Opportunities and Limitations of Hardware Timestamps in Concurrent Data Structures

Olivia Grimes
Lehigh University
Bethlehem, USA
oag221@lehigh.edu

Jacob Nelson-Slivon
Lehigh University
Bethlehem, USA
jjn217@lehigh.edu

Ahmed Hassan
Lehigh University
Bethlehem, USA
ahh319@lehigh.edu

Roberto Palmieri
Lehigh University
Bethlehem, USA
palmieri@lehigh.edu

***Abstract*—Designing high-performance, highly-concurrent linearizable data structures is complex, especially when bulk operations (e.g., range queries) are included. Relying on a single source of synchronization, such as a logical global timestamp, unequivocally eases the design of the synchronization schemes. However, such a design creates a single point of contention, and thus carries performance downsides. As a result, designers often face the dilemma between a simple design and a performance bottleneck. Recently, modern commodity architectures have enabled low-level mechanisms that guarantee that the timestamp registers of all CPUs are synchronized, thus enabling the use of hardware timestamps in data structure designs. Although recent work already exploits this, this work aims at understanding the opportunities and limitations of using hardware timestamps in existing data structure designs. We address this challenge by applying hardware timestamping to three recent state-of-the-art algorithms that use logical timestamps to support range queries in concurrent data structures. Our evaluation shows that the use of hardware timestamps does indeed improve performance compared to the original designs, achieving up to 5.5x improvement. More importantly, by removing the bottleneck of using global logical timestamps in these algorithms, we highlight the design choices that most significantly impact the use of hardware timestamps. Specifically, we show that the mechanism of labeling objects with timestamps plays an important role in maximizing the benefits of leveraging hardware timestamps.**

***Index Terms*—Concurrent Data Structures, Linearizability, Range Query**

I. Introduction

Concurrent data structures are widely used to provide high performance for applications running on multicore machines [1]–[8]. Until recently, they have mostly supported elemental operations, meaning operations that only affect one element in the abstract state of the data structure. A growing number of data structure designs (e.g., [9]–[13]) have been augmented to support the execution of efficient, highly concurrent, linearizable bulk operations, meaning operations that potentially affect multiple elements in the abstract state of the data structure. A range query is a typical implementation of a read-only bulk operation that returns all elements belonging to a given range in the data structure. Supporting efficient bulk operations allows data repositories, such as key-value stores, to enrich the semantics of their operations, resulting in more functionalities offered to applications.

Producing an efficient synchronization strategy for data structures that incorporate both elemental and bulk operations is a very challenging task. Utilizing a global timestamp to synchronize operations is a common (and simple) approach to support linearizable bulk operations on a wide variety of underlying data structures, as demonstrated in literature [9]–[11]. However, this approach comes with a major performance caveat: requiring each operation to access and/or increment a global timestamp produces a single point of contention that can significantly affect the performance of the overall system, especially in large multicore architectures (see our performance evaluation in Section III). This degradation in performance is due to cache-coherence since threads are both reading and writing the logical timestamp, as well as the latency overhead due to NUMA. Other approaches (e.g., [12]) avoid the use of a global timestamp by deploying highly specific synchronization strategies that are difficult to extend to other structures and whose safety is difficult to prove. Transactional memory [14], [15] solves the problem of supporting linearizable bulk operations by relying on the transaction abstraction. However, the generality of transactional memory disallows the exploitation of data structure semantics, which generally results in poor performance.

Since their release in 1993, devices utilizing the Intel Pentium processor have included a per-core timestamp register, allowing access to the CPU's timestamp counter (or simply *TSC*). TSC tracks every cycle that occurs on the CPU, and is thereby a monotonically increasing value. In recent Intel processors, the timestamp counter in each CPU is guaranteed to be synchronized with respect to one another [16]. The `RDTSC` and `RDTSCP` assembly instructions allow programmers to access TSC by loading the timestamp register into the `RDX` and `RAX` registers. By utilizing TSC to synchronize operations, we remove the point of contention caused by threads concurrently trying to access and update the same logical timestamp variable (causing cache-coherence overhead), thus ridding of the major bottleneck that is associated with using a simple, version-based approach to synchronization. Recent literature exploits TSC to improve the performance of their work [5], [13], [17], [18].

This paper aims to analyze the effects of using a hardware timestamp in the context of versioned data structures to understand how design factors impact the usefulness of using a hardware timestamp. Our analysis can help programmers to assess whether their prior work could benefit from the

use of a hardware timestamp, as well as factors to consider when designing a new data structure to most effectively use a hardware timestamp.

In order to conduct our analysis, we provide a simple API allowing programmers to utilize the fast, monotonically increasing timestamp. We focus on three state-of-the-art algorithms meant to enhance data structures by providing linearizable range queries: Bundled References [9], EBR-RQ [10], and vCAS [11]. Each of these data structure designs rely on a logical timestamp to synchronize range queries with the other operations. We replace the logical timestamp in each range query technique with the use of our API.

In our evaluation, we test the three aforementioned techniques on some subset of the following data structures: a lock-free Binary Search Tree [1], a lock-based Citrus Tree [19] (a variant of a Binary Search Tree), and a lock-based Skip List [20]. In the case of augmenting the Binary Search Tree with vCAS, our evaluation reveals that this simple modification to the use of a timestamp produces a speedup of up to 5.5x compared to the original implementation. Our evaluation also reveals cases in which little speedup is observed, for example, when augmenting the Citrus Tree to use EBR-RQ. In either case, the results allow us to analyze how the properties of each algorithm impact the benefits achieved by using the hardware timestamp.

In analyzing our results, we observe that the significance of using TSC relies on the strategy that the algorithm uses to tag an object with a timestamp. We call this step *timestamp labeling* and evaluate under what conditions it enables effectively leveraging hardware timestamps. We find that algorithms that do not require atomically reading the timestamp and labeling the object, such as vCAS and Bundling, benefit greater from hardware timestamps than approaches that do require these steps to be atomic (e.g., EBR-RQ).

A summary of our contributions is as follows:

- Provide an API allowing programmers to utilize the monotonically increasing hardware timestamp, thereby avoiding contention and thus removing the bottleneck produced by using a logical timestamp.
- Evaluate the results of replacing the global timestamp in relevant state-of-the-art algorithms with the hardware's synchronized clock, and determine key performance indicators.
- Outline the ideal cases for using the hardware timestamp, as well as its limitations and design implications.

To the best of our knowledge, this is the first study correlating different data structure designs with performance achievable by the exploitation of hardware timestamps as opposed to logical timestamps.

