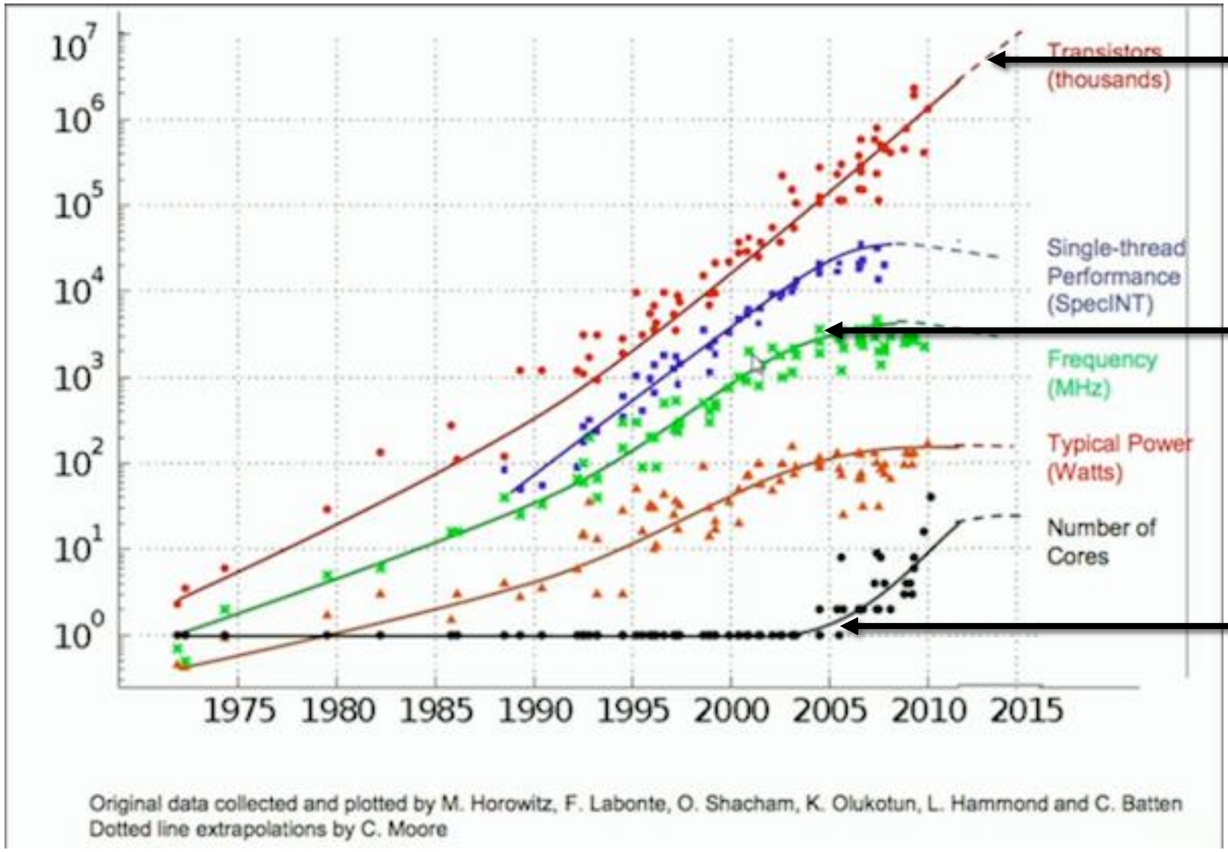
Multicore Scalability through Asynchronous Work

AJIT MATHEW

MASTER'S THESIS DEFENSE

Why multicore scalability is important?



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Moore's Law

Clock Speed Saturates

Rise of Multi-core

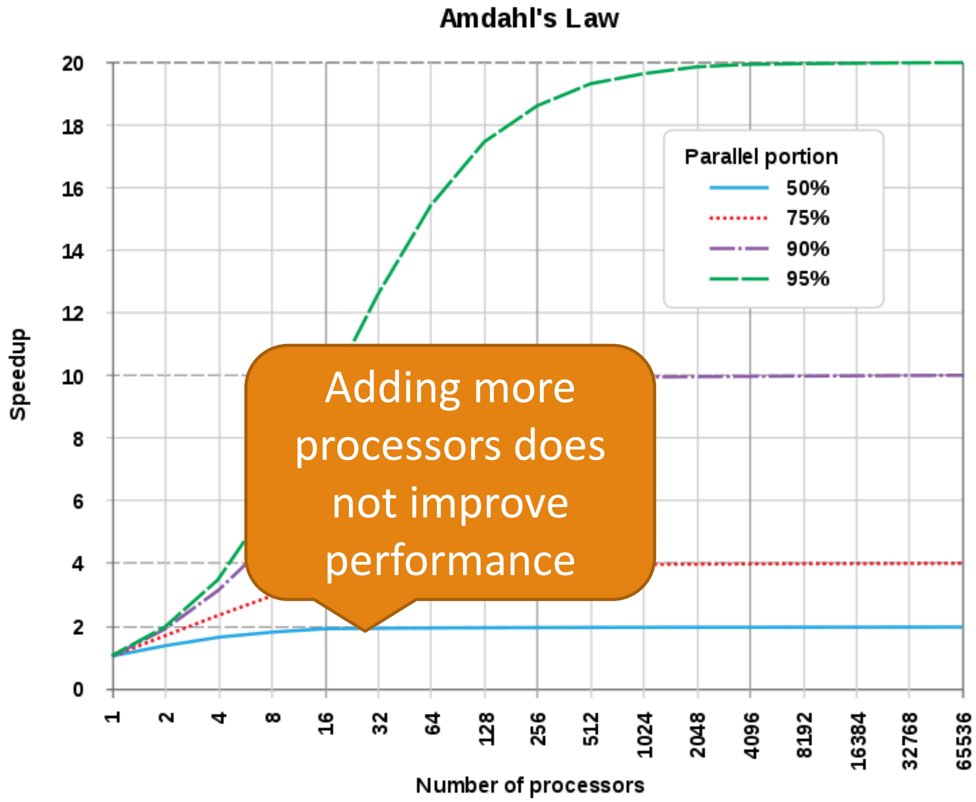
Free Lunch is Over

- We entered Multi-Core Era in around 2005
- To improve software performance, program needs to be parallelized
- In other words:

Parallelize or Perish

Performance in Multi-Core Era

- Amdhal's Law:
  - Performance in multi-cores is limited by the portion of program that cannot be parallelized (called *sequential section* or *critical section*)



Adding more processors does not improve performance

Key Idea

Problem:
Adding more processor does not improve performance

Solution:
Use extra processor to do work to reduce sequential section.

Asynchronous Work for scalability

- Idea: Perform tasks which are necessary for consistency in critical section and perform non-necessary task outside the critical section using background threads.
- Example:
  - free() is an expensive operation
  - Freeing memory in critical section is unnecessary
  - Use a garbage collector to free memory

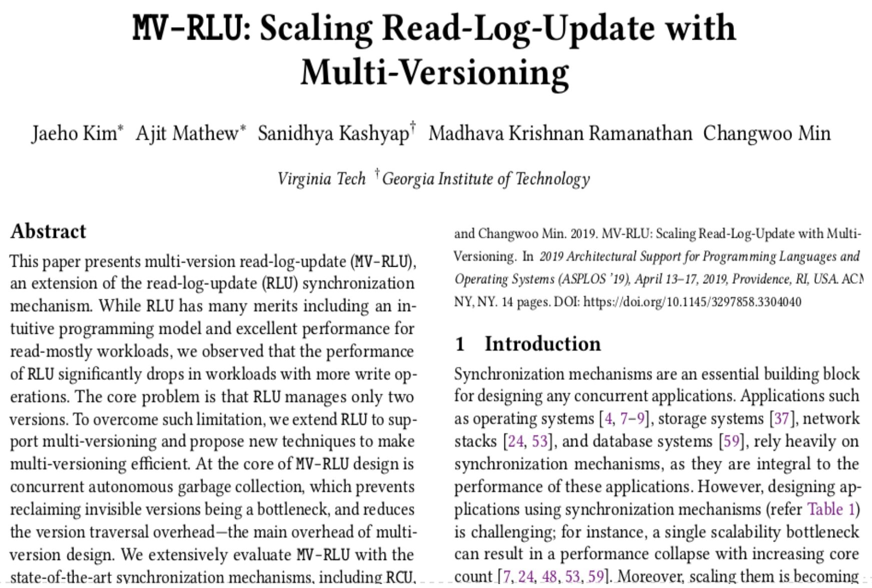
Contribution

- My thesis explores how asynchronous work can be used to improve scalability in multicores
- The thesis presents two systems built on the idea of asynchronous work

MV-RLU & Hydralist

ASPLOS, 2019

Under Submission



MV-RLU: Scaling Read-Log-Update with Multi-Versioning

Jaeho Kim* Ajit Mathew* Sanidhya Kashyap† Madhava Krishnan Ramanathan Changwoo Min

Virginia Tech †*Georgia Institute of Technology*

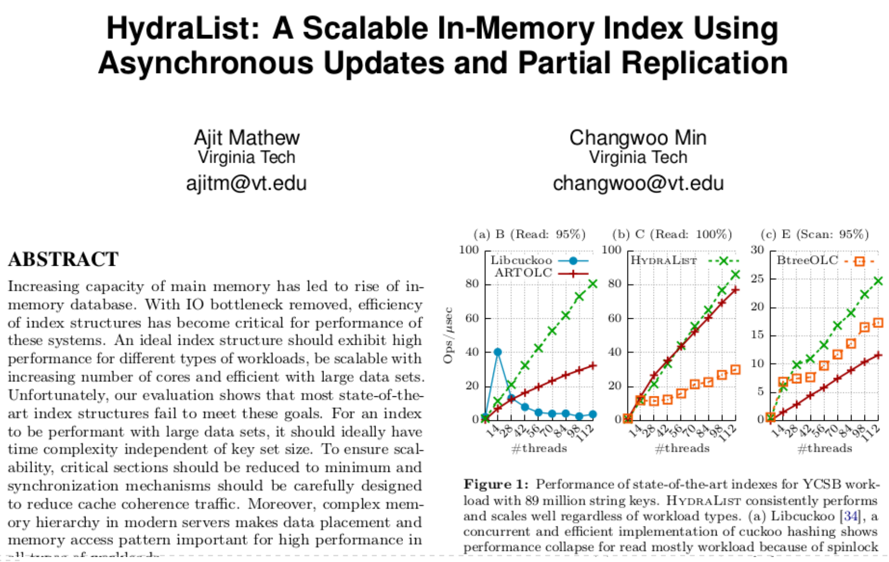
Abstract

This paper presents multi-version read-log-update (MV-RLU), an extension of the read-log-update (RLU) synchronization mechanism. While RLU has many merits including an intuitive programming model and excellent performance for read-mostly workloads, we observed that the performance of RLU significantly drops in workloads with more write operations. The core problem is that RLU manages only two versions. To overcome such limitation, we extend RLU to support multi-versioning and propose new techniques to make multi-versioning efficient. At the core of MV-RLU design is concurrent autonomous garbage collection, which prevents reclaiming invisible versions being a bottleneck, and reduces the version traversal overhead—the main overhead of multi-version design. We extensively evaluate MV-RLU with the state-of-the-art synchronization mechanisms, including RCU,

and Changwoo Min. 2019. MV-RLU: Scaling Read-Log-Update with Multi-Versioning. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM NY, NY. 14 pages. DOI: https://doi.org/10.1145/3297858.3304040

1 Introduction

Synchronization mechanisms are an essential building block for designing any concurrent applications. Applications such as operating systems [4, 7–9], storage systems [37], network stacks [24, 53], and database systems [59], rely heavily on synchronization mechanisms, as they are integral to the performance of these applications. However, designing applications using synchronization mechanisms (refer Table 1) is challenging; for instance, a single scalability bottleneck can result in a performance collapse with increasing core count [7, 24, 48, 53, 59]. Moreover, scaling them is becoming



HydraList: A Scalable In-Memory Index Using Asynchronous Updates and Partial Replication

Ajit Mathew
Virginia Tech
ajitm@vt.edu

Changwoo Min
Virginia Tech
changwoo@vt.edu

ABSTRACT

Increasing capacity of main memory has led to rise of in-memory database. With IO bottleneck removed, efficiency of index structures has become critical for performance of these systems. An ideal index structure should exhibit high performance for different types of workloads, be scalable with increasing number of cores and efficient with large data sets. Unfortunately, our evaluation shows that most state-of-the-art index structures fail to meet these goals. For an index to be performant with large data sets, it should ideally have time complexity independent of key set size. To ensure scalability, critical sections should be reduced to minimum and synchronization mechanisms should be carefully designed to reduce cache coherence traffic. Moreover, complex memory hierarchy in modern servers makes data placement and memory access pattern important for high performance in

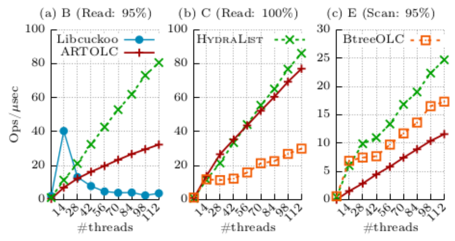


Figure 1: Performance of state-of-the-art indexes for YCSB workload with 89 million string keys. HydraList consistently performs and scales well regardless of workload types. (a) Libcuckoo [34], a concurrent and efficient implementation of cuckoo hashing shows performance collapse for read mostly workload because of spinlock

Torgersen Research Excellence Award 2019

MV-RLU: Scaling Read-Log-Update with Multi-Versioning

Synchronization mechanisms are essential

- A single scalability bottleneck in synchronization mechanism can result in a performance collapse with increasing core count [Boyd-Wickizer,2010]

Scalability of program depends on scalability of underlying synchronization mechanism

Core count continues to rise….

