More Than You Ever Wanted to Know about Synchronization

Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms

Vincent Gramoli
NICTA and University of Sydney, Australia
vincent.gramoli@sydney.edu.au

Abstract

In this paper, we present the most extensive comparison of synchronization techniques. We evaluate 5 different synchronization techniques through a series of 31 data structure algorithms from the recent literature on 3 multicore platforms from Intel, Sun Microsystems and AMD. To this end, we developed in C/C++ and Java a new micro-benchmark suite, called *Synchrobench*, hence helping the community evaluate new data structures and synchronization techniques. The main conclusion of this evaluation is threefold: (i) although compare-and-swap helps achieving the best performance on multicores, doing so correctly is hard; (ii) optimistic locking offers varying performance results while transactional memory offers more consistent results; and (iii) copy-on-write and read-copy-update suffer more from contention than any other technique but could be combined with others to derive efficient algorithms.

Categories and Subject Descriptors D.1. Programming Techniques [*Concurrent Programming*]: Parallel programming

Keywords Benchmark; data structure; reusability; lock-freedom

1. Introduction

The increasing core count raises new challenges in the development of efficient algorithms that allow concurrent threads to access shared resources. Not only have developers to choose among a large set of thread synchronization techniques, including locks, read-modify-write, copy-on-write, transactions and read-copy-update, but they must select dedicated data structure algorithms that leverage each synchronization under a certain workload. These possibilities have led to an increase in the number of proposed concurrent data structures, each being shown efficient in "some" settings. Unfortunately, it is almost impossible to predict their performance given the hardware and OS artifacts. A unique framework is thus necessary to evaluate their performance on a common ground before recommending developers to choose a specific synchronization technique.

On the one hand, synchronization techniques are usually tested with standard macro-benchmarks [8] whose workloads alternate realistically between various complex patterns. These macro-benchmarks are however of little help when it comes to nailing down the bottleneck responsible of performance drops. On the other hand, profiling tools that measure cache traffic [18] and monitor memory reclamation can be extremely useful in tuning the implementation of an algorithm to a dedicated hardware platform, however, they are of little help in optimizing the algorithm itself.

This is the reason why micro-benchmarks have been so popular to evaluate new algorithms. They are invaluable tools that complement macro evaluations and profiling tool boxes in order to evaluate novel concurrent algorithms. In particular, they are instrumental in confirming how an algorithm can improve the performance of data structures even though the same algorithm negligibly boosts a particular application on a specific hardware or OS. Unfortunately, these micro-benchmarks are often developed specifically to illustrate the performance of one algorithm and are usually tuned for this purpose. More importantly, they are poorly documented as it is unclear whether updates comprise operations that return unsuccessfully without modifying, or whether the reported performance of a concurrent data structure are higher than the performance of its non-synchronized counterpart running sequentially.

Our contribution is the most extensive comparison of synchronization techniques. We focus on the performance of copy-on-write, mutual exclusion (e.g., spinlocks), read-copy-update, read-modify-write (e.g., compare-and-swap) and transactional memory to synchronize concurrent data structures written in Java and C/C++, and evaluated on AMD Opteron, Intel Xeon and UltraSPARC T2 multicore platforms. We also propose *Synchrobench*, an open source micro-benchmark suite written in Java and C/C++ for multi-core machines to help researchers evaluate new algorithms and synchronization techniques. Synchrobench is not intended to measure overall system performance or mimic a given application but is aimed at helping programmers understand the cause of performance problems of their structures. Its Java version executes on top of the JVM making it possible to test algorithms written in languages producing JVM-compatible bytecode, like Scala. Its C/C++ version allows for more control on the memory management.

Our evaluation includes 31 algorithms taken from the literature and summarized in Table 1. It provides a range of data structures from simple ones (e.g., linked lists) and fast ones (e.g., queues and hash tables) to sorted ones (e.g., trees, skip lists). These structures implement classic abstractions (e.g., collection, dictionary and set) but Synchrobench also features special operations to measure the reusability of the data structure in a concurrent library.

This systematic evaluation of synchronization techniques leads to interesting conclusions, including three main ones:

1. **Compare-and-swap is a double-edge sword.** Data structures are typically faster when synchronized exclusively with compare-and-swap than any other technique, regardless of the multicore machines we tested. However, the lock-free use of compare-and-swap makes the design of these data structures, and especially the ones with non-trivial mutations, extremely difficult. In

PPoPP'15, February 7–11, 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3205-7/15/02...$15.00
http://dx.doi.org/10.1145/2688500.2688501

Table 1. Algorithms of Synchrobench

#	Algorithm	Ref.	Synchronization	Strategy	Authors	Data structures	Language
1	Practical binary tree	[7]	lock	optimistic	Stanford U.	binary tree	Java
2	Contention-friendly tree	[15]	lock	pessimistic	INRIA&NICTA&U. Sydney	binary tree	Java
3	Logical ordering tree	[22]	lock	pessimistic	Technion & ETHZ	binary tree	Java
4	Lock-free tree	[24]	read-modify-write	optimistic	Toronto U.&FORTH&York U.	binary tree	Java
5	Fast lock-free tree	[58]	read-modify-write	optimistic	U. of Texas, Dallas	binary tree	C/C++
6	Speculation-friendly tree	[14]	transaction	optimistic	EPFL & INRIA	binary tree	C/C++
7	Transactional red black tree	[8]	transaction	optimistic	Sun (Oracle)	binary tree	Java&C/C++
8	Citrus tree	[4]	read-copy-update	optimistic	Technion	binary tree	C/C++
9	j.u.c.copyOnWriteArraySet	[32]	copy-on-write	optimistic	Oracle	dynamic array	Java
10	java.util.Vector	[1]	lock	pessimistic	Oracle	dynamic array	Java
11	ReusableVector	[34]	transaction	optimistic	EPFL & U. Sydney	dynamic array	Java
12	j.u.c.ConcurrentHashMap	[49]	lock	pessimistic	SUNY	hash table	Java
13	Michael's hash table	[54]	read-modify-write	optimistic	IBM	hash table	C/C++
14	Cliff Click's hash map	[12]	read-modify-write	optimistic	Azul Systems	hash table	Java
15	Contention-friendly hash table	[13]	read-modify-write	optimistic	INRIA & EPFL	hash table	Java
16	Resizable hash table	[51]	read-modify-write	optimistic	Lehigh U. & Tianjin U.	hash table	Java
17	Elastic hash table	[26]	transaction	optimistic	EPFL & UniNE	hash table	Java&C/C++
18	Lazy linked list	[41]	lock	optimistic	Sun&Brown U.&Rochester U.	linked list	Java&C/C++
19	Lock-coupling linked list	[47]	lock	pessimistic	Brown U. & MIT	linked list	Java&C/C++
20	j.u.Collections.synchronizedSet	[32]	lock	pessimistic	Oracle	linked list	Java
21	Harris' linked list	[39]	read-modify-write	optimistic	Cambridge U.	linked list	Java&C/C++
22	Reusable linked list	[34]	transaction	optimistic	EPFL & U. Sydney	linked list	Java
23	Elastic linked list	[27]	transaction	optimistic	EPFL & UniNE	linked list	Java&C/C++
24	j.u.c.ConcurrentLinkedQueue	[56]	read-modify-write	optimistic	IBM & Rochester U.	queue	Java
25	ReusableLinkedQueue	[34]	transaction	optimistic	EPFL & U. Sydney	queue	Java
26	Optimistic skip list	[43]	lock	optimistic	Sun&Brown U.&Rochester U.	skip list	C/C++
27	Fraser skip list	[30]	read-modify-write	optimistic	Cambridge U.	skip list	C/C++
28	j.u.c.ConcurrentSkipListMap	[49]	read-modify-write	optimistic	SUNY	skip list	Java
29	No hot spot skip list	[16]	read-modify-write	optimistic	INRIA & U. Sydney	skip list	Java&C/C++
30	Rotating skip list	[21]	read-modify-write	optimistic	U. Sydney	skip list	C/C++
31	Elastic skip list	[26]	transaction	optimistic	EPFL & UniNE	skip list	Java&C/C++

particular, we observed that there are only few existing full-fledged binary search trees using single-word compare-and-swap and we identified a bug in one of them.

