Memory Tagging: Minimalist Synchronization for Scalable Concurrent Data Structures

Dan Alistarh
dan.alistarh@ist.ac.at
IST Austria

Trevor Brown
me@tbrown.pro
University of Waterloo

Nandini Singhal*
nandini12396@gmail.com
Microsoft (R&D) India

ABSTRACT

There has been a significant amount of research on hardware and software support for efficient concurrent data structures; yet, the question of how to build correct, simple, and scalable data structures has not yet been definitively settled. In this paper, we revisit this question from a minimalist perspective, and ask: what is the smallest amount of synchronization required for correct and efficient concurrent search data structures, and how could this minimal synchronization support be provided in hardware?

To address these questions, we introduce *memory tagging*, a simple hardware mechanism which enables the programmer to "tag" a dynamic set of memory locations, at cache-line granularity, and later validate whether the memory has been concurrently modified, with the possibility of updating one of the underlying locations atomically if validation succeeds. We provide several examples showing that this mechanism can enable fast and arguably simple concurrent data structure designs, such as lists, binary search trees, balanced search trees, range queries, and Software Transactional Memory (STM) implementations. We provide an implementation of memory tags in the Graphite multi-core simulator, showing that the mechanism can be implemented entirely at the level of L1 cache, and that it can enable non-trivial speedups versus existing implementations of the above data structures.

ACM Reference Format:
Dan Alistarh, Trevor Brown, and Nandini Singhal. 2020. **Memory Tagging: Minimalist Synchronization for Scalable Concurrent Data Structures**. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '20), July 15–17, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3350755.3400213

1 INTRODUCTION

The advent of multi-core processors has spurred a significant amount of work on fast concurrent data structures, as well as hardware and software synchronization mechanisms to support the efficient implementation of such designs. At the hardware level, the most notable recent development has been hardware transactional memory (HTM) [1, 2], accompanied by a supporting cast of software or hybrid techniques aimed at helping programmers take advantage of its semantics [3, 4], as well as a myriad of specialized software/hardware techniques for scalable synchronization, e.g. [5–10]. Efficient concurrent variants are now known for many classic data structures, such as lists, e.g. [11], [12], [13], hash tables, e.g. [12], [14], [6], skip lists, e.g. [15], [16], [17], [18], search trees, e.g. [19], [20], queues [21], stacks [22], [23], or priority queues, e.g. [24], [25], [26]. At a very high level, the field makes progress by eliminating *every last bit of unnecessary synchronization at the logical level* when implementing a given semantics, as well as mapping the remaining synchronization requirements *as efficiently as possible* onto the existing hardware synchronization mechanisms.

Example: The Concurrent Linked List. To illustrate, consider the classical problem of building a correct concurrent singly-linked list [11–14]. Several lock-based and non-blocking techniques have been introduced to address this problem by progressively reducing the amount of synchronization required by the concurrent implementation, at the cost of increasing complexity, both in the algorithmic design, and especially in the correctness arguments.

Concretely, in list implementations, a thread P positioned at node `crt` needs to be able to validate that neither node `crt` nor its predecessor `pred` have been changed concurrently since P's last read. Intuitively, this is needed since, for correctness, it is important to enforce the invariants that 1) threads cannot modify a concurrently deleted node, and 2) should not be able follow the `next` pointer of a deleted node.

For instance, *hand-over-hand locking* [27] ensures these invariants by making sure that thread P holds locks on both the current node and on its predecessor, and seeks to minimize the amount of synchronization by releasing locks as the thread progresses through the list. At the same time, a deleting thread has to hold a lock on the deleted node and on its predecessor. Unfortunately, locking operations cause significant synchronization overheads and loss of parallelism, since *readers now have to write*, as traversals need to take locks. A more efficient technique is using *marking / mark bits* [14], logically associated with nodes, and modified by threads to signal a change in the list's logical structure, such as the removal of a node. Mark bits can be co-located with the nodes' next pointers, leading to the Harris-Michael *pointer marking* design [11, 12], which leads to efficiency that is close to state-of-the-art [6].

Examining the marking technique from a "minimalist" perspective, one notices that even the limited synchronization required by mark bits is sub-optimal. Consider the interaction between the hardware core upon which a traversing thread P is running and the cache coherence mechanism: when first reading `crt`, the thread/-core needs to bring the cache-line corresponding to the node locally in Shared (S) state. If the node is marked concurrently for deletion, the node's cache-line state must have necessarily moved to Invalid (I), since its value was changed. To check for marking, thread P

*Part of this work was performed while the author was an intern at IST Austria.

SPAA '20, July 15–17, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-6935-0/20/07…$15.00
https://doi.org/10.1145/3350755.3400213

must read the cache-line again, causing it to be elevated to shared (S) or exclusive (E) state, possibly invalidating other threads' caches, and introducing extra delay. This transition is done just to realize that marking occurred, which will likely cause this operation to restart. Ideally, one would like to avoid this extra synchronization, since it adds latency and wastes memory bandwidth.

A Thought Experiment: Memory Tagging. Leaving practical considerations aside, consider the above example, but where the thread P is allowed to check, before inspecting whether node `crt` has been marked, whether the corresponding cache-line has become Invalid (I) since its previous access, upon which the line had been in Shared (S) state. If this is the case, thread P could reasonably assume that its read validation will fail, and avoid the wasted coherence traffic due to bringing in the corresponding cache line.

The mechanism we introduce, called *memory tagging* (in short, *MemTags*), allows threads to do exactly this: a thread can *tag* a set of memory locations before accessing them, read them, and can then *validate* any of these tags at any later point in time, or *untag* any tagged location so it is no longer tracked. Validation should *fail* if the memory location has been invalidated by a concurrent update since the tag was set. If validation *succeeds*, then the thread can be certain that its read value is still current.

In addition, we provide two operations which build on the validation semantics. The first is the natural counterpart to *compare-and-swap (CAS)*, called *validate-and-swap (VAS)*, which allows the thread to swap in a new value in a location conditional on its tag validations succeeding. VAS is more expressive than CAS, but has similar cost, and is easier to implement than HTM, since we update a single word. The second operation is *Invalidate-and-Swap (IAS)*, which atomically invalidates all tagged cache lines, and, if successful, changes the value at a single location. While the reason for introducing VAS is intuitively straightforward, the necessity of the IAS operation is more subtle—in Section 4 we will argue that IAS is in fact necessary to correctly and efficiently implement data structures in the absence of marking.

It is natural to ask what are the performance and expressivity benefits of this mechanism. As a simple example, for linked lists, tagging enables *two* efficient variants. The first, called *VAS-based marking*, uses tags to alleviate the synchronization issue discussed above: starting from a Harris-Michael list with pointer marking, we tag the predecessor `pred` and successor `succ` for the key to be inserted or deleted. If they are logically unmarked, we use `VAS` on these two locations to perform the pointer swing corresponding to the operation. Correctness follows since `VAS` will only succeed if the nodes have not been updated since last being read. As discussed, tagging improves performance if multiple threads attempt to update the same node, since `VAS` ensures that validations fail locally, at the core, instead of causing additional coherence traffic.

The second, more interesting variant is *hand-over-hand (HoH) tagging*: threads traverse the list in a hand-over-hand fashion, maintaining tags on the predecessor and on the current node, and untagging earlier nodes when they are no longer needed. In contrast to hand-over-hand *locking*, readers *do not write*, so we can recover the simplicity of this technique without the expensive lock acquisitions. Of note, for this design to be correct, the pointer update corresponding to `insert` or `delete` operations must be performed via `invalidate-and-swap (IAS)`. This variant does not require pointer marking for correctness, and allows traversals to overtake eachother.

