Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures

Tudor David   Rachid Guerraoui   Vasileios Trigonakis [*]

School of Computer and Communication Sciences,
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
{tudor.david, rachid.guerraoui, vasileios.trigonakis}@epfl.ch

Abstract

We introduce "*asynchronized concurrency (ASCY)*," a paradigm consisting of four complementary programming patterns. ASCY calls for the design of concurrent search data structures (CSDSs) to resemble that of their sequential counterparts. We argue that ASCY leads to implementations which are *portably scalable*: they scale across different types of hardware platforms, including single and multi-socket ones, for various classes of workloads, such as read-only and read-write, and according to different performance metrics, including throughput, latency, and energy. We substantiate our thesis through the most exhaustive evaluation of CSDSs to date, involving 6 platforms, 22 state-of-the-art CSDS algorithms, 10 re-engineered state-of-the-art CSDS algorithms following the ASCY patterns, and 2 new CSDS algorithms designed with ASCY in mind. We observe up to 30% improvements in throughput in the re-engineered algorithms, while our new algorithms out-perform the state-of-the-art alternatives.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

Keywords Concurrent data structures, scalability, multi-cores, portability

1. Introduction

A *search data structure* consists of a set of elements and an interface for accessing and manipulating these elements. The three main operations of this interface are a *search* operation and two update operations (one to *insert* and one to *delete* an element), as shown in Figure 1. Search data structures are said to be *concurrent* when they are shared by several processes. Concurrent search data structures (CSDSs) are commonplace in today's software systems. For instance, concurrent hash tables are crucial in the Linux kernel [40]

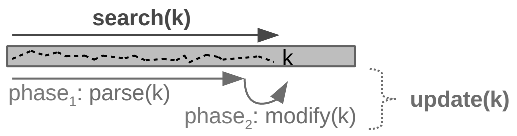


Figure 1: Search data structure interface. Updates have two phases: a parse phase, followed by a modification phase.

and in Memcached [42], while skip lists are the backbone of key-value stores such as RocksDB [18]. As the tendency is to place more and more workloads in the main memory of multi-core machines, the need for CSDSs that effectively accommodate the sharing of data is increasing.

Nevertheless, devising CSDSs that scale and leverage the underlying number of cores is challenging [4, 5, 8, 19, 47]. Even the implementation of a specialized CSDS that would scale on a specific platform, with a specific performance metric in mind, is a daunting task. Optimizations that are considered effective on a given architecture might not be revealed as such on another [4, 12]. For example, NUMA-aware techniques provide no benefits on uniform architectures [12]. Similarly, if a CSDS is optimized for a specific type of workload, slightly different workloads can instantly cause a bottleneck. For instance, read-copy update (RCU) [41] is extensively used for designing CSDSs that are suitable for read-dominated workloads. However, it could be argued that this is achieved at the expense of scalability in the presence of updates.

The motivation of this work is to ask whether we can determine characteristics of CSDS algorithms that favor implementations which achieve what we call *portable scalability*, namely that *scale* across various platforms, workloads, and performance metrics. At first glance, this goal might look fuzzy for it raises a fundamental question: what scalability can we ideally expect from a given data structure, architecture, performance metric, and workload combination?

In fact, we can provide a practical estimation of an upper bound for a data structure's scalability, on a particular hardware and workload combination. We step on the observation that the coherence traffic induced by stores on shared data is the biggest impediment to the scalability of concurrent software. This is valid for practically any contemporary multi-core. Yet, some stores cannot be removed because

[*] Authors appear in alphabetical order.

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey.
Copyright © 2015 ACM 978-1-4503-2835-7/15/03…$15.00.
http://dx.doi.org/10.1145/2694344.2694359

they are inherent to the semantics of the data structure; typically those stores are employed by any standard sequential implementation of the same data structure (one that is not supposed to be shared by several processes). Assume however that we deploy, as is, such a sequential implementation on a multi-core and have it shared by multiple threads. Obviously, this deployment would result in incorrect (e.g., non-linearizable [31]) executions. The performance of these *asynchronized executions*, however, constitutes a reasonable indication of what can be ideally expected from a correct, *synchronized* implementation of the same structure.

We thus consider that a CSDS achieves *portable scalability* when its scalability closely matches that of asynchronized executions (a) across different types of hardware, including single and multi-sockets, (b) for various classes of workloads, such as read-dominated and read-write, and (c) according to different performance metrics. In the CSDS context, aspect (c) means that as we increase the number of threads, we want to remain as close as possible to the asynchronized execution in terms of *throughput* and *latency*, without sacrificing *energy* (i.e., without consuming more *power* than implementations that do not scale as well in terms of throughput or latency).

With this pragmatic objective in mind, we perform an exhaustive evaluation of CSDSs on six different processors: a 20-core and a 40-core Intel Xeon, a 48-core AMD Opteron, a 32-core Oracle SPARC T4-4, a 36-core Tilera TILE-Gx36, and a 4-core Intel Haswell. We measure four dimensions of scalability: throughput, latency, latency distribution, and power. We consider the state-of-the-art algorithms for linked lists, hash tables, skip lists, and BSTs. To the best of our knowledge, this is the most extensive CSDS evaluation to date. We find that for each data structure, there are CSDS algorithms whose performance is within 10% of the asynchronized versions. We observe that in general, the algorithms whose memory accesses to shared state best resemble those of a sequential – asynchronized – algorithm tend to achieve portable scalability. We further identify four patterns through which this resemblance to sequential implementations is achieved:

ASCY$_1$: The search operation should not involve any waiting, retries, or stores.

ASCY$_2$: The parse phase of an update operation should not perform any stores other than for cleaning-up purposes and should not involve any waiting, or retries.

ASCY$_3$: An update operation whose parse is unsuccessful (i.e., the element not found in case of a remove, the element already present in case of an insert) should not perform any stores, besides those used for cleaning-up in the parse phase.

ASCY$_4$: The number and region of memory stores in a successful update should be close to those of a standard sequential implementation.

None of these patterns is fundamentally counter-intuitive and each of them has already been identified as important in some form or another. We find that the existing algorithms that scale the best already apply some of these patterns. To our knowledge however, they have never been put in a coherent form and collectively applied and evaluated. We refer to these patterns as *asynchronized concurrency (ASCY)*, for together they indeed call for the design of concurrent algorithms to resemble that of their sequential counterparts in terms of access to shared state.

We apply ASCY to several existing state-of-the-art algorithms and obtain up to 30% improvements in throughput, accompanied by reduced latencies. Interestingly, ASCY not only leads to better throughput, but also results in CSDSs that consume less power (by 1.4% in average), hence further improving energy efficiency. We also present a hash table (CLHT) and a BST (BST-TK), two *new algorithms* designed and implemented from scratch with ASCY in mind. CLHT (cache-line hash table) places each hash-table bucket on a single cache line and performs in-place updates so that operations complete with at most one cache-line transfer. CLHT outperforms state-of-the-art hash tables in virtually every scenario. BST-TK (BST Ticket) is a new concurrent BST that significantly reduces the number of acquired locks per update over existing algorithms. BST-TK is consistently the best lock-based BST, compared to the state of the art.

Our evaluation also highlights a number of other interesting observations. We show, for instance, that the fact that an algorithm is lock-based or lock-free does not have a major effect on the scalability of CSDSs. Similarly, we show that the use of hardware transactional memory also makes little difference. We highlight however a number of hardware-related bottlenecks that should be taken into consideration by system designers.

In summary, the main contributions of this paper are:

- The analysis and comparison of a large number of state-of-the-art CSDS algorithms in a wide range of settings (i.e., platforms, workloads, and metrics), representing the most extensive evaluation to date. This evaluation helps identify characteristics of *portably scalable* algorithms and revisit some beliefs regarding CSDSs.
- *Asynchronized Concurrency*: A design paradigm which yields portably scalable CSDSs. When ASCY is applied, increasing throughput, reducing latency, and reducing power consumption go hand in hand.
- ASCYLIB: a CSDS library, including 34 highly optimized and portable implementations of linked lists, hash tables, skip lists, and BSTs, together with a companion memory allocator with garbage collection. ASCYLIB includes two novel CSDS algorithms designed from scratch, namely CLHT and BST-TK, and re-engineered versions of ten state-of-the-art CSDS algorithms. ASCYLIB is available at `http://lpd.epfl.ch/site/ascylib`.

