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

Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhav K. Ramanathan, Changwoo Min

Published in ACM ASPLOS 2019

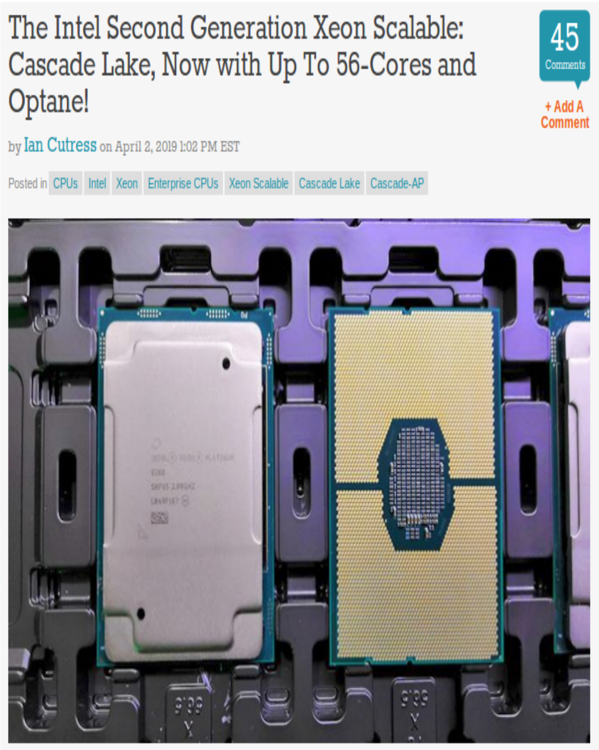
Jaeho Kim
Gyeongsang National University (GNU)
jaeho.kim@gnu.ac.kr

Contents

- Motivation
- What is RCU
- What is RLU
- Design of MV-RLU
- Evaluation
- After MV-RLU
- Conclusion

CPU-core Count Continues to Rise..

→ Many-core Era


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

45 Comments

+ Add A Comment

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

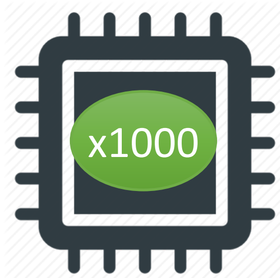

Marvell Announces ThunderX3: 96 Cores & 384 Thread 3rd Gen Arm Server Processor

45 Comments

+ Add A Comment

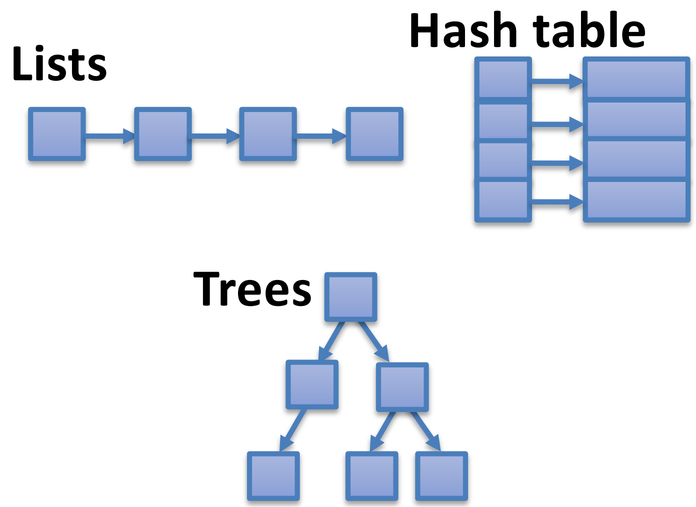
by Andrei Frumusanu on March 16, 2020 8:30 AM EST

Posted in Servers CPUs Marvell Arm Enterprise Enterprise CPUs Cavium ThunderX3



Concurrency algorithms are essential building block

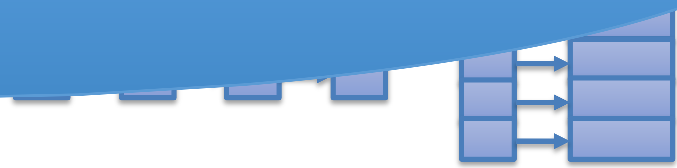
- Data structures are essential for the most applications
- Synchronization mechanisms are essential building block of today's application

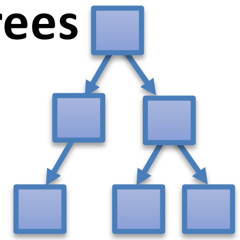


Concurrency algorithms are essential building block

- Data structures are essential for the most applications
- Synchronization mechanisms are essential building block of today's applications

Making scalable concurrent data structures
is key for improving system performance