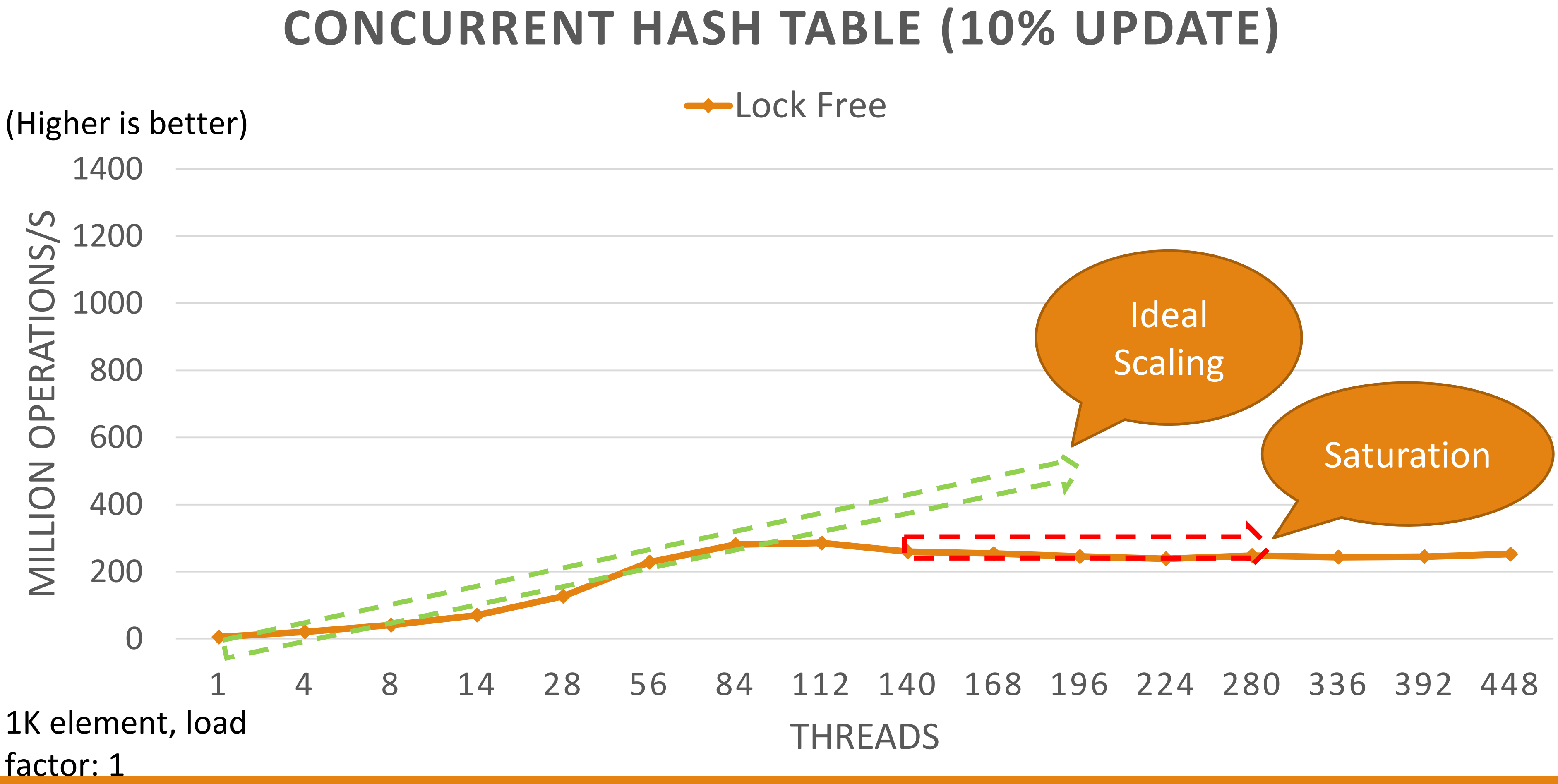
The Intel Second Generation Xeon Scalable: Cascade Lake, Now with Up To 56-Cores and Optane!

by Ian Cutress on April 2, 2019 1:02 PM EST

Posted in CPUs Intel Xeon Enterprise CPUs Xeon Scalable Cascade Lake Cascade-AP

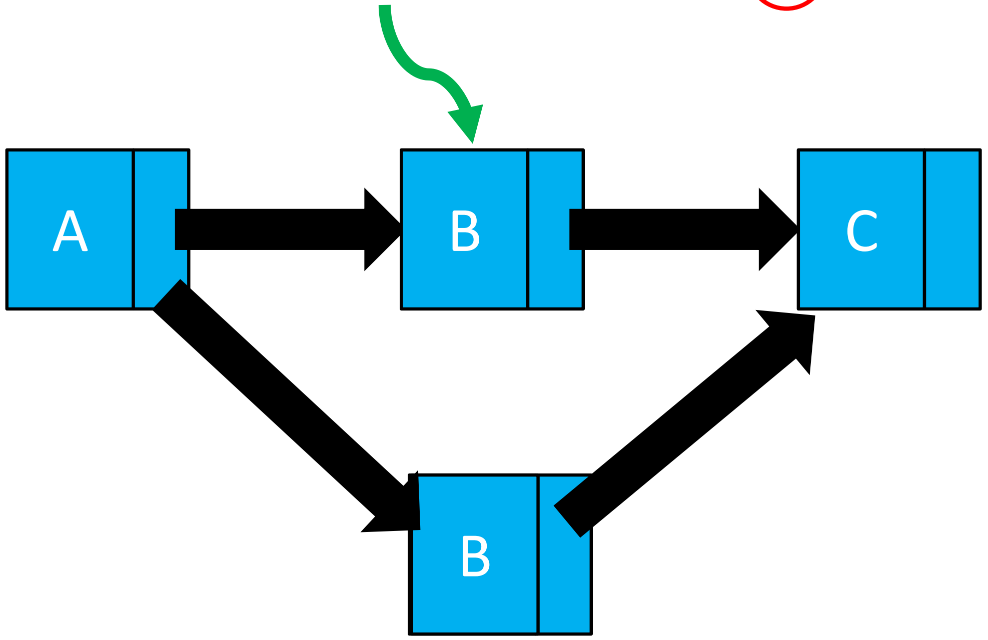
AMD Second Gen EPYC Beastly Server CPUs Could Rock 64 Cores, 128 Threads And 256MB Cache

Can synchronization mechanisms scale at high core count?



Read Copy Update (RCU)

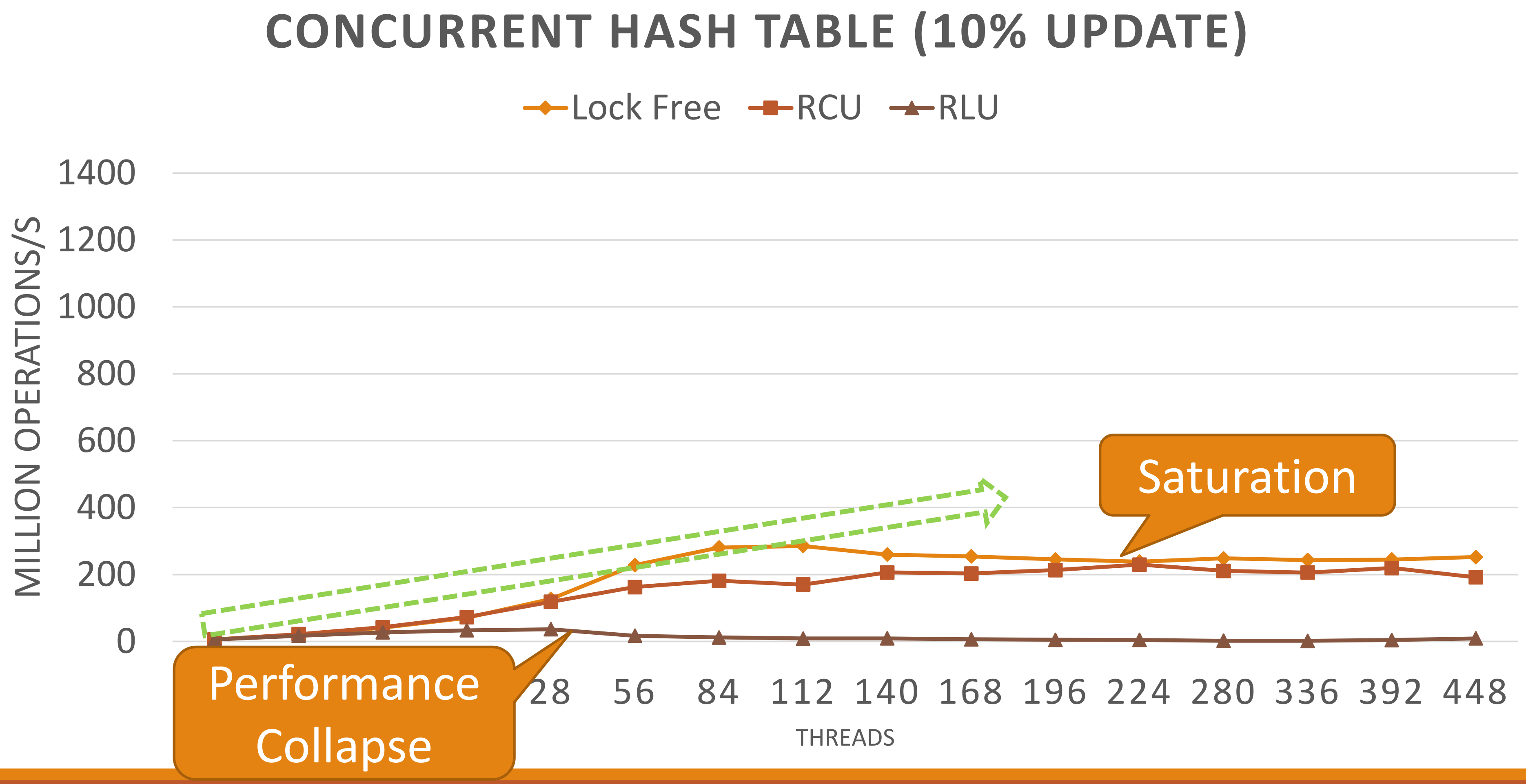
- Widely used in Linux kernel
- Readers never block ☺
- Multi-pointer update is difficult ☹



Read-Log-Update (RLU) [Matveev'15]

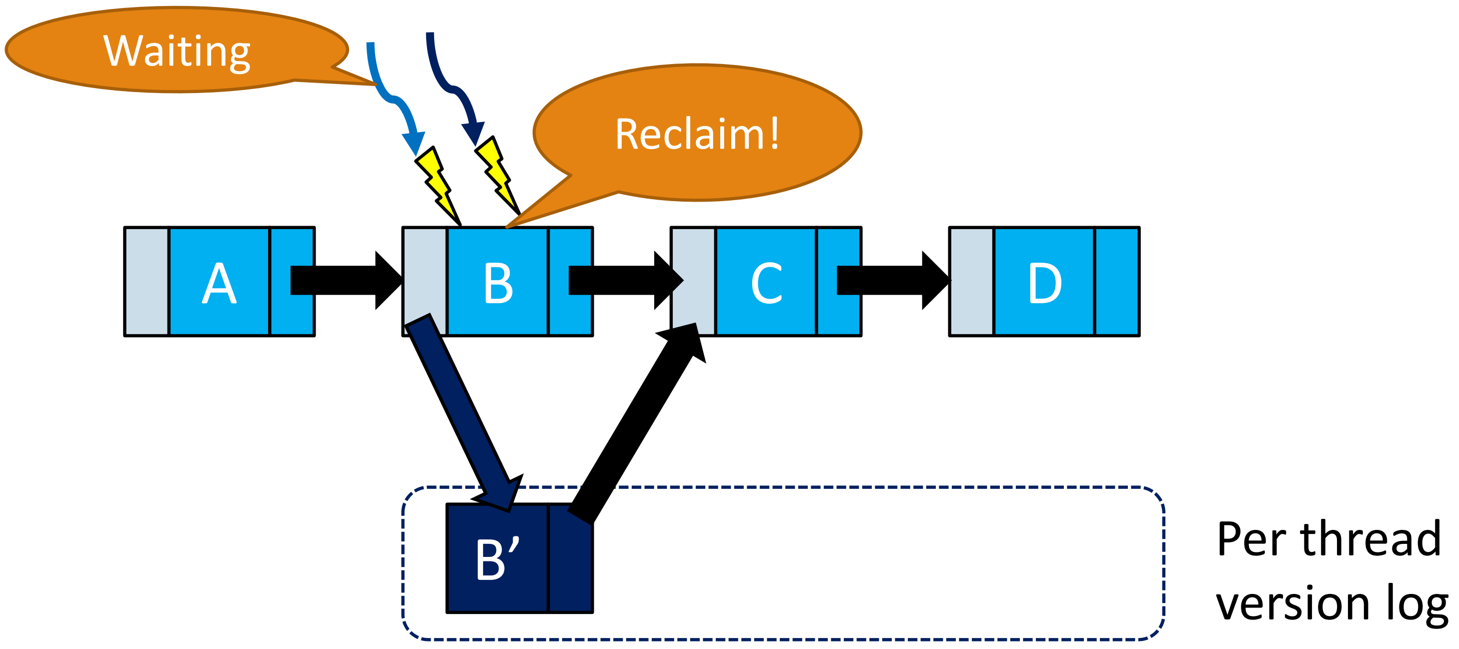
- ❑ Readers do not block
- ❑ Allow multi-pointer update
- ❑ Key idea: Use global clock and per thread log to make updates atomically visible

Even RCU and RLU does not scale



Why does not RLU scale?

