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*.

Keywords multi-version; concurrency control; garbage collection; synchronization.

ACM Reference Format:
Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan 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.

Although, there have been significant research efforts to design scalable synchronization mechanisms, they either do not scale [17, 18], only support certain types of workloads [3, 42, 45, 55], or are difficult to use [45]. Figure 1 shows the performance of hash tables for a read-dominant workload with 1,000 elements. Unfortunately, none of the existing approaches scale beyond 100 cores and each approach suffers from different bottlenecks and limitations, as summarized in Table 1.

Current state-of-the-art work can be broadly categorized into linearizable and non-linearizable approaches. Linearizable approaches include lock-based, lock-free, delegation-style, and software transactional memory (STM) approaches. Lock-based approaches suffer from limited parallelism due to the mutual-exclusive nature of locking [30]. Lock-free approaches provide higher parallelism but they suffer from cache-line bouncing (e.g., frequent `compare-and-swap` retries) for memory hot spots and have additional memory

*Joint first authors

[1] In this paper, we interchangeably use a logical core, core, and hardware thread.

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

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304040

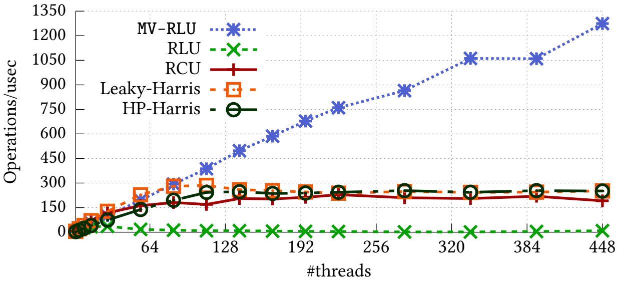
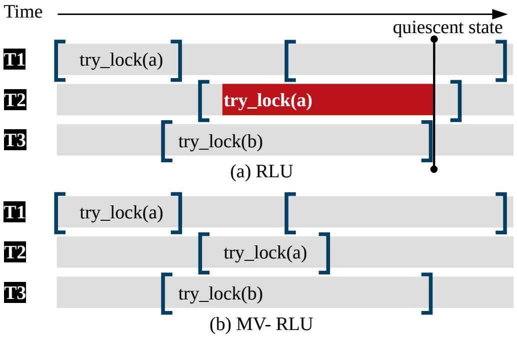


Figure 1. Performance comparison of concurrent hash tables having 1,000 elements with a load factor 1. Hash table access follows 80–20 Pareto distribution with 10% update. None of the existing approaches scale beyond 100 cores, except MV-RLU. Refer the setup detail in §6.

reclamation overhead [43, 45, 47]. Delegation and combining approaches use single thread execution to reduce the cost of synchronization, such as cache-line bouncing, but the single-thread execution itself becomes a bottleneck, especially during long running workloads (e.g., look up a large hash table) [55]. STM provides excellent programmability but its centralized metadata management (e.g., lock table) often becomes a scalability bottleneck [17].

In non-linearizable category, read-copy-update (RCU) approach is catered towards read-mostly workloads [45]. RCU provides excellent performance for read-mostly workloads because a writer does not block readers, while modifying an object. Due to its excellent performance, several sub-systems heavily rely on RCU in the Linux kernel [44] and even in user-space applications [12, 21]. However, RCU programming is notoriously difficult because all the changes need to properly use single pointer update [42]. To mitigate this issue, read-log-update (RLU) [42] extends the RCU framework to ease concurrent programming. RLU allows read-only traversals, while supporting multiple updates by internally maintaining copies of multiple objects in a per-thread log, which is similar to the reader-writer programming model. RLU improves programmability in two ways. Firstly, it allows atomic multi-pointer updates which simplifies the design of many concurrent data structures like doubly linked list and tree. Secondly, because every reader gets a consistent snapshot of the data-structure protected by RLU, it removes the need of data structure specific pre-commit validation step required by most lock-free data structures. Despite its ease of programming, RLU shows limited scalability. For example, Figure 1 shows the scalability of the hash table benchmark with 10% updates, in which RLU's scalability starts degrading after 28 cores, and at 448 cores, it is more than 20× slower than RCU, which uses a spinlock to coordinate multiple writers. The reason for such performance is that RLU maintains only two versions of an object, which is a *dual-version concurrency control (DVCC) scheme or a restricted form of multi-version concurrency control (MVCC)*. When a writer tries to modify an object that already has two versions, it has to ***synchronously*** wait for all prior threads to leave the "RLU critical section" to reclaim one version (see Figure 2). This takes up to 99.6% of CPU time at 448 cores in Figure 1. In other words, DVCC significantly hinders concurrency.



(a) RLU

(b) MV- RLU

NOTE. [ ]: start and end of a critical section

Figure 2. Because RLU maintains only up to two versions of an object, thread T2's try_lock(a) call, which actually tries to create a third version of a, is blocked until a quiescent state is detected and the old a is reclaimed (marked in red). However, MV-RLU immediately creates a third version so T2 can proceed without blocking.

In this paper, we propose *Multi-Version Read-Log-Update* (MV-RLU), an extension of the RLU framework that supports multiple versions for better scalability in terms of throughput. Upon write-write conflict, unlike RLU, MV-RLU *avoids synchronous waiting* for object reclamation by utilizing other existing versions of a given object. MV-RLU uses timestamp ordering to see a consistent snapshot of objects; a thread chooses the correct version of an object using commit timestamps of a version (when the version was committed) and its local timestamp (when a thread starts a critical section). This is a well-known approach in multi-version concurrency control (MVCC) database systems [13, 34, 38, 52]. However, it is still challenging to design scalable synchronization frameworks based on MVCC such that they include the following: 1) a scalable global timestamp allocation scheme; 2) reduced memory stalls in version management, and 3) a scalable garbage collection (GC) mechanism. In addition, while providing multi-versioning, MV-RLU preserves the programming model of RLU, which allows atomic multi-pointer update. Hence, this results in easier concurrent programming model in comparison to RCU.

In this work, we make the following contributions:

- **Concurrent programming framework:** We present MV-RLU, an extension of RLU with multi-versioning, which enables designing multi-version-enabled concurrent data structures easily.
- **Avoid RLU overhead:** We decouple the synchronous garbage collection (or log reclamation) from the grace period detection, as well as adopt scalable timestamp allocation [33]—two big issues in RLU.

Approach	**Algorithm**	**Parallelism**			**Linearizable**	**Programming**	**Amplification** †		**Main Performance Overhead**
		RR	**RW**	**WW**		**Difficulty**	**Read**	**Write**	
Lock	mutex	×	×	×	✓	medium	1	1	lack of parallelism
	rwlock	●	×	×	✓	medium	1	1	limited parallelism
Lock-free	Harris list [25]	●	●	△	✓	high	1	1	cacheline bouncing; memory reclamation
Delegation-style	ffwd [55]	×	×	×	✓	low	1	1	single-threaded execution of a critical section
	NR [6]	●	×	×	✓	low	1	`# NUMA`	limited parallelism
STM	SwissTM [18]	●	▲	△	✓	lowest	2	2	centralized metadata management (e.g., lock table)
	STO [31]	●	▲	△	✓	low	2	2	high amplification ratio
RCU-style	RCU [43]	●	●	×	-	high	1	1	single writer
	RLU [42]	●	●	△	-	medium	1	2	synchronous log writing (`rlu_synchronize`)
	MV-RLU	●	●	△	-	medium	1	`1+1/V`	version chain traversal

NOTE. ×: no parallelism ●: full parallelism ▲: read-write conflict for the same data △: write-write conflict for the same data `V`: number of versions at GC

Table 1. High-level comparison of synchronization mechanisms. Each mechanism has a unique design goal, strategy to scale, and target class of workloads. For example, lock-free maximizes parallelism for performance while delegation/combining utilizes single thread execution to minimize synchronization costs; the primary goal of transactional memory is ease-of-programming; RCU and RLU are designed for read-mostly workloads. In MV-RLU, we extend RLU for write-heavy workloads using multi-versioning, while maintaining the optimal performance of RLU for read-mostly workloads along with its intuitive programming model, which is similar to readers-writer locking. † Amplification is defined by the ratio of the actual reads (or writes) from the memory to the reads (or writes) requested from an application. The amplification of STM approaches is 2 because reads and writes should be buffered and logged for atomic transactions.