This paper proceeds as follows: Section 2 elaborates on TSC and describes our simple API. In Section 3, we evaluate improvements in performance achieved by using TSC instead of a logical timestamp. Section 4 provides a detailed analysis of timestamp labeling and its effect on leveraging TSC. Finally, Section 5 discusses related work, and we conclude in Section 6.

II. The Hardware Timestamp

In this section, we describe TSC and the memory ordering guarantees provided by the assembly instructions that access the timestamp register. We then introduce our simple API, written in C++, which allows programmers to utilize the CPU's timestamp counter.

A. TSC

Beginning with the Pentium processor, Intel 64 and IA-32 architectures have included the ability for programmers to access the CPU timestamp counter, called TSC [21]. Other modern processor vendors, namely AMD, Sparc, and ARM, also provide a hardware timestamp counter [22]. For example, ARM provides the Performance Monitors Cycle Count Register, or simply PMCCNTR [23]. Similar to RDTSC/P, PMCCNTR is a register that stores the value of the processor cycle counter, which counts the processor's clock cycles. We focus only on Intel processors as they are most commonly used in the context of data structures [9]–[11], [13]. These processors include per-core timestamp registers which store the value of TSC. In order to use TSC as a timestamping mechanism on a multi-core system, it must be (1) monotonically increasing, and (2) each core must be synced with respect to one another. Until more recently, this second condition was not necessarily met with regard to TSC. Some processor families increment the timestamp counter at a constant rate, meaning all cores are synced with respect to one another. Other processor families, however, capture the number of cycles by incrementing TSC with every internal processor clock cycle. The timestamp counter in these cores may become out of sync since the duration of clock cycles may vary for a variety of reasons, such as energy saving mechanisms and other types of interrupts.

Invariant TSC is a more recent enhancement to the hardware timestamp counter and allows the second condition required for using TSC as a timestamping mechanism to be met. The availability of invariant TSC is indicated by a bit in the CPUID, and ensures that TSC is incremented at a constant rate [21]. TSC is thus kept in sync amongst cores and can be used to synchronize operations. For the remainder of this paper, we assume the use of invariant TSC when referring to TSC.

B. Comparing Assembly Instructions

There are two assembly instructions that load the value of TSC into the timestamp register: `RDTSC` and `RDTSCP`. Each instruction performs the same basic operation but comes with different memory ordering guarantees. On a 64-bit architecture, `RDTSC` and `RDTSCP` load the high-order 32 bits of the timestamp register into `RDX` and the low-order 32 bits into `RAX`. In a 32-bit architecture the timestamp is instead loaded into `EAX` and `EDX` registers respectively. A bitwise `OR` is performed to reconstruct and store the timestamp value into a local variable.

`RDTSC` has no memory ordering guarantees, meaning it is possible that the processor could reorder the instruction

from the order in which it appears in the program. This would clearly create a problem with using a timestamp since the point at which the timestamp is read in program order affects whether the operation occurring is properly serialized or not. To solve this issue, we use the `CPUID` assembly instruction in conjunction with the `RDTSC` instruction. `CPUID` is a serializing instruction that forces the CPU to complete every preceding instruction of the code before fetching and executing the next instruction, thus ensuring that the read of the timestamp occurs at the correct time. A negative to this approach is that the `CPUID` instruction incurs decent overhead as it requires over 200 clock cycles to complete.

`RDTSCP` has a pseudo-serializing property, such that it waits until all preceding instructions have executed before reading the counter, however, it allows for the possibility that subsequent instructions may begin execution before the read operation of TSC is performed. This presents a similar issue to the `RDTSC` instruction of breaking serializability. Since `CPUID` produces substantial overhead and `RDTSCP` has the pseudo-serializing property, we fix this issue by adding an `LFENCE` instruction after the `RDTSCP` instruction. Adding the `LFENCE` instruction guarantees that `RDTSCP` will be executed before any subsequent instructions (including memory accesses), thus solving the issue related to using the `RDTSCP` instruction.

In order to understand the potential performance of using a logical timestamp versus accessing TSC, we compare the throughput of incrementing the logical timestamp using the atomic fetch-and-add operation, with accessing TSC using both `RDTSC` and `RDTSCP` respectively. We test the performance of using `RDTSC` and `RDTSCP` both with and without the memory-ordering guarantees provided by the fences as previously discussed.

Figure 1 shows the results of performing this experiment. We use the same machine as is subsequently described in the Evaluation. This machine has four NUMA zones, each equipped with a total of 24 cores, allowing for 48 hardware threads (using hyperthreading) per NUMA zone, and a total of 192 hyperthreads. Each thread is pinned to a core in order to achieve the best performance. We pin threads by saturating one NUMA zone before beginning to pin threads to the next one. Within a NUMA zone, we pin each pair of hardware threads to their shared physical core consecutively. The same pattern is applied to the remaining three NUMA zones as the threads continue to scale. This policy for pinning threads is optimal for the `Logical TS` approach since hyperthreads on the same core share its L1 cache and are thus able to take better advantage of caching.

Our results indicate that with fences, the `RDTSCP` instruction always outperforms the `RDTSC` instruction due to the overhead incurred by the `CPUID` instruction. While the `LFENCE` and `CPUID` instructions add significant overhead as evident by the performance achieved without fences, both `RDTSC` and `RDTSCP` still vastly outperform the logical timestamp approach, even with their memory-ordering guarantees. Aside from assessing the overhead of `LFENCE` and `CPUID`,

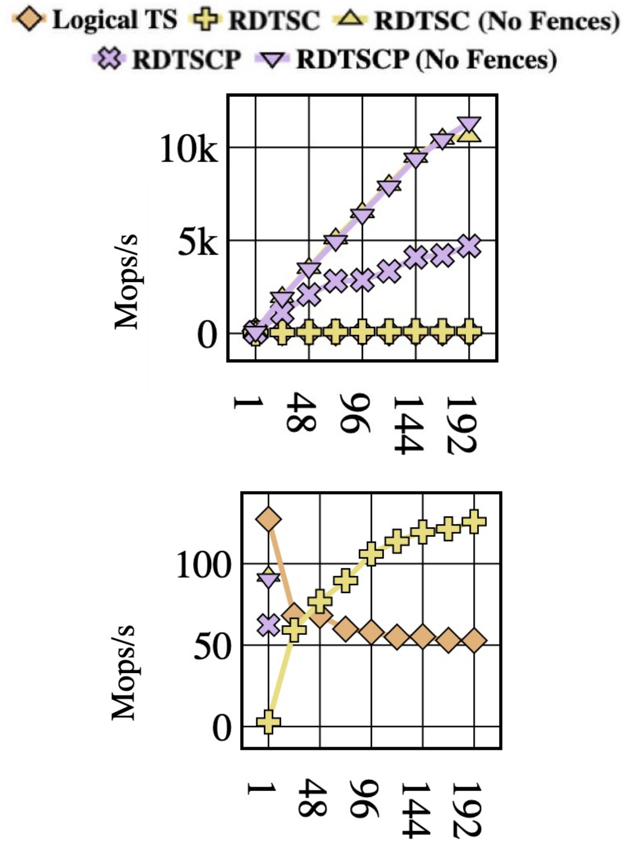

Fig. 1: Throughput of accessing and incrementing a logical integer variable in a loop (denoted `Logical TS`) and accessing TSC with `RDTSC` and `RDTSCP` respectively in a loop. The x-axis represents the number of threads. The figure on the bottom differs from the figure on the top only in that it does not include (most of) the `RDTSCP` data to better highlight the differences between `RDTSC` and `Logical TS`.

