Concurrency** **I/O** **Asymmetry** *simultaneously*.

☺ ☺ ☺

Thesis Contributions



Analysis

Guidelines to achieve high performance

Design

Efficient replication and logging algorithms

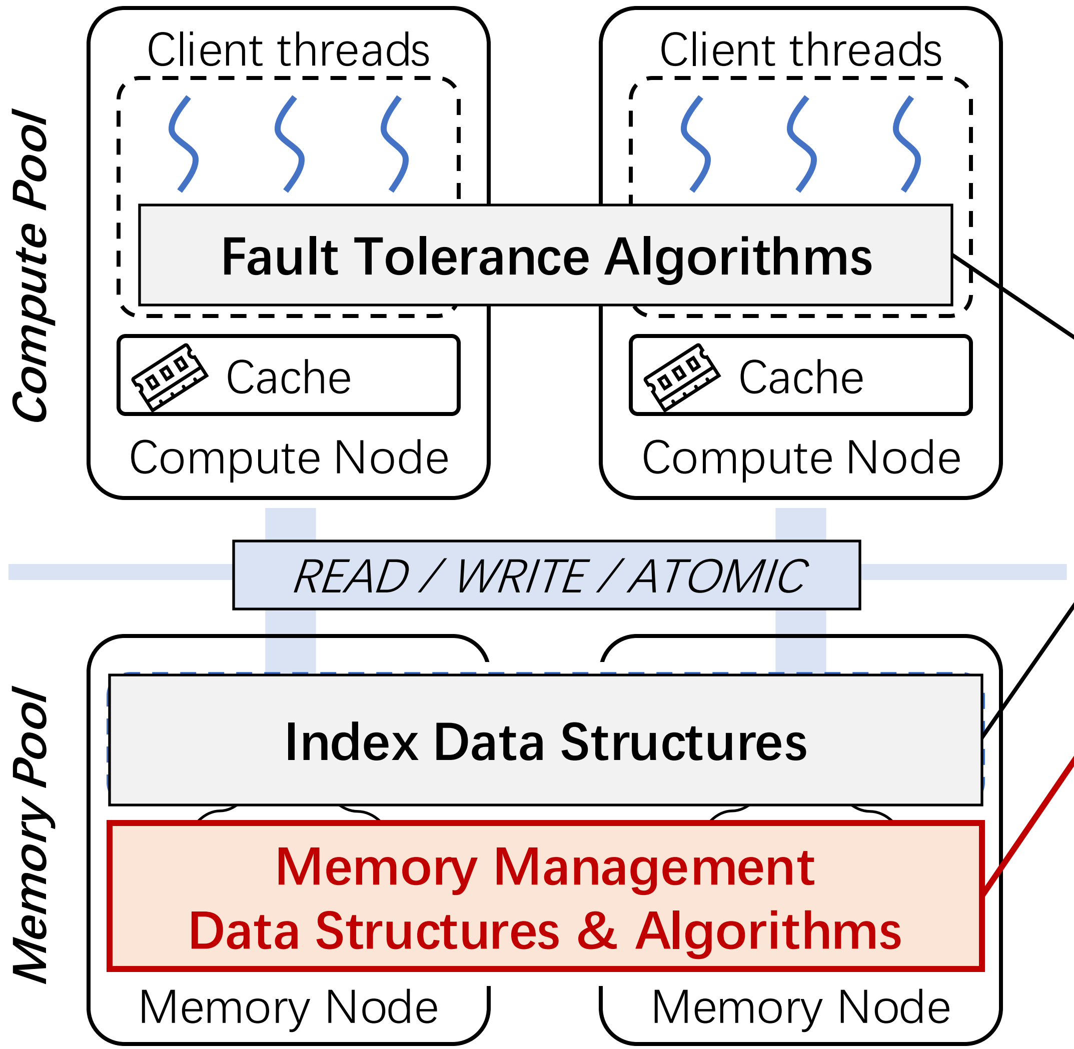
A high-performance range index data structure

Data structures and algorithms for memory allocation and cache replacement

Deployment

A production-level disaggregated caching service

Thesis Contributions



Analysis

- Guidelines to achieve high performance

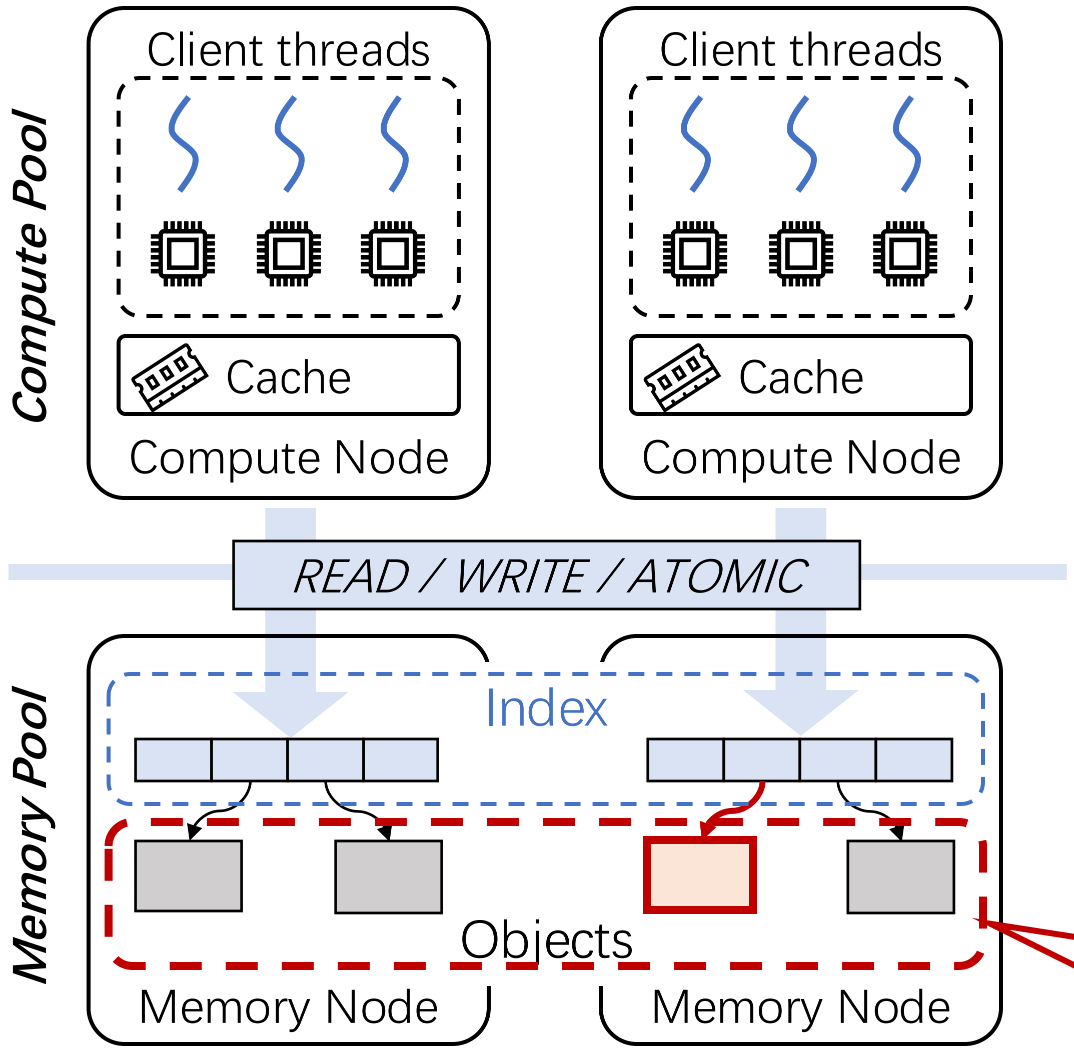
Design

- Efficient replication and logging algorithms
- A high-performance range index data structure
- Data structures and algorithms for memory allocation and cache replacement

Deployment

- A production-level disaggregated caching service

Memory Management Responsibilities



Insert Operation:

- **Allocate** a free memory space
  - **Evict** objects when no free space
- Write the data to the allocated space & modify the hash index

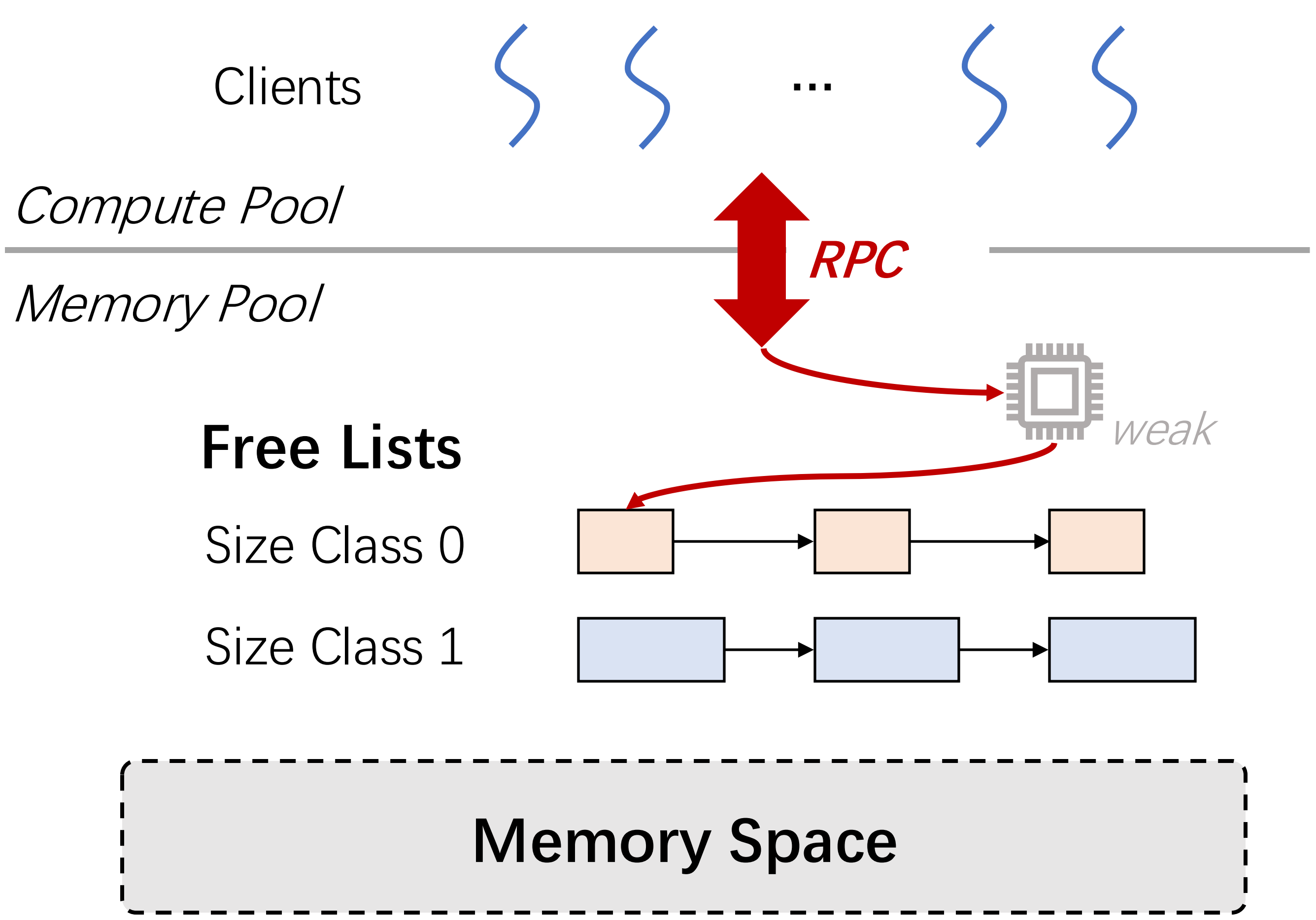
Responsibilities:

- **Memory Allocation**
  Allocate/free memory to store objects
- **Caching Algorithms**
  Host hot objects in the memory pool

Limited & expensive memory capacity!

Memory Allocation: Two approaches

Server-Centric: Maintain data structures with **MN CPUs**



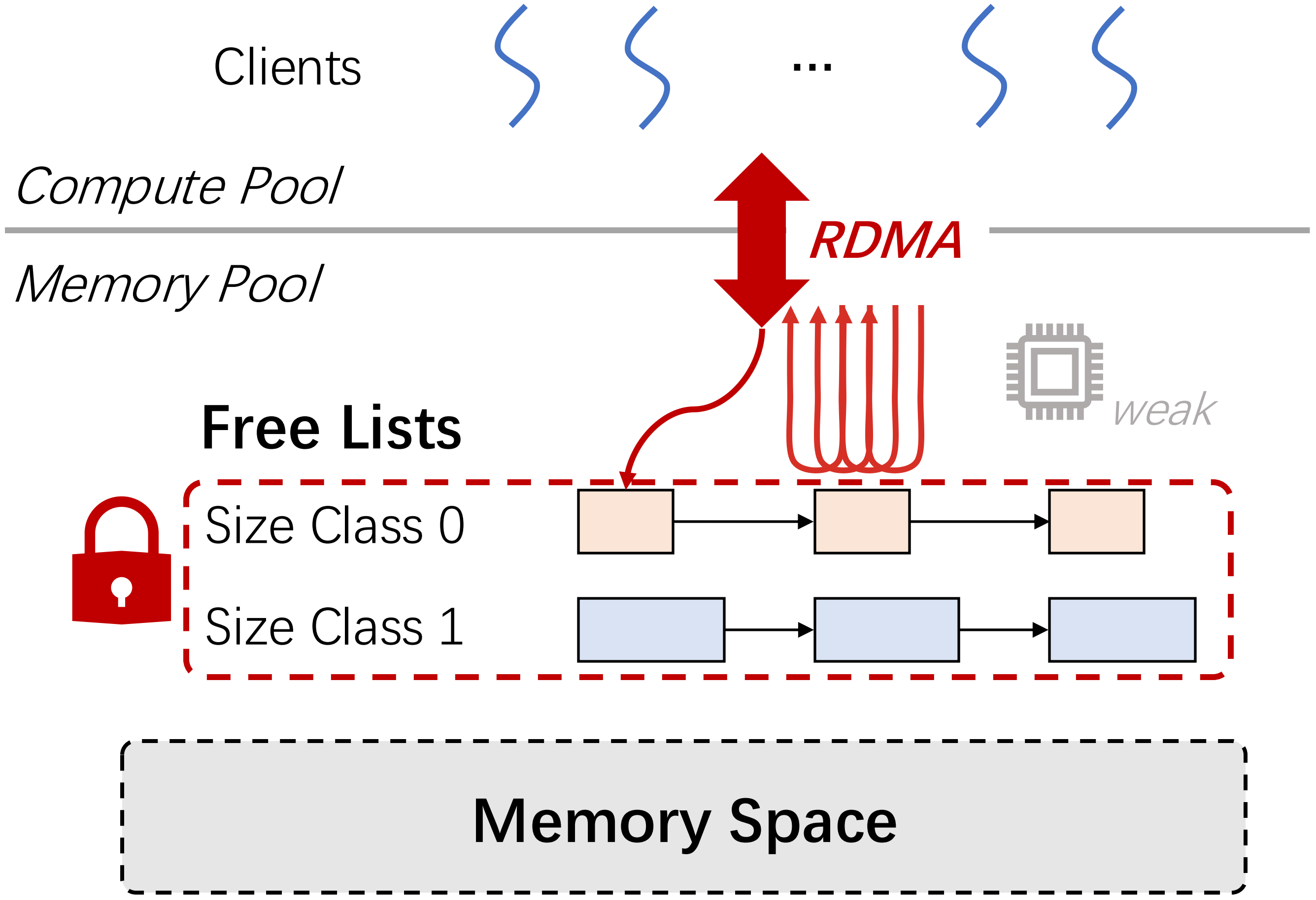
Asymmetry
- Weak CPUs on MNs limits system performance

Concurrency
- Simplified concurrency control

I/O efficiency
- Use RPCs with only one network I/O

Memory Allocation: Two approaches

Client-Centric: Maintain data structures with **RDMA**



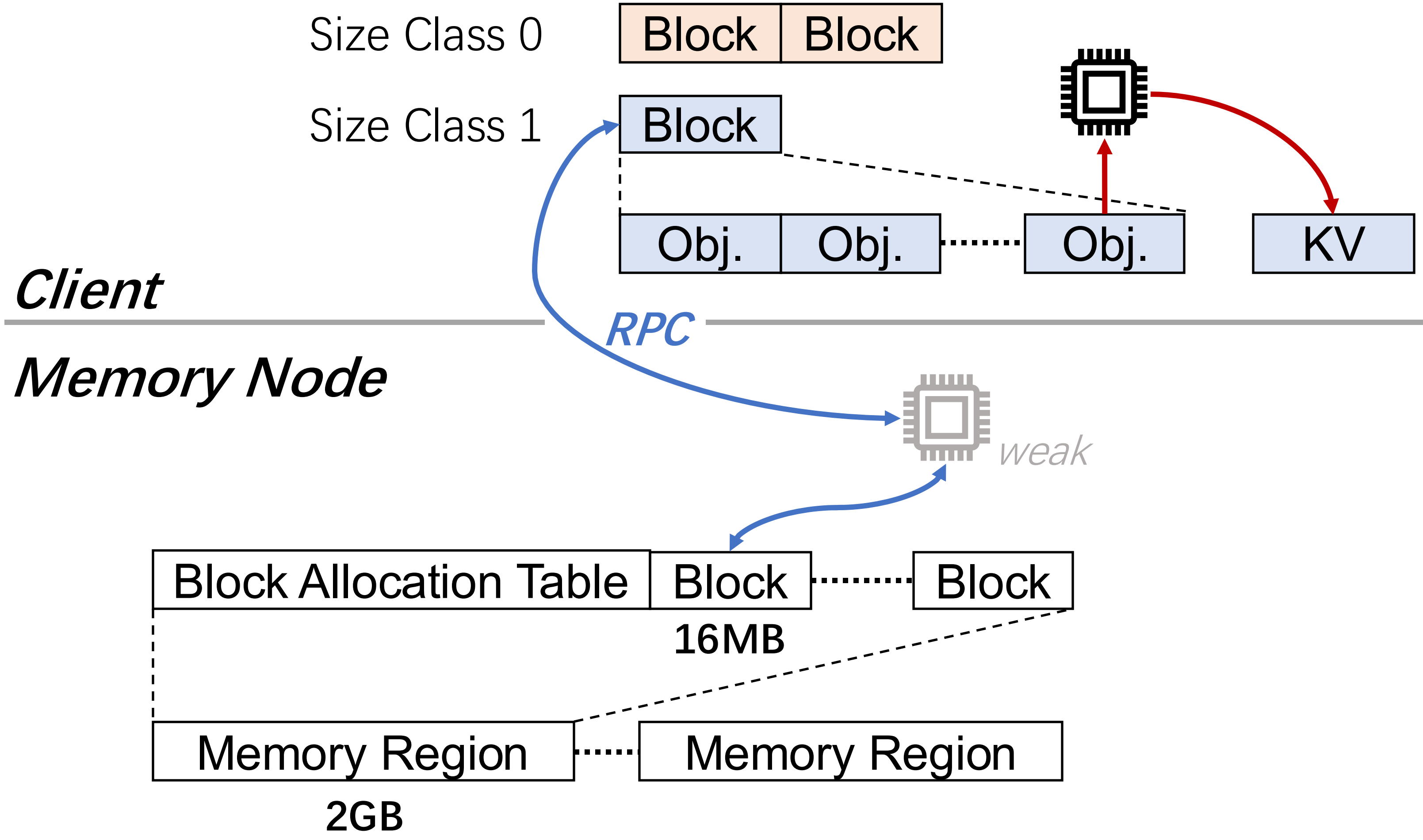
☺ ***Asymmetry***
➢ Clients directly access data w/ RDMA

☹ ***Concurrency***
➢ RDMA-based remote locks

☹ ***I/O amplifications***
➢ Multiple numbers of I/Os to access or modify remote data structures