Our patterns are not very precise guidelines and cannot be used to automatically generate the implementation of a CSDS from its sequential counterpart. They cannot be used to derive any theoretical lower bound either. Yet, as we show in the paper, they provide very useful hints both for optimizing existing CSDSs and for designing new algorithms.

The rest of the paper is organized as follows. We provide in Section 2 some background related to search data structures. We present ASCYLIB in Section 3. In Section 4, we provide our evaluation of CSDS algorithms. In Section 5 we identify and illustrate the benefits of ASCY. We describe in Section 6 the use of ASCY in the design of CLHT and BST-TK. We present related work in Section 7. We discuss the limitations of ASCY, and conclude the paper in Section 8.

2. Search Data Structures

Basic interface. A search data structure consists of a set of elements and three main operations: *search*, *insert*, and *remove*. An element consists of a *key* and a *value*. The key uniquely identifies the element in the set. The value is often a pointer to a structure that contains the actual data.

The three main operations have the following semantics:

- *search(key)* looks for an element with the given key; if it is found, returns the value of the element, otherwise returns `NULL`.
- *insert(key, val)* attempts to insert a new element in the data structure; the insertion is successful iff there is no other element with the same key.
- *remove(key)* attempts to remove the element with the given key; it is successful iff such an element exists.

A common attribute of search data structures is that the updates (insertions and removals) comprise two distinct phases. First, they *parse* the structure until the update point is reached. Then the actual modification is attempted.

Concurrent search data structures. We study the most basic and commonly-used CSDSs: *linked lists, hash tables, skip lists*, and *binary search trees (BSTs)*. We are interested in *linearizable* [31] implementations of the aforementioned data structures.

It is common to classify linearizable implementations based on whether and how they make use of locks [29]. One can distinguish *fully lock-based, hybrid lock-based*, and *lock-free* [20] algorithms. Fully lock-based algorithms use locks to protect all three operations and are *blocking* [29], in the sense that a thread might have to wait for a lock to be released. Hybrid lock-based (henceforth called "lock-based") algorithms use locks to protect the actual updates to the structure. They are otherwise lock-free. For instance, a removal might parse the list (in a lock-free manner) until the target node is found, get the lock, and then do the actual deletion. Hybrid algorithms are also blocking. Finally, lock-free algorithms do not use locks and are *non-blocking* [20, 27]. They typically use the underlying atomic operations, such as compare-and-swap (CAS), provided by the hardware.

3. The ASCYLIB Library

ASCYLIB contains 32 fully/hybrid lock-based and lock-free CSDSs, as well as 5 sequential implementations.[1] These include existing state-of-the art designs, and optimized versions, which we adapt in order to enable one, or more, ASCY patterns. In addition, ASCYLIB contains two novel CSDS algorithms built from scratch based on ASCY. Some implementations in ASCYLIB were initially based on the Synchrobench benchmark suite [22].

Algorithms. Table 1 contains a short description of the existing algorithms we implement.[2] ASCYLIB further contains 10 re-engineered (using ASCY) state-of-the-art CSDS designs. In particular, we apply ASCY_{1-2} on *harris* linked list and *fraser* skip list. We also apply ASCY_3 on *pugh*, *lazy*, and *copy* linked lists/hash tables, on *java* hash table, on *pugh* and *herlihy* skip lists, and on *drachsler* BST. Finally, we create a *urcu* hash-table variant that uses SSMEM (see below) instead of RCU for memory management and is closer to ASCY_4. In all our experiments, these optimizations result in better performance (see §4 and §5). We then use ASCY as the base to design two new CSDS algorithms, a lock-based and a lock-free variants of a hash table and a BST (see §6).

Memory management. We develop SSMEM, a memory allocator with epoch-based garbage collection (GC) [20]. SSMEM uses ideas similar to the RCU [41] mechanism: freed memory can only be reused once it is certain that no other thread holds a reference to this location. When some memory is freed, it does not become available until a GC pass decides that it is safe to be reused. The amount of garbage SSMEM allows before performing GC is configurable. Furthermore, SSMEM is non-blocking: it is based on per-thread counters that are incremented to indicate activity.

4. Evaluating the State of the Art CSDSs

In this section, we present a cross-platform evaluation of the state-of-the-art CSDS implementations and compare them with their asynchronized counterparts. We observe that regardless of the platform, the asynchronized executions perform the best. We further note that the algorithms with memory accesses to shared state that best resemble the asynchronized implementations are also the closest to these asynchronized upper bounds. Our evaluation also enables us to quantify the impact that various hardware features have on CSDS algorithms.

We start by describing the platforms and experimental settings used throughout this paper. We consider four multi-processors (with multiple sockets) and a chip multi-processor (with one socket). We also briefly experiment with an Intel Haswell desktop processor with hardware transactional-memory support.

[1] We use these as incorrect asynchronized CSDSs.

[2] The *urcu* and the *tbb* hash tables belong to the corresponding libraries and are not our own implementations.

	Name	Type	Short description
linked list	*async*	seq	A sequential linked list. We use it as an *incorrect asynchronized* concurrent set for performance upper bounds.
	coupling [29]	flb	All operations use hand-over-hand locking (grab next lock and release the previous) while parsing the list.
	pugh [49]	lb	Operations search/parse the list optimistically. Updates lock and then validate the target node. Removals employ pointer reversal so that a search/parse always finds a correct path.
	lazy [25]	lb	Nodes are deleted in two steps: marking and physical deletion. Searching/Parsing the list simply ignores marked nodes. Updates parse the list, grab the locks, validate the locked nodes, and perform the update.
	copy [48]	lb	Similar to Java's *CopyOnWriteArrayList*. Updates create new copies of the list and are protected by a global lock.
	harris [23]	lf	Nodes are deleted in two steps: mark with CAS and delete with a second CAS. Operations remove the logically deleted nodes while searching/parsing the list. If cleaning-up fails, searching/parsing is restarted.
	michael [44]	lf	A refactored implementation of *harris* for easier memory management.
hash table	*async*	seq	A sequential hash table. We use it as an *incorrect asynchronized* concurrent set for performance upper bounds.
	coupling [29]	flb	Uses one *coupling* list per bucket, with a single per-bucket lock.
	pugh [49]	lb	Uses one *pugh* list per bucket, with a single per-bucket lock.
	lazy [25]	lb	Uses one *lazy* list per bucket, with a single per-bucket lock.
	copy [48]	lb	Uses one *copy* list per bucket, with a single per-bucket lock.
	urcu [13]	lb	Part of the URCU (User-space RCU) (version 0.8) library. After each successful removal, it waits for all ongoing operations to complete before freeing the memory. Supports resizing.
	java [37]	lb	Similar to Java's *ConcurrentHashMap*. Protects the hash table with a fixed number of locks (we use 512 locks). Supports resizing.
	tbb [36]	flb	Part of Intel's Thread Building Blocks (version 4.2) library. Uses reader-writer locks. Supports resizing.
	harris [23]	lf	Uses one *harris-opt* list per bucket.
skip list	*async*	seq	A sequential skip list. We use it as an *incorrect asynchronized* concurrent set for performance upper bounds.
	pugh [49]	lb	Maintains several levels of *pugh* lists. Parses towards the target node without locking.
	herlihy [26]	lb	Update operations optimistically find the node to update and then acquire the locks at all levels, validate the nodes, and perform the update. Searches simply traverse the multiple levels of lists.
	fraser [20]	lf	Optimistically searches/parses the list and then does CAS at each level (for updates). A search/parse restarts if a marked element is met when switching levels. The same applies if a CAS fails.
bst	*async-int*	seq	A sequential internal BST. We use it as an *incorrect asynchronized* concurrent set for performance upper bounds.
	async-ext	seq	A sequential external BST. We use it as an *incorrect asynchronized* concurrent set for performance upper bounds.
	bronson [7]	lb	Partially external. A search/parse can block waiting for a concurrent update to complete.
	drachsler [15]	lb	Internal tree. Uses logical ordering to allow sequential read operations. Acquires ≥ 3 locks for removals.
	ellen [17]	lf	External tree. Updates help outstanding operations on the nodes that they intend to modify.
	howley [32]	lf	Internal tree. All three operations perform helping and might need to restart.
	natarajan [46]	lf	External tree. Minimizes the number of atomic operations and optimistically searches/parses the tree.

