Multicore Scalability through Asynchronous Work

Ajit Mathew

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Changwoo Min, Chair
Haibo Zeng
Dongyoon Lee

December 2, 2019
Blacksburg, Virginia

Keywords: Multicore, Synchronization, Index Strutures

Multicore Scalability through Asynchronous Work

Ajit Mathew

(ABSTRACT)

With the end of Moore's Law, computer architects have turned to multicore architecture to provide high performance. Unfortunately, to achieve higher performance, multicores require programs to be parallelized which is an untamed problem. Amdhal's law tells that the maximum theoretical speedup of a program is dictated by the size of the non-parallelizable section of a program. Hence to achieve higher performance, programmers need to reduce the size of sequential code in the program. This thesis explores asynchronous work to reduce sequential portions of program. Using asynchronous work, a programmer can remove tasks which do not affect data consistency from the critical path and performed by background thread. Using this idea, the thesis introduces two systems. First, a synchronization mechanism, Multi-Version Read-Log-Update(MV-RLU), which extends Read-Log-Update (RLU) through multi-versioning. At the core of MV-RLU design is a concurrent garbage collection algorithm which reclaims obsolete versions asynchronously reducing blocking of threads. Second, a concurrent and highly scalable index-structure called Hydralist for multi-core. The key idea behind design of Hydralist is that an index-structure can be divided into two component (search layer and data layer) and updates to data layer can be done synchronously while updates to search layer can be propagated asynchronously using background threads.

Multicore Scalability through Asynchronous Work

Ajit Mathew

(GENERAL AUDIENCE ABSTRACT)

Up until mid-2000s, Moore's law predicted that performance CPU doubled every two years. This is because improvement in transistor technology allowed smaller transistor which can switch at higher frequency leading to faster CPU clocks. But faster clock leads to higher heat dissipation and as chips reached their thermal limits, computer architects could no longer increase clock speeds. Hence they moved to multicore architecture, wherein a single die contains multiple CPUs, to allow higher performance. Now programmers are required to parallelize their code to take advangtage of all the CPUs in a chip which is a non trivial problem. The theoretical speedup achieved by a program on multicore architecture is dictated by Amdhal's law which describes the non parallelizable code in a program as the limiting factor for speedup. For example, a program with 99% parallelizable code can achieve speedup of 20 whereas a program with 50% parallelizable code can only achieve speedup of 2. Therefore to achieve high speedup, programmers need to reduce size of serial section in their program. One way to reduce sequential section in a program is to remove non-critical task from the sequential section and perform the tasks asynchronously using background thread. This thesis explores the technique in two systems. First, a synchronization mechanism which is used co-ordinate access to shared resource called Multi-Version Read-Log-Update (MV-RLU). MV-RLU achieves high performance by removing garbage collection from critical path and performing it asynchronously using background thread. Second, an index structure, Hydralist, which based on the insight that an index structure can be decomposed into two components, search layer and data layer, and decouples updates to both the layer which allows higher performance. Updates to search layer is done synchronously while updates to data layer is done asynchronously using background threads. Evaluation shows that both the systems perform better than state-of-the-art competitors in a variety of workloads.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis presents an index structure and a sychronization mechanism which leverages asynchronous work to scale performance with increasing number of cores. The rest of the chapter presents the problem, proposed solution and the contribution of the thesis

1.1 Problem: Scalability in Multicores

Until mid-2000s, clock speeds in CPU followed the Moore's Law curve *i.e.* roughly about every two year, the performance of CPU doubled. This meant that programmers did not have to optimize their code to achieve speedup, rather wait for CPUs to get faster. But the free lunch was finally over when CPUs reached their thermal disspation limits preventing further increase in clock speeds of CPUs. To continue increasing performance of CPUs, computer architects adopted multicore architecture. To take advantage of the new architecture and achieve linear scalability (doubling CPUs, doubles program performance), programmers need to parallelize their program which is a non-trivial problem.

Amdhal's law predicts the maximum theoritical speedup when using multiple CPUs. Amdhal's law can be formulated as follows:

$$S_{latency}(s) = \frac{1}{(1-p) + p/s} \tag{1.1}$$

where

- $S_{latency}$ is the maximum theoretical speedup;
- s is the speedup of the part of the task that is parallelizable;
- p is the proportion of execution time of code that is parallelizable.

According to Amhdal's law, the theoritical speedup is limited by the part of program that cannot be parallelized. For example, a program with 50% parallel portion achieves a maximum theoritical speedup of only 2 whereas a program with 99% parallel portion achieves a maximum speedup of 20. This means to achieve high performance in multicore architecture, serial section in program needs to be minimized

1.2 Solution: Asynchronous Work

One way to reduce sequential part of a program is to remove non-critical tasks which do not affect the correctness of program from the critical path. For example, freeing of memory can be removed from critical path and done at a later time using background thread. This reduces the size of the sequential section (also called critical section) of program which in turn leads to higher performance as stated by Amdhal's law. Tasks moved out of sequential section and performed by background threads are called asynchronous work. In this thesis, we will explore how asynchronous work can be leverage to scale performance in multicore environment.

1.3 Contribution

Using asynchronous work as a core design idea for scalability, the thesis makes the following contribution:

1. A new synchronization mechanism, Multi-Version Read-Log-Update (MV-RLU) which extends an existing synchronization mecahnism, Read-Log-Update [56] using multi-versioning. At the core of MV-RLU design is a concurrent and autonomous garbage collection, which prevents reclaiming invisible versions being a bottleneck, and reduces the version traversal overhead—the main overhead of multi-version design. The garabge collection is done asynchronously thus reducing the size of MV-RLU critical section. Evaluation shows that MV-RLU significantly outperforms other techniques for a wide range of workloads with varying contention levels and data-set size.

2. A new index structure, Hydralist, which is a concurrent, scalable and high performance index structure for massive multi-core machines. The key insight behind design of HydraList is that an index structure can be divided into two components (search layer and data layer) which can be updated independently leading to lower synchronization overhead. Moreover, this further enables the replication of search layer reducing cache misses and remote memory accesses. As a result, our evaluation shows that HydraList outperforms other index structures especially in a variety of workloads and key types

1.4 Thesis Organization

Chapter 2

Background and Related Work

2.1 Synchronization Mechanisms

2.1.1 Background

Threads reading or writing to shared memory can cause data inconsistency and hence accesses to shared resources require synchronization. This is done using specialized algorithms called synchronization mechanisms. Synchronization mechanisms are an essential building block for designing any concurrent applications. Applications such as operating systems [11, 17, 18, 19], storage systems [52], network stacks [36, 64], and database systems [72], rely heavily on synchronization mechanisms, as they are integral to the performance of these applications. However, designing applications using synchronization mechanisms (refer Table 2.1) is challenging; for instance, a single scalability bottleneck can result in a performance collapse with increasing core count [17, 36, 61, 64, 72]. 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 [70, 71], ARM [35, 75], and Xeon servers [23] 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 [30, 31], only support certain types of workloads [9, 56, 58, 66], or are difficult to use [58].

2.1.2 Related Work

Current state-of-the-art work in synchronization mechanisms can be broadly categorized into linearizable and non-linearizable synchronization mechanisms based on the consistency guarantee they provide, which will be discussed in this section.

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

2.1.3 Linearizable Synchronization Mechanisms

Locks

Mutex and reader writer lock allow mutual exclusion in critical section. Mutexex allow only one thread to enter critical section at a time whereas reader writer locks allow multiple readers to concurrently read inside the critical section but allow only exclusive access for writers. Hence reader writer locks allow higher degree of parallelism.

Lock-Free

Downside of lock based approach is that they are blocking *i.e.* if one thread attempts to acquire a lock already held by another thread, then the thread blocks until the lock is release. In lock-free approaches, blocking is avoided because access to shared memory does not need to be serialized. Lock-free algorithms are usually implemented using atomic Compare-And-Swap (CAS) operation. Lock-free approaches provide higher parallelism but they suffer from cache-line bouncing (e.g., frequent CAS retries) for memory hot spots and have additional memory reclamation overhead [57, 58, 60]. Also designing correct lock-free algorithm is non-trival.

Delegation and Combining

In this approach, the critical section is executed by a single thread (called the servers) and all threads (called clients) needing access to the shared submit requests to the server and wait for response. In high contention workload, multiple threads trying to modify same cache line results in large cache coherence traffice which can cause performance. This situation is alleviated using delegation approach as the contention levels are always low on shared resource because only server thread has acces to shared memory. But the single-thread execution itself becomes a bottleneck, especially during long running workloads (e.g., look up a large hash table) [66].

Software Transcational Memory (STM)

STM borrows idea of transaction from databases and applies to shared memory. In STM, critical sections are converted into transactions and read/writes to shared memory in a transaction is tracked. On the exit of critical section (*i.e.*transaction commit), STM checks if any read or write causes consistency guarantees to be broken (called conflicts). If no conflicts are found the transaction is committed otherwise, it is aborted and thread retries. STM provides excellent programmability but its centralized metadata management (e.g., lock table) often becomes a scalability bottleneck [31].

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 [37]	●	●	△	✓	high	1	1	cacheline bouncing; memory reclamation
Delegation-style	ffwd [66]	×	×	×	✓	low	1	1	single-threaded execution of a critical section
	NR [15]	●	×	×	✓	low	1	# NUMA	limited parallelism
STM	SwissTM [30]	●	▲	△	✓	lowest	2	2	centralized metadata management (e.g., lock table)
	STO [41]	●	▲	△	✓	low	2	2	high amplification ratio
RCU-style	RCU [57]	●	●	×	-	high	1	1	single writer
	RLU [56]	●	●	△	-	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 2.1: 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.

2.1.4 Non-Linearizable Synchronization Mechanisms

Read-Copy-Update (RCU)

RCU [58] is a widely used synchronization mechanism in the linux kernel. In RCU writes happen in three steps. First the writer reads the memory to be modified (Read). Then the writer makes a copy of the memory and modifies the copy (Copy). Note, any concurrent readers are not blocked as modifications are happening to the copy of shared memory not the actual copy. Then finally using a CAS opertaion, the copy is swapped with the original shared memory (Update) which is then garbage collected. RCU works well in read heavy workloads as writers do not block readers. But in write heavy workloads, RCU does not perform well. Moreover, since RCU is based on CAS operation, only making multiple pointer updates correctly is difficult making programming with RCU cumbersome.

Read-Log-Update (RLU)

RLU improves the programmability of RCU by extending it to allow atomic multi-pointer updates. 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. In RLU, reader gets a consistent snapshot of the data-structure protected by RLU, which removes the need of data structure specific pre-commit validation step required by most lock-free data structures.

2.2 Index Structures

2.2.1 Background

In databases, a column in a row is designated as the representative of the row and is called the primary key. Operations on databases are done usually done using the primary key. To accelerate search of a row, databases use index structures. They can be broadly classified into three categories.

Hash Index

Hash indexes [32, 40, 51] stores keys in a location designated by the hash of the key. Hash indexes allow lookup, insert and remove in $O(1)$ time. But they do not support range queries as hashing scatter keys randomly making them ineffective in large number of use case.

Tree based index

Second class of index structures are tree-based indexes like B-Tree. Many variant of B-Tree have been proposed including B+ tree, k-ary tree [67], PALM [68]. They usually have $O(\log n)$ access time where n is the number of keys stored in an index.

Trie based index

Third, trie-based index structure [47, 54, 62] which use digital representation of key instead of hash or key comparison. Search in a trie index has $O(k)$ complexity where k is the length of the key.

Hybrid Index

Finally, there are hybrid index structures which are creative combination of aforementioned index classes like Masstree [55] (B+ tree and trie) and Wormhole [76] (hash table and B+ tree).

2.2.2 Related Work: Scaling Index Structures

The work in this thesis is inspired from previous work on concurrent data structure design and many core scalability. In this section we refine the most relevant ondes to the following principles:

Reduce Synchronization Overhead

The scalability of an index structure heavily depends on the underlying synchronization mechanism that is being used [12, 26, 27]. Traditional concurrent B-Tree uses lock coupling to reduce the number of locks held while traversing the tree [10]. In lock coupling, a reader, while traversing from root to leaf node, holds lock on a node until it has acquired lock on its child node. Once the lock on child node is acquired, the parent node lock is released and the process is repeated until the leaf node is reached. This approach reduces contention on a root of B-Tree but frequently acquiring and releasing locks does not scale well in multicore systems as it creates large cache coherence traffic [16]. Optimistic synchronization techniques, such as optimistic lock coupling [13, 48] and OLFIT [16], have been proposed to reduce the synchronization overhead of frequent lock acquisition. In optimistic synchronization, readers (*i.e.*, lookup) check the version number before and after reading a node and retry if the versions do not match. Unlike a typical reader-writer lock, which modifies the lock variable regardless of readers and writers, readers using optimistic synchronization do not modify the lock variable so it reduces cache invalidation traffic and improves scalability.

Reduce the Size of Critical Section

Amdhal's law predicts the maximum theoretical speedup when using multiple processors. If for a program, the maximum theoretical speedup, which is determined by a sequential portion of a program, is achieved, adding more cores to the program will not yield higher performance. In this case, the only way to achieve higher performance is to reduce the size of critical section (*i.e.*, sequential portion). For example, if 95% of a program can be parallelized, it will reach 12× speedup with 32 cores. But if the serial section in the program is reduced to allow 99% parallelization, the speedup at 32 cores jumps to 24× with the maximum theoretical speedup of 90×. This means reducing the size of critical section is effective to improve performance and scalability. A general technique to achieve this is to delegate non-critical jobs like garbage collection to background threads [50] or to use specialized hardware like vector processing units [46, 67, 68] for faster execution in a critical section.

Reduce Cache Misses

Previous work has identified that data cache misses are a significant component of database execution time and can be more than 50% of total execution time for certain workloads and configurations [6]. Therefore, many techniques have been proposed to reduce cache misses. Cache sensitive B+ trees stores all child nodes of a given node in contiguous memory [65] to reduce prefetcher-unfriendly pointer chasing and to make memory accesses hardware prefetcher friendly. Masstree uses many techniques like software prefetching, cache optimized fan-out to reduce cache misses [55].