Two-Level Memory Allocation

Key Idea: Decouple into *compute-light* and *compute-heavy* components

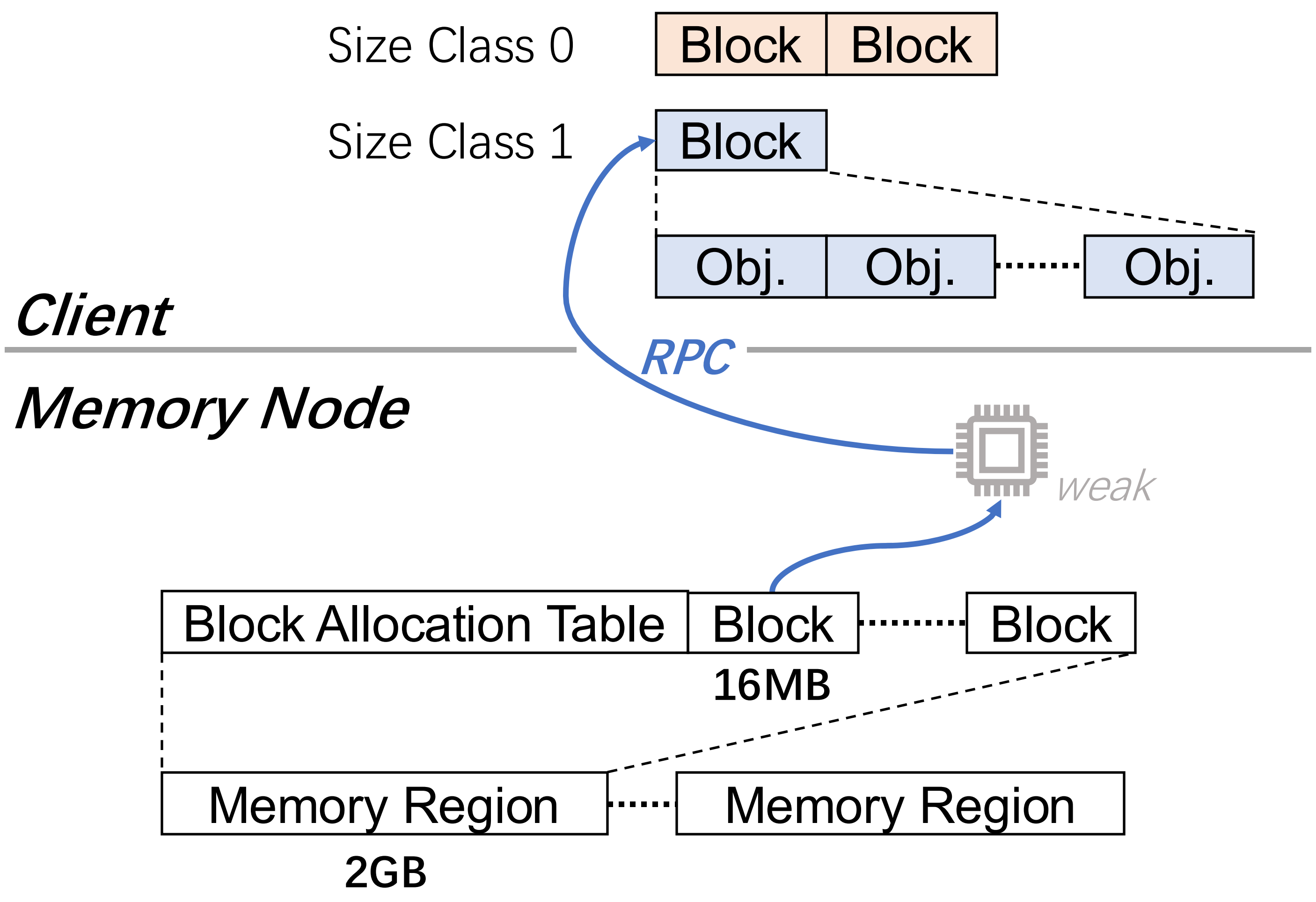


compute-heavy
fine-grained object allocation

compute-light
coarse-grained block allocation

Two-Level Memory Allocation

Key Idea: Decouple into *compute-light* and *compute-heavy* components

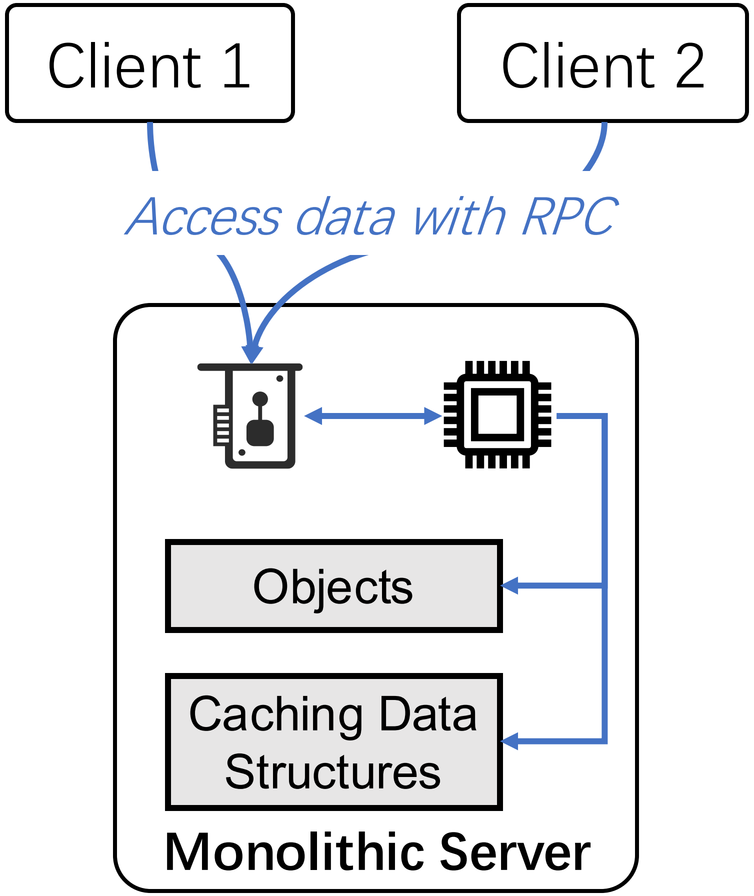


☺ ***Asymmetry***
- Scheduling right computation to suitable hardware

☺ ***Concurrency***
- Clients mostly operate on local data structures without conflicts

☺ ***I/O amplifications***
- Rely on RPCs to manipulate remote data structures off the critical path

Executing Caching Algorithms



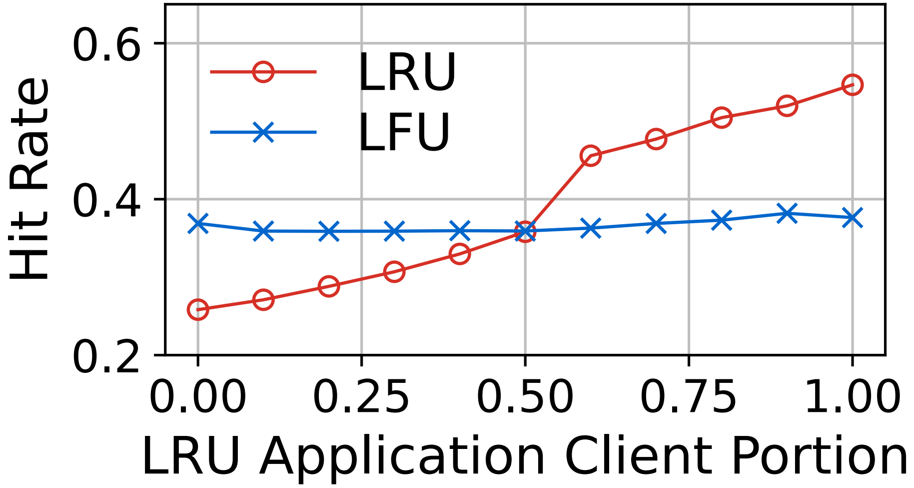
- **Monitoring** object accesses to evaluate object hotness
- Maintaining object hotness information in **caching data structures** to efficiently locate cold objects on eviction
  - E.g., lists for LRU

Goal: Achieve high cache hit rates

Changing resources on DM affects cache hit rates

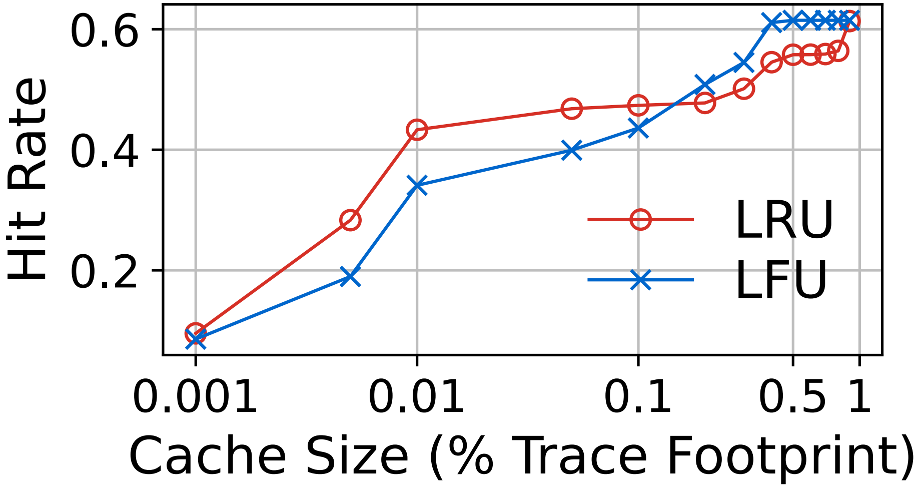
Changing compute resources

Alters the overall data access pattern



Changing memory resources

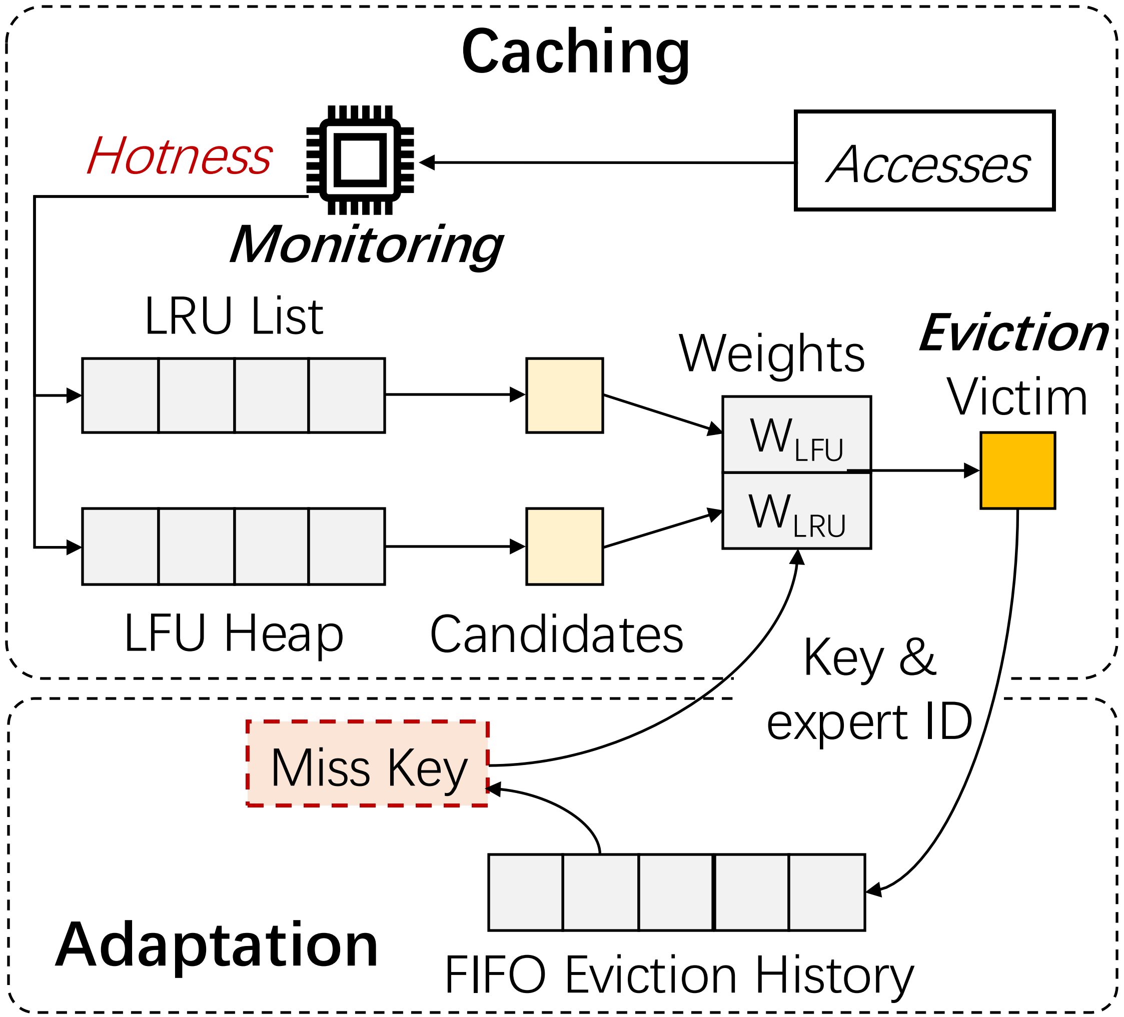
Changes the performance of caching algorithms



Adopt adaptive caching to adapt to the caching resources and workloads

Existing Adaptive Caching Schemes

Model cache eviction as **multi-armed-bandit (MAB)**

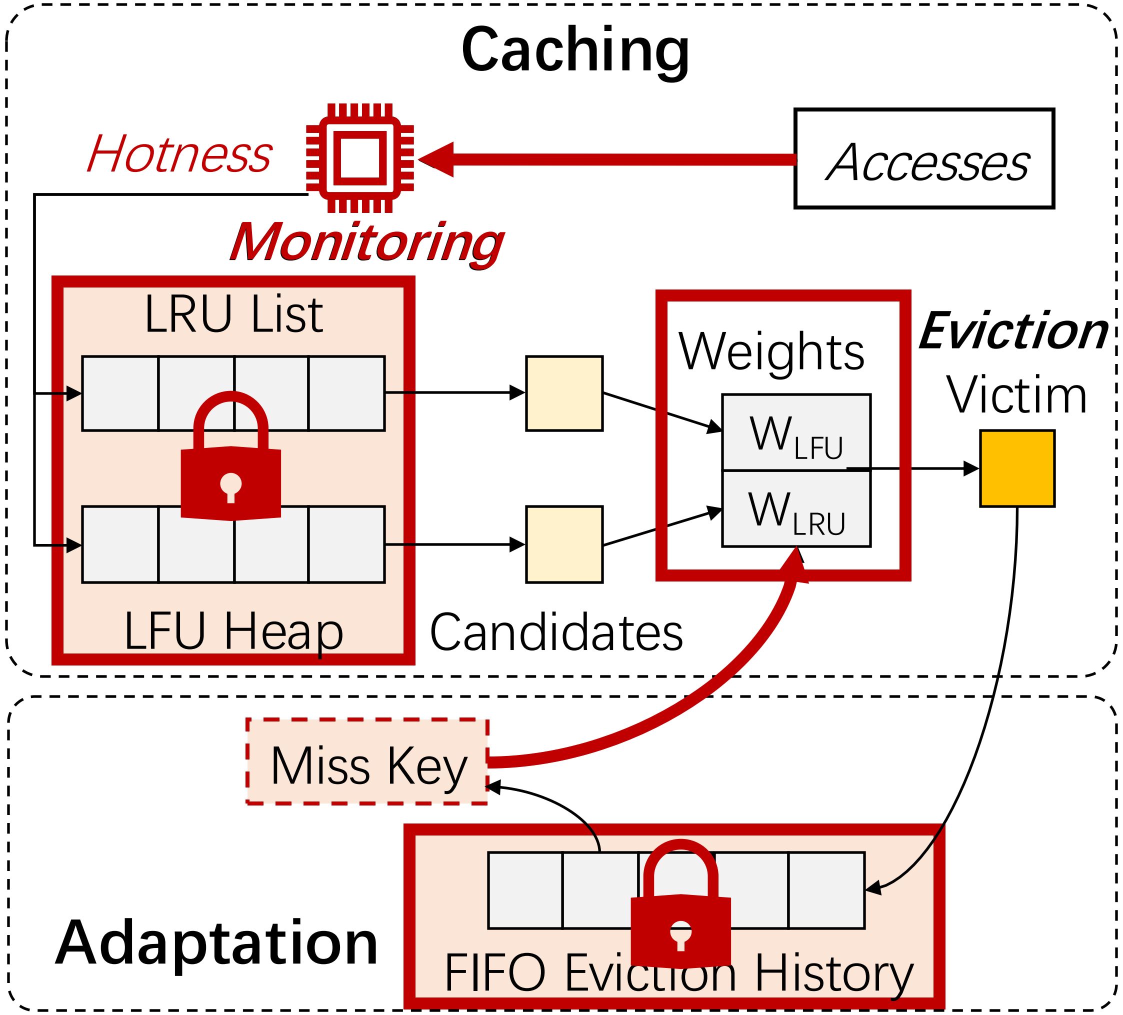


- Monitor object accesses with server CPUs
- Execute multiple caching algorithms as experts in MAB
- Expert weight indicate their performance
- Evict objects according to expert weights

- Record evicted key and the evicting expert in a FIFO eviction history
- Adjust weights with regret minimization

Existing Adaptive Caching Schemes

Model cache eviction as **multi-armed-bandit (MAB)**



☹ ***Asymmetry***

- ➢ Rely on server CPUs to **monitor** object access and **update expert weights**