Table 1: A short description of the existing CSDS algorithms we consider. "seq" stands for sequential, "flb" for fully lock-based, "lb" for (hybrid) lock-based, and "lf" for lock-free.

Opteron. The 48-core AMD Opteron contains four Opteron 6172 [10] multi-chip modules (MCMs). Each MCM has two 6-core dies. It operates at 2.1 GHz and has 64 KB, 512 KB, and 5 MB (per die) L1, L2, and LLC data caches respectively.

Xeon20. The 20-core Intel Xeon consists of two sockets of Xeon E5-2680 v2 Ivy-Bridge 10-core (20 hyper-threads). It runs at 2.8 GHz and includes 32 KB, 256 KB, and 25 MB (per die) L1, L2, and LLC, respectively.

Xeon40. The 40-core Intel Xeon consists of four sockets of Xeon E7-8867L Westmere-EX 10-core (20 hyper-threads). It clocks at 2.13 GHz and offers 32 KB L1, 256 KB L2, and 30 MB (per die) LLC, respectively.

Tilera. The Tilera TILE-Gx36 [51] is a 36-core chip multi-processor. It clocks at 1.2 GHz and has 32 KB, 256 KB, and 9 MB[3] L1, L2, and L3 data caches, respectively.

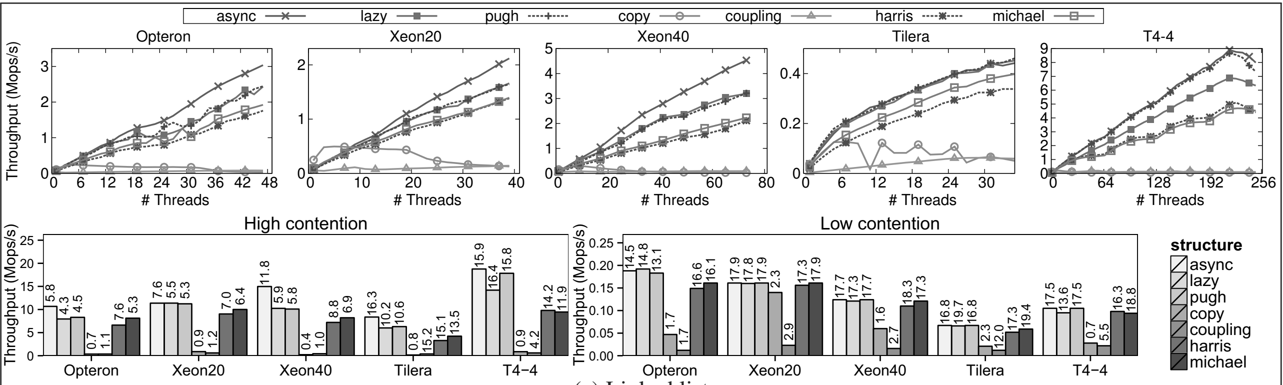
T4-4. The Oracle SPARC T4-4 is a four-socket multi-processor with 8 cores per socket and a total of 256 hardware threads (chip multi-threading). It operates at 2.85/3 GHz and has 16 KB, 256 KB, and 4 MB (per die) L1, L2, and LLC data caches, respectively.

Experimental settings. Each of our measurements represents the median value of 11 repetitions of 5 seconds each. We manually pin threads on cores in order to take advantage of the locality within sockets. Each operation is either a search, or an update, based on the update percentage we select. In general, we initialize the structure with a number of elements (N). The operations choose a key at random in the $[1 \ldots 2N]$ range. This provides the guarantee that on average, half of the operations are successful and that the structure size remains close to N (the update percentage is split to half insertions and half removals). On all architectures, except Tilera, we set SSMEM to trigger GC when 512 memory locations have been freed. On Tilera, we set this value to 128 in order to optimize for the smaller TLBs of 32 entries. On the asynchronized implementations, we disable GC to avoid data corruption. Finally, we use 64-bit long keys and values. It is straightforward to replace both with larger structures.

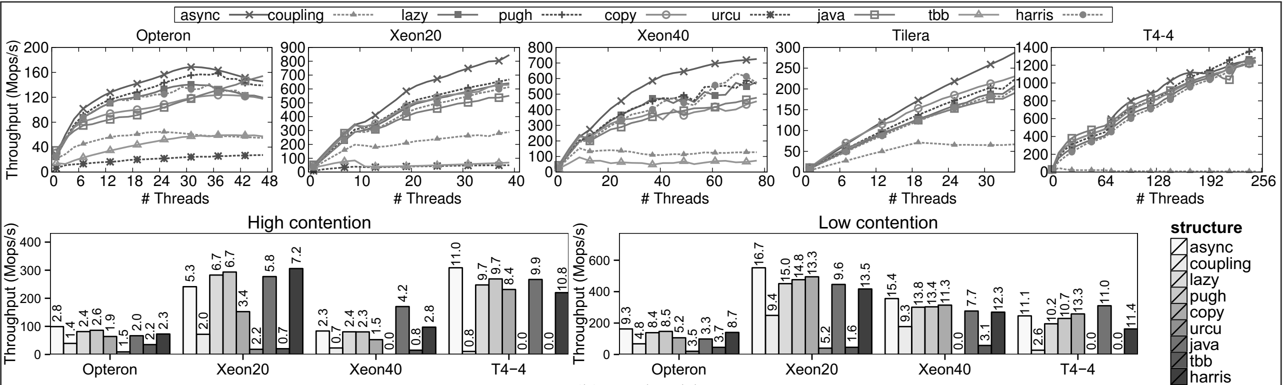
Cross-Platform evaluation. We now look at the behavior of a large sample of the state-of-the-art CSDS algorithms, as presented in Table 1. Figure 2 shows cross-platform results on various workloads, spanning low, average, and high degrees of contention. The histograms plot the throughput and the scalability ratio compared to the single-threaded execution (on top of each bar) on 20 threads.

On average and low-contention levels, algorithms exhibit good scalability in terms of throughput: in the experiments with 20 threads, the average scalability of the best

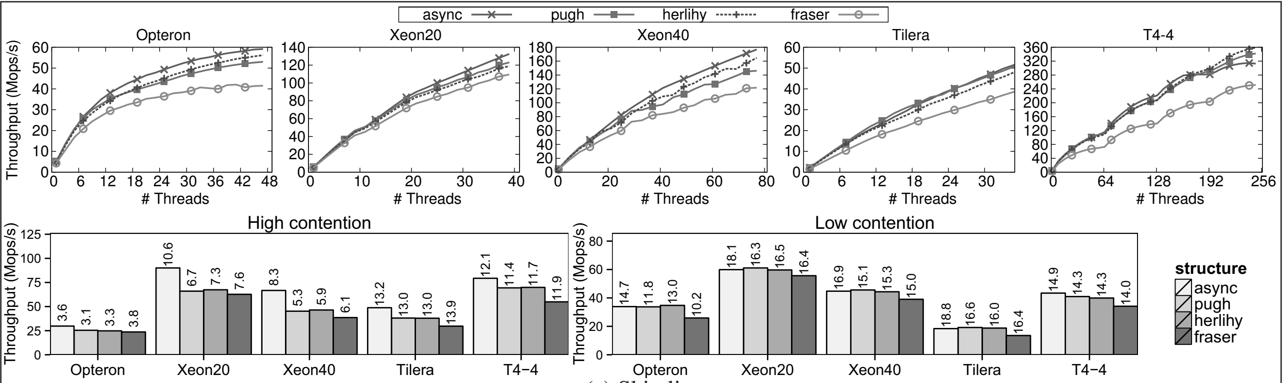
[3] The 36 L2 caches are utilized as a distributed LLC.