Reduce Cross-NUMA Memory Access

To accommodate many cores in a single machine, computer architects have adopted Non-Uniform Memory Architecture (NUMA) wherein cores are clustered into groups called NUMA nodes or simply node. NUMA architecture allows scaling of a machine to large number of cores but a side effect of this design is that cross-NUMA memory/cache-line accesses are more expensive than within a node. Thus, concurrent data structures should carefully handle such NUMA-ness to scale in a large multi-core architecture. Our experiments have shown that Wormhole performs worse than ART for read-only workload because of high cross-NUMA traffic, even though Wormhole uses fewer comparisons in case of lookup. A common technique to reduce cross node communication is to replicate shared memory across all NUMA nodes [15, 25]. The challenge with replication is to maintain consistency across all replicas while ensuring minimal synchronization overhead.

Chapter 3

Multi-Versiong Read-Log-Update

3.1 Motivation

As described in section §2.1, RLU provides great programmability by allowing multi-pointer atomic updates on data structures and also removes the need of data structure specific pre-commit validation step required by most lock-free algorithm. Despite its ease of programming, RLU shows limited scalability. For example, Figure 3.1 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 3.2). This takes up to 99.6% of CPU time at 448 cores in Figure 3.1. In other words, DVCC significantly hinders concurrency.

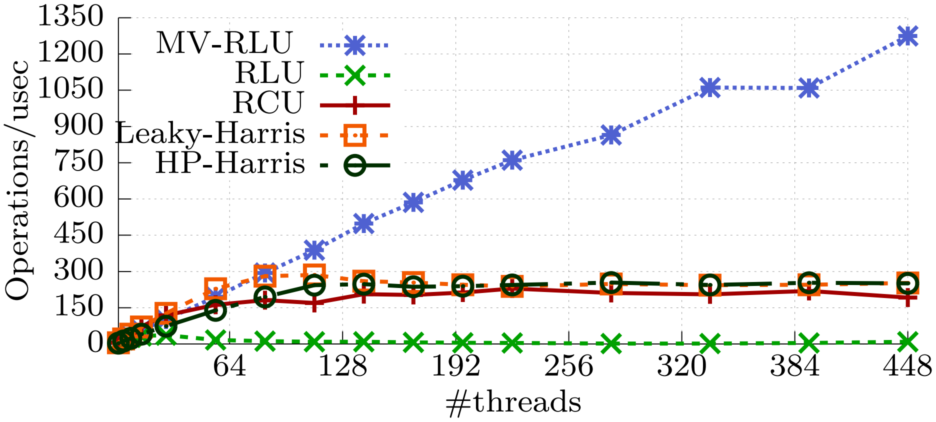
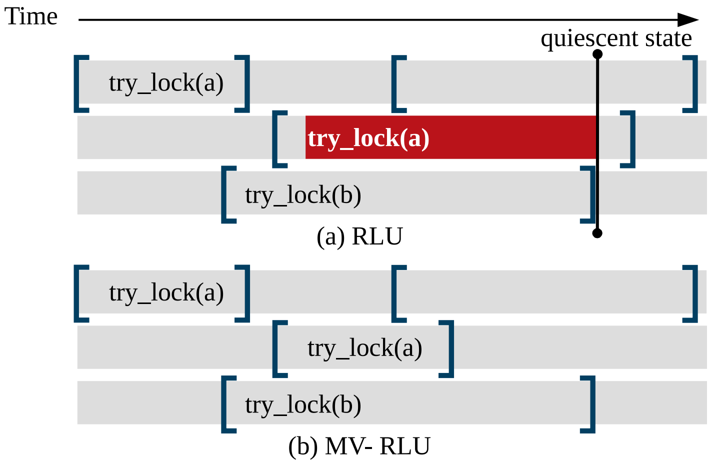


Figure 3.1: Performance comparison of concurrent hash tables having 1,000 elements with a load factor 1. Hash table access follows 80–20 Pareto distribution with 10% update. None of the existing approaches scale beyond 100 cores, except MV-RLU.



(a) RLU

(b) MV- RLU

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

Figure 3.2: 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.

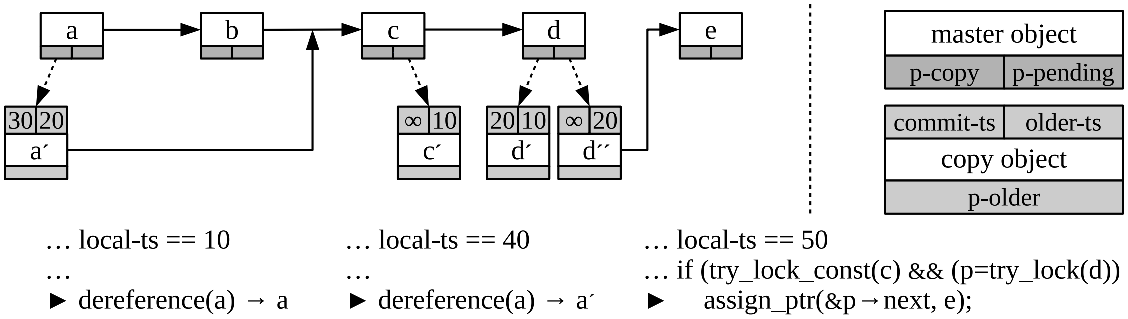
3.2 Multi-Version Read-Log-Update:Overview

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 section will provide an overview of MV-RLU algorithm.

3.2.1 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



```
… local-ts == 10            … local-ts == 40            … local-ts == 50
…                           …                           … if (try_lock_const(c) && (p=try_lock(d))
▶ dereference(a) → a        ▶ dereference(a) → a´       ▶    assign_ptr(&p→next, e);
```

Figure 3.3: 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 [39]. 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.

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 3.3, thread T3 uses a well-known hand-over-hand locking [39] 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.[1] 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.

3.2.2 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

[1] 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.

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 3.3. A master object, 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.

3.2.3 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 3.3, 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.

3.2.4 Consistency Guarantee

MV-RLU satisfies snapshot isolation, in which a thread observes a consistent snapshot of objects [14]. Snapshot isolation has been widely adopted by major database systems because it allows better performance than stricter consistency levels [74].

However, it is not serializable as it permits write skew anomaly [14]. 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 [4, 28, 53] 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 3.3, 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.

3.3 Design of MV-RLU

3.3.1 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 maintiaining 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 [23, 70, 75]. Finally, the performance of MV-RLU should be optimal even for smaller core counts because previous studies [59] 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.

3.3.2 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 §3.3.5). 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.

3.3.3 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.

3.3.4 Writing an Object

To modify an object, a writer tries locking the master object with try_lock. On failure (i.e., p-pending $\neq$ 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.

3.3.5 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 §3.3.2). 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.

3.3.6 Abort a Critical Section

On failure of a try_lock call (§3.3.4), 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.

3.3.7 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 3.1) 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 §3.5.2 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 [43], 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 2.1 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.

3.3.8 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.

3.3.9 Timestamp Allocation

Prior works [44, 45, 53, 72, 77] have shown that timestamp allocation is a major bottleneck in MVCC-based designs. To alleviate this issue, we use a hardware clock (RDTSC 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 [44], 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 §3.5.3 for correctness.

3.4 Implementation

We implemented MV-RLU in C comprising of 2,250 lines of code.[2] 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 not

[2] 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.

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 derefer-ence 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.[3] 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 §3.6.3).

3.5 Correctness of MV-RLU

3.5.1 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.

3.5.2 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. □

[3] 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.

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. □

3.5.3 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. □

3.5.4 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. □

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? (§3.6.2)
- What is the impact of our proposed approaches on MV-RLU? (§3.6.3)
- What is its impact on real-world workloads? (§3.6.4)

3.6.1 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 [57], RLU [56], RLU-ORDO (i.e., RLU using ORDO timestamp), Versioned-programming [79], SwissTM [30], Harris-Michael linked list [37], and predicate-RCU (PRCU) for Citrus tree [7, 17]. 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 [77] and KyotoCabinet [3]. 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).

3.6.2 Concurrent Data Structures

We first compare the scalability of three concurrent data structures (§3.6.2). Then we analyze MV-RLU behavior with different data set sizes (§3.6.2) and vary contention levels with different skewness of access (§3.6.2).

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 3.4 shows that MV-RLU outperforms all synchronization mechanisms because of multi-versioning coupled with efficient garbage collection. RCU does not scale with increasing update ratio because rlu_sync-hronize 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 3.2. 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 [5]. For the read-mostly workload, SwissTM scales up to 28 threads and then its performance is saturated. As update ratio increases, SwissTM shows

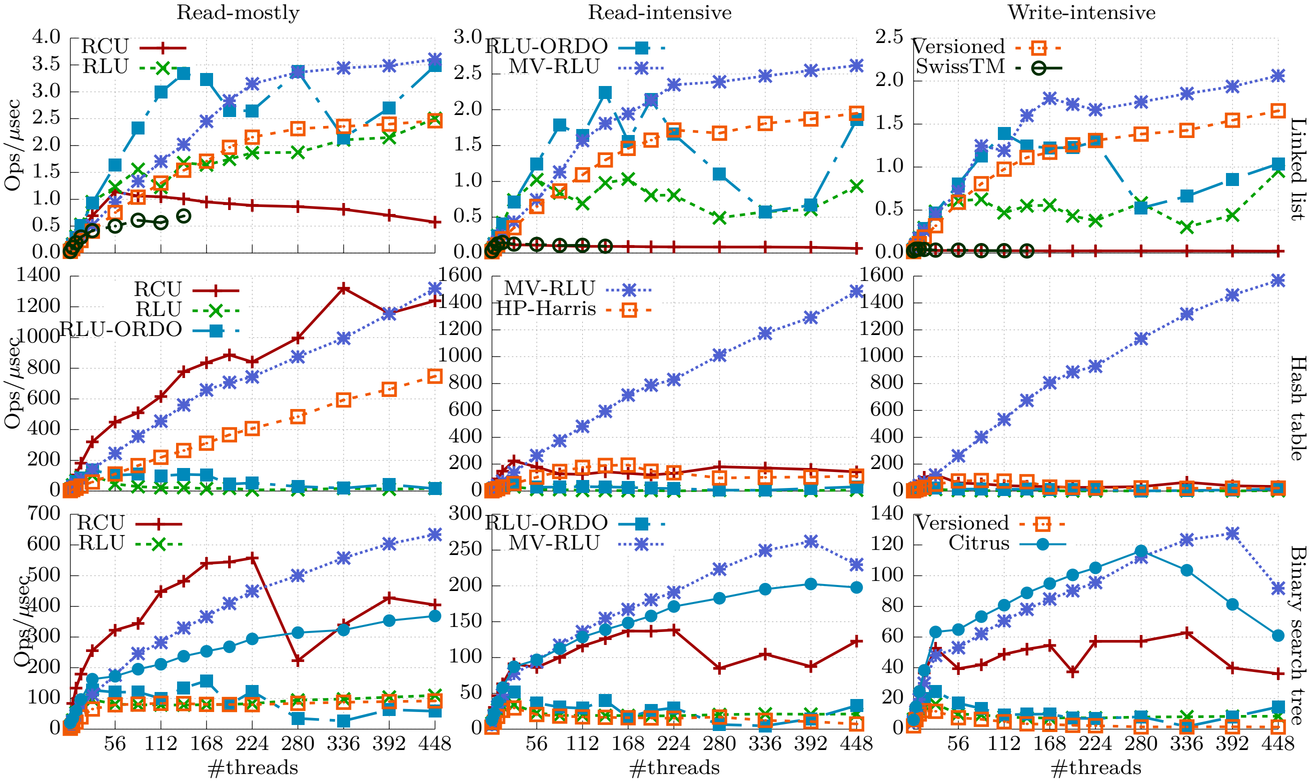


Figure 3.4: Throughput for a 10,000 item linked list, hash table (1K buckets), and binary search tree with read-mostly, read-intensive, and write-intensive workloads (first column: read-mostly, second column: read-intensive, third column: write-intensive), (first row: linked list, second row: hash table, third row: binary search tree)

significantly worse performance because of frequent read-write and write-write conflicts. We will further analyze its abort ratio under high contention in §3.6.2.

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 3.4 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 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 [60] 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 [**?**] analysis of HP-Harris hash table at 448 threads for write-intensive workload. The analysis indicates that memory barrier in object dereference 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 [7, 17]. 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 list and hash table in Figure 3.5. 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.

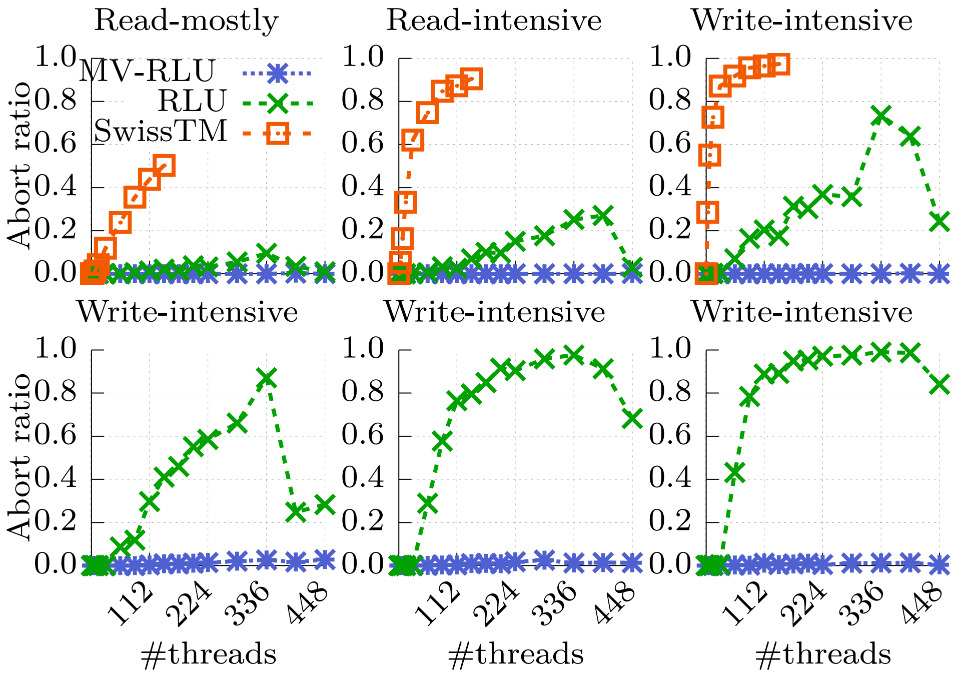


Figure 3.5: Abort ratio of linked list (top) and hash table (bottom) with 10,000 items. Load factor of hash table is 10. Workloads are read-mostly, read-intensive, and write-intensive, respectively, from left to right.

