Concurrent Search Data Structures Can Be Blocking and Practically Wait-Free

Tudor David
EPFL
tudor.david@epfl.ch

Rachid Guerraoui
EPFL
rachid.guerraoui@epfl.ch

ABSTRACT

We argue that there is virtually no practical situation in which one should seek a "theoretically wait-free" algorithm at the expense of a state-of-the-art blocking algorithm in the case of search data structures: blocking algorithms are simple, fast, and can be made "practically wait-free".

We draw this conclusion based on the most exhaustive study of blocking search data structures to date. We consider (a) different search data structures of different sizes, (b) numerous uniform and non-uniform workloads, representative of a wide range of practical scenarios, with different percentages of update operations, (c) with and without delayed threads, (d) on different hardware technologies, including processors providing HTM instructions.

We explain our claim that blocking search data structures are practically wait-free through an analogy with the birthday paradox, revealing that, in state-of-the-art algorithms implementing such data structures, the probability of conflicts is extremely small. When conflicts occur as a result of context switches and interrupts, we show that HTM-based locks enable blocking algorithms to cope with them.

1. INTRODUCTION

With multi-core architectures being ubiquitous, concurrent data structures have become performance-critical components in many widely-used applications and software systems. In particular, *search data structures* are heavily used by numerous popular systems, such as Memcached [42], RocksDB [15], LevelDB [20], MySQL [50], MongoDB [49], MonetDB [48] and the Linux kernel [40]. Basically, search data structures are implementations of the *set* abstraction: they provide operations to search for a particular element, to insert, and to remove an element. The most common examples include *linked lists*, *skip lists*, *hash tables*, and *binary-search trees (BSTs)*.

However, despite the fact that a large body of work has been dedicated to concurrent search data structure (CSDS) algorithms [6, 9, 11, 13, 14, 18, 23, 24, 28, 32, 37, 39, 43, 51, 53, 57, 58], their design and implementation remains an onerous task. The difficulty lies in providing implementations that are both correct, i.e., linearizable [31], as well as efficient. Essentially, an ideal concurrent data structure (i) is easy to design, implement and reason about, (ii) provides high aggregate throughput and scalability, and (iii) ensures limited latency delays due to concurrency for all requests.

In theory, *wait-free* algorithms [27, 30] are the only ones that satisfy requirement (iii). These algorithms prevent known issues of locking (i.e., convoying, deadlocks, priority inversions), without the risk of starvation. In essence, a wait-free algorithm guarantees that every thread completes its operation in a finite number of its own steps, thus promising limited latency, even under high contention.

Nevertheless, despite the significant amount of research dedicated to the design of wait-free CSDSs [2, 16, 27, 37, 38, 57, 58], these algorithms exhibit low throughput, roughly half of that of a state-of-the-art *blocking* or *lock-free* search data structure (as we show experimentally in the paper). Essentially, the reason for the difference in throughput between wait-free algorithms on the one hand, and lock-free and blocking algorithms one the other hand, is related to the amount of concurrency-related information that needs to be associated with the data. While in the case of lock-free and blocking algorithms, data and concurrency-related information can be efficiently manipulated in an atomic manner, this is not the case for wait-free algorithms. This results in more pointer chasing and thus slower traversals of the structures.

Lock-free algorithms have been shown to provide high aggregate throughput in the case of CSDSs [9] (and have been proven to ensure wait-freedom with high probability for a large class of schedulers [3]). However, they are difficult to design and implement correctly [21], with memory management making the task even more complex [7, 44].

Blocking algorithms are considered to be significantly easier to implement and use, mainly because data can only be modified in critical sections protected through mutual exclusion mechanisms. It is thus natural to ask whether it is possible to have blocking CSDS algorithms that provide high aggregate throughput, but also ensure that on realistic workloads, no individual request is significantly delayed due to contention, i.e., these algorithms are *practically wait-free*.

We believe this question can only be answered empirically. To this end, we conduct the most exhaustive study of blocking CSDSs to date. We deploy a wide variety of state-of-the-art CSDS algorithms, on the latest hardware technology. In particular, we enable locks to use Hardware Transactional Memory (HTM) in order to prevent threads from holding locks while not scheduled, which is a rather novel use-case for HTM. Our evaluation is extensive both in the metrics we collect, as well as in the scenarios we study: we consider numerous uniform and non-uniform workloads, representative of a wide range of practical scenarios. We also test different

SPAA'16, July 11–13, 2016, Pacific Grove, California, USA.

DOI: http://dx.doi.org/10.1145/2935764.2935774

data structures with varying sizes and percentages of update operations, with and without delayed threads.

Our results indicate that state-of-the-art blocking implementations of search data structures behave practically wait-free in all realistic scenarios. Even in scenarios of extreme contention, requests experience acceptable delays with blocking CSDSs. The usage of HTM reveals effective in the face of frequent context switches and other interrupts.

The main conclusion we draw from our study is that there is virtually no practical situation in which one needs to seek an algorithm providing a strong theoretical progress guarantee at the expense of a state-of-the-art blocking algorithm in the case of search data structures: blocking algorithms provide good throughput and latency, even under high contention, and are much simpler than their non-blocking counterparts.

Our explanation for this fact is that state-of-the-art blocking CSDSs are specifically devised to minimize the probability of high contention for any memory address. This reason is different from the perspective taken by recent work [3], namely that if thread accesses are scheduled stochastically, lock-free algorithms behave in a wait-free manner even if there is high contention for a particular memory address. Our conclusion suggests instead that the main reason for non-wait-free CSDS algorithms, in the theoretical sense, to behave wait-free in the practical sense is the small probability of actual contention for a surprisingly wide spectrum of workloads. This follows from the fact that, in the case of state-of-the-art CSDSs, only short portions of the update operations may cause conflicts when occurring concurrently. State-of-the-art blocking algorithms simply go through the nodes and follow the next pointers, and only lock the area which needs to be modified. Indeed, the time needed to traverse the structure in order to reach the point where the operation needs to actually be performed generally dominates the total execution time. In addition, problematic scenarios, such as large delays that may occur due to context switches or interrupts can often be handled efficiently using HTM technologies available in commodity processors.

The probability of conflicts can be modeled using variations of the *birthday paradox*. We show that in common workloads, this probability is below 1%, with the probability of repeated conflicts for the same request being much smaller. Even in contended situations with a high number of conflicts, it is extremely rare that a wait-free algorithm outperforms state-of-the-art blocking algorithms.

Three remarks are, however, in order. (1) We do not claim that blocking search data structure algorithms behave in a wait-free manner under *every* conceivable scenario. We could create a scenario where (i) the data structure has a small number of nodes, (ii) these are accessed repeatedly, (iii) by a very large number of concurrent threads, and (iv) with a high update rate, and in which latency would indeed suffer.[1] (2) Our conclusion only applies to search data structures. Indeed, we show in the paper that in the case of data structures which have the potential of inducing much more contention on a small number of memory addresses, such as queues and stacks, a blocking algorithm is not ideal. (3) We do not claim that non-blocking algorithms cannot offer the same performance as blocking ones. In fact, several lock-free algorithms are known to provide performance comparable to blocking algorithms. Rather, we show that state-of-the-art blocking algorithms, which are often much simpler to design and implement than their non-blocking counterparts, provide no disadvantage.

[1]We do however claim this does simply not occur in the vast majority of practical situations. Practitioners, which often have some knowledge about their workloads and system requirements, can use our work to decide when blocking implementations are sufficient.

The rest of the paper is structured as follows. We discuss CSDSs in Section 2. We present our experimental methodology and settings in Section 3. In Section 4, we show that blocking CSDSs provide high throughput and scalability, while in Section 5 we perform an extensive evaluation of the degree to which blocking CSDS algorithms are practically wait-free. Section 6 discusses the analogy with the birthday paradox. We examine the extent to which our conclusions apply to structures other than CSDSs in Section 7. Section 8 presents related work. We conclude the paper in Section 9.

2. CONCURRENT SEARCH DATA STRUCTURES: FROM THEORY TO PRACTICE

In this section, we look at the metric types indicative of the performance of a concurrent algorithm: coarse-grained and fine-grained. Moreover, we clarify why theoretically wait-free algorithms fail to perform well in the former, and explain what we mean by practical wait-freedom in the context of blocking algorithms.

