Optimistic Concurrency Control Using Version Locks

Abstract

Traditional lock based pessimistic concurrency control (PCC) techniques are easy to use and are widely available in the standard library (e.g., POSIX). But such PCC based locks (e.g., Mutex) is know for their poor scalability with the increasing number of threads. On the other hand, lock-free programming (OCC) enables better scalability but it is notoriously hard to program even the simple data structures. Writing complex data structures such as B+tree requires some serious engineering effort if not impossible.

To overcome this problem, we propose VERSIONLOCK, an optimistic concurrency control (OCC) technique that focuses on performance, scalability, and simplicity. We designed VERSIONLOCK using C++ and we expose the VERSIONLOCK with a simple set of APIs as found in the C++ standard library and POSIX library. We designed a hashtable with VERSIONLOCK and evaluated it using the YCSB workloads. We compare VERSIONLOCK's performance with the RW lock (PCC) and lock-free hashtable (OCC). Over results show that, VERSIONLOCK outperforms RW lock by up to 40% and it performs on par or better than lock-free while being much simple to use than lock-free.

1 Introduction

The introduction of multi-core machines marked the end of the *free lunch* [6, 14] making concurrent programming indispensable when designing a high-performance system software or applications. The lock based concurrency control has been evolving for the last two decades and it has been widely used in the many critical systems software such as the Linux Kernel, database systems, web servers and applications, cloud and datacenter applications etc.

The concurrency control can be categorized as 1) pessimistic concurrency control (PCC) which proactively acquires lock before modifying the shared memory, and 2) optimistic concurrency control (OCC) which modifies the shared memory and resolves inconsistency by performing conflict resolution. The most commonly available locks such as the Mutex [5], Readers-writer [4] lock are examples of PCC. Two-phase locking (2PL) [15] and lock-free programming [9, 12, 16] are common examples of the OCC. 2PL is most commonly used in the database transactions and not see as a viable candidate for designing low level data structures. On the other hand, lock-free programming are used to design common low-level data structures such as a linked list, hash table, binary search tree etc. Lock-free programming have also been used in the prior research to design more complex data structures such as a B+tree [7].

Overall, PCC are easy to use and reason about the correctness but this comes at a poor scalability and performance with the increasing core count. On the contrary, OCC particularly the lock-free programming enables better scalability but it is notoriously hard to program even the simple data structures. Writing complex data structures such as B+tree requires some serious engineering effort if not impossible. Further, lock-free programming is also prone to ABA problem [1] which it even more harder. We explain more on OCC and PCC in §2.

So the key question is, *can we make locks optimistic (for scalability) while retaining the simplicity of the PCC?*. We propose VERSIONLOCK as answer to this question. VERSIONLOCK is an OCC technique that supports non-blocking reads and exclusive writes. VERSIONLOCK is designed with scalability, performance, and simplicity in mind. We designed VERSIONLOCK using C++ and we expose the VERSIONLOCK with a simple set of APIs as found in the C++ standard library and POSIX library. We designed a hashtable with VERSIONLOCK and evaluated it using the YCSB workloads [11]. We compare VERSIONLOCK's performance with the RW lock (PCC) and lock-free hashtable (OCC). Over results show that, VERSIONLOCK outperforms RW lock by up to 40% and it performs on par or better than lock-free while being much simple to use than lock-free.

2 Background

In this section we explain the different concurrency control mechanisms and their pros and cons. Finally we end this section by explaining the motivation behind the VERSIONLOCK design.

2.1 Pessimistic Concurrency Control

Pessimistic Concurrency Control (PCC) works on the notion that there will be a concurrent access and to avoid that it begins the operation (read/write) by first obtaining an exclusive access to the shared memory and then it proceeds to execute the operation on the shared memory. Locks are the most popular example for PCC. For example, imagine a shared counter (global) which is being concurrent accessed by multiple threads and is protected using a mutex [5]. The threads even before accessing the counter has to first acquire the mutex and only the thread that successfully acquires the mutex lock will access the counter while others keep waiting on the mutex until it is released. Mutex (a.k.a binary semaphore) one of the simplest PCC and there is also more relatively advanced PCC techniques such as the Readers-writer (RW) lock which allows multiple readers to access the shared counter while giving exclusive access the the writers. In this work we will consider RW lock as representative PCC technique.