Legend
Object
Version of Object
Header



Synchronous waiting due to restriction on number of versions per object is bottleneck in RLU design

How to scale RLU?

Problem:
Restriction in number of versions causes synchronous waiting

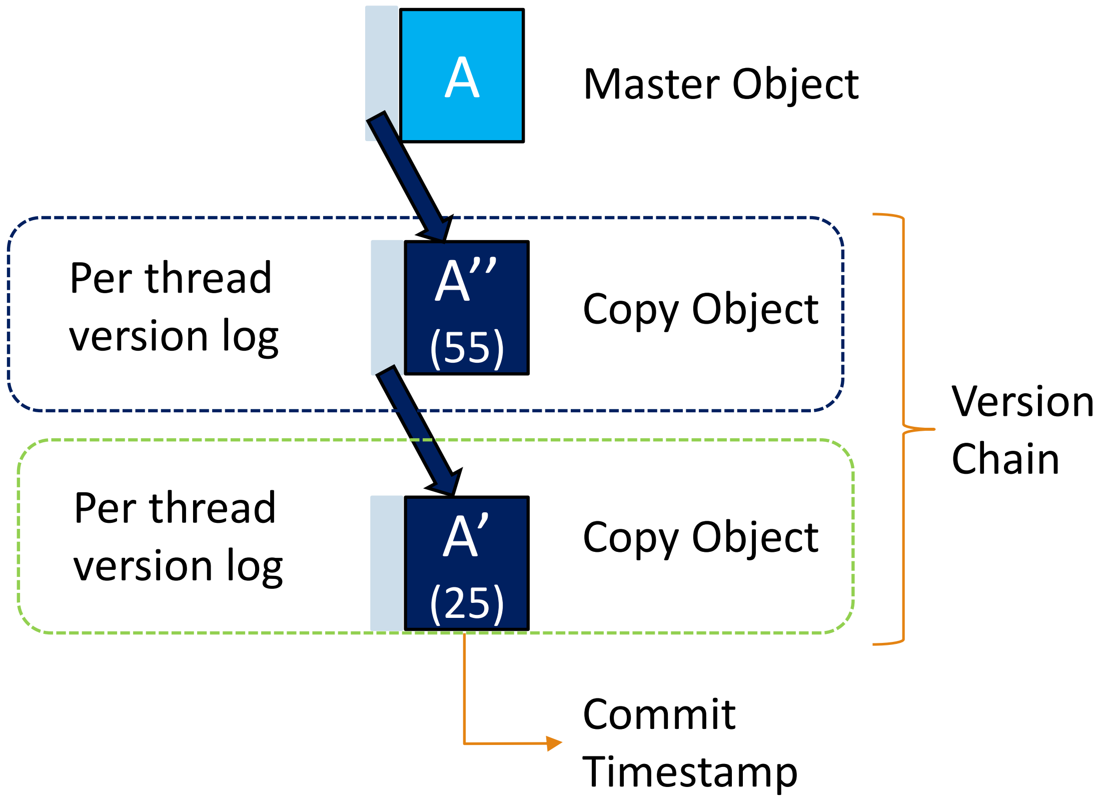
Solution:
Remove restriction on number of version == Multi-Versioning

Scaling RLU through Multi-Versioning

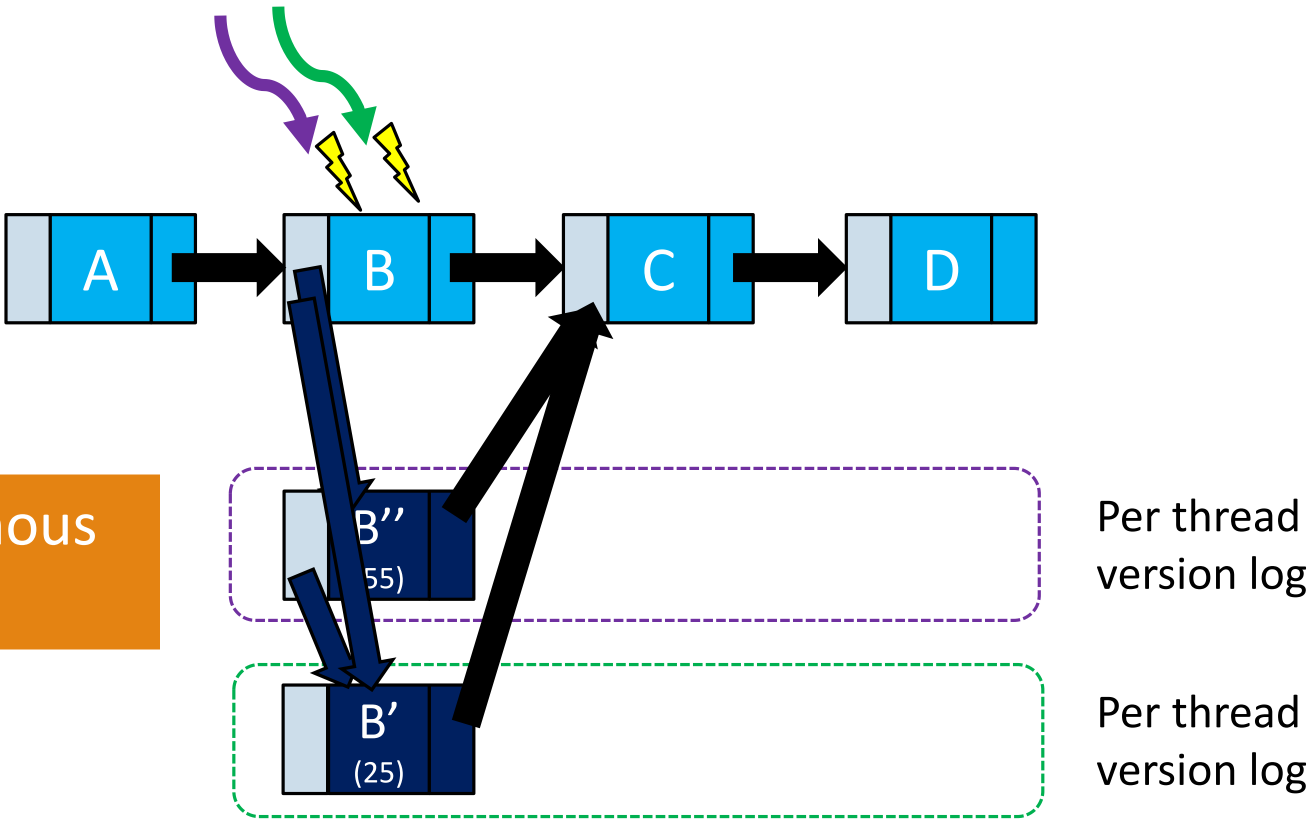
- Multi-Version Read-Log-Update (MV-RLU)
  - Allows multiple version to exists at same time
  - Removes synchronous waiting from critical path
- Scaling Multi-versioning
  - Scalable Timestamp allocation
  - Concurrent and autonomous garbage collection

MV-RLU shows 4-8x performance improvement with KyotoCabinet

Design: Overview



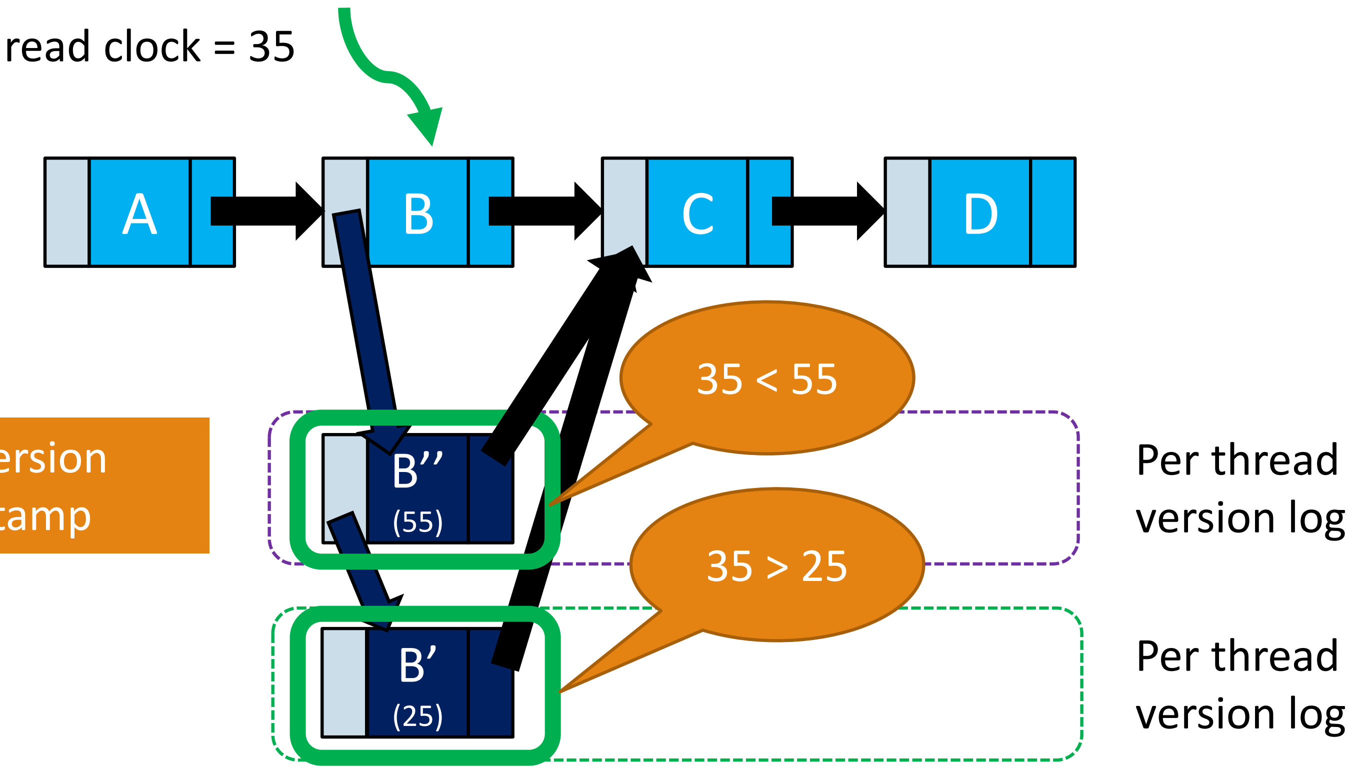
Updates in MV-RLU



No synchronous waiting

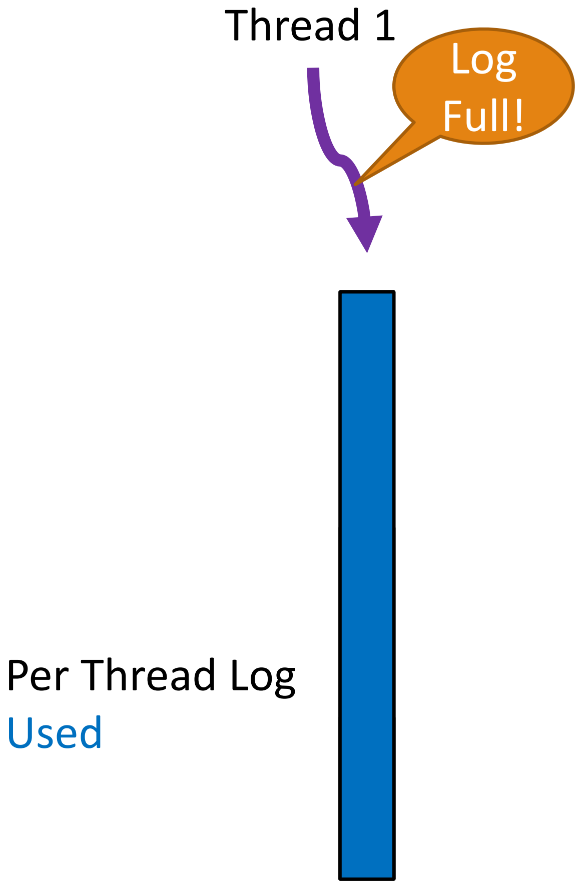
Reads in MV-RLU

Reader note the global clock at start of critical section



read clock > version commit timestamp

Memory is limited!



