Snapshot-based Synchronization: A Fast Replacement for Hand-over-Hand Locking

Eran Gilad[1], Trevor Brown[2], Mark Oskin[3], and Yoav Etsion[1]

[1] Technion – Israel Institute of Technology
erangi@cs.technion.ac.il, yetsion@tce.technion.ac.il
[2] Institute of Science and Technology, Austria
[3] University of Washington, Seattle, USA

Abstract. Concurrent accesses to shared data structures must be synchronized to avoid data races. Coarse-grained synchronization, which locks the entire data structure, is easy to implement but does not scale. Fine-grained synchronization can scale well, but can be hard to reason about. Hand-over-hand locking, in which operations are pipelined as they traverse the data structure, combines fine-grained synchronization with ease of use. However, the traditional implementation suffers from inherent overheads.

This paper introduces snapshot-based synchronization (SBS), a novel hand-over-hand locking mechanism. SBS decouples the synchronization state from the data, significantly improving cache utilization. Further, it relies on guarantees provided by pipelining to minimize synchronization that requires cross-thread communication. Snapshot-based synchronization thus scales much better than traditional hand-over-hand locking, while maintaining the same ease of use.

1 Introduction

Hand-over-hand locking[4] is a fine-grained synchronization technique that prevent data races among concurrent operations. Commonly applied to pointer-based data structures, operations lock nodes as they traverse the data structure. In order to prevent bypassing, a node's lock is released by the owning operation only after it acquires the next node's lock. Generally, operations that traverse the same path are *pipelined*. As the pattern guarantees a node will not be concurrently accessed by two threads, data races are avoided.

The fine nature of hand-over-hand locking exposes more parallelism. Given each thread locks at most two nodes at once, multiple threads can operate on a data structure concurrently. Threads are ordered, namely one is forced to wait for another, only when trying to access the same node. In a tree, ordering always applies to the root, as locks are associated with nodes. However, threads operating on different branches need not be ordered once their paths diverge.

The concept of hand-over-hand locking is appealing: fine-grained locking exposes large amounts of parallelism, and ordering provides thread safety. Ordering

[4] Also known as *lock coupling*, *chain locking*, *latch coupling*, *crabbing* etc.

also makes hand-over-hand locking easy to apply to sequential data structures (that have properties discussed later), providing a quick way to parallelize existing sequential code. Indeed, the popular textbook The Art of Multiprocessor Programming [1] uses hand-over-hand locking to demonstrate fine-grained locking. However, naïve hand-over-hand locking suffers from a few inherent limitations, causing it to be rarely used in the real world.

Poor cache utilization: Memory latencies are the most significant shortcoming of hand-over-hand locking. Acquiring and releasing per-node locks cause memory state modifications. As a thread makes its way to a certain node, it modifies the state of each node it passes. The modification is not performed on the data that the data structure is designed to hold (keys, values and pointers) but rather to the state of each node's lock. Consequently, even read-only accesses still require changes to memory for each node accessed. In the memory system, writes to a node that are performed on one core invalidate any cached copies of that node on other cores. Accessing nodes that are not in the cache can be two orders of magnitude slower than accessing cached nodes. Given a large enough number of threads operating on the same data structure, the overhead incurred by poor cache utilization can exceed the potential benefits of parallelism.

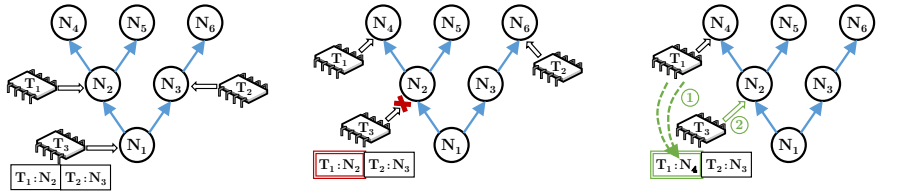
Entrance bottleneck: Locking each node during traversal provides thread safety, but also turns the entrance to the data structure into a bottleneck. Consider operations on a tree: as every thread must go through the root, the root's lock effectively serializes all accesses. While parallelism increases as threads diverge in the tree, the serialized entrance caps potential speedup on parallel execution. The effect of the bottleneck is determined by the number of threads and the depth of the tree, which yield a ratio between threads actively traversing the tree and threads stalled at the entrance.