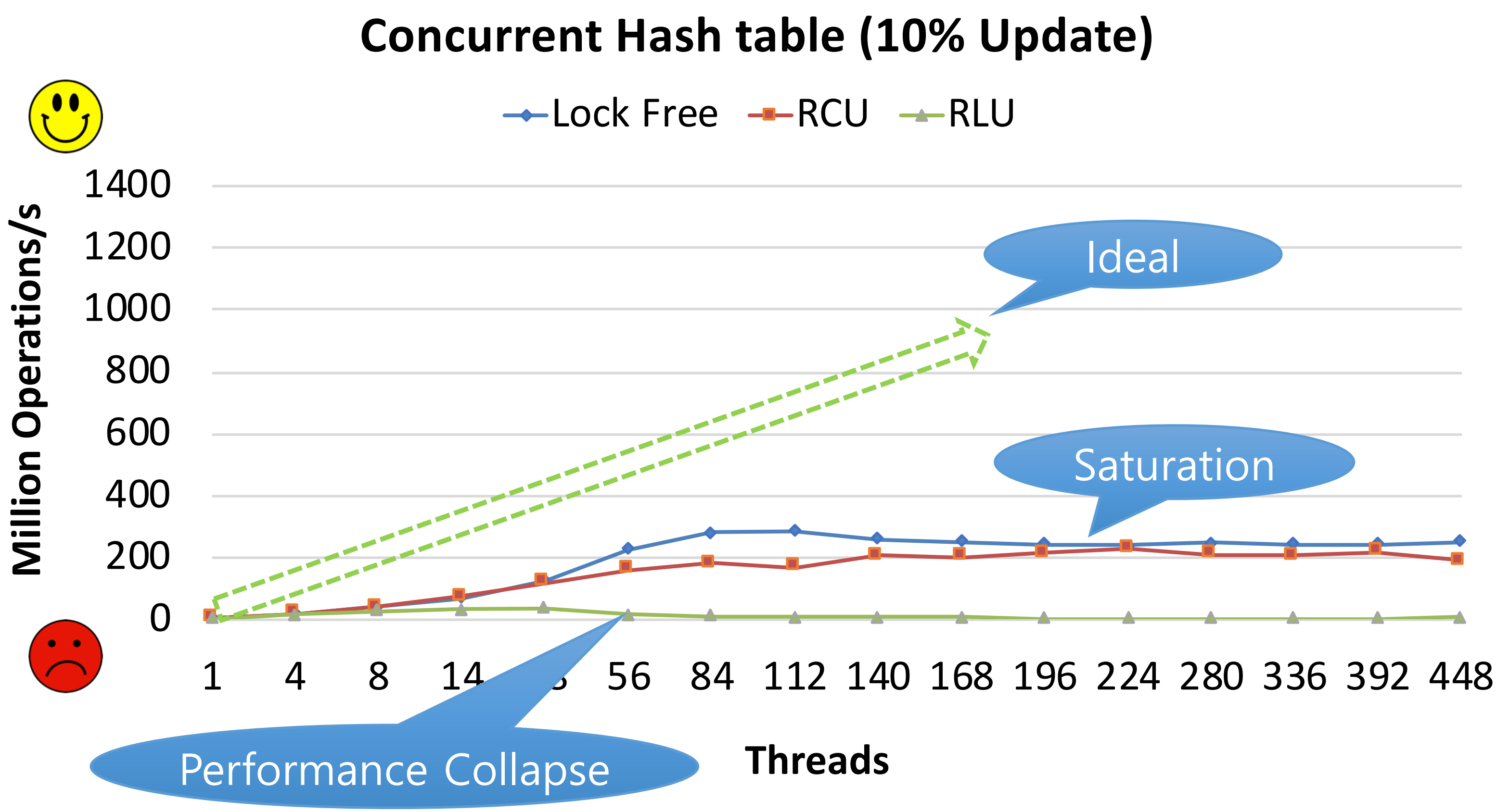




Synchronization Approaches

- **Blocking**
  - Spinlock
  - Ticket lock
  - Mutex
  - Read-write lock
  - Etc.
- **Non-blocking**
  - Lock-free
  - Software Transactional Memory (STM)
  - RCU, RLU, and MV-RLU
  - Etc.

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 ☹
  - Programming with RCU is not easy
  - Difficult to apply RCU to complex data structures
- Good performance only for read-intensive workloads ☹

Basic Operations of RCU

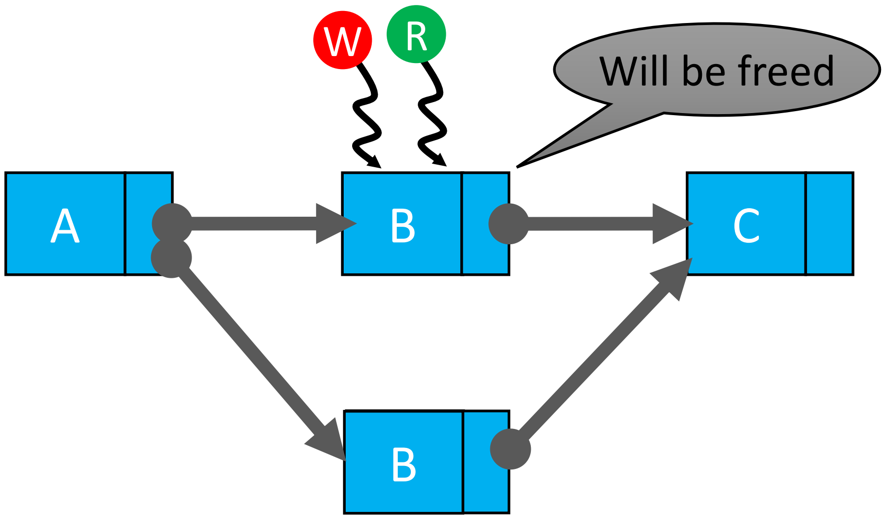
1) Copy and Update node B

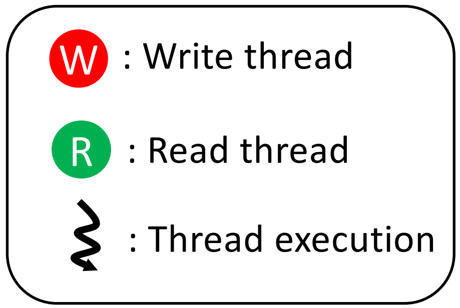
2) During the update, another thread can still read the old node B

3) Previous node points the new node by updating a single pointer

- Make B' reachable and B unreachable

4) Node B will be freed when there are no threads to read





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

- **RCU + STM (Software Transactional Memory)**

- An extension to RCU
  - Readers never block ☺
  - Support multi-pointer atomic updates ☺
  - Provide better programmability with DB transaction-like APIs ☺

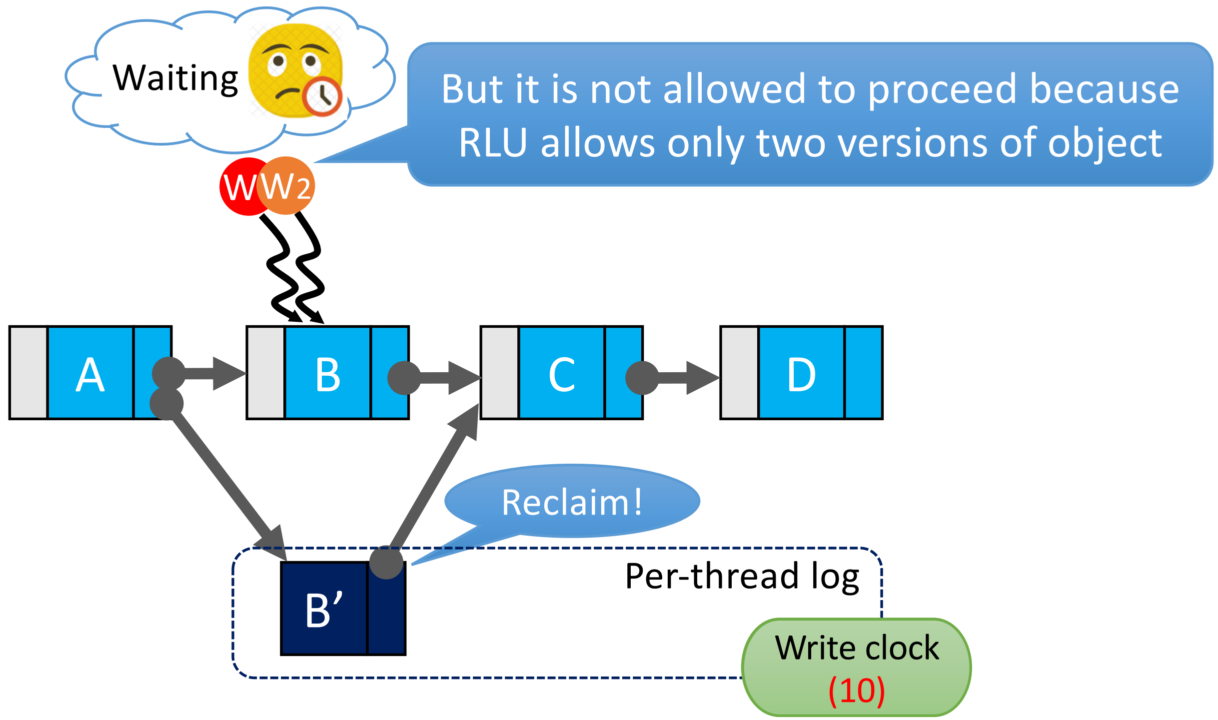
- Key idea: Use global clock and per-thread log to make multiple updates atomically visible

Why does not RLU scale?