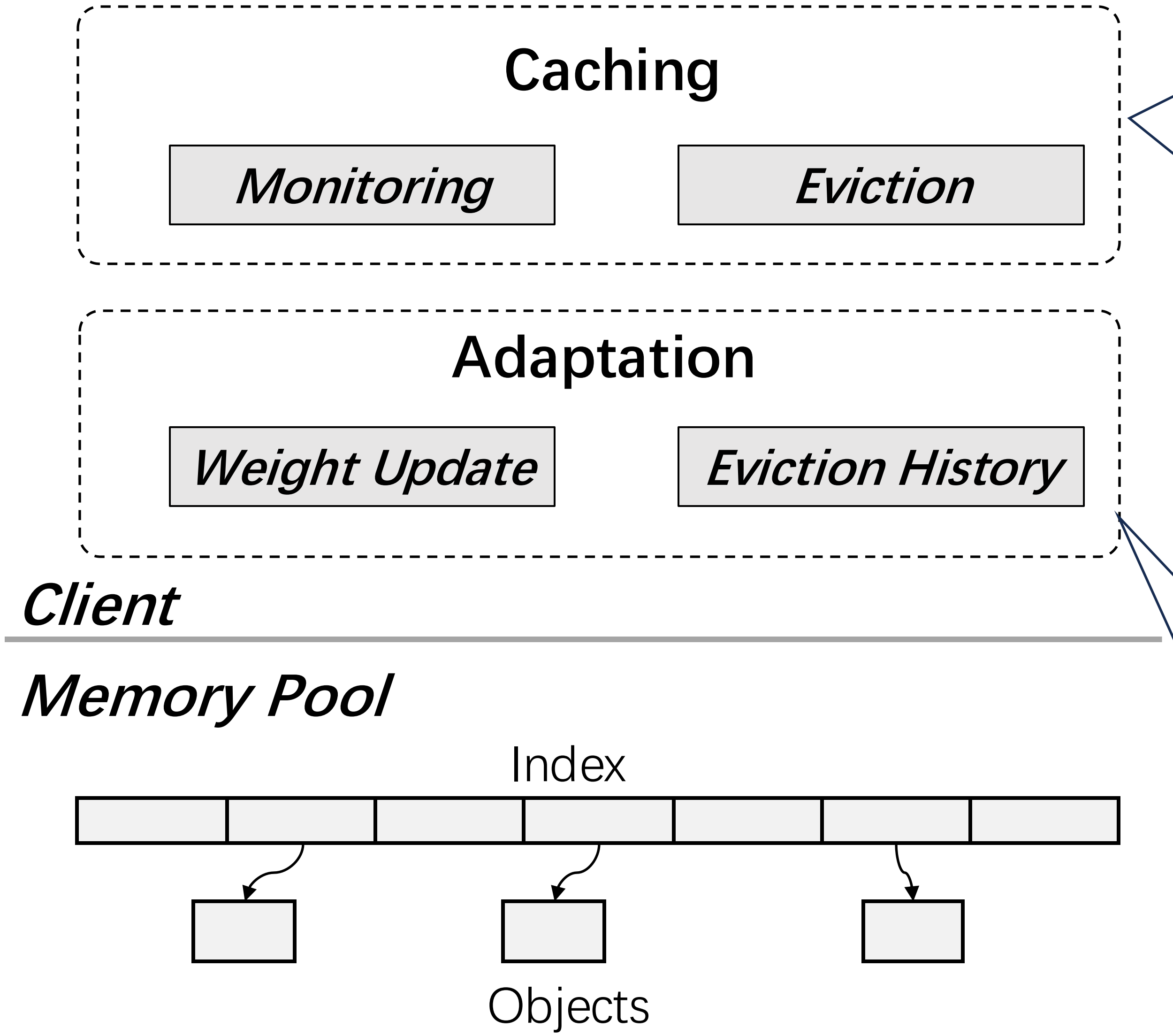
☹ ***Concurrency***

- ➢ **Globally-shared data structures** for experts and eviction history

☹ ***I/O amplifications***

- ➢ Multiple numbers of I/Os to maintain **various data structures**

Adaptive Caching Framework on DM



Client-Centric Caching Framework

☺ Asymmetry:
- ➢ Distributed access monitoring

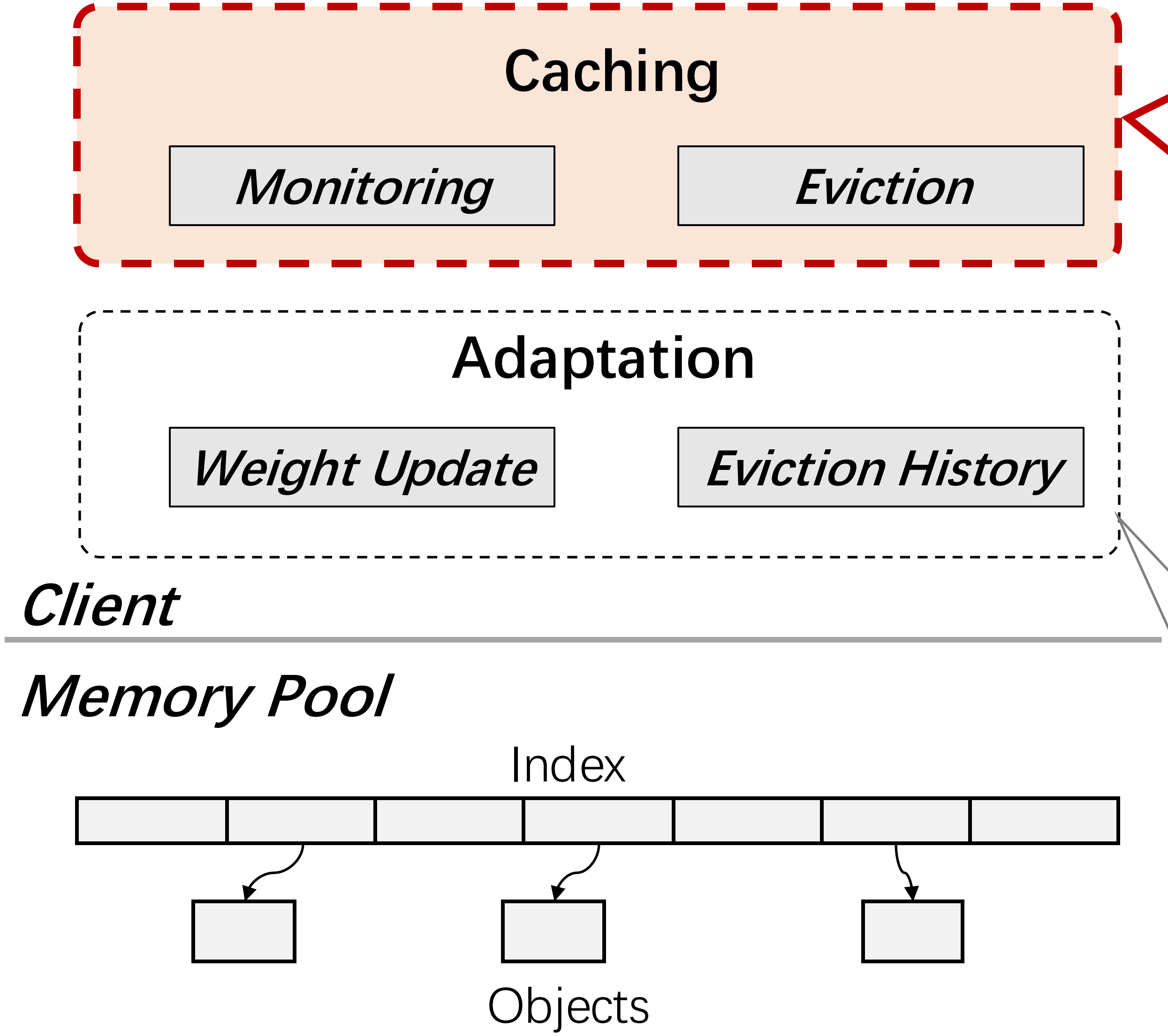
☺ Concurrency & I/O:
- ➢ Sample-based eviction

Distributed Adaptive Caching

☺ Asymmetry:
- ➢ Lazy weight update algorithm

☺ Concurrency & I/O:
- ➢ Lightweight eviction history

Adaptive Caching Framework on DM



Client-Centric Caching Framework

☺ Asymmetry:
- Distributed access monitoring

☺ Concurrency & I/O:
- Sample-based eviction

Distributed Adaptive Caching

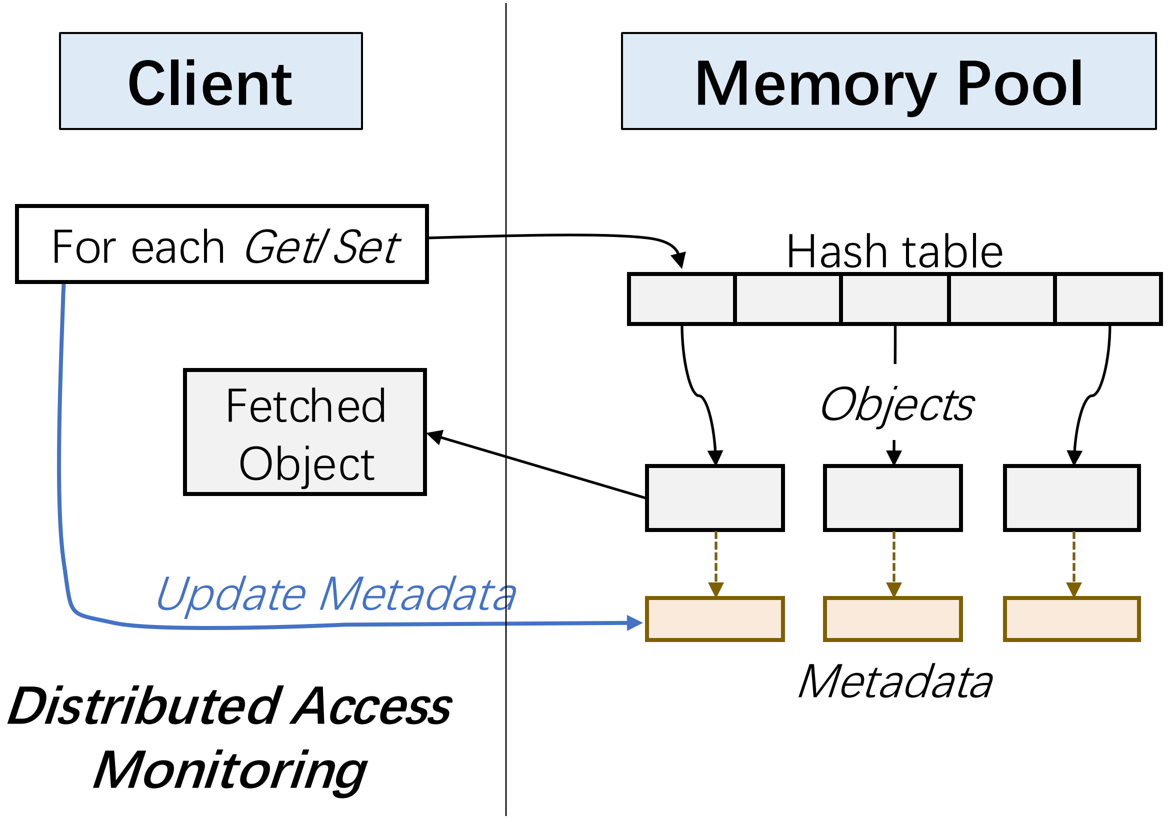
☺ Asymmetry:
- Lazy weight update algorithm

☺ Concurrency & I/O:
- Lightweight eviction history

The Client-Centric Caching Framework

Key idea: <u>Distributed Access Monitoring</u> + Sample-Based Eviction

☺ **Asymmetry**

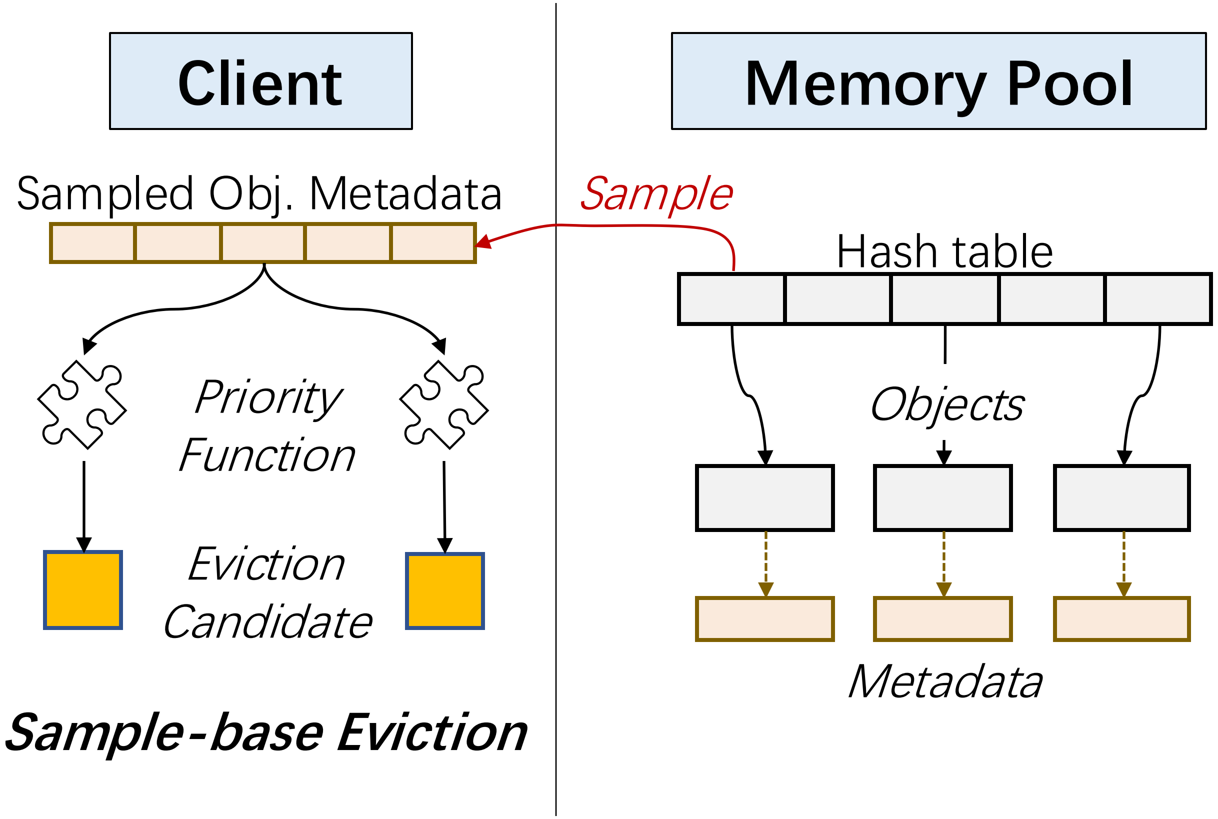


- ➢ Record access information in a per-object metadata
  - Timestamp, Frequency, etc.
- ➢ Update the metadata with one-sided RDMA verbs after each data access

The Client-Centric Caching Framework

Key idea: Distributed Access Monitoring + **Sample-Based Eviction**

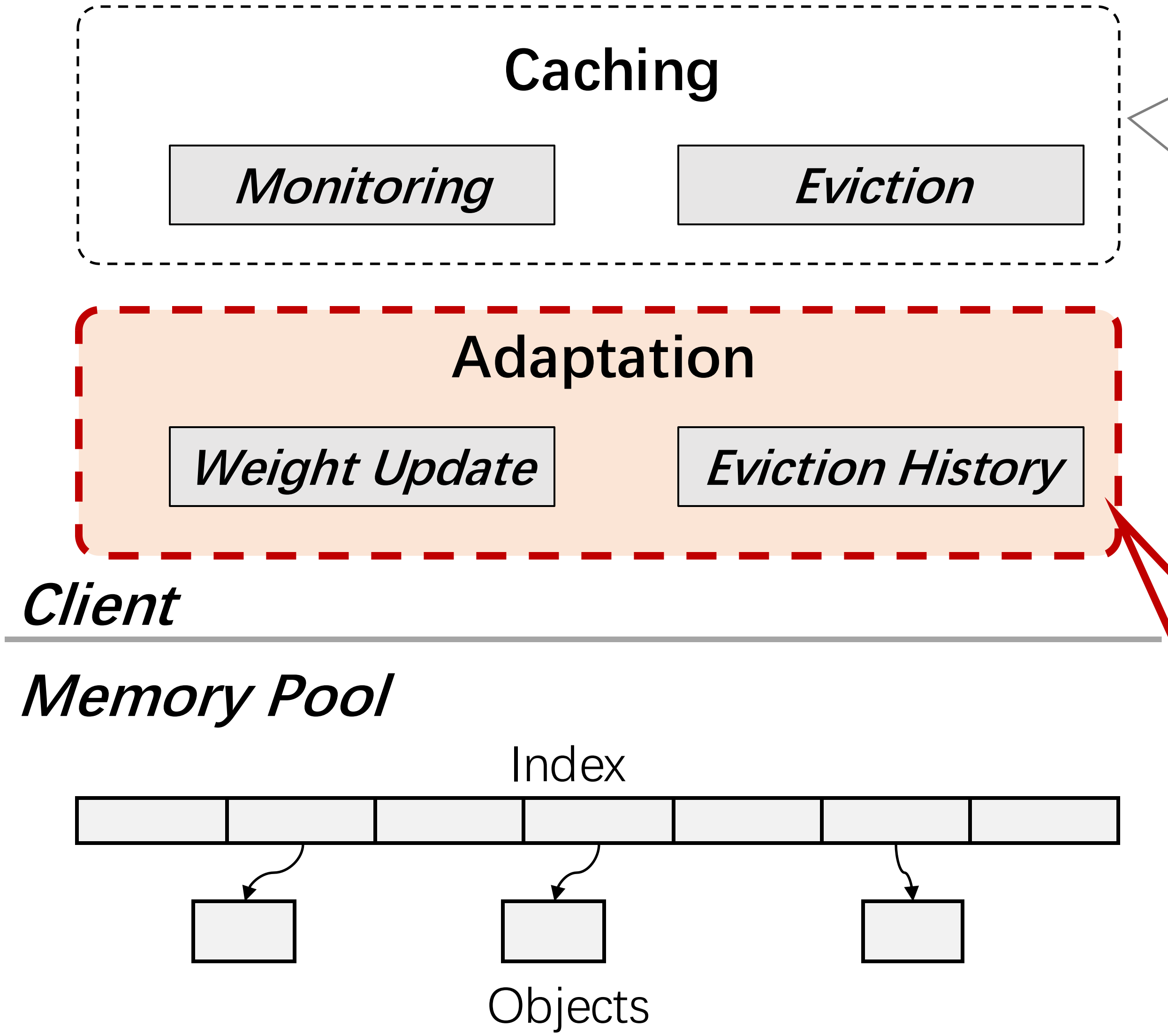
☺ **Concurrency & I/O**



- Approximate the precise execution with sampling
- Sample multiple object metadata
- Map metadata to object hotness with priority functions
  - E.g., LRU: last access timestamp
- No need to maintain caching data structures

Various caching algorithms can be integrated by defining different priority functions

Adaptive Caching Framework on DM



Client-Centric Caching Framework

<u>Asymmetry:</u>
- Distributed access monitoring

<u>Concurrency & I/O:</u>
- Sample-based eviction

Distributed Adaptive Caching

<u>Asymmetry:</u>
- Lazy weight update algorithm

<u>Concurrency & I/O:</u>
- Lightweight eviction history

Distributed Adaptive Caching

Design: **Lightweight Eviction History** + Lazy Weight Update

☺ **Concurrency & I/O**

Key Idea: ***In-Cache History Entries*** + Logical FIFO Queue

Sample-Friendly Hash Table

Hist. Entry	Hist. Entry	Slot	Hist. Entry

✓ No additional index required
✓ Memory space saved for history entries

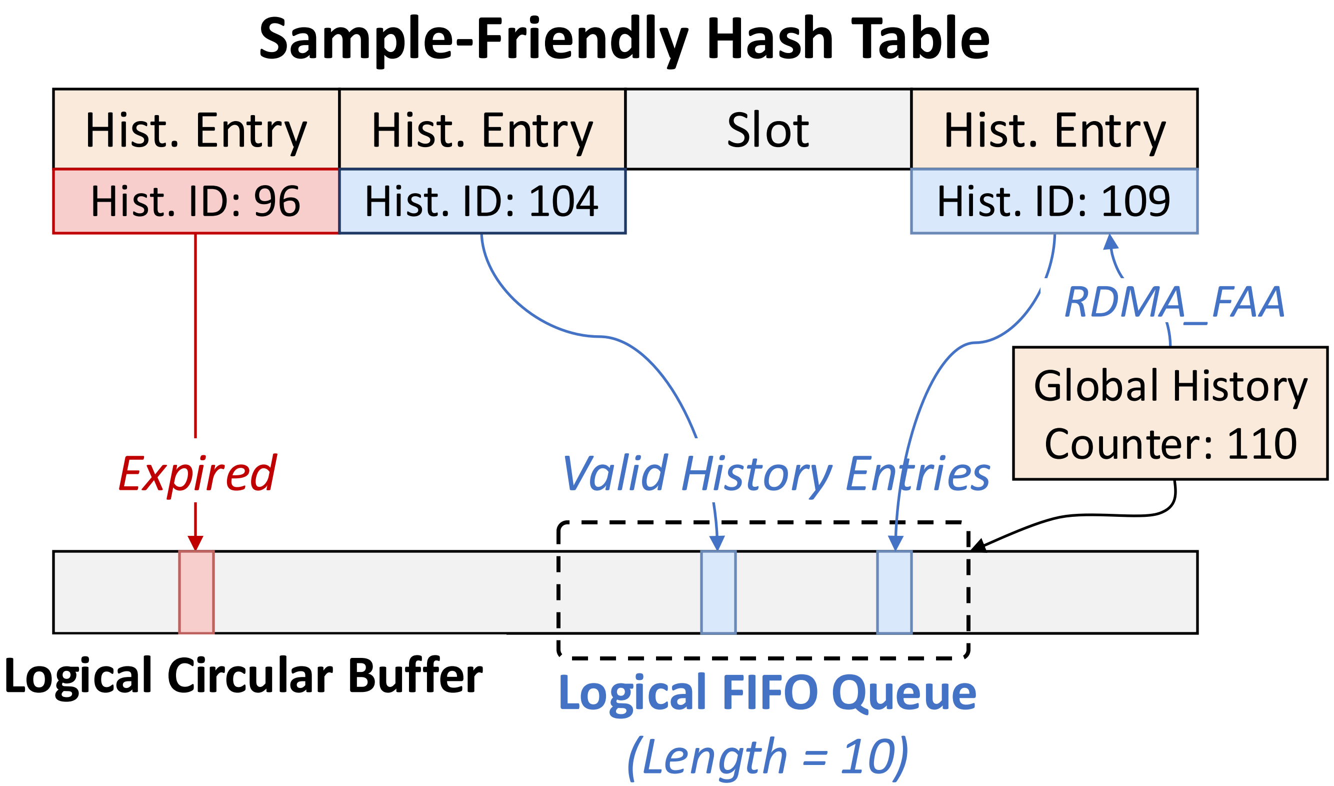
➢ Reuse hash table slots to store history entries