Extra locking: As each node is associated with a different lock, moving from one node to the next requires both to be locked at the beginning of the transition. Albeit for a short while, the extra locking delays the divergence of threads that share an initial prefix of their paths. This initial prefix always includes the entrance of the data structure, which should be evacuated quickly.

1.1 Snapshot-based Synchronization

Snapshot-based synchronization is designed to address the shortcomings of basic hand-over-hand locking while maintaining the same ease of use. The fundamental insights driving snapshot-based synchronization are: (1) the number of locations that must be locked at any given moment is bound by the number of threads, not the number of nodes; and (2) as long as nodes are locked in the correct order, a thread cannot overtake (namely, race with) the thread in front of it, even if it somehow gets a delayed view of the first thread's traversal.

Building on those insights, snapshot-based synchronization decouples locks from nodes and associates them with threads. Each lock is then dynamically assigned to a single memory location, which represents the location of the node currently accessed by the thread. At any given moment, the set of locked locations can be considered to be a snapshot of all threads' locations. As depicted in



(a) T_3 enters. Snapshot: T_1 at N_2, T_2 at N_3

(b) T_3 can't access N_2 - snapshot indicates T_1 might be there

(c) After loading T_1's new location, T_3 can proceed

Fig. 1: (a) Thread T_3 creates a snapshot when entering tree; (b) uses it to detect potential collision; and (c) moves on after ensuring T_1 is no longer at N_2.

Figure 1, a thread that obtains such a snapshot when entering the data structure can query it throughout the traversal; as long as a node it wishes to access is not in the snapshot, the thread can freely access that node. If the node's location happens to exist in the snapshot, the current thread must wait until the thread at that location moves on.

Snapshot-based synchronization's main component is therefore the snapshot, which marks the locations of all other threads when taken. As threads move on, the snapshot quickly becomes outdated. However, observing outdated location can merely cause unnecessary waits; necessary waits to threads traversing the same path will never be missed. Crucially, since threads that complete an operation can reenter the data structure, a snapshot cannot be used indefinitely, and a thread must obtain a fresh snapshot at the beginning of each operation.

To facilitate location-based synchronization, threads must report their whereabout in a place that is visible to other threads. Reporting should take place often to reduce unnecessary stalls caused by false synchronization. However, the use of snapshots allows location reports to be seldom read – only when a snapshot indicates possible contention must a thread reload the locations of the others.

2 Snapshot-based Synchronization Design

In this section we describe the basic design of snapshot-based synchronization and its core components. While the basic design overcomes most of the limitations of hand-over-hand locking, some are rooted deep in the pipelining pattern. Optimizations that address those limitations are discussed on the next section.

Hand-over-hand locking pipelines threads that traverse the same path. In other words, a thread can access a node that was locked by the thread in front of it only once the leading thread moved on and unlocked the node. Bypassing within such a pipeline is impossible, so data races are avoided. Threads whose paths diverge are no longer synchronized, consequently hand-over-hand locking is only applicable to data structures that have no cycles (and algorithms that introduce no such cycles by, say, revisiting a node during a rebalancing phase). Snapshot-based synchronization is designed as a substitute for hand-over-hand

Table 1: API for hand-over-hand vs. snapshot-based synchronization

Operation	**Hand-over-hand**	**Snapshot-based sync.**
Lock head	head–>lock()	moveToHead(head)
Before accessing node	node–>lock()	waitForLoc(node)
After access granted	prev–>unlock()	moveToLoc(node)

locking, and its correctness is guaranteed only when the latter is safe. Graph data structures that have cycles, for instance, can neither be synchronized using hand-over-hand nor using snapshot-based synchronization.

The central component of snapshot-based synchronization is the snapshot. As depicted in Figure 1, when a thread enters the data structure, it records the location of all other threads. Before the thread moves to another location, it checks if the snapshot recorded any other thread at that location. If so, it must not access the location until it verifies the other thread has moved. This verification is done by obtaining the latest location of the other thread (and possibly additional ones, as discussed later). Consequently, each thread must report its current location once it moves.

Snapshot-based synchronization manages two kinds of data: private (per thread) and public (shared). Snapshot-based synchronization reduces cross-thread communication by serving most reads from private data, falling back to reading public data only when encountering possible contention. Each thread stores the snapshot in private memory. The current location of each thread, on the other hand, is stored publicly and is available to all other threads. However, public data is read only when a snapshot must be created or updated.