The benefit of tagging is that checking whether tags are valid is *local*, in the sense that it can be done entirely at the level of the core's L1 cache, without additional coherence traffic. (We provide an implementation proposal at the level of L1 cache in the later sections.) This is in contrast to methods such as marking or locking, which are not local, and can incur significant overheads.

General Tagging. While the above uses of tagging are straightforward, MemTags can bring about non-trivial simplifications and performance improvements for more complex data structures. This is the case for binary search trees, skip-lists, *balanced* search trees (chromatic trees and (a,b)-trees), as well as NOrec STM [28], k-word CAS (kCAS) [29] and range query implementations. At a conceptual level, tagging challenges the data structure designer to ask what is the *minimal* amount of state to be validated to ensure data structure correctness. In particular, we highlight (1) the broad applicability of hand-over-hand (HoH) tagging for data structure traversals, as it offers the simplicity of hand-over-hand locking without the performance cost, and (2) the performance advantages of `VAS` over `CAS` as its fail-fast behavior leads to markedly lower cache coherence overheads, and (3) the fact that HoH tagging can enable a simple and efficient fast-path mechanism for complex data structures such as *balanced* trees, which tend to be notoriously complex, e.g. [30, 31].

The usual trade-off in this area is between the complexity of the reads–which should avoid coherence traffic–and the complexity of the updates–which should perform a minimal number of pointer changes. Primitives such as kCAS [29] or LLX/SCX [8] allow the programmer to explore this complex trade-off, but unfortunately tend to have extremely complex implementations, and still require non-trivial synchronization.

An immediate use of MemTags would be tagging the entire path from the root to the leaf. Upon further thought, we notice that if we know an upper bound D on the number of consecutive nodes on a path which might be deleted by any single operation, then we can always perform searches via *HoH tagging* with a "window" of $D + 1$ nodes. Clearly, upon successful validation, the current node has *not* been deleted concurrently, since its entire "neighborhood" must be unmodified. Furthermore, we show that this idea can be extended in balanced search trees to perform node modifications via *a single* `IAS`, which the updater uses to invalidate all concurrent traversals. The `IAS` invalidates the corresponding locations at other cores (if they are tagged), causing any future validations to fail. The resulting technique is new, and achieves three desirable properties: (1) it implements an atomic node modification without synchronizing on the entire path from the root, (2) without updating more than one location, and (3) with minimal cache coherence overhead.

Another natural use of MemTags is to obtain cheap lock-free *snapshots*: a thread can tag the set of locations, and then validate. If validation succeeds, the snapshot is valid. This simple idea can be used to perform efficient range queries, and can be extended to speed up kCAS implementations [29]. We can further extend this idea to efficiently implement the single global sequence lock in NOrec STM [28]: active reader transactions tag the lock, and abort as soon as the lock becomes invalid. Thanks to the semantics of

tagging, readers can detect conflicts cheaply and validate read set significantly faster, removing a key bottleneck in NOrec.

Implementation, Limitations and Experimental Validation. We present a hardware implementation proposal for MemTags exclusively at the level of the L1 cache, without changes to the underlying coherence protocol. We discuss the feasibility of this proposal, as well as its shortcomings. One main limitation is that on current systems MemTags are bound to be *advisory*: validation can fail spuriously, which implies that MemTags will always have to be backed by a "fallback path" in order to ensure progress. This limitation is shared with most HTM implementations. A second limitation is *complexity*, as correct application of tagging forces the programmer to carefully understand the synchronization required by the implementation. Third, since it would require hardware modifications, MemTags would be relatively complex to implement.

To address the first issue, we first show that MemTags support natural fallback paths similar to those used by hardware lock elision (HLE) [4] and that, marking-based designs and LLX/SCX-based designs provide correct fall-back paths for MemTag-based data structures. Second, we show that spurious invalidations are negligible in practice for reasonable data structure sizes. In particular, we show MemTags can serve as a natural and efficient fast-path for marking and LLX/SCX-based implementations.

We implement MemTags in the Graphite simulator [32] for a tiled multi-core processor with variable number of cores, and test it on standard workloads for a range of data structure designs, such as linked lists, search trees, as well as NOrec STM [28] on the STAMP benchmark [33]. Results suggest that MemTags can speed up the fast path of these implementations. For instance, on standard search-insert-remove workloads, MemTags can speed up highly optimized linked list by 10 to 50%, balanced search trees by up to 2× versus the optimized implementations based on LLX/SCX [8], and NORec by up to 50% versus the pure software implementation.

2 RELATED WORK

Due to the breadth of the area, we will focus on closely related mechanisms, namely OPTIK [7], LLX/SCX [8], Lease/Release [10], and HTM with early release [9].

OPTIK. OPTIK [7] is a design pattern relying on *version numbers* for detecting conflicting concurrent operations. Version numbers are *integrated* into a *versioned* lock, which allows operations to read a lock's version number, try to acquire a *particular version* of the lock, and release the lock. The lock's version number is incremented at the end of a successful critical section that modifies the shared state protected by the lock. OPTIK improves performance in scenarios where threads often perform *optimistic* reads (outside of a critical section), then subsequently acquire a lock, and then validate, only to discover that the validation fails and the lock must be released. Rather than acquiring the lock only to release it, an attempt is made to acquire the lock with a particular version, but the lock acquisition will fail if the data has changed.

Suppose an algorithm uses OPTIK to lock a *sequence* of nodes. For this, it might read the version of each lock, then speculate by reading the contents of nodes, and finally acquire each of the locks protecting these nodes. Suppose the last node in this sequence is changed before the algorithm has acquired any of its locks. Then, the algorithm would *acquire* all but its last lock before failing, even though it was doomed to fail before it even began locking. Such scenarios negate much of the benefit of OPTIK. By contrast, consider a MemTags implementation of this pattern, using `VAS` to acquire each lock. One could *tag* all of the locks and check that they are not held, perform the desired speculative transaction, and then execute a sequence of VAS instructions to acquire the locks. In the previous example, locking would fail *immediately*, before *any* changes are made to shared memory, and failed locking would generate no coherence traffic.

As a software mechanism, OPTIK is much cheaper to implement than MemTags. Yet, the cost of failure with OPTIK is much higher than that of MemTags. Further, unlike MemTags, OPTIK cannot be applied to relatively simple data structures such as skip-lists. One can perhaps think of tagging as generalized hardware support for OPTIK locks, although MemTags provide additional benefits.

LLX/SCX. Reference [8] introduces three new software operations, load-link-extended (LLX), validate-extended (VLX) and store-conditional-extended (SCX), which can implement complex synchronization patterns. LLX, SCX and VLX operate on Data-records. Any number of types of Data-records can be defined, each type containing a fixed number of mutable fields (which can be updated), and a fixed number of immutable fields (which cannot). A successful LLX returns a snapshot of the mutable fields of one Data-record. (The immutable fields can be read directly, since they never change.) An SCX operation by a thread P *depends on* a set D of Data-records, and is used to atomically store a value in one mutable field of one Data-record in D and *finalize* a subset of Data-records in D, meaning that those Data-records cannot undergo any further changes. *Note that finalizing generalizes marking.* SCX succeeds only if each Data-record it depends on has not changed since P last performed an LLX on it. A successful VLX ensures that each of the Data-records has not changed since the caller last performed an LLX on it.

The reader will have noticed that LLX is analogous to tagging, VLX to validate and SCX to VAS. However, with tagging, we are able to *avoid* some of the modifications to data structures that are needed to use the LLX/SCX primitives, such as the addition of a marked bit and an extra pointer field to each data structure node, reducing space overhead. Additionally, given an implementation of a data structure from LLX/SCX, we can produce an improved implementation using VAS, with significantly better performance. This is evident from the pseudocode and experimental results.