Distributed Adaptive Caching

Design: **Lightweight Eviction History** + Lazy Weight Update

☺ **Concurrency & I/O**

Key Idea: In-Cache History Entries + ***Logical FIFO Queue***

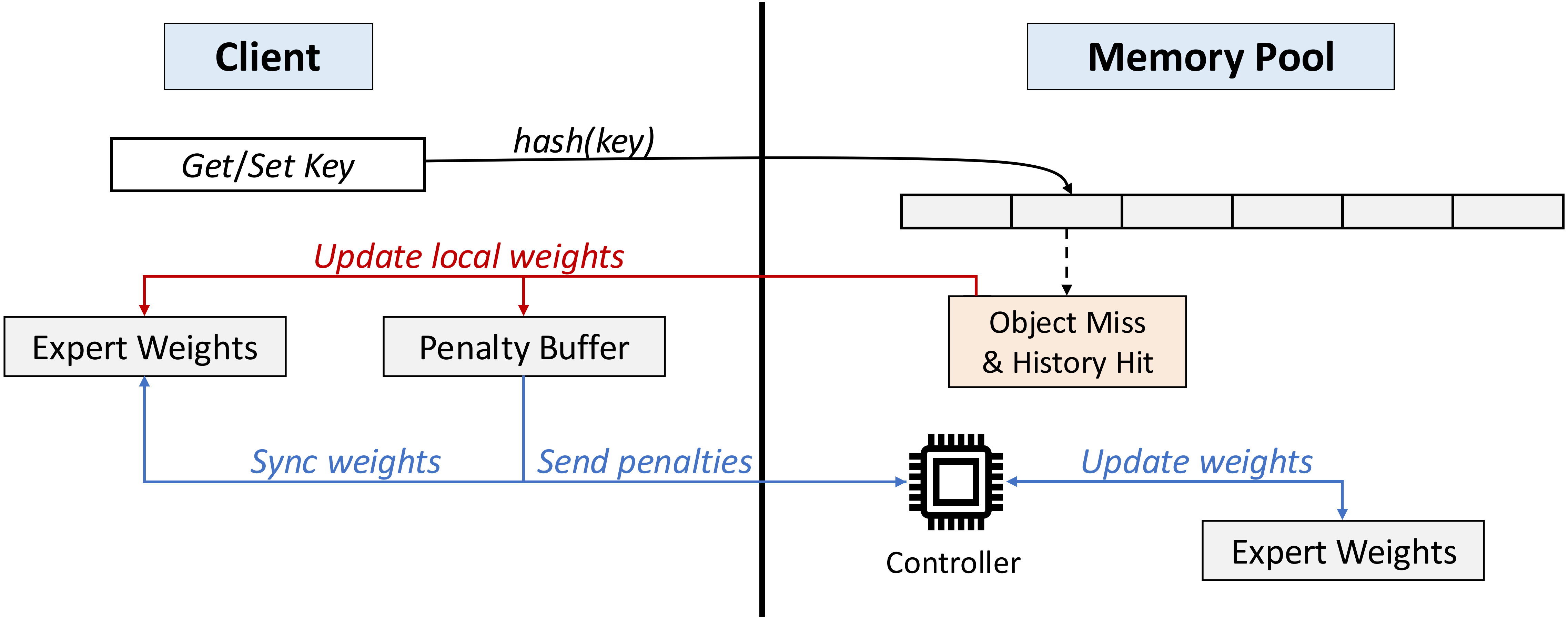


- Reuse hash table slots to store history entries
- Achieve FIFO with computation without maintaining a queue

Distributed Adaptive Caching

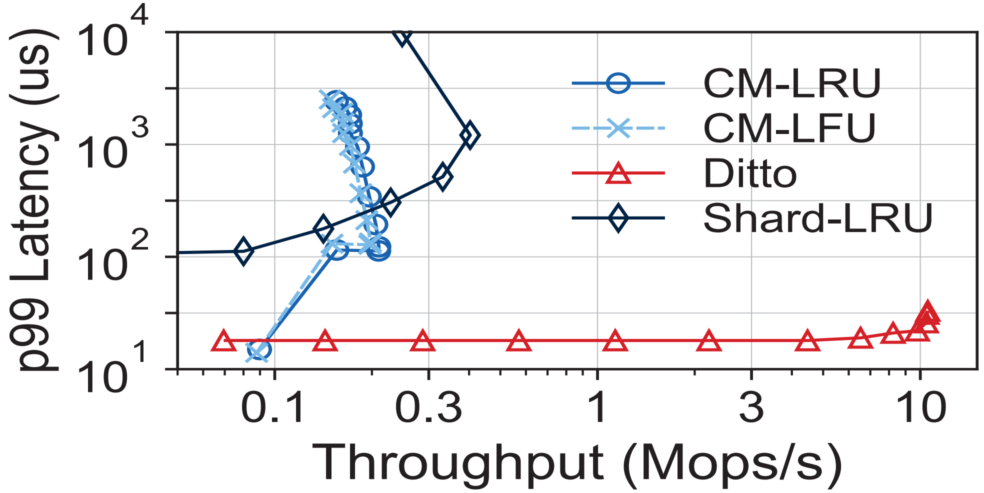
Design: Lightweight Eviction History + Lazy Weight Update

☺ Asymmetry

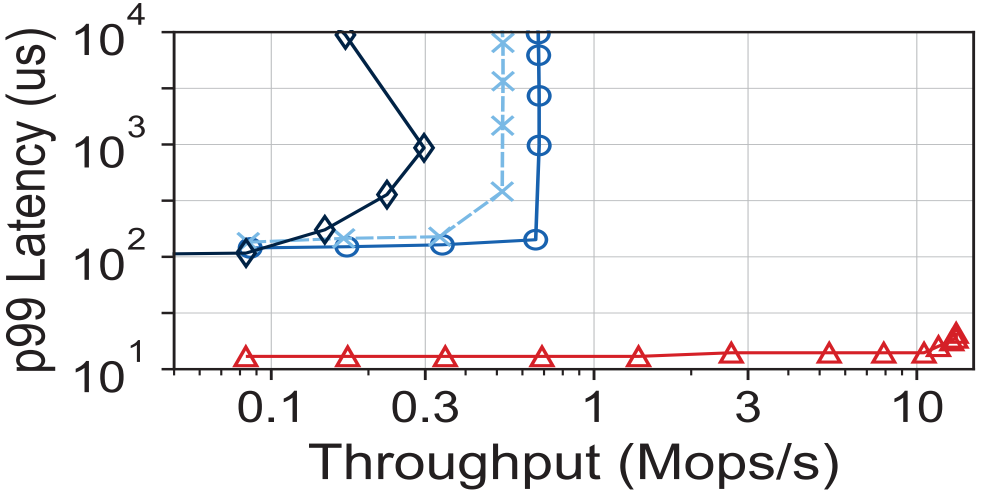


Evaluation – Overall Performance

YCSB-A
(50% UPDATE, 50% SEARCH)



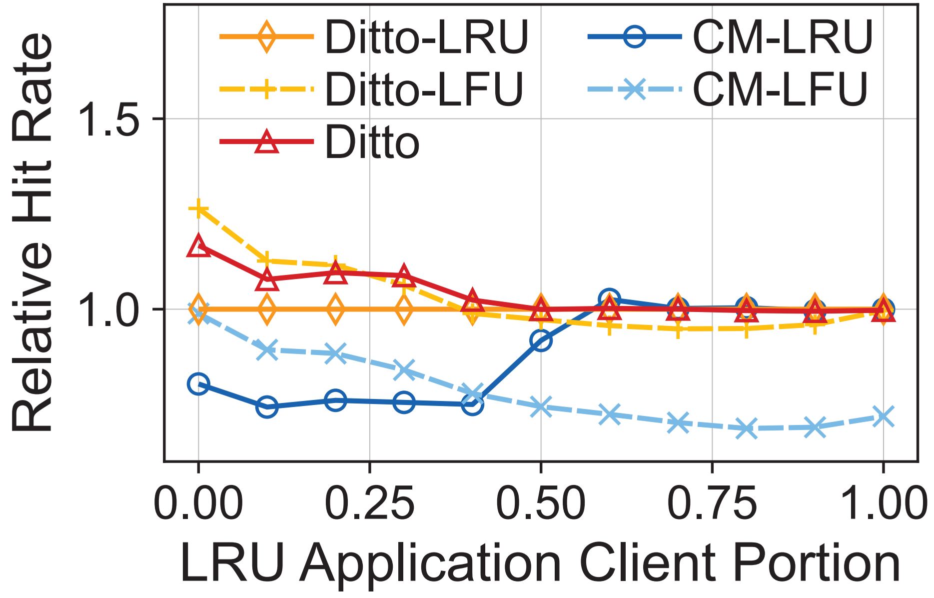
YCSB-C
(100% SEARCH)



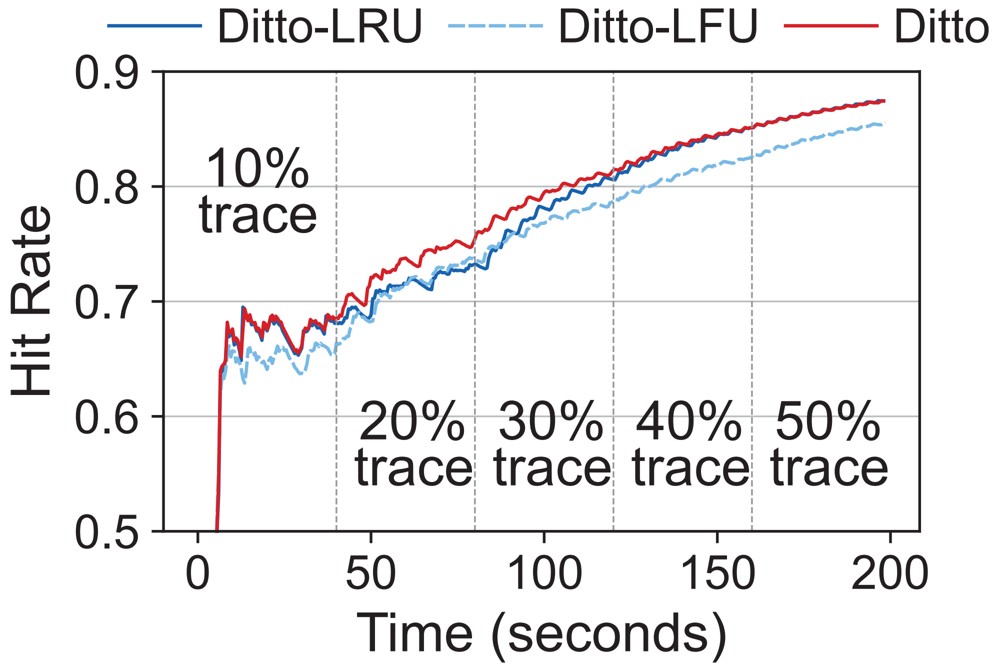
Ditto reaches up to 9x higher throughput than CliqueMap on DM

Evaluation – Hit Rates

Cache Hit Rates under Changing Compute Resources

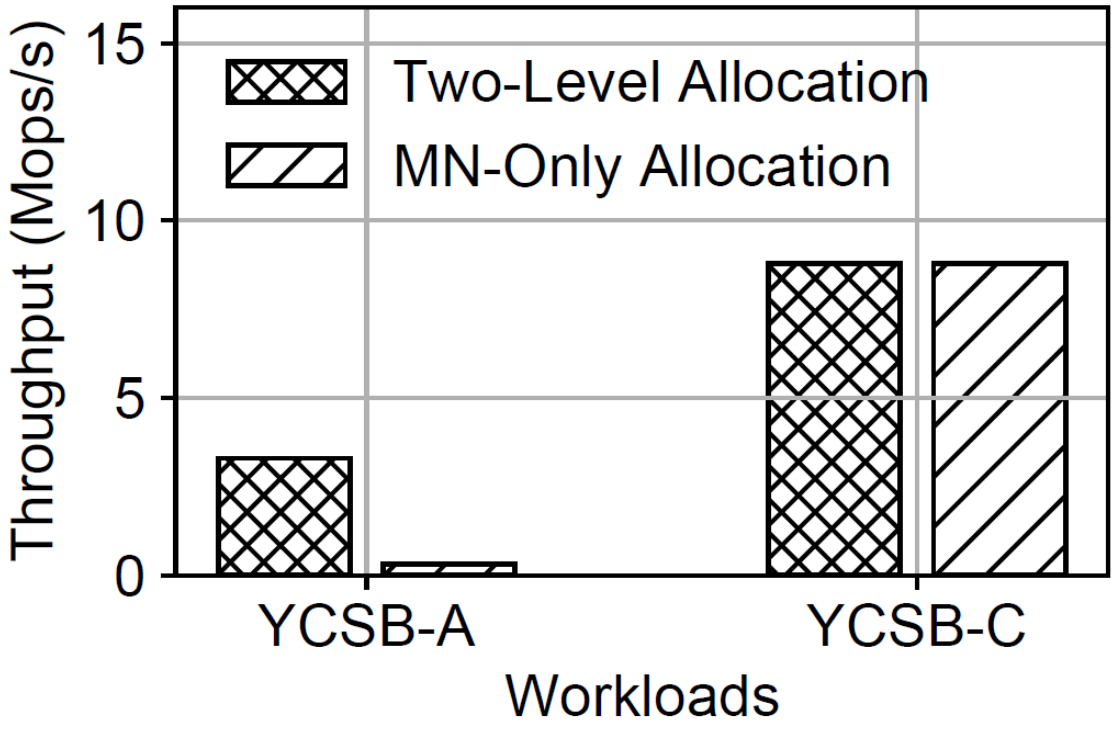


Cache Hit Rates under Changing Memory Resources



Ditto approaches the better one of LRU and LFU and can exceed both under changing workloads

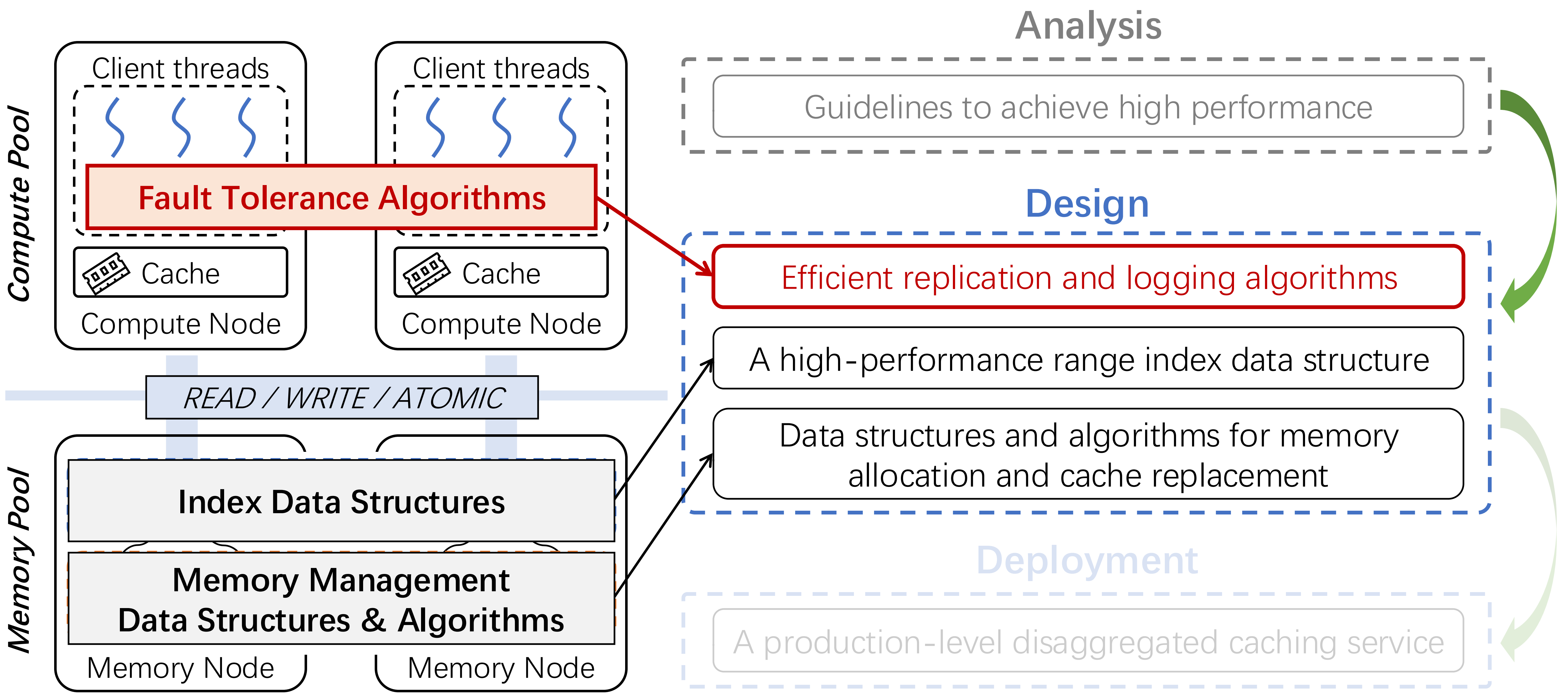
Evaluation – Memory Management



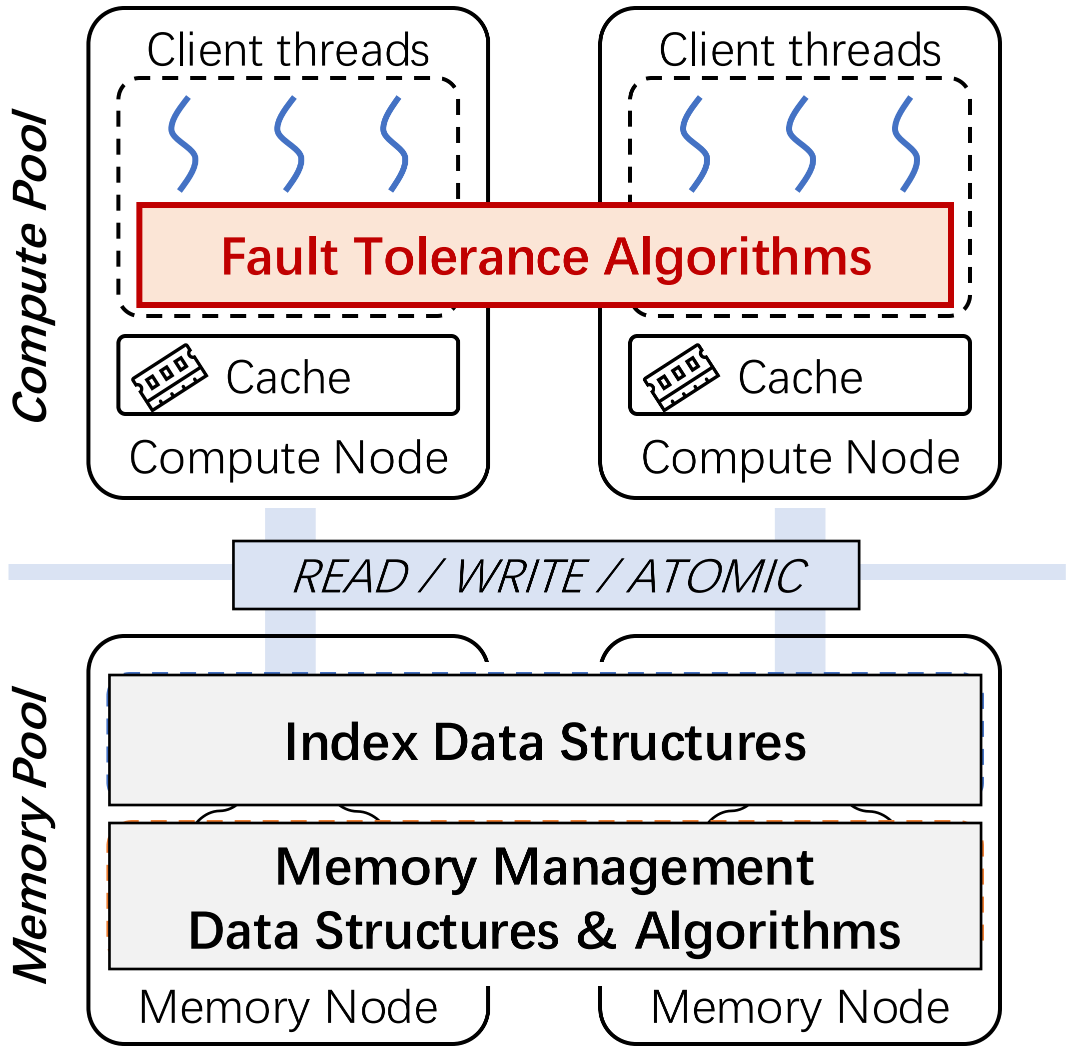
- Two-level memory allocation performs significantly better on write-intensive workloads due to the optimized **asymmetry**