2.1 Performance and progress

The performance of concurrent algorithms is usually evaluated according to two main types of metrics:

- *coarse-grained performance metrics*: these capture the overall performance of the system, and usually include metrics such as throughput (the number of requests completed system-wide per unit of time), average latency (average duration of an operation), scalability (variation in throughput and latency as more threads are added to the system), and fairness (the difference in observed throughput and average latency between threads);

- *fine-grained performance metrics*: these metrics capture the behavior of individual requests; these include for example metrics such as latency distribution, variability, outliers, as well as other algorithm-specific metrics. Such metrics might be useful, for instance, to identify the quantity of particularly slow requests in a system.

An ideal algorithm would provide good results for both these metric types. However, optimizing for one of them can have a negative impact on the other. For example, attempting to maximize system throughput might result in an algorithm with a large variability in the operation latencies. Ensuring bounded latencies for all requests is likewise likely to limit throughput.

One of the main knobs used to adjust the relative importance of these metrics is the progress guarantee provided by the algorithm. The vast majority of published algorithms are either (i) *blocking*, (ii) *lock-free*, or (iii) *wait-free* [30].

- *Blocking* algorithms ensure mutually exclusive access to (parts of) the data structure using, for example, mechanisms such as locks. More broadly, threads have to explicitly release resources before others can use them, thus potentially preventing the progress of other threads indefinitely.

- *Lock-free* algorithms ensure that at least one thread in the system is able to make progress at any point in time.

- *Wait-free* algorithms ensure that every thread in the system will eventually complete its operation.

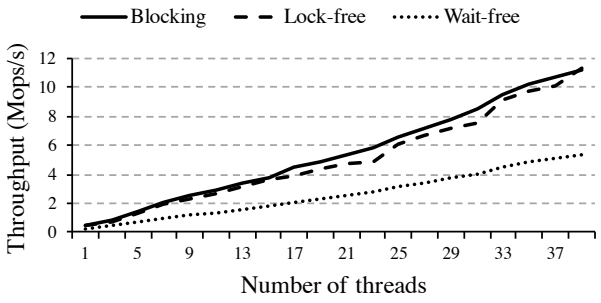

Figure 1: The throughput of blocking, lock-free and wait-free linked lists of size 1024, with 10% of the operations being updates.

As we pointed out, blocking algorithms are generally simple. The downsides are the potential of several threads being indefinitely delayed by a slow thread holding a lock, deadlocks, livelocks, convoying or priority inversions. Lock-free algorithms present the risk of some threads starving. In theory, wait-freedom is the most desirable property for a concurrent algorithm. It ensures that no request is indefinitely delayed due to contention.

A significant amount of effort has been dedicated in recent years [16, 37, 38, 57, 58] to dispel the belief that wait-free algorithms provide low throughput in practice [17, 30]. In the context of search data structures however, such algorithms still lag significantly behind alternatives providing weaker progress guarantees in terms of system-wide throughput, as we will discuss below.

2.2 A closer look at wait-free CSDS algorithms

A search data structure contains a set of elements, and allows access to each of them, regardless of their position in the data structure. These structures store elements of arbitrary sizes, indexed using keys, and are usually implemented as linked data structures. They have a simple base interface, consisting of three operations:

- *get(k)* returns the value associated with key k in case such an entry exists, or returns false in case the entry is not present;
- *put(k,v)* inserts a key-value pair in case an entry for key k is not present, or returns false otherwise;
- *remove(k)* removes the entry corresponding to key k in case it exists in the data structure, or returns false otherwise.

Given the practical importance of these structures, a lot of effort has been dedicated to their efficient concurrent implementations.

In the following, we illustrate the current difference in throughput between a state-of-the-art wait-free algorithm [58], a state-of-the-art blocking algorithm [24] and a lock-free algorithm [23] for a linked list (for space limitations, we only give results for the linked list; we also experimented with a skip-list, a BST and a hash table). We use a recent 20-core (40 hardware threads) Intel server. We depict the throughput of a linked list with 1024 elements and 10% updates (5% inserts, 5% removes) as we increase the number of threads from 1 to 40.

As conveyed in Figure 1, the throughput of the wait-free algorithm is around 50% of its blocking and lock-free counterparts. This trend pertains to other CSDSs as well (skip lists, BSTs and hash-tables[2]).

[2]In the case of a hash table with average occupancy per bucket equal to 1, and hence on average no linked components, the wait-free algorithm is only around 33% slower. For the others, it is also 50%.

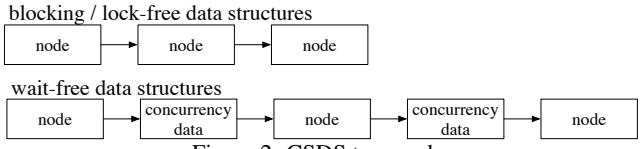

Figure 2: CSDS traversals.

The main reason for the difference in throughput between wait-free and blocking CSDSs is that in a CSDS, the time needed to traverse the structure in order to reach the point where the operation needs to actually be performed generally dominates the total execution time. In essence, the smaller the number of memory locations an operation needs to read or write, the faster the operation. Due to their simpler nature, blocking algorithms have to chase a smaller number of pointers during an operation: the most efficient such algorithms simply have to go through the nodes and follow the next pointers, and only lock the area which needs to be modified.

In the case of wait-free algorithms, the underlying reason for their inefficiency is a fundamental mismatch between current architectures and the complexity of wait-free algorithms. In non-blocking CSDS algorithms, the common approach is to associate some concurrency information with each *next* pointer of a node, and update this information and the pointer atomically. While the unused three least significant bits of a pointer (in a 64-bit architecture) are generally sufficient for this concurrency information in the case of lock-free algorithms, this is not the case for the most efficient wait-free algorithms (which might need, for example, version numbers associated with next pointers). In terms of practical implementations, this usually translates in additional objects being interposed between data structure nodes, resulting in slower traversals of the structure. This is illustrated in Figure 2. Thus, wait-free algorithms fail to provide at least one of the characteristics of an ideal concurrent algorithm we have previously identified: high aggregate throughput.

Of course, our experiment tells only half of the story: the main goal of wait-freedom is not high system-wide throughput, but rather the promise that every request will eventually return (fine-grain latency). In the rest of the paper, we focus on showing that in addition to providing good coarse-grained performance metrics (Section 4), blocking CSDS algorithms also exhibit "wait-free behavior" for a very wide variety of workloads (Section 5).

2.3 Practical wait-freedom

We now look at how the theoretical guarantees of wait-free algorithms manifest in practice in the context of concurrent search data structures.

In theory, threads may crash. Wait-free algorithms ensure that despite the crash of a thread, the requests of other threads still get served. In practice, threads usually do not crash, and when they do, they do not crash independently: even when software bugs occur in a multi-threaded program, it is preferable to stop or restart the entire application rather than work with state that might have been corrupted.

Threads can however be temporarily delayed due to I/O, context switches, scheduling decisions or other interrupts. In practice, we expect wait-freedom to translate to a bound in the delay a request can suffer due to contention, i.e., all requests finishing before a certain deadline as long as the thread itself is scheduled (i.e., taking steps).

We thus argue that a data structure implementation which has a negligible percentage of requests that exhibit significant delays due to other concurrent threads in the system for a wide array of

Linked lists	Skip lists	Hash tables	BSTs
Lock-coupling list [30]	Pugh skip list [53]	Lock-coupling hash table [30]	Practical binary tree [6]
Lazy linked list [24]	Herlihy skip list [28]	Lazy hash table [24]	Logical ordering tree [13]
Pugh linked list [53]		Pugh hash table [53]	BST-TK tree [9]
Copy-on-write linked list [52]		Copy-on-write hash table [52]	
		ConcurrentHashMap [39]	
		Intel TBB [34]	
		URCU hash table [11]	

Table 1: Blocking search data structure algorithms considered in our evaluation.

workloads is *practically wait-free*. In practice, most systems indeed provide a Service Level Agreement (SLA) tolerating small percentages (e.g. 0.1% or 1%) of slow requests. Hence, what we call practical wait-freedom is bounded delays due to contention, for all but a potentially infinitesimal percentage of requests, under all realistic workloads.