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 3.6 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.

Contention

We evaluate the scalability of MV-RLU for the hash table benchmark with skewed access by

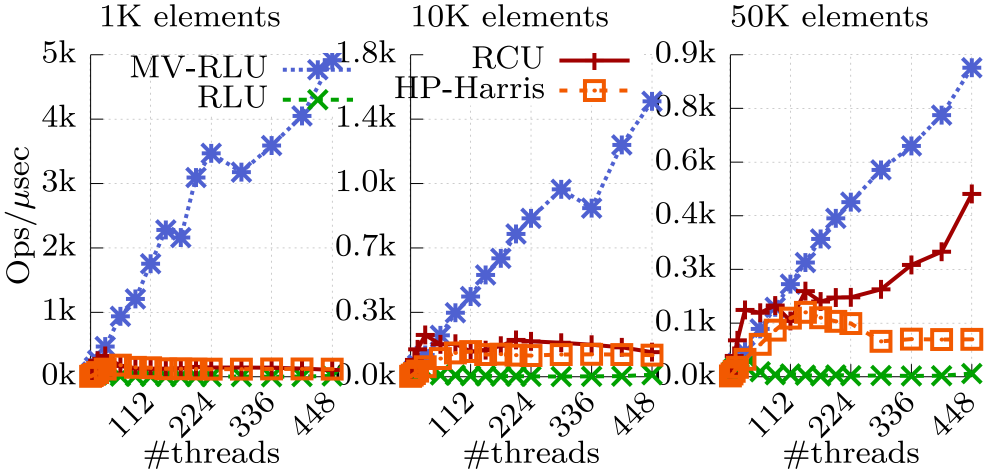


Figure 3.6: Hash table performance with 1K, 10K, and 50K items for the read-intensive workload. Load factors are 1, 10, 10, respectively, from left to right.

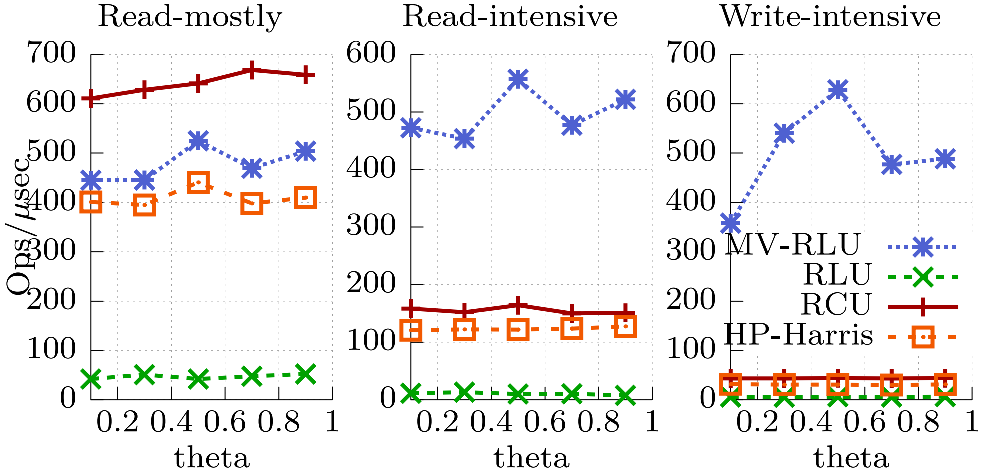


Figure 3.7: Performance of hash table with 10K nodes for read-mostly, read-intensive, and write-intensive workloads (left to right) at 336 threads. X-axis is theta value of Zipfian distribution. Higher theta value means that data access is more skewed.

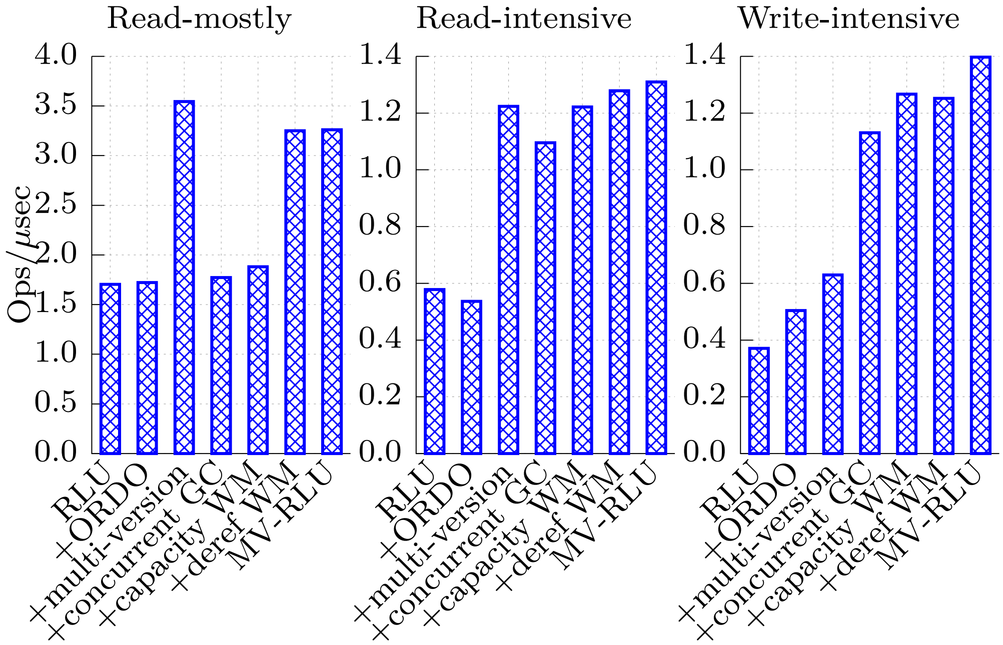


Figure 3.8: Contribution of MV-RLU features to performance. Design features are cumulative. The performance results of linked list with 10K items are presented at 336 threads for the read-mostly, read-intensive, and write-intensive workloads from left to right.

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 3.7 clearly shows the benefit of multi-versioning in MV-RLU. The performance of MV-RLU is nearly constant regardless of sknewness 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.

3.6.3 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 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.

3.6.4 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) [78], Silo (OCC) [72] and Hekaton (MVCC) [28]. 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 [21] 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.

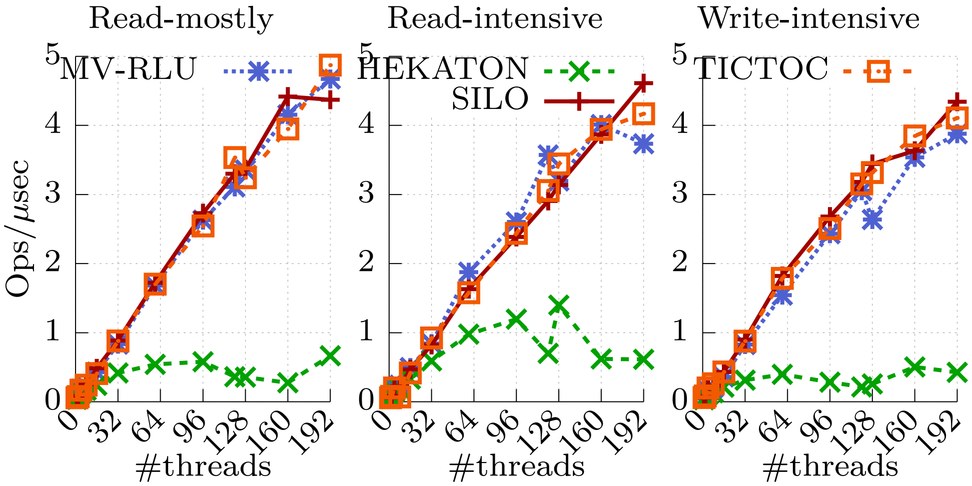


Figure 3.9: Performance comparison of MV-RLU, HEKATON, SILO, and TICTOC for YCSB benchmark for read-mostly, read-intensive, and write-intensive workloads with Zipf distribution of 0.7. MV-RLU outperforms HEKATON by 7–10×.

KyotoCabinet

KyotoCabinet Cache DB [3] 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 [29]. 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 3.10 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.

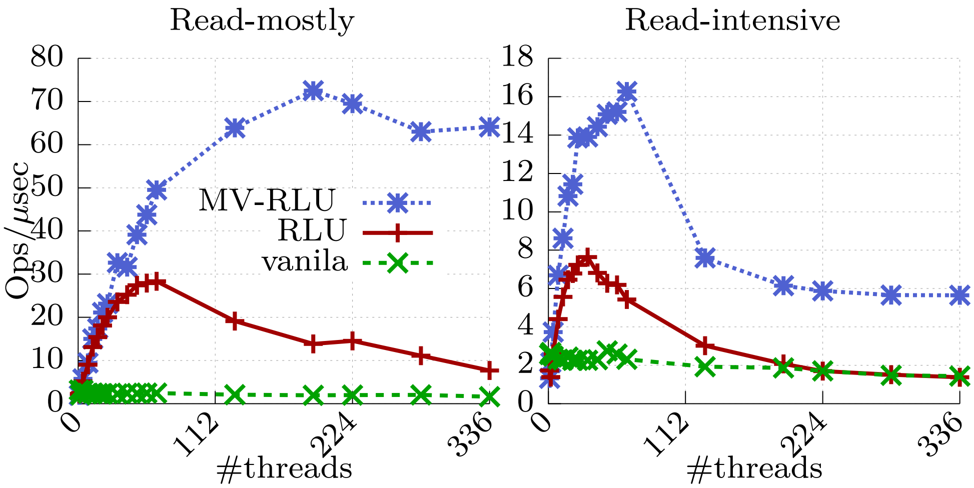


Figure 3.10: Performance comparison of MV-RLU, RLU, and the stock version in Kyoto-Cabinet benchmark with read-mostly (left) and read-intensive (right) workload, in which MV-RLU outperforms RLU by 3.8–8.4×.

Chapter 4

Hydralist

4.1 Motivation

In a modern server, time complexity of operations is not the sole factor that determines the performance of an index. With advent of multicore architecture, most modern database use concurrent index structures which allow multiple threads to perform operations on the index concurrently and use synchronization mechanism like locks, memory barriers and atomic instructions to coordinate access to shared memory. The underlying synchronization mechanism can greatly affect the performance of index and can even cause performance collapse at high core count. For example, in Figure 4.1, Libcuckoo [51], an efficient and concurrent implementation of cuckoo hashing shows performance collapse at 14 cores because the lock used to synchronize access to buckets becomes the bottleneck. Moreover, most higher performance servers today are non-uniform memory access (NUMA) machines and have complex memory hierarchy. Such machines can have long memory stalls which makes traditional assumption of uniform memory access time inaccurate in data structure design and reduces performance significantly [69]. *So for high performance in modern servers, an index structure should have 1) efficient time complexity of operations which is (ideally) independent of number of keys stored, 2) low synchronization overhead and 3) memory access patterns which are cache efficient and (mostly) NUMA-local.* We show that most index structures do not perform well for all different workloads and data sizes because they violate at least one of the three aforementioned performance factors.

4.2 Overview of HydraList

In this section, we describe our design goals for (§4.2.1) followed by the overview of our design choices to achieve high performance and scalability (§4.2.2).

4.2.1 Design Goals

Hydralist has three main design goals to be a generic in-memory index for modern multicore systems:

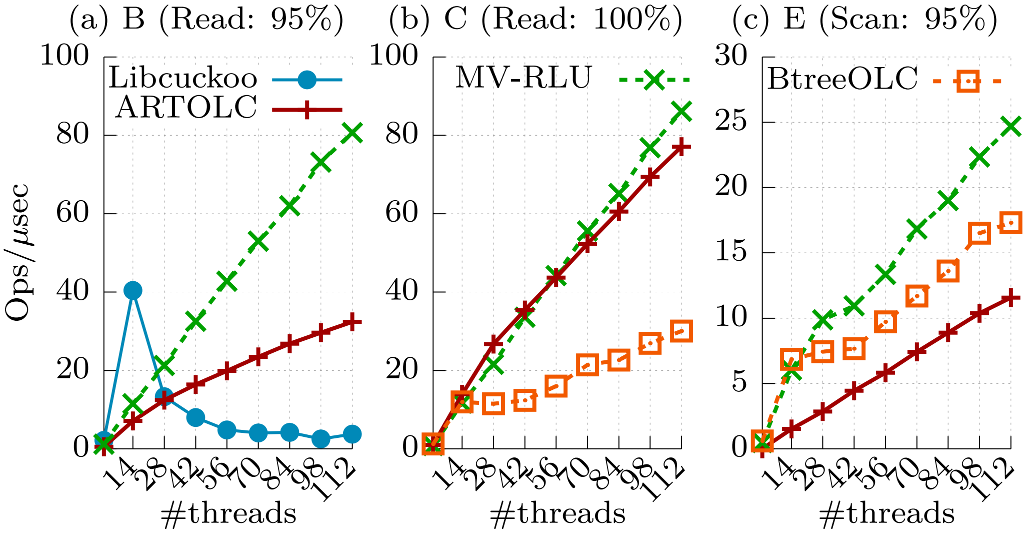


Figure 4.1: Performance of state-of-the-art indexes for YCSB workload with 89 million string keys. MV-RLU consistently performs and scales well regardless of workload types. (a) Libcuckoo [51], a concurrent and efficient implementation of cuckoo hashing shows performance collapse for read mostly workload because of spinlock at high core count. (b) Throughput of ART [48] with optimistic (ARTOLC) locking is 2.6× higher that B+ tree with same locking mechanism. (c) B+ tree (BtreeOLC) shows higher scan performance than ART because finding next key in scan is cache efficient.