RW Lock. RW lock supports better concurrency by allowing multiple readers to exist in the critical section. Going by our previous shared counter example, when the counter is protected using RW lock (instead of a mutex) then all threads that just wants to read the counter will be allowed given that there is no on-going write. RW lock similar to mutex gives an exclusive access to the writers i.e., only one writer is allowed to update the counter and the rest of the writer thread would wait for the write lock to be available. Note that if a writer is holding the lock then readers are also blocked until the writer finishes.

In C++ standard library, RW lock is available in the form of shared mutex [5]. In the pthreads [3] library RW lock has *read_lock* and *write_lock* which the threads acquire based on the operations they intend to execute. For instance, the threads that just wants to read the shared counter will acquire *read_lock* while the threads that requires to update the shared counter will acquire *write_lock*. As stated earlier, in RW lock *write_lock and read_lock* has the same precedence, *i.e.*, if the *write_lock* is held by a thread no other thread can either acquire the *write_lock* or the *read_lock*. In contrary, when the *read_lock* is held, the threads that require *read_lock* are allowed to acquire the lock while the threads that requires to acquire the *write_lock* will have to wait for threads currently holding the *read_lock* to release it.

2.2 Optimistic Concurrency Control

Optimistic Concurrency Control (OCC) techniques in general do not wait for locks to be available rather they execute the operations and then check for violations. Upon detecting any violation *i.e.*, concurrent access from other threads usually they fallback to the retry mechanism. Two-phase Locking (2PL) and lock-free concurrency are some of the popular OCC techniques. 2PL technique [15] is mostly used to design database transactions while lock-free programming are used to design common low-level data structures such as a linked list, hash table, binary search tree etc. Lock-free programming have also been used in the prior research to design more complex data structures such as a B+tree. In this work we will consider lock-free programming as the OCC representative as we primarily focus on designing VERSIONLOCK to be used with low-level data structures.

Lock-free Programming. Lock-free programming uses hardware instructions to guarantee atomicity for reads and writes across multiple threads. CPU instructions [2] support primitive atomics (*e.g.*, compare and swap (CAS), fetch and add (FAA) etc) that synchronizes concurrent access at the hardware level. These instructions mostly support 8 bytes updates while some instructions such as CAS support 16 byte operations as well. Such instructions work by using lock-prefix *i.e.*, the CPU issues lock on the cacheline that is being updated and hence prevents that particular cacheline from concurrent reads and writes. Atomic instructions at a very high level can be considered as a mutex implemented at the hardware level.

Although, atomic operations are beneficial they are notoriously hard for programming even the simple low-level data structures. It is extremely hard for even the programmers with a lot of experience to write a correct concurrent lock-free data structure. Further, since there is no notion of locks, ABA problem is highly prevalent in such data structures. So the programmers generally rely on the high-level languages such as Java that internally supports garbage collection. For the lock-free programming to be developed in unsafe languages such as C/C++ requires programmers to define their won garbage collection mechanism. The issue is that designing a lock-free highly efficient garbage collection technique is a hard task. Further, with all these issues it is very hard if not impossible to make complex data structures such as a B+tree or a radix tree entirely lock-free. Even the existing works have just been research prototype or they tend to suffer from high performance overhead than their well optimized lock based counterparts.

3 Design

In this section we explain our design goals followed by the VERSIONLOCK design.

3.1 Design Goals

Avoid read-modify writes for read locks. In the PCC techniques, acquiring lock involves modifying the shared memory metadata, this is a problem especially for readers. Essentially, the read operations becomes read-modify-write operation which incurs high overhead. In VERSIONLOCK, we aim to make the read lock acquire and release operation without any writes to the shared memory.

Avoid cacheline contention for read locks. In the lock-free programming, contention on a single cacheline causes cacheline bouncing which is one of the chief performance and scalability inhibitor for both read and write operations. In VERSIONLOCK, we aim to minimize the effects of cacheline

bouncing by making writes exclusive.