This characteristic of an implementation does not imply a particular theoretical guarantee provided by the algorithm. It simply identifies implementations whose execution (in terms of request delays due to contention and fairness) cannot be distinguished from that of an algorithm that actually provides the theoretical guarantee of wait-freedom.

In the context of blocking CSDSs, there are two possible causes for which a thread's operation can be delayed by other concurrent threads: *locks* and *restarts*. To quantify the extent to which a thread is delayed as a result of them, we can measure the time an operation waits in order to acquire locks, as well as the number of times an operation has to restart. These two fine-grained metrics are indicative of different sources of delays to which wait-free algorithms may serve as better alternatives:

- Large average waiting times to acquire locks can be for the most part linked with i) other threads suffering delays while holding locks or ii) a large number of threads attempting to acquire the same lock. In the following sections, we show that in the context of state-of-the-art blocking CSDS algorithms, the latter case occurs only extremely rarely.
- In contrast, restarts capture a pattern present in a large fraction of the state-of-the-art blocking CSDS algorithms: if at some point during an operation, a state which does not allow it to progress correctly is encountered, the operation is restarted. These possible inconsistencies triggering restarts are a result of the fine-grained locking and optimistic approaches used in state-of-the-art algorithms. In general, a large percentage of requests having to be restarted repeatedly can be linked to high contention for a particular area of the data structure. The theoretical possibility of triggering restarts indefinitely might result in threads suffering from starvation.

We note that a more generic metric such as latency distribution is not appropriate in the case of all linked search data structures, given the fact that request latencies are often dominated by the time needed to reach the point of the data structure which needs to be accessed. Therefore, accesses to different parts of the data structure naturally have very different latencies. In addition, if context switches or other interrupts occur, threads may not even be taking steps, resulting in potentially unbounded latencies even for a theoretically wait-free algorithm.

In the following, we look at the extent to which blocking CSDSs provide the desired coarse-grained performance metrics, as well as the fine-grained performance metrics indicative of practical wait-freedom discussed above.

3. EXPERIMENTAL SETTING

For our study, we evaluated a wide range of blocking CSDS algorithms, summarized in Table 1, and part of ASCYLIB [5], an open source library implemented in C containing blocking and lock-free CSDSs. We report on their behavior under a wide range of conditions, representative of a wide variety of workloads. We enhance the library with benchmarks allowing us to report on the metrics presented in this paper. Roughly, we argue that the state-of-the-art algorithms have good throughput, scalability and behave practically wait-free regardless of the search data structure, thus representing ideal implementations for the vast majority of workloads.

For clarity, in the following experiments, we only highlight the behavior of the blocking algorithm exhibiting the best performance in our tests for each data structure, as our intention is to show that at least one practically wait-free algorithms exists for each structure type. Nevertheless, the conclusions we draw are in fact valid for multiple state-of-the-art algorithms for each data structure.

The best performing algorithms per data structure (which are shown in the following figures) are the lazy linked list, Herlihy's skip-list, the lazy linked list-based hash table (one lazy linked list per bucket, with per-bucket locks, with average load factor per bucket set to 1), and the BST-TK external binary search tree.

3.1 State-of-the art algorithms

Before we go into any details, it is important to understand the principles according to which the best performing blocking CSDS algorithms operate today [9]:

- Read operations do not perform any stores, and do not trigger any restarts;
- Update operations can be divided into a parse phase and a write phase: in the parse phase, the area of the data structure where the update needs to be applied is reached in a synchronization-free manner, while during the write phase, the actual updates are applied by writing to a small neighborhood of nodes in the data structure.
- Conflicts arise between two threads if they are executing their write phase concurrently, and the nodes accessed during these phases by the two threads intersect.
- Parses and reads are not obstructed by concurrent updates.

In short, in all these algorithms, the only blocking portions are short sequences of code in the update operations in which the actual modifications to the data structures are performed.

3.2 Implementation details

Our structures use either test-and-set locks or ticket locks in our experiments. We observe no benefits from using more complex locks, such as MCS locks, due the low degree of contention for any particular lock in these data structures. In addition, it has been

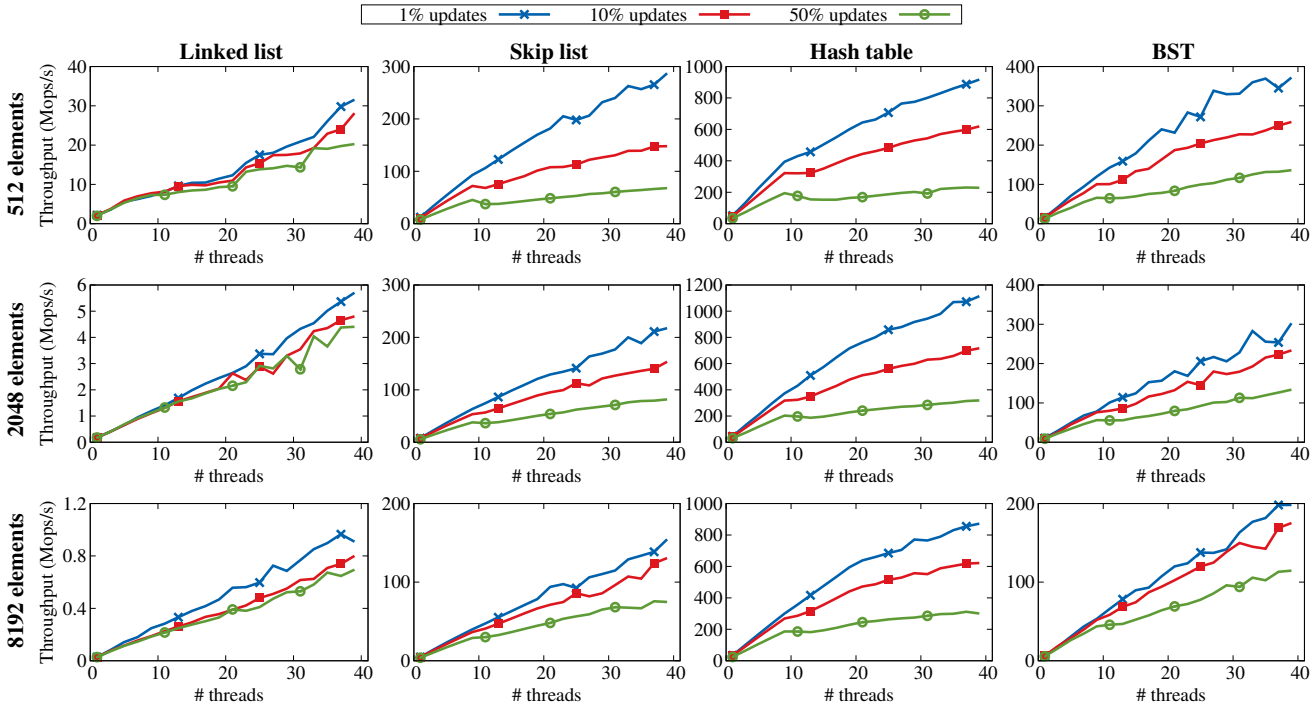

Figure 3: Throughput scalability of blocking implementations.

shown that these simple locks often perform well in practice [10], even for moderately high degrees of contention. Our implementations use an epoch-based memory management scheme, similar in principle to RCU [41].

In order to test our implementations, we use an Intel machine having two Xeon E5-2680 v2 Ivy Bridge processors with 10 cores each (20 cores in total). In addition, each core supports two hardware threads. The cores run at 2.8 GHz and have caches of sizes 32KB (L1), 256 KB (L2) and 25 MB (LLC per processor). The machine runs Ubuntu Linux 14.04 (kernel version 3.13.0). We bind threads to cores such that threads first use all physical cores before using both hardware contexts of the same core.

3.3 Methodology

We present results for different structure sizes and update ratios, which we believe to be representative of those used in practical systems. We consider structures consisting of 512, 2048 and 8192 key-value pairs, and percentages of update requests of 1%, 10%, and 50%. Half of the updates are inserts, and the other are removes. These parameters, are, in fact, similar to the ones we observe in real systems (e.g., in LevelDB [20], RocksDB [15], Memcached [42], MySQL [50], MongoDB [49], MonetDB [48]). We also address smaller structures and higher degrees of contention in a separate experiment.