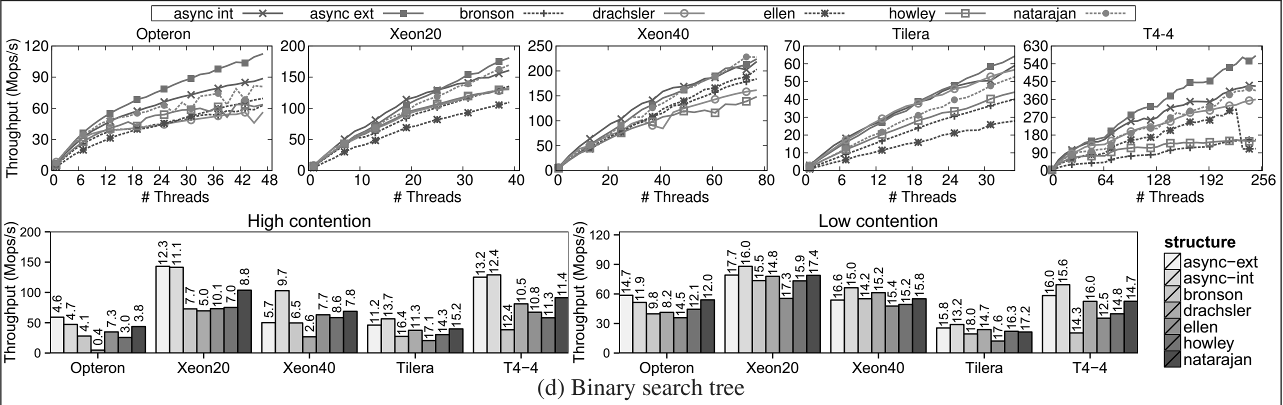
(a) Linked list



(b) Hash table



(c) Skip list



(d) Binary search tree

Figure 2: Cross-platform results of the data structures in ASCYLIB on average (top graphs – 4096 elements, 10% updates), high (20 threads, 512 elements, 25% updates), and low contention (20 threads, 16384 elements, 10% updates).

performing CSDS algorithm (per data structure) is 16.2 in the low contention case, whereas for the average and high-contention levels, this value is 14.1 and 9.8, respectively. While trends are generally valid across platforms, we do notice a certain variability: for each workload, the standard deviation of the average scalability values of different platforms is $\sim$2. In some cases, scalability trends differ significantly between platforms. For instance, on the Opteron the hash tables do not scale beyond 32 threads (including *async*), due to bandwidth limitations. In short, the main hash table structure is initialized from a single memory node, thus all requests are directed to that node. This problem does not emerge on *java*, because of the fine-grained resizing (per one of the 512 regions) that spreads the hash table on multiple nodes.

In addition, we observe that there are various algorithms per data structure that are the "best". On linked lists (Figure 2a), *pugh* is consistently competitive across workloads and platforms, but *lazy* is close in throughput. On hash tables (Figure 2b), several algorithms perform close to each other (e.g., *pugh*, *lazy*, *copy*, *harris*). On skip lists (Figure 2c), *herlihy* and *pugh* perform similarly. Finally, on BSTs (Figure 2d), *natarajan* is generally the best. Overall, we see that, per data structure, both lock-based and lock-free algorithms are close in terms of performance. Lock-freedom is more important when we employ more threads than hardware contexts (not shown in the graphs). In these deployments, lock-freedom provides better scalability than lock-based designs.

It is worth noting that the workloads we evaluate are uniform: the frequency and the distribution of updates are constant. We briefly experiment with non-uniform workloads (not shown in the graphs), such as those with update spikes and continuously increasing structure size. We notice that our observations are valid in these scenarios as well.

Dissecting asynchronized executions. In Figure 2, we also depict the behavior of the asynchronized implementations for each data structure. We notice that except for some corner cases, the asynchronized implementations outperform alternatives on all the platforms. The reason for the rare corner cases in which some concurrent implementations can perform better than *async* is that asynchronized structures can become malformed in concurrent scenarios. For instance, an update operation on a skip list could update several pointer fields. We observe that these pointers are sometimes not properly set due to concurrency, leading to longer average path lengths.

For each data structure and regardless of the platform and workload, there is at least one concurrent algorithm that performs and scales close to the asynchronized implementation (*async*) of the data structure. On average, the best concurrent implementations are 10% slower than their asynchronized counterparts and exhibit similar scalability trends. The next section explores methods in which the portable scalability of these algorithms can be improved even further. Our

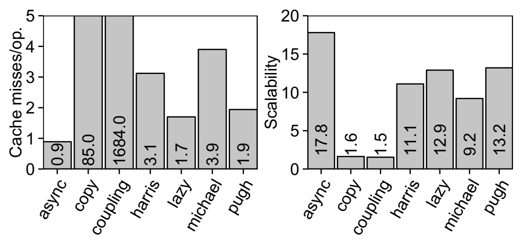


Figure 3: Cache misses per operation and scalability for various linked-list algorithms.

empirical results thus confirm our intuition: asynchronized executions represent good approximations for the ideal behavior of CSDSs.

Intuitively, the asynchronized implementations perform and scale better than the concurrent alternatives because they trigger cache coherence less (i.e., fewer cache-line transfers): they modify only the data that is semantically necessary, as they do not employ synchronization, thus leading to a smaller number of memory stores. Cache coherence is the source of the most significant scalability bottleneck for concurrent algorithms on multi-cores, because the number of cache-line transfers tends to increase with the number of threads. Hence, it is essential for a CSDS algorithm to limit the amount of cache traffic it performs during each operation, which is directly linked to the number of stores on shared data. Stores cause cache-line invalidations, which in turn generate cache misses of future accesses.

We confirm this line of reasoning by showing a practical correspondence between the number of cache misses and the performance of an algorithm. In doing so, we use linked-list algorithms as an example. Figure 3 shows the number of cache misses per operation generated by various linked-list algorithms, as well as their scalability compared to single-thread throughput. We use a workload where the list has 4096 elements on average, 10% of the operations are updates, and 20 threads concurrently access the data structure. Clearly, the asynchronized execution has the fewest number of cache misses. It is also interesting to note that the number of cache misses per operation is directly correlated to the scalability and performance of the algorithms: the fewer cache misses an algorithm generates, the better it scales. This correlation pertains to the other data structures as well.

We also take a closer look at how the "best" concurrent algorithms access memory. We look at the memory access pattern of a CSDS algorithm by studying the number of loads and stores (often performed through read-modify-write instructions), as well as the branches in each operation and phase of the algorithm. We notice that the CSDS algorithms that tend to scale and perform the best also tend to have an average number of loads, stores, and branches closer to the asynchronized implementations than the alternatives. This is generally valid for all the operations and phases. Thus, we make the observation that the more an algorithm's memory access pattern resembles that of the asynchronized execu-

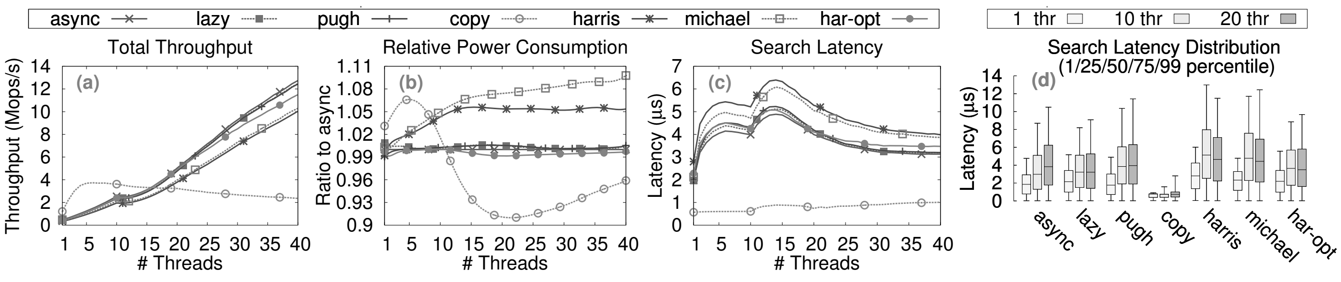


Figure 4: Linked list with 1024 elements and 5% updates (2.5% successful).

tion, the better it scales regardless of the platform and workload. In the next section, we look at ways in which this similarity can be achieved and validate this observation.

Hardware considerations. Our evaluation has revealed a number of other hardware-related observations.

We fine-tune several algorithms in ASCYLIB using Intel's TSX [34] hardware transactional memory (HTM), in order to assess whether HTM can be used to optimize CSDSs. Basically, in 60% of the scenarios, HTM improves throughput with up to 5%. In the remaining 40%, the results are either unaffected, or the throughput decreases (up to 5% as well). We use a 4-core desktop processor (8 hyper-threads) with the first iteration of Intel's TSX. As larger machines become available and the technology matures,[4] HTM might become even more helpful.