- **Scalable garbage collection:** We propose concurrent garbage collection to achieve high performance under write-intensive workloads—one of the primary limitations of MVCC-based designs.
- **Workload-agnostic garbage collection:** We also propose an autonomous garbage collection algorithm that does not require workload-specific manual tuning, by maintaining a balance between chain traversal and garb-age collection deferring.
- **Application:** We extensively evaluate and analyze MV-RLU with concurrent data structures (list, hash table, and tree). We further apply MV-RLU to a key-value store [36] and a database benchmark [65] to compare concurrency control schemes. Nearly all cases with various read-write ratios, skewness, and data set sizes, MV-RLU shows the best (or second best) performance among other state-of-the-art techniques and scales well up to 448 cores.
- We will open source MV-RLU including the core framework and relevant benchmarks for the community to reproduce and build upon our results.

2 Overview of MV-RLU

2.1 Programming Model

MV-RLU follows the programming model of RLU, which resembles readers-writer locking. Each critical section begins by `read_lock` and ends by `read_unlock`. Before modifying an object, a thread locks an object using `try_lock`. Unlike readers-writer lock, there is no `lock` or `unlock`. Thus, on `try_lock` failure, a thread should `abort` and re-enter the critical section by calling `read_lock`. MV-RLU automates this process by managing the per-object lock with additional metadata, while guaranteeing that a thread will observe a consistent snapshot of objects. Moreover, object metadata is hidden from users, as MV-RLU provides its own API, such as `alloc`, `free`, `dereference`, `assign_ptr`, and `cmp_ptr`.

The benefit of MV-RLU programming model is that it can act as a drop-in replacement for RCU and RLU. Moreover, the readers-writer lock style model is familiar to ordinary users and many fine-grained locking techniques are applicable to MV-RLU. For example, in Figure 3, thread `T3` uses a well-known hand-over-hand locking [30] to insert node `e` after `d`. It first locks node `c` and `d` to prevent a race condition between removing node `d` and inserting node `e`.[2] In addition, we extend the API with `try_lock_const`, which allows a thread to lock an object that it does not intend to modify (node `c`), as an optimization.

2.2 Multi-Versioning

While MV-RLU uses a lock-based programming model, it also relies on multi-versioning (MVCC) with timestamp ordering. We chose to utilize MVCC because it further enables disjoint-data access parallelism, as it is only restricted when two threads try to modify the same data (i.e., write-write conflict). Hence, MVCC can further improve scalability.

In MV-RLU, an object consists of one *master object* and zero or more *copy objects*, as shown in Figure 3. A master object,

[2] Note that hand-over-hand locking is required only in update operations (e.g., `insert`, `remove`). For read operations (e.g., list traversal), hand-over-hand locking or data-structure specific validation steps, which are required in fine-grained locking, are not necessary because MV-RLU provides a consistent snapshot of the data structure to readers.

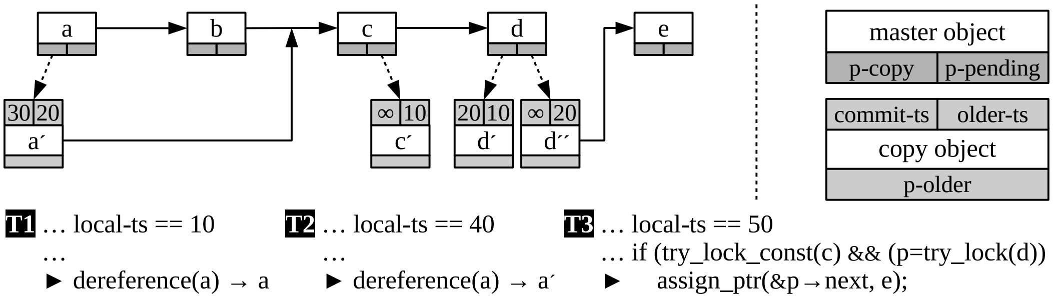


Figure 3. Illustrative snapshot of concurrent operations in the `MV-RLU`-based linked list. ▶ denotes where a thread executes. At time 30, node `b` is removed, then the `next` of node `a'`, whose `commit-ts` is 30, points to node `c`. A thread gets a local timestamp (`local-ts`) at the beginning of a `MV-RLU` critical section (`read_lock`) and uses it to choose the newest object committed before entering the critical section (`commit-ts`). For instance, thread `T1`'s dereferencing node `a` returns `a`, which is the oldest among node `a`'s versions. However, `T2`'s dereferencing node `a` returns `a'`. Thus, `T1` traverses node `b`, while `T2` does not, as `T1` and `T2` start before and after the removal of node `b`, respectively. To modify data in `MV-RLU`, a thread first tries to acquire the given object's lock. `MV-RLU` copies its newest object and returns its pointer upon success. For example, `T3` wants to insert a new node `e`. It first locks nodes `c` and `d` with the hand-over-hand locking technique [30]. While it sets the `next` of `d"` to `e`, the new node `e` is not yet visible, until `T3` commits its write set (a set of modified objects in a critical section) at the end of its critical section. Upon commit, `T3` moves its pending object `d"` (`p-pending`) to the head of version chain (`p-copy`) and sets the commit timestamp of `d"` (`commit-ts`) to allow node `e` to be visible. Finally, when no threads access node `b` (i.e., in quiescent states), `MV-RLU` reclaims it.

which is allocated using `alloc`, whereas copy objects are created when a thread successfully locks an object (`try_lock`). A copy object is a *pending version* (`p-pending`) and is moved to the *version chain* (`p-copy`) only when the thread commits (`read_unlock`). In `MV-RLU`, all threads are guaranteed to have a consistent view of the objects; `MV-RLU` achieves this consistency via timestamp ordering. A thread entering the system is assigned a timestamp (`local-ts`) and it chooses the correct version of the object by comparing its `local-ts` with the `commit-ts` of an object. For example, thread `T1` traverses node `b`, while `T2` does not observe it as `T1` and `T2` start before and after the removal of node `b`, respectively. Moreover, each thread maintains a *per-thread circular log* to store copy objects. It maintains a *write set* of all copy objects in the same critical section that will be atomically committed at the end. Upon `abort`, it rewinds the log tail to discard all copy objects in the write set and unlock all objects in the write set.

2.3 Garbage Collection

`MV-RLU` creates a new version, whenever a thread modifies an object. It also safely reclaims an object, only when the object becomes invisible to threads entering critical section and has no existing references. For example, in Figure 3, it is not safe to reclaim node `b` until `T1` exits its critical section. `MV-RLU` uses a grace period detection technique like `RLU` to find safely reclaimable versions. Unlike `RLU`, `MV-RLU` decouples synchronous grace period detection from garbage collection by moving the the synchronous detection off the critical path. To this end, a background thread, called `gp-detector`, detects grace period periodically or on-demand. We propose a *concurrent autonomous garbage collection* scheme, in which each thread reclaims its own log space. This design prevents garbage collection from becoming a scalability bottleneck. In addition, `MV-RLU` automatically decides when to trigger garbage collection without workload-specific manual tuning by considering write back and version traversal overhead.

2.4 Consistency Guarantee

`MV-RLU` satisfies snapshot isolation, in which a thread observes a consistent snapshot of objects [5]. Snapshot isolation has been widely adopted by major database systems because it allows better performance than stricter consistency levels [62]. However, it is not serializable as it permits write skew anomaly [5]. Write skew anomaly occurs if two threads concurrently read overlapping objects, concurrently make disjoint writes, and finally concurrently commit, in which both of them do not see writes performed by the other. Modern MVCC database systems [13, 38, 54] have an additional validation phase upon commit to detect write skew anomalies and guarantee serializability by aborting such transactions at the expense of maintaining and validating read sets. Under `MV-RLU`, a programmer can make an `MV-RLU` critical section serializable by locking read-only objects (`try_lock_const`) to generate a write-write conflict. One example is inserting and removing an element to/from a singly-linked list. As shown in Figure 3, it is possible to serialize `insert` and `remove` operations, by using hand-over-hand locking, i.e., lock node `c` which is read-only, to prevent node `d` from getting deleted in the middle of operation.