We evaluate the extent to which blocking CSDSs provide the two main features of an ideal concurrent algorithm: (i) high system-wide performance, and (ii) practical wait-freedom. We start by showing that blocking CSDSs provide the desired coarse-grained performance metrics, after which we look at the fine-grained metrics indicative of wait-freedom, and study how they evolve as we modify the parameters of our workload and environment.

Unless otherwise specified, the distribution of accesses over the key space is uniform. Keys and values have 64 bits in size. Using larger values is straightforward: instead of the 64-bit value we would simply manipulate pointers to these larger values. Each of the worker threads in our benchmarks continuously issues requests.

In each of the workloads, we consider a key space twice as large as the structure size. Given that our workloads have an equal number of inserts and removes, this ensures that on average the data structure size remains close to the initial size throughout the experiment. We use runs of 5 seconds and report the average results of 11 runs.

In order to capture the system-wide performance and the degree to which the individual requests of these algorithms are delayed, we consider the following metrics: (a) the throughput as we increase the number of threads from 1 to 40 (the maximum number of hardware threads on our machine), (b) the average throughput per thread and the standard deviation of this quantity, (c) the average percentage of time spent waiting for locks by each thread and the standard deviation, and the (d) percentage of operations that have to restart at least once. We also look at the distribution of the values for the last two metrics among requests to identify any outliers. Where not specified, measurements are taken using 20 concurrent threads.

4. COARSE-GRAINED METRICS

In this section, we evaluate the first performance characteristic of an ideal CSDS: we verify the extent to which blocking CSDSs provide good aggregate throughput and scalability for various structure sizes and update ratios.

Figure 3 presents the evolution of the throughput as a function of the number of threads. We note that the structures exhibit no decrease in scalability as the number of concurrent threads increases. Of course, as we increase the percentage of updates more cache invalidations are generated, resulting in slightly larger latencies and overall lower throughput increases. However, the scalability trends remain the same. This is particularly noticeable in the case of the hash table, due to the much higher incurred throughput and the lower latencies of the requests. Once we have to use both sockets of the multi-core (for more than 10 threads), the scalability slope

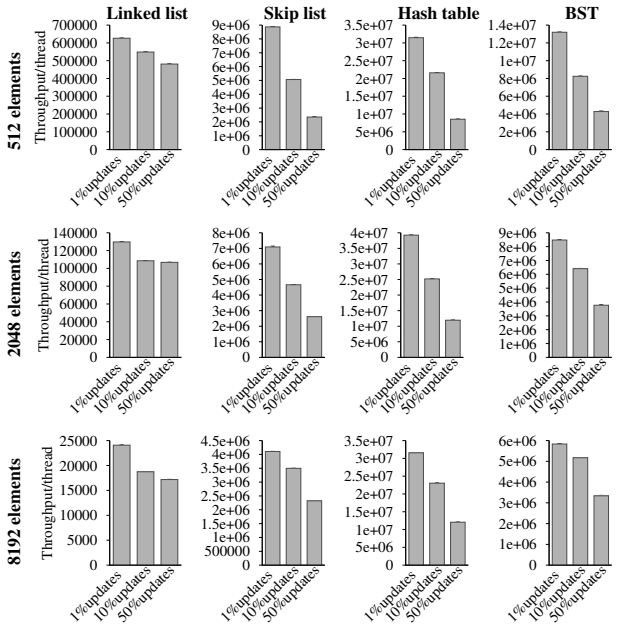

Figure 4: Per-thread throughput (and standard deviation).

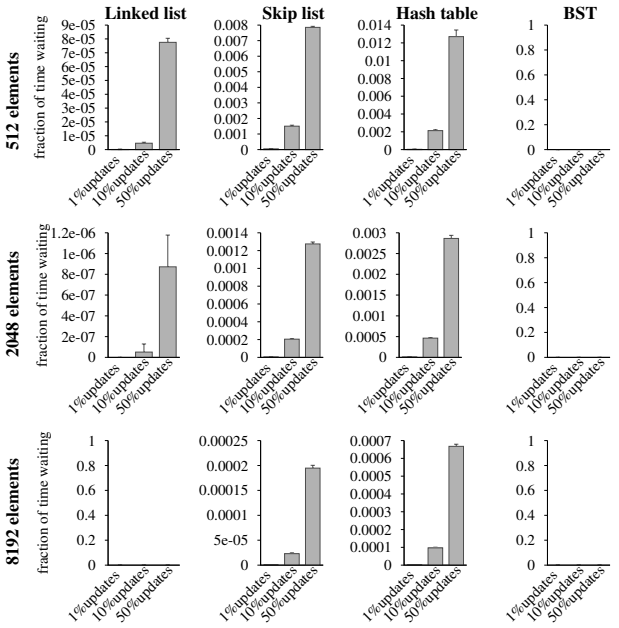

Figure 5: Fraction of time threads spend waiting for locks (and standard deviation).

slightly reduces due to increased latencies to access data. This effect of the higher cache coherence latencies on the total throughput is inherent to each data structure, and cannot be bypassed regardless of the algorithm and its progress guarantees [9, 10].

In addition, in Figure 4, we also present the average throughput per thread and the standard deviation of this metric. We depict the standard deviation using error bars. On average, the standard deviation is 0.2% of the average per-thread throughput. This quantity is so small compared to the per-thread throughput, that it is not visible on the graphs. This is valid for all the data structures and for all the workloads. Given that threads continuously issue requests, we can also conclude that the average latency is identical among threads. The observation we extract from this experiment is thus that blocking CSDSs ensure a high degree of fairness.

The conclusion we can draw from these experiments is that the blocking nature of these algorithms is not an obstacle to high throughput or scalability, even as we modify the size of the structure or the percentage of update operations. In addition, all threads exhibit high performance: there is no skew between the throughputs of the threads.

5. PRACTICAL WAIT-FREEDOM

While as we have shown in the previous experiments, blocking algorithms have good throughput scalability and are fair, in this section we look closer at the fine-grained performance metrics indicative of practical wait freedom. As presented in Section 2, these metrics are the amount of blocking and retries as we vary the data structure size and update ratio.

We study how these metrics evolve as we very the percentage of updates, decrease the size of the data structure, use non-uniform workloads, cause threads to become unresponsive, and induce frequent context switches.

5.1 Structure size and update ratio

In this experiment we collect the performance metrics as we vary the structure size and the update ratio. Figure 5 presents the percentage of time threads spend waiting for locks. The percentage is relative to the total execution time when threads are continuously

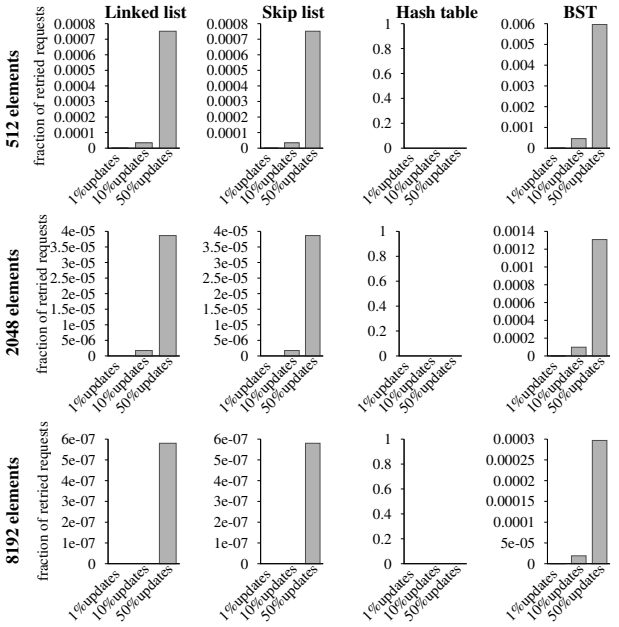

Figure 6: Fraction of requests that are restarted.

issuing requests. We measure this amount of time by using ticket locks: once a thread has acquired its ticket, if it is not immediately its turn to be served, we measure the time until this event occurs. We note that this percentage is under 2% in all situations, with most values being significantly below this percentage. In the case of the 8192-element linked list, for example, no thread has to wait in order to acquire locks. For the 2048-element linked list, the standard deviation is large due to the fact that the waiting times are between 0 and a few hundred cycles: even one brief delay waiting for a lock makes a thread an outlier. In the case of the BST, the tested algorithm uses trylocks, and restarts the operation in case the locking

attempt fails. Therefore, the time spent waiting for locks is zero, but this is compensated by the slightly higher percentage of operations that are restarted.