- ❑ Garbage Collection is needed
  - ❑ Reclaim obsolete version
  - ❑ Should be scalable

Detecting obsolete version

- Quiescent State based Reclamation (QSBR)
  - Grace Period: Time interval between which each thread has been outside critical section at least once
- Grace Period detection is delegated to a special thread

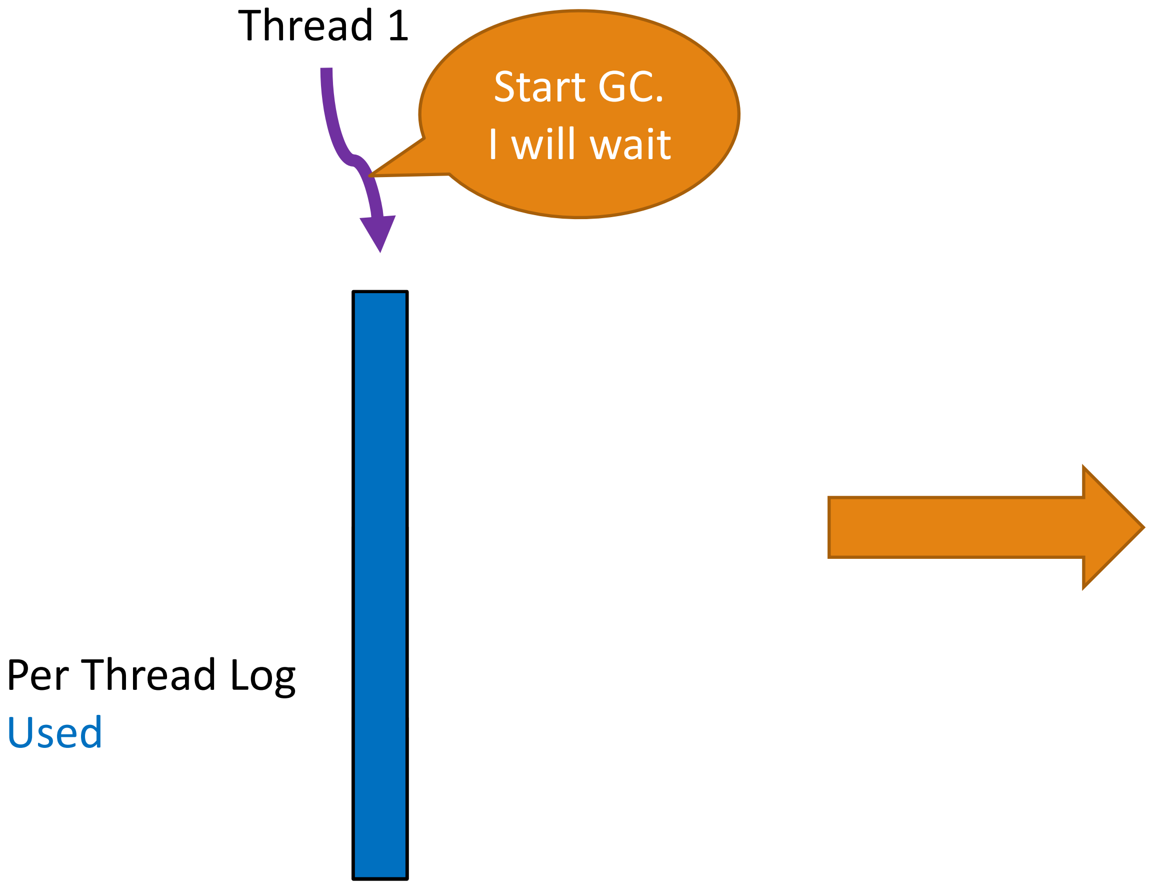
Reclaiming Obsolete version

- Concurrent Reclamation
  - Every thread reclaims it’s own log
  - Cache friendly
  - Scales with increasing number of threads

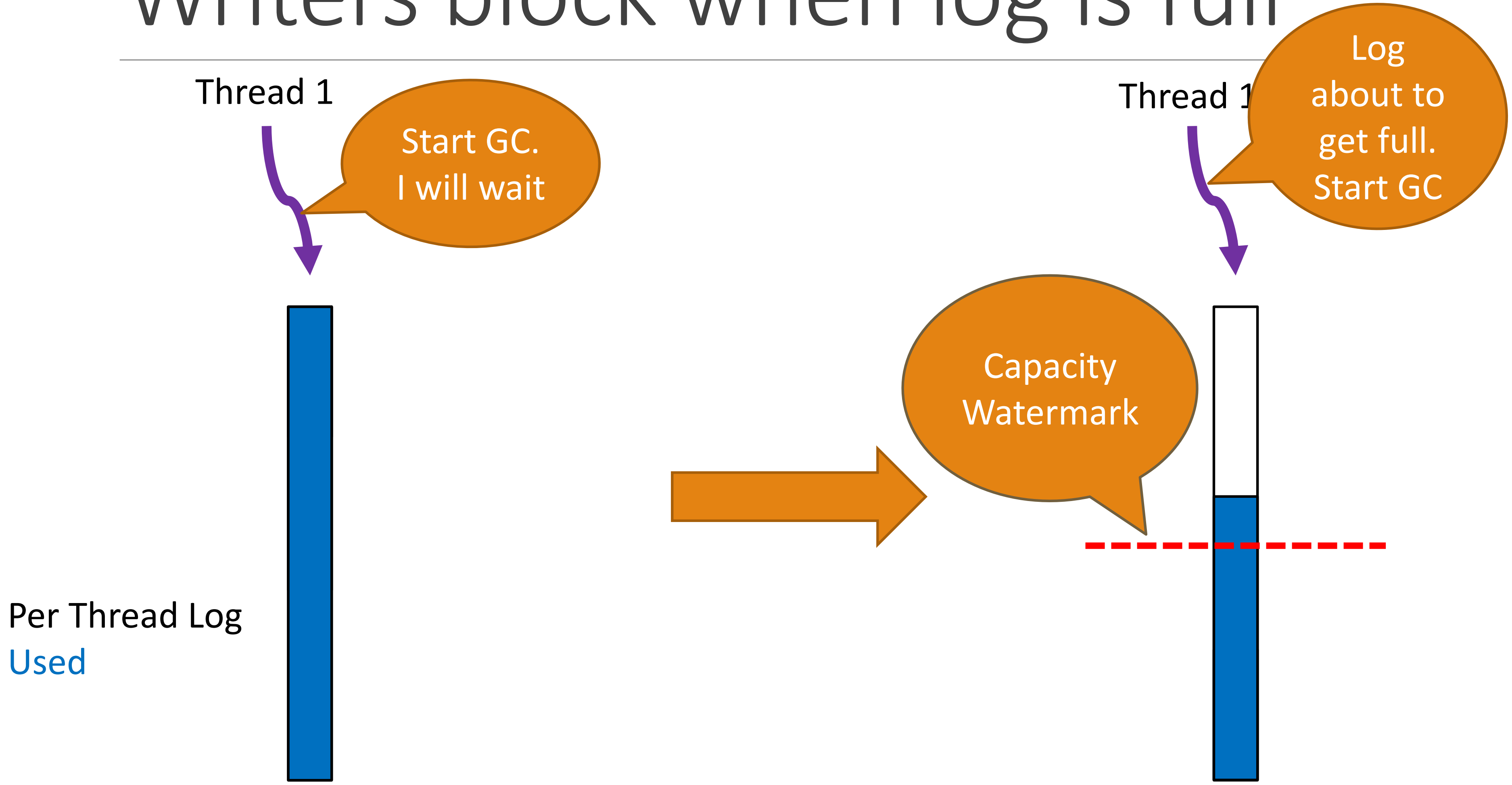
Triggering Garbage Collection

- Ideally, triggers should be workload agnostic
- Two conditional Trigger or Watermark
  - Capacity Watermark
  - Dereference Watermark

Writers block when log is full



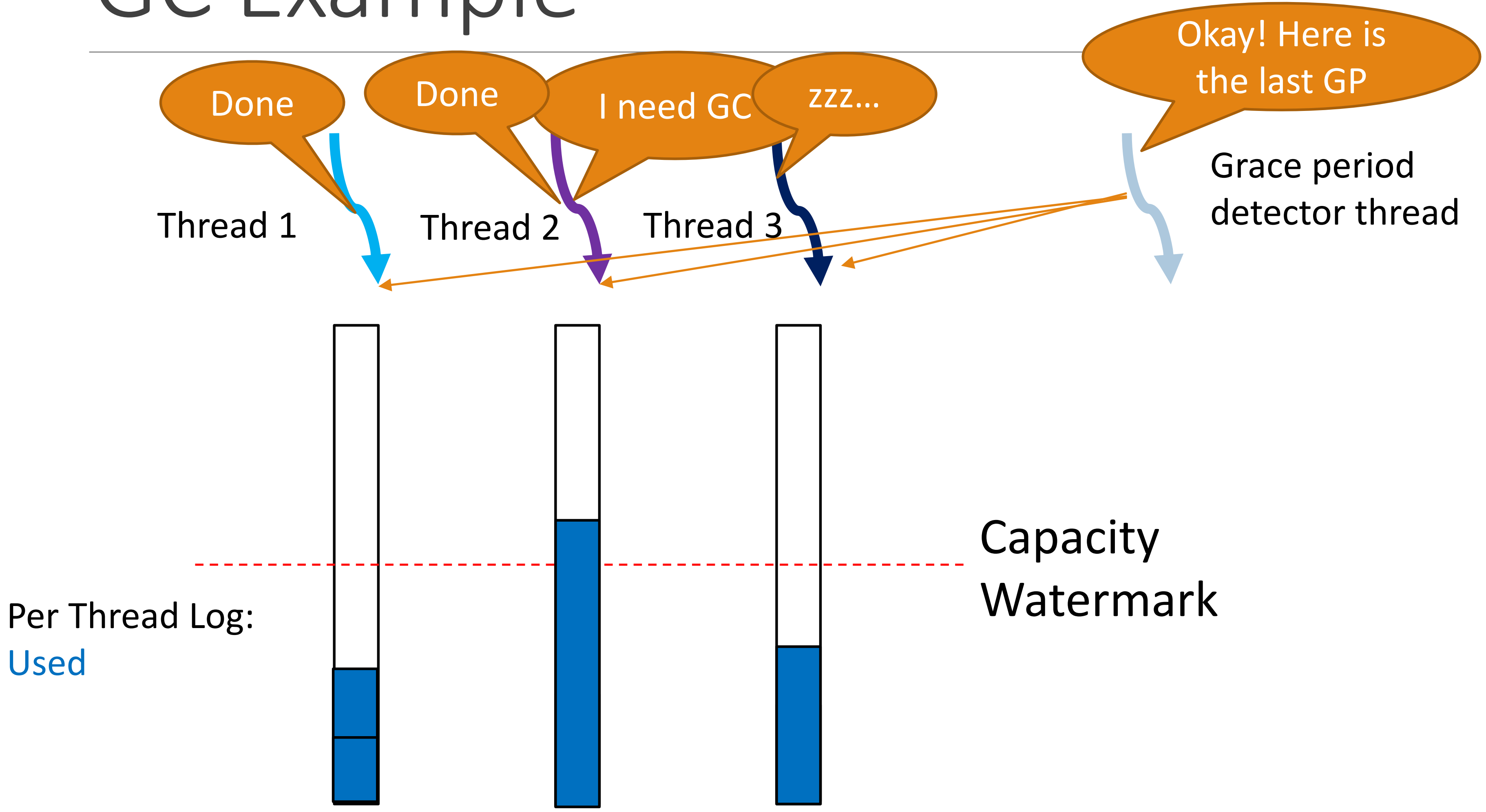
Writers block when log is full



Prevent log from filling up using capacity watermark

- Writers block when log is full
- To prevent blocking, start GC when log is almost full
- Capacity Watermark
  - Trigger GC when a thread’s log is about to get full
  - Works well in write heavy workload

GC Example


Done
Done
I need GC
zzz…
Okay! Here is the last GP
Thread 1
Thread 2
Thread 3
Grace period detector thread
Capacity Watermark
Per Thread Log:
Used

More detail

- ❑ Scalable timestamp allocation
- ❑ Version Management
- ❑ Proof of correctness
- ❑ Implementation details