3 Design of `MV-RLU`

3.1 Design Goals

We design `MV-RLU` with three main goals: First, `MV-RLU` should be scalable to a wider range of workloads (e.g., read-mostly and write-intensive workloads), while maintaining programmability it inherits from RLU. Prior works are optimized for a narrow scope of workloads for example, RLU and RCU are designed for read-mostly workloads while delegation and combining is applicable to synchronization dominant workloads. These restrictions severely limit their use. Second, MV-RLU should scale with increasing core count (> 100). This is critical because servers and virtual machines with more than 100 cores are now a norm [11, 57, 63]. Finally, the performance of MV-RLU should be optimal even for smaller core counts because previous studies [46] have shown that many systems improve scalability at the cost of lower core performance. These design goals have complex interactions, thereby leading to no optimal system satisfying all these goals, which we achieve with MV-RLU.

3.2 Version Representation

The alloc function creates an object (called *master object*), along with its header. The header has two pointers: p-copy is the head of a version chain and p-pending points to the uncommitted version of the master object, if any. A version chain is a singly linked list, which links different versions of the same master object (called *copy object*) via p-older. Readers will most likely read the most recent version of an object. Therefore, to make the common case for reads faster, version chain is ordered from *newest to oldest* version to prevent expensive version chain traversal. In the commit phase, a writer thread moves the uncommitted version (p-pending) to the head of the version chain (p-copy) and hence preserves the newest to oldest invariant of the version chain.

The per-thread circular log stores copy objects. Copy objects commited in the same critical section are grouped into a *write set*. Each write set has a *write set header* that holds its commit timestamp (commit-ts). Copy objects also have an object header, which stores commit timestamp (commit-ts) of the copy object. It is the same as the commit-ts of its write set and is duplicated to reduce memory accesses during version traversal (dereference). Initially, the commit-ts of a copy header and a write set header are ∞. If the commit-ts of a copy header is ∞, then we use the commit-ts of its write set. This is necessary because all copy objects should be visible at the same time during the commit phase (see §3.5). The header also stores the commit timestamp of the last committed version of the same master object (older-ts), which we use for version chain traversal.

3.3 Reading an Object

Reading an object (dereference) is finding a correct version by traversing the version chain. On entry into a critical section (read_lock), a thread first sets its local timestamp (local-ts) to the current hardware clock. Then, it traverses the version chain and finds the first copy object, whose commit timestamp (commit-ts) is smaller than the thread's local timestamp. If the version chain is empty or there are no copy object with commit-ts less than or equal to the thread's local-ts, then the thread reads the master object.

3.4 Writing an Object

To modify an object, a writer tries locking the master object with try_lock. On failure (i.e., p-pending ≠ NULL), the writer aborts and then retries. To maintain the newest-to-oldest ordering, MV-RLU applies a *write-latest-version-only* rule. It aborts if the writer's local-ts is less than the commit-ts of the latest copy object at the head of the version chain. If the aforementioned conditions are met, the writer creates a header for the copy object in its log and tries to atomically install it to p-pending using a compare-and-swap (CAS) operation. On success, writer appends the new version to the it's current write set and returns new version's pointer to the caller.

3.5 Commit a Write Set

read_unlock denotes the exit of the MV-RLU critical section, which commits the thread's write set if it exists. To make the entire write set atomically visible, we first move the pending objects (p-pending) to the head of a version chain (p-copy). We then update the commit timestamp (commit-ts) of the write set to the current hardware clock. This is the linearization point of the commit operation, as all new copy objects have a commit-ts and are now visible to new readers (see §3.2). Finally, we update the commit-ts in header of copy-objects and set the p-pending of the corresponding master-object to NULL, which unlocks objects.

3.6 Abort a Critical Section

On failure of a try_lock call (§3.4), we abort that corresponding thread, which unlocks the master object in its uncommitted write set by updating the p-pending to NULL. Finally, we free the log space by rolling back the tail pointer of the log to the beginning of the write set of the thread which was aborted.

3.7 Garbage Collection

Garbage collection is critical because its performance bounds the write performance in MV-RLU. There are three key challenges to garbage collection: 1) finding safe reclaimable versions, 2) avoiding garbage collection being a bottleneck, 3) deciding when to trigger garbage collection

Finding safe reclaimable objects. We use a RCU-style grace period detection algorithm to decide whether an object is safe to reclaim or not. We delegate the grace period detection to a special thread: gp-detector, which detects each grace period expiration. With this approach, we decouple quiescent state detection and thread operation. Decoupling is important because, in RLU, even for read-intensive workload (e.g., 10% write in Figure 1) quiescent state detection becomes a scalability bottleneck, as a thread running rlu_synchronize has to wait for other threads to finish their critical sections.

An object is obsolete if it has a newer version. When all threads reading an obsolete object exit the critical section, then it becomes invisible and is safe to reclaim. We use grace periods to determine if threads are reading an obsolete object. *A copy object is safe to reclaim if one grace period has elapsed since it became obsolete.* A noteworthy point is that we cannot reclaim copy objects, which are the latest versions of the master object, because they are visible to all threads. Unfortunately, this increases the dereference cost, as readers will have to access the version chain to get the correct version of the object. To reduce this cost, we write back the latest copy object to the master object and then reclaim the copy object. Thus, we *safely reclaim the latest copy object, by first writing back to the master object after at least one grace period since the creation of the copy object, and later reclaiming it after another grace period* (see §4.2 for correctness).

Concurrent garbage collection. To avoid one thread from becoming a performance bottleneck, we devise a concurrent garbage collection algorithm. Every thread checks whether it should garbage collect its log at the MV-RLU critical section boundary (read_lock, read_unlock, abort). If a thread requires garbage collection, it indicates gp-detector to broadcast the begin timestamp of the latest detected grace period (graceperiod-ts) to each thread. When threads receive the timestamp, they perform garbage collection of their own logs, by removing all copy objects that have commit-ts less than the second-to-last broadcasted graceperiod-ts (i.e., two elapsed grace periods) and write back copy objects to their master if they are the latest version and their commit-ts is less than the last broadcasted graceperiod-ts (i.e., one grace period has elapsed). This scheme has optimal performance because grace period detection becomes asynchronous and garbage collection is concurrent. The communication between gp-detector and MV-RLU threads is merely accessing shared memory. Moreover, each thread reclaiming its own log not only ensures cache locality, but also is hardware prefetcher friendly [32], thereby avoiding cache thrashing and cross-NUMA memory accesses. To prevent two or more threads writing copy objects back to the master at the same time, we add a *reclamation barrier*, which prevents triggering of the new garbage collection routine before the last garbage collection is completed. To ensure liveness of the system, we have to guarantee the termination of garbage collection routine. Liveness can be an issue if a thread takes longer time across MV-RLU boundary, causing entire garbage collection to wait at the reclamation barrier, which we prevent by allowing the gp-detector thread to reclaim the log of a thread if it did not initiate garbage collection.

Autonomous garbage collection. The optimal time to trigger garbage collection depends on workload characteristics, such as read-write ratio and write skew. There are two conflicting goals for garbage collection: On one hand, it is better to defer garbage collection as much as possible until there is almost no space left in a per-thread log because we can save write-back costs from the newest copy to a master object (i.e., larger V in Table 1 is better). On the other hand, it is better to reclaim logs as early as possible because we can reduce version traversal overhead by writing back the newest copy object and pruning the version chain. To deal with these conflicting goals, our autonomous garbage collection scheme decides when to trigger garbage collection considering both log space utilization and version chain traversal overhead.

Our design has two conditional triggers for garbage collection we call watermarks. The first is a *capacity watermark*, which triggers garbage collection when the log space is insufficient; and a *dereference watermark*, which triggers garbage collection when the access ratio of copy objects to master objects is high. Since we do not want threads to block because of the lack of log space (called *high watermark*), we define a *low watermark* for the log space with a goal to trigger garbage collection early enough to avoid thread blocking at high watermark. This design is autonomous as we do not need manual tuning for different workloads as they change the frequency of garbage collection automatically, based on the workload behavior. For example, in the case of write-heavy and skewed workload, the capacity water mark triggers garbage collection, whereas in read-mostly and uniform workloads, the dereference watermark triggers garbage collection. Moreover, the calculation of these watermarks do not require any synchronization among threads, thereby leading to almost negligible overhead.