Lease/Release. Lease/Release is a hardware mechanism aimed at improving the performance of contended data structures [10]. It allows a thread to *lease* a cache line for a bounded period of time, during which concurrent accesses will not be able to invalidate it. This can significantly improve the performance of data structures such as stacks, queues, or even locks. While tags could be used in conjunction with leases, they are aimed at a completely different application—*search* data structures versus producer/consumer data structures—and would not benefit search data structures.

Recently, [34] introduced an HTM-based mechanism to reduce the overheads of contention in the context of implementing concurrent queues. The goal of this mechanism is different from tagging, since we aim to speed up read operations.

Transactional Memory. It is also interesting to contrast tagging with HTM [1] implementations such as TSX [2], which assume by default that all the code between the beginning and end of a transaction should be executed speculatively. This is known to lead to spurious conflicts and loss of performance [35]. Techniques such as *teleportation* [36] aim to reduce the capacity and coherence overheads of this assumption by allowing the programmer to split transactions into smaller basic blocks which are less likely to conflict.

Early release [9] is a mechanism allowing to select memory locations which should no longer be monitored by HTM as part of the transaction *before* the transaction ends. As such, early release can be seen as a *weaker version of tagging*: if early release needs the programmer to explicitly release (untag) all the locations swept up by the HTM implementation, MemTags allows explicit and dynamic tagging, untagging, and `VAS/IAS`. On the other hand, early release still supports full transactional semantics. In [9], the authors provide examples and empirical results on data structures such as linked lists, AVL trees, and B-trees, suggesting that early release falls short in terms of the trade-off between programming complexity and performance benefits. However, the examples given do not significantly innovate upon the underlying data structures. Here, we show that such fine-grained synchronization mechanisms can indeed enable simpler and more efficient advanced data structures, such as balanced search trees. In particular, we note that our `IAS` semantics cannot be supported through early release–at the same time, our example in Section 5.1 suggests that invalidation may be required for significant performance savings. Compared with early release, MemTags require specific tag addition, removal, and validation. This adds more flexibility, but also requires an in-depth understanding of the underlying data structure. This fits the scope of our application scenario, which includes advanced performance optimization for concurrent data structures.

3 SEMANTICS AND IMPLEMENTATION

Semantics. The `AddTag (& node, size)` method takes as argument a memory location that needs to be tagged and a range, derives the cache-line(s) corresponding to this memory location, and stores them in a set data structure, in *tagged* state. The `RemoveTag (& node)` operation takes an address as argument, and untags it. It does so by removing the cache lines corresponding to the memory locations from tagged state. At any point after the first `AddTag`, if (1) a cache line that is part of tagged set gets evicted due to cache replacement policy, or (2) a cache line that is part of the tagged set gets an invalidation/flush request (due to other threads wanting to write to this cache line), then those cache lines are added to the *evicted* set. The `Validate ()` method returns `False` if any of the tagged cache lines have been moved to the *evicted* set since they were tagged, and `True` otherwise. The `Validate-and-Swap (VAS)` method takes as argument a target location, and its desired updated value; it atomically validates that none of the tagged addresses have been evicted; if this is the case, it performs the update on this location atomically. The operation returns `True` if successful, and `False` otherwise. `VAS` will fail due to a failure of tag set validation (due to an eviction). Notice that this automatically incorporates CAS semantics, since we expect the target location to have been read after being tagged—thus, this location cannot be concurrently updated without invalidating the corresponding line. (Alternatively, we can augment the VAS semantics to be equivalent to CAS by requiring an expected value as part of the argument and checking against it.) The `Invalidate-and-Swap (IAS)` method atomically 1) checks that none of the currently tagged addresses have been evicted; 2) invalidates them at other cores; 3) if both these steps are successful, the operation then performs an update of one location. The operation returns `True` if successful, and `False` otherwise. The `ClearTagSet ()` operation empties the set of tagged locations.

Hardware Implementation. We assume basic knowledge of MESI cache coherence; please see [37] for a primer. We propose to implement MemTags at the level of the L1 cache by adding extra state to each core's *load buffer* structure, which serves as a directory keeping track of the current state of all cache lines, and handles cache misses. In particular, we add two possible states for every cache line, which are *transition-to-tagged*, and *tagged*. The first state means that the line will be tagged as soon as the corresponding cache miss gets served. When this request gets served, the line moves to *tagged* state. If a tagged line gets invalidated, then it moves to *evicted* state. The mechanism needs to maintain metadata for such lines until the following validation request, at which point this metadata is reset. Upon validation, the system checks the metadata for all states, while temporarily pausing the serving of new coherence requests, to maintain atomicity. If none of the tagged lines have been evicted, then the validation succeeds. In either case, the contents of the tagged and evicted buffers are reset to empty. Tag removal and `VAS` are implemented similarly. The invalidation step in `IAS` can be implemented by piggy-backing on the cache coherence mechanism, that is, by following the pattern required for elevating the state of the corresponding cache lines to E/M. The number of tags which can be held concurrently is upper bounded by a system-wide constant `Max_Tags`, after which the system will stop accepting tagging requests.

Tags are compatible with out-of-order execution, in the sense that a tag operation being "hoisted" ahead of a preceding store will not affect correctness of the data structure. However, we do enforce that `validation` operations are not re-ordered with respect to `tag` operations. In addition, the system may recover from branch misspeculation by evicting all existing tags, which ensures that any subsequent validation will fail. Similarly, tag set overflow can be handled gracefully by simply returning False to validation requests once the size of the validation set has exceeded `Max_Tags`. Since tags are advisory, these behaviors are permissible.

We note that this mechanism can be extended to MOESI/MESIF-style cache coherent implementations. Due to space constraints, we leave the description to the full version of our paper.

Fall-Back Path. One key issue is that hardware MemTags will not be able to guarantee progress, since cache lines could in theory be evicted for many reasons. At the same time, there is no immediate way to simulate MemTags in *software*.

To provide a fall-back path for the case when tags would fail repeatedly in a spurious manner, we use a mechanism similar to Hardware Lock Elision (HLE) [4]. We allocate a separate `Mode` line, which takes values `SLOW` and `FAST`, depending on whether the system is on the fall-back or fast path, respectively. This line is modified

only on a mode change. All operations begin by first reading this line, to determine which code path will be executed, and all tag validations or VAS operations on the fast path will include this line as part of their tag set. An operation which experiences a large number of consecutive validations (higher than some set threshold) will change the Mode to SLOW and proceed to execute a software-only variant of the data structure, for instance one based on kCAS or LLX/SCX. This causes all fast-path MemTags operations to fail their validation and have no effects. The Mode can be re-set to FAST after some pre-defined period.

We do not necessarily lose simplicity or generality with this fallback path. For all the data structures we discuss in the later sections, it is natural to pair the tagged fast path with either a fallback path based on hand-over-hand locking, or one based on lock-free algorithms with node marking. In terms of complexity, this should be close to free. In terms of performance, this is reasonable as long as the fallback path is infrequent.

4 EXAMPLE: TAGGED LINKED LIST

We now consider different ways in which tagging can be used to build efficient data structures. Our example will revolve around the simple linked-list data structure, but our goal will be to illustrate the usefulness and pitfalls of using this minimalist technique. The focus here is less on providing full detail (which we will do for more complex applications in the later sections), but on isolating more general patterns.