Similarly, as shown in Figure 6, the percentage of operations delayed due to restarts is significantly smaller than 1% in all situations. This value is 0 in the case of the hash table: each bucket is protected by a lock, so once the operations have acquired the lock they never restart.

We also run an experiment in which we look at the distribution of the two sources of delays (the number of restarts and the time spent blocked) on a per-request basis. We want to identify any outliers: requests significantly delayed due to concurrency, which would violate practical wait-freedom. We consider a workload using a linked list of 512 elements, 40 threads and 10% updates. Only 0.01% of the requests had to wait for locks, with no requests waiting for more than 6μs. In addition, out of the 26 million operations, 2900 had to restart once, 9 had to restart 2 times, and none had to restart more than that.

The conclusion we can draw from these experiments is that blocking CSDS algorithms cause negligible request delays due to concurrency, thus allowing threads to complete their requests in a finite number of their own steps. When these small delays do occur, there are no requests that are affected significantly more than others.

These metrics also allow us to confirm the fact that our conclusion applies to the state-of-the-art search data structure algorithms, and not to more naive implementations, which have more frequent or longer critical sections. For example, we consider a lock-coupling linked list [30]. This algorithm, while using fine-grained locks, acquires locks as the structure is traversed. We measure the percentage of time threads spend waiting for locks for this algorithm. With 20 concurrent threads and just 1% updates, threads spend around 10% of their time waiting for locks, regardless of the structure size. Therefore, we do not claim such algorithms are practically wait-free.

5.2 Non-uniform workloads

We now look at non-uniform workloads, in which some keys are more popular than others. We use a Zipfian distribution of requests over the key space with $s = 0.8$. Zipfian distributions are known to model a large percentage of real workloads [8]. We show results for each of the data structures, using a workload with 20 threads, 2048 data structure size and 10% updates. We provide numbers for the time threads have to wait for locks, and the average number of retries that have to be performed. The results of this experiment are depicted in Figure 7. While the values observed in this experiment are slightly higher than for uniform workloads, the delays remain very low: threads spent at most 1% of their time waiting for locks, and only restart at most 0.30% of their operations regardless of the data structure.

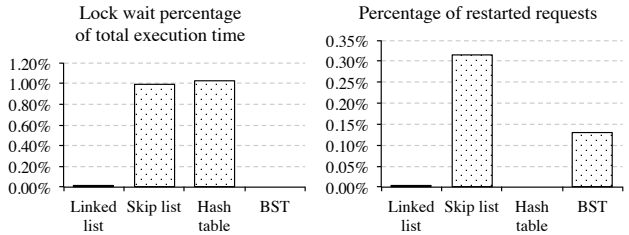

Figure 7: Percentage of time spent waiting for locks and percentage of requests restarted for blocking search data structures on a workload with a Zipfian request distribution.

Thus, we can conclude that blocking CSDS algorithms behave practically wait-free on such non-uniform workloads as well.

5.3 High contention

We present the following experiment: we consider a scenario in which 40 threads concurrently access a data structure, with 25% of the operations being updates. We start with a structure having 512 elements and in subsequent runs reduce its size down to an extreme-contention configuration with a structure consisting of 16 elements on average out of 32 possible keys. We report the percentage of time threads spend waiting for locks, as well as the percentage of update operations that restart at least once, and more than three times (reads do not restart). The percentage of updates restarted at least three times provides a measure of the number of such requests that are significantly delayed. As before, the hash table does not restart (as we use per-bucket locks), while the BST does not wait for locks.

Figure 8 presents the results of this experiment. In the case of the linked list, in the most extreme contention configuration, with only 16 elements in the structure, threads spend about 30% of their time waiting for locks, with 20% of the operations restarting at least once, and 1.8% of the operations restarting more than three times. Arguably, these numbers stretch the limits of what could be considered as practical wait-freedom, and in this particular configuration non-blocking algorithms may represent a better alternative. However, a data structure of size 32 already spends only around 1% of the time waiting for locks, restarts only 0.6% of its operations and repeatedly restarts only 0.02% of them, thus warranting the claim of practical wait-freedom. The values of these metrics continue to decrease steeply with the increasing data-structure size. With a linked list of 512 elements, practically no requests are significantly delayed or restarted more than three times.

Other data structures behave similarly: while for the very smallest structures our metrics can be non-negligible, they decrease exponentially as we increase data structure size. In the case of the hash table, since there are no restarts and we use per-bucket locks, the time spent waiting for very small hash tables is somewhat larger than for the other structures. This can be addressed by using finer-grained locks: i.e, use per-node locks instead of per-bucket locks. Nevertheless, since the hash tables used in practice are usually large in size, we use per-bucket locks throughout the paper.

The conclusion we can draw from this experiment is that there are indeed some extreme cases in which blocking data structures exhibit a non-negligible percentage of delays. However, these cases require extremely small data structures, a very high degree of concurrency and relatively high update ratios. We argue that such particular situations are rare in practice in the context of CSDSs. In fact, we have not observed such highly contested structures in popular practical systems which use CSDSs. In the majority of cases, even under high contention, blocking CSDSs still behave practically wait-free.

5.4 Unresponsive threads

We now look at a scenario in which threads become slow. In the first instance, we cause a thread to be delayed for a random interval between 1000 and 100000 ns every 10 updates, while holding locks. This range of delays captures many of the events that might occur in practice, such as accessing data from memory, SSD, or sending packets over the network. In essence, given that threads usually hold locks for short intervals only, this experiment looks at a worst-case scenario, where delays only happen while the locks are held. We present results for a workload having 20 concurrent threads, using data structures consisting of 2048 key-value pairs

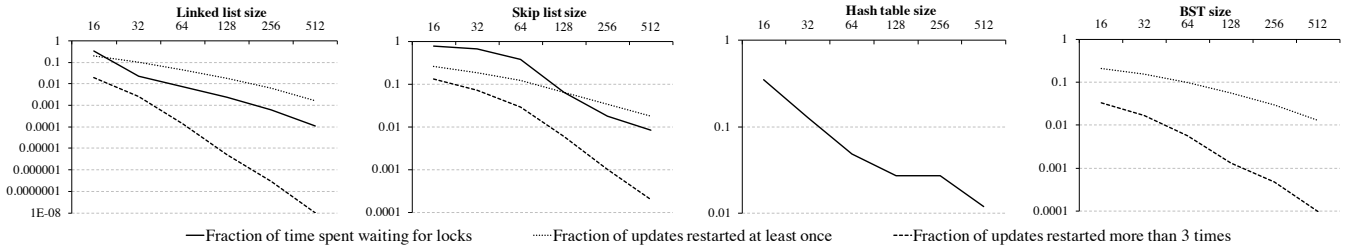

Figure 8: Delayed requests and time spent waiting for locks as a function of structure size.

and 10% update operations. In Figure 9, we provide results for the time spent waiting for locks and the number of restarted requests. Given that BST-TK uses trylocks [22], the waiting time for locks is normally 0. However, to better capture the effects of a slow thread on this algorithm, we depict the average time spent by threads on retries due to trylock failures. We note that threads spend at most 1% of their execution time waiting for locks. In fact, aside from the hash table, the percentages for the other data structures are significantly lower. We can lower this percentage for the hash table as well by replacing the per-bucket locks with finer grained locks for the linked lists representing each bucket. The percentage of restarted requests is also small: at most 0.015% for the skip-list. Thus, even under such workloads with temporary delays, the practically wait-free behavior of blocking CSDSs is not affected.

There is of course a limit up to which threads can be delayed before the system behavior is affected and practical wait-freedom jeopardized. In essence, an unresponsive thread might be problematic if (i) the delay happens while a thread is holding a lock and (ii) the thread is unresponsive for a large period of time. We find that such a scenario is possible, for example, in workloads where the number of threads significantly outnumbers the available cores (multi-programming). In an example using 4 threads per hardware context (160 threads in total), we find that around 3300 context switches occur every second. Roughly, that means that every thread executes for 12 ms, after which it is swapped out for 37 ms. Although critical sections in our algorithms are very short, a few of these context switches are bound to happen while locks are held. In such scenarios, while algorithms' performance is affected regardless of their progress guarantee, we find this to be more obvious in the case of blocking algorithms, particularly under workloads with higher update ratios.