Achieve high scalability. Locks need to be scalable across many CPU cores in order foster wider adoption. We aim to make VERSIONLOCK highly scalable for both read and write operations.

Simple lock APIs and interface. We aim to expose VERSIONLOCK with some simple APIs to achieve better programmability and possibly to avoid significant changes to the existing codebase when adopting VERSIONLOCK.

3.2 VERSIONLOCK Design

Synchronization using counters. VERSIONLOCK has a *64-bit atomic counter* at its core. The counter represents the version of the VERSIONLOCK and it is monotonically increasing. The counter is incremented atomically upon lock acquire and release and it only happens for write lock. In VERSIONLOCK, acquire and release of read lock does involve updating the counter; it incurs just a lookup to the counter. More details on the read and write locks in §3.2.1.

Concurrency model. VERSIONLOCK supports non-blocking reads and exclusive writes. When the counter is even it denotes that the lock is free and the writer acquires the lock by atomically incrementing it. Therefore, if the counter is odd then it means that the lock is currently held by the writer. So other writes spins on the counter until the counter is back to even again. Lock is released by atomically incrementing the counter once more (even-odd-even) and now any writers waiting on the lock would try to acquire it.

For reads, lock acquire involves looking up the current version number (*start_version*) and taking a snapshot of it locally. Then the read operation on the shared memory can be executed. Upon completion, lock release operation verifies if the *start_version* has changed by comparing it with global version counter *i.e.*the current counter value. A change in the version denotes that a write operation has completed since the read started. This may result in the read operation reading a stale or inconsistent value. In order, to avoid this problem, the application has to retry the read operation by calling lock acquire once again. This retry loop is delegated to the user side code *i.e.*the VERSIONLOCK read lock release just returns a bool value to signify if the version number has changed.

3.2.1 VERSIONLOCK APIs

Figure 1 shows the VERSIONLOCK APIs. *write_lock* is used to acquire lock for write operation which internally manages the version counter. If the lock is taken, then the call to *write_lock* keeps spinning until the lock is available. On the other hand, the *try_write_lock* API just checks if the lock is available and acquires if so, upon lock unavailability it simply returns false to the caller. If application needs asynchronous lock acquire operation it can leverage the *try_write_lock* API. The locks acquired using *write_lock* and *try_write_lock* can be released by calling *write_unlock* which internally increments the counter to signify the lock release.

```
1  /* acquires lock, spins until lock is available*/
2  void write_lock();
3  /* if lock is not available returns false immediately*/
4  inline bool try_write_lock();
5  /* unlocks write_lock acquired by write_lock or try_write_lock */
6  void write_unlock();
7
8
9  /* allows the reader and returns the current version number*/
10 uint64_t read_lock();
11
12 /* returns VersionNumber upon success, spins until lock is
13  * available if writer is present*/
14 inline uint64_t read_lock_wait();
15
16 /* returns true if version number matches, application
17  * needs to retry if false is returned*/
18 inline bool read_unlock(uint64_t local_version_number);
```

Figure 1: VERSIONLOCK APIs to be used directly in the application

For read operations, a call to *read_lock* is necessary to obtain the current version number. The *read_lock* API is the non-blocking version *i.e.*, it does check the lock status and it simply returns the current version number even if the lock is held by the writer during the call. On the other hand, *read_no_wait* API yields for the writer, *i.e.*, if a writer a holds the lock during the call, it waits until the lock is released and then acquires the read lock by returning the new version number (the one after the write completion). Applications can use either of two these APIs based on their requirement. Read operations releases the lock by calling *read_unlock* API which internally performs the version matching. Unlike the other APIs, *read_unlock* API takes an argument which is nothing but the local copy of the version number obtained by caller during the lock acquire operation. The *read_unlock* functions simply compares this local copy with the current version number from the counter. It returns true or false based on the comparison operation; true signifies the read operation successful and a false value means that the there was an concurrent write during the read operation and hence this read operation is invalid. Upon receiving false value, application can retry the critical section with calling *read_lock* again.