1. A thread modify node B
   - Create a new version of B in per-thread log

2. The thread commit the modifies
   - Update the write clock
   - Mean that updates are atomically visible

3. Second thread tries to modify node B again
   - Wait for reclamation of node B’



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

Contributions of this study

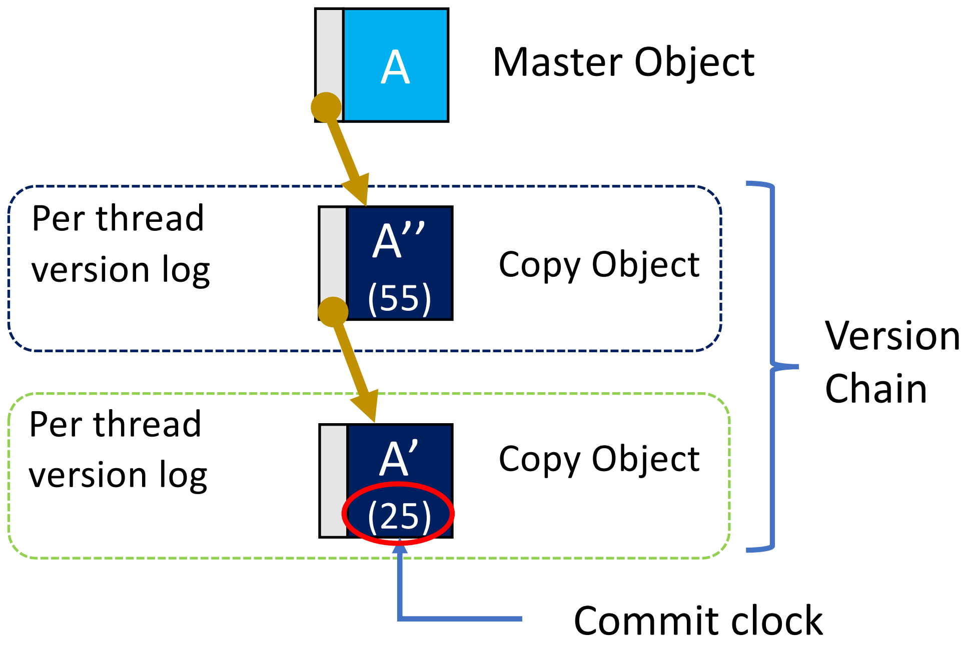
- Multi-Version Read-Log-Update (MV-RLU)
  - Allow multiple versions to exists at same time
  - Removes synchronous waiting from critical path

- Scaling Multi-Versioning
  - Concurrent and autonomous garbage collector

Design: Overview

- Master object
  - Have zero or more copy objects

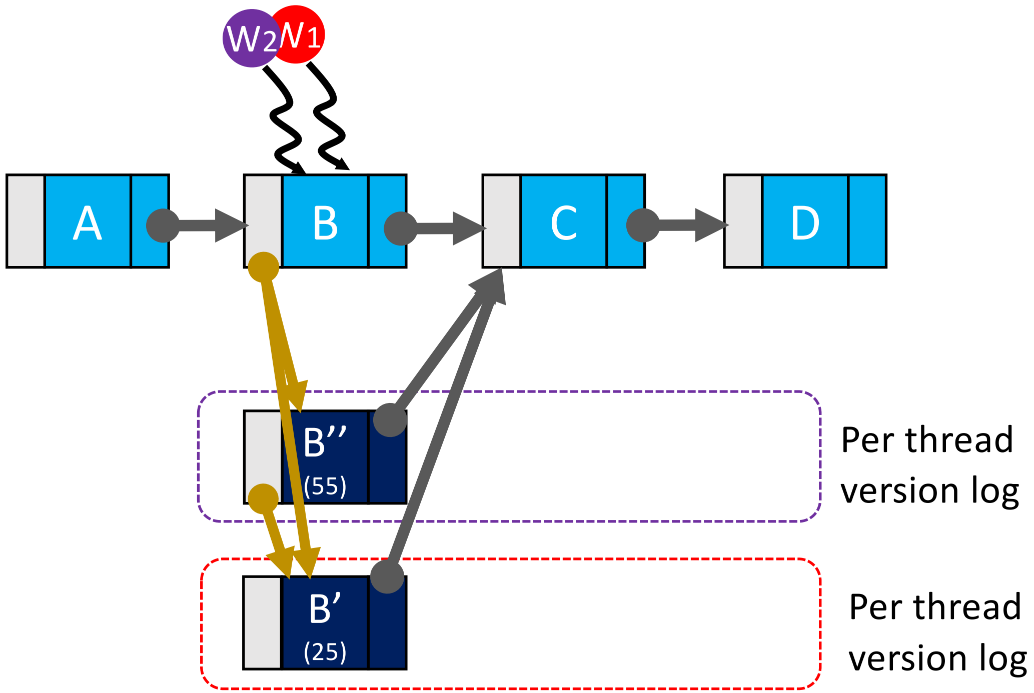
- Copy object
  - Timestamp (clock) when committed
  - Pointer of next older version
  - Stored in per-thread log