3.8 Freeing an Object

To free a master object, first the object must be locked using try_lock method which prevents any concurrent update to the object. Then, the object can be freed using the free method. Internally, since it might not be safe to deallocate the object immediately, free puts the object in a free list. Also, to prevent any further update of freed object, objects in the free list are not unlocked after commit (read_unlock). After two grace periods since adding the object to free list, the object memory is deallocated.

3.9 Timestamp Allocation

Prior works [33, 34, 38, 59, 65] have shown that timestamp allocation is a major bottleneck in MVCC-based designs. To alleviate this issue, we use a hardware clock (RDTSCP in x86 architecture) for timestamp allocation. However, hardware clocks can have a constant skew between them which can lead to incorrect ordering. We avoid this inconsistency by using the ORDO primitive [33], which provides the notion of a globally synchronized clock by calculating the maximum uncertainty window (called ORDO_BOUNDARY) among

CPU clocks and uses it to compare timestamps. Only timestamps with difference more than ORDO_BOUNDARY can be ordered non-ambiguously. To remove the ambiguity in ordering, we add ORDO_BOUNDARY to the timestamp when allocating commit-ts to a copy object (i.e., new_time API in ORDO) and subtract ORDO_BOUNDARY from the timestamp when allocating for garbage collection. Also, try_lock fails when the difference between local-ts of writer and the last commit-ts of an object to be locked is less than the ORDO_BOUNDARY. See §4.3 for correctness.

4 Correctness of MV-RLU

4.1 Definitions

- **Grace period.** An interval in which every thread in the system has been outside the critical section.
- **Latest copy object.** The newest version of a master object. It is at the head of a version chain.
- **Obsolete object.** An object that has become invisible to new readers because of the presence of a newer version.

4.2 Garbage Collection

Lemma 1. *An obsolete copy-object is safe for reclamation if one grace period has elapsed since it became obsolete.*

Proof. We cannot immediately reclaim an obsolete copy-object, as other threads may be reading it. However, on detecting a grace period, we can reclaim the obsolete copy because there is no old references to that copy. □

Lemma 2. *It is safe to write back a copy-object to its master object if one grace period has elapsed since its creation.*

Proof. After creating a copy-object, its master object becomes obsolete. From Lemma 1, the master object has now no references after the lapse of one grace period. Hence, it is safe to write back the copy-object to the master object. □

Lemma 3. *It is safe to reclaim the latest copy-object after two grace periods since its creation, if it is written back to the master object after the first grace period.*

Proof. From Lemma 2, it is safe to write back the latest copy after the first grace period, which makes the latest copy obsolete. From Lemma 1, it is safe to reclaim the latest copy after another grace period. Since after grace period detection, a latest copy object can turn into an obsolete copy object, we reclaim all objects after two grace periods. □

Theorem 1. *MV-RLU garbage collection removes objects that are invisible to readers.*

Proof. MV-RLU garbage collector removes object only after two grace periods and writes back latest copy object after the first grace period. From Lemma 1, 2, and 3, we claim that the garbage collection of MV-RLU removes objects that are invisible to readers. □

4.3 Timestamp Allocation

Theorem 2. *Clock skew among physical clocks do not affect the snapshot of a reader.*

Proof. ORDO_BOUNDARY is greater than or equal to the maximum clock difference between clocks with the smallest and the largest skew in the system. When assigning commit-ts, we add ORDO_BOUNDARY to the commit-ts (i.e., new_time in ORDO) to prevent threads with smaller clock-skew, reading objects committed by threads with larger clock-skew after a critical section has begun. We reduce the grace period timestamp by ORDO_BOUNDARY, to prevent reclamation of objects that are still visible to threads with smaller clock-skew. □

4.4 Isolation Guarantee

MV-RLU has the following invariants:

Invariant 1. *dereference ensures that a reader cannot see objects newer than its local-ts.*

Theorem 3. *MV-RLU provides snapshot isolation.*

Proof. MV-RLU provides snapshot isolation if it always provides a consistent snapshot of data structures to every reader. Threads in MV-RLU modify objects (both copy and master) during writes and garbage collection. Hence, we show that the snapshot remains unaffected by these operations.

- Two threads cannot write to same object at the same time, as only one thread holds the lock to the object.
- Uncommitted copy objects have commit-ts of ∞. From Invariant 1, these objects are not visible to reader. Hence, writes do not affect the snapshot of readers.
- Both write back and garbage collection do not affect the snapshot of reader. [1]
- Clock skew among CPU clocks does not affect the snapshot of readers. [2]
- A Reclamation barrier ensures that a garbage collection routine cannot start until the last routine has ended, which prevents write-backs by two threads to the same master.

Thus, MV-RLU provides snapshot isolation. □

5 Implementation

We implemented MV-RLU in C comprising of 2,250 lines of code.[3] We made several implementation-level optimizations. For instance, we implemented per-thread logging using a circular array, which enables memory access during log reclamation to be sequential and hardware prefetcher friendly. In addition, we avoid false cache-line sharing by aligning copy objects to cache-line size. Our current implementation statically allocates the log and is prone to blocking if the log frequently crosses the high watermark. Fortunately, we did

[3] We wrote MV-RLU from scratch without reusing RLU code because it required significant code changes to add multi-versioning to RLU and implement our optimizations.

not observe such scenarios in our evaluation and hence we did not implement dynamic resizing of each threads log.

In MV-RLU, the most performance critical code is dereference and its first step of distinguishing whether a given address points to a master object or a copy-object. We require this because the version chain traversal starts from the master object. To distinguish an object type without accessing its header, we maintain copy objects and master objects in different address spaces.[4] By doing so, a thread can distinguish object's type without reading its header, thereby reducing memory access. Since accessing object headers for type information is a common case, there is a noticeable improvement in performance (see §6.3).

6 Evaluation

We evaluate MV-RLU by answering the following questions:

- Does MV-RLU achieve high scalability across several data structures under varying contention levels with varying intensity and data set size? (§6.2)
- What is the impact of our proposed approaches on MV-RLU? (§6.3)
- What is its impact on real-world workloads? (§6.4)

6.1 Experimental Setup

Evaluation platform. We use a 448-core, 337 GB, Intel Xeon Platinum 8180 CPU (2.5GHz) for our evaluation. It comprises of eight NUMA sockets with 28 cores (hyperthreaded 56 cores) per socket. We use jemalloc for scalable memory allocation on Linux 4.17.3.

Configuration. We configure per-thread log size to 512 KB, high and low capacity watermarks, and dereference watermark to 75%, 50%, and 50% of the log capacity, respectively. We compare MV-RLU with several state-of-the-art approaches: RCU [43], RLU [42], RLU-ORDO (i.e., RLU using ORDO timestamp), Versioned-programming [67], SwissTM [18], Harris-Michael linked list [25], and predicate-RCU (PRCU) for Citrus tree [1, 7]. For RLU, we only present results for its non-deferring version because we did not observe any noticeable differences in results with or without RLU deferring. For all other algorithms, we use default parameters.

Workloads. We evaluate three data structures and two real-world applications. The data structures include a linked list, hash table, and binary search tree. For real-world applications, we use DBx1000 [65] and KyotoCabinet [36]. For all workloads, we ran three types of workloads: 1) read-mostly (98% read and 2% update), 2) read-intensive (80% read and 20% update), and 3) write-intensive (20% read and 80% update).

6.2 Concurrent Data Structures

We first compare the scalability of three concurrent data structures (§6.2.1). Then we analyze MV-RLU behavior with different data set sizes (§6.2.2) and vary contention levels with different skewness of access (§6.2.3).

6.2.1 Performance and Scalability

Linked list. We compare MV-RLU with RCU, RLU, RLU-ORDO, Versioned Programming, and SwissTM with 10,000 (10K) items. The first row of Figure 4 shows that MV-RLU outperforms all synchronization mechanisms because of multiversioning coupled with efficient garbage collection. RCU does not scale with increasing update ratio because rlu_synchronize becomes a bottleneck. In the case of a read mostly workload, RCU is scalable only up to 56 threads (on 2 NUMA nodes), then its scalability gradually degrades as more threads start to contend on the write operation, which is also evident in read- and write-intensive workloads. RLU shows better scalability than RCU because it allows concurrent write operations except for the case when two writers want to modify the same object. However, in the case of read- and write-intensive workloads with higher update ratios, its performance is saturated after 56 threads because of the frequent synchronous log reclamation, as evident in Figure 2. RLU-ORDO generally performs better than RLU because it avoids the global clock being a bottleneck.