we included the results of accessing `RDTSC` and `RDTSCP` without fences because if an algorithm implicitly provides proper synchronization around the timestamp, there may not be a need for any fences. However, since fences ensure safety and consistency, we use them in our evaluation and thus assume the use of fences when referencing the two assembly instructions going forward.

As clear from the top plot in Figure 1, the `RDTSCP` instruction is very fast to access, achieving over 95x more throughput than the Logical TS approach. Additionally, the bottom plot in Figure 1 reveals that the `RDTSC` approach improves over the Logical TS approach by as much as 2.6x with 192 hyperthreads.

It is also worth discussing the single-threaded behavior as seen in the bottom plot in Figure 1. With one thread, the `Logical TS` approach benefits from the positive effect of caching, while the `RDTSC/P` approaches suffer from their respective additional memory-protecting instruction (i.e., `CPUID` and `LFENCE`). However, as the thread count increases, the

inverse relationship applies. The `Logical TS` approach is unable to benefit from caching due to the use of many cores and eventually NUMA zones, while the `RDTSC/P` approaches appear to become less inhibited by the overhead incurred by the memory-protecting instructions since each core has its own hardware timestamp to access.

Clearly, the `RDTSCP` assembly instruction with the use of fences is the fastest and safest option. From this point on we only use the `RDTSCP` assembly instruction with its `LFENCE` serializing instruction.

C. Hardware Timestamp API

Listing 1: Hardware Timestamp API

```
1  timestamp_t get_next_timestamp() {
2    unsigned long long cycles_low,
3        cycles_high;
4    asm volatile (
5        "RDTSCP\n\t"
6        "mov %%rdx, %0\n\t"
7        "mov %%rax, %1\n\t"
8        "LFENCE\n\t": "=r" (cycles_high),
9            "=r" (cycles_low)::
10           "%rax", "%rcx", "%rdx");
11   timestamp_t ts = (((uint64_t)cycles_high
12       << 32) | cycles_low);
13   return ts;
14 }
```

Our simple API is provided in Listing 1. We call the `RDTSCP` instruction which loads the value of TSC into the `RDX` and `RAX` registers as previously described. We then move the value in these registers into variables *cycles_low* and *cycles_high*. The `LFENCE` instruction ensures that we read TSC before executing any further instructions.

We must clobber the `RAX`, `RCX` and `RDX` registers (Line 10) to indicate that they will be overwritten by the inline assembly code. `RCX` must be clobbered in addition to `RAX` and `RDX` because `RDTSCP` additionally loads the CPU ID into `RCX`. Finally, we reconstruct the full timestamp with a bitwise `OR` and return the value of TSC.

For each of the three aforementioned algorithms that we analyze, we replace the use of a logical timestamp with the use of our API. In each case, modifying the code simply requires replacing each call to read or increment the logical timestamp with a call to our API to read the next timestamp from TSC. We remove the global timestamp variable in each implementation and add no additional lines of code. Replacing code with our API is thus a simple programming task and using it for a new project is very straightforward.

III. Evaluation

To evaluate the use of a hardware timestamp, we replace the logical timestamp with our API in three state-of-the-art techniques for supporting linearizable range queries: Bundled References [9], vCAS [11], and EBR-RQ [10]. Before digging deeper into our evaluation, we first describe the characteristics of each algorithm relevant to our analysis. We refer the reader to their respective publications for more details.

Bundled References implements range queries for lock-based linked data structures. A bundle stores a history of links between nodes such that each link is augmented with a timestamp, allowing a range query to traverse a consistent snapshot of the data structure. Bundling increments the logical timestamp during an update operation, and each range query reads it to define its linearizable snapshot. Since Bundling exclusively augments lock-based data structures, its range query operation is also blocking.

vCAS implements range queries for lock-free data structures. vCAS introduces the so called versioned CAS (or vCAS) object, which is meant to replace mutable objects in the structure. Akin to bundling, each vCAS object records the history of updates to it. When applied to lock-free linked data structures, it allows a range query to traverse a consistent snapshot and properly retrieve constituent elements in a non-blocking manner. Unlike Bundling, vCAS increments the logical timestamp at the start of a range query operation and this timestamp is read by update operations to tag any modified vCAS objects.

Finally, EBR-RQ takes advantage of a property of epoch-based-reclamation (EBR) algorithms that make them ideal for implementing range queries. Specifically, EBR temporarily places deleted nodes in what is called a limbo list, and waits to reclaim the nodes until the algorithm is sure that no process needs to access them. EBR-RQ harnesses this property by augmenting nodes with insertion and deletion times, allowing range queries to scan the current state of the data structure followed by the limbo lists. These lists are created by EBR algorithms and used to ensure all nodes in the range at the time the range query is linearized will be captured.

As with vCAS, EBR-RQ increments the logical timestamp at the start of a range query operation. EBR-RQ is implemented using both a lock-free and lock-based approach. In both approaches, reading the timestamp by an update operation must be atomic with assigning it to the modified node. To accomplish this, the lock-based version uses a readers-writer lock, while the lock-free version uses a double-compare-single-swap (DCSS) primitive. This design severely limits the ability of the lock-based version of EBR-RQ to benefit from the use of TSC, as is subsequently analyzed in this section. In the next section, we also discuss how the use of DCSS in the lock-free version prevents it from effectively using TSC.

Although all three techniques address the same problem, their different design choices directly influence the efficacy of porting them to use hardware timestamps. Specifically, we identify three significant algorithmic differences between them:

- The first difference is the progress guarantees that they offer. EBR-RQ and Bundled References are lock-based while vCAS is lock-free.
- Second, they differ in the operation responsible for updating the timestamp (range queries in vCAS and EBR-RQ, and updates in Bundling).
- Third, EBR-RQ adopts more complex handling of the timestamp as we mentioned earlier.