We will start from the classic Harris-Michael Linked List [11, 12, 14], which implements an *ordered set* with operations: `insert`, `delete` and `search`. Recall that this design assumes a singly-linked list with pointers to head and tail nodes. Each node has a boolean `marked` field indicating whether that node is in the set. The algorithm maintains the invariant that every unmarked node is reachable; if a node is not found or is marked, then it is logically not in the set. The algorithm makes heavy use of a helper function called `locate`, which takes a target key k as its argument, and returns nodes *pred* and *curr* such that $pred.key \leq k < curr.key$. This subroutine also removes marked nodes whenever it encounters them by unlinking them.

VAS-Based Linked-List. The first way to apply tagging is straightforward: 1) we can complement value-based validation with tag validation, and 2) we can replace CAS with VAS. Pseudocode for this variant can be found in Algorithm 1. For instance, in this case, an `insert` of key k works as follows. It first invokes `locate` to locate *pred* and *curr* for the key. If $pred.key \neq k$, then the cache lines containing *pred* and *curr* are *tagged*. Next, we check that these nodes are *unmarked* and the next node of *pred* is still *curr*—intuitively, we are checking that both are *reachable* from the head of the list. After the mark check, we perform validation on the cache lines containing *pred* and *curr*. If the validation fails, then the tagged set is cleared and the operation is restarted. The pointer change needed to finish the insertion is then implemented via VAS, which atomically: (1) validates that the tagged memory locations (*pred* and *curr*) have not been modified since they were tagged and (2) performs the swap. The `delete` operation is similar.

The benefits of tagging in this case are limited. This design is very close to the lock-free, but can avoid some cache coherence traffic due to helping when threads attempt to perform concurrent swaps on the same location if they are performed via VAS. In this case, threads which fail will not pay the additional cost of value-based validation, since their validation fails locally. We note that a natural backup path for our implementation is the baseline lock-free implementation itself.

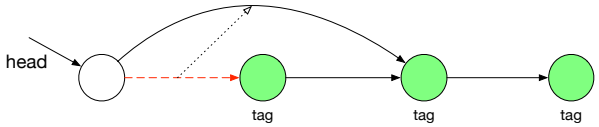


Figure 1: Counterexample sketch for naive HoH-tagging list algorithm without invalidation. Thread 1 (whose tags are in green) traverses the list, tagging each newly visited node, and untagging its grandparent at every step in a HoH manner. Thread 2 (red) deletes the first green node concurrently, by swinging the corresponding pointers in its predecessor via VAS. If the pointer swing does not invalidate the node which is pointed to, then there is no way for Thread 1 to tell that it is accessing a deleted part of the list, for instance inserting elements there. This counterexample would be circumvented if Thread 2 would *invalidate* the nodes which it is deleting, which would cause Thread 1 to fail its validation and restart its operation.

The Hand-over-Hand (HoH) Tagging Puzzle. It is natural to ask whether we could leverage tagging to obtain a lock-free implementation which completely avoids the use of mark bits. One tempting approach is to follow the classic hand-over-hand locking design [14], replacing *locking* with *tagging*. Such a design could work as follows. When a thread P traverses the list, it tags each node when first visiting it. Tagging can be done in a "hand-over-hand" fashion, to ensure progress and correctness while simultaneously minimizing the number of locations which need to be validated: whenever a new node is first visited by a list traversal, it is tagged, and validated to ensure that it has not been modified since the last read. The idea would be to ensure that the node cannot be concurrently modified without thread P failing its validation: intuitively, maintaining tags on consecutive nodes and being able to validate them achieves that neither of the two nodes has been modified. Please see the `locate` method in Algorithm 2 for the precise implementation of this procedure. It is therefore reasonable to ask whether this intuition can be turned into a correct hand-over-hand tagging algorithm if we perform the pointer swinging for inserts and deletes via VAS, as in the previous implementation.

As the attentive reader guessed, VAS and tag validation are not sufficient to ensures correctness for this hand-over-hand design. One counterexample is depicted and described in Figure 1. At a high level, the key shortcoming of this approach is that the thread P holding tags on a sequence of nodes cannot tell whether the "first" node in this sequence (in the traversal order) is being removed by some concurrent operation, since there is no mechanism to signal this deletion at the node level: the corresponding pointer change happens at the *predecessor* of that node. Circumventing this counterexample is exactly one of the benefits of *node marking*.

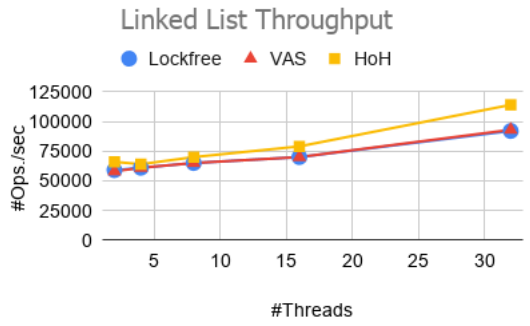


Figure 2: Throughput versus number of threads for the lock-free Harris linked list, compared with our VAS-based and HoH-based implementations with 35% inserts, 35% deletes workload. As expected, the HoH variant is significantly more efficient, since it reduces synchronization cost.

Unfortunately, if we were to directly implement marking, we would end up with the bland VAS-based design.

Efficient Transient Marking via Invalidation. We can resolve this conundrum by noting that it is sufficient for any concurrent delete to *invalidate* the node it is deleting, making sure that any concurrent traversal which has tagged this now-deleted node fails its node validation, and therefore restarts. This mechanism is similar to what would occur if the node were marked, or in a correct implementation of hand-over-hand locking, where the deleting node has to hold the lock on the node it is deleting.

The resulting pseudocode is presented in Algorithm 2, and its performance relative to the lock-free and VAS baselines is presented in Figure 2. The design has a few notable properties: (1) traversals can overtake each other; (2) a delete operation does not have to write to the node it is deleting. (The operation does invalidate the node. In the unlikely case where all the other threads are tagging the cache line corresponding to that node, then the cost of writing and the cost of invalidation are similar—however, this operation can be significantly cheaper in the simple case where no other thread has the node in L1 cache.) The performance benefit of these properties is visible in Figure 2.

Correctness. The correctness of the HoH design rests on the following invariant: at the time a node is successfully validated, it is guaranteed to be present in the data structure, in the sense that it can be located via a traversal starting at the head node. This invariant is preserved inductively. It holds trivially as long as the head pointer is being validated. Then, every time a thread successfully validates all its tagged nodes, after adding a new node to its traversal, it ensures that (1) the previously validated nodes are still in the list, and (2) the newly traversed node is pointed to by a node that is in the list. Together, this guarantees the extension of the invariant to the newly added node, and allows the thread to untag the "oldest" traversed node (technique described further in Sec. 5.1).