3.2.2 VERSIONLOCK Core API Implementation

Figure 2 and Figure 3 shows the implementation of VERSIONLOCK's read and write locking operations respectively.

write_lock. Lines 1-26 in Figure 2 shows the write_lock implementation. Line 12-13 retrieves the version number and checks the lock status. Lines 16-22 is the lock acquire logic. Line 19 performs a atomic CAS to switch the version number (incrementing the version by 1). The *retry_lock_acquire* will be executed if the lock is already held or if the CAS fails. This retry loop makes our write_lock implementation a spin_lock.

write_unlock. Lines 28-39 in Figure 2 shows the write_unlock operation. Line 37 releases the lock by atomically incrementing the version number by 1. Lines 33-35

```
1  inline uint64_t get_version_number() {return this->version_number;}
2
3  void version_lock::write_lock() {
4
5          volatile uint64_t version;
6          volatile bool lock_status;
7
8  retry_lock_acquire:
9          /* check if lock is already held
10          * even-- lock is free
11          * odd-- lock is held*/
12         version = get_version_number();
13         lock_status = version % 2;
14
15         /* if lock is free acquire the lock and return*/
16         if (!lock_status) {
17                 /* return if CAS succeeds--
18                  * meaning lock is acquired*/
19                 if (smp_cas(&this->version_number, version,
20                                 version + SWITCH_LOCK_STATUS))
21                         return;
22         }
23
24         goto retry_lock_acquire;
25
26 }
27
28 inline void version_lock::write_unlock() {
29
30         uint64_t version = get_version_number();
31
32         /* RESET version_number to 0 to avoid overflow*/
33         if (unlikely(version + 1 == UINT64_MAX - 1))
34                 smp_cas(&this->version_number,
35                         version, RESET_LOCK);
36
37         smp_faa(&this->version_number, SWITCH_LOCK_STATUS);
38         return;
39 }
```

Figure 2: Code snippet of VERSIONLOCK *write_lock* and *write_unlock* functions.

```
1  inline uint64_t get_version_number() {return this->version_number;}
2
3  inline uint64_t version_lock::read_lock() {
4
5          /* just return the current version number
6           * no need of checking the lock status*/
7          return get_version_number();
8  }
9
10 inline bool version_lock::read_unlock(uint64_t
11                 initial_version_number) {
12
13         uint64_t current_version_number = get_version_number();
14
15         /* if both version numbers are equal means no write
16          * has happened during the read operation.
17          * so read can return successful*/
18         if (initial_version_number == current_version_number)
19                 return true;
20
21         /* if version numbers mismatch means that some writer
22          * has completed before the read operation*/
23         return false;
24 }
```

Figure 3: Code snippet of VERSIONLOCK *read_lock* and *read_unlock* functions.

handles version number overflow *i.e.*, it resets the version number to 0.

read_lock. The read_lock just returns the current version number (Figure 3). It is the applications' responsibility to take store a local copy of this returned version number. Because, this version number needs to be passed to release the read_lock. Passing a wrong version number can result in a livelock situation.

read_unlock. Lines 10-23 in Figure 3 shows the read_unlock code. Lines 17-18 compares the local copy of the version number (passed by the caller) with the current version number. If the comparison succeeds it returns true else false is returned.

3.3 VERSIONLOCK Example Using a HashTable

Figure 4, illustrates VERSIONLOCK usage with a hashtable example, Lines 1-13 illustrates the insert function. Line 6 gets the hash bucket and Line 7 acquires the per-bucket VERSIONLOCK. Line 8 calls the list_insert function which inserts the node in the bucket chain. Line 12 unlocks the per-bucket write_lock.

Lines 16-30, shows the lookup operation. Similar to the insert, Line 24 acquires the per-bucket read_lock and stores the local copy of version number in *version*. After the lookup operation in the Line 25, read_unlock is called on the bucket lock passing the local copy of the version number. If read_lock succeeds read operation ends on line 29, else the read operation enters the retry loop.

4 Implementation

To evaluate the efficacy of our version locks, we compare it against two popular concurrency control schemes : *reader-writer lock* and *lock-free*. We implement three variants of an open chaining hash table, each integrated with one of the aforementioned concurrency control mechanism. We use the same number of hash buckets for each hash table implementation. For fine-grained locking with reader-writer locks, we employ a lock for each bucket of the hash table to achieve scalability. We use the YCSB workloads to evaluate performance of the hash table variants, details are discussed in §5.1. Our implementation is publicly available at https://github.com/madhavakrishnan/version-lock In our repository, we provide the implementation of our hash table variants as well as detailed steps to generate the YCSB workloads.