Versioned-programming supports multiple versions like MV-RLU. It's scalability trend is similar to MV-RLU, but it is up to 24% slower than MV-RLU in all cases. Unlike MV-RLU, all committed, uncommitted, and aborted versions are part of the version chain, which results in expensive version chain traversal. This overhead hampers scalability, as a thread spends 79% of CPU time to obtain the correct version of an object at 224 threads for the read-intensive workload. In the case of SwissTM, we show results only up to 140 threads because it crashes after 140 threads [16]. For the read-mostly workload, SwissTM scales up to 28 threads and then its performance is saturated. As update ratio increases, SwissTM shows significantly worse performance because of frequent read-write and write-write conflicts. We will further analyze its abort ratio under high contention in §6.2.3.

Hash table. We create a hash table with 1,000 buckets, with each bucket pointing to a singly linked list. The hash table is initialized with 10,000 items. For each operation, we first find a bucket corresponding to the key using a hash function then access the desired key from the linked list in the bucket. The second row of Figure 4 shows the results of hash table performance for three types of workloads. For all three workloads, MV-RLU shows near linear scalability up to 448 threads and it is 652× faster than others for read- and write-intensive workloads. In particular, MV-RLU performs best in read- and write-intensive workloads. On the other

[4] In user-space, we allocate log space from a mmap-ed region. In kernel space, we use kmalloc for a master object and vmalloc for log space, so copy objects are located between VMALLOC_START and VMALLOC_END.

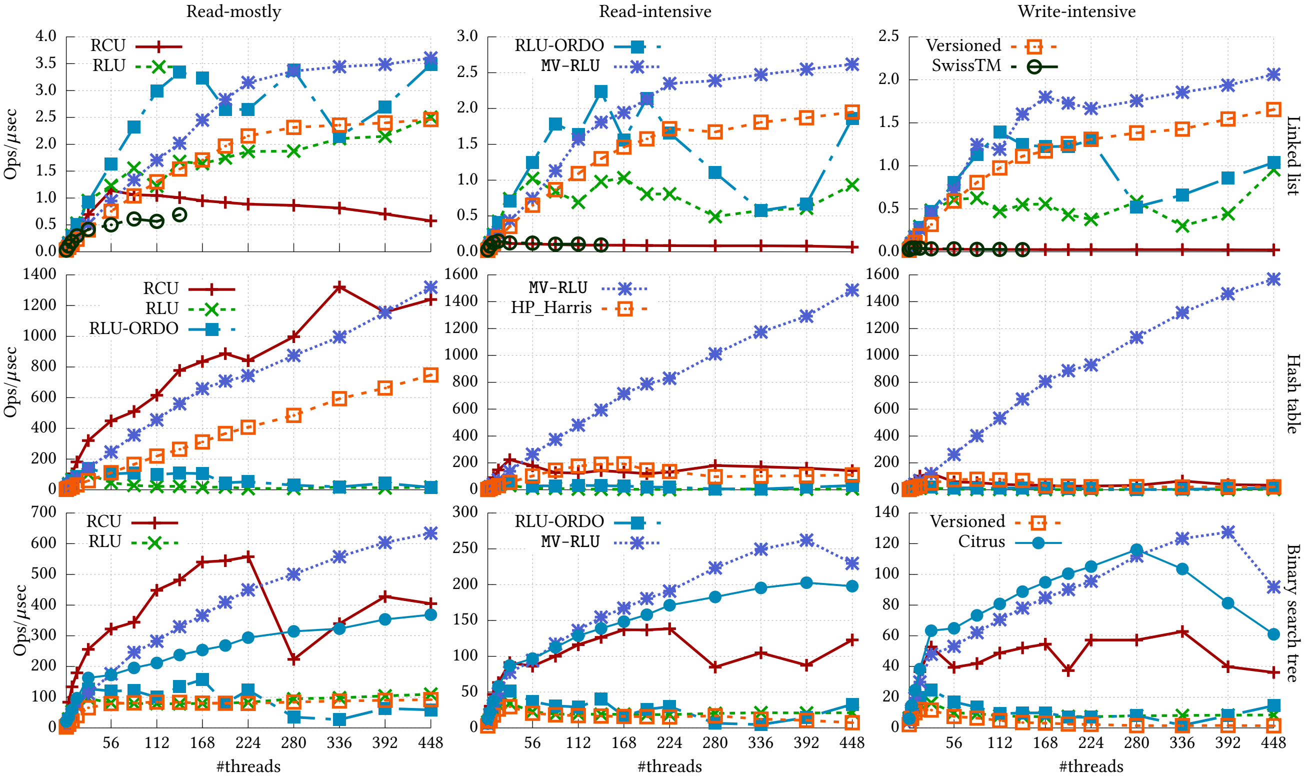


Figure 4. Throughput for a 10,000 item linked list, hash table (1K buckets), and binary search tree with read-mostly, read-intensive, and write-intensive workloads (first column: read-mostly, second column: read-intensive, third column: write-intensive), (first row: linked list, second row: hash table, third row: binary search tree)

hand, for read-mostly workload, RCU performs best because RCU hash table uses a per-bucket lock for writes to allow for more parallelism than linked list. However, as update ratio increases, the fine-grained locking for concurrent writers becomes a bottleneck, which we observe after 392 cores in the read-mostly workload for RCU. Moreover, at 448 cores, 71% of CPU time is spent in lock acquisition for the write-mostly scenario. RLU performs the worst, as after every write it calls the rlu_synchronize function which blocks the caller thread. Moreover, the writeback defer optimization and RLU-ORDO do not improves its scalability because the chances of writer-writer conflicts and rlu_synchronize cost increases with increasing thread count.

HP-Harris is a hash table where each bucket points to a lock free linked list. The lock free linked list uses hazard pointers [47] for safe memory reclamation. It performs well for read-mostly workloads but performs poorly for higher write ratios. To understand the low throughput, we performed Perf [41] analysis of HP-Harris hash table at 448 threads for write-intensive workload. The analysis indicates that memory barrier in object derefernce which is required by hazard pointers is the performance bottleneck.

Binary Search Tree. We implement a binary search tree (BST) implementation using MV-RLU, which is similar to the one using RLU, which we compare the RLU, RLU-ORDO, RCU, Versioned-Programming, and Citrus trees with predicate RCU. Citrus tree with predicate RCU is an optimized version of Citrus tree, which reduces the number of threads waiting during an rcu_synchronize call using data structure specific predicates [1, 7]. Even in this case, MV-RLU outperforms others for all varying workloads. For the read-mostly workload, RCU shows the best performance up to 224 threads, however the performance drops sharply when the number of physical cores (224) exceeds, due high overhead of rcu_synchronize function call which is required to safely delete a node. RLU shows scalability only up to 28 threads, but later suffers from the rlu_synchronize bottleneck. RLU-ORDO shows slightly higher performance than RLU but still shows similar performance trends. Versioned-programming only scales up to 28 threads. Beyond that, allocating a logical timestamp becomes the performance bottleneck, which is different from the one we observed in the linked-list case because the critical section of the tree data-structure is smaller. Since the allocation of epochs is closely coupled with writer-writer conflict detection, physical clocks cannot be used as logical timestamps in Versioned programming without significantly modifying the algorithm.