2. **Transactions offer more consistent performance than locks.** We observed that optimistic locking techniques that consist of traversing the structure and locking before revalidating help reducing the number of locks used but also present great variations of performance depending on the considered structure and the amount of contention. Transactional memory provides more consistent performance, it features an efficient contention manager that avoids repeatedly restarting the same operations, a phenomenon observable in optimistic lock-based data structures that lack this contention management.
3. **Copy-on-write-based structures are still in their infancy.** Copy-on-write-like techniques (including read-copy-update) provide impressively high performance for read-only workloads but very low performance for update workloads. Software transactional memory, which is known to be badly-suited for heavily contended applications, may even handle conflicts more effectively than this technique. As most shared data structures get updated by multiple threads, more research effort is necessary for structures to fully exploit copy-on-write, probably by combining it with other synchronization techniques.

In Section 2 we describe the internals of Synchrobench and how the hardware and system can impact its results, and in Section 3 we indicate how Synchrobench is used. In Section 4 we present a thorough evaluation of all data structures of Table 1. In Section 5 we discuss the related work and we conclude in Section 6.

2. Synchronizing Algorithms

In this section, we present the different synchronizations, architectures, languages and structures of Synchrobench and mention how the hardware and operating system may affect its results.

2.1 Synchronization Techniques

Mutual exclusion is perhaps the most common technique to synchronize the concurrent accesses to shared data: acquiring a *lock* prevents any other thread from accessing the same lock until it gets released. Some implementations of locks like traditional OS implementations of mutex are not well-suited for multi-threaded environment as they trigger context switches whose overhead becomes predominant in an in-memory context. Test-and-set spinlocks are prone to the bouncing problem where acquiring a lock invalidates the cache of all threads reading the lock as they are waiting for its release. Ticket locks are sometimes detrimental [6], sometimes favorable [18]. Algorithms 1–3, 10, 12, 18–20 and 26 of Table 1 are lock-based and use macros to easily select a lock implementation.

Lock-freedom often refers to the non-blocking progress property, ensuring that the system as a whole always makes progress. This property, offered by some algorithms synchronized exclusively with *read-modify-write* primitives, like compare-and-swap (CAS), guarantees that one slow thread does not affect the overall system performance, making it appealing for modern heterogeneous machines. Algorithms 4, 5, 13–16, 21, 24 and 27–30 use exclusively CAS for synchronization. They all execute optimistically as they read the value at location x that may get stale by the time they execute a CAS to compare the value of x they observed against the current value of x. If the value has indeed been overridden then

the CAS fails and must be retried later on (these algorithms do not implement a lock with CAS).

Transactional memory [46] (TM) is appealing for simplifying concurrent programming. TM offers *transactions* used to delimit a region of code that should execute atomically. A dedicated contention manager detects race conditions to abort one of the conflicting transactions or let some transaction commit. Algorithms 6, 7, 11, 17, 22, 23, 25 and 31 are based on transactions. Synchrobench uses the standardized TM interface so that several TM algorithms can be evaluated in each of these algorithms, in particular it was successfully tested with 7 software TMs: $\mathcal{E}$-STM [27], SwissTM [23], LSA [61], NOrec [17], PSTM [34], TinySTM [25] and TL2 [20]. (In the reminder we evaluate transactions with E-STM and PSTM.)

Copy-on-write [32], in the context of concurrent libraries, is an optimistic execution strategy that stems from the immutability property in functional programming and consists of letting multiple threads of execution share the same reference to a memory location as long as this location is not updated. It makes reads particularly fast by adding overhead to the writes that have to create a copy. Algorithm 9 uses copy-on-write, it acquires a lock to create a copy of the entire array hence leading to performance losses upon updates but does not access any lock during read-only operations. Similarly, read-copy-update [53] provides a read-only fast path. It is popular for its use in the Linux kernel. The key idea is to let read-only operations access the data structures even when it is being modified. This is achieved through atomic updates so that pending readers observe the data before its modification while subsequent readers observe the data after modification. Algorithm 8 uses a user-level read-copy-update library to synchronize a binary search tree that accepts concurrent updates.

2.2 Data Structures

A linked list is a simple data structure organizing elements into a list. To avoid race conditions, the pessimistic lock-based technique of Algorithm 19, called hand-over-hand locking, consists of traversing the list by locking each element and then its successor before unlocking the element. Optimistic lock-based alternatives consist of traversing the list without locking, but locking only the part to be modified before validating that no data races occurred, as in Algorithm 18. The lock-free technique used by Algorithm 21 uses a logical deletion mark to make sure that a remove operation can be executed atomically with a single-word CAS. Algorithms 22 and 23 use different transaction models while Algorithm 20 wraps a linked list into a synchronizedSet wrapper.

A hash table data structure uses a hash function to map a key to a bucket implemented typically as a sorted linked list in which the associated value is stored to ensure constant access time. The lock-free hash table of Algorithm 13 uses the same logical deletion of the lock-free linked list [54]. The lock-based hash table is taken from the java.util.concurrent library of the JDK 7. The interesting aspect of the transaction-based hash table (Algorithm 17) is that it can be reused (i.e., extended with a new operation or existing operations can compose). Reusability is discussed in Section 4.6.

Skip lists are typically probabilistic data structures that provide an average complexity of $O(\log n)$. They organize nodes as towers that point towards their successor towers depending on their level, hence allowing to jump over small towers to obtain the desired complexity [60]. Algorithms 27 and 30 are concurrent skip lists that use a low-order bit as a logical deletion mark as in the lock-free linked list. In Java and in particular in the java.util.concurrent package of the JDK, this logical deletion mark was implemented using a special reference as in Algorithms 28 and 29. Other non-blocking skip lists [29, 63] that were proposed before multi-core became mainstream are not part of Synchrobench. One implements a dictionary [63] while the other always removes a logically deleted towers [29]. Both Algorithms 29 and 30 of Synchrobench are deterministic skip lists as opposed to classic ones.

Concurrent binary search trees usually offer a logarithmic access time complexity but are typically intricate as they require rotations to maintain the tree balanced, modifying potentially the root of the tree. Algorithm 4 uses a single-word CAS to implement a non-blocking binary search tree, however, it cannot rotate. A recent lock-based binary search tree (Algorithm 2) was proposed to rotate the tree when it gets unbalanced offering to check the presence of an element without acquiring any lock. Algorithm 3 combines this lock-free lookup with on-time removal during deletes. Algorithm 7 uses transactions to provide a concurrent red-black tree that rotates, however, aborting a rotation also means aborting the update operation that triggered it. Algorithm 6 copes with this issue by splitting each update into multiple transactions.