Lessons Learned. The key property we are leveraging in this implementation is that *node marking only needs to be transient:* it is sufficient for the deleting operation to "abort" concurrent traversals

Algorithm 1 VAS-based Linked List

```
1:  type node { key k, node* next }
2:  class list { node* head }
3:  procedure HelpIfNeeded(node* pred, node* curr)
4:      if curr is not marked then return false          ▷ curr does not need unlinking-help
5:      AddTag(pred, sizeof (node))
6:      if pred is marked then ClearTagSet() and return true   ▷ restart LOCATE from scratch
7:      if pred does not point to curr then ClearTagSet() and return true   ▷ restart
8:      AddTag(curr, sizeof(node))                       ▷ if we get here, curr is marked
9:      succ ← (curr → next)    ▷ marked nodes do not change (so succ is same for all helpers)
10:     Validate-and-swap (&(pred → next), succ)         ▷ help unlinking step
11:     ClearTagSet() and return true                    ▷ restart
12: end procedure
13: procedure LOCATE(key k)
14:     curr ← head          ▷ Note: head (sentinel node with key −∞) cannot be marked
15:     do
16:         pred ← curr
17:         curr ← (curr → next)
18:         if HelpIfNeeded(pred, curr) then start again
19:     while curr → k < k                   ▷ continue search until curr→k is too large
20:     return ⟨ pred, curr ⟩
21: end procedure
22: procedure INSERT (key k)
23:     ⟨ pred, curr ⟩ ← LOCATE(k)           ▷ note LOCATE does not perform tagging
24:     if curr→ key = k then return false
25:     end if
26:     AddTag(pred, sizeof(node))
27:     AddTag(curr, sizeof(node))
28:     if pred or curr is marked, or pred does not point to curr then
29:         ClearTagSet() and start again
30:     end if
31:     node ← new Node(k, curr)             ▷ node to be inserted between pred and curr
32:     if ¬Validate-and-swap(&(pred → next), node) then
33:         ClearTagSet() and start again                ▷ failed operation
34:     end if
35:     ClearTagSet() and return true
36: end procedure
37: procedure DELETE(key k)
38:     ⟨ pred, curr ⟩ ← LOCATE(k)           ▷ note LOCATE does not perform tagging
39:     if curr→ key ≠ k then return false
40:     AddTag(pred, sizeof(node))
41:     AddTag(curr, sizeof(node))
42:     succ ← (curr→next)
43:     if pred or curr is marked, or pred does not point to curr then
44:         ClearTagSet() and start again
45:     end if
46:     if ¬Validate-and-swap(&(curr → next), marked succ) then   ▷ marking step
47:         ClearTagSet() and start again
48:     end if
49:     Validate-and-swap(&(pred → next), succ) and CearTagSet()   ▷ unlinking step
50: end procedure
51: procedure SEARCH(key k)
52:     ⟨ pred, curr ⟩ ← LOCATE(k)           ▷ note LOCATE does not perform tagging
53:     return (curr→key = k)          ▷ return the Boolean result of the comparison
54: end procedure
```

on the node it is deleting (since traversals that start *after* a node is unlinked cannot reach it). Therefore, the correct instantiation of the hand-over-hand tagging pattern will have to perform IAS, invalidating the node it is deleting and its predecessor, and swapping the corresponding pointer accordingly. At the same time, we note that this pattern can lead to significant performance benefits, since it minimizes synchronization costs. We build on this observation in the next section, where we consider more complex applications of this pattern.

Intuitively, this pattern provides linearization points as follows: a successful modification is linearized at the VAS/IAS that performs it. A read-only operation is linearized at the last successful validate. In the case of hand-over-hand tagging, we must also prove that if we perform a successful validate/VAS/IAS, then the nodes in the tag set are reachable from the root/head.

Algorithm 2 Linked List using HoH tagging

```
 1: type node { key k, node* next }                          ▷ marking is not used
 2: class list { node* head }
 3: procedure LOCATE(key k)
 4:     pred ← head
 5:     AddTag(pred, sizeof(node))
 6:     curr ← (pred → next)
 7:     AddTag(curr, sizeof(node))          ▷ validate checks whether pred, curr are in the list
 8:     do
 9:         if ¬Validate() then ClearTagSet() and start again
10:         RemoveTag(pred, sizeof(node))
11:         succ ← (curr→next)
12:         AddTag(succ, sizeof(node))   ▷ validate checks whether pred, curr, succ are in the list
13:         pred ← curr
14:         curr ← succ
15:     while succ → k < k                        ▷ continue search until succ → k is too large
16:     return ⟨ pred, curr ⟩                      ▷ Note: pred and curr remain tagged
17: end procedure
18: procedure INSERT(key k)
19:     ⟨ pred, curr ⟩ ← LOCATE(k)                              ▷ performs hand-over-hand tagging
20:     if curr →k = k then ClearTagSet() and return false
21:     node ← new Node(k, curr)                  ▷ node to be inserted between pred and curr
22:     if ¬Validate-and-swap(&(pred → next), node) then
23:         ClearTagSet() and start again                                   ▷ failed operation
24:     end if
25:     ClearTagSet() and return true
26: end procedure
27: procedure DELETE(value v)
28:     ⟨ pred, curr ⟩ ← LOCATE(k)                              ▷ performs hand-over-hand tagging
29:     if curr → k ≠ k then ClearTagSet() and return false
30:     if ¬Invalidate-and-swap(&(pred → next), curr → next) then
31:         ClearTagSet() and start again
32:     end if
33:     ClearTagSet() and return true
34: end procedure
35: procedure SEARCH (key k)
36:     ⟨ pred, curr ⟩ ← LOCATE(k)                              ▷ performs hand-over-hand tagging
37:     ClearTagSet()      ▷ tagging inside LOCATE established a time when curr was in the list
38:     return (curr→key = k)                  ▷ return the Boolean result of the comparison
39: end procedure
```

5 APPLICATIONS OF TAGGING

5.1 Tagged (a,b)-Tree

Overview. We now describe a more complex application of MemTags to implement a balanced tree data structure, notably an (a,b)-tree, which is a generalization of a B+tree. We build upon the LLX/SCX based implementation of Brown et al. [8]. In fact, this section will outline a general way of using MemTags as a fast-path for data structures designed using LLX/SCX, for which we present one specific instance.

Like B+trees, (a,b)-trees are leaf-oriented, which means all key-value pairs in the dictionary are contained in the *leaves* of the tree. Excluding the root, each leaf in an (a,b)-tree has between a and b keys, and each internal node has between a and b child pointers, where a and b can be any values chosen such that $b \geq 2a - 1$. Note that a B+tree is simply an (a, 2a-1)-tree.

As in B+trees, the balance invariant in (a,b)-trees is quite strict: all leaves in an (a,b)-tree have the same level (where the level of a leaf is the number of ancestors it has). Maintaining this strict balance condition can make rebalancing after insertions and deletions quite expensive, both in terms of the number of modifications performed on the tree (in a sequential setting), and in terms of synchronization (in a concurrent setting). To overcome the high cost of rebalancing (a,b)-trees in a concurrent setting, the (a,b)-tree properties can be violated *temporarily* while updates are in progress.

More specifically, a node can transiently contain fewer than a keys/pointers, in which case we say a *degree violation* occurs at that node (unless that node is the root). Additionally, leaves can transiently occur at different levels. To keep track of level differences (so once can rebalance to remove such differences), each node is augmented with a *flag* bit. If the flag bit of a node is set, then that node is *not counted* towards the level of any leaves below it. We say that a *flag violation* occurs at each node whose flag bit is set. We then define the *relaxed level* of a node to be its level minus the number of flag violations at its ancestors. *Rebalancing steps* are performed to remove degree and flag violations while maintaining the invariant: all leaves have the same *relaxed level*. Once there are no violations, the tree is balanced.

Chapter 8 of [31] provides a concurrent implementation using the LLX/SCX primitives. We first give a brief overview of how this implementation works, and then explain how one can obtain a faster implementation by using hand-over-hand tagging.

Using LLX and SCX. Searches in this implementation are performed exactly as in a *sequential* (a,b)-tree. At each node, a thread compares the key k it is searching for with the various keys in the node, and determines which child pointer it should follow, then follows that pointer. This continues until it reaches a leaf, and determines whether k is in the tree.