Outline

- ❑ Background
- ❑ Design
- ❑ **Evaluation**
- ❑ Conclusion

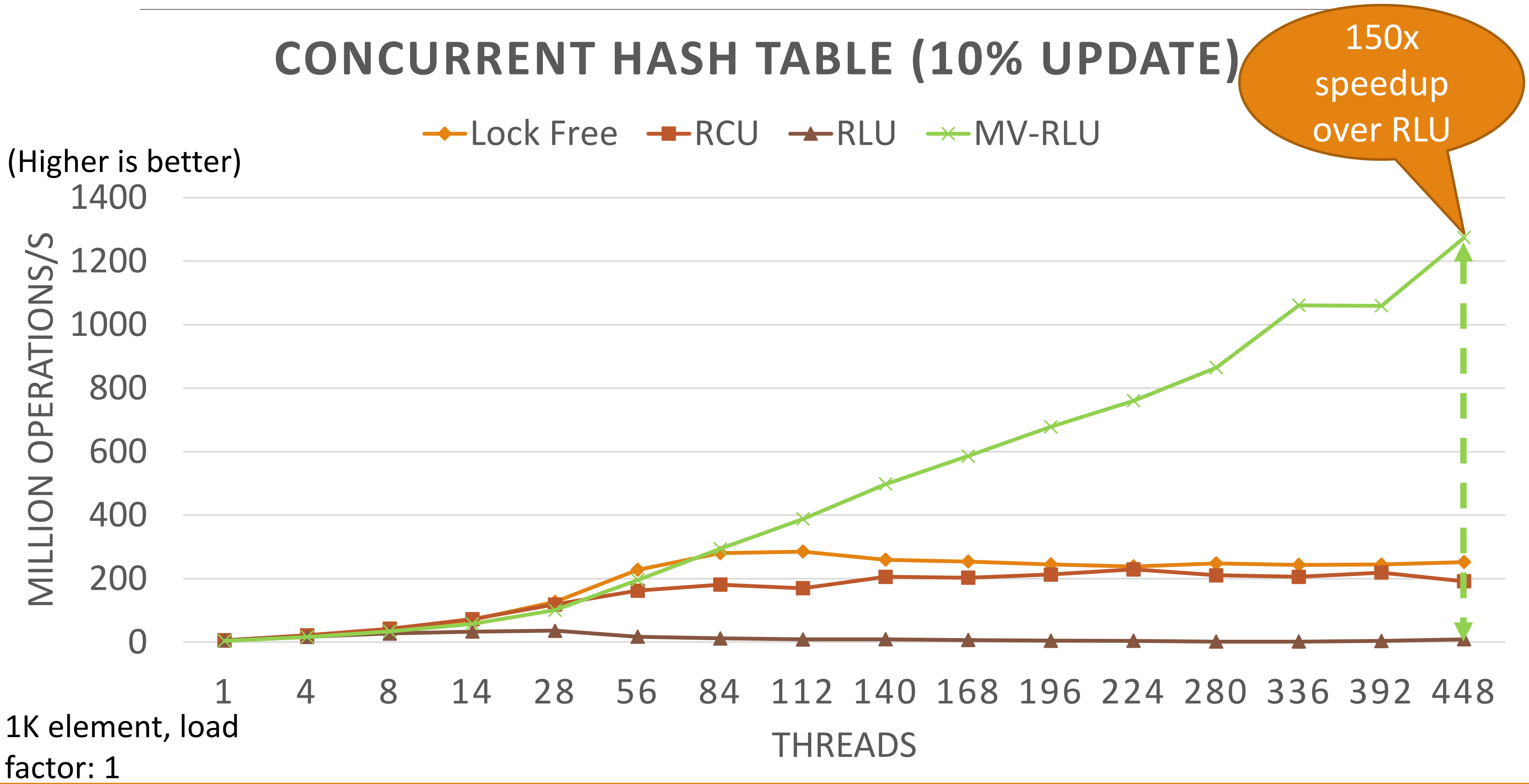
Evaluation Question

- ❑ Does MV-RLU scale?
- ❑ What is the impact of our proposed approaches?
- ❑ What is its impact on real-world workloads?

Evaluation Platform

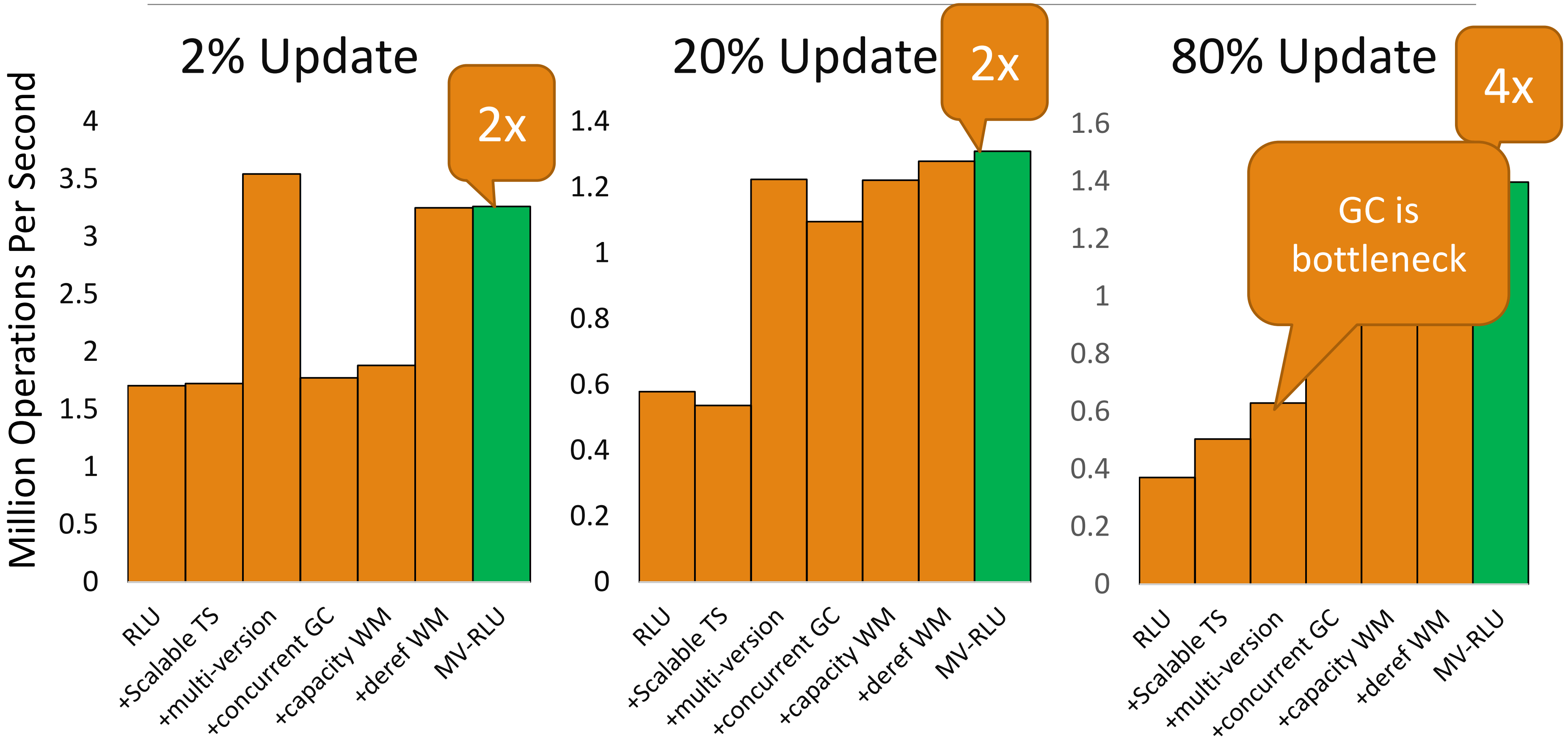
- 8 socket, 448 core server
- Intel Xeon Platinum 8180
- 337 GB Memory
- Linux 4.17.3

Microbenchmark

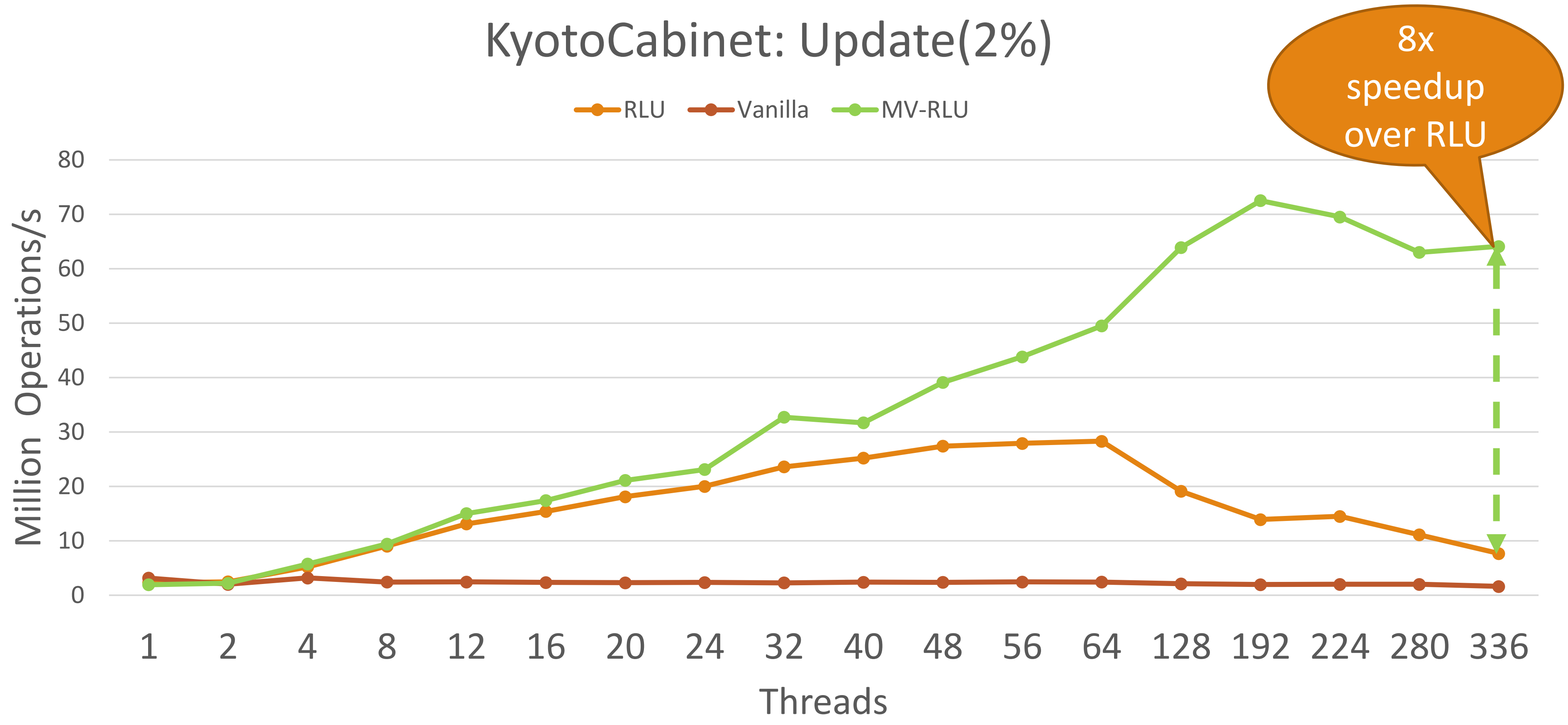


1K element, load factor: 1

Factor Analysis



Key Value Benchmark



Conclusion

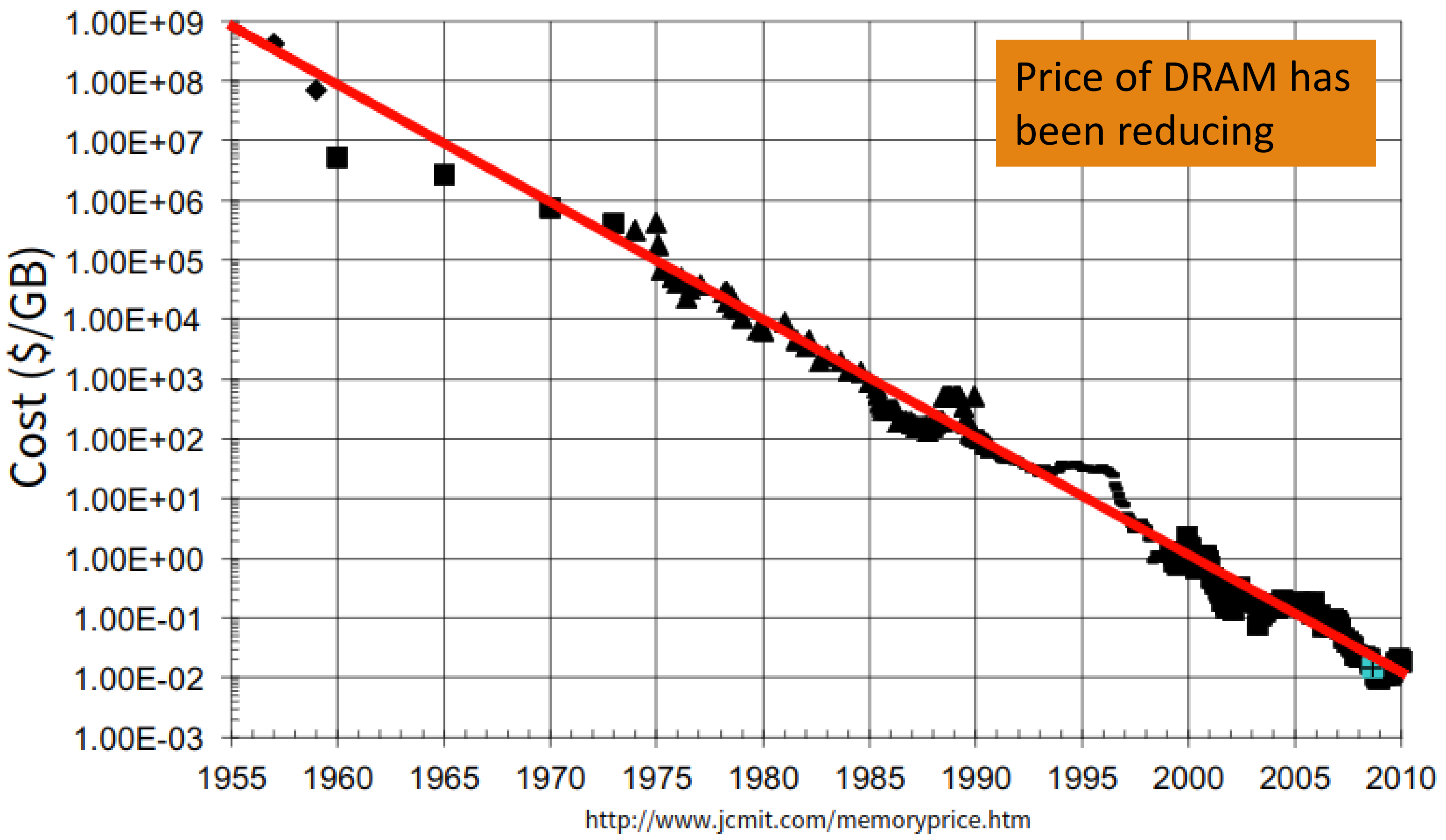
- MV-RLU: Scaling RLU through Multi-Versioning
- Multi-Versioning removes synchronous waiting
- Concurrent and autonomous garbage collection
- MV-RLU show unparallel performance for a variety of benchmark.