- The performance is similar on read-only workloads since there are no allocation on the critical path

Thesis Contributions



Thesis Contributions

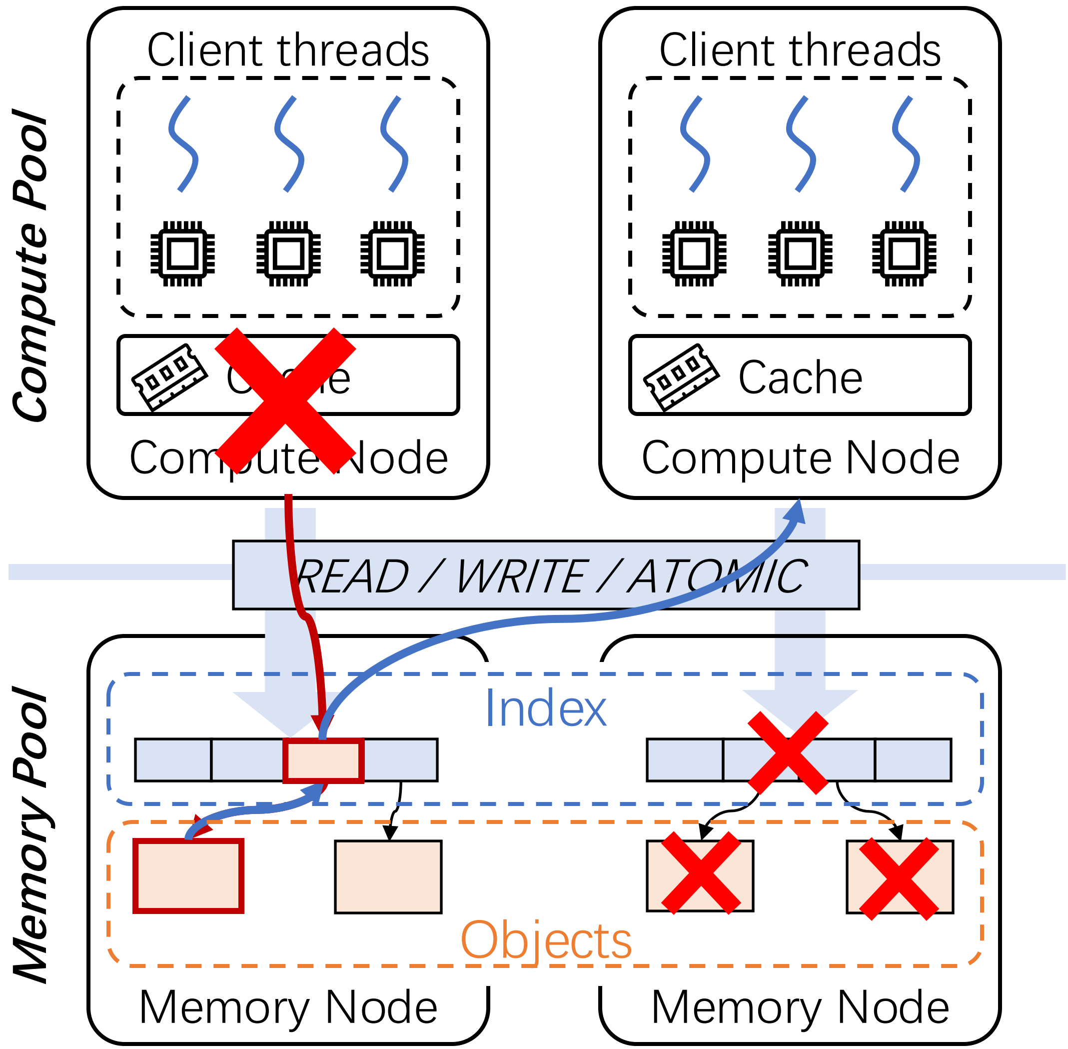


Goal:

- ➢ Do not return wrong data
- ➢ Do not lose data

What algorithms do we need?

The Failure Model



Compute Node Failures

- Data corruption

Akin to *file system inconsistency*

Algorithm: *write-ahead logging*

Memory Node Failures

- Data loss

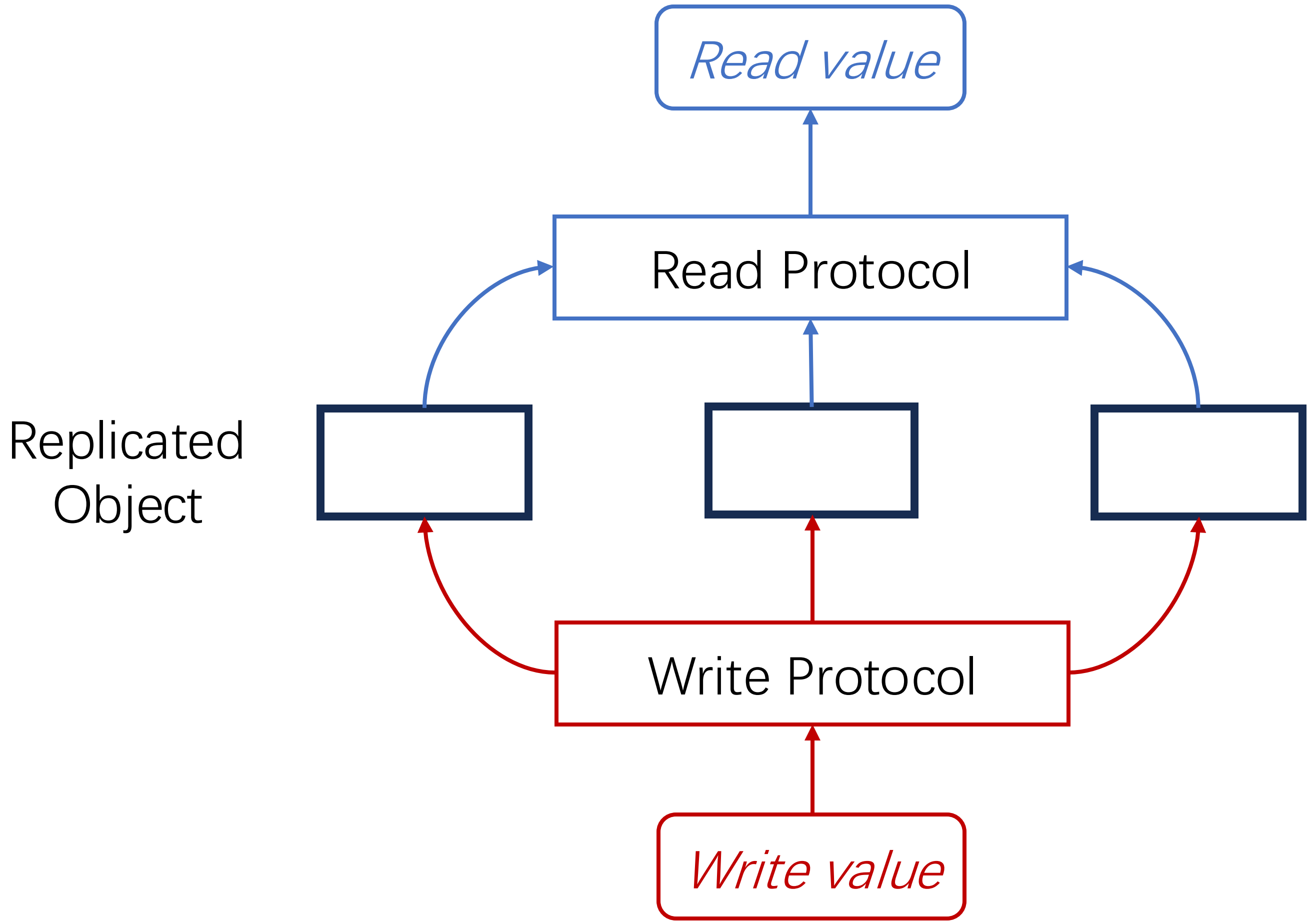
Akin to *node failures in distributed storage*

Algorithm: *replication*

Existing algorithms for logging and replication are not suitable!

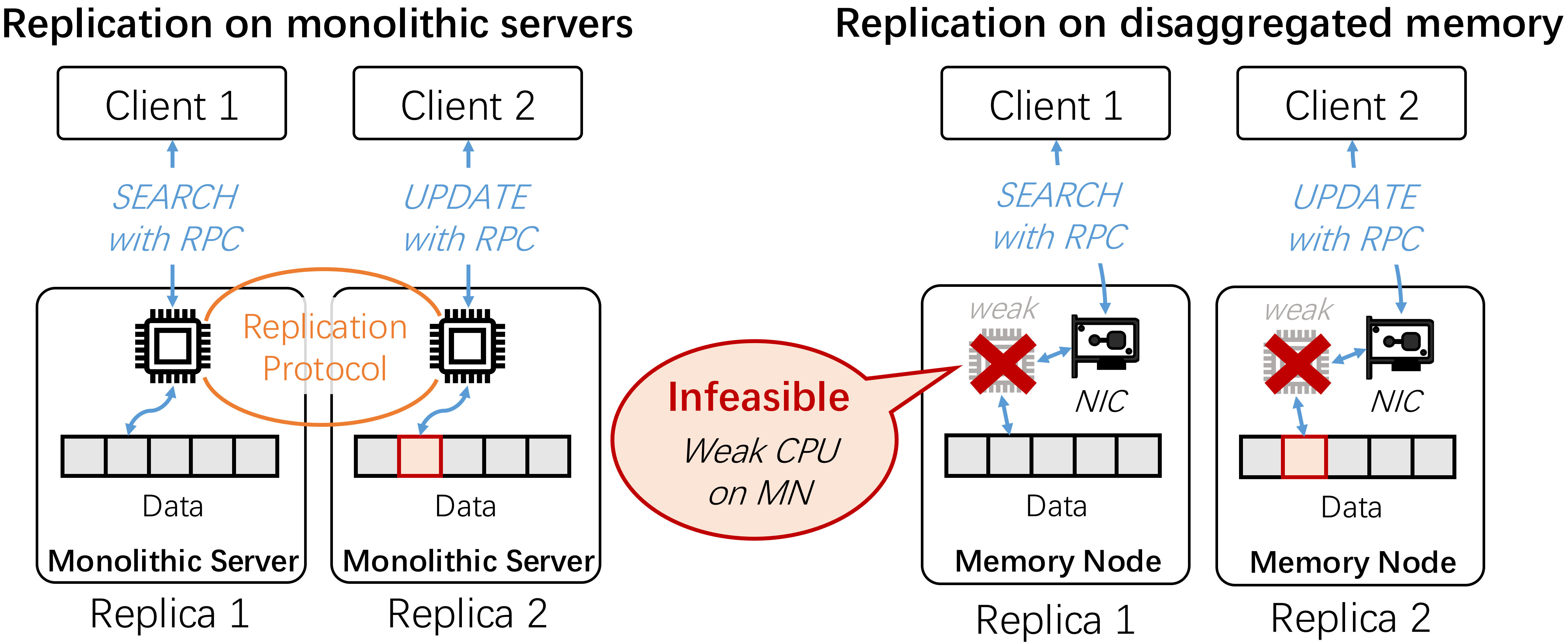
Replication protocols

How to replicate data with strong consistency?



- The replicated object should behave as if there is only one single object
- Read value always return the most recent write value

Existing replication protocols are symmetric

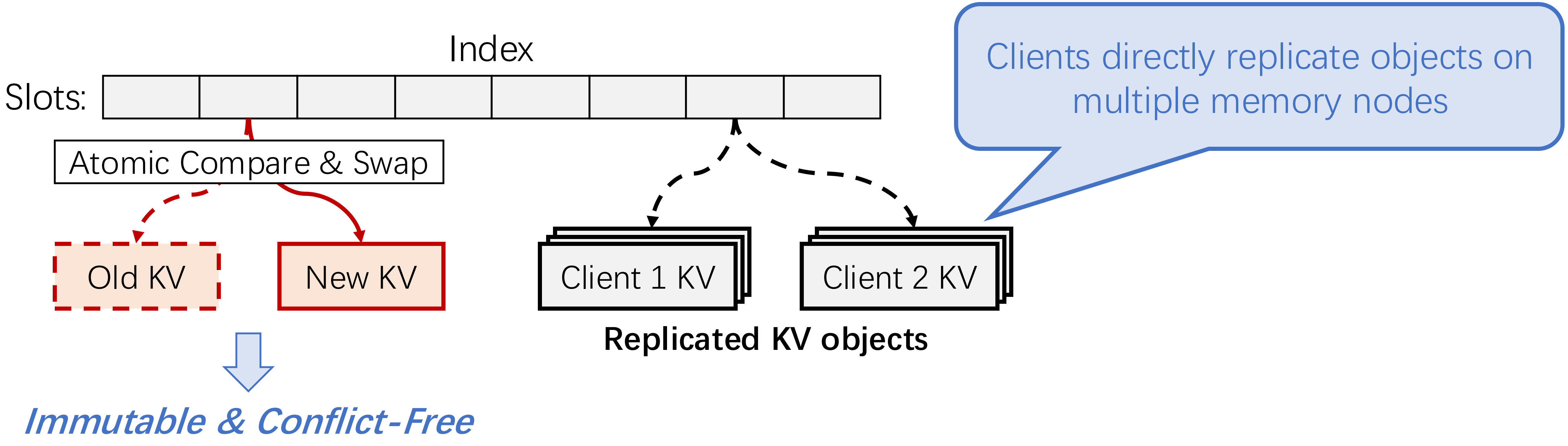


The Client-Centric Replication Protocol

Replication protocols are required for **mutable shared data**

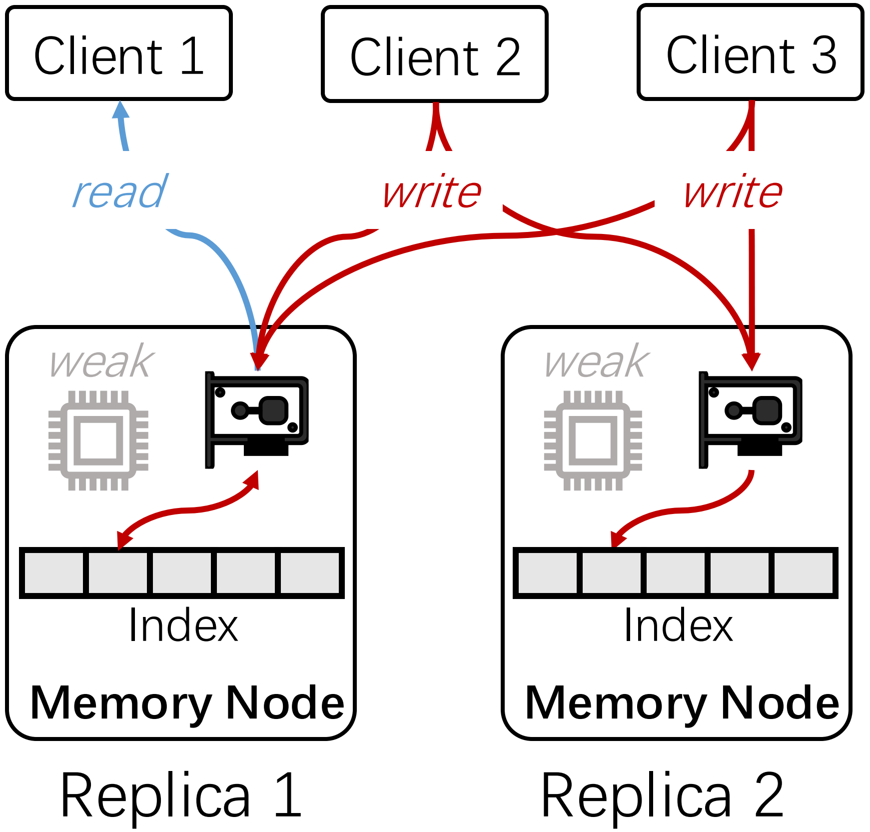
Reduce mutable shared data with the index

- Hash index stores addresses of KV objects
- Update KV objects in an out-of-place manner



The Client-Centric Replication Protocol

How to deal with the **mutable shared hash index**?



Two key problems:

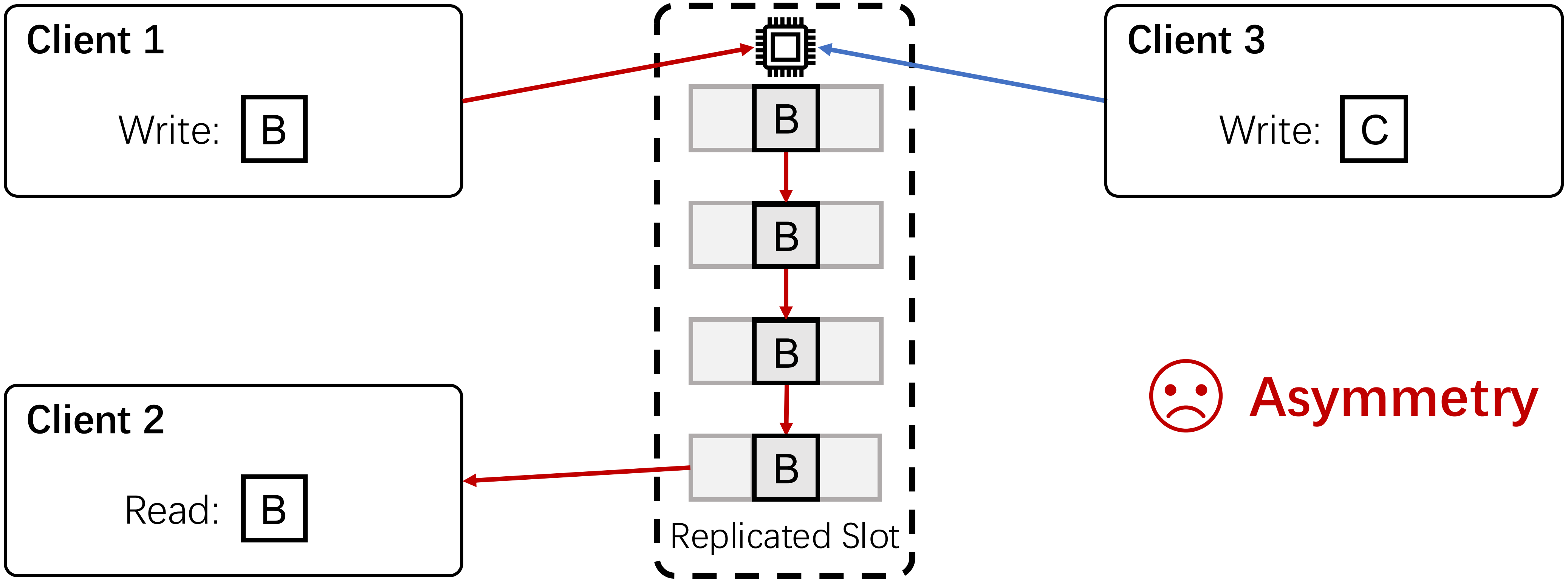
- How to protect readers from reading incomplete writes?
- How to efficiently resolve write-write conflicts among clients?