The queue is an abstraction that offers to enqueue and dequeue element in a first-in-first-out order and can be implemented using a linked list or an array. Algorithm 24 relies on a linked list structure and uses CAS while Algorithm 25 exploits transactions to offer a library that can be reused by someone who ignores the implementation details [34]. Again note that the list of algorithms we evaluate is far from being exhaustive, in particular efficient double ended queue algorithms [44, 64] have not been tested yet. Vectors are dynamically sized arrays (Algorithm 10) while copyOnWriteArraySet (Algorithm 9) is a wrapper from the java.util.concurrent package to convert any Set implementation into an array with fast read-only traversals.

2.3 Languages and Memory Management

Synchrobench is written in C/C++ and Java. This corresponds to two versions that are not intended to be directly compared but rather provide implementations highly tuned by their authors to run efficiently on the JVM or as native code. Both versions offer similar parameters to measure performance in terms of operations executed per second, the reusability cost in terms of the cost related to the use of composite operations, more detailed statistics in terms of the commit-abort ratio of transaction-based workloads, etc. It also offers similar ways of making the ratio of injected updates attempted or effective to avoid misinterpretation of results. The Java version allows, however, to choose whether iterations are part of the same JVM instance and to set a warmup. This warmup is a period where the benchmark runs before the statistics starts being collected and is particularly useful to let the JIT compiler of the server JVM optimize selected code regions.

Implementing memory reclamation correctly is a difficult task in unmanaged languages [55] especially when considering optimistic concurrency control. The crux of the problem lies in allowing threads to access data that are to be freed. In optimistic executions, for example when using CAS, some transactional memories or RCU, the garbage collector should deallocate memory locations only after these threads are done accessing the to-be-deleted data. The garbage collector has the difficult task of choosing the right time that minimizes efficiently memory usage without producing inconsistencies. Algorithms 8 and 13, and the C/C++ versions of Algorithms 21, 26 and 29 do not reclaim memory mainly because we did not implement it ourselves (Algorithms 13, 26 and 29) or because the source code from the authors did not include it (Algorithms 8 and 21). All other algorithms reclaim memory.

Memory management can also dramatically affect performance. It was recently shown in particular for hardware transactions that writing the same application in Java and C could lead to different performance results [59]. The authors did not use a memory manager optimized for concurrent execution for C-based programs and showed that the performance could suffer from this lack of optimization hence recommending the use of more advanced memory

allocators in concurrent applications. We actually observed that Java can be more efficient than C even when TCMalloc is used, as detailed in Section 4.4.

2.4 Hardware

Comparing synchronizations on different multicore machines is vital to ensure that hardware characteristics (e.g., hyperthreading [23]) are not the cause of the performance observed. Some architectures like our SPARC and x86-64 use distinct memory models (e.g., TSO-limited-RMO vs. x86-TSO). Implementing the same algorithm on both machines may thus require to come up with two implementations with different memory barriers.

While there exist some exceptions [31], algorithms should often be tuned to leverage appropriately the underlying specific hardware. For example, if two machines have different cache line sizes, false-sharing scenarios may occur for the same implementation on only one of the two machines. False-sharing may be dramatic if, for example, the structure representing a node uses a storage space that is not a multiple of the cache line size. In this case, a thread reading a node would typically store another data on one of the cache line partially occupied by the node. The modification to this latter data would invalidate the cache line shared by the node data even though the node data has not been modified, hence producing unnecessary off-chip traffic to re-fetch the memory.

Given that for an Intel Xeon like the one we used, the latency to access the L1D cache is 4 cycles whereas the latency to access the DRAM goes up to 400 cycles, such a false-sharing scenario may slow down accesses by two orders of magnitude. This problem can be easily addressed with a cache-padding technique in C [59] that is tuned to the cache line size of the machine, however, there are other subtleties to consider as even machines with the same architecture (x86-64) and cache line size (64 bytes), like our AMD and Intel ones, do not offer the same cache coherence (e.g., MOESI vs. MESIF).

2.5 Operating Systems

Operating systems also impact the performance for the way they place and migrate thread among cores or strands. Thread placement or *thread pinning* is known to greatly impact performance by either minimizing conflicts or maximizing sharing, typically on TLB and caches. In particular, thread pinning can have a higher impact on AMD Opteron than on Sun UltraSPARC [3] and in general the strategy differs depending on the programming language and the operating system used. For example, thread can be bound to Solaris lightweight processes (LWP) on the HotSpot JVM. When a LWP for a thread is created, the kernel assigns a thread to a locality group (or lgroup). In Solaris 10 the scheduler balances threads over dies, then over cores, then over pipelines whereas in Solaris 11 the scheduler groups the threads of a unique process on the same lgroup until the workload exhausts half of the resources of this lgroup.[1]

Since we used Solaris 10 (cf. Section 4), our Java application threads on UltraSPARC T2 are bound (by default by the HotSpot JVM) to lightweight processes (LWP) that are evenly scattered across the cores: for example, if Synchrobench uses 8 threads, one would be bound to each separate core. On Solaris, our C/C++ application threads are also scattered evenly among cores. On Linux, our Java and C/C++ application threads tend to be scattered across NUMA nodes both on AMD and Intel. In Section 4.7, we present the impact that explicit pinning of application threads may have on the performance of Synchrobench on Linux/AMD.

[1] `https://blogs.oracle.com/dave/entry/thread_placemet_policies_on_numa`.

3. The Synchrobench User Interface

In its simplest form, Synchrobench produces binaries/classes that correspond to data structures like arrays, binary trees, hash tables, linked lists, queues and skip lists, each written in C/C++ or Java and synchronized using copy-on-write, mutual exclusion, read-copy-update, read-modify-write or transactions. All these data structures export a simple default API with add, remove, contains/get that implement a set and a dictionary abstractions. This simple programming interface is extended with composite functions that are costly to execute but that guarantee that when provided as a library the algorithm can be reused by another programmer who does not necessarily understand the implementation internals, as discussed in Section 4.6.

Synchrobench runs a loop during ℓ milliseconds that executes a set of operations determined by multiple parameters. It starts spawning t threads that will all execute the same benchmark algorithm b by executing, in expectation, $u\%$ updates operations (whose effectiveness is given by f) and $100 - u\%$ read-only operations that take a key in range r as an argument, picked with the U distribution. Among the operations $u - a\%$ (with $a < u$) are basic updates (add/remove) and $100 - u - s\%$ (with $s < 100 - u$) are basic read-only operations (contains/get) in that they access a local part of the shared data structure whereas the remaining $a\%$ and $s\%$ operations are composite operations (e.g., move/containsAll) that comprise sub-operations accessing different locations of the benchmark. The parameters are described below.