- **Multicore Scalability:** Performance of Hydralist should scale with increasing core count. This is important as the number of cores in servers is rising which makes scalability of index structure critical for performance of databases.

- **Data Scalability:** Hydralist should be able to efficiently index large number of keys as increasing memory size allows more data to be stored in a database.

- **Versatility:** Hydralist should be performant for wide range of workloads. This eliminates the need for workload specific performance tuning.

None of the existing index structures achieve all of the aforementioned design goals. Hash tables have fast lookup but do not support scan operation making them unusable for a wide variety of applications. Skiplists and B-Tree based indexes do not scale well with increasing data size as their performance depends on number of keys stored. Trie based indexes on the other hand work well with large data sets but perform poorly on scan heavy workloads. Finally, our evaluation shows that most indexes do not perform well with increasing number of cores. We describe how Hydralist is able to achieve these design goals in the following sections.

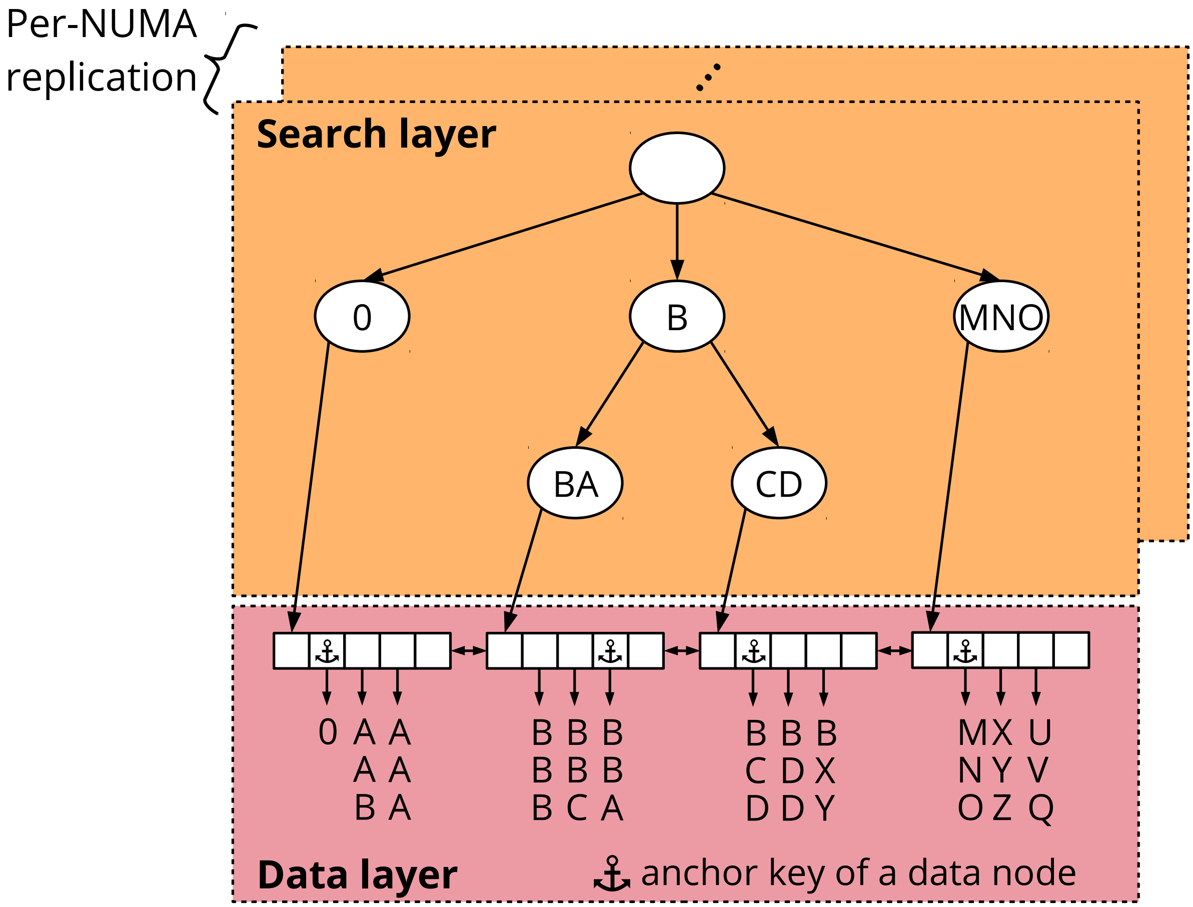


Figure 4.2: Hydralist uses Adaptive Radix Tree (ART) [48] as *search layer* and a slotted doubly linked list as *data layer*. To reduce remote memory accesses, the search layer is replicated per-NUMA node. Each node in data layer is called a *data node* which is indexed in the search layer using a representative key called *anchor key*.

4.2.2 Design Overview

To understand how Hydralist is able to achieve its design goals, we need to understand the general structure of index structures. Most tree-based index structures can be divided into two layers [1]: a search layer and a data layer. The data layer stores key-value pairs while the search layer stores partial keys (as in trie) or a subset of keys (as in B+ tree) which allows the reader to locate the key in data layer efficiently. Keys in data layer can be chained (as in B+ tree) allowing a faster scan or stored independently (as in trie). In these indexes, if needed, updates to the data layer are synchronously propagated to the search layer increasing the critical section of update operation which inhibits scalability. For

[1] Not all tree-based index structures can be decomposed into these two layers. For example a binary search tree stores value in every node.

example, an insert operation at a leaf node of a B+ tree (update to data layer) can cause a split which in turn can cause a split in the internal nodes of the tree (update to search layer). The insert operation is completed only when the search layer is consistent with the data layer. Therefore to design a high performance index structure, we need to design an efficient search layer, data layer and a synchronization mechanism to update both layers efficiently. In the rest of this section, we will explain the design overview of Hydralist and motivation behind our design choices.

Search Layer With increasing memory sizes in modern servers, it is expected that the number of keys stored in a database will also increase. For example, a study of Facebook's KV cache workload reveals that size of most keys lies between 20 and 40 bytes [8]. This means 32 GB of key data can contain 1.6 billion to 800 million keys which makes indexes with search layer whose lookup cost is proportional to the number of keys stored a poor choice. B+ tree, for example, with 100 million keys requires at least 26 (*i.e.*, $\log(100M)$) comparisons to find a key. Trie based index on the other hand have search cost proportional to the length of the key. Hence the number of keys stored in the index does not affect the performance of trie. But long keys can make search slower. This problem is alleviated by path compression wherein nodes with single child are merged together effectively reducing the search cost. Adaptive Radix Tree (ART) is a radix tree based index which supports resizing of internal nodes and path compression making it highly space efficient and performant. Therefore, we extend ART as the search layer in MV-RLU.

Data Layer Scan performance of ART is poor because leaf nodes which store the key-value pairs in ART are not chained together unlike B+tree. This forces the readers to jump between multiple levels to perform the scan operation. To alleviate this problem, we use slotted doubly linked list, called data list as the data layer in Hydralist (see Figure 4.2). Every node in the data list (called data node) stores multiple key-value pairs and is indexed in the search layer using a unique key (called anchor key).

Asynchronous Update In MV-RLU, search layer and data layer are decoupled, *i.e.*, updates from data layer to search layer are not propagated in a synchronous fashion rather updates to search layer are done using background threads. We use operational logging technique [9], wherein updates to search layer are enqueued in a per-thread queue. Periodically all operational logs are merged and search layer is updated. The search algorithm of Hydralist is designed to tolerate transient inconsistency when data layer is updated but updates to search layer is pending.

Synchronization Since the two layers are decoupled, Hydralist uses two different synchronization mechanisms. These mechanisms are chosen based on properties of the layers. Since the search layer is only updated by a single background thread and with progress of

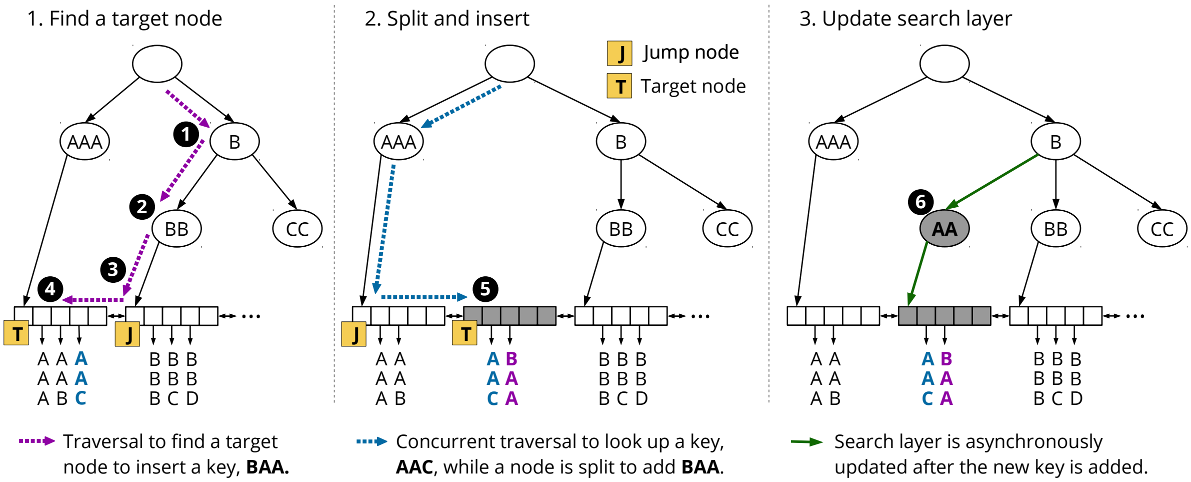

1. Find a target node
2. Split and insert
3. Update search layer
J Jump node
T Target node
AAA
B
BB
CC
AA
1
2
3
4
5
6
T
J
A A A
A A A
A B C
B B B
B B B
B C D
A A
A A
A B
A B
A A
C A
Traversal to find a target node to insert a key, BAA.
Concurrent traversal to look up a key, AAC, while a node is split to add BAA.
Search layer is asynchronously updated after the new key is added.

Figure 4.3: An illustrative example of inserting a key BAA in Hydralist. 1) First the writer has to find the node where this key should be inserted (*i.e.*, target node). For this it will perform a longest prefix match search ❶, then using the smaller child of the node perform a walk to the leaf node ❷ and jump to a data node (called a jump node) pointed by the leaf node ❸. Then it will traverse backwards to reach the target node ❹. 2) Since the target node is full, it is split and the key is added to the new node ❺. Note that lookup for a key in new node will be successful even though it does not have a reference in the search layer. Lookup for BAA will follow the path shown in blue until the search layer is updated. 3) Finally the search layer is updated asynchronously and the new node has a direct reference from the search layer (node ❻).

readers is important, we use Read-Optimized Write Exclusive protocol to synchronize ART in search layer which allows non-blocking reads. To allow parallelism and lock free traversal of data layer, we use optimistic version locking to synchronize nodes in data layer. They are described in detail later (§4.3.6).

Replication of Search Layer To reduce remote memory accesses, we replicate the search layer across NUMA nodes. This is possible because the two layers are decoupled and the search algorithm in Hydralist can tolerate inconsistency between them.

4.3 Design of HydraList

In this section, we will discuss the design of Hydralist. First we will discuss how search layer and data layer are organized and describe how search and other operations are done using an illustrative example in Figure 3.3.

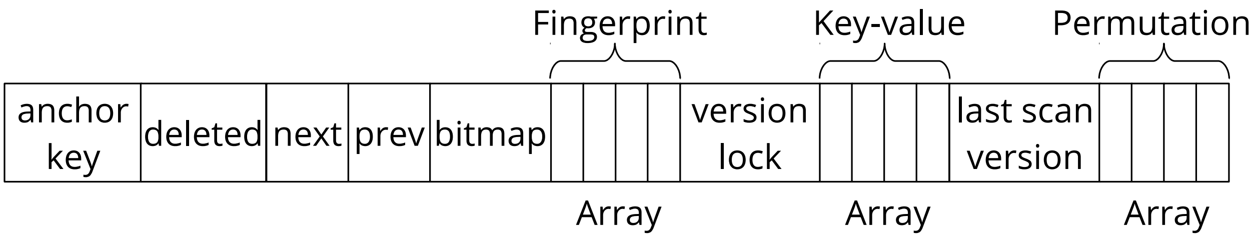


Figure 4.4: Layout of a data node

4.3.1 Data Layer and Search Layer

Figure 4.4 illustrates the layout of a data node. Every data node is assigned a unique key called the anchor key which is the smallest key of the node when it was inserted in the data list. For example, in Figure 4.2 the anchor node for second data node is BBA. Invariant in the data layer is that *any key stored in a data node should be greater or equal to the anchor key of the current node and should be smaller than the anchor key of the next data node (i.e.,* node->anchor_key <= key < node->next->anchor_key.

This invariant assigns a key range to every node and thus maps every key to a unique node in the data list. Key-value pairs are stored in contiguous memory locations in a data node but not in sorted fashion as maintaining sorted order for keys, especially string keys, is expensive. Instead, Hydralist stores 1-byte hash (called *fingerprint*) of every key and maintains a bitmap of every valid key-value slot. And also maintains a *permutation array* which stores the position of keys if the keys were sorted. Permutation array reduces the cost of sorting key-value array on every scan operation.

The ART-based search layer of Hydralist indexes all the nodes in a data list by storing anchor keys and the pointers of data nodes. A newly created data node might not be immediately indexed as updates to search layer are propagated asynchronously. However, this does not lead to incorrect results because the search operation in Hydralist can tolerate such inconsistencies which is described in the next section.