Replicate the shared index data structure

First attempt: Chain Replication

On monolithic servers:

- Write value flows through the chain of replicated nodes
- Read from the tail of the chain
- Rely on CPUs on the head of the chain to resolve conflicts



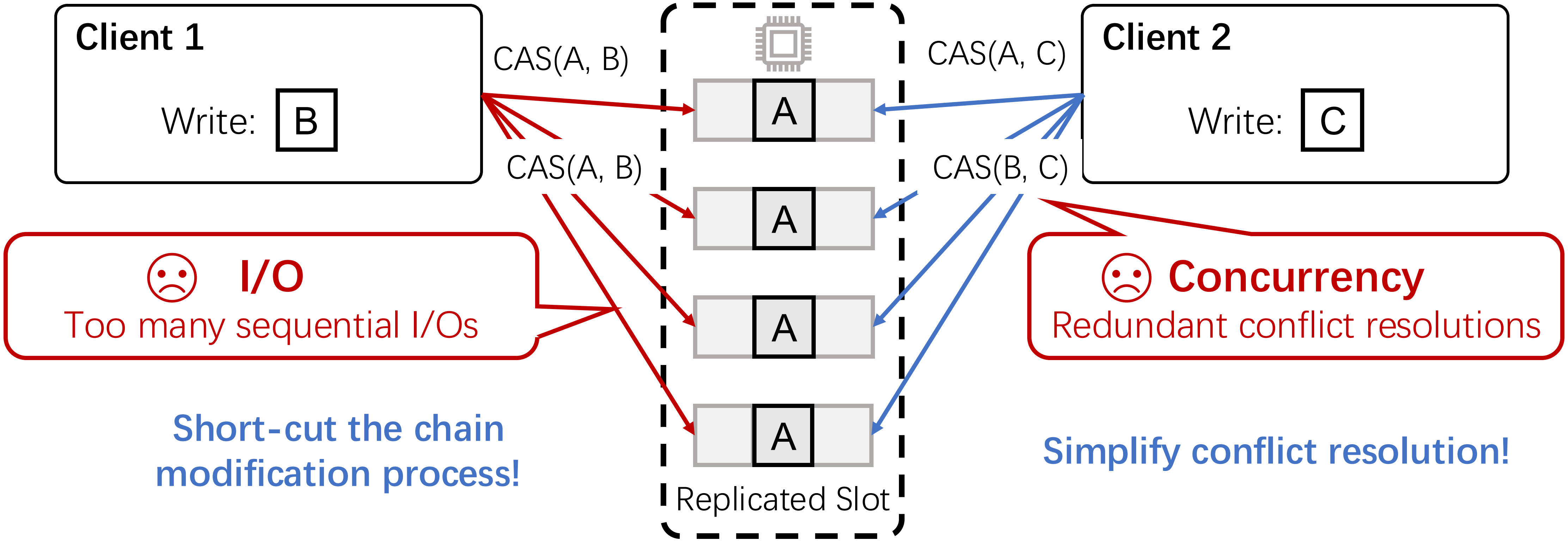
Replicate the shared index data structure

First attempt: make chain replication *asymmetric*

☺

On disaggregated memory:

- Use CAS to decide the write order on the head of the chain
- Sequentially use CAS to modify all replicated slots

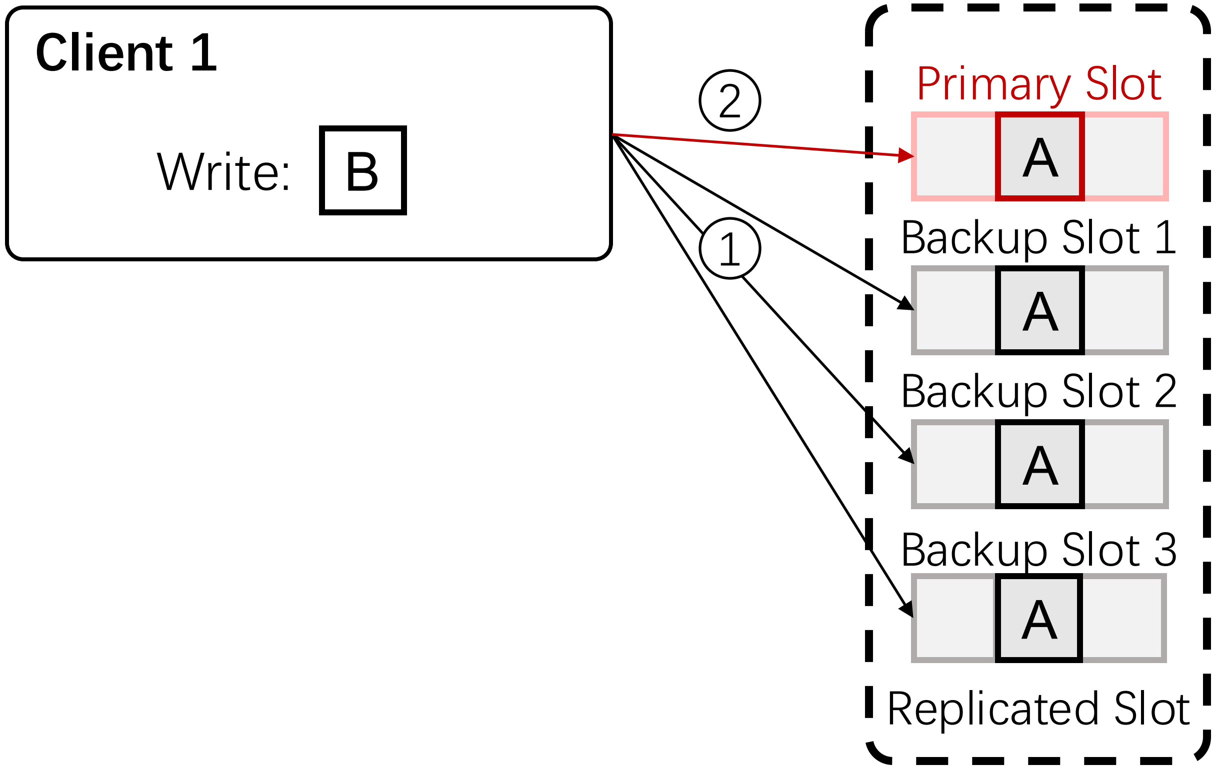


Replicate the shared index data structure

Our attempt eliminates the sequential I/O

Short cut the sequential chain modification

- Separate index replicas into a single primary and multiple backups
- Resolve conflicts in all backups and then modify primary

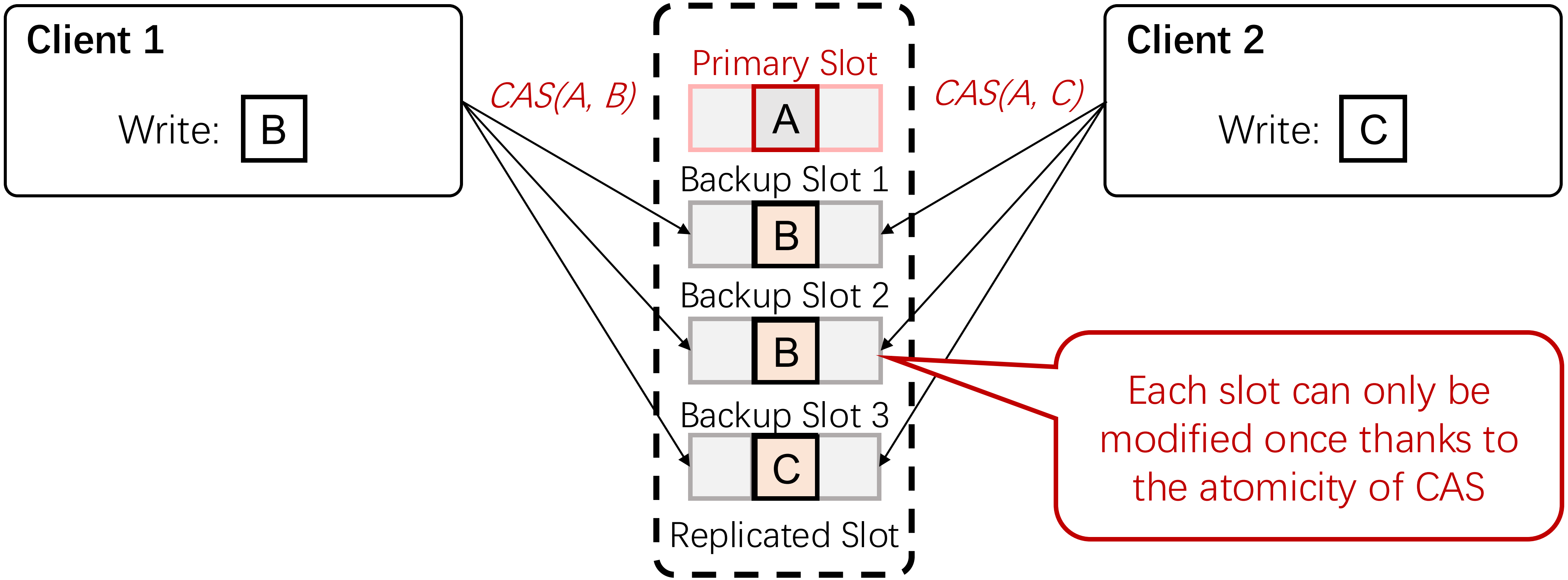


Replicate the shared index data structure

Our attempt improves concurrency

Simplify the conflict resolution process

- Out-of-place KV modification ⇨ Conflict clients write different values
- Last-writer-wins conflict resolution ⇨ Simple rules on client sides

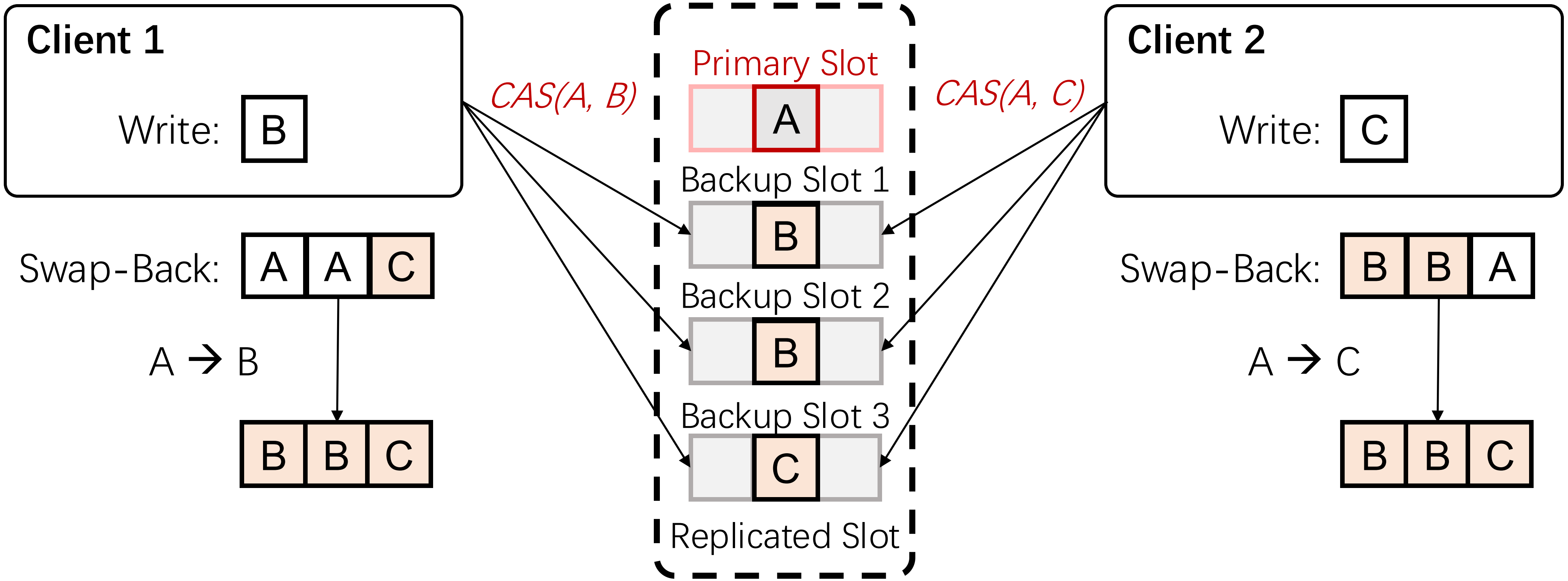


Replicate the shared index data structure

Our attempt improves concurrency

A client is a last writer if:

- Successfully modified all backup slots
- Or, modified a majority of backup slots
- Or, wrote the smallest value when there is no majority

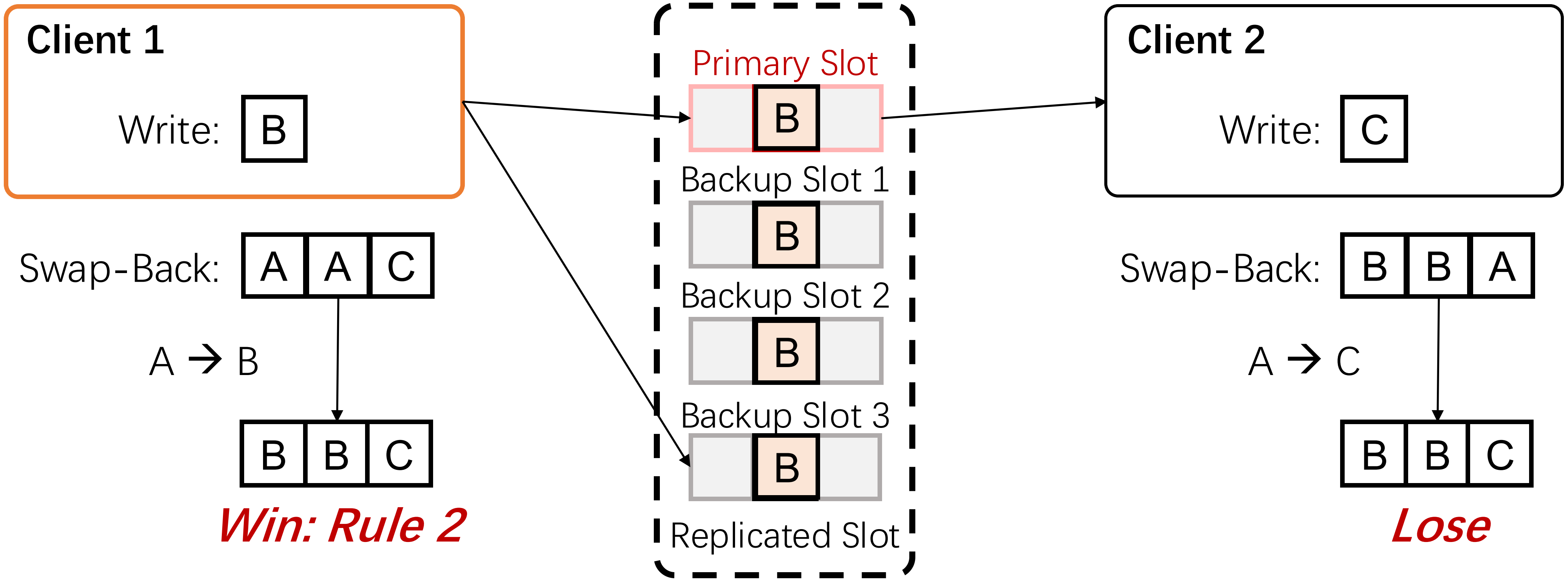


Replicate the shared index data structure

Our attempt improves concurrency

A client is a last writer if:

- Successfully modified all backup slots
- ***Or, modified a majority of backup slots***
- Or, wrote the smallest value when there is no majority

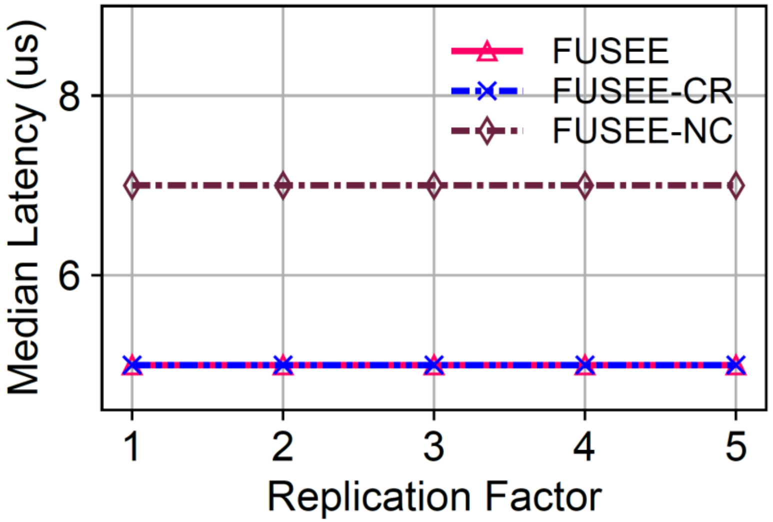


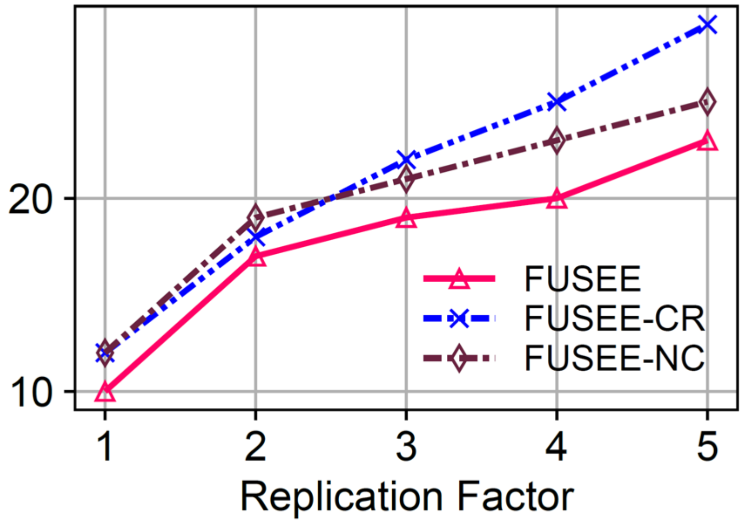
Replicate the shared index data structure

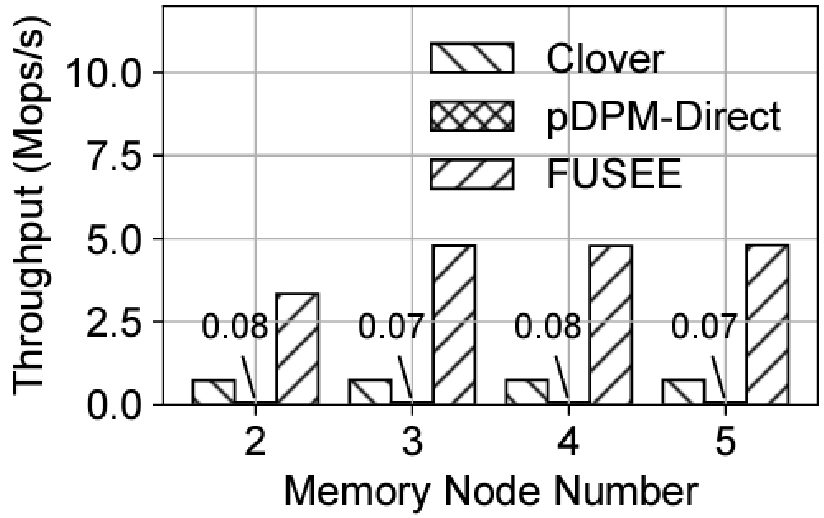
Our attempt optimized for Asymmetry, I/O, and Concurrency

Bounded latency and high throughput for replication:

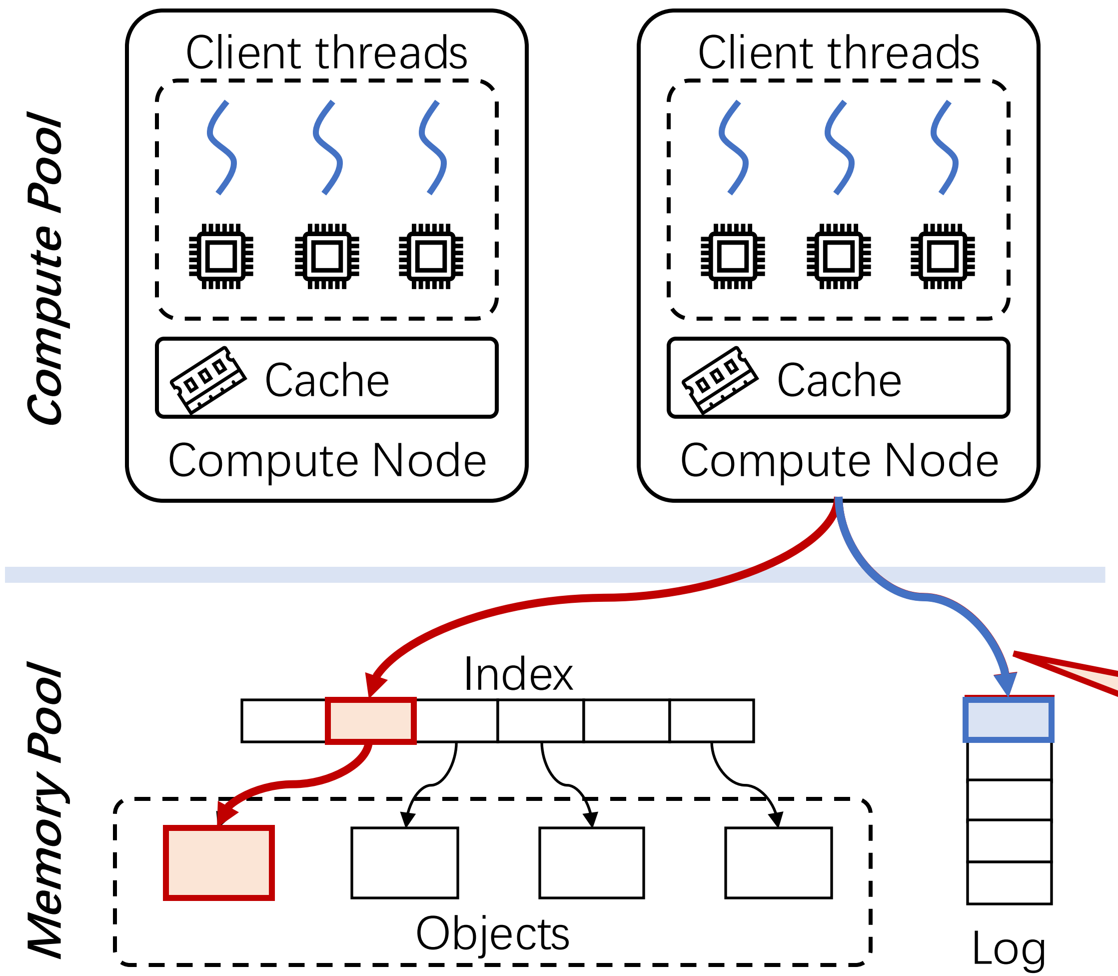
- *1 RTT for reads*
- *2~4 RTTs for writes*
- *4x high throughput*