To insert a key-value pair (k, v), a thread first performs a search for k, terminating at a leaf u with parent p. If u already contains k, then the insertion terminates and returns false.[1] So, suppose k is not in u. If u contains fewer than b keys, then the operation attempts to perform the change illustrated in Figure 3(a). This entails performing LLXs on p and u (which return snapshots of the contents of these nodes), creating a *new copy* n *of* u containing all of the pairs in u as well as (k, v), and finally using SCX to replace u with n. (Note that u does not have to be finalized when it is replaced, because leaves are never modified.) Note that this SCX will succeed only if neither p nor u have changed since we performed LLX on them.

On the other hand, if u already contains b keys, a single node cannot accommodate the pairs in u as well as (k, v), so the operation attempts to perform the change illustrated in Figure 3(b). This entails performing LLXs on p and u, creating two new leaves n_L and n_R that evenly share the pairs in u as well as (k, v), creating a new *flagged parent* n for n_L and n_R, and finally using SCX to replace u with n and finalize u. Note that n is flagged to preserve the invariant that all leaves have the same relaxed level.

Delete is similar to insert: it replaces u with a new copy that does not contain the key being deleted.

Whenever a thread performing an insert or delete of key k creates a violation, it executes a *cleanup* procedure that repeatedly searches towards key k, looking for violations and performing any applicable rebalancing steps, terminating when it no longer finds any violations. Because each rebalancing step either eliminates a violation or moves a violation upwards along the path to the root (and no other movement of violations is possible), the violation created by the thread will be eliminated before this procedure terminates.

Each rebalancing step is implemented using LLX and SCX, similarly to insert and delete. Note, however, that the rebalancing steps are slightly more complex, and some involve using SCX to atomically replace and finalize up to three nodes (operating on up to four nodes atomically, including the ancestor whose pointer is changed to perform the replacement). LLX and SCX are well suited

[1] Here, we consider insert-if-absent functionality. If it is desirable to replace the existing value with the new value v, the data structure can be modified slightly to allow this.

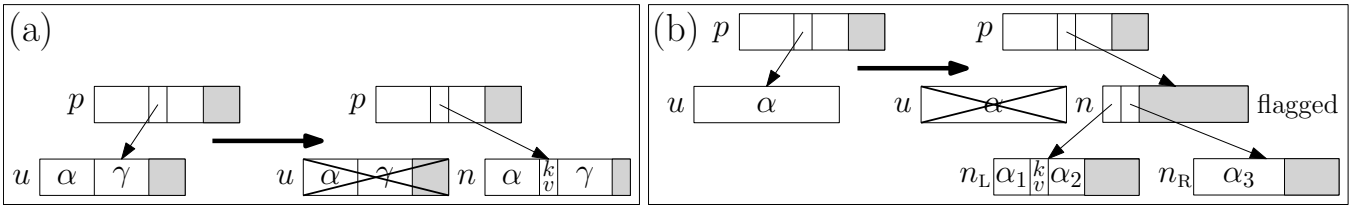


Figure 3: Two cases for Insert(k, v); (a) applies if $\alpha + \gamma < b$ (u is not full); (b) applies if $\alpha = b$ (u is full).

to this task, but their implementations have significant synchronization overhead compared to our new tagging mechanism, including *marking* each node prior to its removal from the tree, and having threads participate in a sort of collaborative operation-locking protocol (in which nodes are locked exclusively for an operation—not a thread—prior to being modified or finalized). Much more efficient implementations are enabled by our tagging mechanism.

The Generic Transformation. We now present a set of guidelines for obtaining a correct and efficient tagged data structure from an existing LLX/SCX variant. First, we notice that the LLX operation can be directly simulated by *tagging* the corresponding objects being read. Note that LLX semantics require the operation to return a snapshot of the corresponding nodes: however, upon close inspection, we observe that this snapshot return value *is never used*.[2] Therefore, it is sufficient to simply replace each LLX call with a tag on the corresponding object, followed by a read on it.

For emulating SCX, we first note that this operation has *two* main uses. The first is similar to k-compare-single-swap (KCSS), in which we wish to validate against a set of locations and swing a pointer. Clearly, this usage can be directly simulated via VAS, especially since we notice that all the locations which were part of the corresponding LLX are tagged by the previous step in the transformation. The second usage of SCX is in *finalizing*, which is implemented via *marking* in [31], and is designed to cause any SCX that depends on a finalized node to fail. Naturally, this usage of SCX should be replaced by IAS on the corresponding locations, implementing transient marking.

Key Difficulty in the Transformation. Thus, the key non-trivial step is in mapping an LLX/SCX implementation using finalizing to a tagged implementation using hand-over-hand tagging (avoiding the need for finalizing by using HoH tagging similarly to our linked list). Crucially, the purpose of finalizing nodes in the (a,b)-tree is to prevent erroneous changes to nodes that have been deleted: By finalizing deleted nodes, and disallowing changes to finalized nodes, it becomes impossible to modify a deleted node.

Similarly, the goal of HoH tagging is to guarantee that validate/-VAS/IAS succeeds ***only if*** *all tagged nodes are* ***currently reachable*** *from the root (i.e., semantically, they are not finalized).*

The exact way to do this, and in particular to *untag* nodes, is data-structure specific, as described below.

(a,b)-Tree using Hand-over-Hand Tagging. We avoid some special cases in the code (as in [31]) by setting the initial *root* pointer to point to a *sentinel* internal node containing a single pointer, which points to an empty leaf node. This sentinel node is never deleted (so the *root* pointer always points to it) nor modified by rebalancing (so it always contains a single pointer). All keys in the set are always found in the subtree rooted at the child of *root*.

Each operation in this implementation tags nodes in hand-over-hand fashion. Let us reason about when it is safe to *untag* nodes.
Observation: By inspection of the (a,b)-tree operations in [8], the longest path of nodes that can be *atomically* removed from the tree has length 2. In other words, an operation may remove a node and its parent, but no operation also removes its grandparent. Thus, for a node to be deleted, a child pointer must change in its parent or grandparent.

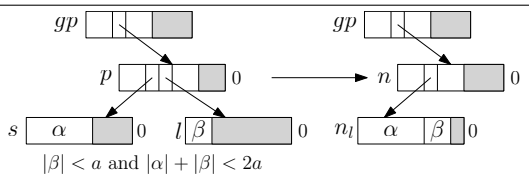
Algorithm 3 (a,b) tree using HoH tagging

```
 1: type node { weight w, array(key) keys, array(node*) ptrs }
 2: class abtree { node* root, constant a, constant b }
 3: procedure LOCATE (key k)
 4:     gparent ← NIL
 5:     parent ← NIL
 6:     curr ← root                          ▷ root pointer never changes, so no need to tag it
 7:     AddTag (curr, sizeof (node))
 8:     while curr is internal node do
 9:         next ← selectChildPointer (k, curr)      ▷ determine which subtree should contain k
10:         AddTag (next, sizeof (node))
11:         if ¬Validate() then ClearTagSet () and start again
12:         if gparent ≠ NIL then RemoveTag (gparent, sizeof (node))
13:         gparent ← parent
14:         parent ← curr
15:         curr ← next
16:     end while
17:     return ⟨ gparent, parent, curr ⟩                    ▷ these remain tagged
18: end procedure
19: procedure INSERT (key k)
20:     ⟨ gparent, parent, curr ⟩ ← LOCATE (k)
21:     if curr contains key then return false
22:     if gparent ⇝ (does not point to) parent or parent ⇝ curr then ClearTagSet () and restart
23:     let parentIndex be the index in gparent → ptrs[] where we saw a pointer to parent
24:     let currIndex be the index in parent → ptrs[] where we saw a pointer to curr
25:     if curr contains fewer than b keys then
26:         newNode ← create new copy of curr with key k added to it
27:         if ¬Invalidate-and-swap (&(parent → ptrs[currIndex], newNode) then
28:             ClearTagSet () and restart
29:         end if
30:     else                                   ▷ curr overflows into two new leaves
31:         let left and right be pointers to new leaves that evenly share keys (curr → keys ∪{k})
32:         let newNode be a pointer to a new internal node that points to left and right
33:         if ¬Invalidate-and-swap (&(gparent → ptrs[parentIndex], newNode) then
34:             ClearTagSet () and restart
35:         end if
36:     end if
37:     ClearTagSet ()
38:     Rebalance (k)
39:     return true
40: end procedure
41: procedure DELETE (key k, value v)
42:     Similar to INSERT, but instead of overflow,
43:     we merge any two nodes whose keys would fit in one node
44:     (by creating a new node and performing IAS)
45: end procedure
```

Pseudocode appears in Algorithms 3, 5, 4. Consider an update operation (insert or delete) that involves a node u, its parent p and

[2] At the same time, if the snapshot return value is required, we can simulate snapshot semantics via tag validation.