Snapshot-based synchronization leverages modern hardware features to reduce overheads: loads from local caches are much faster than loads from main memory, and stores do not stall subsequent operations. The snapshot is read often but can be efficiently cached. Threads frequently report their locations publicly, but due to micro-architecture features such as out-of-order execution and store buffers, location reports do not stall subsequent instructions even if they incur a cache miss.

2.1 Interface and algorithms

Snapshot-based synchronization's interface is similar to hand-over-hand locking's, and converting code using the latter to the former is straightforward. However, the underlying operations differ significantly, and the interface naming represents the actual semantics. Briefly, when using snapshot-based synchronization, operations must start with a call to moveToHead. Before accessing a location, waitForLoc must be called to make sure no other thread is present at that location. Lastly, moveToLoc is used to publish the new location of the thread, preventing others from accessing it. Table 1 compares the two interfaces.

moveToHead Since most synchronization is done using the private snapshot, it is crucial that the snapshot is sufficiently up-to-date. In particular, a snapshot *must include the location of each thread that entered the data structure before the current thread and has not completed its operation yet.* Using a snapshot that does not include all threads ahead might yield a race.

The pipelining pattern must be maintained by snapshot creation as well. A snapshot is used to ensure a thread does not bypass (race with) threads in front of it. Given all threads enter the data structure via a single entry point, a snapshot must be created right before attempting to enter and must record all threads ahead. However, the snapshot needs not include threads that are behind in the pipeline – it is up to those threads behind to make sure they stay behind.

The moveToHead operation is implemented as follows:

1. Establish ordering among threads competing at the entrance
2. Once the leading thread allows, create a snapshot by gathering the locations of all threads ahead
3. Wait for the entrance to become available
4. Move to the entrance and update current location
5. Allow following thread to create a snapshot

Two threads must not create a snapshot at the same time. Doing so will cause both to miss each other, and since one will eventually enter ahead of the other, the missing location will cause a race.

A significant part of entering the data structure requires serialization. Measures must therefore be taken to mitigate the bottleneck. Those measures are detailed in Section 3. moveToHead has no equivalent operation in hand-over-hand. Instead, in hand-over-hand the order in which threads lock the root of the data structure determines the order in which they will lock (and access) all other nodes, until their paths diverge.

waitForLoc Before a thread can access a location, it must make sure no other thread will concurrently modify that location. To do so, the thread must:

1. Check if the snapshot contains any other thread at that location
2. If no thread was observed at that location, waitForLoc can safely return
3. Else, the current thread must wait until the thread ahead moves
4. Update its snapshot

The minimal update of the snapshot depends on the modifications done by the data structure algorithms. Consider a thread T_1, which executes an operation that does not modify the layout of the data structure (e.g., updates a value in a binary search tree), and a thread T_2 which is behind T_1. If T_2 waits for T_1 before accessing some location, only T_1's location must be updated in T_2's snapshot. However, if T_1 deletes a node, it might prevent T_2 from waiting to some T_0 that T_2 observed at the deleted node. In such cases, T_2's snapshot must be recreated.

(a) T_1 locks all N_1, preventing T_2 from moving towards N_3

(b) T_1 locks the pointer to N_2, allowing T_2 to move to N_3

Fig. 2: Locking nodes vs. locking pointers. The latter allows more parallelism.

moveToLoc Moving to the next location is simple: a thread just updates its publicly visible location. This move is equivalent to locking the next node and unlocking the previous one in hand-over-hand. The overhead, however, is noticeably lower: the state of involved nodes is not changed, and only one location is locked at any given moment. Hand-over-hand's excessive locking is due to the lack of support for a single atomic modification of multiple memory locations in current hardware[5], which does not allow two locks to be modified at once.

2.2 Locking granularity

Hand-over-hand relies on locks, and must therefore bind a lock to every object it wishes to protect. The most natural locking granularity is one lock per node[6]. Locking a node prevents all its fields from being accessed by other threads. Consider a tree in which node N_1 points to N_2 and N_3, depicted in Figure 2a. Thread T_1 locks N_1, and is now considering whether it needs to delete N_2 (which will also involve modifying the pointer on N_1). Thread T_2 is heading towards N_3, but must pass through N_1. While neither N_2 nor the pointer to N_2 will be accessed by T_2, per-node locking will force T_2 to wait until T_1 unlocks N_1.