https://github.com/cosmoss-vt/mv-rlu

Hydralist

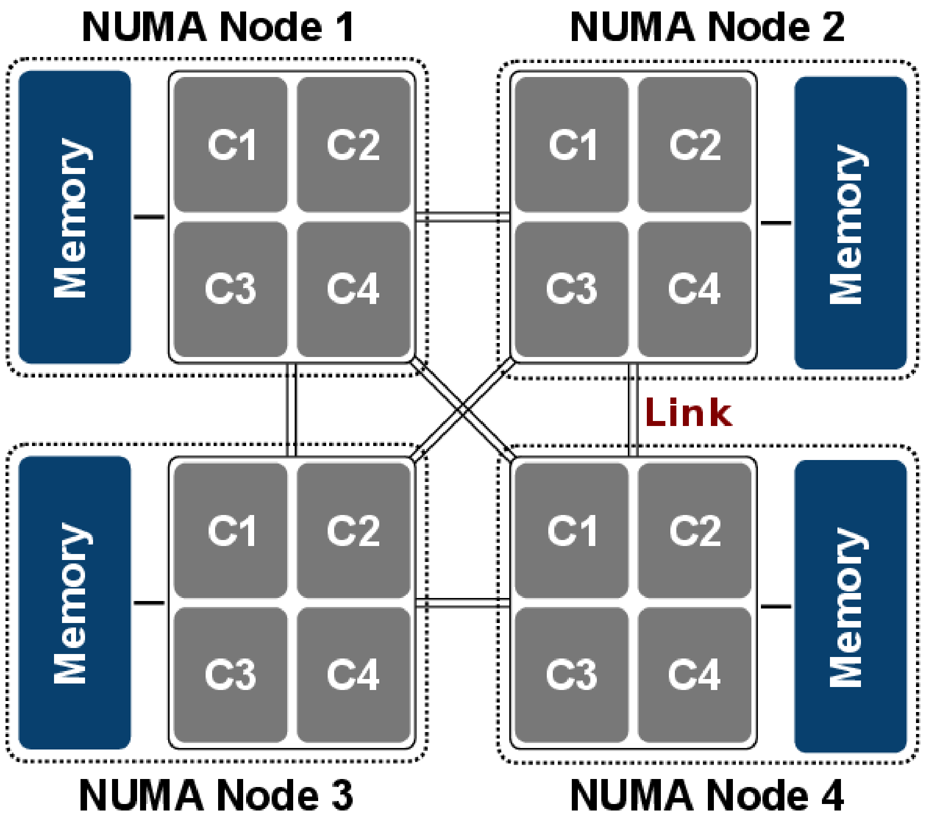
A SCALABLE IN-MEMORY INDEX USING ASYNCHRONOUS UPDATE AND PARTIAL REPLICATION

Trends in Memory Capacity



Price of DRAM has been reducing

Non-Uniform Memory Architecture



http://www.iue.tuwien.ac.at/phd/weinbub/dissertationsu16.html

Rise of In-Memory Storage

- Larger and cheaper memory has allowed databases to store their data and meta-data in memory
- I/O bottlenecks caused due to slow disk reads

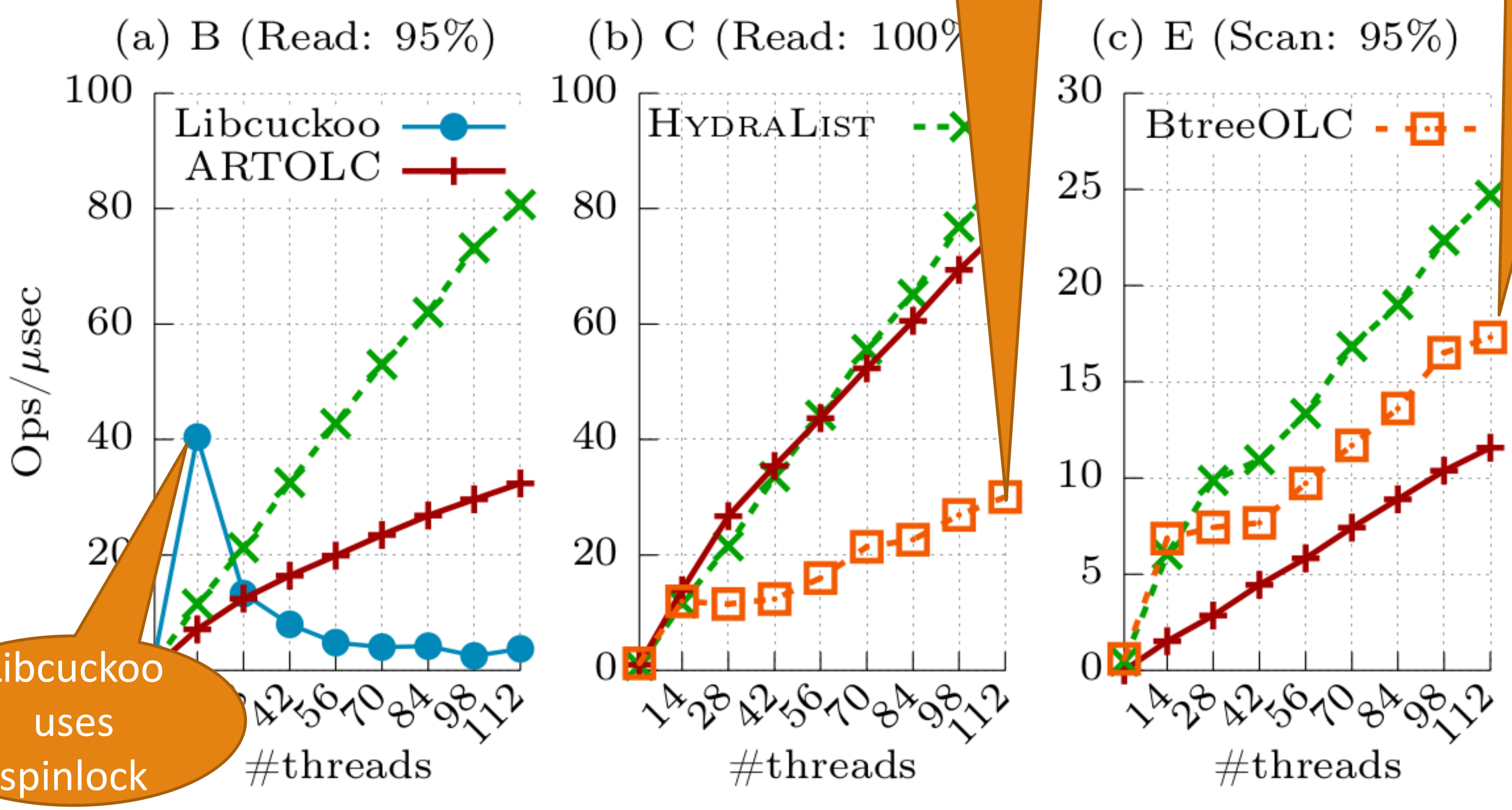
Index is the new bottleneck

- Since data is in memory, IO bottlenecks caused due to expensive disk accesses are avoided.
- Significant efforts on optimizing query execution has allowed compiled transactions to remove overhead of buffer management, latching, etc.
- A recent study of contemporary in-memory database systems shows that index lookup can contributes up to 94% of query execution time.[Kocberber, MICRO’13]

Motivation

Same SM but ART has efficient time complexity

Btree has cache efficient scan



Libcuckoo uses spinlock

Designing new Index Structure

- Efficient Time Complexity
- Low synchronization overhead
- Efficient Data Placement
  - Cache Efficiency and NUMA Awareness

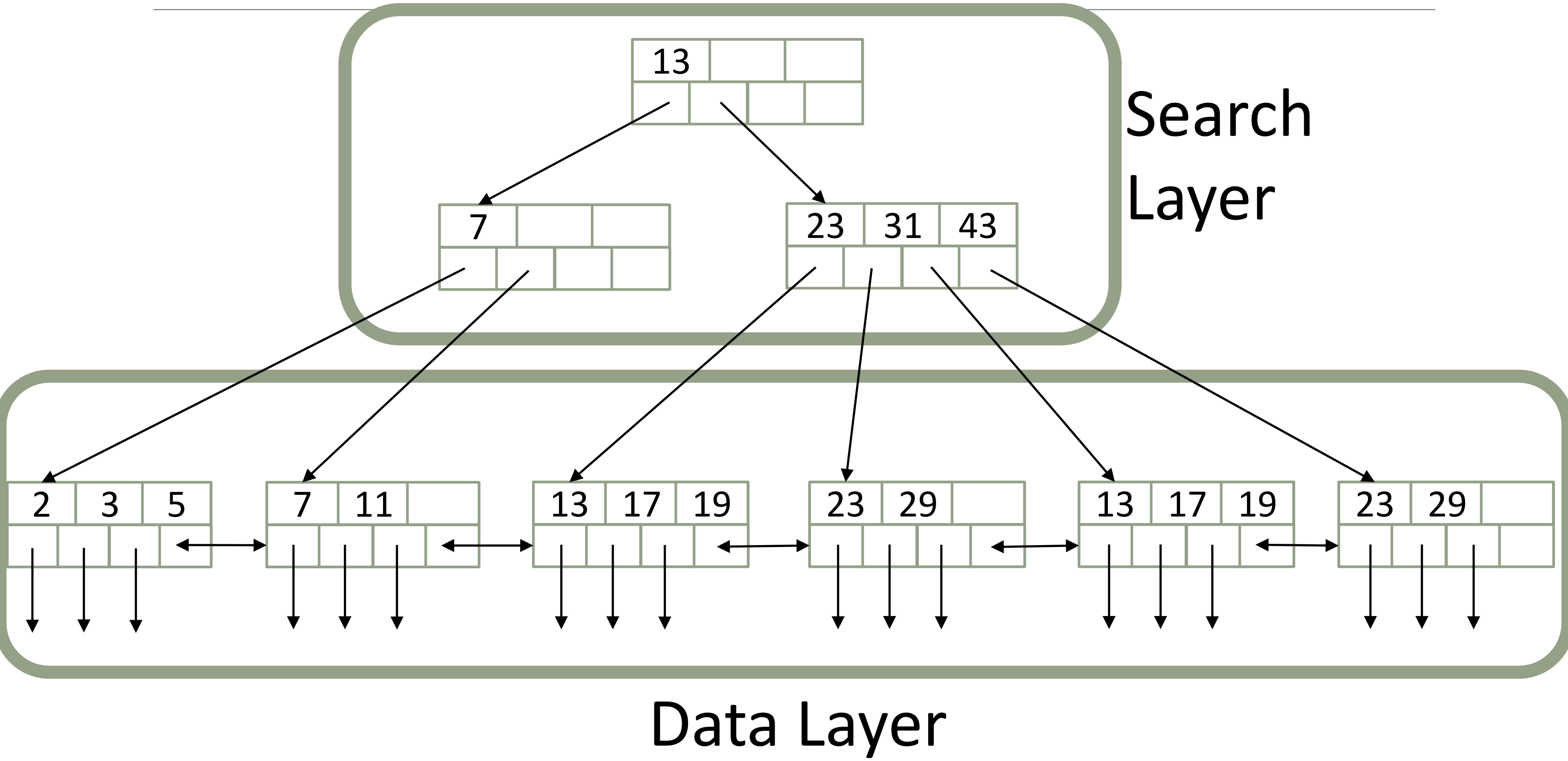
Key Idea

Most index structure can be decomposed into two components:

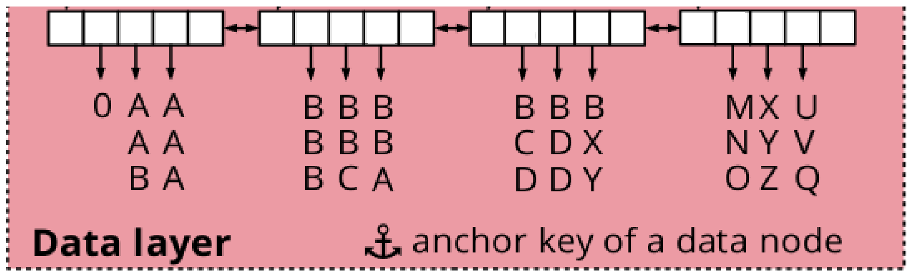
- Data Layer: Stores key and value pair
- Search Layer: Stores partial keys or a subset of keys which allows to accelerate search of keys in data layer.

If we modify the search algorithm of a key, we can allow inconsistency between search layer and data layer. This makes it possible to update search layer asynchronously.