Algorithm 4 Example (a,b)-tree rebalancing step




```
1:  procedure AbsorbSibling (gp, pIndex, p, lIndex, l, sIndex, s)
2:      ▷ arguments: grandparent, index of parent in gp→ptrs, parent,
3:      ▷ index of leaf in p → ptrs, leaf, index of sibling in p → ptrs, sibling
4:      AddTag (gp, sizeof (node))
5:      if gp does not point to p then return
6:      AddTag (p, sizeof (node))
7:      if p does not point to l then return
8:      if p does not point to s then return
9:      AddTag (l, sizeof (node))
10:     AddTag (s, sizeof (node))
11:     newLeaf ← create new node containing all keys from l and s
12:     newInternal ← create new node by copying p, removing the child pointer to s (and the
    corresponding key), and changing the pointer to l to point to newLeaf instead
13:     Invalidate-and-swap (&(gp → ptrs[pIndex]), newInternal)
14: end procedure
```

Algorithm 5 Rebalancing the (a,b)-tree

```
1:  procedure Rebalance (key k)
2:      do
3:          ▷ This algorithm is unchanged from the LLX/SCX version
4:          do a sequential BST search towards k, stopping at any node that has a balance violation
5:          if no balance violations were found on the search path to k then return
6:          do the appropriate rebalancing step for the type of violation found, i.e., one of:
7:              RootUntag, RootAbsorb, AbsorbChild, PropagateTag, AbsorbSibling, Distribute
8:      while true
9:  end procedure
```

its grandparent *gp*. Such an operation should fail if any of *u*, *p* or *gp* is deleted. (Recall that in the implementation using LLX and SCX, if any of these nodes are deleted, they will be finalized (marked), which will cause any subsequent SCX on them to fail.) So, whenever we perform a successful validate/VAS/IAS on the contents of *u*, *p* or *gp*, we want to know that none of these nodes are deleted (or else we might succeed erroneously where an SCX would detect a marked node and fail).

It turns out that we can ensure a validate/VAS/IAS on a deleted node will fail with the following **Synchronization Rule:** *Any operation that deletes a node will be performed via an IAS operation which will invalidate all of the nodes it deletes.* To see why this rule helps, note that as long as a node is tagged *before* it is deleted, its deletion will cause a subsequent validate/VAS/IAS on it to fail. But what about a node that is tagged *after* it is deleted? We avoid such cases by maintaining the following **Invariant:** *All tagged nodes were in the tree when we last validated successfully.*

We initially establish this invariant at the root of the tree by tagging the root node, then validating. If validation succeeds, we then tag the next node on the search path (*in addition* to the root), and validate again, extending the invariant to the first *two* nodes on the search path. And, after tagging the third node on the search path, we validate to extend the invariant yet again. At this point, we have three nodes, *u*, *p* and *gp*, that we know were in the tree when we performed our last validate instruction. The next challenge lies in showing that we can extend the invariant to the fourth node on the search path *without* keeping the root node tagged.

We tag the fourth node on the search path, and validate to extend the invariant to it. Then, we *untag* the root. Note that, immediately after untagging the root, it could be modified to delete our next tagged node, and a since the root is not tagged, a subsequent validate/VAS/IAS will *not* detect any changes to the root. However, any such deletion will still cause validate/VAS/IAS to fail, because of our synchronization rule. Since our nodes were tagged *before* the root was untagged, deleting any of them *after* the root is untagged will cause them to be *invalidated*. And such a deletion *before* the root was untagged will clearly cause validate/VAS/IAS to fail, since the entire path is tagged.

Correctness Argument. We are now ready for the general argument. Suppose the invariant holds after our i-th consecutive successful validate instruction in a search. Let *u*, *p*, and *gp* be the nodes we currently have tagged. We read a pointer to the next node *n* on the search path from $u \rightarrow ptrs$, tag it, then validate. If validation succeeds, then *u*, *p* and *gp* are still in the tree at that time, since they were in the tree when we did our i-th successful validation, and they have not been *invalidated* since then (or validation would fail). It remains to argue that *n* was in the tree when we performed our $(i + 1)$-th validation.

Since our $(i + 1)$-th validation succeeds, *u* must be *unchanged* between our i-th and $(i + 1)$-th validations. Thus, when we did our i-th validation, *n* was a child of *u*, and *u* was in the tree, so *n* was as well. We can prove that *n* is not deleted between our i-th and $(i + 1)$-th validations with the help of our earlier *observation*: for *n* to be deleted, *u* or *p* must change. However, neither *u* nor *p* changes between these two validations, since the latter one succeeds. This proves that the invariant is extended to *n*.

As a result of our Synchronization Rule, we are able to prove that no validate/VAS/IAS ever modifies a deleted node. (I.e., we correctly simulate the *finalizing* performed by SCX.)

Linearization Points. Linearization points for operations are as follows: a successful update is linearized at its successful VAS/IAS. A search is linearized at the last successful validate. The use of tagging and VAS/IAS guarantees that updates are performed in a single atomic step. Note, however, that the HoH tagging argument is needed to argue that the modification is actually performed on *reachable nodes* (and not in a deleted part of the tree).

Generality. The above transformation is specific to the (a,b)-tree only in terms of the way we allow nodes to be *untagged*. In the (a,b)-tree, it suffices to tag *three ancestors*, since no operation atomically deletes a chain of more than *two nodes*, but in other data structures, more or fewer ancestors may need to be tagged. Note, however, that we are able to transform LLX/SCX variants of other data structures into efficient HoH-tagged versions. We have verified this for the chromatic tree, and we believe the same would hold for every tree implemented via LLX/SCX following [8, 31].

Rebalancing. After each successful insertion/deletion of key *k*, a procedure called Rebalance(*k*) is invoked (Algorithm 5). This procedure repeatedly finds any arbitrary balance violation on the search path to *k*, and invokes the appropriate rebalancing step (which will either eliminate the violation, or move it upwards in the tree—along the search path to *k*). This continues until it traverses the entire search path to *k* without seeing any balance violation.

An example rebalancing step, AbsorbSibling, appears in Algorithm 4. The transformation from the LLX/SCX version of AbsorbSibling to Algorithm 4 is straightforward (mechanical, in fact).