Updates in MV-RLU

1. A thread updates node B
   - Creates a new copy of B with commit clock 25

2. Second thread updates B again
   - Create a new copy of object B with commit clock 55



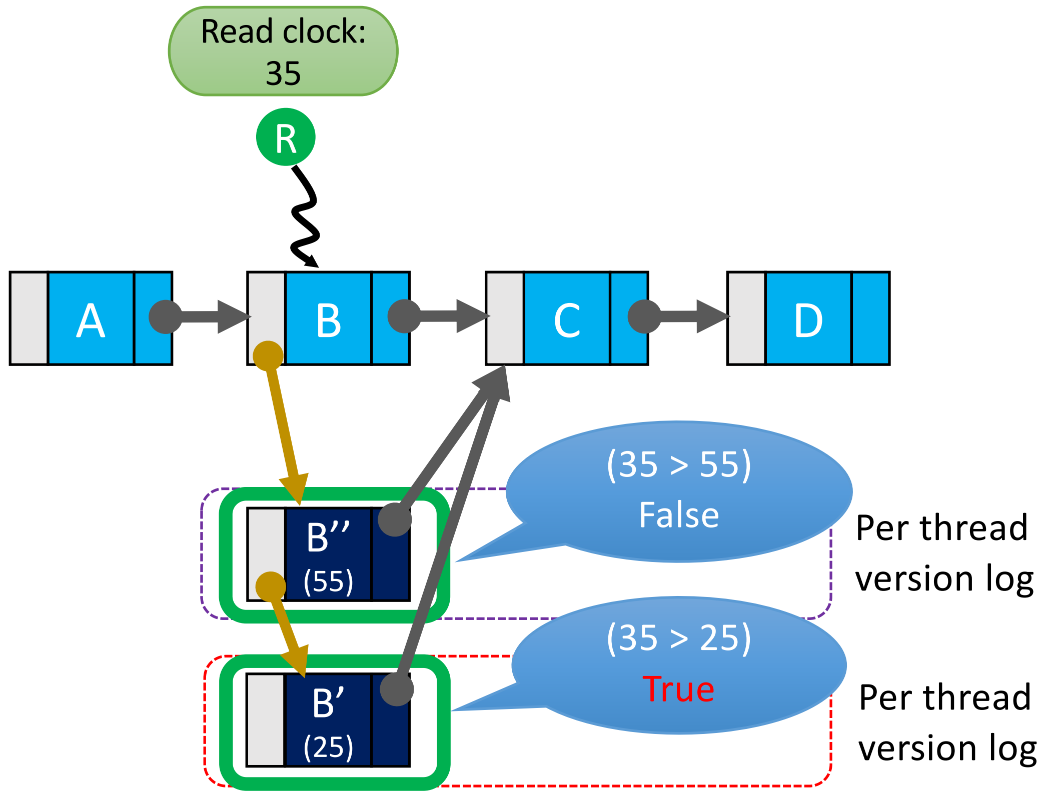
A thread does not need to synchronize with other read/write threads in critical section

Reads in MV-RLU

1. Reader note the global clock at start of critical section

2. Reader traverses the version chain
   - **First node** which satisfies the criteria
   - **Reader clock > commit clock**

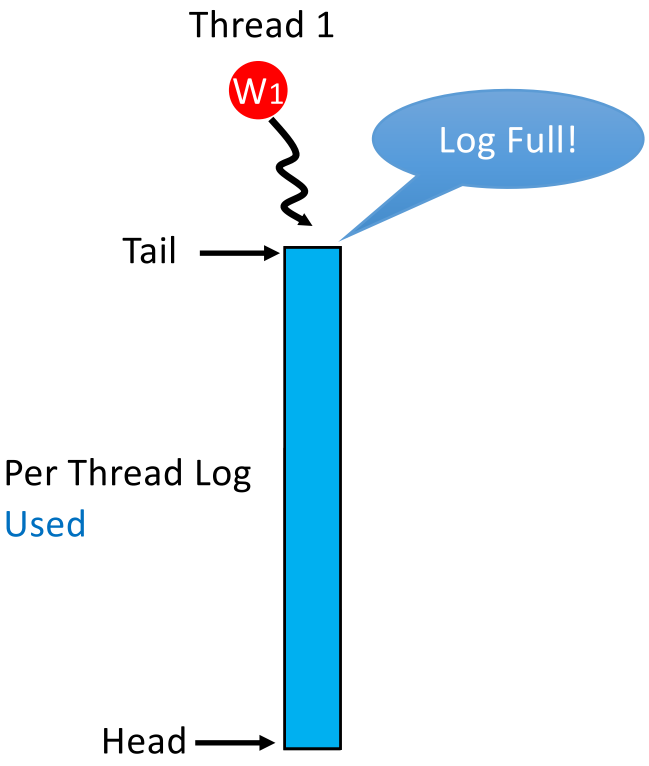
3. B' with commit clock 25 is the right object



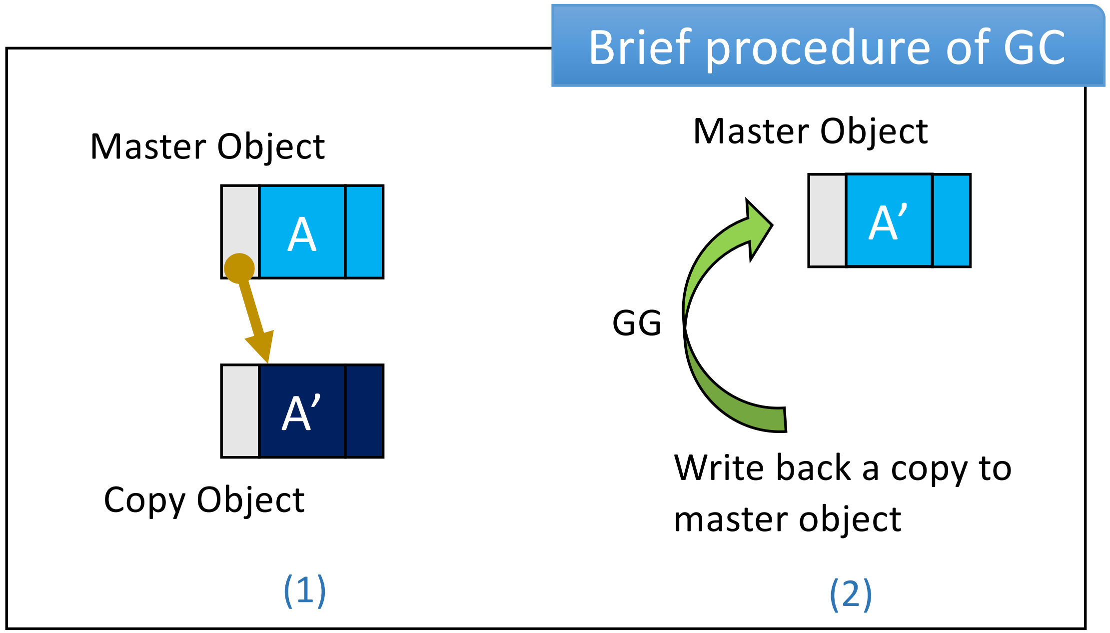
All read threads can read a proper version of object concurrently (Reader never blocks)

Memory is limited!