Given the above techniques, we test the impact of hardware timestamps by measuring the performance of various data structures ported to use them. We test each design with some subset of the following data structures: a lock-free Binary Search Tree [1], a lock-based, unbalanced Binary Search Tree called a Citrus Tree [19], and a lock-based Skip List [20]. Due to the progress guarantees provided by each technique, they are not applicable to all data structures. Specifically, vCAS is tested with the lock-free Binary Search Tree. Then, we use the codebase in [9] to test the lock-based Citrus Tree with the three algorithms. Finally, Bundling is tested with the lock-based Skip List.

We applied vCAS and EBR-RQ to the Skip List structure as well, however, since we did not observe performance gains with using TSC, we decided to omit them from the paper. We additionally tested the range query techniques on a lazy-list data structure, however, we did not see any improvement in performance since the bottleneck in such data structures is the time required to traverse it (i.e., its linear complexity), not the timestamp itself. In general, we choose to include experimental results that allow us to analyze the limitations of hardware timestamps in cases where we do see improvement.

A. The Effect of Ties in TSC

As previously noted, TSC is incremented in accordance with every internal processor clock cycle, and thus the TSC values read by threads are monotonically increasing but not necessarily strictly increasing. One corner case that arises from this is when two threads in different CPUs read the same TSC value. In this section, we discuss how this tie may affect some protocols, but not others. It is worth noting that we include this part for completeness since such ties are theoretically possible, even if in practice they are unlikely to happen. Specifically, we discuss tie TSC values in two algorithms that use TSC: EBR-RQ [10] and a related work, Jiffy [13]. These two algorithms represent the two extreme cases in which on the one hand TSC ties are harmless (in EBR-RQ), and on the other hand TSC ties are algorithmically avoided (in Jiffy).

The EBR-RQ (lock-based) algorithm protects the timestamp with a lock L. EBR-RQ already allows for tie timestamp values for update and single-read operations; thus, tie TSC values are also acceptable for these operations. On the other hand, range queries increment the timestamp at the start of their traversal, by acquiring L in exclusive mode, incrementing the timestamp, setting the linearization point, and then releasing L. Thus, when we replace the increment of the timestamp with reading from TSC, it still remains impossible for two range queries to read the TSC value at the same time since this read is executed in a critical section protected by a lock.

Jiffy on the other hand must make algorithmic considerations to ensure that no two *revisions* in their so-called *revision list*, which stores the history of updates to a node, have the same version. This is theoretically not guaranteed since the values of those revisions are assigned from the TSC, which is monotonically but not strictly increasing. Their solution to overcome this includes adding a wait period, which they note is never used in practice due to the clock-cycle resolution by which TSC increments.

To conclude, while this is an important theoretical question, it is rare and unlikely to occur in practice given the extremely fast increment rate of TSC. Importantly, the empirical study we are performing is not impaired by this discussion.

B. Experimental Setup

All of our code is written in C++ and compiled with `-std=c++11 -O3 -mcx16`. We run our tests on a machine running Ubuntu 20.04, with four Intel Xeon Platinum 8160 processors containing a total of 192 hyper-threaded cores split between four non-uniform memory access (NUMA) zones.

In the following experiments, threads execute a given mixed workload consisting of range queries, updates, and contains operations on keys that are randomly generated with a uniform distribution. Workloads are represented as U-RQ-C such that U is the total percentage of update operations, RQ is the percentage of range query operations, and C is the percentage of contains operations. In each of our experiments, range queries are 100 keys long such that the starting point is randomly generated from the key range. Each of the three data structures evaluated is pre-populated with elements having a key range of 1,000,000. In each experiment, the given structure is initialized with half of the elements in the key range. An equal amount of deletions and insertions are attempted to stabilize the size of the structure. Each data point is the result of averaging five trials performed for three seconds each. The results obtained in our experiments are consistent across trials (i.e., the average coefficient variation across all trials is 1.6%). Our code is public and available on Github.[1]

C. Results

We begin our evaluation by discussing the resulting performance of incorporating hardware timestamps into vCAS for the lock-free binary search tree, the results of which are shown in Figure 2. The most significant source of speedup achieved across all of our experiments is seen with these plots, which highlight the potential for hardware timestamps to significantly improve operation throughput. Figures 2a - 2d and 2i, and Figures 2e - 2h represent a fixed range query percentage of 10% and 20% respectively with updates increasing left to right. Figure 2a represents the outcome of a read-only workload with 10% range query operations. Utilizing TSC allows the throughput on 192 hyper-threaded cores to grow to nearly 200 Mops/s compared to about 67 Mops/s with the use of a logical timestamp, producing nearly 3x speedup. Figure 2e shows the results of another read-only workload, this time with 20% range query operations. In this case, the throughput of utilizing TSC is about 221 Mops/s, while the use of a logical timestamp only achieves about 40 Mops/s, thus producing a speedup of over 5.5x with 192 hyper-threaded cores. Recall that vCAS increments its timestamp for each range query operation, explaining the increase in speedup as the percentage of range query operations increases.