To address this issue, we propose using hardware features recently introduced on modern architectures, such as Intel's Transactional Synchronization Extensions (TSX) [33], which allow us to elide locks. One characteristic of Intel's TSX implementations is the fact that hardware transactions are aborted when interrupts occur. While this is often regarded as a limitation [47, 61], we use

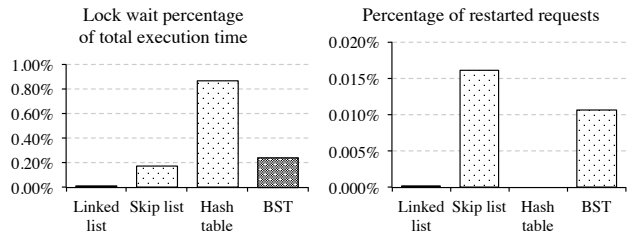

Figure 9: Percentage of time spent waiting for locks and percentage of requests restarted for blocking search data structures when a thread repeatedly suffers delays.

the abort-on-interrupt characteristic of TSX to our advantage, as it enables us to maintain the practically wait-free behavior of CSDS algorithms even in the case of frequent context switches, I/O, or other interrupts: even if a thread is within a critical section when the interrupt occurs, the hardware transaction is aborted and locks are not held when threads are not scheduled. This technique does not change the blocking nature of the algorithms: since Intel TSX is best-effort only, we need to provide a fall-back path which uses the actual locks for the situation when a speculative execution of a critical section is repeatedly aborted. The effectiveness of this approach is thus contingent on hardware transactions not aborting extremely frequently.

As we have observed in our evaluation, and as we will further show in Section 6, the probability of contention between two threads is small. Therefore, only a very limited percentage of updates will be repeatedly aborted due to data conflicts, even when contention is high. In addition, the short critical sections in the write phase do not trigger interrupts in the common case. While it is possible for page faults to occur within a critical section, we argue that this is a fairly rare event, and, given that by using TSX we can re-try the optimistic transactional approach multiple times before reverting back to the pessimistic path, this does not jeopardize wait-freedom. Additionally, TSX also provides us with a degree of tolerance to failed threads. Although in general one would stop a concurrent system rather than work with corrupted state, using TSX-enabled locks could also prevent failed threads from leaving the system in an inconsistent state, or holding locks after crashing.

We validate our hypothesis using a 3 GHz 4-core (8 hardware threads) Intel Haswell Core i7-4770 machine. The processor has 32 KB L1, 256 KB L2, and 8 MB L3 caches. Our algorithms remain unchanged: we only add TSX instructions to the acquire and release methods of the locks. Except for BST-TK, which uses ticket trylocks, all the other algorithms use test-and-set locks. We show results for a workload using 32 concurrent threads (8/physical core), with data structures consisting of 1024 key-value pairs and workloads with varying update ratios. Given the frequent context switches, measuring the amount of time threads spend waiting to acquire locks is not an appropriate metric here. Instead, we measure the number of operations that fail to elide the locks and fall back to normal lock acquisition: it is only these operations that have the potential of delaying other threads for longer amounts of time. We report this as a fraction of the total number of lock acquisition calls. The results are shown in Table 2. We note that in most cases the percentage of operations that acquire the locks is well below 1%. This number is slightly larger in the case of the skip-list, which needs to take multiple locks per update. The conclusion we can draw from this experiment is that the probability of a thread being de-scheduled while holding a lock is extremely small: an interrupt would have to occur while a thread is (i) in the write phase of an update operation and (ii) after the thread failed repeatedly to

Update ratio	Linked list	Skip list	Hash table	BST
20	0.001	0.011	0.001	0.000
50	0.001	0.012	0.001	0.000
100	0.001	0.014	0.002	0.001

Table 2: Fraction of critical sections falling back to acquiring locks using 8 threads/physical core (32 threads in total) with data structures of size 1024.

Update ratio	Linked list	Skip list	Hash table	BST
20	1.11	10.6	2.46	2.21
50	1.23	20.21	3.06	2.65
100	2.26	53.28	2.75	2.56

Table 3: Throughput improvements of TSX-enabled versions vs. default implementations using 8 threads/physical core (32 threads in total) with data structures of size 1024.

elide the locks. We theoretically estimate this probability in Section 6.

We also measure the increase in throughput we obtain in these scenarios when using TSX. Table 3 shows the ratio between the throughput of the TSX-enhanced versions of the algorithms and the default implementations. We note important improvements in all data structures. The benefits of using TSX are particularly impressive in the case of the skip list. This is due to the lager number of locks per update operation, which increases the potential of a thread being de-scheduled while holding a lock.[3]

The behavior revealed in the previous experiments allows us to conclude that TSX indeed enables us to maintain practically wait-free behavior even in the face of frequent context switches, which is a scenario particularly well-known for causing issues in the context of concurrent systems.

Depending on the level of privilege of the code, an alternative to our solution could be to make the critical sections non-preemptable. However, even this might be problematic in the context of virtualization: scheduling multiple virtual machines in the presence of locks is a well-known challenge [19, 36, 55, 59]. Essentially, the guest OS does not have control over the scheduling decisions of the hypervisor. Our proposed approach would maintain the practical wait-freedom of CSDS algorithms even in such a situation, and may prove beneficial for other applications in virtualized environments as well.

In this section, we have shown that blocking algorithms behave practically wait-free on structures of different sizes, with different percentages of updates, under non-uniform workloads, under extreme contention and with frequently unresponsive threads. While it is extremely difficult for an evaluation to address every possible scenario, we argue that these experiments cover most situations that arise in practical systems using CSDSs. Thus, given the behavior of blocking CSDSs observed in this section, we conclude that in practice, for the vast majority of workloads, blocking CSDSs behave practically wait-free.

6. THE BIRTHDAY PARADOX

In essence, the explanation for the wait-free behavior of blocking CSDSs is that the probability of threads being delayed due to contention is very small. We say that a *conflict* occurs when a thread is blocked or restarted by another thread. In state-of-the-art CSDS algorithms, a thread can generally encounter a conflict only during the write phase of an update operation. We provide an estimation of this conflict probability using an analogy with the *birthday paradox*.

Assuming threads that continuously issue requests, we first estimate the fraction of time a thread spends doing update operations (in a concurrent execution, including the acquisition and release of locks):

$$f_u = \frac{u \times dur_u}{u \times dur_u + (1-u) \times dur_r} \quad (1)$$

Here u is the update ratio (the fraction of operations that are updates), dur_u is the average duration of an update, and dur_r is the average duration of a read.

We then estimate the fraction of time a thread spends in its write phase, where d_w is the average duration of the write phase, and d_p is the average duration of the parse phase (recall that in state-of-the-art algorithms updates have a parse phase and an update phase):

$$f_w = f_u \times \frac{d_w}{d_w + d_p} \quad (2)$$

It is then a matter of computing the probability of conflict between threads concurrently executing their write phase. This is highly specific to the data structure, the implementation, and the request distribution. In general however, it can be reduced to variations of the birthday paradox: the probability of randomly chosen variables being similar enough to each other to cause conflicts. We denote by $B_s(k, n)$ the probability that if k threads are concurrently executing their write phases on a structure s of size n, at least one conflict will arise.

The probability of conflict in a system with t threads is thus:

$$p_{conflict} = \sum_{k=1}^{t} \binom{t}{k} {f_w}^k (1-f_w)^{t-k} B_s(k,n) \quad (3)$$

We now provide numeric examples for some of the structures and workloads employed in this paper.

6.1 Hash table

Our hash table implementation has the particularity of using one lock per bucket. Hence, in the above equations, d_p is 0: the lock is acquired immediately after the update starts. In the case of the hash table, since conflicts only appear if two threads want to write to the same bucket concurrently, the problem reduces to the classical version of the birthday paradox. We therefore have:

$$B_{ht}(k,n) = 1 - \frac{\prod_{i=1}^{k-1}(n-i)}{n^{k-1}} \quad (4)$$

We first assume a uniform workload for simplicity (see below for a non-uniform case). In this scenario, an update operation takes approximatively twice as long as a read operation. We assume a scenario with 1024 buckets and 20 threads, with 10% of the operations being updates. f_u is then 0.18, f_w has the same value (given that d_p is 0), and thus, using the formulas above, we obtain $p_{conflict} = 0.0058$. Therefore, the probability of any thread being delayed due to concurrency at any point in time is about 0.58%, a small enough percent to lead to wait-free behavior.