→ Garbage Collection (GC) is Required



- Garbage collection
  - Obsolete version should be properly reclaimed
  - GC should be scalable



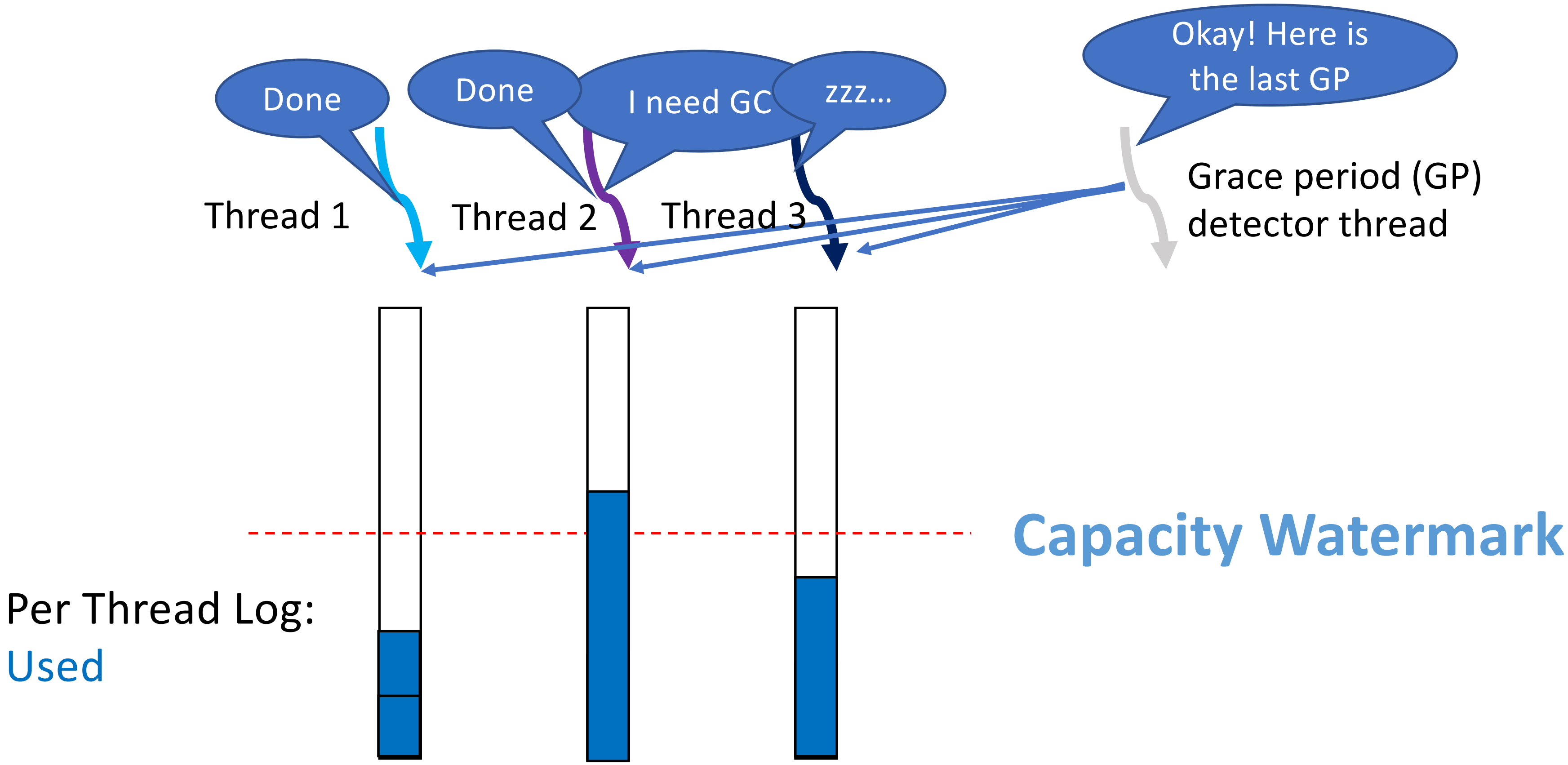
Challenges to Garbage Collection

- How to detect obsolete version in a scalable manner?
  - Reference counting and hazard pointer do not scale well
- How to reclaim obsolete versions?
  - Single thread is insufficient
- When to trigger garbage collection?
  - Eager: Triggering too often wastes CPU cycles
  - Lazy: Increases version chain traversal cost

Solutions for Challenges

- Detecting obsolete version
  - Use grace period detection technique like RCU to find safely reclaimable versions
  - Grace Period (GP): Time interval in which every thread has been the outside critical section
- Scalable garbage collector
  - Every thread reclaim their own log
  - Cache friendly
- Autonomous garbage collector
  - Detect reader's version traverse pattern
  - Trigger GC dynamically according to the reader's pattern

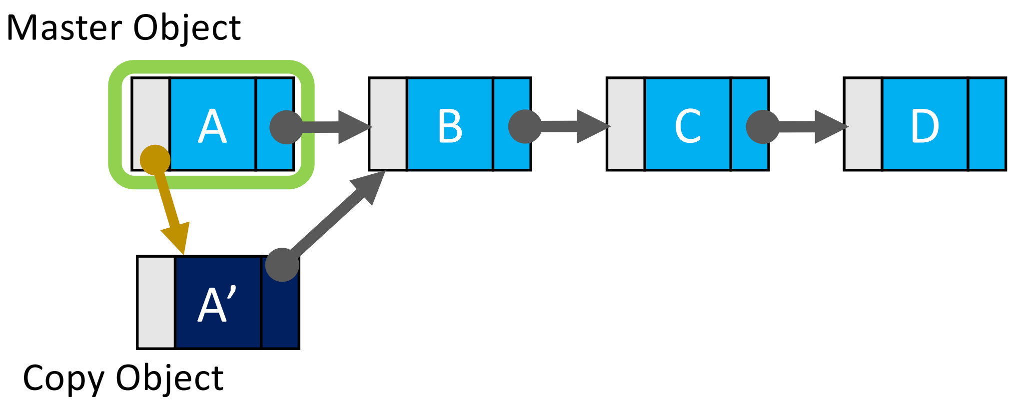
GC Example



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

Capacity Watermark is not sufficient

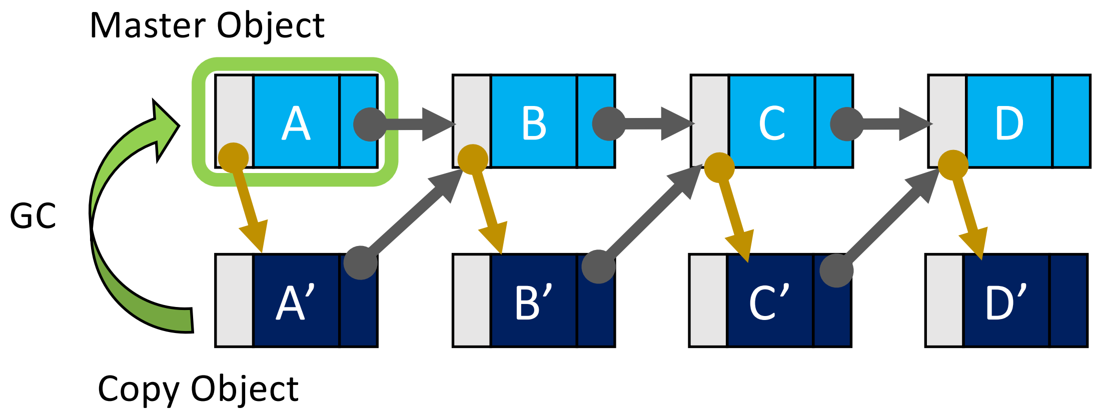
- Capacity watermark will not be triggered in read mostly workload