Analysis on abort ratio. To understand why MV-RLU performs better than RLU and SwissTM, where a transaction can abort, we further analyzed their abort ratios for the linked

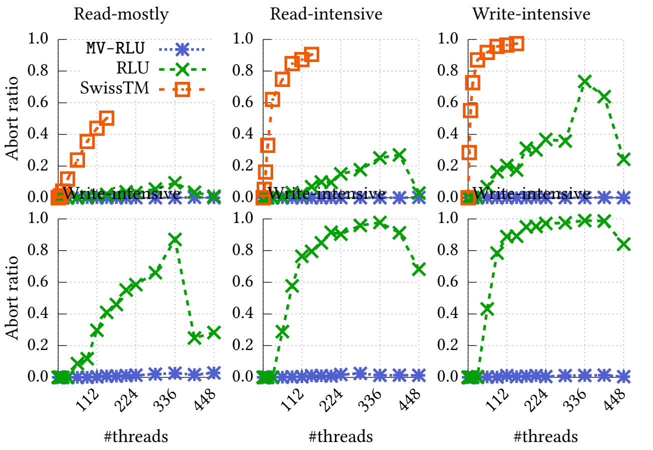


Figure 5. Abort ratio of linked list (top) and hash table (bottom) with 10,000 items. Load factor of hash table is 10. Workloads are read-mostly, read-intensive, and write-intensive, respectively, from left to right.

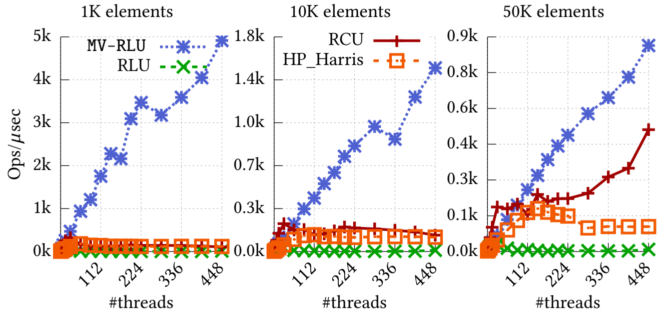


Figure 6. Hash table performance with 1K, 10K, and 50K items for the read-intensive workload. Load factors are 1, 10, 10, respectively, from left to right.

list and hash table in Figure 5. The abort ratio of MV-RLU is very low (0-2.3%) because it allows to have more than two versions unlike RLU and it does not abort upon read-write conflict unlike SwissTM. The abort ratio of RLU increases significantly as the write ratio and the number of threads increase. It shows the limitation of synchronous reclamation strictly maintaining only two versions. SwissTM shows even higher abort ration than RLU because it also aborts upon read-write conflict to guarantee linearizability.

6.2.2 Data Set Size

To understand the behavior of MV-RLU with various data set size, we compare the performance of a hash table with 1K, 10K, 50K items. Load factor of each hash table is configured to 1, 10, 10, respectively, which means that as the hash table size increases, the chance of write-write conflict decreases but the length of the critical section becomes longer. Figure 6 shows performance results for the read-intensive workloads with uniform random access. MV-RLU shows excellent scalability in all cases. On the other hand, RCU nearly scales linearly until 28 threads for 1K and 10K, respectively. For 50K items, RCU scales beyond 224 threads, because of the decrease in contention on the bucket lock. RLU and HP-Harris schemes scale poorly even for large size of the hash table. In particular, while RLU has lower chance of write-write conflict that incurs rlu_synchronize, the longer critical section in larger hash tables increases the length of the rlu_synchronize operation.

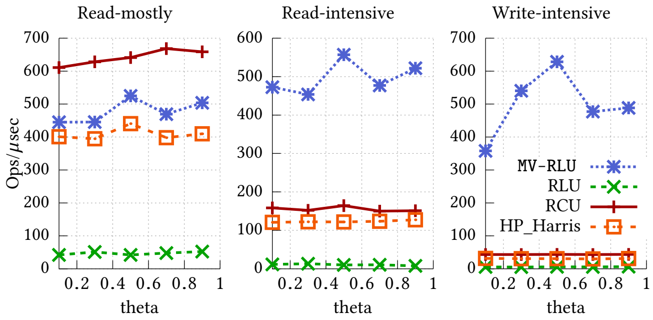


Figure 7. Performance of hash table with 10K nodes for read-mostly, read-intensive, and write-intensive workloads (left to right) at 336 threads. X-axis is theta value of Zipfian distribution. Higher theta value means that data access is more skewed.

6.2.3 Contention

We evaluate the scalability of MV-RLU for the hash table benchmark with skewed access by relying on the Zipf distribution generator [2]. We use a hash table with 10K items with a load factor 10 for read-mostly, read-intensive, and write-intensive workloads. We run these workloads with 336 threads and vary the Zipf theta value. Figure 7 clearly shows the benefit of multi-versioning in MV-RLU. The performance of MV-RLU is nearly constant regardless of skewness and write intensity. We also comfirm that performance trends are same with other core counts including 448 cores. Even in some cases the performance increases with higher skewness due to increased cache-locality. However, the performance of other approaches, RCU, RLU, and HP-Harris, are fundamentally bounded by the write intensity of the workloads.

6.3 Factor Analysis

To understand how our design choices affect performance, we incrementally add MV-RLU features to RLU and run benchmarks for read-mostly, read-intensive and write-intensive workloads.

+ ORDO. For high update workloads, frequent updates to global clocks create large cache coherence traffic that become the scalability bottleneck. ORDO, based on the hardware clock, improves the scalability by 1.6× times.

+ Multi-versioning. We added multi-versioning to RLU with a single garbage collection thread to reclaim invisible versions. This scheme showed 2.3× improvement in read-mostly case because multi-versioning decouples the grace

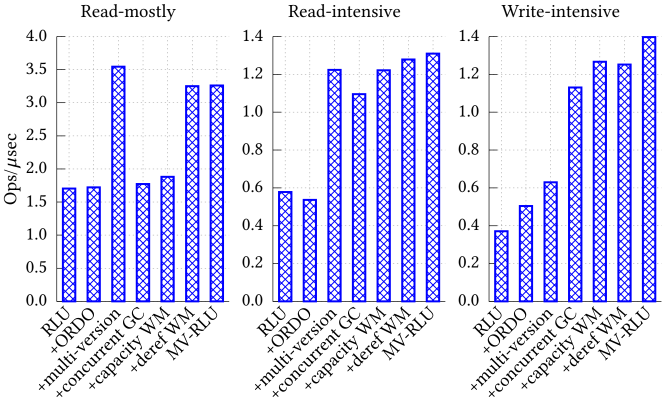


Figure 8. Contribution of MV-RLU features to performance. Design features are cumulative. The performance results of linked list with 10K items are presented at 336 threads for the read-mostly, read-intensive, and write-intensive workloads from left to right.

period detection from operations. However, it suffers in the case of write-intensive workload because a single garbage collection thread becomes the scalability bottleneck.

+ Concurrent GC. Concurrent self-log reclamation improves the scalability in write-intensive workload by 1.8×, as it avoids the garbage collection bottleneck.

+ Autonomous GC: capacity watermark. Starting garbage collection before log becomes full improves the performance in read-intensive and write-intensive case but not in read-mostly case, as few objects are created.

+ Autonomous GC: defererence watermark. Adding our dereference watermark improves the performance in read-intensive case by 1.8×, thereby making MV-RLU garbage collection autonomous, as it works well for various types of workloads.

MV-RLU. The final MV-RLU implementation shows 1.8×, 2× and 3.5× throughput improvement as compared to base-RLU, showing design choices in MV-RLU complement each other and scale efficiently for various types of workloads.

6.4 Applications

DBx1000. DBx1000 benchmark compares scalability of different concurrency control mechanisms in databases. We add MV-RLU as a multi-version concurrency control mechanism in DBx1000 to compare scalabilty of our design against other MVCC and OCC designs such as TicToc (OCC) [66], Silo (OCC) [59] and Hekaton (MVCC) [13]. We protect every transaction by `read_lock` and `read_unlock`. To add headers required for MVCC, we use MV-RLU `alloc` API to allocate records, and we update a record using `try_lock`, which creates a new version of that record, which are later committed atomically at the end of the `read_unlock`. We used the YCSB [10] workload to benchmark the throughput for 2%, 20% and 80% update rates and Zipf-theta value of 0.7. MV-RLU shows scalability similar to other OCC mechanisms and much better performance than Hekaton, which is bottlenecked by the global clock and garbage collection.