- $t \in \mathbb{N}^*$, the number of application threads to be spawned. Note that this does not necessarily represent all threads, as it excludes JVM implicit threads and extra maintenance threads spawned by Algorithms 6, 15, 29 and 30.
- $i \in \mathbb{N}$, the initial size of the benchmark. This corresponds to the number of elements the data structure is initially fed with before the benchmark starts collecting statistics on the performance of operations.
- $r \in \mathbb{N}^*$, the range of possible keys from which the parameters of the executed operations are taken from, not necessarily uniformly at random. This parameter is useful to adjust the evolution of the size of the data structure.
- $u \in [0..100]$, the update ratio that indicates the amount of update operations among all operations (be they effective or attempted updates).
- $f \in \{0, 1\}$, indicates whether the update ratio is effective (1) or attempted (0). An effective update ratio tries to match the update ratio to the portion of operations that effectively modified the data structure by writing, excluding failed updates (e.g., a remove(k) fails because key k is absent).
- $A \in \{0, 1\}$, indicates whether the benchmark alternates between inserting and removing the same value to maximize effective updates. This parameter is important to reach a high effective update ratio.
- $U \in [0..100]$, the unbalance parameter that indicates the extent to which the workload is skewed towards smaller or larger values. This parameter is useful to test balanced structure like trees under unbalancing workloads.
- $d \in \mathbb{N}^*$, the duration of the benchmark in milliseconds.
- $a \in [0..100]$, the ratio of write-all operations that correspond to composite operations. Note that this parameter has to be smaller or equal to the update ratio given by parameter u.
- $s \in [0..100]$, the ratio of snapshot operations that scan multiple elements of the data structure. Note that this parameter has to be set to a value lower than or equal to $100 - u$.
- $W \in \mathbb{N}$, the warmup of the benchmark corresponds to the time in seconds it runs before the statistics start being collected, this option is used in Java to give time to the JIT compiler to compile selected bytecode to native code.

- $n \in \mathbb{N}^*$, the number of iterations run within the same JVM, this option is similar to the aforementioned warmup option except that performance statistics are collected during all iterations.
- $b \in [1..31]$, the benchmark to use. These numbers represent the indices listed in the left-column of Table 1.
- $x \in \mathbb{N}$, the alternative synchronization technique for the same algorithm. In the case of transactional data structures, this represents the transactional model used (relaxed or strong) while it represents the type of locks used in the context of lock-based data structures (optimistic or pessimistic).

4. Experimental Results

In this section we present a thorough comparison of the differently synchronized data structures against the performance of their corresponding non-synchronized data structures running sequentially. We used three different chip multiprocessors:

- A 64-way x86-64 AMD Opteron 6378 machine with 4 sockets of 16 cores each running at 1.4GHz Fedora 18, Java 1.7.0_09 IcedTea and gcc 4.7.2.
- A 32-way x86-64 Intel Xeon E5-2450 machine with 2 sockets of 8 hyperthreaded cores each running at 2.1GHz Ubuntu 12.04.4 LTS, Java 1.7.0_65 IcedTea and gcc 4.8.1.
- A 64-way UltraSPARC T2 with 8 cores of up to 8 hardware threads running at 1.165GHz Solaris 10, Java 1.7.0_05-b05 HotSpot and sparc-sun-solaris2.10-gcc 4.0.4

Each value (both in C/C++ and Java) is averaged over 5 iterations of the same benchmark running for 5 seconds. All our C/C++ benchmarks were compiled with the `-O3` optimization flag, the `-fno-strict-aliasing` flag, linked with the HP atomic operations library[2] and we used TCMalloc on the Intel machine, and the glibc v2.16 malloc on the AMD machine. All our Java benchmarks used the server VMs: in 32 bits for SPARC for efficiency, 64 bits for AMD to exploit additional registers and 64 bits for Intel, which may introduce some overhead. Below we report the performance of all algorithms presented in Table 1.

First, we describe the performance obtained on the Intel machine with most concurrent algorithms, then we discuss reusability of synchronizations and the impact of thread pinning on the performance. For each data structure, the performance of the bare code (without synchronization) running sequentially is actually represented as a horizontal black line to be used as a baseline. Figure 1 reports the performance results on the 32-way Intel Xeon machine of Algorithms 1–4, 6–8, 12–19, 21–23, 26–31 of Table 1. The list of parameters used for these benchmarks includes `-u[0..50]-f1-l5000-s0-a0-i[16384..65536]-r[32768..132072]-W0-b{1–4, 6–8, 12–19, 21–23, 26–31}`.

4.1 Read-Modify-Write

Interestingly, all CAS-based data structures provide relatively good performance under read-only workloads but they also handle contention better than counterparts synchronized with locks, transactions or RCU. Note that these data structures result from a large body of research work and are genuinely tuned for performance. More simplistic approaches are possible: for example comparing-and-swapping a reference from a version of the structure to a new copy as long as the original version was not modified during the copy is valid as well [47] but this would penalize performance. We discuss the difficulty of designing efficient CAS-based structures in Section 4.5 below.

Interestingly, the same CAS-based linked list algorithm (Algorithm 21) performs differently in Java and C/C++. Actually, the C/C++ version corresponds to Harris' pseudocode [39] whose contains operation physically removes the logically deleted nodes it traverses whereas the Java version corresponds to the Java variant presented in Herlihy and Shavit's book [47, Chapter 9] whose contains operation is wait-free and never removes. Although the C implementation uses a low order bit to store the mark directly in the reference, the Java `AtomicMarkableReference` stores the mark and reference separately, requiring an extra read to return the reference. Algorithm 13 is a variant of Michael's hash table that implements each bucket with a Harris' linked list: the aforementioned difference is less visible on hash tables than linked lists due to their lower time complexity.

4.2 Read-Copy-Update

RCU-based write operations are typically costly due to their copying strategy. In addition, RCU does not protect updates against each other and Algorithm 8 does not scale to 32 threads under contention precisely because it uses additional fine-grained locks to avoid race condition among concurrent updates. Also, note that Algorithm 8 is a recent RCU-based tree and one of the first to allow concurrent updates. It uses a user-level library but no memory reclamation or rebalancing technique, as opposed to the transactional tree (Algorithm 6) it is compared against. Although we did not report it here, we also tested the balanced Bonsai tree [11] with kernel-level RCU support, however, it does not tolerate concurrent updates and encapsulating each update operation in a critical section protected by a single global lock would make the algorithm significantly slower even under read-dominated workloads.

4.3 Locks

Under read-only workloads, the performance of lock-based algorithms depends typically on the number of locks acquired/released: the JDK hash table (Algorithm 12) uses lock-stripping so that one lock is acquired for multiple nodes whereas the lazy linked list (Algorithm 18) acquires and releases only 2 locks and the lock-coupling one (Algorithm 19) acquires $\Omega(n)$ locks per operation. In case of contention, locks tend to slow down the algorithm as they block. One exception is however the lock-based tree (Algorithm 2) because it postpones costly restructuring. Finally the lock-based skip list (Algorithm 26) is slower than other skip lists under high contention. The reason is that update operations optimistically execute before validating that the predecessor has not changed in the meantime, which typically occurs frequently under heavy contention, where performance drops due to repetitive restarts. In particular, to avoid a potentially infinite number of restarts under heavy contention, we implemented an exponential backoff waiting time between successive restarts. Note that all locks used here are Pthread spinlocks as we noticed generally higher performance than with mutexes/futexes, the study of other lock implementations is left for future work.

4.4 Language Effects

On the one hand, hash tables are fast regardless of the synchronization techniques due to its low contended nature. On the other hand, the performance of hash tables under read-only workloads depends heavily on the programming language used. Although previous results showed that the higher performance of Java can be explained by a thread-local memory allocator that is better suited than the malloc from the GNU C library [59], all C/C++-based algorithms presented in Figure 1 use the TCMalloc library based on fine-grained spinlocks to minimize thread contention.[3] We believe that Java better optimizes memory management (both placement and allocation) than our C/C++ implementations as to favor cache hits. The linked list, whose access complexity is linear, heavily de-

[2] `http://www.hpl.hp.com/research/linux/atomic_ops`.