5 Evaluation

In this work, we compare pessimistic concurrency control (PCC) and optimistic concurrency control (OCC) using a hash table. We implement variants of an open chaining hash table that only differ in the underlying synchronization primitive used. For PCC, we use a reader-writer lock for each bucket of the hash table to allow concurrent threads working on disjoint buckets to proceed in parallel. In case of OCC, we have two variants : a lock-free hash table and another based on version locking. These variants use atomic instructions such as *fetch-*

```
bool vl_hash_table::insert(uint64_t key, uint64_t val) {

        uint64_t bkt;
        bool ret;

        bkt = get_key_hash(key);
        ht[bkt].lock.write_lock();
        ret = ht[bkt]._list->insert(key, val);
        if (!ret) {
         std::cerr <<  " insert falied " << std::endl;
        }
        ht[bkt].lock.write_unlock();
        return true;
}

uint64_t vl_hash_table::lookup(uint64_t key) {

        uint64_t bkt, version, val;
        bool retry;

        bkt = get_key_hash(key);

loop:
        version = ht[bkt].lock.read_lock();
        val = ht[bkt]._list->lookup(key);
        retry = ht[bkt].lock.read_unlock(version);
        if (!retry)
                goto loop;
        return val;
}
```

Figure 4: Code snippet of a VERSIONLOCK enabled hash table insert and lookup functions.

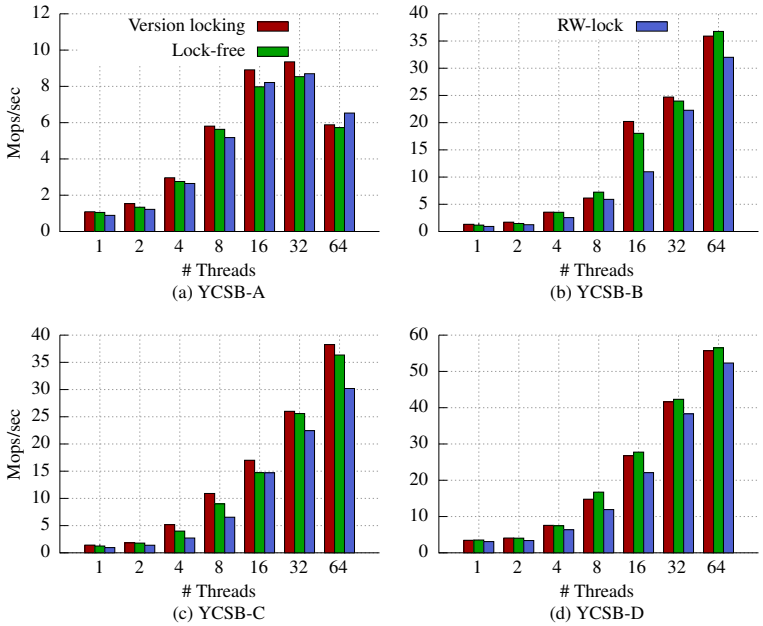

Figure 5: Performance comparison for uniform workloads.

Workloads	Characteristics
YCSB-A	Write-intensive: 50% Writes and 50% Reads
YCSB-B	Read-intensive: 5% Writes and 95% Reads
YCSB-C	Read-only: 100% Reads
YCSB-D	Read-latest: 5% Writes and 95% Reads

Table 1: YCSB workload characteristics.

and-add and *compare-and-swap* for updating items in the hash table while reading an item may involve retries to ensure consistency of the read operation.