Snapshot-based synchronization does not use lock objects, and instead (semantically) locks memory locations. Consequently, locking can be done at any desired granularity. The one we had found most useful is per pointer. Consider the previous example; as depicted in Figure 2b, on a per-pointer synchronization scheme, T_1 would have locked the pointer to N_2. T_2 could have then check N_1's key, determine it needs to go to N_3, and freely move on without being stalled by T_1. On lower parts of the tree, threads usually diverge and locking granularity has little effect. However, contention is a major problem at the top of the tree, and locking pointers eliminates unneeded synchronizations.

[5] Hardware transactional memory does allow multiple modifications to happen effectively atomically, but is not ubiquitous. We discuss software TM in Section 4.

[6] A lock array can service any number of nodes using some hash function but might cause deadlocks, and in our experiments, not faster than storing locks as node fields.

3 Optimized Implementation

The basic snapshot-based synchronization scheme eliminates hand-over-hand's poor cache utilization and excessive locking overheads. However, the root of the data structure remains a bottleneck. Creating a snapshot involves reading the current locations of all threads. Since the locations are constantly being updated by the reporting threads, creating a snapshot incurs multiple cache misses. Given snapshots cannot be created in parallel, taking a snapshot before entering the data structure serializes execution for a large portion of the run. In this section, we discuss major optimizations that improve snapshot-based synchronization's efficiency, and in particular mitigate the entrance bottleneck.

3.1 Copying snapshots

Creating a snapshot involves accessing data constantly updated, incurring multiple cache misses. To avoid creating a snapshot from scratch, a thread can copy the snapshot used by the immediate leading thread. If the complete snapshot resides on a single cache line, copying incurs a single cache miss.

Snapshots can only be copied from the thread that entered immediately before the thread that needs the snapshot. Consider threads T_1, T_2 and T_3 entering a data structure, in this order. T_1's snapshot is created first, thus does not include T_2's location. If T_3 copies from T_1, it might race with T_2. On the other hand, if T_3 copies T_2's snapshot it might obtain a somewhat stale view of T_1's location. However, the worst outcome would be the detection of false collisions. Importantly, care must be taken to avoid using snapshots after re-entrance into the data structure: if T_2 completes its operation, enters the data structure again and tries to copy T_3's snapshot before T_3 gets to copy T_2's, neither will have a valid snapshot. This is a variant of the ABA problem, which we solve using the conventional tool – timestamps. Once a thread detects it copied an invalid snapshot, it simply falls back to creating a new one from scratch.

3.2 Deferring snapshot creation by trailing

A thread that immediately follows a previous thread does not need a snapshot; we call this state *trailing*. Due to the nature of pipelining, no thread can appear between two consecutive threads. As illustrated in Figure 3, while T_2 trails T_1, it can rely on T_1 to resolve any collision with threads in front of them, allowing T_2 to merely ensure it does not bypass T_1. T_2 can thus defer obtaining a snapshot until trailing breaks. Trailing thus eliminates the need to create a snapshot before entering, significantly shortening the bottleneck. Further, trailing eliminates most contention points involving more than two threads, akin to MCS locks [2].

While T_2 trails T_1, T_2 examines the location of T_1 instead of checking the snapshot. As long as T_1 is still at the location T_2 wishes to move to, T_2 will spin; once T_1 moves, T_2 can immediately follow. While this cross-thread communication is more expensive than checking a private snapshot, it is cheaper than creating one. In the heavily-contended entrance, quickly evacuating the entrance

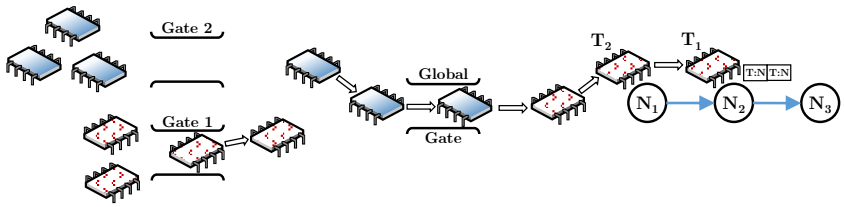


Fig. 3: Local gates order threads coming from the same NUMA node, creating *chains*. The global gate orders the entrance of chains into the data structure. While *trailing*, prev.'s position is examined directly without using a snapshot.

reduces stalls. Trailing stops as soon as T_2 cannot be sure T_1 passed through the memory location it tries to access, whether because T_1 moved too fast to the next location or because T_1 turned another way. Once trailing stops, T_2 cannot rely on T_1 and must create (or copy) a snapshot before moving on.