[3] `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

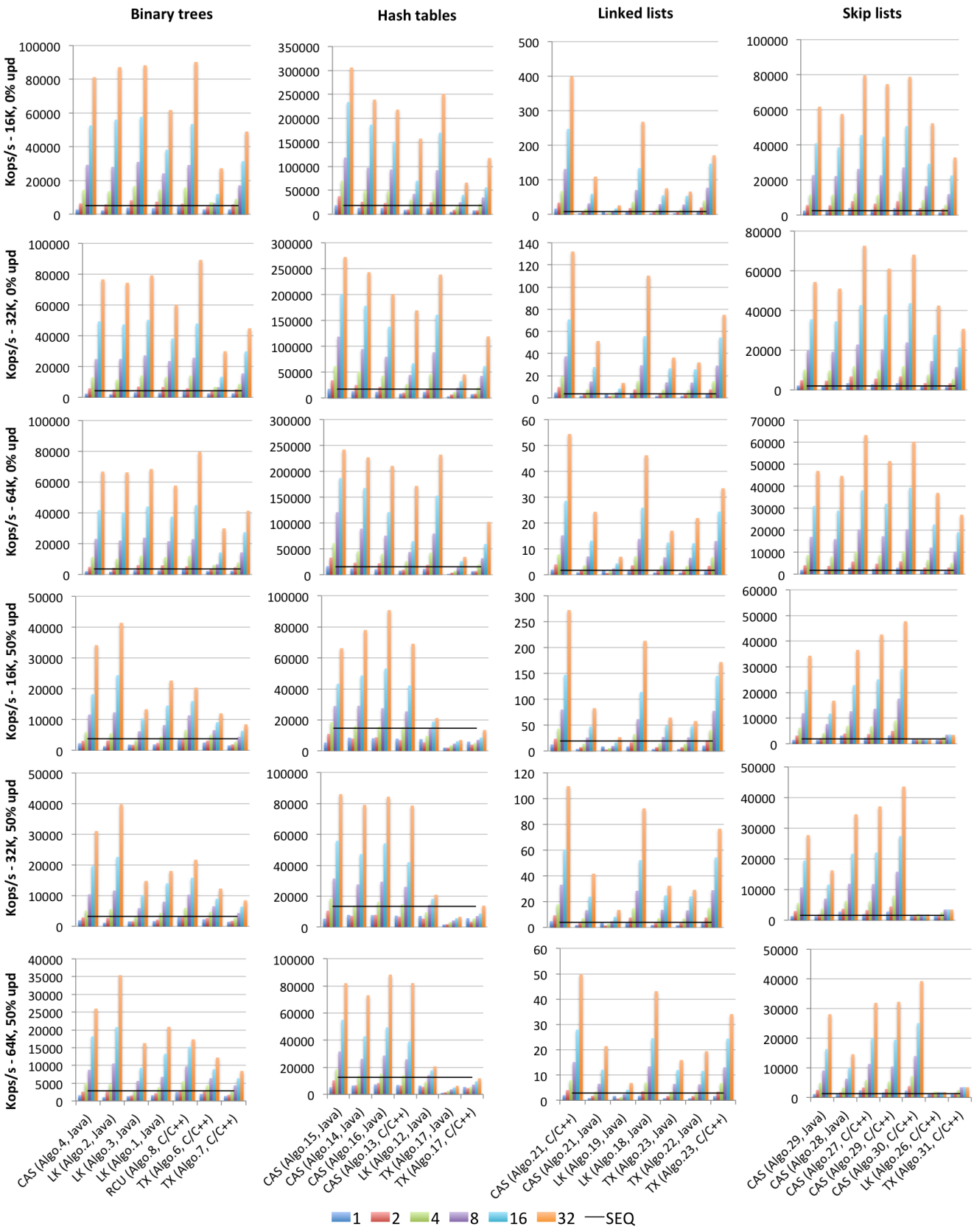

Figure 1. Synchrobench performance results for 1, 2, 4, 8, 16 and 32 threads on the 32-way Intel machine with data structures written in C/C++ and Java, and synchronized with read-modify-write (CAS), locks (LK), read-copy-update (RCU), and transactions (TX), and with sequential performance of non-synchronized data structures as the performance baseline (SEQ)

pends on the synchronization used so that Algorithm 19, performs particularly bad as it acquires/releases $\Omega(n)$ locks per operation.

In C/C++, volatile accesses cannot be reordered or optimized out within a single thread of execution. There is however no guarantee in a concurrent environment. In particular, a volatile access is not part of the atomic operations, as defined by C++11, and is not sequentially consistent. In Java, a volatile access is different as it synchronizes all cached copies of the volatile variable with the one in shared memory and creates a memory barrier (since Java 5). It can typically be useful in multi-threaded programs as it is ordered with most atomic operations via the happens-before relation.

4.5 On the Difficulty of Designing Correct Lock-Free Trees

We investigated various solutions that implement lock-free binary search trees and we realized the difficulty of implementing a full-fledged implementation. In particular, we identified four interesting concurrent tree algorithms [24, 30, 50, 58]. We could not integrate all of them into Synchrobench as some require a double-word CAS not always available [30], some source code showed unexpected behavior unveiling a bug in the implementation [58], and some proprietary code was not disclosed [50].

The difficulty in designing a tree that is lock-free stems from the difficulty of modifying multiple references atomically. Skip lists consist of towers linked to each other through successor pointers and in which each node points to the node immediately below it. They are often considered similar to trees because each node has a successor in the successor tower and below it, however, a major distinction is that the downward pointer is generally immutable hence simplifying the atomic modification of a node. This distinction is probably the reason why skip lists outperform trees under heavy contention as observed in Figure 1. We successfully tested Algorithm 4. Its Java implementation does not require explicit memory reclamation, which simplifies lock-free implementations [55]. Unfortunately, this tree is not balanced.

We also tested Algorithm 5 but observed a problem while benchmarking it. This algorithm implements a balanced tree and is written in C++, thus it aims at solving two difficult problems at once: memory reclamation and rotation. We obtained the code from the authors and observed a problem: after initializing the tree with a given size k, the benchmark would insert i nodes successfully and would remove j nodes successfully while the final size would be different from $k + i - j$. This was only noticeable under heavy contention, (i.e., $\leq$ 1024 elements and $\geq$ 20% updates). We contacted the authors who updated the code by declaring node children as volatile, however, the use of volatile is subtle (cf. Section 4.4) and we did not have enough time to evaluate the new version.

4.6 Reusability of Libraries Based on their Synchronization

Reusability is an appealing property encompassing compositionality and extensibility that lets a programmer, say Bob, reuse Alice's concurrent library without understanding how it is synchronized [33]. Compositionality guarantees that Bob can combine existing functions of Alice's concurrent library into a new one [40]. Extensibility guarantees that Bob can extend Alice's abstract data type with a new function [34]. Most concurrent libraries are not reusable because the synchronization techniques of Alice and Bob's functions risk to deadlock or experience data races: a programmer must first understand and sometimes modify the way the library is synchronized to be able to use it.