We also encounter a number of hardware-related bottlenecks, not specific to CSDSs. For example, the small TLBs on the Tilera (32 entries) require feeding SSMEM with small chunks of memory and keeping the amount of garbage low. Otherwise, if the threads use largely fragmented memory, the TLB misses become a bottleneck. Similarly, on Xeon40, large chunks of memory with a lot of garbage can lead to an excessive number of hardware prefetches that decrease performance up to an order of magnitude. On the Opteron, the interconnect bandwidth becomes a bottleneck when the structure is allocated on a single memory node. If a structure fits on one memory page, there is no straightforward solution to this problem, other than restructuring the data structure. System designers should be aware of the aforementioned issues as they can emerge in all types of software.

5. The ASCY Patterns

Having observed that CSDSs that resemble sequential algorithms scale and perform best, we now look at how this similarity can be achieved. We identify four patterns which are applicable to a broad class of CSDS algorithms and we show that when they are applied, the performance and scalability of CSDSs is systematically improved. These patterns collectively represent *asynchronized concurrency (ASCY)*.

For brevity, we select Xeon20 as the platform in our experiments. This is the most modern processor within our platform set. Also for brevity, we break down the results for each pattern using one of the four data structures. Note that we get similar results on any platform and data-structure combination: the patterns are globally beneficial.

ASCY$_1$. We first examine the search operation of CSDSs and use linked lists as a case study. Figure 4 depicts the behavior of the various linked lists of ASCYLIB on a search-dominated workload (only 2.5% successful updates).

Out of the existing algorithms, the *async*, *lazy* and *pugh* lists deliver the highest total throughput. Both *lazy* and *pugh* linked lists have a search that is identical to the sequential algorithm. Essentially, no stores, waiting, or restarting is involved. These algorithms perform within 10% of *async*. In comparison, the lock-free lists (*harris* and *michael*) diverge from the sequential code as they try to physically remove logically-deleted nodes using CAS. If a physical removal fails, the operation is restarted. Additionally, they also need to unmark[5] every pointer while traversing the list.

Overall, the results can be largely explained by the average search latencies (Figure 4(c)): the closer to the sequential an implementation is, the lower the latency is. On more than 20 threads, the search latencies decrease due to the effects of hyper-threading; the two hyper-threads of a core help each other by keeping the list nodes warm in the shared L1 cache.

We thus identify ASCY$_1$ as a generic pattern: *The search operation should not involve any stores, waiting, or retries.*

We apply this pattern to the search operation of existing algorithms: in the case of linked lists, we apply it to the *harris* lock-free list by removing the physical removal of logically deleted elements from the search operation. We refer to the resulting algorithm as *harris-opt*.

If we now look at the three lock-free algorithms, namely *harris*, *michael*, and *harris-opt*, the effects of applying ASCY$_1$ become evident. In both *harris* and *michael*, the search tries to unlink logically deleted nodes and restarts if it fails, hence violating ASCY$_1$. In contrast, *harris-opt* ignores the deleted nodes while searching. The latency improvements due to ASCY$_1$ are approximately 10-30% as we can observe on the latency graphs (Figure 4(c) & (d)). Additionally, *harris-opt* has a tighter latency distribution (Figure 4(d)) than the other two. *harris-opt* provides more stable

[4] A recent announcement by Intel [35] suggests that this might take a while.

[5] Clear the least significant bit that indicates a logically deleted element.

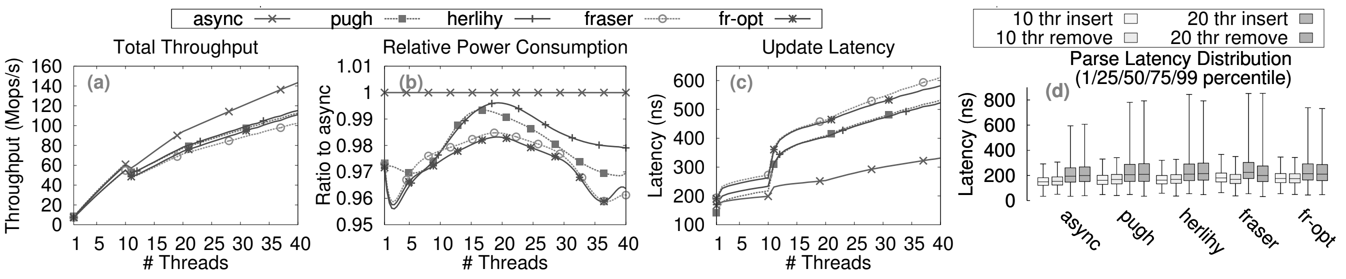


Figure 5: Skip list with 1024 elements and 20% updates (10% successful).

executions because it never restarts or invokes work that is unnecessary to the search. Furthermore, Figure 4(b) plots the power consumption relative to *async*. We observe that *harris* and *michael*, which do not follow ASCY$_1$, deliver lower performance while consuming more power than the rest.

Additionally, the behavior of the *copy* list is worth analyzing. It is apparent from the latency graphs that structuring the data as an array can bring tremendous benefits on serial data accesses. However, *copy* has two major limitations: (i) high memory overhead, as every update creates a new list copy, and (ii) synchronization of updates with a global lock, which easily becomes a bottleneck. In §6.1, we use the idea of array-based structures in the design of a hash table.

Finally, we apply ASCY$_1$ to the *fraser* skip list (not shown in the graphs) and observe performance improvements. In most cases, applying ASCY$_1$ means deferring cleaning-up and helping to the update methods. ASCY$_1$ also requires that updates always leave the data structure in a state that allows any existing node to be found. In addition, when removing nodes, their memory should not be freed while there is the possibility of an ongoing search accessing it. Memory reclaimation is handled by SSMEM in ASCYLIB.

ASCY$_2$. We now focus on the parse phase of the update operations. In a sequential algorithm, this phase is basically identical to the search operation. In CSDSs however, parsing might involve helping, cleaning-up the data structure, or restarting the operation (e.g., due to a failed clean-up attempt). We study this phase more closely using skip-list algorithms.

Figure 5 depicts the behavior of the five skip lists in ASCYLIB on a workload with 10% successful updates. The best performing pre-existing algorithms are *pugh* and *herlihy*, which are within 22% of the asynchronized version. Looking closer at their parse phase, we note that the only stores that are performed are for cleaning-up purposes and that the parse is never restarted. In contrast, a parse in *fraser* might have to restart due to a failed clean-up attempt, or accessing a logically deleted node when changing levels.

Taking these results into account, we establish ASCY$_2$: *The parse phase of an update operation should not perform any stores other than for cleaning-up purposes and should not involve waiting, or retries.*

We apply this pattern in conjunction with ASCY$_1$ (based on [30]) to the *fraser* skip list and refer to the resulting algorithm as *fraser-opt*. *fraser-opt* delivers up to 8% better throughput than *fraser* and has a 5% lower average update latency (Figure 5(c)). Furthermore, Figure 5(d) plots the latency distribution of the parsing phase only. The behavior of both *fraser* and *fraser-opt* is similar. However, the latter eliminates the overhead of useless parses that have to be restarted. *fraser* performs 0.38%, 1.07%, and 1.82% more parses than updates on 10, 20, and 40 threads, respectively. *fraser-opt* has lower overheads: 0.03%, 0.09%, and 0.17%, respectively.

In terms of power consumption (Figure 5(b)), there are two main observations. First, ASCY not only improves the throughput of *fraser*, but also leads to an algorithm that consumes slightly less power than the initial design. Second, in the case of skip lists, the lock-based algorithms seem to consume more power than their lock-free counterparts.

Finally, we also apply ASCY$_2$ to the *harris* linked list. In practice, applying ASCY$_2$ means (i) not helping other threads while parsing the data structure and (ii) avoiding restarting if cleaning-up fails. Similar to ASCY$_1$, for ASCY$_2$ to work, concurrent updates should not render portions of the data structure unreachable.

ASCY$_3$. Another characteristic of sequential algorithms is that if an update operation cannot be completed (i.e., the parse does not find the node in case of a remove, or it finds the node in case of an insert), no additional stores are performed and the update method simply returns "false". This is not always the case in existing CSDS algorithms.