3.3 NUMA awareness

On NUMA systems, accessing remote memory (associated with another NUMA node) is significantly slower than accessing local memory. Keeping as much cross-thread communication within the same NUMA node can therefore reduce memory latencies. While snapshot-based synchronization is agnostic to the memory management of the hosting data structure, adding NUMA-awareness to the synchronization mechanism reduces its overhead.

Snapshot-based synchronization employs a technique that groups threads of the same NUMA node, orders them internally, and lets them enter the data structure in this exact order. The mechanism, depicted in Figure 3, resembles the one used in cohort locks [3]: a per NUMA node gate is first used to create *chains* of threads belonging to that NUMA node. The *head* of each chain (namely the first thread) competes over the global gate only with other heads. Once acquired, the head closes its following chain and announces the last thread in the chain via the global gate. The head of the next chain (probably coming from another NUMA node) will trail the last thread in the chain in front of it.

Threads within the same chain all run on the same NUMA node. Trailing and snapshot copying among those threads are noticeably faster than across NUMA nodes. The ratio between local and remote communication is determined by the length of the chains. Interestingly, if entering the data structure becomes slow (e.g., due to some external interference) and threads accumulate at the entrance, longer chains will be created. This in turn will provide more local communication, allowing threads to leave the head quicker, reducing entrance time.

3.4 Reader synchronization

Read-only operations such as lookups are usually easier to parallelize, as they need not synchronize with other readers (synchronization with write operations

is required, of course). In a hand-over-hand algorithm, readers can thus safely bypass each other. This freedom could be of great use when threads enter the data structure. Unfortunately, the straightforward readers optimization breaks other optimizations. For instance, if writer W_1 trails reader R_1, and R_1 bypasses R_2, then W_1 will race with R_2. Similarly, writers cannot copy snapshots from readers as they might include stale locations of other readers. Our implementation includes a restricted set of reader optimizations. We do not elaborate on them due to lack of space, and leave further reader optimizations for future work.

3.5 Putting it all together

The optimized snapshot-based synchronization overcomes inherent limitations of hand-over-hand: **Poor cache locality** is minimized by decoupling synchronization state from the data structure and using a snapshot to further reduce cross-thread communication. **The entrance bottleneck** is mitigated by using NUMA-aware algorithms, deferring snapshot creation and reusing snapshots. **Extra locking** is avoided by allowing an atomic move from one location to another and by locking pointers rather than nodes. **Reader synchronization** is reduced by allowing readers to bypass each other. The following section shows snapshot-based synchronization is indeed faster than hand-over-hand locking.

4 Evaluation

In this section we compare the actual performance of snapshot-based synchronization (*SBS*) to alternative synchronization mechanisms, revealing both strengths and weaknesses. The alternative mechanisms are (a) traditional hand-over-hand (*HOH*) and (b) software transactional memory (*STM*). Like SBS, STM is a synchronization mechanism external to the data structure, which can be used to parallelize sequential data structures. State-of-the-art concurrent data structures can be much faster, but synchronization is deeply integrated in the structures and associated algorithms. We therefore do not consider them comparable.

4.1 Experimental setup

We perform a series of micro-benchmarks, running a mix of operations on binary trees. We consider both integers (*INT*) and strings (*STR*) as key types – while the former is more common in the literature, the latter is very common in real programs, and sometimes exhibits a different behavior. All evaluations execute a similar number of inserts and deletes, keeping the data structure size stable; we also study the effect of the initial size. Lastly, read-write ratio on all benchmarks is 50-50. We do not analyze other ratios due to space limitations; in short, our evaluation finds snapshot-based synchronization favors write-heavy workloads.

The server used has 2 NUMA nodes and Intel Xeon E5-2630 processors running at 2.4Ghz. Hyperthreading, Turbo Boost and adjacent cache line prefetching were disabled. Each core has 32KB L1 and 236KB L2 caches; each processor has

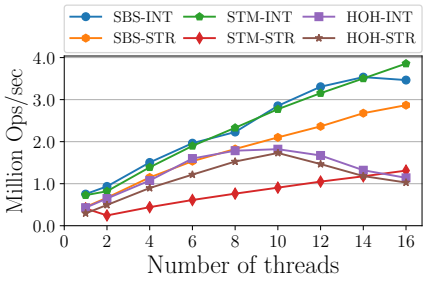


Fig. 4: Scalability using 2 NUMA nodes (init: 10^6)