4.3.2 Search Operation

Every index operation requires finding location of key in Hydralist. Searching a key in Hydralist involves three steps. First, searching the search layer to find a data node (called *jump node*) from where search in the data layer should begin. Second, searching the data layer to find the exact data node (called *target node*) which might contain the key. And finally, searching the key in the target node. Note that in search layer, key is divided into 1-byte tokens and path compression is used for cache efficiency. For example in Figure 4.3, key BBB is represented using two nodes, B and BB. The leaf node of search layer stores the pointer to the data node of the corresponding anchor key.

```
1  node* getLowerBound(node* curNode, uint8 unmatchedToken) {
2    // With each node representing 1 byte of key
3    // A node can have 256 possible children
4    for (int i = unmatchedToken - 1; i >= 0; i--) {
5      if (curNode->getLevelChild(i) != NULL)
6        return curNode->getLevelChild(i);
7    }
8    return NULL;
9  }
10 node* findJumpNode(Key_t key, SearchLayer* sl) {
11   int level = 0;
12   // Longest Prefix Match
13   for (int i = 0; i < key.length(); i++) {
14     if (sl->checkPrefix(key, level)) level++;
15     else break;
16   }
17   slNode* LPMnode = sl->getNode(level);
18   if(LPMnode->type == LEAF_NODE) return sl->getValue(LPMnode);
19   // Find child node whose token is lower bound of unmatched token
20   slNode* Lnode = getLowerBound(LPMnode, key[level]);
21   if (Lnode) {
22     // Get the rightmost leaf node of subtree rooted at Lnode
23     while(!sl->isLeafNode(Lnode))
24       Lnode = getLowerBound(node, 255);
25   } else {
26     // Find child node whose token is upper bound of unmatched token
27     Rnode = getUpperBound(sl->getNode(level), key[level]);
28     // Get the leftmost leaf node of subtree rooted at Lnode
29     while(!sl->isLeafNode(Rnode))
30       node = getUpperBound(Rnode,0);
31   }
32   return sl->getValue(node);
33 }
```

Figure 4.5: Pseudo-code of finding jump node

Finding a Jump Node in Search Layer

Ideally, a jump node should be same as a target node. However, when they are not the same, which is possible when the search layer has pending updates, the jump node should be as close as possible to the target node. Since anchor keys divide the key space into key ranges, finding the a jump node is equivalent to finding the anchor key in the search layer which is lower or upper bound of query key (key). This can be done in our ART-based search layer using algorithm described in Figure 4.5.

Consider a query key $<t_1 t_2 ... t_l>$. First a reader will perform a longest prefix matching (LPM) between the query key and the keys in the search layer (Lines 13–16). If the query matches an anchor key, the search is over. Otherwise the reader has to find the next smaller or larger anchor key. Let p be the length of the longest prefix. Among the children of the last matched node, child node with token L is found such that $LowerBound(t_{p+1}) = L$ (Lines 1–9). Then the reader will walk to the right-most leaf node of the sub-tree rooted at this child (Lines 21–25). If lower bound token does not exists, then the same process is done but instead of walking to the right most leaf node, the reader searches for the left most leaf node

```
1  node* findTagetNode(node* jumpNode, Key_t key) {
2    node* cur = jumpNode;
3    while (1) {
4      if (cur->getAnchor() > key) {
5        cur = cur->getPrev();
6        continue;
7      }
8      if (cur->getNext()->getAnchor() > key) {
9        cur = cur->getNext();
10       continue;
11     }
12     break;
13   }
14   return cur;
15 }
```

Figure 4.6: Pseudo-code of finding target node

of the sub-tree (Lines 25–31).

For example, consider the search layer in Figure 4.3. If the query key is BAA, the LPM search will yield node with token B. Since no child of this node exists with token smaller than AA, node with token BB is used. Since it is a leaf node we immediately get the jump node.

Locating a Target Node in Data List

A target node can be found by traversing in the data list till a node is found which meets the invariant in §4.3.1. This traversal is done lock free as the reader only needs to read anchor keys of nodes which is never modified once a node has been created. The pseudo-code of finding a target node is described in Figure 4.6 (Line 1– 15).

Searching a Key in the Target Node

On finding the target node, a 1-byte fingerprint of the query key is generated. Only those key-value slots are probed whose fingerprint matches with the query key fingerprint and are valid as indicated by the bitmap. We found that with 1-byte fingerprint, on average 1.077 probes were required to key array in Hydralist, which is consistent with the results using the similar technique [63]. To further improve the performance, we use vector instructions to match multiple fingerprints in the node using single instruction.

4.3.3 Split and Merge of a Data Node

In Hydralist, a data node can store 64 key-value pairs. Inserts into a full node leads to splits. To split a data node, the writer will first lock the node, find the median key in the key-value array and move keys which are greater or equal to median key to a new data node. The median key is assigned as the anchor key of the new node. The new node is then inserted into the data list. Two adjacent nodes are merged when delete causes the total number of keys in the two nodes to be less than half the full capacity. Merge algorithm is similar to split. Figure 4.3 (b), illustrates the split process. The target node (first data node in data layer) of query key BAA is full, so the inserting thread creates a new node, finds the median of the three key and the new key (AAC) and moves keys greater or equal to median to the new node.

4.3.4 Decoupling Search Layer and Data Layer

Search layer has to be updated when an insert/delete in data layer causes split/merge of a data node. These structural modification operations (SMO) are traditionally done in a synchronous fashion. In other words, the split or merge are not made visible to other readers/writers until the modification has propagated to the search layer. Performing synchronous updates requires holding locks for multiple nodes, which becomes performance and scalability bottleneck especially for write-heavy workloads. In Hydralist, the search algorithm involves finding an anchor key in the search layer which is closest to the query key (not the exact key), hence a reader can tolerate a lazy updates to search layer as long as data layer is consistent. This insight allows decoupling of the two layers. Updates to data layer is done synchronously while updates to search layer is asynchronous. If the query key is located in a node whose anchor key has not been added to search layer, then the reader will land at a node further away from the target node. Hence the cost of inconsistency between two layers is longer traversal in data layer. However, as we demonstrate in §3.6, this cost is smaller than higher synchronization cost of updating both layers synchronously (see §4.5.3).

Asynchronous Update of Search Layer

Figure 4.7 illustrates asynchronous update in Hydralist which uses a per-thread operational log and two types of background threads, a combiner and updaters. To propagate updates from data layer to search layer, thread causing a split or merge stores the anchor key, pointer to new data node, current timestamp, and the operation (*i.e.*, insertion or deletion of an anchor key) in its per-thread operational log. For example, in Figure 4.3, after creating the new data node, the thread will enqueue an insert operation into its operational log along with other metadata. Per-thread logging is used instead of a global log as it reduces cross-NUMA traffic and synchronization between enqueuing threads. Timestamping ensures global ordering of events. A special background thread, called a *combiner*, periodically (*e.g.*,

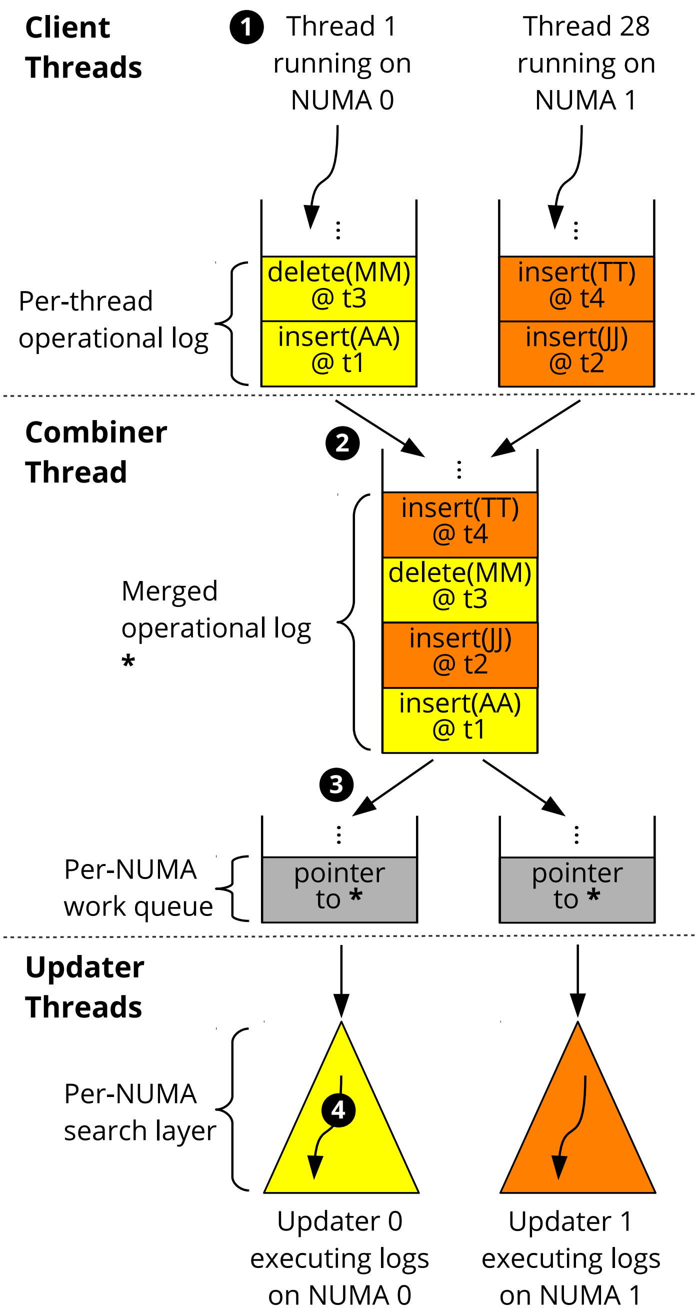


Figure 4.7: Illustration of how a combiner thread and updater threads are used to update search layer asynchronously. ➊ Client threads enqueue operations into their operational log. ➋ A combiner thread periodically reads all operational logs and returns a merged log which has operations sorted according to domestics. ➌ A merged operational log is broadcasted to all updater threads which update search layer. ➍ Per-NUMA updater thread applies operations to its local search layer.

200 microseconds) merges operation logs of all threads, sorts them according to timestamp and adds the pointer of merged log into the work queue of a second background thread, called *updater*, which updates the search layer asynchronously. Sorting operations according to timestamp is important for consistency of search layer as operations should be applied in the order in which they were created. An updater thread dequeues operations from its local work queue and execute the operations against the search layer. Using two different types of background threads allows task level parallelism. It should be noted that only a single thread updates a search layer but Hydralist can have multiple search layer (and hence multiple updaters) to achieve NUMA-awareness (discussed in §4.3.5). To generate timestamps, we use a previous work of generating scalable timestamp using hardware counters called ORDO [44]. We cannot directly use RDTSC instruction as hardware counters in different sockets have a constant skew between them. ORDO takes the constant skew into account and provides hardware timestamps which can be ordered correctly. Note that no special hardware is required to use ORDO primitive.

Reclaiming Merged Operational Logs

Once a merged log is created, its pointer is broadcasted to all updater threads. In other words, a merged log is shared by multiple updater threads. A merged log is safe to be deleted when all the operations in the merged log have been applied to the search layer by the updater threads. These merged logs are named *obsolete logs*. To detect an obsolete log, every merged log is given an monotonically increasing ID by the combiner thread at time of creation. Every updater thread keeps track of the merged log which it consumed. The combiner thread will check the periodically check with all updater threads and free merged logs which have ID smaller than the smallest obsolete log ID among the updater threads.

Reclaiming a Data Node from Search Layer

After a merge operation, the data node gets logically deleted from the data layer but references to the data node exists from the search layer. To safely reclaim the data node, we use a variation of Epoch Based Memory Reclamation (EBMR) techniques [34, 38, 58]. Here epoch is defined as the period between two consecutive cycles of freeing of obsolete merged logs. Start of an epoch guarantees that any deleted data node in the previous epoch has no memory reference from the search layer. Passing the first epoch means that all merged operational logs were applied by updaters. Hence there is no reference from a search layer to a logically deleted data node. However, it is possible that a reader that started reading in the previous epoch still references the deleted data node. To ensure no reader is currently reading the deleted data node, we wait for all the threads to exit the critical section at least once (*i.e.*, passing the second epoch). Then we physically free the memory of all the nodes which were deleted in the previous epoch.

4.3.5 Adding NUMA-Awareness to Search Layer

Since inconsistency between search layer and data layer can be tolerated, performance and scalability of Hydralist can be further improved by replicating search layer in every NUMA node. By doing so, every thread will perform only NUMA-local access while performing time-critical search in the search layer. NUMA replication of search layer is achieved by assigning a updater thread to every NUMA node and making the combiner thread broadcast the merged operational log to work queue of each updater. A client thread will search in the search layer which is in the same NUMA node removing expensive remote memory access during the search part of an index structure operation. NUMA node of a thread can be found using RDTSCP instruction which has a overhead of less than 50 cycles. If client threads are pinned, the node ID of each thread can be cached.

4.3.6 Concurrency

Search and data layer have different properties in terms of contention, memory locality and update frequency. Decoupling search and data layer allows concurrency control of both the layers to be designed independently and optimized for layer specific properties.

Concurrency in Search Layer

Search layer is updated by a single thread which means only reader-writer synchronization is required but writer-writer synchronization is not essential. Moreover, it is important that readers must proceed without blocking because search layers can tolerate delayed updates. Keeping these properties in mind, we use Read-Optimized Write Exclusion (ROWEX) [48] protocol to synchronize search layer in Hydralist. In ROWEX, a writer locks a node before modifying it, thus providing exclusion relative to other writers but readers do not acquire any locks. ROWEX ensures reads are safe by only allowing atomic modifications to fields that can be read concurrently. Nodes are replaced or added to ART using atomic compare-and-swap operation which makes old node invisible and new node visible atomically. Obsolete nodes are freed using garbage collection.