We quantify the impact of this issue by looking at hash-table algorithms. Figure 6 depicts the performance of various hash tables with and without (suffixed with "*-no*") the "read-only fail" of the sequential algorithms. In terms of throughput, algorithms doing no additional stores after unsuccessful parses perform up to 12.5% better than their counterparts doing stores. Additionally, they are within 19-30% of the *async* version. Although this change alters the behavior of only 5% of all operations in this workload, we observe throughput benefits up to 12.5% on *java*. This difference can be attributed to (i) the increased number of cache misses caused by unnecessary synchronization, and (ii) the increase of con-

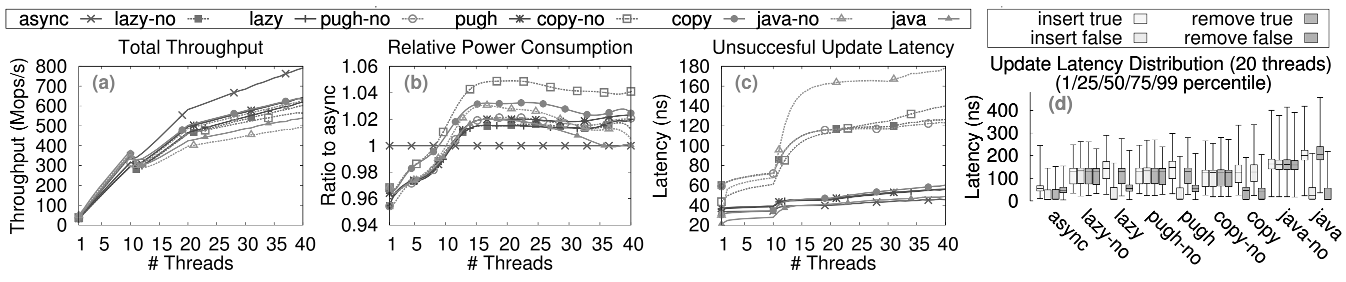


Figure 6: Hash table with 8192 elements, 8192 (initial) buckets, and 10% updates (5% successful).

tention on locks. Basically, (i) manifests as a 2.8% increase in the cache-miss ratio on 40 threads, and (ii) as a 14% increased chance to find a lock occupied.

Given the apparent benefits, we model the behavior of the sequential algorithms through ASCY$_3$: *An update operation whose parse phase is unsuccessful should not perform any stores, besides those used for cleaning-up in the parse phase.*

Figure 6(c) details the benefits of ASCY$_3$ on the average latency of unsuccessful updates. It is clear that turning a failed update into a read-only operation yields a significant (1.5-4x) decrease in latencies. Nevertheless, depending on the algorithm, applying ASCY$_3$ can incur some overhead on successful updates, as we notice on the graph (d). For instance, enabling ASCY$_3$ on *java* requires an additional search to either decide that the update cannot succeed, or proceed to the actual update. In general ASCY$_3$ also reduces the power consumption of the CSDSs. This is achieved by decreasing the number of cache-line transfers.

Finally, we apply this pattern to multiple other existing algorithms: the *pugh*, *lazy*, and *copy* linked lists, the *pugh* and *herlihy* skip lists, and the *drachsler* BST. In many cases, applying ASCY$_3$ simply means checking the outcome of the parse and returning "false" without locking. The algorithms to which we apply ASCY$_3$ trivially maintain correctness, as unsuccessful updates can be seen as search operations.

ASCY$_4$. We now focus on the modification phase of the update operation. We use BSTs as an example for this scenario. Figure 7 includes the corresponding results.

Aside from the asynchronized algorithms, the best performing concurrent implementation is *natarajan*. We argue that the other four concurrent trees synchronize more than the minimum. Indeed, we measure the ratio of atomic operations to the number of successful updates on the same workload as Figure 7. *natarajan* uses two atomic operations per update on average, which is close to the asynchronized versions, whereas the other concurrent trees require more than three. In fact, *natarajan* is also the closest to *async* in terms of the number of stores and the number of affected cache lines. This major difference, together with the differences in the first three ASCY patterns, is reflected in the results, in terms of throughput, latency, and power consumption.

More precisely, *howley* employs helping even while searching or parsing the tree. *ellen* uses helping only on elements that the current operation wants to update. Helping is generally expensive, as it requires additional synchronization in order to be implemented. *bronson* is a complex algorithm that can block waiting for an update to complete. Finally, *drachsler* acquires a large number of locks (3.15 on average) for each successful update.

Figure 7(d) depicts the latency distribution of successful-only operations, isolating the effects of the modification phases. Clearly, the *natarajan* tree has lower latencies and a tighter distribution than the rest. Interestingly, *natarajan* consumes less power than two of the other four concurrent trees, similar power to *drachsler*, and more power than *howley*. *natarajan* simply performs at speeds different than the rest: on 40 threads, it issues 797M memory accesses per second with 13.4% cache-miss ratio, compared to the 578M/23.8% of *drachsler* and the 395M/18.7% of *howley*.

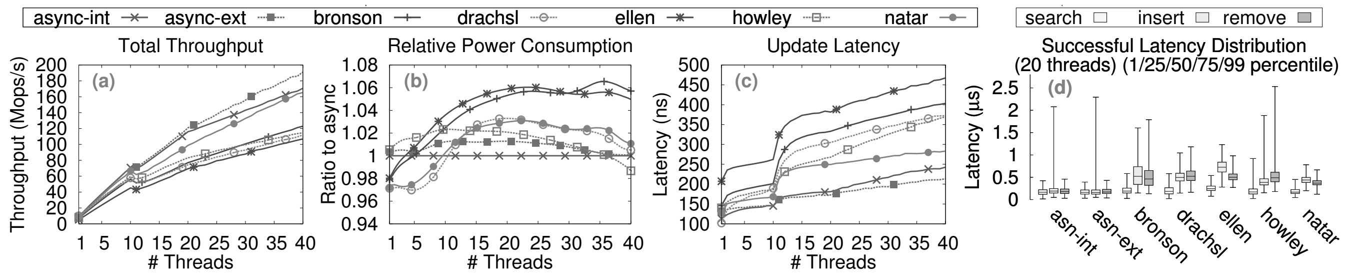


Figure 7: BST with 2048 elements and 20% updates (10% successful).

Still, *drachsler* and *howley* consume 41% and 49% more energy per operation than *natarajan*, respectively.

We can thus identify ASCY$_4$: *The number and region of memory stores in a successful update should be close to those of a standard sequential implementation.*

Essentially, ASCY$_4$ means that updates should not block each other or write to the same memory addresses unless they operate on semantically related elements, such as adjacent nodes in the data structure.

Discussion. It is worth noting that while each pattern can be identified in some of the existing algorithms, we have shown that CSDS algorithms can be improved even further when the ASCY patterns are applied collectively. Moreover, as seen in §4, for each data structure and platform combination, there is an algorithm that is reasonably close in performance to the asynchronized executions, in general a CSDS that resembles the sequential algorithm. In this section, by applying ASCY and bringing these algorithms even closer to their sequential counterparts, we have further improved their performance and scalability. We can therefore conclude that ASCY can help reach implementations that are portably scalable.

6. Designing with ASCY from Scratch

We illustrate the use of ASCY on the design of two new search data structure algorithms: (i) a hash table (CLHT), and (ii) a binary search tree (BST-TK). Due to the limited space, we only present the high-level ideas of the algorithms and refer the reader to [11] for further details.

6.1 Cache-Line Hash Table (CLHT)

CLHT captures the basic idea behind ASCY: avoid cache-line transfers. To this end, CLHT uses cache-line-sized buckets and, of course, follows the four ASCY patterns. As a cache-line block is the granularity of the cache-coherence protocols, CLHT ensures that most operations are completed with *at most* one cache-line transfer.

CLHT uses the 8 words of a cache line as:

concurrency	k_1	k_2	k_3	v_1	v_2	v_3	next

The first word is used for concurrency-control; the next six are the key/value pairs; the last is a pointer that can be used to link buckets. Updates synchronize based on the `concurrency` word and do in-place modifications of the key/value pairs of the bucket. To support in-place updates, the basic idea behind CLHT is that a search/parse does not simply traverse the keys, but obtains an *atomic snapshot* of each key/value pair[6]:

```
val_t val = bucket->val[i];
if (bucket->key[i] == key && bucket->val[i] == val)
    /* atomic snapshot of key/value */
```

[6] For an atomic snapshot to be possible, the memory allocator of the values must guarantee that the same address cannot appear twice during the lifespan of an operation. Additionally, the implementation has to handle possible compiler and CPU re-orderings (not shown in the pseudo-code)

We design and implement two variants of CLHT, lock-based (CLHT-LB) and lock-free (CLHT-LF).