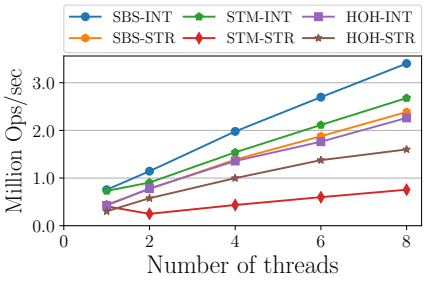


Fig. 5: Scalability using 1 NUMA node (init: 10^6)

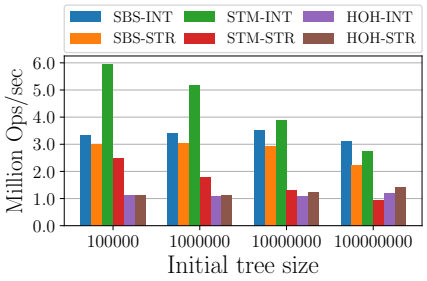


Fig. 6: Effect of initial size (16 threads)

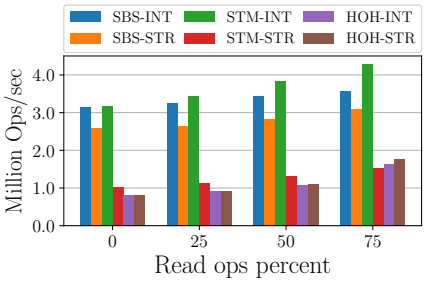


Fig. 7: Effect of read-write ratio (16 threads, 10^6 init size)

a 20MB L3 cache; and the system has 62GB of RAM. Code was written in C++ and compiled with GCC 7.2, which also provided the STM support.

4.2 Scalability

Figure 4 presents the throughput of all workloads running on a varying number of threads, evenly distributed between the 2 NUMA nodes. Evidently, HOH does not scale past 10 threads, and synchronization overhead overwhelms performance as the number of threads increases. On the INT workloads, SBS is slightly slower than STM. However, while STM's scalability is consistent, SBS reaches its peak at 14 threads. The STR workloads demonstrate different trends, as more work is performed during traversal (mostly string comparisons, involving multiple memory accesses in a loop). Extending traversals reduces contention at the entrance, allowing SBS to continue scaling past 16 threads. STM, however, suffers from enlarged read and write sets, causing throughputs to drop.

Figure 5 presents scalability when running on a single NUMA node. The results emphasize the effect of NUMA: as cross-core communication is much faster when running on the same NUMA node, HOH and SBS scale much better. Most

of the gain comes from entering the tree faster due to reduced cross-thread communication latencies. STM, which does not require communication at that point, sees little gain in this scenario. In summary, HOH and SBS are more NUMA sensitive than STM. SBS performs best on most scenarios, but short traversal times (INT) with long communication (NUMA) cap scaling at 14 threads.

4.3 Effect of data structure size

The size of the data structure affects the duration of the traversal. As indicated by the difference between INT and STR workloads, traversal time correlates to entrance contention, which in turn determines scalability. Figure 6 presents the throughput of the 6 benchmarks when running on trees of different sizes - 10^5, 10^6, 10^7 and 10^8; all using 16 threads. Accessing more memory locations as the tree grows causes STM throughput to decrease. SBS, however, has about the same throughput on the smaller 3 sizes. This somewhat unexpected behavior is clearer when examining the results in the opposite direction: SBS throughput does not increase as the tree size becomes smaller, indicating the size is not the dominant factor. For SBS, 16 threads is the scalability limit on 10^7 trees; on smaller trees, entrance is even more contended, canceling the benefit of shorter traversals. In summary, SBS is more appropriate for trees of size 10^7 and higher when using simple INT keys. When using STR keys that increase SBS traversal times and STM's read sets, SBS consistently performs best.

4.4 Effect of read-write ratio

Since in our snapshot-based synchronization implementation readers enter the data structure one-by-one, entrance bottleneck has a similar effect on scalability regardless of reader-write ratio. Figure 7 shows that the write-only throughput of SBS is equivalent to STM, but STM becomes faster as the percent of read operations is increased. Further optimizing readers could make SBS scale better, but the implementation is non-trivial. Instead, snapshot-based synchronization can be integrated with mechanisms such as RCU [4], combining multiple concurrent writers with wait-free readers.