To illustrate reusability violations, consider Alice offering a Collection library where one can put and remove x and checks the presence of y, and Bob would like to reuse Alice's library (without understanding its synchronization internals) to implement a specific object with a function that puts x only if y is absent. If Bob

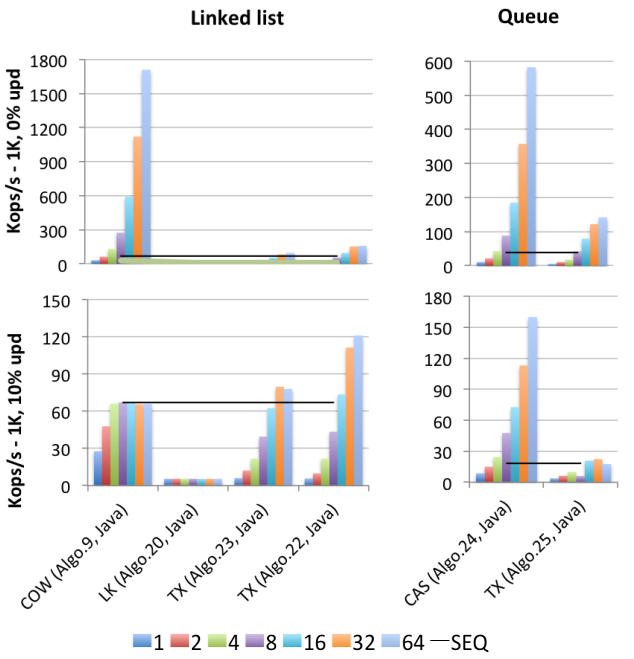

Figure 2. Synchrobench performance results for 1, 2, 4, 8, 16, 32 and 64 threads on the 64-way UltraSPARC-T2 machine with data structures written in Java and synchronized with read-modify-write (CAS), copy-on-write (COW), locks (LK) and transactions (TX) and with sequential data structures as the baseline (SEQ)

does not use a synchronization technique to prevent others from inserting x while he inserts y, then his function would violate its sequential specification. If both Alice and Bob use locks as their synchronization technique then deadlock may occur. If Alice and Bob use CAS in a lock-free manner, then Bob may not be able to implement his function without loosing concurrency. A similar problem arises if Bob would like a function containsAll that checks the presence of multiple elements in the Collection at some indivisible point of the execution. These two problems are at the heart of the inconsistent values returned by the GetOrAdd and AddOrUpdate functions in the C# System.Concurrent.Dictionary and the ConcurrentLinkedQueue.size() of the java.util.concurrent but can be easily solved with TM [34].

Figure 2 reports the reusability tests performed on SPARC with parameters -u[0..10]-f1-W20-l5000-s10-a0-i1024-r2048 -b{9, 20, 22–25} indicating that 10% operations are snapshot containsAll operations (s10). Algorithms 9 and 20 are lists encapsulated in the copyOnWriteArraySet (COW) and synchronizedSet wrappers (LK), respectively, whereas the methods of Algorithm 22 are encapsulated in relaxed transactions (TX). Algorithms 24 and 25 are the lock-free java.util.concurrent.ConcurrentLinkedQueue of the JDK (CAS) and the reusable alternative linked queue that synchronizes with PSTM (TX). Copy-on-write is significantly faster than TM without contention but slower than TM with only 10% effective updates. Note that the copy-on-write wrapper in Java copies all elements in a new array upon modification that makes its performance even more sensitive to contention than the read-copy-update technique. The lock-based wrapper of the linked list performs particularly bad as it acts as a coarse-grained lock. Finally, the lock-free queue (CAS) is much faster than the reusable queue (TX) but may succeed in unexpected cases: it does not guarantee that the containsAll returns false if the parameters are

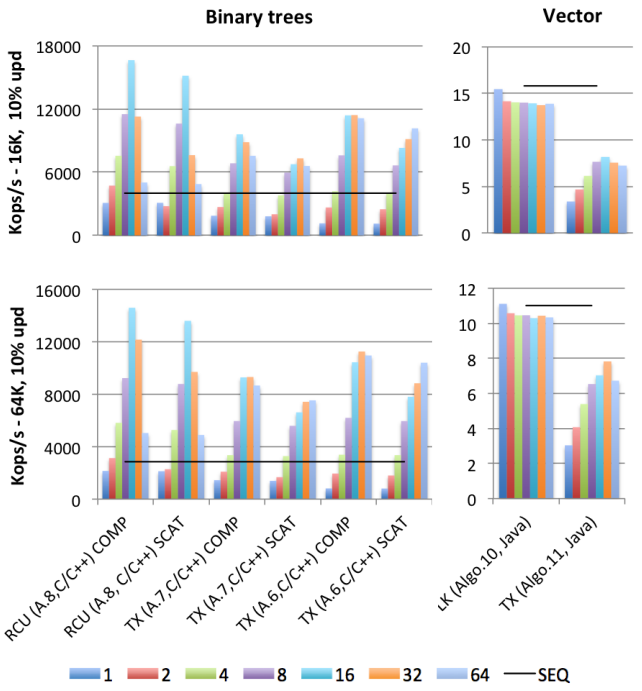

Figure 3. Synchrobench performance results for 1, 2, 4, 8, 16, 32 and 64 threads on the 64-way AMD machine with thread pinning compact (COMP) and scatter (SCAT) of Java and C/C++ data structures synchronized with locks (LK), read-copy-update (RCU) and transactions (TX), and with sequential performance of non-synchronized data structures as the performance baseline (SEQ)

not all present at some point of the execution (as we would have to recode other methods or annihilate concurrency). This performance difference is the price to pay for reusability.

4.7 Effect of Contention and Thread Pinning

Figure 3 depicts the performance impact of thread pinning and contention on AMD. The list of parameters is -u10-f1-l5000-s0-a0-i[16384..65536]-r[32768..132072]-W20-b{6–8, 10, 11}, indicating 10% effective updates, a duration of 5 seconds, no snapshot/write-all operations, 16384 (resp. 65536) values taken out of 32768 (resp. 132072), a 20 s warmup, running Algorithms 6–8, 10 and 11. Although we do not report CAS performance, we observed that CAS generally outperforms others. On the left we evaluate 3 binary trees using 2 explicit thread pinning schedulers for each: the first one is compact (COMP) and places threads on the same NUMA nodes first whereas the second one is scatter (SCAT) and places threads on separate NUMA nodes first. We observed that the compact scheduler favors rapid peak performance whereas scatter favors scalability except for the RCU-based program that suffers contention more than transaction-based programs. On the right we observe that vector data structures be they synchronized with locks or transactions are outperformed by the bare sequential vector running on a single thread, at only 10% effective updates, confirming recent observations [34].

5. Related Work

There exist macro-benchmarks and micro-benchmarks to test performance of hardware features or specific data structures but there is no tool to compare the impact of the synchronization choices.

Transaction benchmark suites. Cao Minh et al. proposed STAMP [8] a benchmark suite that offers 8 application-like benchmarks and propose default parameters for each of these applications. It is dedicated to evaluate transactional memory but does not evaluate other synchronization techniques. In particular, its vacation application that uses a transactional red-black tree cannot be made lock-free easily. STAMP was ported in Java by the University of California, Irvine and is available as part of Deuce [48] or ByteSTM [57]. STAMP was instrumental in showing that algorithms written in C could scale well due to adequate padding whereas the Java ones could scale due to the thread-locality of the memory allocator [59].