6.2 Linked list

In the case of other data structures, this computation is slightly more complex. For example, in our linked list, in which a remove operation has to lock two consecutive nodes, we can use the solution to the "almost birthday paradox" [1] to provide an upper bound

[3]We notice no important difference between the default and TSX-enabled implementations in other experiments used in this paper.

to the probability of conflict. In this case, we have:

$$B_{ll}(k,n) = 1 - \frac{(n-k-1)!}{(n-2k)!n^{k-1}} \quad (5)$$

We provide a numerical example, considering a slightly higher contention example, with a list of 512 elements, 40 concurrent threads and 20% updates. An important distinction with the hash table is that the parse phase dominates the latency of an update operation. In fact, on average, the write phase takes only around 10% of the parse phase duration. That also means that updates are only one tenth more expensive than reads. Applying the presented formulas, we obtain $f_w \approx 0.0215$. We apply the formula for computing the probability of conflict in the case of the almost birthday paradox and obtain $p_{conflict} = 0.0021$. In the case of the linked list, the probability of conflict is thus 0.21%, again a percentage permitting wait-free executions.

6.3 Uniform vs. non-uniform workloads

In the above, we have considered uniform workloads. Of course, one could also repeat the computations considering non-uniform accesses. In that case, a variation of the birthday paradox using non-uniform probability distributions would have to be employed. For instance, using a Poisson approximation to model the probability of conflicts in such scenarios, we have:

$$B_{non-uniform}(k,n) = 1 - e^{-\binom{k}{2}\sum_{i=1}^{n} p_i^2} \quad (6)$$

Here, p_i denotes the probability of element i in the structure to be accessed. Using our linked list example from above and a Zipfian workload with $s = 0.8$, the probability of conflict is 0.47%. This value is slightly larger than in the uniform case, but still well below 1%, allowing practically wait-free executions.

These examples do not take into account slow threads, assuming similar speeds for all participants. One could model a slow thread, by assuming that while a thread t_{slow} writes to a set of memory locations, the other threads do multiple operations, writing to multiple such sets of locations. We give below an intuition for the probability of conflict when using TSX to avoid threads holding locks being interrupted.

6.4 TSX-based algorithms

In the algorithm versions in which the critical sections are wrapped with TSX instructions, speculative execution can be attempted several times. We assume each transactional region is tried five times before falling back to actually taking the locks. In these versions of the algorithms, conflicts occur if data that is being written during the critical section is read or written by another thread, or if data read during the critical section is written by another thread. Hence, while as before, only threads which are in the write phase may be the victims of a conflict, in this case all other threads may be causing the conflict, even those in the process of reading data. We can continue to use Equation 2 in order to estimate the probability of a conflict. However, given that threads during their read or parse phases also need to be taken into account, the computation of the term $B_s(k,n)$ will be different, as we will illustrate in the following. With five re-tries before reverting to locking, we can estimate the probability of a thread reverting to locking as $p_{lock} = p_{conflict}^5$.

We use the same hash table and linked list examples as above. In the case of the hash table, by taking reads into account when computing the probability of conflict with k current writers on a hash table of size n, we have:

$$B_{ht-tsx}(k,n) = 1 - \frac{(n-k)^{t-k}\prod_{i=1}^{k-1}(n-i)}{n^{t-1}} \quad (7)$$

With the same numeric values as before, $p_{lock} = 0.0005\%$.

In the case of the TSX-enhanced linked list, we have:

$$B_{ll-tsx}(k,n) = 1 - \frac{(n-k-1)!}{(n-2k)!n^{k-1}} \left(\frac{(n-2k)(n-2k-1)}{n(n-k-1)}\right)^{t-k} \quad (8)$$

For our linked list example, this results in $p_{lock} = 0.001\%$ (in the case of this fairly contended linked list, the probability of at least a re-try of the transactional region is however non-negligible: 16%).

It is important to note the significance of the low probabilities of conflict computed throughout this section: it simply means that there is a 1% chance that *some* thread in the system is delayed at any point in time. This delay, as shown by our practical evaluation, is likely to be insignificant for any particular thread. In addition, after such small delays the thread quickly returns to its steady state, with no changes to the parameters considered in this section.

7. BEYOND SEARCH DATA STRUCTURES

In the previous sections, we focused our attention on search data structures. One of the main reasons we are able to obtain a practically wait-free behavior for a large spectrum of workloads in the case of state-of-the-art CSDSs is the low probability of conflicts between concurrent operations: accesses are distributed over all the nodes contained in the structure. In this section, we briefly present the limits of our conclusion when applied to concurrent objects other than search data structures.

7.1 Intuition

Intuitively, in the case of data structures such as queues, stacks, priority queues, and counters, the accesses are not distributed among multiple memory addresses, but rather concentrated on a small number of "hotspots". In this case, blocking algorithms essentially serialize operations by only allowing one (or a small number) to execute at a time. Thus, each thread spends a significant amount of time waiting for its turn. HTM-based techniques, which mitigate delays due to interrupts for search data structures, are also not applicable for such data structures: virtually all hardware transactions would repeatedly abort due to data conflicts. In a search data structure, it is only possible to encounter similar scenarios if (i) only a handful of nodes in the CSDS are accessed (ii) continuously by a large number of threads (iii) with a large fraction of the accesses being updates.

7.2 Experimentation

We quantify the extent to which these characteristics hinder practical wait-freedom using standard lock-based algorithms for a queue and a stack. Figure 10 shows the fraction of time threads spend waiting for locks in the case of these data structures. We use 20 concurrent threads, with 50% of the operations being inserts (enqueue/push), and 50% removes (dequeue/pop). The structures contain 1024 nodes. We note that the fraction of time threads spend waiting quickly approaches 1 as we increase the number of threads. This behavior is clearly not compliant with practical wait-freedom.

In the case of such data structures therefore, one can expect to obtain better performance by using lock-free or wait-free algorithms, to the design of which much work has been dedicated [16, 26, 35, 38, 46, 56, 60]. Another technique which has proved effective for such data structures is flat combining [25]. A further avenue which has been pursued in recent years in order to reduce the contention for any particular node in such structures is to relax the semantics of the data structure operations [4, 12, 62].

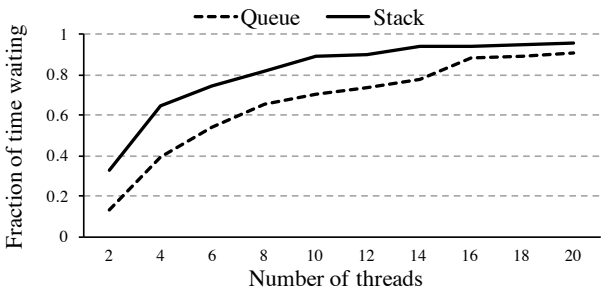

Figure 10: Percentage of time threads spend waiting in blocking queue and stack implementations.

8. RELATED WORK

Asynchronized concurrency [9] is a set of patterns for the design of CSDS algorithms resulting in implementations with nearly-optimal throughput. The choice between blocking or lock-free CSDS algorithms is shown to not be of fundamental importance from the point of view of performance (i.e., throughput, average latency) as long as these patterns are applied. However, the authors only focus on the aggregate, system-wide performance metrics, and do not study metrics indicative of the behavior of individual requests. Similarly, Gramoli [21] studies different classes of concurrent data structures, and observes no inherent difference in aggregate throughput between lock-free and lock-based search data structures. In this paper, we look at CSDSs through the perspective of a different criterion: practical wait-freedom.

We are not the first to ask whether algorithms with weaker progress guarantees do not actually provide wait-freedom, given some assumptions about the execution environment. We are however the first to look at blocking algorithms. In doing so, we also provide the first practical quantification of the effects of progress guarantees on CSDSs. Ellen et al. [17] show that under an unknown-bound synchronous model, obstruction-free algorithms can behave practically wait-free. Herlihy and Shavit [29] suggest that on realistic schedulers, lock-free algorithms behave in a wait-free manner. Alistarh et al. [3] prove that assuming a stochastic scheduler which determines the ordering of accesses to a memory location, lock-free algorithms behave practically wait-free. Our work suggests that in the case of CSDSs, given the fact that the probability of contention for any memory location is low, the characteristics of the scheduler governing the order of memory accesses might not be a determining factor.