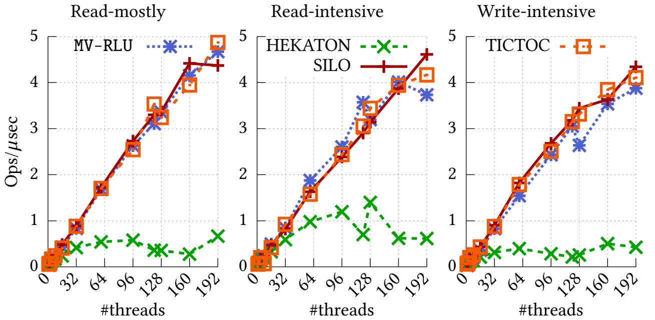


Figure 9. Performance comparison of MV-RLU, HEKATON, SILO, and TICTOC for YCSB benchmark for read-mostly, read-intensive, and write-intensive workloads with Zipf distribution of 0.7. MV-RLU outperforms HEKATON by 7–10×.

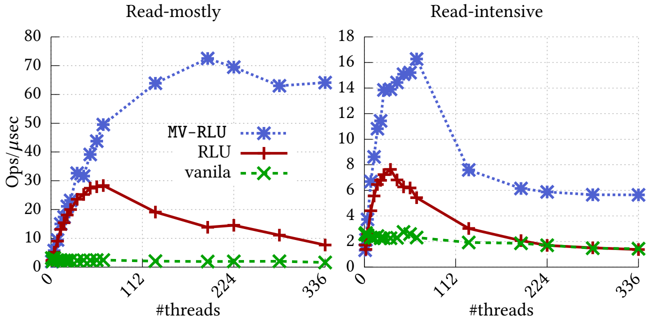


Figure 10. Performance comparison of MV-RLU, RLU, and the stock version in KyotoCabinet benchmark with read-mostly (left) and read-intensive (right) workload, in which MV-RLU outperforms RLU by 3.8–8.4×.

KyotoCabinet. KyotoCabinet Cache DB [36] is a popular in-memory key-value store written in C++. The KyotoCabinet is internally divided into slots. Each slot is further divided into buckets and each bucket points to a binary search tree. To synchronize database operation, KyotoCabinet uses a global readers-writer lock, which is a known scalability bottleneck [14]. In addition to the global readers-writer lock, there is a per-slot lock to synchronize accesses to each slot. We followed the implementation details described in RLU to eliminate the global reader-writer lock from KyotoCabinet using MV-RLU. Note that MV-RLU is a drop-in replacement of RLU. Access to the database is protected by `read_lock` and `read_unlock`. Writers are synchronized using per-slot lock to prevent aborts for a fair comparison with RLU. We initialized the database with 1 GB of data and then measured the throughput for 2% and 20% update ratio. Figure 10 shows the performance of original, RLU-based, and MV-RLU-based KyotoCabinet. Clearly, MV-RLU shows the best scalability with increasing threads. However, with a large thread count, per-slot locking becomes a performance bottleneck, which we used for fair comparison with RLU. We expect that MV-RLU can scale better if we adopt a design that does not use per-slot locking.

7 Discussion and Limitations

One of the limitations of MV-RLU is its weaker consistency guarantee: snapshot isolation, which might restrict its use in some applications that requires a stronger consistency guarantee, such as serializability or linearizability. However, the profound adoption of RCU in the Linux kernel, and several database systems, such as Oracle [51], have shown that linearizability is not a necessity. Moreover, if necessary, a developer can make MV-RLU serializable by locking nodes (other than the one being modified) using try_lock_const to prevent write skew. For example, we can serialize linked list data structures with MV-RLU by locking the predecessor of the node being modified. The aforementioned examples also corroborates our view that snapshot isolation is a practical choice for better performance than having stricter consistency levels. It will be an interesting future research direction to improve the consistency level of MV-RLU with additional validation and further scale its performance. Another limitation is that MV-RLU does not guarantee individual thread level progress, i.e., a thread having a longer critical section that needs to lock contending objects can starve. However, since MV-RLU aborts only upon write-write conflict, we expect that such starvation is less problematic than other approaches (e.g., STM) which aborts even in read-write conflicts.

8 Related Work

In-memory database system. Most in-memory database systems designed adopt either an Optimistic Concurrency Control (OCC) [35, 59, 61, 66] or Multi-Version Concurrency Control (MVCC) [13, 34, 38]. OCC is essentially a single-version concurrency control so it works well under low contention. However, under high contention the performance of OCC degrades significantly with high a abort ratio due to the lack of versions [34, 38, 65]. Recent in-memory database systems, such as Hekaton [13], ERMIA [34], and Cicada [38], adopt MVCC. While there are some common challenges with MV-RLU to optimize MVCC database systems, such as timestamp allocation, version storage, garbage collection [64], those techniques are tightly coupled with the internal design and semantics of database systems. In contrast, the key innovation of this paper is not supporting MVCC, but rather how to do it in a generalized way to performantly support any given data structure. Also, we believe some of our proposed techniques, especially concurrent autonomous garbage collection scheme, can be adopted by database systems.

Software transactional memory. Despite that there have been a lot of research efforts on software transactional memory (STM) [15, 18, 20, 26, 29, 31, 39, 49, 50, 56], STM approaches have not been widely adopted due to poor performance and scalability [17]. One main reason is the high STM runtime overhead to transparently support transactions and guarantee the strictest consistency level, linearizability. While there has been several efforts to mitigate such overheads, the overhead of managing and accessing STM metadata (e.g., lock table) is high. While both STM and MV-RLU aim to ease development of concurrent data structures, we made practical design choices for performance; MV-RLU adopts multi-versioning with snapshot isolation and provides a familiar lock-based programming model for ease-of-programming.

Synchronization framework for CDS. Recently, several synchronization frameworks [3, 6, 19, 23, 27, 28, 40, 55, 60, 67] were proposed to easily convert sequential (or lock-based) data structures to concurrent data structures. This eases the burden of developing new concurrent data structures. Combining [19, 27] and delegation [40, 55] approaches essentially perform single-thread execution to reduce synchronization overhead so their performance is bounded by single core performance. NR [6] transforms a sequential data structure to a NUMA-aware concurrent data structure. It uses flat combining [27] and a shared operational log to synchronize per-NUMA replicas. Unlike typical combining and delegation approaches, NR allows multiple readers to access a replica but restricts readers from accessing replicas that are currently held by writers (replaying log back to its replica). In comparison with these approaches, MV-RLU provides a higher concurrency due to MVCC. Versioned programming [67] is the most similar work with MV-RLU. Like MV-RLU, it converts a pointer-based data structure to a MVCC-enabled data structure supporting snapshot isolation and composability. However, its details including programming model, locking scheme, version traversal, commit protocol, and garbage collection are significantly different.

9 Conclusion

MV-RLU is a synchronization mechanism that extends RLU using mutli-versioning, while preserving the benefits of RLU like multi-pointer update, a simple & intuitive programming model, and good performance for read-mostly workloads. MV-RLU alleviates problems such as high contention of global clock (using a hardware clock-based timestamp); slow read performance with an autonomous garbage collector design, and slow garbage collection with a concurrent garbage collector design that are ubiquitous to most MVCC designs. As a result, MV-RLU outperforms other synchronization mechanisms in several workloads and shows unmatched scalability even in write-intensive workloads.

Acknowledgment

We thank the anonymous reviewers for their helpful feedback. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2014-3-00035).

References

[1] Maya Arbel and Adam Morrison. 2015. Predicate RCU: An RCU for Scalable Concurrent Updates. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, San Francisco, CA, 21–30.

[2] Jens Axboe. 2019. fio: Flexible I/O Tester. https://github.com/axboe/fio.

[3] Silas B. Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2013. OpLog: a library for scaling update-heavy data structures. *CSAIL Technical Report* 1, 1 (2013), 1–12.

[4] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Vancouver, Canada, 1–16.

[5] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2009. Serializable Isolation for Snapshot Databases. *ACM Trans. Database Syst.* 34, 4, Article 20 (Dec. 2009), 42 pages.

[6] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, Xi'an, China, 207–221.

[7] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2012. Scalable Address Spaces Using RCU Balanced Trees. In *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. London, UK.

[8] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2013. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *Proceedings of the 8th European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic.

[9] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. 2013. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Farmington, PA.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*. ACM, Indianapolis, Indiana, USA, 143–154.

[11] Intel Corp. 2017. Intel Xeon Platinum 8180 Processor. https://ark.intel.com/products/120496/Intel-Xeon-Platinum-8180-Processor-38_5M-Cache-2_50-GHz.