[1]https://github.com/sss-lehigh/rdtsc-versioning

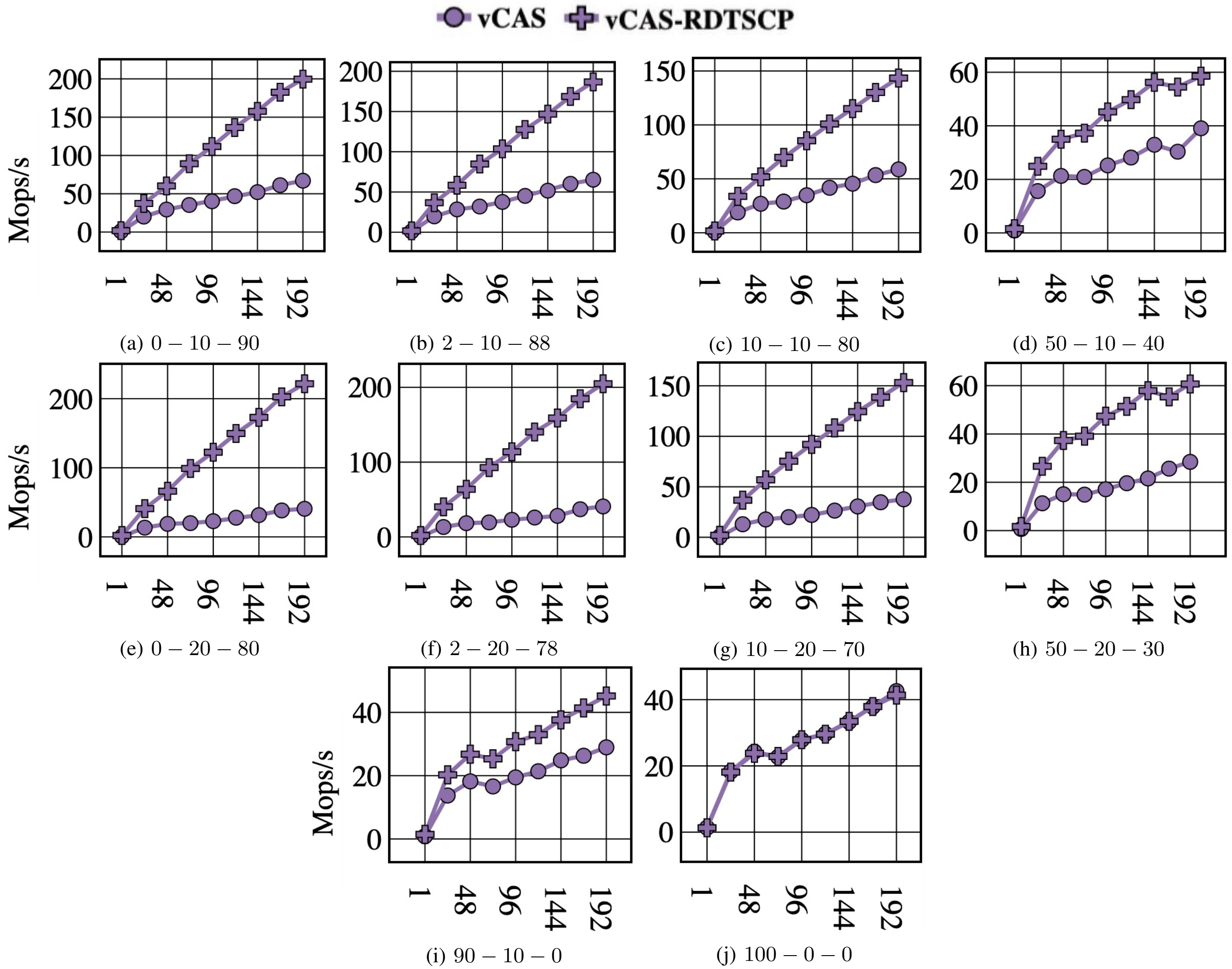

Fig. 2: Throughput of a Binary Search Tree ported to use vCAS using both a logical timestamp approach (vCAS) and a hardware timestamp approach (vCAS-RDTSCP) while varying the number of threads.

As the percentage of update operations increases through Figures 2b - 2d and 2i, and Figures 2f - 2h, the speedup of utilizing TSC compared to using a logical timestamp decreases, but nonetheless there is significant speedup in each case, ranging from 1.6-5x improvement in throughput with 192 hyper-threaded cores. The throughput of the update-only workload (Figure 2j) is the same for both methods of timestamping (using a logical timestamp and using TSC), which is expected since it is the responsibility of range query operations to increment the timestamp in vCAS.

We now move our attention to evaluating the Citrus Tree data structure. Figure 3 contains results for augmenting the lock-based Citrus Tree data structure with vCAS and Bundling. Figure 3a shows the throughput for a read-only workload with a mix of range query operations and contains operations. Since Bundled References increments the timestamp during update operations, there is no difference in throughput between each timestamping approach in Figure 3a, as expected. vCAS on the other hand experiences speedup for such a read-only workload, though notably less than it achieves for the same workload on the lock-free binary search tree (Figure 2a). The rest of the plots in Figure 3 show at least some speedup in all cases.

Figure 4 includes the results of evaluating EBR-RQ with the Citrus Tree data structure. Speedup is achieved by using TSC rather than a logical timestamp in Figures 4a and 4b, in which the range query rate is 10%, and the update rate is 2% and 10% respectively. For each of these workloads, speedup is achieved when saturating all non hyper-threaded cores in the first NUMA zone (i.e., using no greater than 24 threads) and then takes a significant drop. The remainder of the plots experience a similar trend wherein there is a spike through the first NUMA zone up until the use of hyper-threads, and then the throughput drops from there. Additionally, this

vCAS vCAS-RDTSCP Bundle Bundle-RDTSCP

(a) $0 - 10 - 90$ (b) $2 - 10 - 88$ (c) $10 - 10 - 80$ (d) $50 - 10 - 40$

(e) $0 - 20 - 80$ (f) $2 - 20 - 78$ (g) $10 - 20 - 70$ (h) $50 - 20 - 30$

(i) $90 - 10 - 0$ (j) $100 - 0 - 0$

Fig. 3: Throughput of a Citrus Tree ported to use both vCAS and Bundling. Each range query technique uses both a logical timestamp approach (vCAS and Bundle) and a hardware timestamp approach (vCAS-RDTSCP and Bundle-RDTSCP). The x axis represents threads.

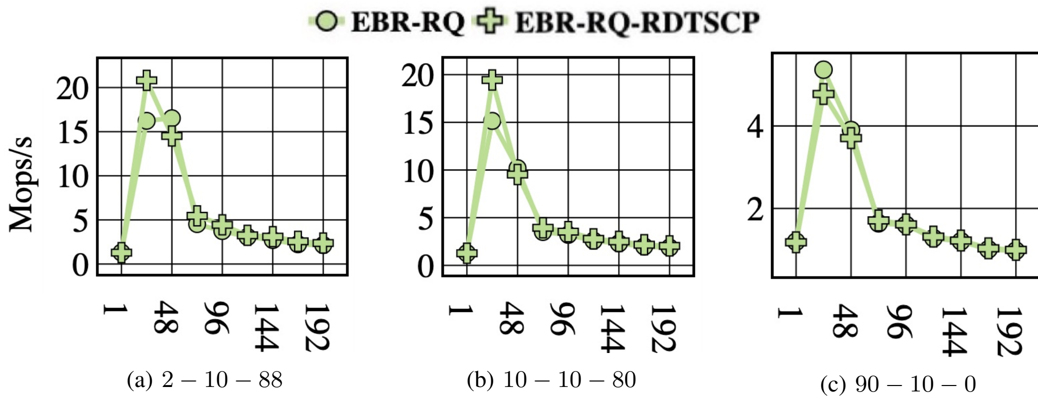

Fig. 4: Throughput of a Citrus Tree ported to use EBR-RQ using both a logical timestamp approach (EBR-RQ) and a hardware timestamp approach (EBR-RQ-RDTSCP). The x axis represents threads.