Concurrency in Data Layer

Data Layer in Hydralist is shared between all readers and writers. Therefore, the concurrency control of data layer should ensure maximum parallelism and reduce cache coherence traffic. To achieve this, Hydralist uses Optimistic Version Locking protocol. When a writer wants to modify a node, it first atomically increments the version number of the node. Since version numbers are initialized to zero, an odd version number indicates the node is being updated and any concurrent reader or writer trying to enter the critical section will have to retry.

After update, the node version is incremented again unlocking the node. A reader reads the version number of a node before and after accessing the node. If the version number does not match, then the reader retries. The advantage of this protocol is traversal in data layer are lock-free and do not modify cache-lines. A drawback of optimistic concurrency is large number of aborts under high contention. To prevent performance collapse due to high contention, Hydralist implements a backoff scheme wherein if a reader or writer performs retries more than a certain threshold, it waits for random period before attempting again. Our evaluation shows that this backoff scheme does not affect performance in low contention case and is effective in preventing performance collapse under high contention (§4.5.3).

4.3.7 Putting It All Together

Before starting an operation, a client thread determines its NUMA node and gets the root of the correspond NUMA node's search layer. The side effect of using optimistic locking techniques is that readers can read data nodes which is logically deleted. To avoid this, after entering the critical section, every reader/writer ensures that current data node is not deleted by checking the deleted field in the data node (Figure 4.4) and follows the invariant of the data list defined in §4.3.1. This is done by checkNodeValidity() described in Figure 4.8 (Lines 1–13).

```
1  bool checkNodeValidity(node* cur, Key_t key,
2      node &head, node jumpNode) {
3    if (cur->getDeleted()) { // Check if current node is deleted
4      if (cur == jumpNode) { // If deleted node is jump node
5        // Modify the start node of search in data layer
6        head = jumNode->getNext();
7      }
8      return false;
9    }
10   //Check if current node meets the key invariant
11   if (cur->checkInvariant(key))
12     return false;
13 }
```

Figure 4.8: Pseudo-code to check node validity

lookup(key)

For lookup of which pseudo-code is presented in Figure 4.9, reader reads the version number of the node using readLock() API (Line 5). The readLock() function returns 0 if the node is locked and in this case reader will retry. Also, reader will check the validity of the node. If the node is already deleted, it will retry the lookup operation. Then the node will find the index of the key in the key-value array using the fingerprint. If the key is found, the value of the key is fetch and before returning the value, the reader checks if the node version is unmodified(Lines 11–17). If not, the value is returned. Otherwise, it will retry the lookup operation.

```
1   Val_t lookup(node* jumpNode, Key_t key) {
2     node* head = jumpNode;
3   restart:
4     node* tagetNode = findTargetNode(head, key);
5     version_t readVersion = cur->readLock();
6     if (!readVersion) // Node locked, retry
7       goto restart;
8     if (!checkNodeValidity(cur, key, head, jumpNode))
9       goto restart;
10    uint8_t fingerPrint = getFingerPrint(key);
11    int index = cur->findKey(key, fingerPrint);
12    if (index == -1) // Key does not exits
13      return nullptr;
14    Val_t ret = cur->getValue(index);
15
16    // Check if the version has changed
17    if (!cur->readUnlock(readVersion))
18      goto restart;
19    return ret;
20  }
```

Figure 4.9: Pseudo-code of lookup

insert(key, value)

Insert operation starts by finding a target node (Figure 4.10, 4). After finding the target for k, the writer tries to lock the data node by incrementing the node version number using atomic compare and swap operation, if it fails meaning another writer already holds the lock, the writer aborts and retries (Line 5). After locking the data node, it checks again if the target node invariant is met and the node is not deleted, due to a concurrent split or a merge (Line 8). If the node is valid, the writer checks if key already exists in the node. To do this, it generates a fingerprint (*i.e.*, 1-byte hash) of the key and checks only those key array slots whose fingerprint matches. Fingerprint matching is accelerated using vector instructions. If such key does not exit, key-value is inserted in the first empty slot (Line 16). Finally, the version of the node is incremented again which means the node is unlocked (Line 17).

remove(key)

Remove algorithm is similar to insert(k,v). After locking the data node and checking its validity, the node tries to find the key in the node (Line 5–13). If successful, it deletes the key by resetting the bit corresponding to the key in the bitmap (Line 16).

```
bool insert(node* jumpNode, Key_t key, Val_t val) {
  node* head = jumpNode;
restart:
  node* tagetNode = findTargetNode(head, key);
  if (!cur->writeLock()) // Acquire write lock
    goto restart;
  // Check the node invariant are met
  if (!checkNodeValidity(cur, key, head, jumpNode)){
    cur->writeUnlock();
    goto restart
  }
  uint8_t fingerPrint = getFingerPrint(key);
  int index = cur->findKey(key, fingerPrint);
  if (index > -1) // If key does exits return
    return false;
  bool ret =  cur->insertKey(key, val, fingerPrint);
  cur->writeUnlock();
  return ret;
}
```

Figure 4.10: Pseudo-code of insert

```
bool remove(node* jumpNode, Key_t key, Val_t val) {
  node* head = jumpNode;
restart:
  node* tagetNode = findTargetNode(head, key);
  if (!cur->writeLock()) // Acquire Write lock
    goto restart;
  if (!checkNodeValidity(cur, key, head, jumpNode)) {
    cur->writeUnlock();
    goto restart;
  }

  uint8_t fingerPrint = getFingerPrint(key);
  int index = cur->findKey(key, fingerPrint);
  if (index == -1) // If key does not exits return
    return false;
  bool ret = cur->removeKey(index);
    cur->writeUnlock();
  return ret;
}
```

Figure 4.11: Pseudo-code of remove

scan(key, range)

Scan operation is described in Figure 4.12. Keys in a data node are not stored in a sorted fashion. This makes sorting keys on every scan operation a performance bottleneck. To reduce the cost of sorting, we use a permutation array which stores the indices of the keys in a sorted fashion (Lines 18–22). The permutation array is computed again only if there has been an update to the node since the last time it was computed. This can be detected by checking if the the version number of the node when permutation array was last generated and the most recent version number do not match. If anchor key of node is less than the start key, then starting index of scan is computed using binary search (Line 27). Then using the permutation array, result array is populated (Lines 28– 33). The process is repeat with the next node until size of the result array is same as scan range length.

```
 1 vector<Val_t>& scan(node* jumpNode, Key_t key, int range) {
 2   vector<Val_t> scanVector(range);
 3   node* head = jumpNode;
 4 restart:
 5   scanVector.clear();
 6   int done = 0;
 7   node* tagetNode = findTargetNode(head, key);
 8   while (done < range) {
 9     versiont_t readVersion cur->readLock();
10     if (!readVersion) // If node is locked, retry
11       goto restart;
12     if (done == 0
13         && !checkNodeValidity(cur, key, head, jumpNode))
14       goto restart;
15     int todo = range - done;
16 
17     // Check if an update has occurred since last scan
18     if (readVersion > cur->getLastScanVersion()) {
19       // If yes, regenerate Permutation Array
20       cur->generatePermuter();
21       lastScanVersion = readVersion;
22     }
23     uint8_t startIndex = 0;
24 
25     // Find index to start the scan
26     if (startKey > cur->getAnchorKey())
27       startIndex = binSearchLowerBound(startKey);
28     for (uint8_t i = startIndex;
29      i < cur->numEntries && todo > 0; i++) {
30       int index = cur->getPermuter(i);
31       scanVector[done++] = cur->getValueArray(index);
32       todo--;
33     }
34 
35     // Recheck read version, if changed retry
36     if (!cur->readUnlock(readVersion))
37       goto restart;
38     cur = cur->getNext();
39   }
40   return scanVector;
41 }
```

Figure 4.12: Pseudo-code of scan

4.4 Implementation

We implemented a prototype of Hydralist in C++. The prototype comprised of 4500 lines of code (LoC) which includes around 2600 LoC of concurrent Adaptive Radix Tree library [1]. We added about 300 LoC to the library to implement the findJumpNode function in §4.3.7. We use Intel Advanced Vector Extension (AVX 512) [42] for comparison of key fingerprint with fingerprint array. The size of data node is set to 64 keys. For 64 keys in a data node and

8-bit fingerprint, only one SIMD instruction is required for comparison. Also, we accelerate bitmap array operations using x86 assembly instructions. For timestamp allocation, we use hardware clock (RDTSC in x86 architecture) to prevent timestamp allocation from becoming a scalability bottleneck [72, 77].

In case of integer keys, Hydralist stores key-value pairs in the same array. For string keys, keys and values are stored in separate arrays. This improves scan performance as it reduces the number of cache misses.

4.5 Evaluation

In this section, we first begin by introducing our evaluation setup including evaluation platform, real-world workload used, and other in-memory indexes compared (§4.5.1). We then show performance results of Hydralist in comparison with other state-of-the-art indexes (§4.5.2). Finally, we analyze the effectiveness of our design choices (§4.5.3).

4.5.1 Experimental Setup

Hardware and Software Platform

We used a Intel Xeon Platinum 8180 server with 112 physical cores. It has four NUMA sockets with 28 physical cores per socket. It has 125 GB of memory. We disabled AutoNUMA [22], which automatically migrates memory from one NUMA domain to another NUMA domain, to avoid performance interference by underlying OS. Also, we enabled huge pages as some index evaluated use huge pages. We used jemalloc as memory allocator to prevent memory allocation from becoming a scalability bottleneck. We used gcc 8.3 with -O3 compiler flag to compile all indexes and benchmarks[2]. All experiments are done on Linux Kernel 5.0.16.

Real-World Workload: YCSB

We used the Yahoo! Cloud Serving Benchmark (YCSB) [21], which is a widely-used key-value store benchmark as it mimics real-world workloads as summarized in Table 4.1. We used an index benchmarking tool, index-microbench [73], which generates a workload file for YCSB and statically split them across threads. For each workload, we test two key types: integer and string. For integer key, we randomly generate 100 million 8-byte random

[2] For YCSB workload A and workload B, we compiled ART with -O0 compiler flag as it segfaults with higher optimization levels. We have notified the author but have not received a response at the time of submission.

Table 4.1: Characteristics of YCSB workloads

Workload	Application	Read/Write (Read %)
Load	Bulk database insert	Insert 0/100 (0%)
A	Session store	Read/Update 50/50 (50%)
B	Photo tagging	Read/Update 95/5 (95%)
C	User profile cache	Read 100/0 (100%)
D	User status update	Read(Latest)/Insert 95/5 (95%)
E	Threaded conversation	Scan/Insert 95/5 (95%)

integers. For string key, we use publicly available 89 million email addresses [33] to mimic the key distribution of real workload. The average length of email addresses is 20 bytes and maximum is limited to 32 bytes. For evaluation, the username and domain name of email IDs are swapped–*e.g.*, abc@xyz.com becomes xyz.com@abc–because that is a common pre-processing for trie-based indexes [47, 55, 73]. The value field in each key type is a 8-byte integer which mimics pointer to record in a real database. For each workload, we first load keys (100 million for integer and 89 million for string) and then run 50 million YCSB transactions on the keys.

In-Memory Index Comparison

We compared Hydralist with several state-of-the-art indexes: Adaptive Radix Tree (ART) [48] with optimistic lock coupling (ARTOLC) and read-optimized-write-exclusive synchronization (ARTROWEX), Masstree [55], B+ tree with optimistic lock coupling (BtreeOLC) [20, 49], open source implementation of Bw-Tree (Bw-Tree) [73], Wormhole [76], and Cuckoo hashing (Libcuckoo) [51]. Obviously, Cuckoo hashing does not support scan operation so we did not present Cuckoo hashing results for YCSB workload E, which requires scan operations.

We did not include the evaluation results of skiplists, including a lock-free version [24] and a NUMA-optimized version [25] because they did not perform and scale well with a large number of keys. We found that these skiplist algorithms do not scale because they use a single background thread to update the skiplist which becomes a bottleneck when number of keys increase. This is consistent with findings of previous work [73]. Increasing the number of background threads to improve scalability while maintaining correctness is non-trivial.

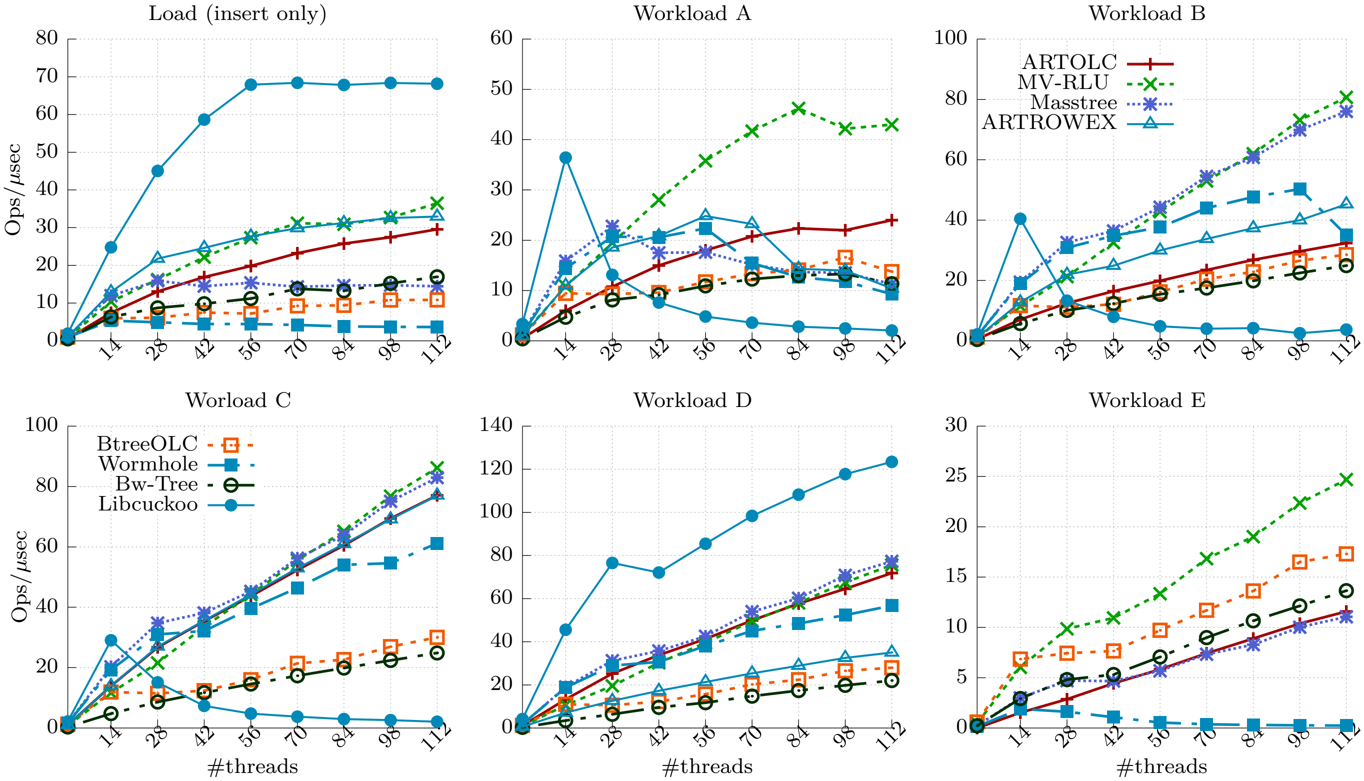


Figure 4.13: Performance comparison of in-memory indexes for YCSB workload: 50 million operations with 89 million string keys.