[12] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. 2012. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems* 23, 2 (Feb 2012), 375–382. https://doi.org/10.1109/TPDS.2011.159

[13] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD/PODS Conference*. ACM, New York, USA, 1243–1254.

[14] David Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. 2015. Adaptive integration of hardware and software lock elision techniques. In *Proceedings of the ACM symposium on Parallelism in algorithms and architectures (SPAA)*. ACM, Prague, Czech Republic, 188–197.

[15] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional Locking II. In *Proceedings of the 20th International Conference on Distributed Computing (DISC)*. Springer Berlin Heidelberg, Stockholm, Sweden, 194–208.

[16] Aleksandar Dragojevic. 2014. SwissTM:open source code. https://github.com/nmldiegues/tm-study-pact14/tree/master/swissTM.

[17] Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guer raoui. 2011. Why STM Can Be More Than a Research Toy. *ACM Communication* (2011), 70–77.

[18] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. 2009. Stretching Transactional Memory. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, Dublin, Ireland, 155–165.

[19] Panagiota Fatourou and Nikolaos D. Kallimanis. 2012. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, New Orleans, LA, 257–266.

[20] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. 2010. Time-Based Software Transactional Memory. In *Proceedings of the IEEE Transactions on Parallel and Distributed Systems*. IEEE, California, USA, 1793–1807.

[21] Jeremy Fitzhardinge. 2011. Userspace RCU. http://liburcu.org/.

[22] Johan De Gelas. 2018. Assessing Cavium's ThunderX2: The Arm Server Dream Realized At Last. https://www.anandtech.com/show/12694/assessing-cavium-thunderx2-arm-server-reality/2.

[23] Rachid Guerraoui and Vasileios Trigonakis. 2016. Optimistic Concurrency with OPTIK. In *Proceedings of the 21st ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, Barcelona, Spain, 18:1–18:12.

[24] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Hollywood, CA, 135–148.

[25] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-lists. In *Proceedings of the 20th International Conference on Distributed Computing (DISC)*. Springer Berlin Heidelberg, University of Lisboa, Portugal, 300–314.

[26] A. Hassan, R. Palmieri, S. Peluso, and B. Ravindran. 2017. Optimistic Transactional Boosting. *IEEE Transactions on Parallel and Distributed Systems* 28, 12 (Dec 2017), 3600–3614.

[27] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat Combining and the Synchronization-parallelism Tradeoff. In *Proceedings of the ACM symposium on Parallelism in algorithms and architectures (SPAA)*. ACM, Thira, Santorini, Greece, 355–364.

[28] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Scalable flat-combining based synchronous queues. In *International Symposium on Distributed Computing*. Springer, 79–93.

[29] Maurice Herlihy and Eric Koskinen. 2008. Transactional Boosting: A Methodology for Highly-concurrent Transactional Objects. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, Salt Lake City, UT, 207–216.

[30] Maurice Herlihy and Nir Shavit. 2011. *The Art of Multiprocessor Programming*. Morgan Kaufmann.

[31] Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2016. Type-aware Transactions for Faster Concurrent Code. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*. ACM, London, UK, 31:1–31:16.

[32] Intel. 2018. Intel 64 and IA-32 Architectures Optimization Reference Manual. https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf.

[33] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. 2018. A Scalable Ordering Primitive for Multicore Machines. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*. ACM, Porto, Portugal, Article 34, 15 pages.

[34] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2015 ACM SIGMOD/PODS Conference*. San Francisco, CA, USA, 1675–1687.

[35] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD/PODS Conference*. ACM, Melbourne, Victoria, Australia, 691–706.

[36] FAL Labs. 2011. Kyoto Cabinet: a straightforward implementation of DBM. http://fallabs.com/kyotocabinet/.

[37] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Seattle, WA, 429–444.

[38] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM SIGMOD/PODS Conference*. ACM, Chicago, Illinois, USA, 21–35.

[39] Heiner Litz, David Cheriton, Amin Firoozshahian, Omid Azizi, and John P. Stevenson. 2014. SI-TM: Reducing Transactional Memory Abort Rates Through Snapshot Isolation. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, Salt lake city, UT, 383–398.

[40] Jean-Pierre Lozi, Florian David, Gaël Thomas, Juli a Lawall, and Gilles Muller. 2012. Remote Core Locking: Migrating Critical-section Execution to Improve the Pe rformance of Multithreaded Applications. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*. USENIX Association, Boston, MA, 6–6.

[41] Linux manual page. 2017. perf Manual. http://man7.org/linux/man-pages/man1/perf.1.html.

[42] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. 2015. Read-log-update: A Lightweight Synchronization Mechanism for Concurrent Programming. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Monterey, CA, 168–183.

[43] Paul E. McKenney. 1998. Structured Deferral: Synchronization via Procrastination. *ACM Queue* (1998), 20:20–20:39.

[44] Paul E. McKenney. 2012. RCU Linux Usage. http://www.rdrop.com/~paulmck/RCU/linuxusage.html.

[45] Paul E. McKenney, Jonathan Appavoo, Andy Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. 2002. Read-Copy Update. In *Ottawa Linux Symposium (OLS)*.

[46] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at what COST?. In *15th USENIX Workshop on Hot Topics in Operating Systems (HotOS) (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland.

[47] Maged M. Michael. 2002. Safe Memory Reclamation for Dynamic Lock-free Objects Using Atomic Reads and Writes. In *Proceedings of the 21st ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*. Monterey, California, 21–30.

[48] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. 2016. Understanding Manycore Scalability of File Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*. USENIX Association, Denver, CO, 71–85.

[49] Donald Nguyen and Keshav Pingali. 2017. What Scalable Programs Need from Transactional Memory. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, Xi'an, China, 105–118.

[50] Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. 2007. Open Nesting in Software Transactional Memory. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, SAN Francisco, CA, USA, 68–78.

[51] Oracle. 2004. *Oracle Database Concepts 10g Release 1 (10.1) Chapter 13 : Data Concurrency and Consistency âĂŤ Oracle Isolation Levels*. https://docs.oracle.com/cd/B12037_01/server.101/b10743/consist.htm.

[52] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *Proceedings of the 39th biennial Conference on Innovative Data Systems Research (CIDR)*. Chaminade, California.

[53] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. 2012. Improving Network Connection Locality on Multicore Systems. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys)*. ACM, Bern, Switzerland, 337–350.

[54] PostgreSQL. 2018. Serializable Snapshot Isolation (SSI) in PostgreSQL. https://wiki.postgresql.org/wiki/SSI.

[55] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. 2017. Ffwd: Delegation is (Much) Faster Than You Think. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Shanghai, China, 342–358.

[56] Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. 2016. Transactional Data Structure Libraries. In *Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, Santa Barbara, CA, 682–696.

[57] Paul Teich. 2017. The New Server Economies of Scale for AMD. https://www.nextplatform.com/2017/07/13/new-server-economies-scale-amd/.

[58] Bill Thomas. 2018. AMD Ryzen Threadripper 2nd Generation release date, news and features. https://www.techradar.com/news/amd-ryzen-threadripper-2nd-generation.

[59] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Farmington, PA, 18–32.

[60] Qi Wang, Timothy Stamler, and Gabriel Parmer. 2016. Parallel Sections: Scaling System-level Data-structures. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*. ACM, London, UK, 33:1–33:15.

[61] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, New Delhi, India, 49–60.

[62] Wikipedia. 2018. Snapshot isolation. https://en.wikipedia.org/wiki/Snapshot_isolation.

[63] Chris Williams. 2018. Broadcom's Arm server chip lives - as Cavium's two-socket ThunderX2. https://www.theregister.co.uk/2018/05/08/cavium_thunderx2/.

[64] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-memory Multi-version Concurrency Control. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, TU Munich, Germany, 781–792.

[65] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, Hangzhou, China, 209–220.

[66] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2015 ACM SIGMOD/PODS Conference*. ACM, San Francisco, CA, USA, 1629–1642.

[67] Yang Zhan and Donald E. Porter. 2010. Versioned Programming: A Simple Technique for Implementing Efficient, Lock-Free, and Composable Data Structures. In *Proceedings of the ACM International Systems and Storage Conference*. ACM, California, USA, 11:1–11:12.