Michael [45] also studies the practical trade-offs between different progress guarantees. However, his analysis only discusses non-blocking algorithms, and does not focus on CSDSs, but mostly on objects where a single point of contention is likely, such as counters and queues. As we have shown in this paper, the properties of state-of-the-art concurrent algorithms for search data structures are significantly different from other concurrent structures.

Rajwar and Goodman [54] propose a hardware mechanism for transactions and identify it as conductive to non-blocking behavior in lock-based programs. However, the transactional support they propose provides starvation freedom, which is not the case in practical implementations such as Intel TSX. In the context of blocking CSDSs, we show that HTM support is needed for wait-free behavior in a particular set of circumstances only. This observation is, we believe, interesting in its own right. We also show that wait-free behavior is possible even with best-effort transactional memory which may revert to locking, and explain why this conclusion does not extend to other concurrent structures beyond CSDSs.

9. CONCLUDING REMARKS

The main conclusion we can draw from this paper is that, practically, one can achieve the behavior of wait-free CSDS algorithms in terms of individual thread progress with blocking implementations. The nature of search data structures is such that there is no single contention point in the structure, rendering locks less problematic than they might be for structures such as concurrent queues, stacks, and counters. Our conclusions only concern CSDSs: we do not claim blocking implementations of objects such as queues and stacks are practically wait-free. It is also important to note that we do not claim that *every* blocking CSDS algorithm is practically wait-free. Rather, we considered state-of-the-art blocking algorithms, which generally have synchronization-free reads and writes with minimal and extremely fine-grained synchronization. We find such practically wait-free algorithms for each data structure we study. In addition to showing that these algorithms are practically wait-free, this work also represents the first detailed quantification of the effects of a progress guarantee on the behavior of practical CSDSs. Moreover, we have shown how new technologies such as Intel TSX can be leveraged to provide the desired performance characteristics of CSDSs. We also note that while the experiments presented in this paper were run on an Intel Xeon server, we have verified our conclusions on other architectures as well, including servers from AMD and Oracle.

Acknowledgements. This work has been supported in part by a VMware Fellowship.

10. REFERENCES

[1] Morton Abramson and WOJ Moser. More birthday surprises. *American Mathematical Monthly*, pages 856–858, 1970.

[2] Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free Made Fast. STOC 1995.

[3] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. Are Lock-free Concurrent Algorithms Practically Wait-free? STOC 2014.

[4] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The SprayList: A Scalable Relaxed Priority Queue. PPoPP 2015.

[5] ASCYLIB. http://github.com/LPD-EPFL/ASCYLIB.

[6] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A Practical Concurrent Binary Search Tree. PPoPP 2010.

[7] Nachshon Cohen and Erez Petrank. Efficient Memory Management for Lock-Free Data Structures with Optimistic Access. SPAA 2015.

[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. SoCC 2010.

[9] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. ASPLOS 2015.

[10] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. SOSP 2013.

[11] Mathieu Desnoyers, Paul E McKenney, Alan S Stern, Michel R Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *Parallel and Distributed Systems, IEEE Transactions on*, 23(2):375–382, 2012.

[12] Dave Dice, Yossi Lev, and Mark Moir. Scalable Statistics Counters. SPAA 2013.

[13] Dana Drachsler, Martin Vechev, and Eran Yahav. Practical

Concurrent Binary Search Trees via Logical Ordering. PPoPP 2014.

[14] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking Binary Search Trees. PODC 2010.

[15] Facebook. RocksDB. http://rocksdb.org.

[16] Panagiota Fatourou and Nikolaos D. Kallimanis. A Highly-efficient Wait-free Universal Construction. SPAA 2011.

[17] Faith Ellen Fich, Victor Luchangco, Mark Moir, Nir Shavit, and Sun Microsystems Laboratories. Obstruction-Free algorithms can be practically wait-free. DISC 2005.

[18] Keir Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, 2004.

[19] Thomas Friebel and Sebastian Biemueller. How to Deal with Lock Holder Preemption. Xen Summit North America 2008.

[20] Google. LevelDB. http://leveldb.org.

[21] Vincent Gramoli. More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. PPoPP 2015.

[22] Rachid Guerraoui and Vasileios Trigonakis. Optimistic Concurrency with OPTIK. PPoPP 2016.

[23] Timothy L Harris. A Pragmatic Implementation of Non-blocking Linked Lists. DISC 2001.

[24] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, III Scherer, William N., and Nir Shavit. A Lazy Concurrent List-Based Set Algorithm. In *Principles of Distributed Systems*, volume 3974. 2006.

[25] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. SPAA 2010.

[26] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A Scalable Lock-free Stack Algorithm. SPAA 2004.

[27] Maurice Herlihy. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[28] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. SIROCCO 2007.

[29] Maurice Herlihy and Nir Shavit. On the Nature of Progress. OPODIS 2011.

[30] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised First Edition*. 2012.

[31] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[32] Shane V Howley and Jeremy Jones. A non-blocking internal binary search tree. SPAA 2012.

[33] Intel. Intel Transactional Synchronization Extensions Overview. 2013.

[34] Intel Thread Building Blocks. https://www.threadingbuildingblocks.org.

[35] Amos Israeli and Lihu Rappoport. Efficient wait-free implementation of a concurrent priority queue.

[36] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv—Optimizing the Operating System for Virtual Machines. Usenix ATC 2014.

[37] Alex Kogan and Erez Petrank. A Methodology for Creating Fast Wait-free Data Structures. PPoPP 2012.

[38] Alex Kogan and Erez Petrank. Wait-free Queues with Multiple Enqueuers and Dequeuers. PPoPP 2011.

[39] Doug Lea. Overview of package util.concurrent Release 1.3.4. http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html, 2003.

[40] Paul E. McKenney, Dipankar Sarma, and Maneesh Soni. Scaling Dcache with RCU. *Linux Journal*, 2004(117), January 2004.

[41] Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[42] Memcached. http://www.memcached.org.

[43] Maged M Michael. High performance dynamic lock-free hash tables and list-based sets. SPAA 2002.

[44] Maged M Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *Parallel and Distributed Systems, IEEE Transactions on*, 15(6):491–504, 2004.

[45] Maged M. Michael. The Balancing Act of Choosing Nonblocking Features. *ACM Queue*, 11(7):50–61, 2013.

[46] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. PODC 1996.

[47] Mohamed Mohamedin, Roberto Palmieri, Ahmed Hassan, and Binoy Ravindran. Brief announcement: Managing resource limitation of best-effort HTM. SPAA 2015.

[48] Monetdb. http://www.monetdb.org.

[49] Mongodb. http://www.mongodb.org.

[50] Mysql. http://www.mysql.com.

[51] Aravind Natarajan and Neeraj Mittal. Fast Concurrent Lock-free Binary Search Trees. PPoPP 2014.

[52] Oracle. CopyOnWriteArrayList in Java docs. http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArrayList.html.

[53] William Pugh. Concurrent Maintenance of Skip Lists. Technical report, 1990.

[54] Ravi Rajwar and James R. Goodman. Transactional Lock-free Execution of Lock-based Programs. ASPLOS 2002.

[55] Xiang Song, Jicheng Shi, Haibo Chen, and Binyu Zang. Schedule Processes, Not VCPUs. APSys 2013.

[56] Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. IPDPS 2003.

[57] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. OPODIS 2012.

[58] Shahar Timnat and Erez Petrank. A Practical Wait-free Simulation for Lock-free Data Structures. PPoPP 2014.

[59] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards Scalable Multiprocessor Virtual Machines. VM 2004.

[60] Valois, John D. Implementing lock-free queues. ICPDCS 1994.

[61] Zhaoguo Wang, Hao Qian, Haibo Chen, and Jinyang Li. Opportunities and Pitfalls of Multi-core Scaling Using Hardware Transaction Memory. APSys 2013.

[62] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. The lock-free k-LSM relaxed priority queue. PPoPP 2015.