4.5.2 Performance Evaluation

Figure 4.13 and Figure 4.14 show the performance and scalability of in-memory index structures with YCSB workload.

Insert Only

Libcuckoo has the highest throughput for both integer and string keys because it randomly scatters the key which lowers contention. Hydralist throughput for integer key is 38.5% higher than ARTOLC because asynchronous update of search layer reduces size of critical section. Throughput of Hydralist and ARTROWEX is similar in case of string keys because overhead of string comparison is higher in Hydralist which is avoided by ART because of trie structure. Masstree, Wormhole and B+ tree showed performance saturation after 28 threads.

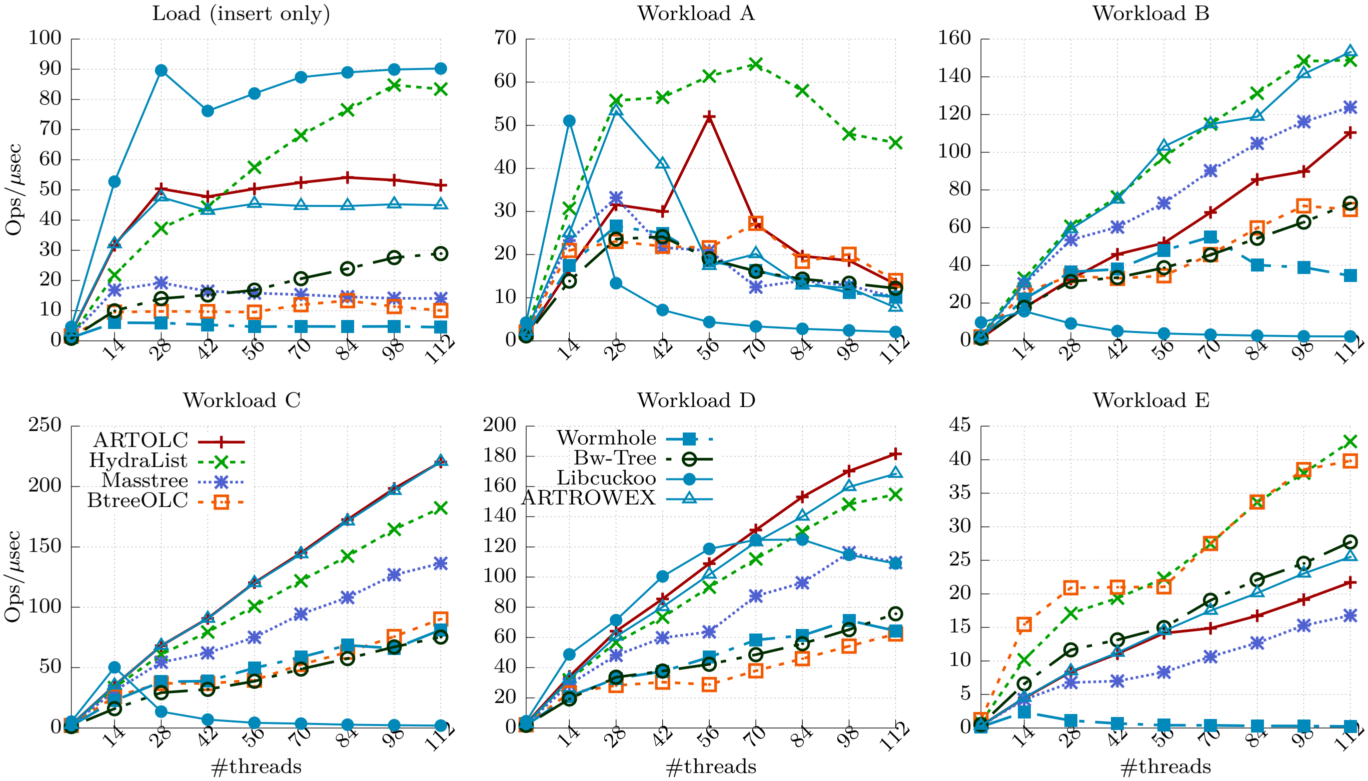


Figure 4.14: Performance comparison of in-memory indexes for YCSB workload: 50 million operations with 100 million integer keys.

Workload A

This workload performs large number of reads and updates with skewed access pattern which causes performance saturation or meltdown of most index structures at high core count as updates are serialized. Hydralist avoids performance collapse unlike Libcuckoo, ARTROWEX, and ARTOLC as it reduces contention using a backoff scheme. Thus Hydralist throughput is 6.4× higher in case of integer and 1.75× higher for string keys.

Workload B

All indexes show near linear performance scaling as this is a read mostly workload. For string keys, Masstree and Hydralist perform best. Masstree performs well because common prefix of string in our workload reduces the number of comparisons. B+ tree and Bw-Tree show lower performance because of large number of keys and high string comparison cost.

Workload C

Libcuckoo shows performance collapse because the lookup pattern is Zipfian and the per-bucket spinlock becomes bottleneck. In case of 112 thread for integer keys, performance profiling of Libcuckoo using Linux perf shows that 98% of the cycles are in acquiring spinlock. B+ tree and Bw-Tree show lower performance because of large number of keys which requires more number of memory access and comparisons. Performance of Wormhole is lower, even though its theoretical asymptotic complexity is better (*i.e.*, $O(log(length\ of\ key))$ because it uses a reader-writer lock to synchronize access to leaf nodes. Reader-writer lock take 15% of CPU cycles for 112 threads and hence becomes a scalability bottleneck. Performance of B+Tree and Bw-Tree is lower because of higher worst-case time complexity of lookup operation. Hydralist performs lower by about 20% than ART in case of integer keys but the performance is comparable in case of string keys.

Workload D

All indexes show near linear performance scaling as this is read-mostly workload. Performance of ARTOLC and Hydralist are comparable for both integer keys and string keys. In this workload, performance of Libcuckoo does not collapse because lower contention on buckets.

Workload E

Hydralist and B+ tree outperform all index structures in scan workload because slotted nodes structure of the data layer in the indexes allow readers to read values with next key in the range with fewer cache references as compared to ART or Masstree which have to traverse different levels. Wormhole performs poorly because scanning every node requires sorting of key array. Also Wormhole uses reader-writer lock which generates large cross-NUMA traffic leading to poor performance.

4.5.3 Analysis on Design Choices

In this section, we thoroughly analyze how our design choices affect performance and scalability (§4.5.3), and how replicating search layer affects cross-NUMA traffic and memory consumption (§4.5.3), and how well Hydralist scales according to the key sizes (Figure 4.5.3).

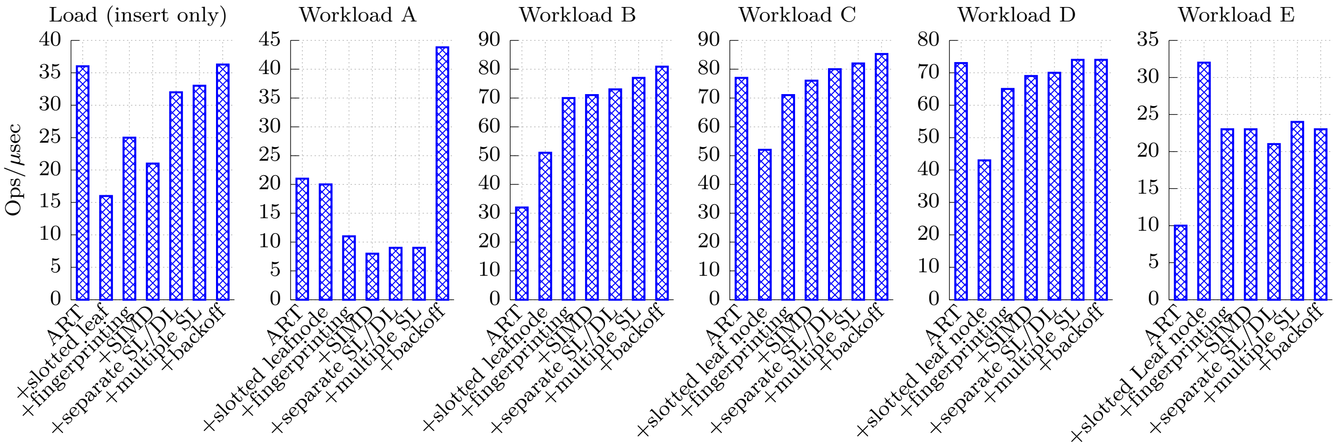


Figure 4.15: Factor analysis of YCSB Workload for 112 threads with string keys.

Factor Analysis

To better understand the performance, we incrementally added features to Hydralist starting with ART and benchmarked each increment using YCSB workloads. The throughput of 112 threads for string keys are summarized in Figure 4.15.

+slotted leaf node We added multiple key-value slots to the leaf node of ART. Keys are stored in a sorted fashion. Binary search is used to search within a node. Adding slotted leaf node to ART improves the scan performance by 3.2×. However, the performance in insert-only workload reduces by 2.4× as inserting into a sorted list requires multiple string comparison and key shifting making it expensive. This feature reduces the performance of workload C and D as binary search is not cache efficient.

+fingerprinting Instead of maintaining sorted keys in slotted leaf nodes, we use 1-byte fingerprint to search the key in a node. This improves throughput of insert-only, B, C, and D workloads as it reduces the overhead of string comparisons. However, it reduces the performance of scan operation because it adds overhead of sorting array before scanning a node.

+SIMD Using SIMD instructions in comparing fingerprints improves performance in workload B, C, and D as it makes fingerprint comparison faster.

+separate SL/DL Separating search and data layer and updating search layer asynchronously improves the performance in insert-only case by 52% with no/negligible impact on performance with other workloads. The improvement in performance can be attributed to reduced size of critical section due to removal of updates to search layer from the critical path.

+multiple SL Adding search layer for each NUMA node improves performance further by about 9% in case of insert-only work load. It also improves performance in Workload B, C, and D by 5%, 2.5%, and 5.75% respectively.

+backoff A thread is forced to retry when it fails to acquire a write lock on a data node. This causes high contention and leads to performance collapse. We added backoff feature wherein if a thread has retried a certain number of times it will wait before attempting again. Adding backoff improved performance by 4.5× in Workload A which is a high contention workload.

Impact of Search Layer Replication

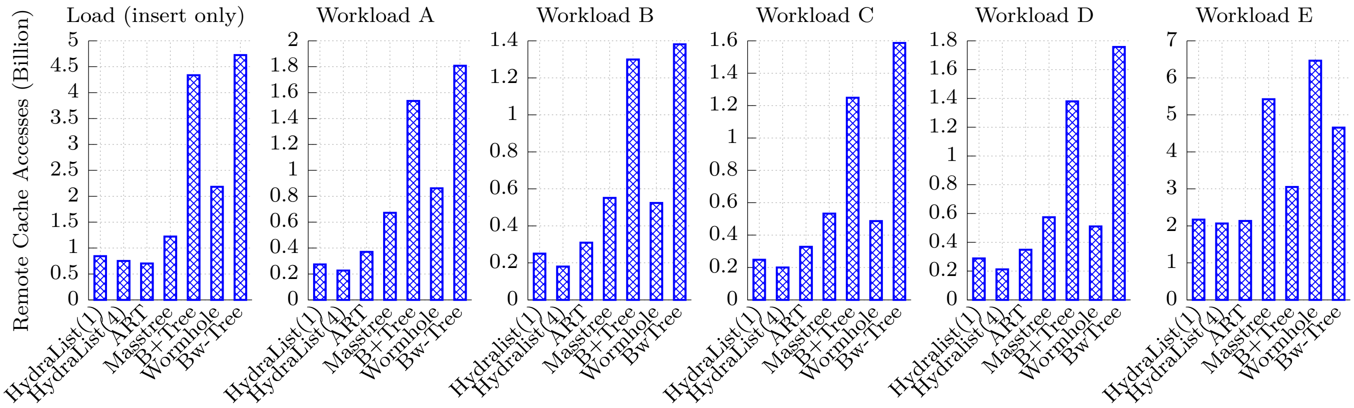


Figure 4.16: Comparison of remote memory access. Hydralist (n), where n is the number of search layer.

Cross NUMA Traffic To measure the remote memory access, we measured the total number of remote cache lines accessed using Intel's Performance Counter Monitor for 50 million transactions of YCSB workload with string keys. As Figure 4.16 shows, Hydralist has the least number of remote memory accesses among the measured indexes for all the workloads. Remote cache line accesses reduce when Hydralist uses four search layers because lookup in search layer does not access remote memory. B+ tree and BwTree have large number of remote cache line accesses for Workload A and C as their search operation involves accessing more memory locations. However, in case of Workload E, their remote cache line accesses is lower than Masstree because of slotted leaf node structure which leads to smaller number of cache misses. NUMA traffic in Workload E is high in case of Wormhole because of reader-writer locking.