4.5 Entrance bottleneck analysis

Serialization at the entrance limits parallelism; we now dive deeper into this part of execution. In our implementation, execution can be divided into 3 parts: (1) initial ordering, (2) accessing the head, and (3) traversing the tree. The first and last parts are mostly parallel. Accessing the head, however, can be done by a single thread at a time. A thread can not access the head until it detected the previous thread moved to another node. Single-threaded execution thus takes place between the time one thread detects it can access the head to the time the following thread detects it can move on.

Before a thread can allow the following one to access the head, it needs to move to another node. If the thread is the first in a chain, it must first make the

Table 2: Breakdown of overhead between accessing the head and allowing the following thread to access the head.

Operation	Overhead	Frequency
Evacuate global gate	Cache misses on local read and remote write	Once per chain
Create a snapshot	Varies	Rare, due to trailing
Await node after head	Varies	Always
Move to node after head	Sometimes cache miss on local write	Always
Arrival of updated location to next thread	Cache miss on read	Once per chain remote miss, otherwise local miss

global gate available for the next chain once it accessed the head. It must then move to the other node and report its new location. Lastly, the following thread must read that report. The overheads of this sequence are detailed in Table 2. In our experiments, on a 16-thread write-only SBS execution the sequence took an average of 700 cycles. Multiplying this sequence latency by the throughput of 3M ops/s yields 2.1G cycles. The latency incurred by the traversal of the head is the execution's critical path, and matches our processors 2.4GHz frequency. In summary, Scalability is limited by the rate in which threads access the head. Our implementation minimizes accesses to remote memory, but cache misses that involve communication with a core residing on the same NUMA node incur significant overhead. Serialized execution time can be reduced by either eliminating operations or using faster cross-core communication; x86 MONITOR and MWAIT, once available in user mode, are certainly of interest [5].

5 Related Work

The hand-over-hand locking scheme (also known as lock coupling, latch coupling, crabbing etc.) was first described by Bayer and Schkolnick [6] as a way to construct concurrent B-trees. It has since been used to parallelize various data structures. As the major synchronization mechanism, it was used in linked lists [1], B-trees [7], skip lists [8], relaxed red-black trees [9] and a Treap [10]. As a utility for a certain part of the algorithm, it was also used in priority queues [11], B+-trees [12, 13], B^{link}-trees [14, 15] and hash tables [16].

Data structures with properties allowing hand-over-hand synchronization have been defined as Unipath [17] and Dominance Locking [10]. Those properties allow serializability verification [18, 19] and even automatic parallelization [10].

Locking individual memory locations has been supported in various forms. Lock-box [20] provided architectural support for SMT threads to lock particular addresses without using conventional synchronization mechanisms. The Synchronization State Buffer [21] extended this idea to a many-core system, while vLock [22] offered a software solution. TL2 [23] incorporated an array of locks in an STM library, allowing a fixed (yet large) set of locks to protect any number of

locations. ROKO [24] synchronized accesses using versioning memory locations, and O-structures [25] added renaming to eliminate false dependencies.

6 Conclusions

Hand-over-hand locking is a widespread fine-grained synchronization technique. The simple interface makes hand-over-hand attractive, and it has been used to parallelize multiple data structures. Furthermore, the method is simple to reason about, allowing verification and automatic parallelization. However, fine-grained locking comes at a price: locking causes cache misses on every node access. As all threads enter at the same place, the top of the data structure becomes a bottleneck that disallows scaling past a small number of threads.

Snapshot-based synchronization is a drop-in replacement for hand-over-hand locking, but uses a very different synchronization mechanism under the hood. Leveraging the data structure layout, private snapshots allow threads to avoid data races without communicating with other threads. Leveraging modern hardware, communication minimally interferes with the surrounding algorithm. In our evaluation, on large data structures snapshot-based synchronization is on average 2.6× faster than hand-over-hand locking and 1.6× faster than STM.

While its interface is simple and easy to use, Snapshot-based synchronization's implementation is considerably more complex than simple per-node locks. Albeit undesired in general, complexity brings about many optimization opportunities. We consider the implementation described in this paper a baseline: other implementations, possibly using newer hardware features, can make snapshot-based synchronization scale even better. In particular, reducing data structure entrance time and relaxing reader-to-reader synchronization are of interest.

Acknowledgements This research was funded, in part, by Google and the Israel Ministry of Science, Technology, and Space. Trevor Brown was supported in part by the ISF (grants 2005/17 & 1749/14) and by a NSERC post-doctoral fellowship. Eran Gilad was supported by the Hasso-Plattner Institute fellowship.