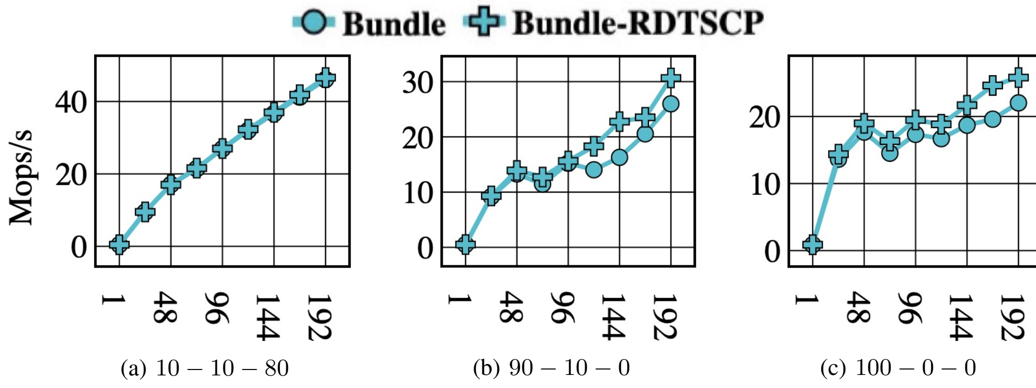

Fig. 5: Throughput of a Skip List ported to use Bundled References using both a logical timestamp approach (Bundle) and a hardware timestamp approach (Bundle-RDTSCP). The x axis represents threads.

is the only figure wherein the use of TSC performs worse than the use of a logical timestamp, though only a few data points reflect this behavior and the performance of TSC is only slightly worse. These behaviors are due to the coarse-grained locking nature of EBR-RQ in conjunction with the lock-based approach taken by the Citrus Tree. Further analysis of the algorithmic implications of this behavior is provided in Section IV.

Finally, Bundling is evaluated using the lock-based Skip List. The results are shown in Figure 5. While the TSC method of timestamping is never worse than the logical timestamp approach, it only shows speedup for update-heavy workloads (Figures 5b and 5c). Due to the fact that Bundled References updates its timestamp during update operations, we see speedup in this workload configuration since with the use of TSC, performing an update no longer requires atomically incrementing the timestamp. The lack of performance improvement for read-heavy workloads indicates that a bottleneck in the Skip List data structure itself outweighs the bottleneck associated with the use of a logical timestamp in Bundled References.

IV. Consequences of Timestamp Labeling

As demonstrated by our results, using TSC to synchronize operations presents an opportunity to take advantage of the simplicity and general purpose nature of a timestamp-based approach, while avoiding the performance bottleneck that results from using a logical timestamp. However, this may not always apply. For example, the implications of a clever algorithmic design of some component of a data structure may have been hidden previously due to the use of a logical timestamp. Conversely, an algorithm may be designed in such a way that it is unable to realize any performance gains even when utilizing TSC. Eliminating the timestamp bottleneck allows us to focus on how the algorithmic choices made by a programmer affect a data structure's ability to scale and generally achieve high throughput. To take full advantage of the speed provided by using TSC, it is important to understand how design choices impact the benefits of using it. Before analyzing the correlation between the granularity of each approach and the performance achieved by utilizing TSC, we first introduce our notion of timestamp labeling and overview the consequences of design choices related to it.

Timestamp Labeling. Timestamps are helpful constructs for concurrent algorithms because they encapsulate the order of events in a globally accessible variable. In order to record the history of steps, an algorithm necessarily must label objects with a timestamp to convey some information related to the moment that a particular event occurred relating to that object (e.g., insertion and deletion timestamps in EBR-RQ). We denote this action *timestamp labeling* and observe that the behavior of an algorithm during this step directly influences the algorithm's ability to harness hardware timestamps.

Consider the three algorithms we evaluate. Each has a unique treatment of timestamp labeling. For instance, EBR-RQ requires that the read of the timestamp and labeling of a data structure node occur in the same logical instant, leading to a global lock that protects the timestamp. Alternatively, Bundled References requires that the timestamp labeling step occurs atomically with the original structural modifications in order to capture the total order of updates. Compared to EBR-RQ, this has a finer granularity since it only entails holding the locks required by the original data structure operation. Finally, vCAS takes an approach whereby threads help each other to perform timestamp labeling, which also represents the linearization point of the update that is labeling the versioned CAS object. When applying vCAS to traversal data structures, this step corresponds to the linearization point of a given operation (e.g., setting a child pointer). Through helping, timestamp labeling is not the responsibility of a single thread but can be delegated to the first operation that is able to perform the step.

The above algorithmic characteristics directly correlate to the performance observed in Section III. The global locking technique of EBR-RQ causes the introduction of hardware timestamps to be ineffectual since contention on the lock is the primary bottleneck, not the timestamp itself. On the other hand, Bundled References and vCAS have a much more fine-

grained timestamp labeling policy that allows them to benefit from hardware timestamps by reducing the negative effects (i.e., coherence traffic) of simultaneous accesses to a logical timestamp. In the remainder of this section, we investigate the consequence of these behaviors in more detail.

vCAS and Bundling. Recall that Figure 3 contains results for augmenting a Citrus Tree to use vCAS and Bundling respectively. The fine-grained timestamp labeling nature of both vCAS and Bundled References lends itself to taking advantage of hardware timestamps since alleviating the bottleneck associated with accessing a logical timestamp is not overshadowed by other algorithmic behaviors. Removing the timestamp bottleneck in vCAS allows it to achieve performance comparable to Bundling in the cases where Bundling previously outperformed vCAS, revealing that the logical timestamp is a critical bottleneck in vCAS.

This case exemplifies that by leveraging the hardware timestamp, we are able to truly understand the limitations of the data structures as it directly relates to the design of algorithms, rather than attributing overhead to the bottleneck of a logical timestamp. For example, the read-intensive workloads in Figure 3 demonstrate that vCAS obtains superior speedup when using TSC relative to Bundling since the logical timestamp is incremented by range queries, whereas Bundling simply reads it. Furthermore, the update-heavy workloads reveal that the design of vCAS is better suited for such workloads compared to Bundling. Since contention on the logical timestamp is removed from Bundling, we know that other characteristics of the algorithm are the reason for its worse performance. With this knowledge, the update methods of each approach can be analyzed to determine design factors that allow vCAS to outperform Bundling, without concern for the bottleneck produced by a logical timestamp. In this case, the larger speedup achieved by vCAS for update-heavy workloads is likely due to their use of helping for update operations.