Read mostly workload: one copy object

Worst Case of Version Traversal

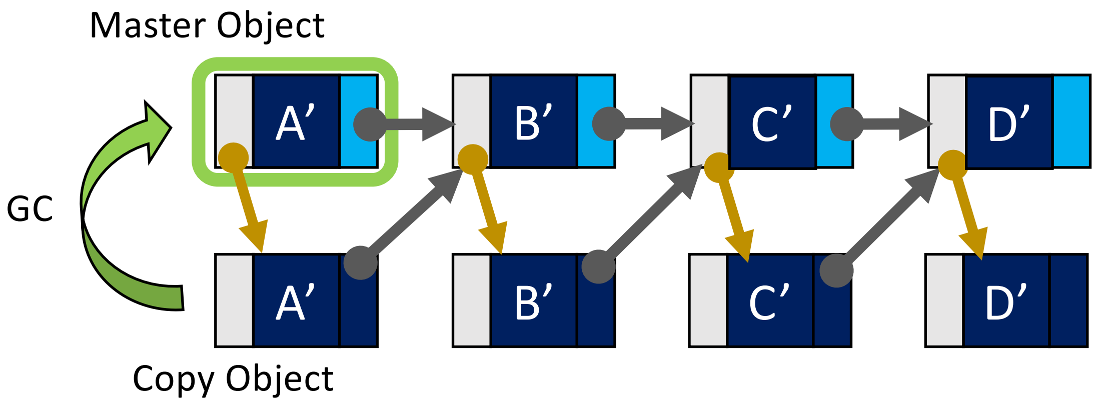
- In worst case, every object read will require access to version chain
- To alleviate the cost, garbage collector should be clever



Pointer chasing slow down read performance due to cache pollution

Reduced Version Traversal Cost

- After the GC, readers can traverse only master objects



Dereference Watermark

to reduce version traversal

- To reduce pointer chasing, we employ dereference watermark
- Readers check if they are accessing version chain too often
- If yes, trigger GC for the write-back

Combination of capacity watermark and dereference watermark makes GC trigger workload agnostic

More detail

- Scalable timestamp allocation
- Version management
- Proof of correctness
- Implementation details

Session: Synchronization ASPLOS'19, April 13–17, 2019, Providence, RI, USA

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, RLU, software transactional memory (STM), and lock-free approaches, on concurrent data structures and real-world applications (database concurrency control and in-memory key-value store). Our evaluation shows that MV-RLU significantly outperforms other techniques for a wide range of workloads with varying contention levels and data-set size.

CCS Concepts • **Computing methodologies** *Concurrent algorithms*.

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 even more difficult because of two reasons: 1) The increase in unprecedented levels of hardware parallelism by virtue of recent advances of manycore processors. For instance, a recently released AMD [57, 58], ARM [22, 63], and Xeon servers [11] can be equipped with up to at most 1,000 hardware threads.[1] 2) With such many cores, a small, yet critical serial section can easily become a scalability bottleneck as per the reasoning of Amdahl's Law.

Please refer to the paper for details

Evaluation Question

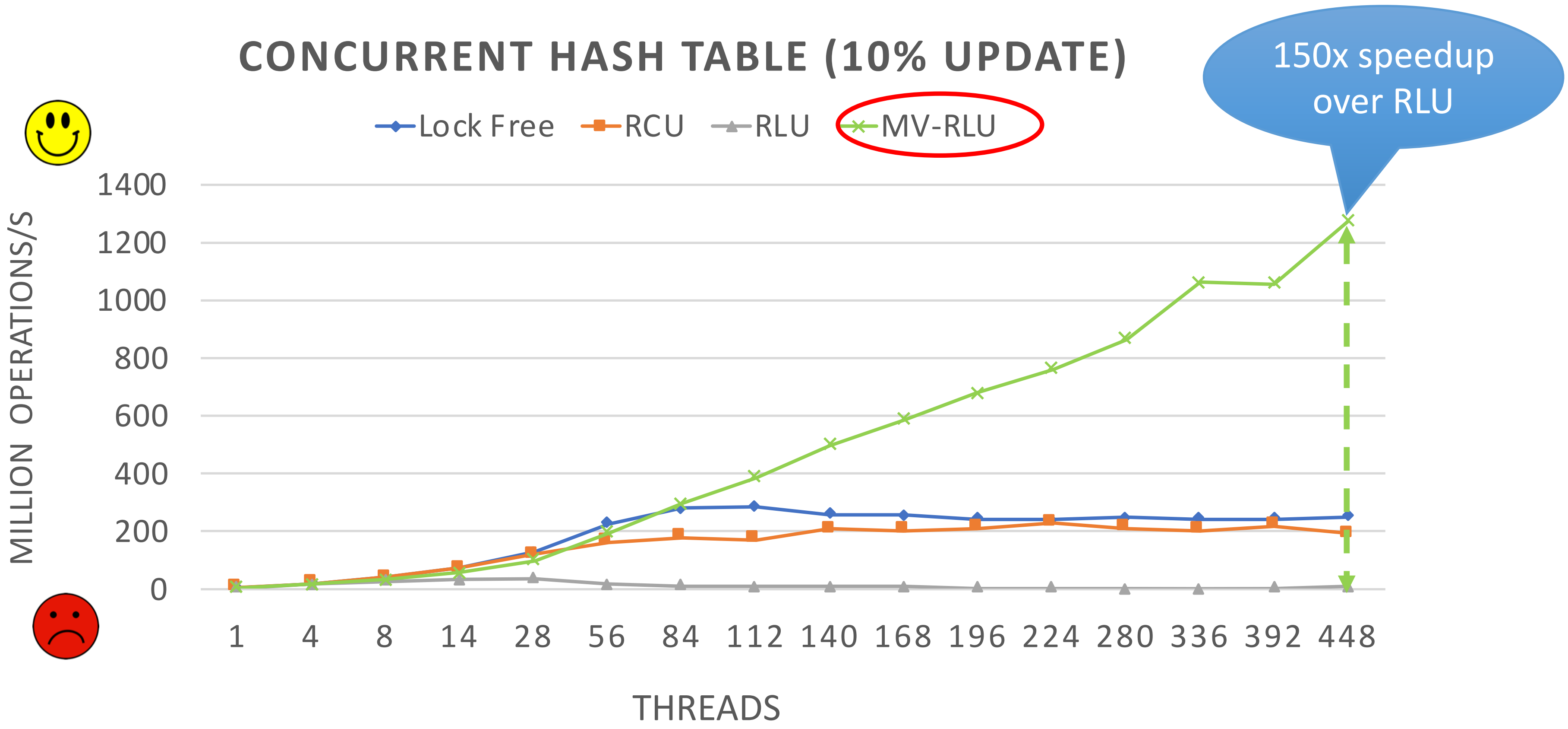
- Does MV-RLU scale?
- What is the impact of our proposed approaches?
- What is its impact on real-world workloads?