Our evaluation is aimed at answering the following questions:

- Can version locking outperform lock-free and a reader-writer lock-based design of the hash table ? Does the performance scale with increasing thread counts ? (§5.2)
- What is the impact of data hotspots on the performance of different hash table variants ? (§5.3)
- What is the performance implication of using read-modify-write primitives for read-only operations, as done in the case of reader-writer locks ? (§5.4)
- How the performance varies when the computation and memory are physically separated across different NUMA domains ? (§5.5)

5.1 Evaluation Setup

We ran our evaluations on a system with Intel(R) Xeon(R) Gold 5218 2.3 GHz CPU that has two NUMA nodes and 16 cores (32 hyperthreads) per node. The system is running Fedora 29 on Linux kernel 4.18.16. We use YCSB [11] – a standard key-value store benchmark for evaluating the hash table variants. YCSB provides various read-write ratios for different workloads, see Table 1. We use index-microbench [17] to generate the YCSB workloads. We use both uniform and Zipfian distribution with random integer keys in our workloads. For all evaluations, we first populate the hash table with 64 million items and then run the workloads, which performs 64 million operations.

5.2 Scalability

In this section, we evaluate the scalability of the different concurrency control mechanisms using uniform workloads wherein each item has an equal probability of being accessed. Figure 5 shows the throughput for the hash table variants in million operations per second (Mops/sec) for the various workloads as we vary the number of threads. For the write-intensive YCSB-A workload (Figure 5(a)), version locking delivers up to 10% higher throughput than both the lock-free and reader-writer lock (rw-lock) for all thread counts except at 64 threads. Better performance than rw-lock is due to no read-modify-writes required for the 50% read operations in this workload. Also, version locking has better performance than lock-free since we use an 8-byte compare-and-swap (CAS) while lock-free uses a 16-byte CAS to complete read operations. However, at 64 threads, rw-lock outperforms both lock-free and version locking approaches. Our evaluation machine has 32 cores (64 hyperthreads) across the two NUMA nodes, so we believe a simple serialization for rw-lock in comparison to retrying as done in version locking and lock-free is the primary reason for this behaviour. Employing expo-

nential backoff to limit retrying can possibly help to limit performance degradation, although we have not investigated it in more detail.

Other workloads are read-intensive and read-mostly, and we see better performance for version locking and lock-free as compared to rw-lock. This is expected since readers still need to acquire and release the shared lock for reading an item from the hash table incurring memory writes in the process. For YCSB-B, the performance gap between version locking and rw-lock is 46% at 16 threads. Upon increasing the number of threads further, the improvement decreases as all threads are utilized in our system. For the read-only YCSB-C workload (Figure 5(c)), version locking achieves higher throughput than other variants for all thread counts including at 64 threads where it outperforms lock-free and rw-lock by 5% and 21%, respectively. The performance trend for YCSB-D is similar to YCSB-B, so we skip discussing it in more detail and exclude it in our later evaluations.

5.3 Performance under skewed workloads

We now evaluate our hash table variants with skewed workloads, generated from a Zipfian distribution (Zipfian coefficient = 0.99). Skewed workloads are common in production environments, as noted in [10, 11]. The results are shown in Figure 6.

For YCSB-A, the performance improvement for version locking over others is smaller (between 4-10%) compared to uniform workloads. For skewed workloads, some key-value pairs are accessed with high probability, so contention is not spread as evenly throughout the hash table. As a result, the probability of retries for both read and write operations for the OCC variants is higher under workload skew. Still, version locking has better performance than lock-free and rw-lock, except at 64 threads where rw-lock achieves the highest throughput.

Under read-mostly and read-only (Figure 6(b),(c)) workloads, OCC variants always outperform rw-lock. Also, version locking has similar or better performance than lock-free design for each configuration.

5.4 Optimism vs Pessimism for read operations

In this section, we want to compare the optimistic and pessimistic approach for read operations and discuss their impact on performance of the application. Specifically, we want to look at performance of read-only operations for both uniform(Figure 5(c)) and Zipfian workloads(Figure 6(c)). In each case, OCC variants comprehensively outperform pessimistic locks as they perform read operations without writing to memory. Hence, threads reading the same or different items do not conflict with each other. In contrast, when using a rw-lock, a shared lock is acquired before initiating the read and upon its completion, the lock is released. As a result, performing a read operation involves read-modify-write primitives that cause cacheline invalidations, resulting in performance degradation. Hence, OCC is better suited for read-intensive workloads. While this project doesn't deal with workload characterization, we believe insights from such studies will help in making informed choices regarding the best suited concurrency control in an application.