Example: B+Tree



Hydralist Design

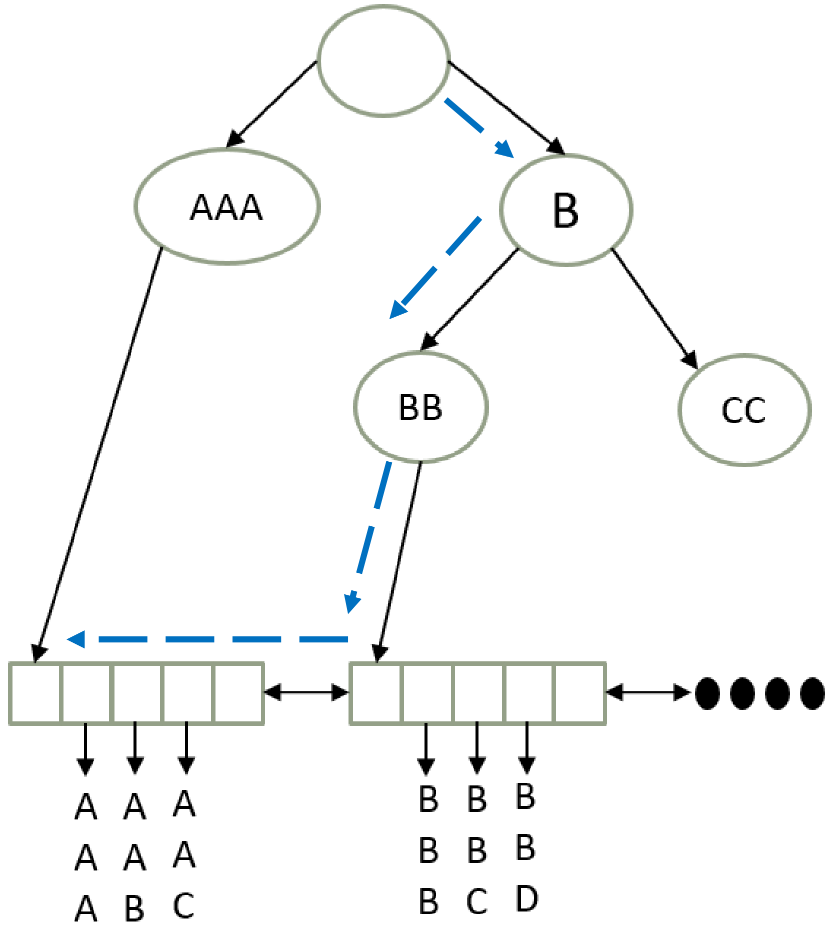


Searching in Hydralist

- ❑ Searching the search layer (Jump Node)
- ❑ Searching the data layer (Target Node)
- ❑ Search within a data node

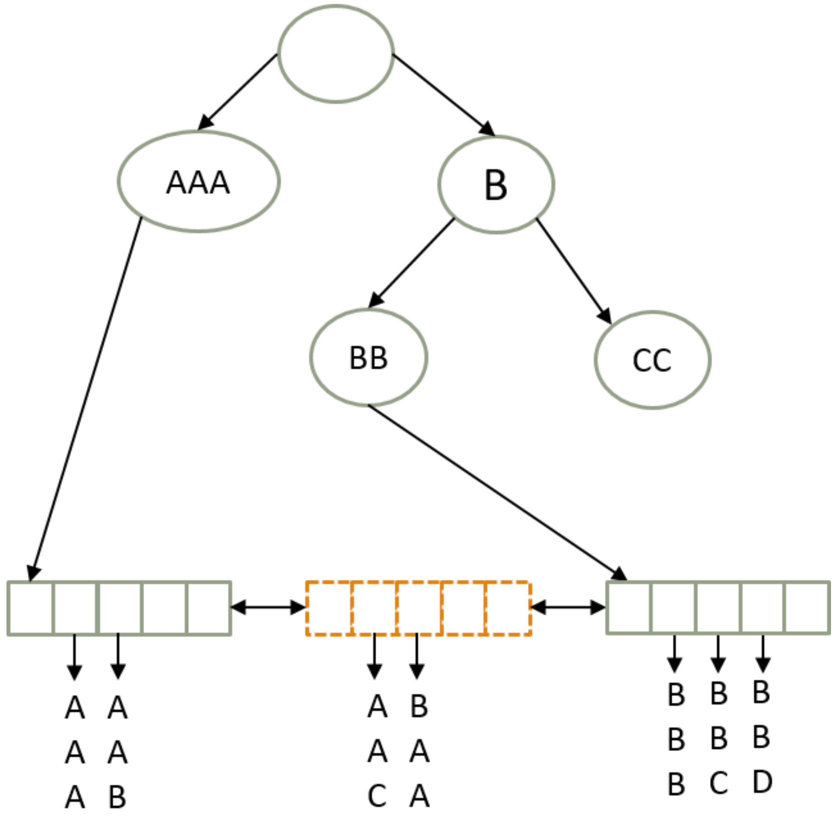
Insert in Hydralist

- Insert Key: BAA
- Find anchor key closest to BAA
- Check if current node is the target node.
- If not traverse, data layer



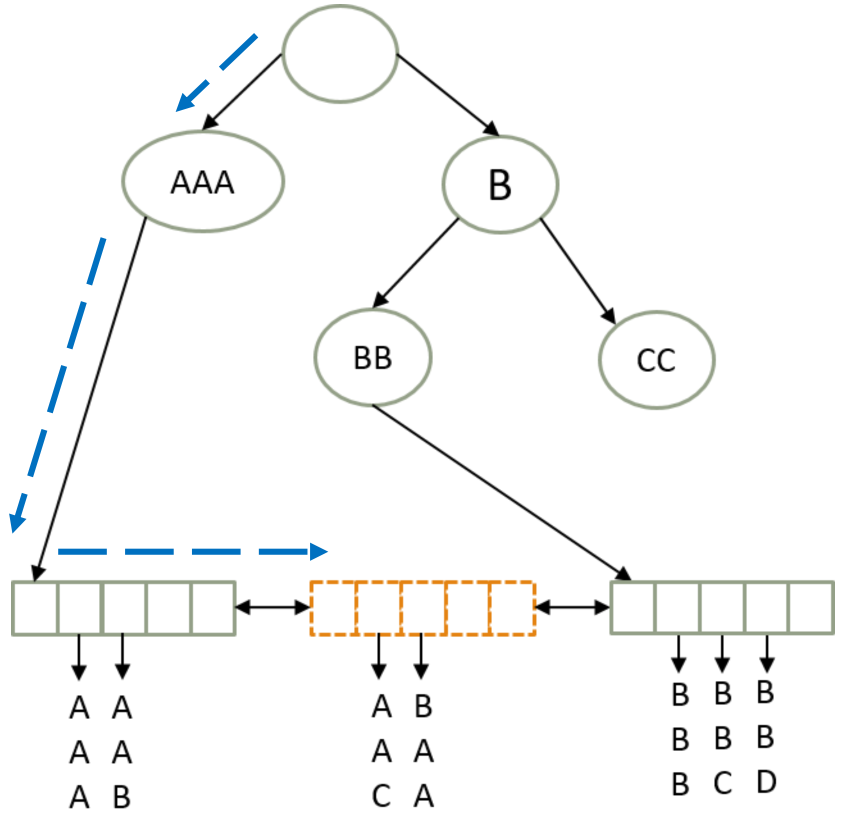
Insert in Hydralist

- Insert Key: BAA
- Find anchor key closest to BAA
- Check if current node is the target node.
- If not traverse, data layer
- Target Node is full so split



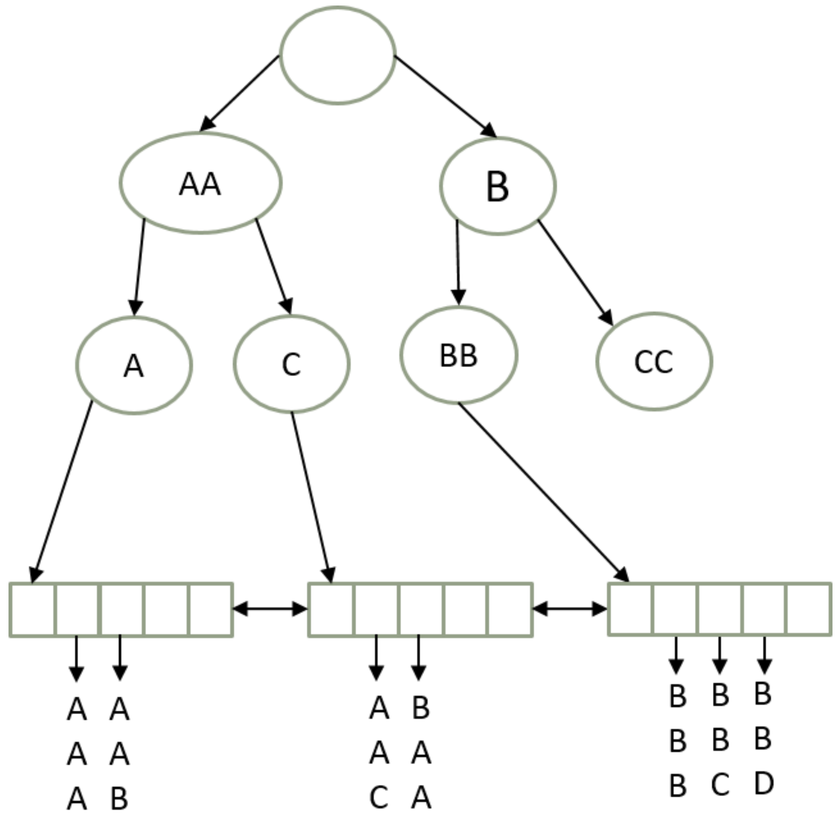
Concurrent Search

- Search Key: AAC
- Still visible to new thread, even though Search Layer is not update



Finally Search Layer is Updated

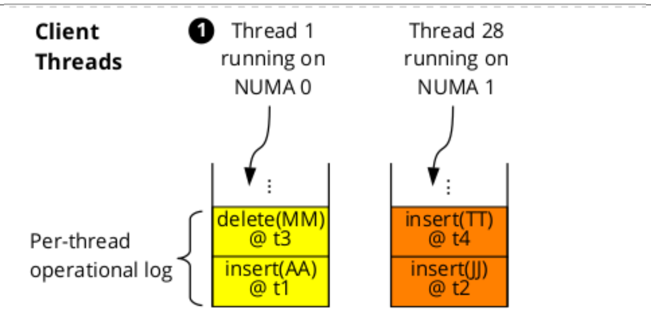
- Anchor key of new node is inserted into search layer



How search layer is updated

- Each thread has a operation log or OpLog
  - When a thread splits/merges a node, it adds the operation to OpLog
  - Also adds timestamp, anchor key of the node
- Combiner thread
  - A background thread which combines all Oplogs periodically
  - Sorts operations according to time stamps
- Updater thread
  - Background thread which applies merged log to search layer
  - This is done asynchronously
  - Does not affect the consistency of the index
- NUMA replication
  - Search layer can be made NUMA-local by assigning per NUMA-updater thread

Search Layer Update


Client Threads
1
Thread 1 running on NUMA 0
Thread 28 running on NUMA 1
Per-thread operational log
delete(MM) @ t3
insert(AA) @ t1
insert(TT) @ t4
insert(JJ) @ t2

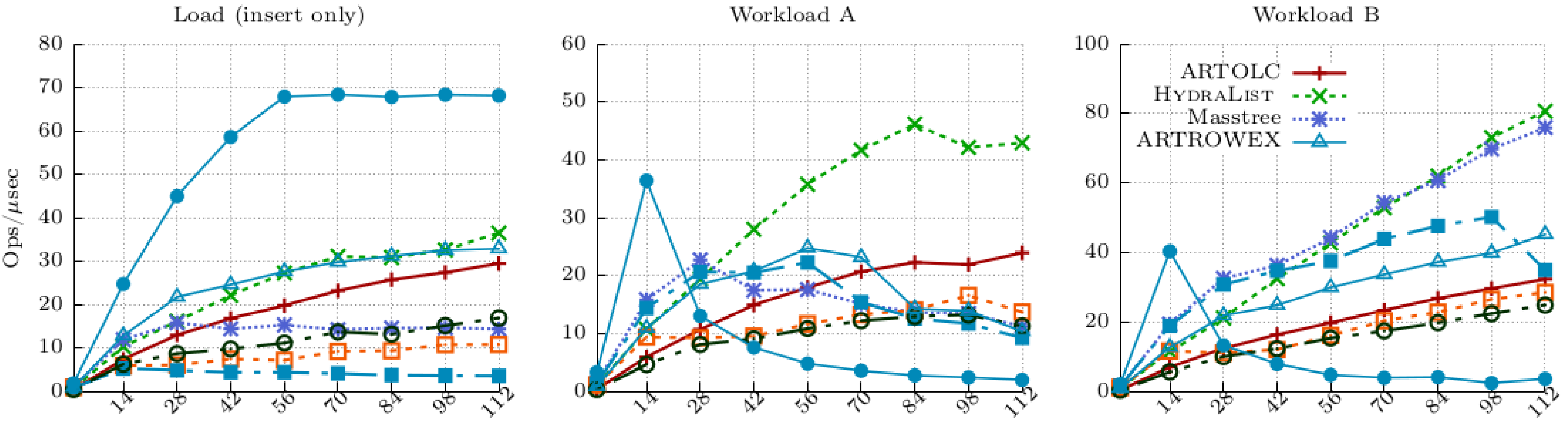
Other Details

- Design of data node
- Storing key hash instead of sorted keys
- SIMD to accelerate search within node
- Optimistic locking