Evaluation Setup

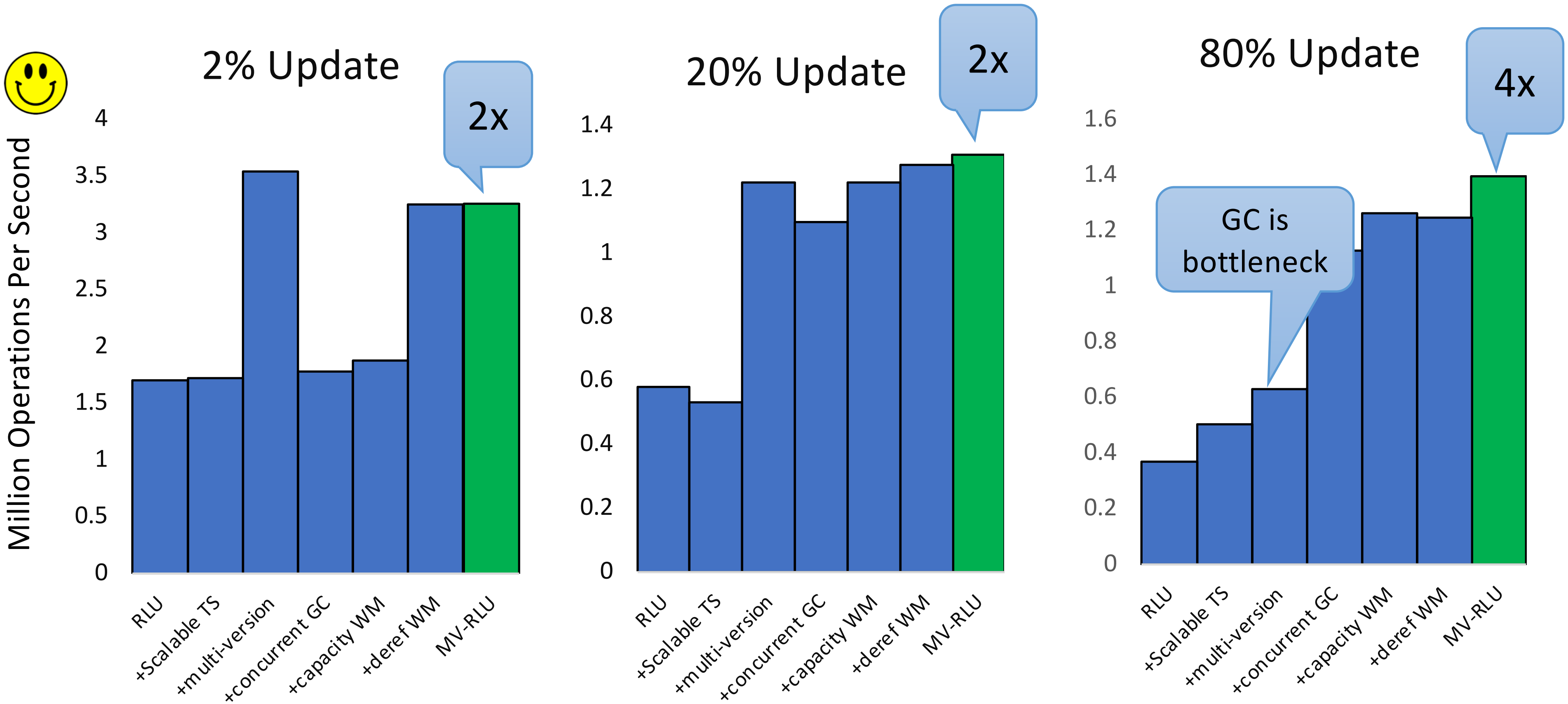
- Evaluation Platform
  - Supermicro server: 448-core on 8 sockets (with hyperthreading)
  - Intel Xeon Platinum 8180
  - DRAM size: 337 GB
  - OS: Linux 4.17.3

- Workloads
  - Microbenchmark:
    - random access on linked-lists, hash tables, and binary trees
  - Kyoto Cabinet benchmark: key-value database workload