5.5 Compute-Memory Separation

In our prior evaluations, we use all hyperthreads in our machine and do not consider the placement of hash table in memory and where the threads accessing the hash table are running. By default, Linux tries to allocate memory on the same NUMA node as the node where the thread is running, so locality of compute and memory is achieved out-of-the-box. Now, we want to compare the performance of the hash table variants when the compute and memory are separated across different NUMA nodes. We pin threads on one NUMA node while the entire hash table resides on another NUMA node and run uniform workloads, similar to §5.2. Our motivation is to study the impact of locality on the performance of the hash table variants and how the trends deviate from our prior results. The results are shown in Figure 7.

For the write-intensive YCSB-A(Figure 7(a)), version locking and lock-free have similar performance across all thread counts and the throughput is only slightly better than rw-lock up to 8 threads. As we increase the number of threads, all variants have similar performance. This is a deviation from our prior evaluation of YCSB-A for both uniform and Zipfian workloads where version locking was able to significantly outperform rw-lock. Accessing remote memory is expensive compared to local memory, thus an optimized concurrency control does not translate into performance boost for the application. The key insight from this evaluation is that the application performance is heavily impacted by the locality of compute and memory, independent of specialization done within the application.

For workloads involving mostly read operations(Figure 7(b),(c)), performance for all hash table variants is lower compared to equivalent settings when remote memory pinning is not done(Figure 5(b),(c)) since accessing remote memory has higher latency than local memory. However, version locking still achieves better performance than other variants due to limited or lack of contention for its operations.

6 Conclusion

In this work, we explored optimistic concurrency control using version locks. Our implementation of version locks within a hash table shows significant performance improvements over other popular thread-safe implementations including a lock-free hash table and another protected using reader-writer locks for both uniform and skewed workloads. We also discuss the impact of compute-memory locality on the performance of an application and show that concurrency control is just one of the factors that determines the overall performance.

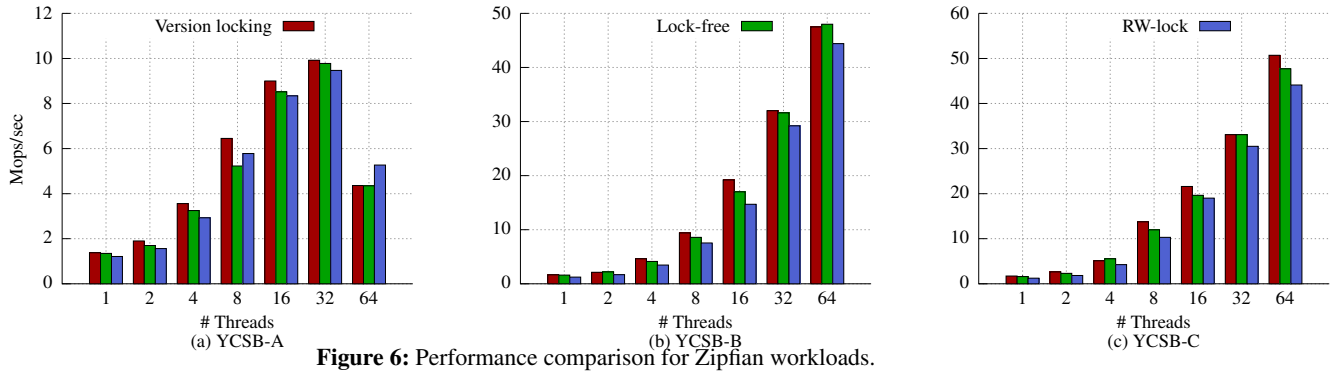

Figure 6: Performance comparison for Zipfian workloads.