Write-Ahead Logging



Log records operation sequence

- ➢ Create a log entry before modifying data
- ➢ Commit the log entry after modifying data

☺ **Asymmetry**

No remote CPU intervention

☺ **Concurrency**

Client logs are independent

☹ **I/O**

Two additional I/Os on the critical path

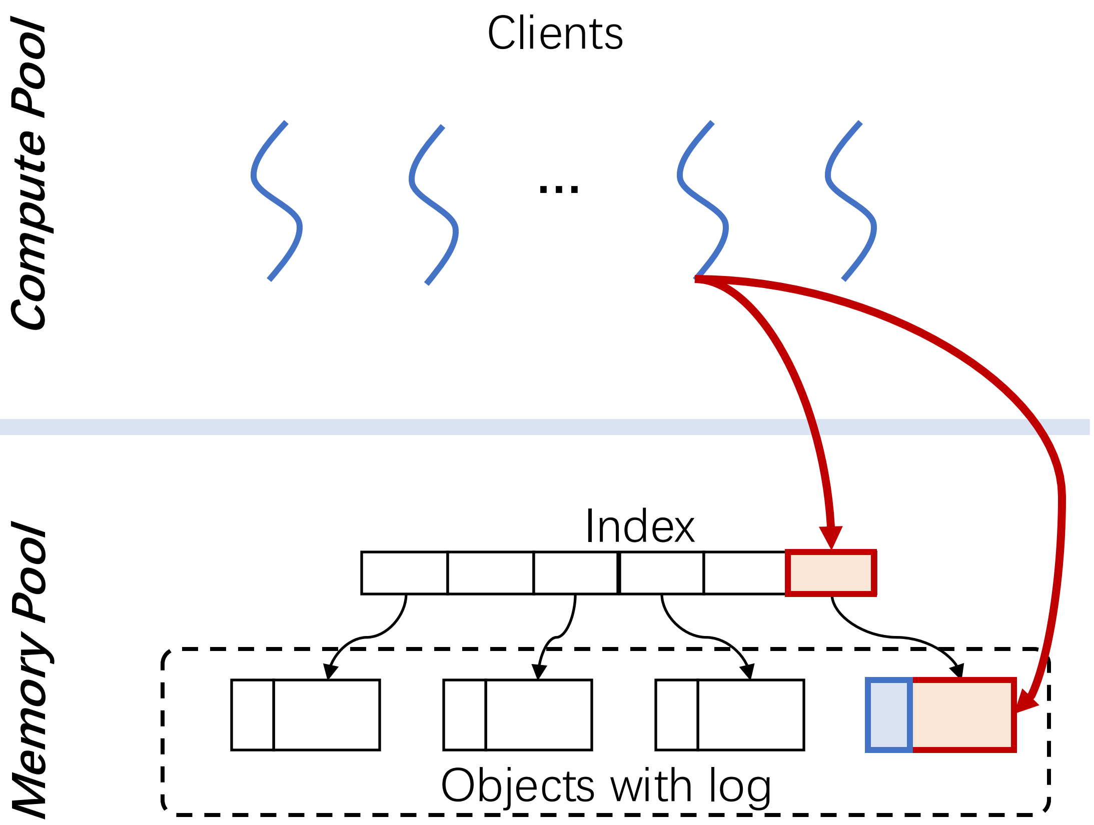
Reduce the I/O overhead for logging

Log-Structured File Systems

- Store data as a sequential log
- Best leverage the sequential write performance of disks

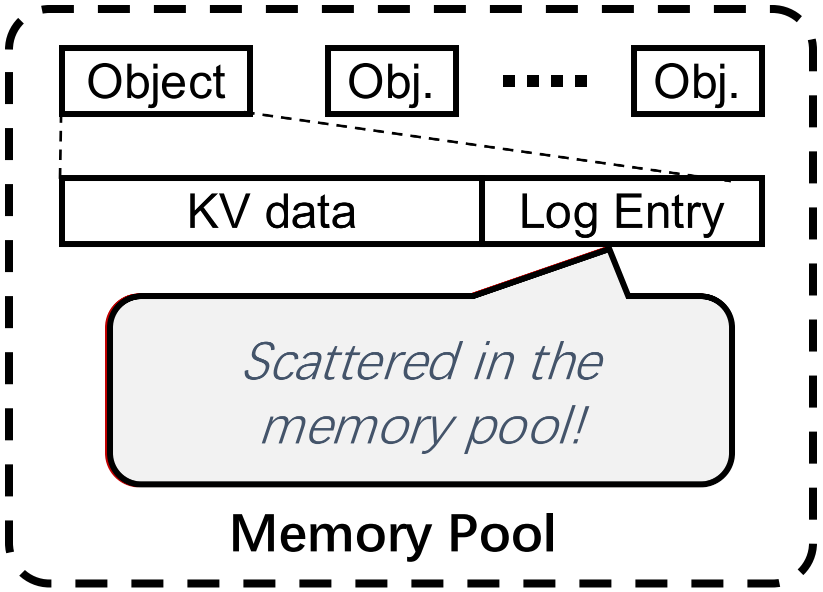
Can we store logs with data to reduce the I/O numbers?

If we can store logs in data…



The Embedded Operation Log

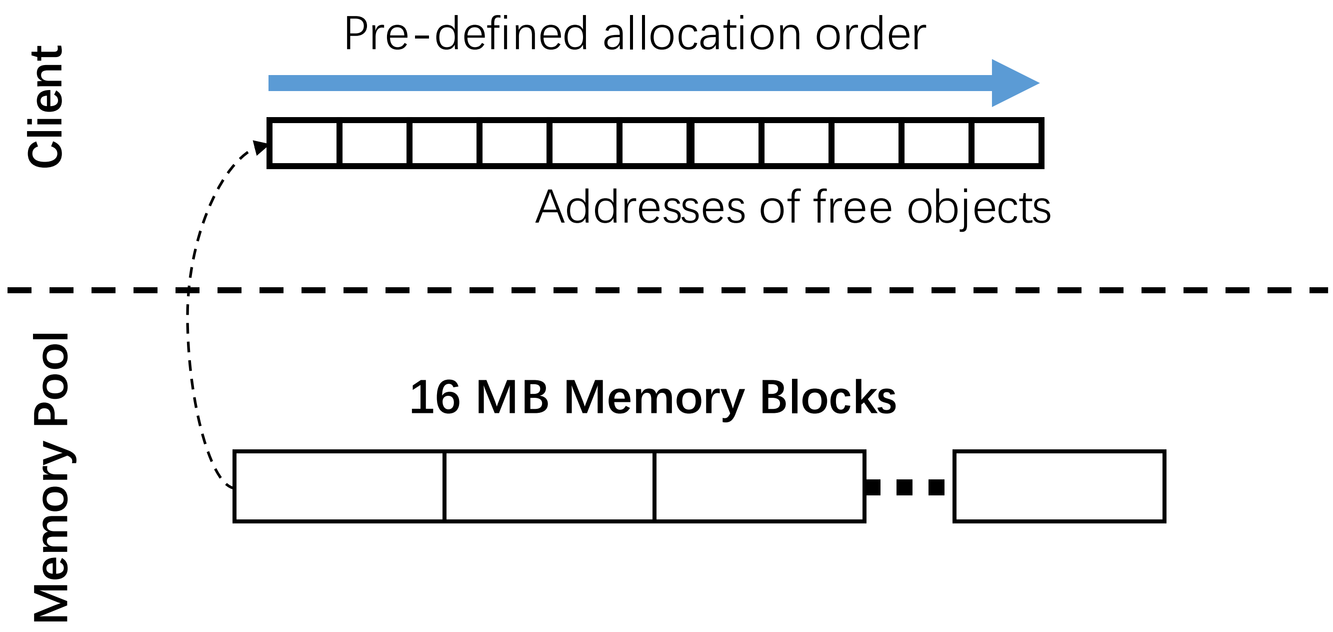
Key Idea: Embed log entries into KV objects to eliminate the log creation I/O



How to construct operation sequence?

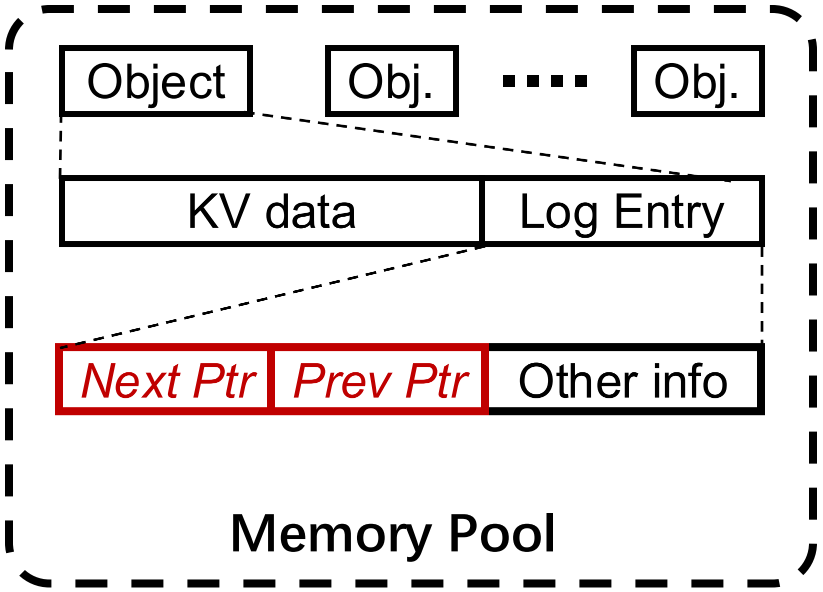
Operations that modifies data always allocate a memory block before

➢ ***Use memory allocation order!***



The Embedded Operation Log

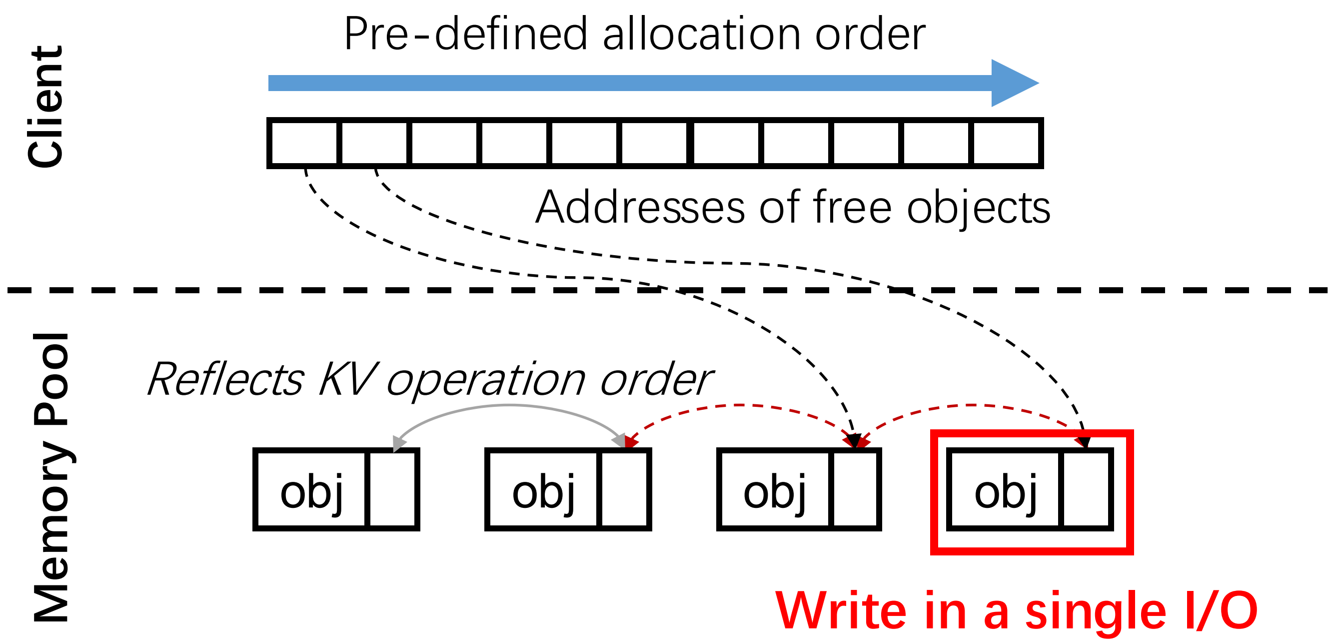
Key Idea: Embed log entries into KV objects to eliminate the log creation I/O



How to construct operation sequence?

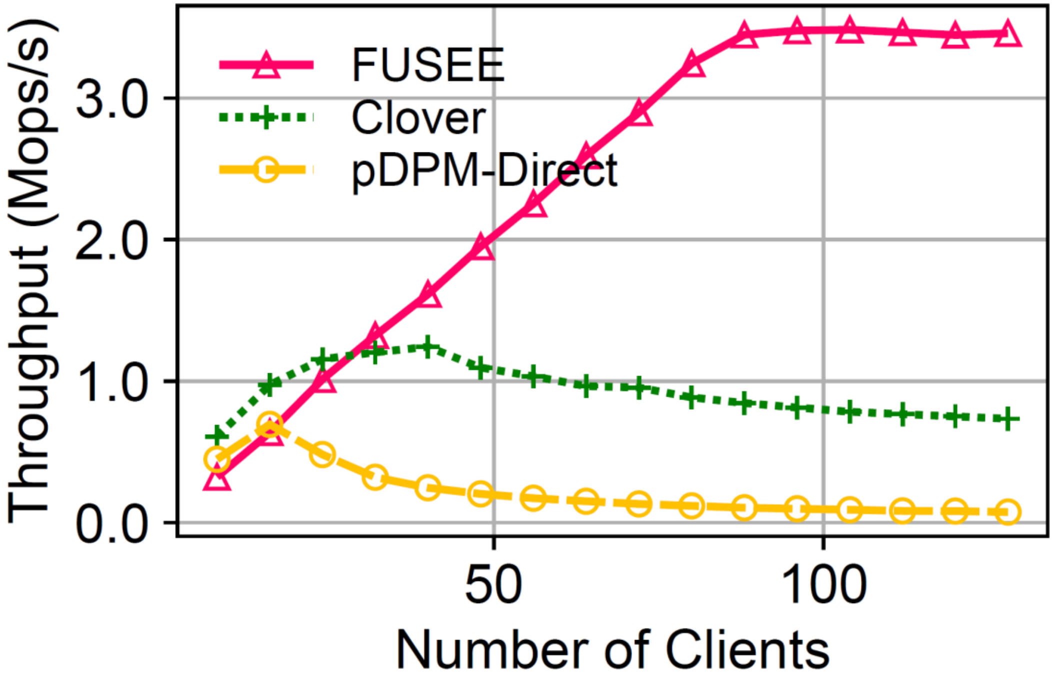
Operations that modifies data always allocate a memory block before

➢ ***Use memory allocation order!***

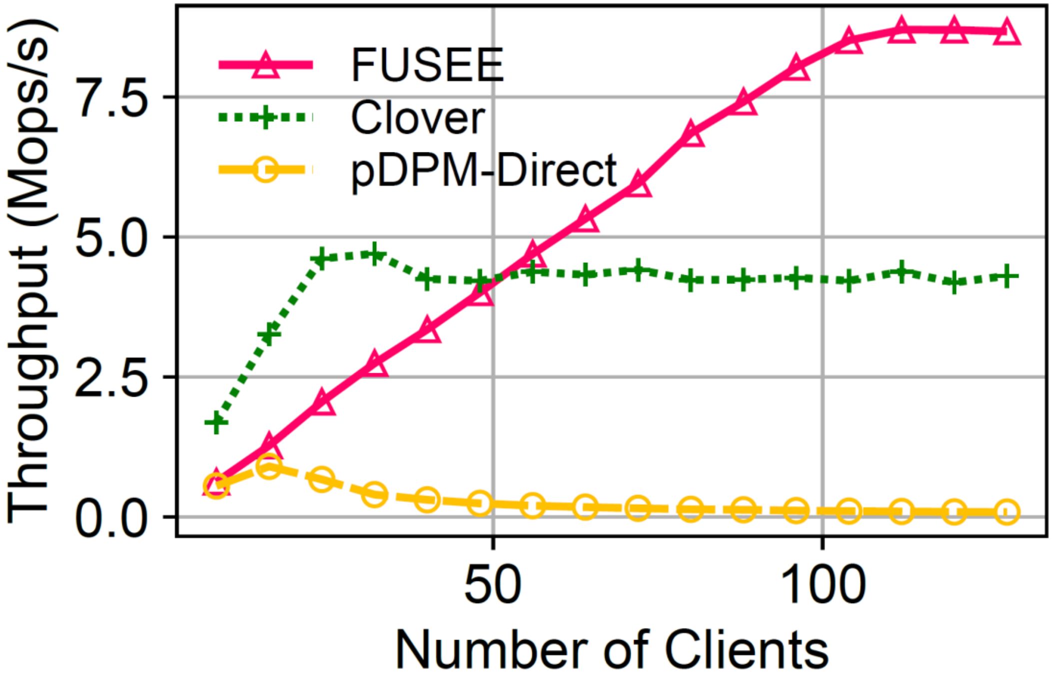


Overall Performance

YCSB-A
(50% UPDATE, 50% SEARCH)



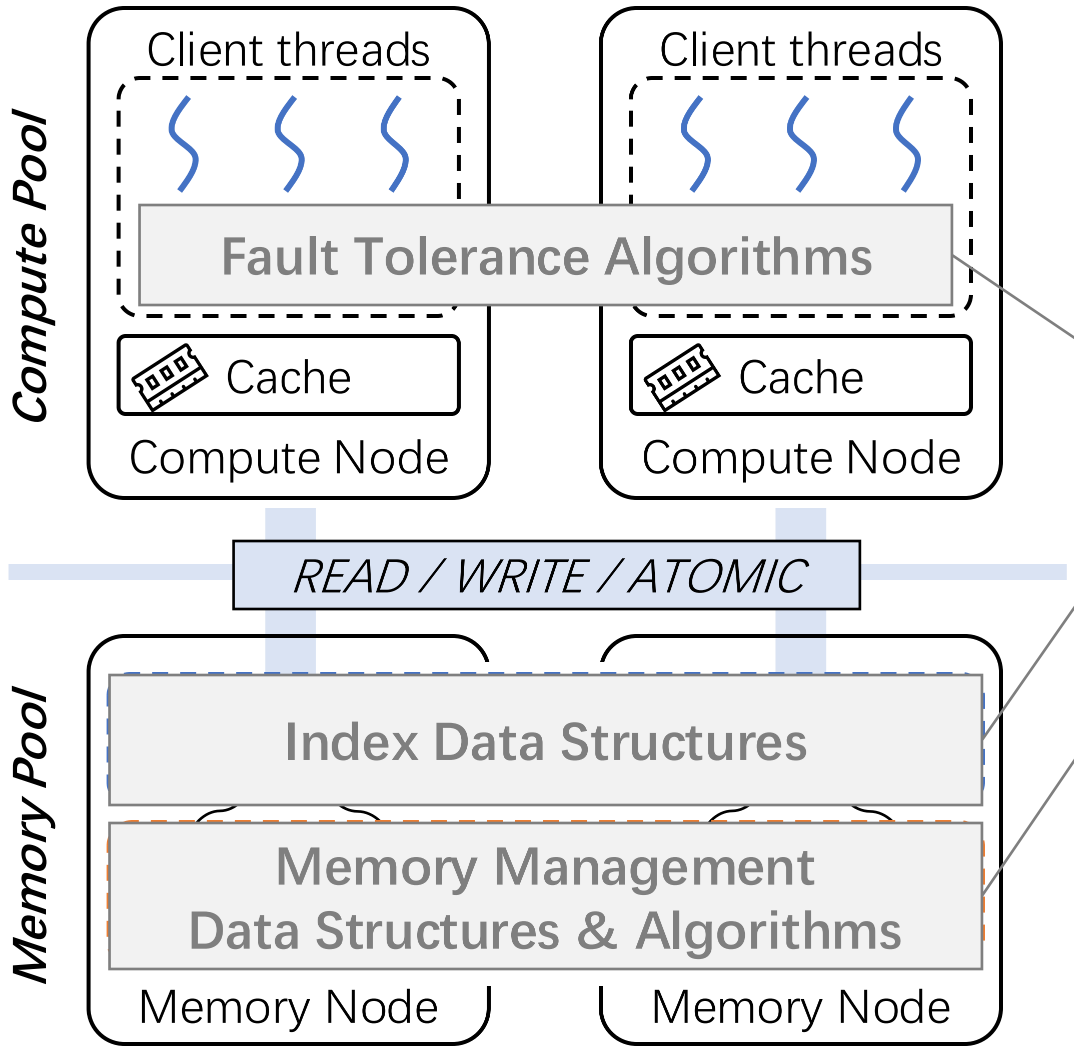
YCSB-C
(100% SEARCH)



Reaches up to 4.9x and 117x higher throughput than Clover[4] and pDPM-Direct[4]

[4] Shin-Yeh Tsai et al. “Disaggregating persistent memory and controlling them remotely…” USENIX ATC 2020.