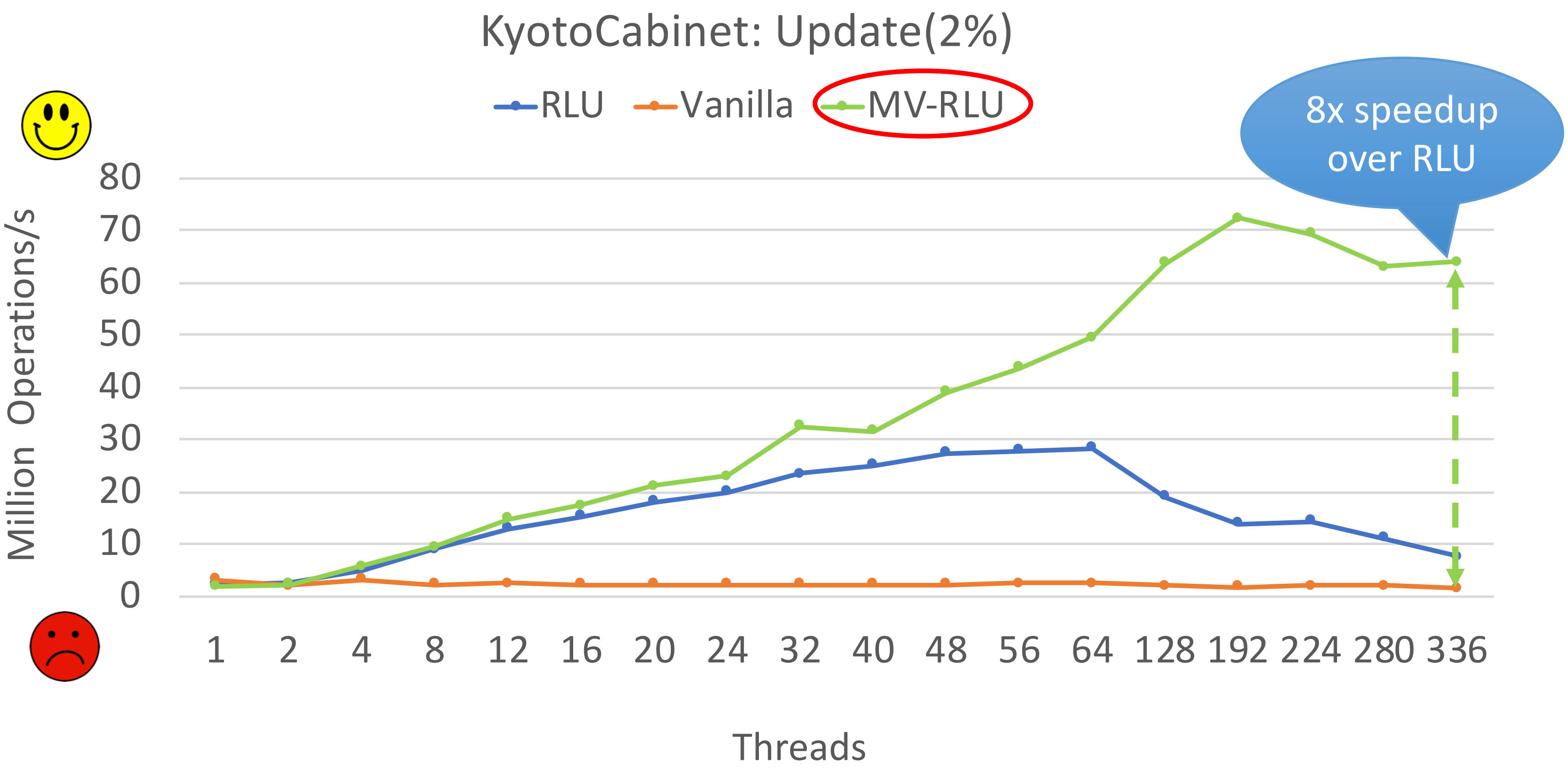
Microbenchmark Result



Factor Analysis

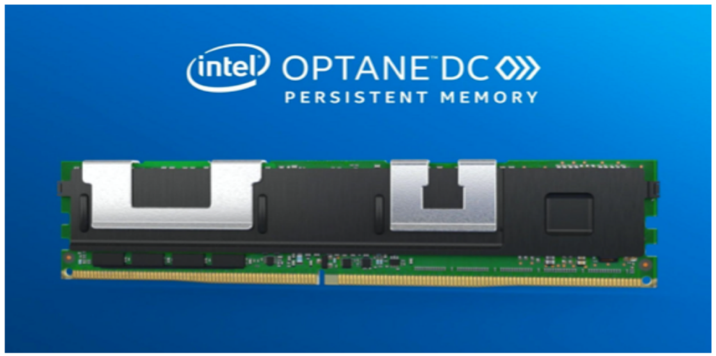


Key Value Benchmark



After MV-RLU

Can we leverage MV-RLU for persistent memory?



Durable Transactional Memory (DTM)

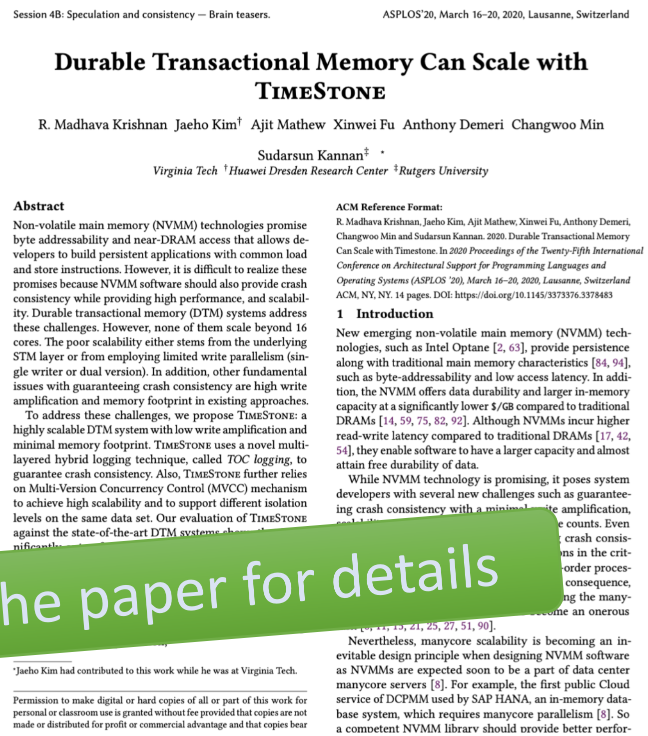
- DTMs are software framework supporting ACID properties
- DTMs make persistent memory (PM) programming easier
- Relieves the burden on PM application developers

- Existing DTMs have serious problems
  - Existing DTMs: PMDK, DUDETM[ASPLOS17], Romulus[SPAA18]
  - Poor Scalability
  - High Write Amplification (up to 6x)

Our proposed DTM

- A scalable and high performance DTM leveraging MV-RLU

- Our Solution: **TimeStone**
  - published in ACM ASPLOS20



Session 4B: Speculation and consistency — Brain teasers. ASPLOS'20, March 16–20, 2020, Lausanne, Switzerland

Durable Transactional Memory Can Scale with TimeStone

R. Madhava Krishnan Jaeho Kim† Ajit Mathew Xinwei Fu Anthony Demeri Changwoo Min
Sudarsun Kannan‡ *
Virginia Tech †Huawei Dresden Research Center ‡Rutgers University

Abstract

Non-volatile main memory (NVMM) technologies promise byte addressability and near-DRAM access that allows developers to build persistent applications with common load and store instructions. However, it is difficult to realize these promises because NVMM software should also provide crash consistency while providing high performance, and scalability. Durable transactional memory (DTM) systems address these challenges. However, none of them scale beyond 16 cores. The poor scalability either stems from the underlying STM layer or from employing limited write parallelism (single writer or dual version). In addition, other fundamental issues with guaranteeing crash consistency are high write amplification and memory footprint in existing approaches.

To address these challenges, we propose TimeStone: a highly scalable DTM system with low write amplification and minimal memory footprint. TimeStone uses a novel multi-layered hybrid logging technique, called *TOC logging*, to guarantee crash consistency. Also, TimeStone further relies on Multi-Version Concurrency Control (MVCC) mechanism to achieve high scalability and to support different isolation levels on the same data set. Our evaluation of TimeStone against the state-of-the-art DTM systems ...

*Jaeho Kim had contributed to this work while he was at Virginia Tech.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear

ACM Reference Format:
R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min and Sudarsun Kannan. 2020. Durable Transactional Memory Can Scale with Timestone. In *2020 Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland* ACM, NY, NY. 14 pages. DOI: https://doi.org/10.1145/3373376.3378483

1 Introduction

New emerging non-volatile main memory (NVMM) technologies, such as Intel Optane [2, 63], provide persistence along with traditional main memory characteristics [84, 94], such as byte-addressability and low access latency. In addition, the NVMM offers data durability and larger in-memory capacity at a significantly lower $/GB compared to traditional DRAMs [14, 59, 75, 82, 92]. Although NVMMs incur higher read-write latency compared to traditional DRAMs [17, 42, 54], they enable software to have a larger capacity and almost attain free durability of data.

While NVMM technology is promising, it poses system developers with several new challenges such as guaranteeing crash consistency with a minimal write amplification, ... e counts. Even ... crash consis- ... ns in the crit- ... -order proces- ... consequence, ... ng the many- ... ome an onerous ... 21, 25, 27, 51, 90].

Nevertheless, manycore scalability is becoming an inevitable design principle when designing NVMM software as NVMMs are expected soon to be a part of data center manycore servers [8]. For example, the first public Cloud service of DCPMM used by SAP HANA, an in-memory database system, which requires manycore parallelism [8]. So a competent NVMM library should provide better perfor-

Please refer to the paper for details

Conclusion

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

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

Thank you!