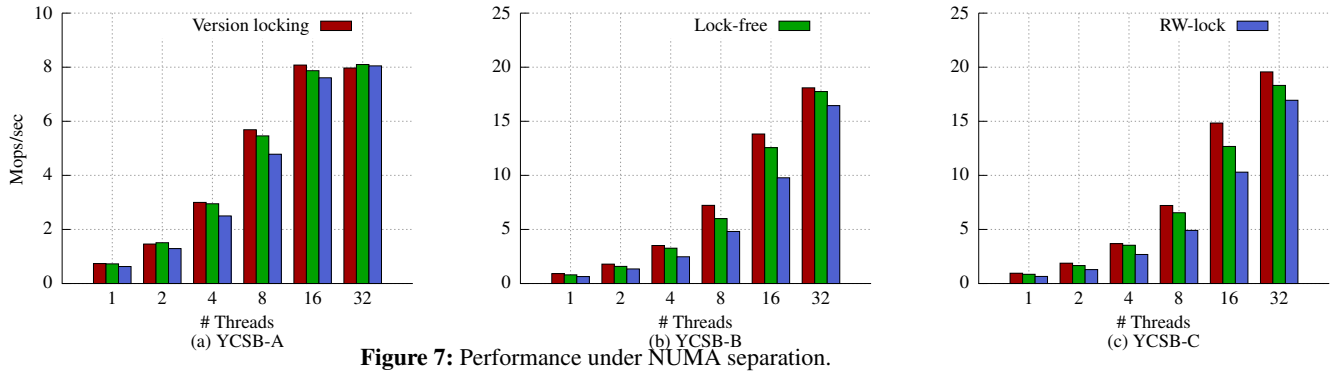

Figure 7: Performance under NUMA separation.

Based on our exploration and insights, we have several areas for future work:

- *Version locking for complex data structures* : we only looked at hash table in our current work but we believe there is an opportunity for optimizing complex data structures like B+-tree, Skiplist. The hierarchical nature of such data structures with complex locking protocols hints at greater impact from utilizing version locking.
- *NUMA-aware synchronization* : our evaluation with separating compute from memory shows diminishing performance gains from version locking. We want to explore NUMA-aware synchronization, both in the context of blocking and non-blocking primitives, similar to prior work [8, 13].
- *Composability* : performance is only one metric that we used in this work, we believe there are other important metrics for adoption and usage of a particular synchronization primitive, for instance, locks are popular since reasoning about their correctness is generally simple. Version locking by its very nature is composable, which enables atomic operations across multiple data structures. Composability is a nice property to implement transactional memory, we want to survey real-world applications to find out more use cases where composability is required and how version locking can be used in such applications.

References

[1] ABA Problem. https://en.wikipedia.org/wiki/ABA_problem.

[2] Atomic operations library. https://en.cppreference.com/w/cpp/atomic.

[3] POSIX (The Portable Operating System Interface). https://www.gnu.org/software/libc/manual/html_node/POSIX.html.

[4] Reader-Writer Locks. https://www.modernescpp.com/index.php/reader-writer-locks.

[5] std::muxtex. https://cplusplus.com/reference/mutex/mutex/.

[6] The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. http://www.gotw.ca/publications/concurrency-ddj.htm.

[7] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. Bztree: A High-performance Latch-free Range Index for Non-volatile Memory. In *Proceedings of the 44th International Conference on Very Large Data Bases (VLDB)*, Rio De Janerio, Brazil, August 2018.

[8] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J Marathe, and Nir Shavit. Numa-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 157–166, 2013.

[9] Bapi Chatterjee, Nhan Nguyen, and Philippas Tsigas. Efficient Lock-Free Binary Search Trees. In *Proceedings of the 33th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, Paris, France, July 2014.

[10] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. Hotring: A hotspot-aware in-memory key-value store. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*, pages 239–252, Santa Clara, CA, February 2020.

[11] 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.

[12] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A Persistent Lock-free Queue for Non-volatile Memory. In *Proceedings of the 21st ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, wien, Austria, March 2018.

[13] Elad Gidron, Idit Keidar, Dmitri Perelman, and Yonathan Perez. Salsa: scalable and low synchronization numa-aware algorithm for producer-consumer pools. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 151–160, 2012.

[14] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. Scalable and practical locking with shuffling. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 586–599, Ontario, Canada, October 2019.

[15] Butler W. Lampson and David B. Lomet. A new presumed commit optimization for two phase commit. VLDB93.

[16] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms.

[17] 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.