Guerraoui et al. developed STMBench7 [36], an extended adaptation of the OO7 database benchmark for software transactional memory. Besides transactions one can specify three different workloads and a coarse-grained lock but no fine-grained locks or other synchronizations. Zyulkyarov et al. developed Wormbench [68] to represent applications by a series of complex atomic operations, but provided limited control to the user. Harmanci et al. designed TMunit [37], a framework to write reproducible transactional tests using a high level language that was later extended to support 6 TM libraries [38]. More recently, Zyulkyarov et al. [69] proposed a profiling and visualization tool for very large pre-existing transaction-based applications to easily identify its bottleneck data structures.

Micro-benchmark suites. In the last decade micro-benchmarks have been instrumental in the understanding of performance of transactional memory. In particular, the red-black trees proposed by Dice, Shavit and Shalev in C and by Herlihy in Java (Algorithm 7 of Table 1) have been extensively used to evaluate the performance of various transactional memory algorithms [14, 20, 23, 42, 45, 67]. In addition, Felber et al. released a linked list, a skip list and a bank benchmark written in C to measure the performance of TinySTM [25] from which Synchrobench was initially derived. Deuce [48] offers an integer set IntSet and the bank benchmark written in Java to test the performance of TL2, $\mathcal{E}$-STM and LSA.

Fraser [30] developed the lock-free library under the BSD licence to benchmark three types of data structures, including trees and skip lists with various-size compare-and-swaps. This library was carefully tuned to architectures SPARC, x86, IA_64 and PowerPC. It was proposed more than a decade ago and does not include the latest achievements in concurrent data structures. The only data structure in common with ours is Fraser's skip list.

Sundell and Tsigas [62] proposed the NOBLE library that offers queue, stack, linked list, register and snapshot objects to be tested on SPARC, Mips and x86-Intel. Unfortunately, it does not offer logarithmic data structures that reduce the contention of stacks/queues/registers and offer better asymptotic complexity than linked lists. Nbds[4] offers a transactional dictionary, Fraser's skip list and a port in C of Cliff Click's hash table. Liblfds[5] is a library with lock-free data structures, however, the provided binary tree does not support delete operations and the skip list only supports logical deletions.

Java concurrent libraries and benchmarks. Apart from C/C++, there are also Java-specific concurrent libraries that can be used for benchmarking. The Guava library [5] is a Java-based library that offers collections, caching, primitive support, string processing and I/O tools. The JSR 166 specification group[6] designed the java.util.concurrent package that provides an invaluable set of low-level atomic tools as well as various lock-based and lock-free data structures. It does not leverage transactions but offers alternative

[4] https://code.google.com/p/nbds/.

[5] http://www.liblfds.org.

[6] http://gee.cs.oswego.edu/dl/concurrency-interest/.

copyOnWrite wrappers to make some objects reusable. In particular, it offers three data structures also provided in Synchrobench: Algorithms 9, 24 and 28 of Table 1. SPECjbb is a benchmark based on the emulation of a three-tier client/server system using an in-memory database backed by Java collections to stress the JVM. It performs BigDecimal computations, XML processing and accesses collections. Unfortunately these collections are all thread-local and not accessed concurrently. SPECjbb code has to be modified to test a data structure algorithm under contention [9, 16].

Low-level benchmark suites. Ferdman et al. proposed CloudSuite [28] to illustrate the mismatch between scale-out applications and modern processors. They studied typically large distributed applications, including data stores, map reduce, streaming, solvers and web services and monitored the behavior at a fine granularity using hardware profilers like VTune. They report on the cache-misses, the instruction-level and data-level parallelism, and on-chip/off-chip traffic. These tools could complement Synchrobench.

The Intel Threading Building Blocks (TBB) library[7] provides concurrent data structures written in C++. The TBBench was proposed to specifically measure the performance of this library when compiled with different compilers [52]. This is orthogonal to our work as it does not aim at comparing the same data structure with different synchronization techniques. Wicht [66] proposed an empirical evaluation on a multicore machine of four trees and a skip list implemented in C++. Recently, Cederman et al. showed that lock-free structures may be fairer and more efficient than lock-based ones on x86-64 [10] by considering constant-time data structures. Finally, David et al. [18] proposed a benchmark in C for locks on x86-64, SPARC and Tilera architectures but did not explore other programming languages or lock-free synchronizations.

Synchrobench in the literature. First, Synchrobench helped measuring the performance of relaxed, strong and distributed TM models. Felber et al. used Synchrobench (and its Algorithm 23) to illustrate the performance of relaxed transactions on pointer-based structures [27]. Dragojević et al. used Synchrobench (and its Algorithms 17, 23 and 31) to evaluate the scalability of software transactional memory with compiler support on SPARC and x86 [23]. Gramoli et al. used Synchrobench (Algorithms 17 and 23) to compare the performance of distributed TM on multicore machines and manycore machines without cache-coherence [35].

Second, preliminary versions have been used to test the performance of trees, deterministic and probabilistic skip lists. Crain et al. used Synchrobench and Algorithms 6 and 7 (resp. Algorithm 2) to show that reshaping a classic binary search tree is beneficial to optimistic synchronization techniques [14] (resp. pessimistic techniques [15]). Crain et al. [16] and Dick et al. [21] used Algorithms 27, 29 and 30 to show that deterministic skip lists could outperform probabilistic ones. Umar et al. [65] used Algorithms 6 and 7 to show that a locality-aware k-ary tree could achieve higher performance than concurrent binary trees. Alistarh et al. used Synchrobench to compare the performance of their SprayList [2]. Recently, Drachsler et al. [22] used Algorithm 2 of Synchrobench to show that deleting immediately nodes does not necessarily induce high contention whereas Arbel and Attiya [4] used Algorithm 26 to show that their RCU-based tree was offering higher performance, an observation that we confirmed in Section 4. David et al. are currently testing the scalability of Synchrobench structures [19].

6. Conclusion

Our work encompasses multiple hardware platforms, 31 data structure algorithms from the recent literature to measure the performance one could expect from 5 synchronization techniques, hence offering the most extensive comparison of synchronization techniques. Our conclusion is threefold. First, CAS allows to develop the fastest algorithms for multicores at the expense of great complexity. Second, TM is a mature synchronization technique offering performance that are more consistent across update ratios and programs than locks. Finally, copy-on-write and read-copy-update are more sensitive to contention than other techniques, yet they help achieving high performance under read-only workloads. More research is thus necessary to develop efficient RCU-based contended structures.

Availability

Synchrobench is publicly available online at `https://github.com/gramoli/synchrobench`.

Acknowledgments

I wish to thank Tim Harris, the shepherd of this paper, the anonymous reviewers for their comments and the persons who gave feedback and helped improve Synchrobench including Tyler Crain, Dave Dice, Ian Dick, Alexandar Dragojević, Pascal Felber, Rachid Guerraoui, Maurice Herlihy, Konrad Lai, Doug Lea, Patrick Marlier, Mark Moir, Di Shang and Vasileios Trigonakis. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

[1] JSE-7. `http://docs.oracle.com/javase/7/docs/api/`.

[2] D. Alistarh, J. Kopisky, J. Li, and N. Shavit. The SprayList: A scalable relaxed priority queue. Technical Report TR-2014-16, MSR, 2014.

[3] J. Antony, P. P. Janes, and A. P. Rendell. Exploring thread and memory placement on NUMA architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/Hypertransport. In *HiPC*, pages 338–352, 2006.