Comparing the results of Figure 2 and Figure 3 clearly indicates that for an augmentation approach like vCAS, using TSC with a lock-free data structure vastly outperforms the use of TSC on a lock-based structure, as it is able to take full advantage of the fine-grained timestamp labeling.

Lock-based EBR-RQ. As previously discussed, EBR-RQ implements its range query operation with a global readers-writer lock to protect critical regions of code associated with accessing and/or incrementing the timestamp, resulting in coarse-grained timestamp labeling compared to vCAS and Bundling. Specifically, the lock is acquired in exclusive mode when the timestamp is incremented at the beginning of a range query operation and is acquired in shared mode when performing an update to (atomically) read the timestamp and set the linearization point value at the proper address. This is why, unlike vCAS and Bundling, EBR-RQ does not solely rely on using an atomic variable to store the logical timestamp to synchronize updates and range queries. As a result, in EBR-RQ we must retain the readers-writer lock when accessing TSC. While with vCAS and Bundling we were able to see performance improvements when eliminating the bottleneck associated with using a logical timestamp, this reliance on using locks essentially creates the same bottleneck as if we were still using a logical timestamp. Thus, there is little improvement when using TSC over a logical timestamp for the lock-based EBR-RQ implementation.

Lock-free EBR-RQ. Thus far we have only focused on the lock-based EBR-RQ implementation, as the lock-free implementation prevents the use of TSC altogether. Specifically, the critical section in the update method is replaced with a lock-free approach, namely by leveraging the lock-free double-compare single-swap (DCSS) atomic primitive provided by Harris et al. [24]. DCSS takes five arguments: two addresses to read from, two corresponding expected values, and one new value. It swaps the new value with the second address only if the two expected values match the value stored at their respective addresses, all atomically. DCSS allows the update method to succeed only if the timestamp contains a certain value. More specifically, an update first reads the timestamp and is only (potentially) successful if the timestamp has not changed when later performing the DCSS.

Using TSC in place of a logical timestamp necessitates that there is no algorithmic reliance on using the address of the timestamp, as it is non-existent when using TSC. This case arises when the algorithm uses a mechanism to ensure lock-freedom that requires validating that the value of the timestamp does not change over time, thus relying on checking the value at some address. The lock-free EBR-RQ approach requires access to the address of the timestamp in order to perform the DCSS, which creates an algorithmic dependence on the logical timestamp. This dependency is inherent in the proposed labeling solution since reading the timestamp must be atomic along with the update to a variable, all without using locks. As a result of this requirement, the logical timestamp cannot be eliminated.

A. General Takeaways

Analyzing the algorithmic design of Bundling, vCAS and EBR-RQ provides us with many lessons indicating how to optimally utilize TSC as a timestamping mechanism. These lessons can be summarized as follows:

- The more granular the approach to timestamp labeling, the more speed-up will be achieved when utilizing TSC to synchronize bulk operations.
- Lock-free data structures allow for the most advantageous speedup when utilizing TSC as a timestamping mechanism.
- To ensure maximal performance benefits from the use of TSC on a versioned data structure, a programmer should ensure that their algorithmic design does not rely on a dependency that places the timestamp in a critical section or validates that its value did not change.
- Using TSC instead of a logical timestamp on a structure that itself is lock-free, and is augmented using a lock-free (or more generally, non-blocking) approach to support bulk operations enables achieving higher performance

than current state-of-the-art solutions, with half of the processing power (i.e., half the amount of cores).

V. Related Work

TSC has been used in past literature for synchronization purposes, however no work that we are aware of analyzes how design factors of algorithms impact the ability of TSC to maximize performance.

Jiffy [13] is a lock-free, linearizable, ordered key-value store that offers both: (1) batch updates, i.e., some set of put and remove operations that are executed atomically, and (2) consistent snapshots which are used by range queries. The relevance of Jiffy is that it uses the CPU's timestamp counter to synchronize operations, as is done in this paper to three other algorithms that allow support for linearizable range queries in various data structures. Jiffy, however, is specifically built as a lock-free skip list structure, whereas we evaluate techniques meant to augment an assortment of data structures to support linearizable range queries.

OpLog [17] is a general-purpose library meant to provide update-heavy data structures with good scalability. It achieves scalability by writing updates to a per-core log and only combining logs when required by a read. To synchronize update and read operations, OpLog relies on utilizing TSC as a timestamping mechanism. While OpLog focuses on supporting scalable, update-heavy workloads in data structures, our work focuses on how to optimally utilize TSC based on the design factors of various data structures for a variety of workloads.

TSC has additionally been used in a concurrent stack implementation [5], a shared memory synchronization mechanism [18], and a serializable, though not linearizable, transactional database engine [25]. In each case, it is used as a timestamping mechanism to synchronize operations and does not analyze TSC as we do in our work.

Looking more generally at the realm of concurrent programming, two pieces of work [22], [26] have analyzed TSC in the contexts of using it as a synchronization primitive, and using it to boost transactional memory, respectively. The first work, ORDO [22], is a scalable ordering primitive for multicore machines that utilize TSC. The ORDO primitive is motivated by the assumption that hardware clocks are inherently skewed and attempts to fix this skew for a variety of underlying architectures (Intel, AMD, Sparc and ARM) to provide a properly synchronized global timestamp. However, processor vendors claim that the hardware timestamp *is* in fact synchronized amongst cores [16], and many others make this assumption in their work [5], [13]–[15], [17]. In our paper, we analyze patterns of highly concurrent data structure designs with the assumption that TSC provides us with a synchronized clock.

The second work aims to utilize the hardware timestamp counter to boost timestamp-based transactional memory [26]. The work focuses on identifying properties of the hardware timestamp which make it suitable in the context of transactional memory. Other work has also used TSC in the context of transactional memory [14], [15]. Our work instead focuses on the use of a hardware timestamp in the context of concurrent data structures that involve linearizable operations with specific data structure semantics, rather than generic transactions.

VI. Conclusion

In this paper, we study the effects of replacing logical timestamps with hardware timestamps in existing algorithms that support linearizable range queries in concurrent data structures. We do so by developing a simple API that can be used as a drop-in replacement for the logical timestamps and port three existing techniques to use our timestamp replacement. Our results demonstrate that the key consideration when deciding whether hardware timestamps will positively influence performance is how objects are labeled with them. In other words, the relationship between reading the timestamp and writing the value to a field in an object is paramount. As our paper demonstrates, this plays a critical role in the efficacy of hardware timestamping. For example, some techniques do not require that these steps happen atomically and are therefore able to benefit from the reduced contention on the timestamp. In contrast, other approaches which rely on an auxiliary synchronization mechanism do require that they happen atomically, thus limiting the ability to take advantage of fast hardware timestamps. Additionally, we identify cases when hardware timestamps cannot replace their logical counterpart. Our work determines that while the use of logical timestamps is a fundamental building block in many algorithms, not all of them are able to capture the benefits of hardware timestamping equally.