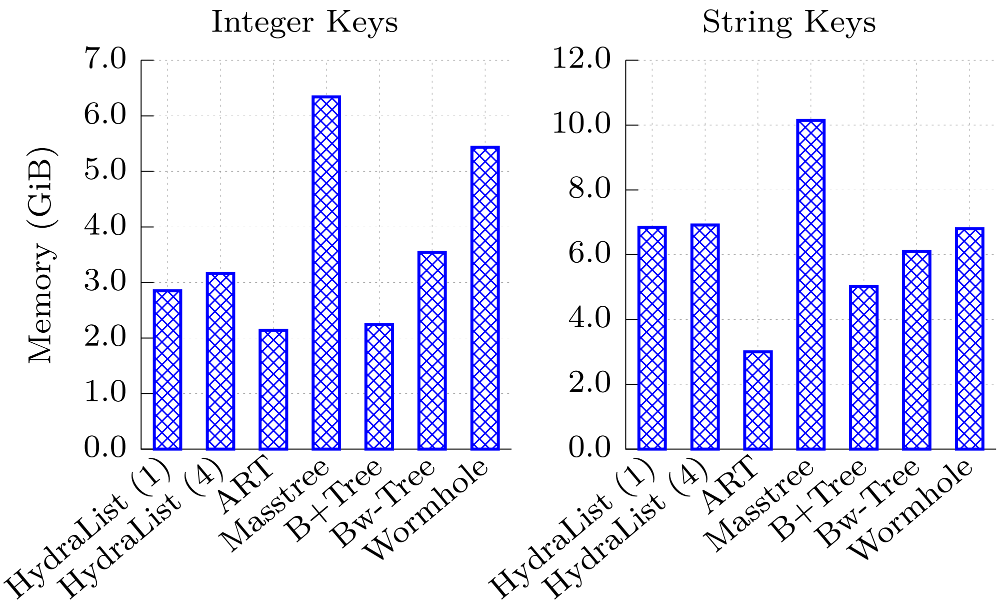


Figure 4.17: Comparison of memory consumption to store 100 million integer keys or 89 million string keys.

Memory Consumption We measured the peak memory consumption of different indexes after inserting 100 million integer keys and 89 million string keys (see Figure 4.17). We measured the resident set size (RSS) by reading /proc/[pid]/statm in Linux. For this experiment, huge pages were disabled to reduce overhead of fragmentation. ART is the most memory efficient memory among all the indexes because of path compression and adaptive sizing of nodes. The memory overhead of replicating search layer is 10.8% and 1% for integer and string key which is highly efficient compared to NR[15] where the memory overhead will be $N\times$ where N in number of NUMA nodes. Data node in Hydralist is larger than leaf node of B+ tree because it stores fingerprint, bitmap and permutation array. This leads to higher memory consumption.

Impact of Key Set Size

To understand the performance of index structures with increasing key set size, we measured the throughput of indexes for Workload B with string keys after adding 100K, 1M and 89M keys. Increasing key set size from 100K to 89M reduces the throughput of B+Tree by 5×. In case of Hydralist and ARTROWEX, the reduction of throughput is only 2×. Masstree shows the least amount of variation in throughput with increasing size but the performance of Masstree for small key set is less.

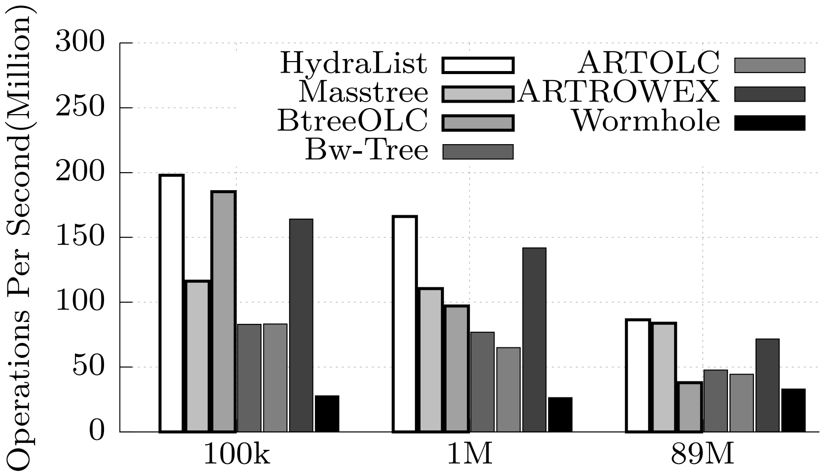


Figure 4.18: Performance comparison of indexes with varying key set size with 112-thread and string keys.

Chapter 5

Conclusions

This thesis showed how asynchronous work can be leverage to achieved scalability in multicore environment. Based on this idea, the thesis introduces two systems. First, a synchronization mechanism, MV-RLU, that extends RLU using mutli-versioning, while preserving the benefits of RLU like multi-pointer update, a simple and intuitive programming model and good performance for read-mostly workloads. The key contribution of the thesis is the design of a concurrent and autonomous garbage collection algorithm for MV-RLU which removes GC from critical path. As a result, MV-RLU outperforms other synchronization mechanisms in several workloads and shows unmatched scalability even in write-intensive workloads. Second, the thesis introduces a new highly scalable and performant index-structure, Hydralist. The key idea behind Hydralist is that an index can be divided into two component, search layer and data layer. In Hydralist, these two layers are decoupled and updates to data layer is done synchronously while updates to search layer is propagated asynchronously using background threads. Moreover, this design allows replication of search layer across NUMA node which further improves performance. We evaluated Hydralist against state-of-the-art index structures with a variety of real world workloads and found HydraList performance to be up to 6.4× higher than next best index for certain workloads.

Bibliography

[1] ART Synchronized. https://github.com/flode/ARTSynchronized.

[2] fio: Flexible i/o tester. https://github.com/axboe/fio.

[3] Kyoto Cabinet: a straightforward implementation of DBM. http://fallabs.com/kyotocabinet/.

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

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

[6] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 266–277, Edinburg, Scotland, September 1999.

[7] Maya Arbel and Adam Morrison. Predicate rcu: An rcu for scalable concurrent updates. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 21–30, San Francisco, CA, February 2015. ACM. ISBN 978-1-4503-3205-7.

[8] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1097-0. doi: 10.1145/2254756.2254766. URL http://doi.acm.org/10.1145/2254756.2254766.

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

[10] Rudolf Bayer and Mario Schkolnick. Concurrency of operations on b-trees. *Acta Informatica*, 9:1–21, 1977.

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

[12] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2012.

[13] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A Practical Concurrent Binary Search Tree. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 257–268, Banglore, India, January 2010.

[14] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable Isolation for Snapshot Databases. *ACM Trans. Database Syst.*, 34(4):20:1–20:42, December 2009. ISSN 0362-5915.

[15] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 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)*, pages 207–221, Xi'an, China, April 2017. ACM. ISBN 978-1-4503-4465-4.

[16] Sang K. Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, pages 181–190, Roma, Italy, September 2000.

[17] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 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, March 2012.

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

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

[20] Douglas Comer. Ubiquitous b-tree. *ACM Computer Surey.*, 11(2):121–137, June 1979. ISSN 0360-0300.

[21] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, pages 143–154, Indianapolis, Indiana, USA, June 2010. ACM. ISBN 978-1-4503-0036-0.

[22] Jonathan Corbet. AutoNUMA: the other approach to NUMA scheduling, 2012. https://lwn.net/Articles/488709/.

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

[24] Tyler Crain, Vincent Gramoli, and Michel Raynal. No Hot Spot Non-blocking Skip List. In *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS)*, pages 196–205, 2013.

[25] Henry Daly, Ahmed Hassan, Michael F. Spear, and Roberto Palmieri. NUMASK: High Performance Scalable Skip List for NUMA. In *Proceedings of the 32nd International Conference on Distributed Computing (DISC)*, pages 18:1–18:19, New Orleans, USA, October 2018.

[26] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 33–48, Farmington, PA, November 2013.

[27] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 631–644, Istanbul, Turkey, March 2015.

[28] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD/PODS Conference*, pages 1243–1254, New York, USA, June 2013. ACM. ISBN 978-1-4503-2037-5.

[29] David Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. Adaptive integration of hardware and software lock elision techniques. In *Proceedings of the ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 188–197, Prague, Czech Republic, January 2015. ACM. ISBN 978-1-4503-2821-0.

[30] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching Transactional Memory. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 155–165, Dublin, Ireland, June 2009. ACM. ISBN 978-1-60558-392-1.

[31] Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guer raoui. Why STM Can Be More Than a Research Toy. *ACM Communication*, pages 70–77, 2011. ISSN 0001-0782.

[32] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 371–384, Lombard, IL, April 2013.

[33] fonxat. 300 Million Email Database, 2018. https://archive.org/details/300MillionEmailDatabase.

[34] Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004. No. UCAM-CL-TR-579.

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

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

[37] Timothy L. Harris. A Pragmatic Implementation of Non-blocking Linked-lists. In *Proceedings of the 20th International Conference on Distributed Computing (DISC)*, pages 300–314, University of Lisboa, Portugal, October 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45414-4.

[38] Thomas E Hart, Paul E McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, 2007.

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

[40] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch Hashing. In *Proceedings of the 22th International Conference on Distributed Computing (DISC)*, pages 350–364, Arcachon, France, September 2008.

[41] Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Type-aware Transactions for Faster Concurrent Code. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, pages 31:1–31:16, London, UK, April 2016. ACM. ISBN 978-1-4503-4240-7.

[42] Intel. Intel AVX-512 Instructions, 2017. https://software.intel.com/en-us/articles/intel-avx-512-instructions.

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

[44] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. A Scalable Ordering Primitive for Multicore Machines. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*, pages 34:1–34:15, Porto, Portugal, April 2018. ACM. ISBN 978-1-4503-5584-1.

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

[46] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 468–479, Davis, CA, USA, December 2013.

[47] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE)*, pages 38–49, Brisbane, Australia, April 2013.

[48] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The ART of Practical Synchronization. In *Proceedings of the International Workshop on Data Management on New Hardware*, pages 3:1–3:8, San Fransico, USA, June 2016.

[49] Viktor Leis, Michael Haubenschild, and Thomas Neumann. Optimistic lock coupling: A scalable and efficient general-purpose synchronization method. *IEEE Data Eng. Bull.*, 42(1):73–84, 2019. URL http://sites.computer.org/debull/A19mar/p73.pdf.

[50] Justin Levandoski, David Lomet, and Sudipta Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE)*, pages 302–313, Brisbane, Australia, April 2013.

[51] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, pages 27:1–27:14, Amsterdam, The Netherlands, April 2014.

[52] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 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)*, pages 429–444, Seattle, WA, March 2014.

[53] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM SIGMOD/PODS Conference*, pages 21–35, Chicago, Illinois, USA, May 2017. ACM. ISBN 978-1-4503-4197-4.

[54] Witold Litwin. Trie hashing. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 19–29. ACM, 1981.

[55] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys)*, pages 183–196, Bern, Switzerland, April 2012.

[56] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: A Lightweight Synchronization Mechanism for Concurrent Programming. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 168–183, Monterey, CA, October 2015. ACM. ISBN 978-1-4503-3834-9.

[57] Paul E. McKenney. Structured deferral: Synchronization via procrastination. *ACM Queue*, pages 20:20–20:39, 1998. ISSN 1542-7730.

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

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

[60] Maged M. Michael. 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)*, pages 21–30, Monterey, California, August 2002.

[61] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. Understanding Manycore Scalability of File Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, pages 71–85, Denver, CO, June 2016. USENIX Association. ISBN 978-1-931971-30-0.

[62] Donald R Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968.

[63] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2015 ACM SIGMOD/PODS Conference*, San Francisco, CA, USA, June 2016.

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

[65] Jun Rao and Kenneth A. Ross. Making B+- Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD/PODS Conference*, pages 475–486, Dallas, TX, May 2000.

[66] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. Ffwd: Delegation is (Much) Faster Than You Think. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 342–358, Shanghai, China, October 2017. ACM. ISBN 978-1-4503-5085-3.

[67] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. K-ary search on modern processors. In *Proceedings of the International Workshop on Data Management on New Hardware*, pages 52–60, Providence, RI, June 2009.

[68] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. Palm: Parallel architecture-friendly latch-free modifications to b+ trees on many-core processors. In *Proceedings of the 37th International Conference on Very Large Data Bases (VLDB)*, pages 795–806, Seattle, WA, August 2011.

[69] Nir Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, March 2011. ISSN 0001-0782. doi: 10.1145/1897852.1897873. URL http://doi.acm.org/10.1145/1897852.1897873.

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

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

[72] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–32, Farmington, PA, November 2013. ACM. ISBN 978-1-4503-2388-8.

[73] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 ACM SIGMOD/PODS Conference*, pages 473–488, Houston, TX, USA, June 2018.

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

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

[76] Xingbo Wu, Fan Ni, and Song Jiang. Wormhole: A Fast Ordered Index for In-memory Data Management. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys)*, pages 18:1–18:16, Dresden, Germany, April 2019.

[77] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. In *Proceedings of the 40th International Conference on Very Large Data Bases (VLDB)*, pages 209–220, Hangzhou, China, September 2014. VLDB Endowment.

[78] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2015 ACM SIGMOD-/PODS Conference*, pages 1629–1642, San Francisco, CA, USA, June 2016. ACM. ISBN 978-1-4503-3531-7.

[79] Yang Zhan and Donald E. Porter. Versioned Programming: A Simple Technique for Implementing Efficient, Lock-Free, and Composable Data Structures. In *Proceedings of the ACM International Systems and Storage Conference*, pages 11:1–11:12, California, USA, June 2010. ACM. ISBN 978-1-4503-4381-7.