Thesis Contributions



Analysis

- Guidelines to achieve high performance

Design

- Efficient replication and logging algorithms
- A high-performance range index data structure
- Data structures and algorithms for memory allocation and cache replacement

Deployment

- A production-level disaggregated caching service

Industrial Requirements

Compatibility

- Compatible to existing applications and ecosystems

Reliability

- Handle complex failure situations and reduce failure domain

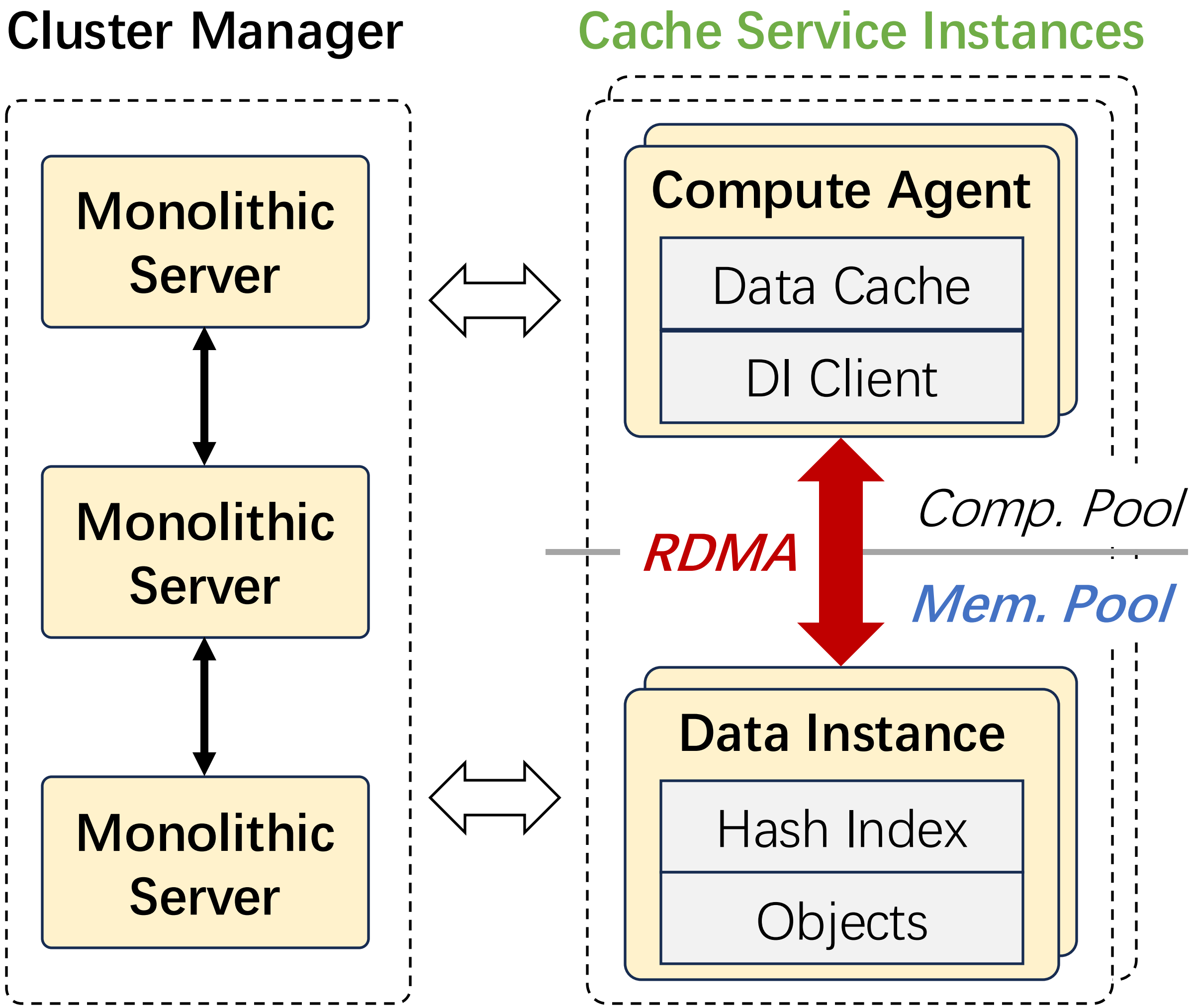
Adaptivity to bursty workloads

- Efficiently scale resources on request bursts

Performance

- Mitigate the performance penalty incurred by memory disaggregation

Disaggregated Memory Caching (DMC)



Compatibility

✓ *Disaggregating a Redis server*

Reliability

✓ *Decoupled replication* to handle compute and memory failures

✓ *On-demand connection management* to reduce failure domain

Adaptivity to bursty workloads

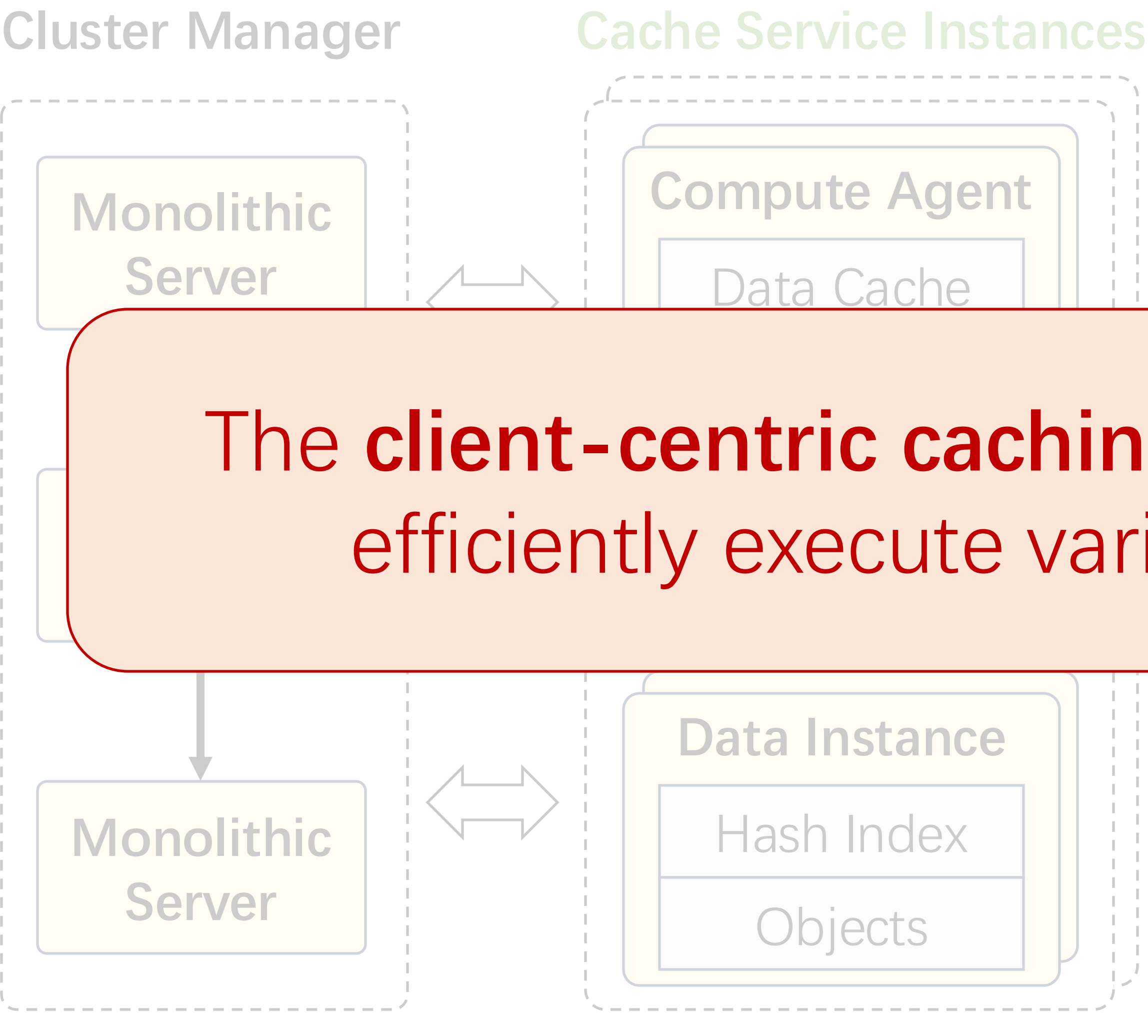
✓ *Logical data sharding* to efficiently scale cache service instances

✓ *Copy-free region migration* scheme to efficiently scale the memory pool

Performance

✓ *Data cache* to reduce data access latency

Disaggregated Memory Caching (DMC)



Compatibility

✓ *Disaggregating a Redis server*

Reliability

✓ *Decoupled replication* to handle compute and

cache

✓ *Copy-Free region migration* scheme to efficiently scale the memory pool

Performance

✓ *data cache* to reduce data access latency

The **client-centric caching framework** is adopted to efficiently execute various caching algorithms

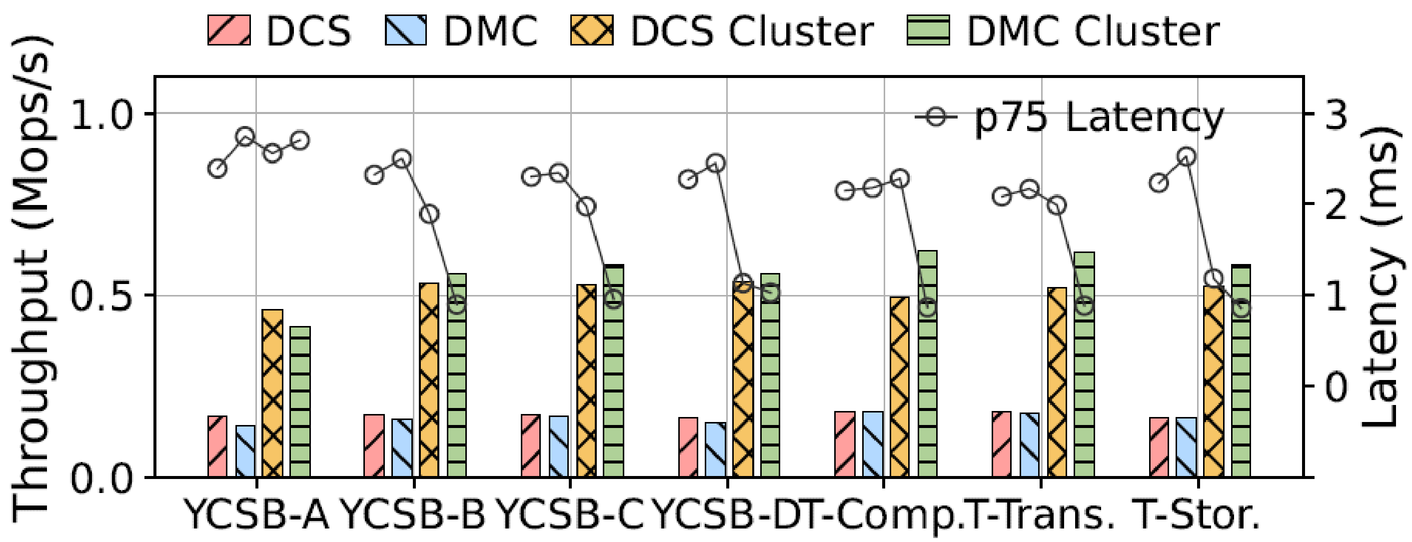
Evaluation Results

Memory utilization

- ➢Achieve 2.6 times higher memory utilization
- ➢Reduce 45% over-provisioned memory thanks to on-demand memory allocation

Performance

- ➢Less than 15% penalty
- ➢Up to 25% improvement



Lessons Learned

Memory disaggregation is necessary

- Key to memory utilization is on-demand allocation
- Can we directly achieve on-demand allocation in a monolithic cluster
- No, the poor elasticity will become a bottleneck

The overhead of memory disaggregation is affordable

- The throughput can still satisfy SLA

System design is critical to fully exploit DM

- Decouple failures, achieve shared everything in the software layer

Summary

Analysis

Guidelines to achieve high performance

Design

Efficient replication and logging algorithms

A high-performance range index data structure

Data structures and algorithms for memory allocation and cache replacement

Deployment

A production-level disaggregated caching service

Achieving practical resource disaggregation by designing efficient data structures and algorithms **in a bottom-up manner**

Optimize for I/O

- Balance I/O sizes and numbers

Optimize for concurrency

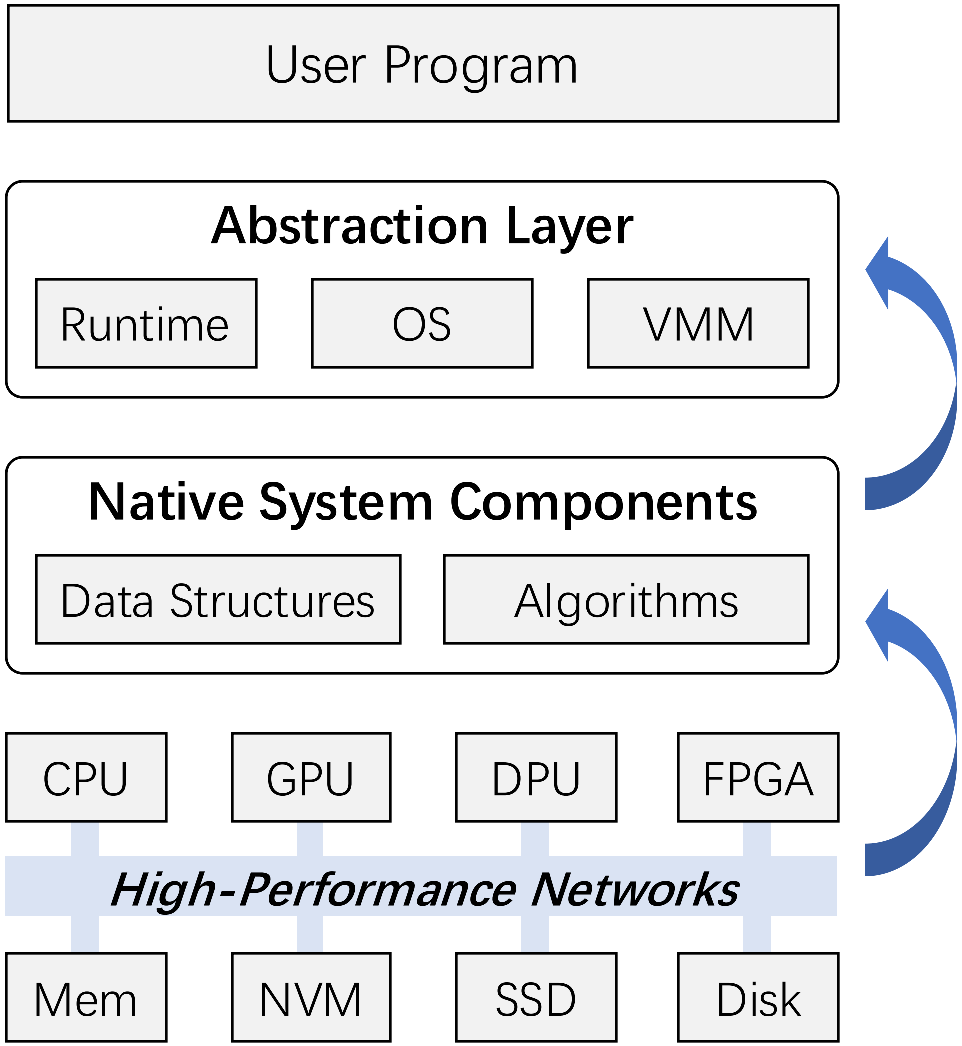
- Avoiding & mitigating concurrency conflicts

Optimize for asymmetry

- Remove memory-side CPUs from critical paths

So, what’s next?

Achieve ***high-performance*** resource disaggregation for ***arbitrary program***



Compatibility

Extend from ***individual components*** to a compatible ***runtime system***

- **Cooperation** of data structures and algorithms on heterogeneous devices

Performance

Extend from ***memory disaggregation*** to more ***general disaggregated hardware***

- Identify different **I/O**, **concurrency**, and **asymmetry** characters

Conclusion

- Resource disaggregation is a promising next-generation data center architecture
- Disaggregated programs suffer from poor performance
  - Requirements to achieve high performance: I/O, concurrency, asymmetry
  - Efficient data structures and algorithms for disaggregated memory
  - Industrial practice and lessons learned
- Resource disaggregation is far from being well addressed
  - Extend to more diverse hardware and various software

Publication List

1. **J. Shen**, P. Zuo, X. Luo, Y. Su, J. Gu, H. Feng, Y. Zhou, and M. R. Lyu. Ditto: An Elastic and Adaptive Memory-Disaggregated Caching System. *SOSP 2023.*
2. **J. Shen**, P. Zuo, X. Luo, T. Yang, Y. Su, Y. Zhou, and M. R. Lyu. FUSEE: A Fully Memory-Disaggregated Key-Value Store. *FAST 2023*.
3. X. Luo, P. Zuo, **J. Shen**, J. Gu, X. Wang, M. R. Lyu, and Y. Zhou. SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory. *OSDI 2023*.
4. **J. Shen**, T. Yang, Y. Su, Y. Zhou, and M. R. Lyu. Defuse: A Dependency-Guided Function Scheduler to Mitigate Cold Starts on FaaS Platforms. *ICDCS 2021*.
5. **J. Shen**, P. Zuo, B. Che, Z. Chen, X. Zhang, S. Goren, Z. Xiang, P. Chen, Y. Miao, J. Feng, and M. R. Lyu. Productionizing a Memory-Disaggregated Caching System. Submitted to NSDI 2025.
6. T. Yang, **J. Shen**, Y. Su, X. Ren, Y. Yang, and M. R. Lyu. Characterizing and Mitigating Anti-patterns of Alerts in Industrial Cloud Systems. *DSN 2022*.
7. T. Yang, **J. Shen**, Y. Su, X. Ling, Y. Yang, and M. R. Lyu. AID: Efficient Prediction of Aggregated Intensity of Dependency in Large-scale Cloud Systems. *ASE 2021*.
8. T. Yang, B. Li, **J. Shen**, Y. Su, Y. Yang, and M. R. Lyu. Managing Service Dependency for Cloud Reliability: The Industrial Practice. *ISSRE 2022 Workshops*.
9. S. Jiang, **J. Shen**, S. Wu, Y. Cai, Y. Yu, and Y. Zhou. Towards Usable Neural Comment Generation via Code-Comment Linkage Interpretation: Method and Empirical Study. *TSE 2023*.
10. T. Yang, C. Lee, **J. Shen**, Y. Su, Y. Yang, and M. R. Lyu. An Adaptive Resilience Testing Framework for Microservice Systems. In *arXiv preprint*, 2022.

Thank you

Q&A