Evaluation

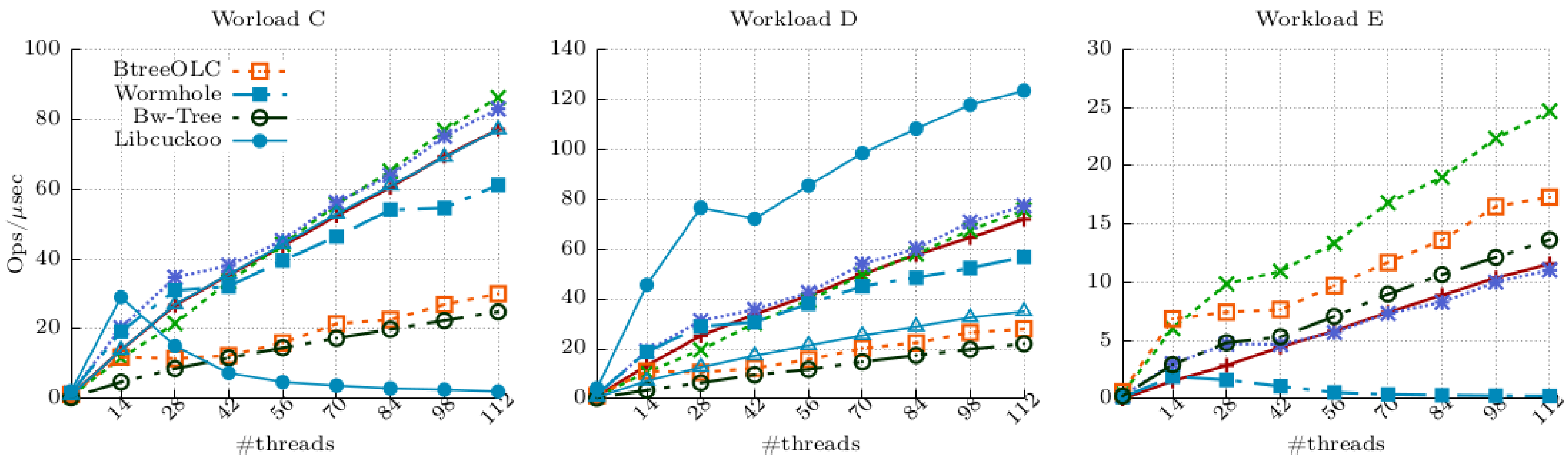
- Intel Xeon Platinum 8180 Server
- 112 Cores
- YCSB Benchmark
- 89 Million string keys (email addresses)
- 50 million operations

YCSB



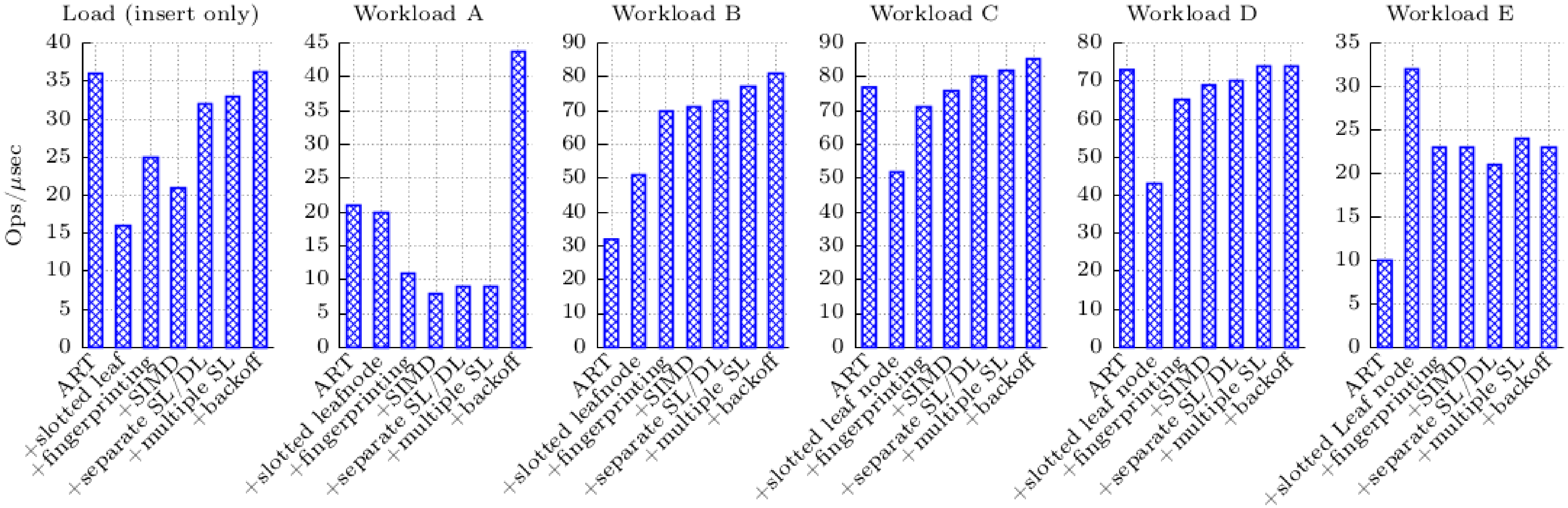
Load (insert only)
Workload A
Workload B
Ops/μsec
ARTOLC
HYDRALIST
Masstree
ARTROWEX
0
10
20
30
40
50
60
70
80
100
14
28
42
56
70
84
98
112

YCSB



Worload C
Workload D
Workload E
BtreeOLC
Wormhole
Bw-Tree
Libcuckoo
Ops/μsec
#threads
0
14
28
42
56
70
84
98
112

Factor Analysis

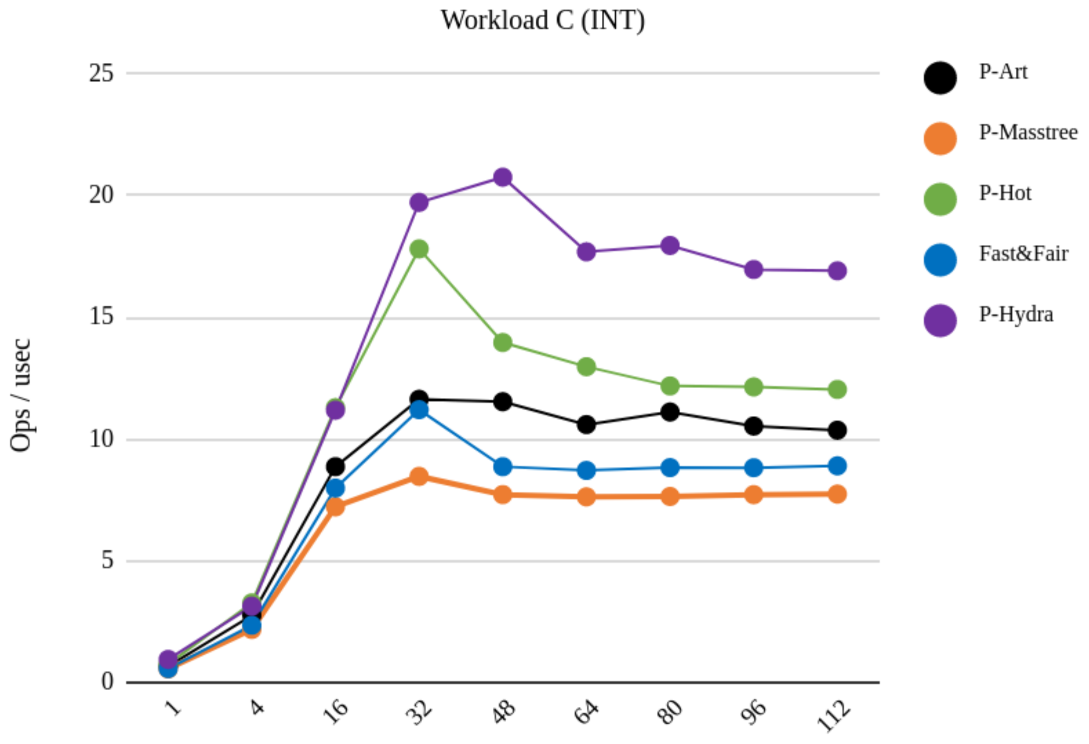


Conclusion

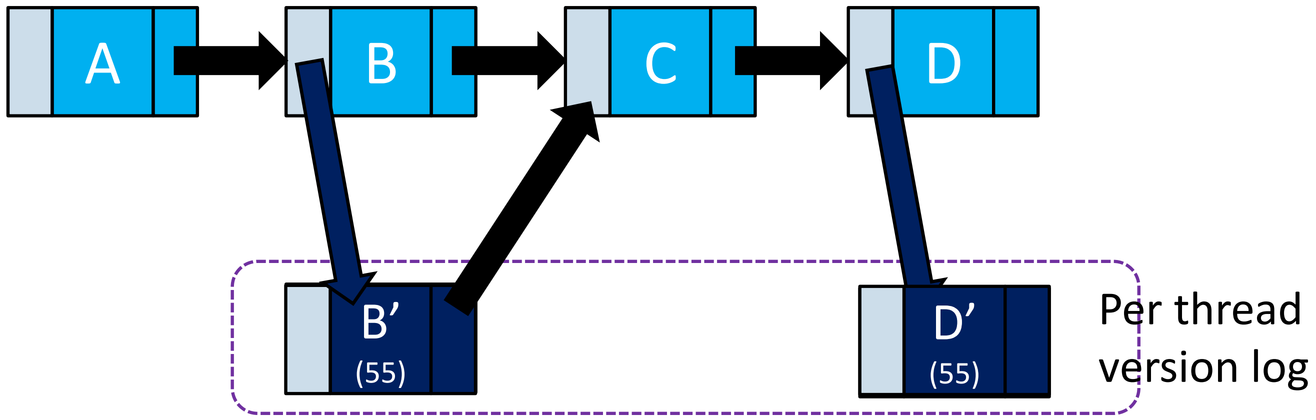
- ❑ Multicore scaling is difficult because of unprecedented levels of parallelism
- ❑ Possible solution: Asynchronous Work.
  - ❑ Will require redesigning algorithm to allow delegation
- ❑ MV-RLU: a new scalable synchronization mechanism
  - ❑ Modification: Add multi-versioning to RLU
  - ❑ Result: Remove synchronous waiting from critical path
- ❑ Hydralist: a new scalable index structure
  - ❑ Modification: Separate Search Layer and Data Layer and modify search algorithm
  - ❑ Result: Remove update to search layer from critical path

Future Work

- Timestone: MV-RLU for NVM (Accepted at ASPLOS, 2020)
- NVM-Hydralist: Preliminary Performance is better than other index
- RDMA-Hydralist



Multi-pointer update

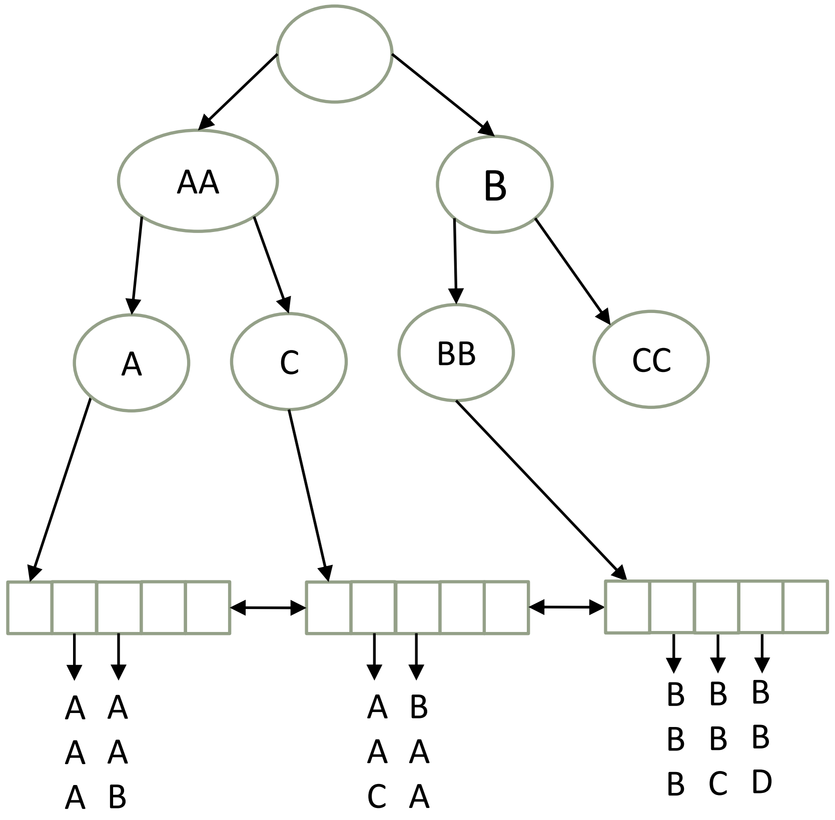


Snapshot Isolation

- MVRLU Serializable Snapshot Isolation
- SSI works well for any application which can tolerate stale read
- RCU is widely used which means lot of application for MV-RLU

Log size

- Memory in computer systems is increasing
- Persistent memory can increase total main memory significantly
- Log size is a tuning parameter.


AA
B
A
C
BB
CC
A A A
A A B
A A C
B A A
B B B
B B C
B B D