[4] M. Arbel and H. Attiya. Concurrent updates with RCU: Search tree as an example. In *PODC*, pages 196–205, 2014.

[5] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: a dialect of Java without data races. In *OOPSLA*, pages 382–400, 2000.

[6] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium, Ottawa, Canada*, 2012.

[7] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *PPoPP*, pages 257–268, 2010.

[8] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, pages 35–46, 2008.

[9] B. D. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun. Transactional collection classes. In *PPoPP*, pages 56–67, 2007.

[10] D. Cederman, B. Chatterjee, N. Nguyen, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas. A study of the behavior of synchronization methods in commonly used languages and systems. In *IPDPS*, pages 1309–1320, 2013.

[11] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable address spaces using RCU balanced trees. In *ASPLOS*, pages 199–210, 2012.

[12] C. Click. A lock-free hash table, 2007. `http://www.azulsystems.com/events/javaone_2007/2007_LockFreeHash.pdf`.

[13] T. Crain, V. Gramoli, and M. Raynal. A contention-friendly methodology for search structures. Technical Report RR-1989, INRIA, 2012.

[14] T. Crain, V. Gramoli, and M. Raynal. A speculation-friendly binary search tree. In *PPoPP*, pages 161–170, 2012.

[15] T. Crain, V. Gramoli, and M. Raynal. A contention-friendly binary search tree. In *Euro-Par*, pages 229–240, 2013.

[16] T. Crain, V. Gramoli, and M. Raynal. No hot spot non-blocking skip list. In *ICDCS*, pages 196–205, 2013.

[7] `https://www.threadingbuildingblocks.org`.

[17] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: streamlining STM by abolishing ownership records. In *PPoPP*, pages 67–78, 2010.

[18] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *SOSP*, pages 33–48, 2013.

[19] T. David, R. Guerraoui, and V. Trigonakis. Asynchronized concurrency: the secret of scaling concurrent search structures. In *ASPLOS*, 2015. To appear.

[20] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, pages 194–208, 2006.

[21] I. Dick, A. Fekete, and V. Gramoli. Logarithmic data structures for multicores. Technical Report 697, University of Sydney, 2014.

[22] D. Drachsler, M. Vechev, and E. Yahav. Practical concurrent binary search trees via logical ordering. In *PPoPP*, pages 343–356, 2014.

[23] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui. Why STM can be more than a research toy. *Commun. ACM*, 54(4):70–77, 2011.

[24] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *PODC*, pages 131–140, 2010.

[25] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*, pages 237–246, 2008.

[26] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. Technical Report LPD-REPORT-2009-002, EPFL, 2009.

[27] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *DISC*, pages 93–108, 2009.

[28] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Quantifying the mismatch between emerging scale-out applications and modern processors. *TOCS*, 30(4):15:1–15:24, 2012.

[29] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *PODC*, pages 50–59, 2004.

[30] K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University, September 2003.

[31] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, page 285, 1999.

[32] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. *Java Concurrency in Practice*. Addison-Wesley, 2005.

[33] V. Gramoli and R. Guerraoui. Democratizing transactional programming. *Commun. ACM*, 57(1):86–93, 2014.

[34] V. Gramoli and R. Guerraoui. Reusable concurrent data types. In *ECOOP*, pages 182–206, 2014.

[35] V. Gramoli, R. Guerraoui, and V. Trigonakis. TM2C: A software transactional memory for many-cores. In *EuroSys*, pages 351–364, 2012.

[36] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: a benchmark for software transactional memory. In *EuroSys*, pages 315–324, 2007.

[37] D. Harmanci, P. Felber, V. Gramoli, and C. Fetzer. TMunit: Testing software transactional memories. In *4th ACM SIGPLAN Workshop on Transactional Computing*, 2009.

[38] D. Harmanci, V. Gramoli, P. Felber, and C. Fetzer. Extensible transactional memory testbed. *J. of Parallel and Distributed Computing*, 70(10):1053–1067, March 2010.

[39] T. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, pages 300–314, 2001.

[40] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP*, pages 48–60, 2005.

[41] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, and N. Shavit. A lazy concurrent list-based set algorithm. *Parallel Processing Letters*, 17(4):411–424, 2007.

[42] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *PPoPP*, pages 207–216, 2008.

[43] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A simple optimistic skiplist algorithm. In *SIROCCO*, pages 124–138, 2007.

[44] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, 2003.

[45] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, 2003.

[46] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[47] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kauffman, 2008.

[48] G. Korland, N. Shavit, and P. Felber. Deuce: Noninvasive software transactional memory. *Transactions on HiPEAC*, 5(2), 2010.

[49] D. Lea. JSR-166 specification request group. `http://g.oswego.edu/dl/concurrency-interest`.

[50] J. J. Levandoski and S. Sengupta. The BW-Tree: A latch-free B-tree for log-structured flash storage. *IEEE Data Eng. Bull.*, 36(2):56–62, 2013.

[51] Y. Liu, K. Zhang, and M. Spear. Dynamic-sized nonblocking hash tables. In *PODC*, pages 242–251, 2014.

[52] A. Marowka. TBBench: A micro-benchmark suite for Intel threading building blocks. *JIPS*, 8(2):331–346, 2012.

[53] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-copy update. In *AUUG*, 2001.

[54] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82, 2002.

[55] M. M. Michael. The balancing act of choosing nonblocking features. *Commun. ACM*, 56(9):46–53, 2013.

[56] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.

[57] M. Mohamedin, B. Ravindran, and R. Palmieri. ByteSTM: Virtual machine-level Java software transactional memory. In *COORDINATION*, pages 166–180, 2013.

[58] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In *PPoPP*, pages 317–328, 2014.

[59] R. Odaira, J. G. Castaños, and T. Nakaike. Do C and Java programs scale differently on hardware transactional memory? In *IISWC*, pages 34–43, 2013.

[60] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33, June 1990.

[61] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *DISC*, pages 284–298, 2006.

[62] H. Sundell and P. Tsigas. NOBLE: A non-blocking inter-process communication library. Technical report, Chalmers University of Technology, March 2002.

[63] H. Sundell and P. Tsigas. Scalable and lock-free concurrent dictionaries. In *SAC*, pages 1438–1445. ACM, 2004.

[64] H. Sundell and P. Tsigas. Lock-free deques and doubly linked lists. *J. Parallel Distrib. Comput.*, 68(7):1008–1020, 2008.

[65] I. Umar, O. J. Anshus, and P. H. Ha. DeltaTree: A practical locality-aware concurrent search tree. Technical Report 2013-74, University of Tromso, Norway, Oct. 2013.

[66] B. Wicht. Binary trees implementations comparison for multicore programming. Technical report, Switzerland HES-SO University of applied science, 2012.

[67] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. S. Lee. Kicking the tires of software transactional memory: why the going gets tough. In *SPAA*, pages 265–274, 2008.

[68] F. Zyulkyarov, A. Cristal, S. Cvijic, E. Ayguade, M. Valero, O. Unsal, and T. Harris. Wormbench: A configurable workload for evaluating transactional memory systems. In *MEDEA*, pages 61–68, 2008.

[69] F. Zyulkyarov, S. Stipic, T. Harris, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. Profiling and optimizing transactional memory applications. *International Journal of Parallel Programming*, 40(1):25–56, 2012.