References

1. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc. (2008)
2. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. on Computer Systems **9**(1) (1991)
3. Dice, D., Marathe, V.J., Shavit, N.: Lock cohorting: A general technique for designing numa locks. In: Symp. on Principles and Practices of Parallel Programming (PPoPP). (2012)
4. Desnoyers, M., McKenney, P.E., Stern, A.S., Dagenais, M.R., Walpole, J.: User-level implementations of read-copy update. IEEE Trans. on Parallel and Distributed Systems **23**(2) (Feb 2012)

5. Akkan, H., Lang, M., Ionkov, L.: Hpc runtime support for fast and power efficient locking and synchronization. In: Intl. Conf. on Cluster Computing. (2013)
6. Bayer, R., Schkolnick, M.: Concurrency of operations on b-trees. Acta informatica **9** (1977)
7. Rodeh, O.: B-trees, shadowing, and clones. ACM Trans. on Storage **3**(4) (2008)
8. Sánchez, A., Sánchez, C.: A theory of skiplists with applications to the verification of concurrent datatypes. In: NASA Formal Methods Symp. (NFM). (2011)
9. Ohene-Kwofie, D., Otoo, E.J., Nimako, G.: Concurrent operations of o2-tree on shared memory multicore architectures. In: Australasian DB Conf. (ADC). (2013)
10. Golan-Gueta, G., Bronson, N., Aiken, A., Ramalingam, G., Sagiv, M., Yahav, E.: Automatic fine-grain locking using shape properties. In: Object-oriented Programming, Systems, Languages, and Applications (OOPSLA). (2011)
11. Tamir, O., Morrison, A., Rinetzky, N.: A heap-based concurrent priority queue with mutable priorities for faster parallel algorithms. In: Intl. Conf. on Principles of Distributed Systems (OPODIS). (2016)
12. Srinivasan, V., Carey, M.J.: Performance of b+ tree concurrency control algorithms. Very Large Databases Journal (JVLDB) **2**(4) (1993)
13. Mao, Y., Kohler, E., Morris, R.T.: Cache craftiness for fast multicore key-value storage. In: European Conf. on Computer Systems (EUROSYS). (2012)
14. Evangelidis, G., Lomet, D., Salzberg, B.: The hb^{Π}-tree: A multi-attribute index supporting concurrency, recovery and node consolidation. Very Large Databases Journal (JVLDB) **6**(1) (1997)
15. Jaluta, I., Sippu, S., Soisalon-Soininen, E.: Concurrency control and recovery for balanced b-link trees. Very Large Databases Journal (JVLDB) **14**(2) (2005)
16. Ellis, C.S.: Distributed data structures: A case study. IEEE Trans. on Computers (1985)
17. Gilad, E., Mayzels, T., Raab, E., Oskin, M., Etsion, Y.: Towards a deterministic fine-grained task ordering using multi-versioned memory. In: Computer Architecture and High Performance Computing (SBAC-PAD). (2017)
18. Attiya, H., Ramalingam, G., Rinetzky, N.: Sequential verification of serializability. In: Symp. on Principles of Programming Languages (POPL). (2010)
19. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: Symp. on Principles and Practices of Parallel Programming (PPoPP). (2006)
20. Tullsen, D.M., Lo, J.L., Eggers, S.J., Levy, H.M.: Supporting fine-grained synchronization on a simultaneous multithreading processor. In: Symp. on High-Performance Computer Architecture (HPCA). (1999)
21. Zhu, W., Sreedhar, V.C., Hu, Z., Gao, G.R.: Synchronization state buffer: Supporting efficient fine-grain synchronization on many-core architectures. In: Intl. Symp. on Computer Architecture (ISCA). (2007)
22. Yan, J., Tan, G., Zhang, X., Yao, E., Sun, N.: Vlock: Lock virtualization mechanism for exploiting fine-grained parallelism in graph traversal algorithms. In: Intl. Symp. on Code Generation and Optimization (CGO). (2013)
23. Dice, D., Shalev, O., Shavit, N.: Transactional locking ii. In: Intl. Symp. on Distributed Computing (DISC). (2006)
24. Segulja, C., Abdelrahman, T.: Architectural support for synchronization-free deterministic parallel programming. In: Symp. on High-Performance Computer Architecture (HPCA). (2012)
25. Gilad, E., Mayzels, T., Raab, E., Oskin, M., Etsion, Y.: Architectural support for unlimited memory versioning and renaming. In: Intl. Parallel & Distributed Processing Symp. (IPDPS). (2018)