Acknowledgment

Authors would like to thank the anonymous reviewers for the constructive comments. This material is based upon work supported by the National Science Foundation under Grant No. CNS-2045976.

References

[1] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, "Non-blocking binary search trees," ser. PODC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 131–140. [Online]. Available: https://doi.org/10.1145/1835698.1835736

[2] A. Natarajan and N. Mittal, "Fast concurrent lock-free binary search trees," *SIGPLAN Not.*, vol. 49, no. 8, p. 317–328, feb 2014. [Online]. Available: https://doi.org/10.1145/2692916.2555256

[3] J. D. Valois, "Lock-free linked lists using compare-and-swap," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 214–222. [Online]. Available: https://doi.org/10.1145/224964.224988

[4] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the 15th International Conference on Distributed Computing*, ser. DISC '01. Berlin, Heidelberg: Springer-Verlag, 2001, p. 300–314.

[5] M. Dodds, A. Haas, and C. M. Kirsch, "A scalable, correct time-stamped stack," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 233–246. [Online]. Available: https://doi.org/10.1145/2676726.2676963

[6] N. Shafiei, "Non-blocking patricia tries with replace operations," *Distrib. Comput.*, vol. 32, no. 5, p. 423–442, oct 2019. [Online]. Available: https://doi.org/10.1007/s00446-019-00347-1

[7] D. Basin, E. Bortnikov, A. Braginsky, G. Golan-Gueta, E. Hillel, I. Keidar, and M. Sulamy, "Kiwi: A key-value map for scalable real-time analytics," *ACM Trans. Parallel Comput.*, vol. 7, no. 3, jun 2020. [Online]. Available: https://doi.org/10.1145/3399718

[8] H. Avni, N. Shavit, and A. Suissa, "Leaplist: Lessons learned in designing tm-supported range queries," in *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, ser. PODC '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 299–308. [Online]. Available: https://doi.org/10.1145/2484239.2484254

[9] J. Nelson-Slivon, A. Hassan, and R. Palmieri, "Bundling linked data structures for linearizable range queries," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 368–384. [Online]. Available: https://doi.org/10.1145/3503221.3508412

[10] M. Arbel-Raviv and T. Brown, "Harnessing epoch-based reclamation for efficient range queries," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 14–27. [Online]. Available: https://doi.org/10.1145/3178487.3178489

[11] Y. Wei, N. Ben-David, G. E. Blelloch, P. Fatourou, E. Ruppert, and Y. Sun, "Constant-time snapshots with applications to concurrent data structures," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 31–46. [Online]. Available: https://doi.org/10.1145/3437801.3441602

[12] M. Rodriguez and M. F. Spear, "Optimizing linearizable bulk operations on data structures," in *ICPP 2020: 49th International Conference on Parallel Processing, Edmonton, AB, Canada, August 17-20, 2020*, J. N. Amaral, L. K. John, and X. Shen, Eds. ACM, 2020, pp. 24:1–24:10. [Online]. Available: https://doi.org/10.1145/3404397.3404414

[13] T. Kobus, M. Kokociński, and P. T. Wojciechowski, "Jiffy: A lock-free skip list with batch updates and snapshots," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 400–415. [Online]. Available: https://doi.org/10.1145/3503221.3508437

[14] E. Giles, K. Doshi, and P. Varman, "Hardware transactional persistent memory," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 190–205. [Online]. Available: https://doi.org/10.1145/3240302.3240305

[15] R. M. Krishnan, J. Kim, A. Mathew, X. Fu, A. Demeri, C. Min, and S. Kannan, *Durable Transactional Memory Can Scale with Timestone*. New York, NY, USA: Association for Computing Machinery, 2020, p. 335–349. [Online]. Available: https://doi.org/10.1145/3373376.3378483

[16] Intel, "clock() or gettimeofday() or ippgetcpuclocks()?" Intel, Tech. Rep., 2010. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/best-timing-function-for-measuring-ipp-api-timing.html

[17] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, "Oplog: a library for scaling update-heavy data structures," MIT CSAIL, Tech. Rep., 2013.

[18] M. Arbel and A. Morrison, "Predicate rcu: An rcu for scalable concurrent updates," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 21–30. [Online]. Available: https://doi.org/10.1145/2688500.2688518

[19] M. Arbel and H. Attiya, "Concurrent updates with RCU: search tree as an example," in *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*. ACM, 2014, pp. 196–205. [Online]. Available: https://doi.org/10.1145/2611462.2611471

[20] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit, "A simple optimistic skiplist algorithm," in *Structural Information and Communication Complexity, 14th International Colloquium, SIROCCO 2007, Castiglioncello, Italy, June 5-8, 2007, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4474. Springer, 2007, pp. 124–138. [Online]. Available: https://doi.org/10.1007/978-3-540-72951-8_11

[21] Intel, "Intel 64 and ia-32 architectures software developer's manual," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[22] S. Kashyap, C. Min, K. Kim, and T. Kim, "A scalable ordering primitive for multicore machines," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3190508.3190510

[23] ARM, "Pmccntr, performance monitors cycle count register, vmsa." [Online]. Available: https://developer.arm.com/documentation/ddi0406/c/System-Level-Architecture/System-Control-Registers-in-a-VMSA-implementation/VMSA-System-control-registers-descriptions–in-register-order/PMCCNTR–Performance-Monitors-Cycle-Count-Register–VMSA

[24] T. L. Harris, K. Fraser, and I. A. Pratt, "A practical multi-word compare-and-swap operation," in *Proceedings of the 16th International Conference on Distributed Computing*, ser. DISC '02. Berlin, Heidelberg: Springer-Verlag, 2002, p. 265–279.

[25] H. Lim, M. Kaminsky, and D. G. Andersen, "Cicada: Dependably fast multi-core in-memory transactions," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 21–35. [Online]. Available: https://doi.org/10.1145/3035918.3064015

[26] W. Ruan, Y. Liu, and M. Spear, "Boosting timestamp-based transactional memory by exploiting hardware cycle counters," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, dec 2013. [Online]. Available: https://doi.org/10.1145/2541228.2555297