CLHT-LB. The lock-based variant of CLHT uses the `concurrency` word as a lock. Search operations traverse the key/value pairs and return the value if a match is found. Updates first perform a search to check whether the operation is at all feasible (recall ASCY$_3$) and if so, they grab the lock, apply the update, and release the lock. If there is not enough space for an insertion, the operation either links a new bucket by using the `next` pointer, or resizes the hash table.

CLHT-LF. The lock-free variant of CLHT is more elaborate than the lock-based. This is due to the fact that the key/value-pair insertions have to appear atomic. With locks, we implement atomicity by allowing for a single concurrent writer per bucket. However, without locks, several updates can concurrently alter the same key or value. Even worse, if concurrent insertions on the same bucket do not synchronize, there is no way to avoid duplicate keys on different slots.

In order to solve these complications, we devise the `snapshot_t` object. `snapshot_t` handles a word (8 bytes) as an array of bytes (*map*) with a *version* number:

```
struct snapshot_t {
  uint32_t version;  /* a 4-bytes integer */
  uint8_t  map[4];   /* an array of 4 bytes */
};
```

Naturally, `snapshot_t` occupies the `concurrency` word of a bucket. `snapshot_t` provides an interface to atomically get or set the value of an index in the map. The `version` number is used to enable sets/gets to do atomic changes with respect to the other spots in the map. In short, atomicity is implemented by reading the value of the `snapshot_t` object before the atomic section and by using the version number to get/set the target index in the map using a CAS on the whole object. For instance, if a another concurrent insertion has already been completed, the current operation will fail the CAS, because the version number will be different. We then use the fields of the map as flags that indicate whether a given key/value pair is valid, invalid, or is being inserted.

Evaluation. We compare CLHT to *pugh*, one of the best performing hash tables in §4. In contrast to the linked-based hash tables, CLHT performs in-place updates, thus avoiding memory allocation and garbage collection of hash-table nodes. Nevertheless, we use the SSMEM allocator for values.

Figure 8 includes the results. Noticeably, *clht-lb* and *clht-lf* outperform *pugh* by 23% and 13% on average, respectively. CLHT's design significantly reduces the number of cache-line transfers. For example, on the Opteron for 20% updates, *clht-lb* requires 4.06 cycles per instruction, *clht-lf* 4.24, and *pugh* operates with 6.57. Interestingly, *clht-lb* is consistently better than *clht-lf* on 20 threads. On more threads (e.g., 40), however, *clht-lf* often outperforms *clht-lb*.

Discussion. CLHT supports operations with keys up to 64-bits. To support longer keys, the 64-bit keys in CLHT can be used as a first filter. The operation has to compare the full key, that is stored separately, only if there is a match with

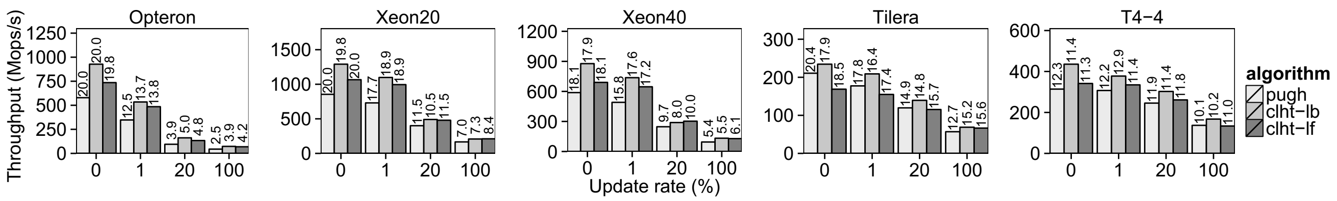


Figure 8: CLHT with 4096 elements on 20 threads for various update rates.

the 64-bit filter. This technique has already been shown to work well in practice [19].

6.2 BST Ticket (BST-TK)

BST-TK reduces the number of cache-line transfers by acquiring less locks than existing BSTs. Intuitively, on any lock-based BST an update operation parses to the node to modify and, if possible, acquires a number of locks and performs the update. This is precisely how BST-TK proceeds.

More specifically, BST-TK is an external tree, where every internal, router node is protected by a lock and contains a version number. The version numbers are used in order to be able to optimistically parse the tree and later detect concurrency. Update operations proceed as described in Figure 10. We are able to simplify the design of BST-TK by making the simple observation that a ticket lock already contains a version field. Accordingly, we consolidate steps 3 and 6, with steps 4 and 7 respectively. We do this by modifying the interface of the ticket lock in order to try to acquire a specific version of the lock (i.e., the one that the parsing phase has observed). If the lock acquisition fails, the version of the lock has been incremented by a concurrent update, hence the operation has to be restarted. We further optimize the tree by assigning two smaller (32-bits) ticket locks to each node, so that the left and the right pointers of the tree can be locked separately. Overall, BST-TK acquires one lock for successful insertions and two locks for successful removals.

Evaluation. We compare BST-TK to *natarajan*, the best performing BST in §4. Figure 9 depicts the results. In general, *bst-tk* behaves very similarly to *natarajan* (within 1% on average). It might have been expected that *bst-tk* would outperform the latter, because it uses less atomic operations per update. Although this is true, *bst-tk* has slightly increased parsing overhead compared to *natarajan* (0.045% vs. 0.032% with 20% updates on the Opteron). For simplicity, we did not implement certain optimizations that could prove beneficial

```
update()
1. parse() /* keeps track of the versions numbers */
2. if (!can_update()) { return false; } /* ASCY3 */
3. lock() /* 1 node for insert, 2 nodes for remove */
4. if (!validate_version()) { goto 1; }
5. apply_update()
6. increase_version()
7. unlock()
```

Figure 10: Update operations in BST-TK.

under high contention. For instance, an insertion does not have to be restarted if the router node is locked by another insertion. Instead, it can be blocked and wait for the ongoing insertion to finish and then proceed almost normally.

7. Related Work

CSDS design. A large body of work has been dedicated to the design of efficient CSDSs [7, 15, 17, 20, 25, 29, 32, 41, 43, 44, 46, 49, 50, 52, 53]. These efforts usually aim at optimizing a specific CSDS (e.g., BST) in terms of throughput, for a specific set of workloads and platforms. In contrast, ASCY is a paradigm that targets the scalability of various CSDSs across different platforms, for various workloads, and according to several performance metrics.

Memory reclamation is of key importance in CSDSs [6, 14, 16, 28, 45]. Various techniques have been proposed, such as *quiescent state* [13, 23, 24], *epochs* [20], *reference counters* [14, 21, 53], *pointers* [6, 28, 45], and, recently, hardware transactional memory [1, 16]. ASCYLIB uses an epoch-based allocator that reduces the performance overheads.

Read-copy update (RCU) [41] is a concurrent programming technique that is heavily used in the Linux Kernel [3]. In short, RCU guarantees that readers always find a consistent view of the data structure. Writers perform atomic updates, after which, in the case of removals, they wait for all concurrent readers to finish in order to perform memory reclamation. Relativistic programming [52] is a technique

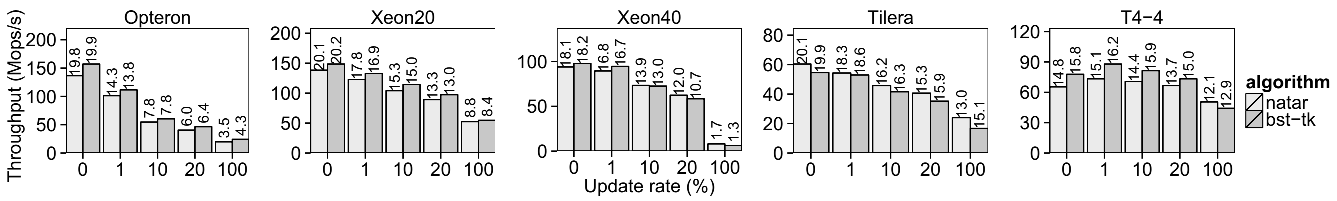


Figure 9: BST-TK with 4096 elements on 20 threads for various update rates.

that is related to RCU and maintains efficient reads even with large updates to the data structure (e.g., a resize). Arbel and Attiya [2] present an RCU-like search tree which allows concurrent updates. While out-performing classic RCU-based structures, the design is still shown to lag behind other state-of-the-art CSDSs that are closer to ASCY, in particular in write-intensive scenarios. In general, RCU-like approaches purposefully optimize the performance of search operations at the possible expense of updates. Our ASCY_1 pattern is similar in the sense that it dictates that readers must be unaware of concurrency. However, our three remaining patterns achieve the benefits of sequential reads without sacrificing the performance of updates.

Hunt et al. [33] study the energy efficiency of lock-free and lock-based concurrent data structures (queues and linked lists). They find that lock-free algorithms tend to be more energy efficient than their lock-based counterparts. We observe that in the context of CSDSs, as long as the algorithms apply ASCY, there is no inherent difference between lock-free and lock-based algorithms. Essentially, this is because the ASCY patterns help reduce the number and the granularity of locks.

Scalability. Recent work [12] has argued that scalability of synchronization is mainly a property of the hardware. In particular, synchronization primitives are shown to be inherently non-scalable on NUMA architectures due to expensive cache-line transfers. To bypass these problems, ASCY reduces the amount of synchronization on CSDSs, leading to designs that scale even in the presence of non-uniformity.

Clements et al. [9] link commutative interfaces to the existence of scalable implementations. In essence, they argue that commutative operations can lead to cache conflict-free implementations, that are inherently scalable from the memory-system point of view. Although basic CSDS interfaces commute, as defined by Clements et al., certain structures such as lists and trees do not allow for conflict-free implementations. We show, however, that even in these cases, scalable algorithms can be devised following ASCY.

Data structures in systems. Boyd-Wickizer et al. [5] performed a scalability study of the Linux kernel. They identify several bottlenecks, among which, one in the directory entry lookup operation (even though it is optimized using RCU). In general, numerous key-value stores, such as Memcached [42], SILT [38] or Masstree [39] are based on a CSDS. In some cases, these structures have been shown to be scalability bottlenecks, as for example in Memcached [5, 19, 47]. Fan et al. [19] achieve a 3-fold performance increase over the traditional Memcached, mainly by optimizing its hash table. ASCY can be used to recognize possible optimizations in systems such as Memcached and develop scalable CSDS implementations.

Baumann et al. [4] argue that in principle, non-portable, hardware-specific optimizations in the OS kernel should be removed. They attribute the existence of such optimizations to "the basic structure of a shared-memory kernel with data structures protected by locks" and propose to rethink the OS structure. We show that, by applying ASCY, we reach portably-scalable implementations. We believe that ASCY can help alleviate the difficult problem of CSDSs in OSes.

8. Concluding Remarks

This paper introduced *asynchronized concurrency (ASCY)*: a paradigm consisting of four complementary programming patterns to govern the design of portably scalable concurrent search data structures (CSDSs). We showed that ASCY can be used both to optimize existing algorithms and to assist in the design of new ones. In particular, using ASCY, we have optimized 10 state-of-the-art algorithms and designed 2 new algorithms from scratch, a hash table (CLHT) and a binary search tree (BST-TK). These are part of ASCYLIB, a new CSDS library that contains 34 highly-optimized cross-platform implementations of linked lists, hash tables, skip lists, and BSTs. ASCYLIB is available at `http://lpd.epfl.ch/site/ascylib`.

It is important to note that it is not always straightforward to apply some of the ASCY patterns. For instance, internal BSTs require either helping (e.g., *ellen*) or additional structures (e.g., *drachsler*) to implement ASCY_{1-2}. Similarly, in order to apply ASCY_3 on some lock-based hash tables, such as *java* and CLHT, we have to add a complete search operation before starting with the code of the update. As conveyed by our results, doing so is beneficial overall, because it reduces the coherence traffic. Enabling ASCY in these cases, however, results in overhead in successful updates.

Clearly, we expect ASCY to be applicable to other *search* data structures, such as prefix-trees, or B-trees. However, given that the ASCY patterns are based on the breakdown of operations to search and to parse-then-modify updates, some of the patterns might be meaningless for other abstractions such as queues and stacks. Still, we argue that the basic principle of ASCY (i.e., bring the concurrent-software design close to the asynchronized one) is generally beneficial.

Finally, it is important to note that we have focused this work on the basic CSDS interface, which is the common denominator for all search data structures. We did not consider data-structure-specific operations, such as *iterations, move*, or *max*. It is not clear whether it is easy, or possible, to implement these on top of ASCY-compliant CSDSs.

Acknowledgments

We wish to thank the anonymous reviewers for their helpful comments. We also like to thank Anne-Marie Kermarrec for her comments in the early stages of this work, as well as, Eric Ruppert for the fruitful discussions while designing the CLHT hash table. Finally, we wish to thank Oracle Labs, the EPFL DIAS lab, and EcoCloud for providing us with some of the platforms employed in this work. Part of this work was funded from the SNSF projects 200021-143915 (Reusable Concurrent Data Types) and 200021-147067 (Scalable and Secure Computing in Social Networks).

References

[1] Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. StackTrack: An Automated Transactional Approach to Concurrent Memory Reclamation. EuroSys 2014.

[2] Maya Arbel and Hagit Attiya. Concurrent Updates with RCU: Search Tree As an Example. PODC 2014.

[3] Andrea Arcangeli, Mingming Cao, Paul E McKenney, and Dipankar Sarma. Using Read-Copy-Update Techniques for System V IPC in the Linux 2.5 Kernel. USENIX ATC 2003.

[4] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. SOSP 2009.

[5] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. OSDI 2010.

[6] Anastasia Braginsky, Alex Kogan, and Erez Petrank. Drop the anchor: lightweight memory management for non-blocking data structures. SPAA 2013.

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

[8] Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using RCU balanced trees. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 199–210. ACM, 2012.

[9] Austin T Clements, M Frans Kaashoek, Nickolai Zeldovich, Robert T Morris, and Eddie Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. SOSP 2013.

[10] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE Micro*, 30(2):16–29, March 2010.

[11] Tudor David, Rachid Guerraoui, Tong Che, and Vasileios Trigonakis. Designing ASCY-compliant Concurrent Search Data Structures. Technical report, EPFL, Lausanne, 2014.

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

[13] 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.

[14] David L Detlefs, Paul A Martin, Mark Moir, and Guy L Steele Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002.

[15] Dana Drachsler, Martin Vechev, and Eran Yahav. Practical Concurrent Binary Search Trees via Logical Ordering. PPoPP 2014.

[16] Aleksandar Dragojević, Maurice Herlihy, Yossi Lev, and Mark Moir. On the power of hardware transactional memory to simplify memory management. PODC 2011.

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

[18] Facebook. RocksDB. `http://rocksdb.org`.

[19] Bin Fan, David G Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. NSDI 2013.

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

[21] Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *Parallel and Distributed Systems, IEEE Transactions on*, 20(8):1173–1187, 2009.

[22] Vincent Gramoli. More than You Ever Wanted to Know about Synchronization. PPoPP 2015.

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

[24] Thomas E Hart, Paul E McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, 2007.

[25] 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.

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

[27] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. ICDCS 2003.

[28] Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: a mechanism for supporting dynamic-sized lock-free data structures. Technical report, 2002.

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

[30] Maurice P Herlihy, Yosef Lev, and Nir N Shavit. Concurrent lock-free skiplist with wait-free contains operator, May 3 2011. US Patent 7,937,378.

[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] Nicholas Hunt, Paramjit Singh Sandhu, and Luis Ceze. Characterizing the performance and energy efficiency of lock-free data structures. INTERACT 2011.

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

[35] Intel. Intel xeon processor e3-1200 v3 product family - specification update. `http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e3-1200v3-spec-update.pdf`, 2014.

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

[37] 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.

[38] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. SOSP 2011.

[39] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. EuroSys 2012.

[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] Zviad Metreveli, Nickolai Zeldovich, and M Frans Kaashoek. CPHASH: A Cache-partitioned Hash Table. PPoPP 2012.

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

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

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

[47] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. NSDI 2013.

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

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

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

[51] Tilera. Tilera TILE-Gx. `http://www.tilera.com/products/processors/TILE-Gx_Family`.

[52] Josh Triplett, Paul E McKenney, and Jonathan Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. USENIX ATC 2011.

[53] John D Valois. Lock-free linked lists using compare-and-swap. PODC 1995.