It may surprise the reader that nodes gp, p, l, s are first found in Rebalance(k), where the *flag violation field* and *number of keys* in l and s are used to decide which rebalancing step to perform, but they are only tagged *later* in AbsorbSibling. In the LLX/SCX version, LLX is performed on these nodes in the exact same place. Let us see why this is correct. The flag violation field and number of keys in a node *cannot* be changed in this algorithm. (One must *replace* a node with a new copy to accomplish this.) Only the *pointers* of a node can be changed. And, after tagging, we explicitly check that gp still points to p, and p still points to l and s. (These same checks are performed in the LLX/SCX version, after each corresponding LLX.) Thus, any change that makes it incorrect to perform AbsorbSibling will cause the IAS at the end of AbsorbSibling to fail (just as the corresponding SCX would fail).

5.2 Tagged NOrec TM

NOrec (No ownership records) [28] is a software transactional memory (STM) system that has very low fast-path latency. NOrec combines three key ideas: (1) a single global sequence lock; (2) an indexed write set; and (3) value-based conflict detection. NoRec uses a sequence lock to protect the transaction commit protocol. Readers check, after each read, to see if any writer has recently committed; if so, they perform value-based validation to make sure their previous reads, would return the values previously seen if performed at the current point.

Multiple read-only transactions can run and commit concurrently. A reader can upgrade to writer status at any time, but there can be only one writer, system-wide. While it precludes write concurrency, this system avoids the need for buffered writes, and requires only minimal instrumentation: readers check the sequence lock immediately after reading, to see if they must abort; writers acquire the sequence lock immediately before writing, if they do not hold it already.

Writing transactions do not attempt to acquire the sequence lock until their commit points, allowing speculative readers and writers to proceed concurrently. Any changes made by a writing transaction are buffered in a write log, which must be searched on each read to avoid possible read-after-write hazards. NOrec satisfies opacity [38], the standard correctness condition for transactional memory. After a transaction reads, it validates to determine whether the read set is still consistent. We now list the NOrec operations, briefly sketching their implementation.

- ReadSequence - Repeatedly read the global sequence lock until it is unlocked. Return the sequence number observed in the final read.
- TXBegin - Beginning a transaction in NOrec simply entails invoking ReadSequence and saving the result in a local variable V. Intuitively, V is the last sequence number at which the read set is known to be consistent.
- TXValidate - Invoke ReadSequence and if it returns V, then no writers have committed since the transaction began, so return `True`. Otherwise, value-based validation (VBV) is needed.
  VBV: Invoke ReadSequence and let V' be the return value. For all addresses in the read set, check whether they contain the same values that were seen previously. If not, return `False` (triggering a transaction abort). Invoke ReadSequence and check whether it is equal to V'. If so, the read set is consistent as of the last read of the sequence lock (since no writing transactions committed between these two invocations of ReadSequence), so V is set to V' and `True` is returned. Otherwise, TXValidate starts again.
- TXRead - First check if this transaction has already written to this address. If so, the value saved in the read set is returned. Otherwise, TXRead reads the address and saves it in the read set, then invokes TXValidate. If TXValidate returns `True`, TXRead returns the saved value. If not, the transaction aborts.
- TXWrite - The value to be written is stored in a *write buffer*, alongside the target address it should be written to.
- TXCommit - The validation protocol tries to atomically increment the sequence lock from V to $V+1$ using CAS (acquiring the lock). If this CAS succeeds, then no further validation is required because no other write has invalidated the read-/write set. So, the contents of the *write buffer* are written to their target addresses, and $V+2$ is written the sequence lock (releasing the lock).

Tagged NOrec. The key idea behind applying MemTags is that tagging the read set should allow for much more efficient validation. This serves two purposes. In the favorable case where the read transaction would succeed, it is able to do so immediately, via successful validation. In the unlucky case where the read transaction would abort, it would not need to perform value-based validation in order to simply fail. In both cases, we save significantly on cache traffic; this speeds up the read transactions, as well as the try commit. Acquisition of the global lock is done by IASing on it rather than spinning until the CAS succeeds. If the IAS fails, the sequence number is updated by invoking transaction validate to fetch the count at which the concurrent writer has committed. This is a simple modification, but induces significant performance benefits since the coherence traffic required by transactions is greatly reduced, leading to higher throughput. We illustrate the performance benefits of this in the case of NOrec applied to the STAMP Vacation benchmark in Figure 8. Examination of the simulator traces confirms that this performance improvement comes because of reduced coherence messaging.

6 EXPERIMENTAL EVALUATION

Setup and Methodology. To emulate the performance impact of MemTags, we implemented this mechanism in Graphite [32], a tiled multi-core chip simulator, with 1 to 64 1GHz cores, having private 32 KB caches, and 256 KB inclusive L2 caches, implementing a MESI coherence protocol. The cacheline size is the standard 64B. Simulation is run in *full* mode, which accurately models the application's instructions. We implemented the semantics of MemTags at each core's L1 cache. We note that Graphite does not simulate reordering, and that it only implements basic branch prediction.

The tests were performed in a controlled simulation environment, and results are consistent across runs. The implementation uses C++ and POSIX threads. The main metric we investigate is operation *throughput*, i.e. number of operations completed per time unit. Each data point is obtained by averaging over 5 runs.

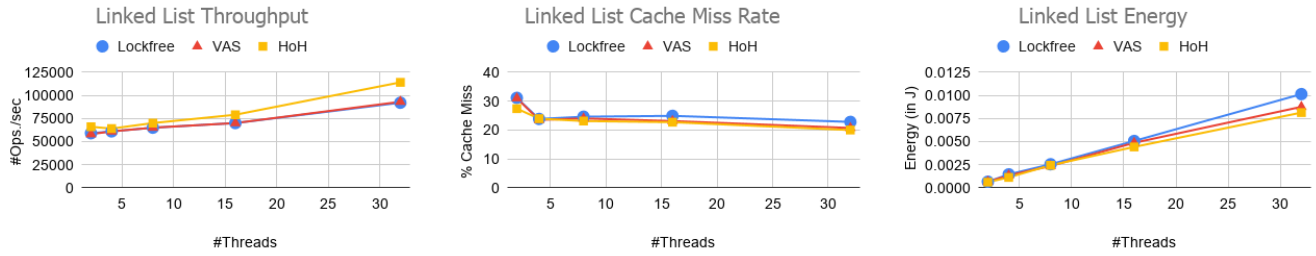


Figure 4: Throughput, Cache Miss Rate and Energy results for the lock-free Harris linked list, compared with our VAS-based and HoH-based implementations with 35% inserts, 35% deletes workload.

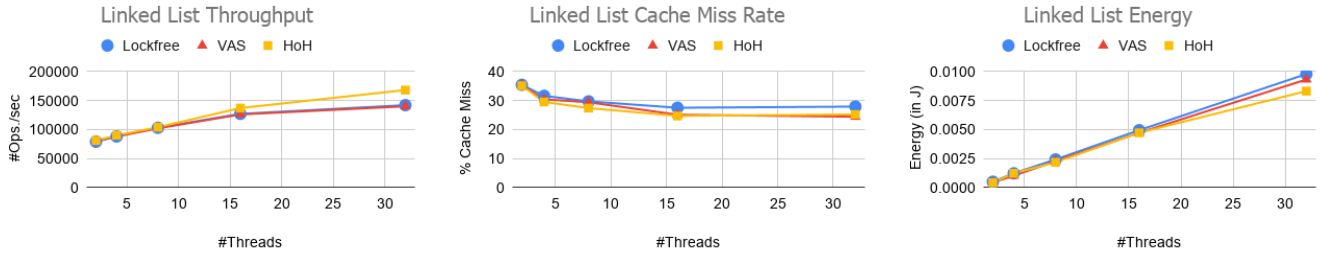


Figure 5: Throughput, Cache Miss Rate and Energy results for the lock-free Harris linked list, compared with our VAS-based and HoH-based implementations with 15% inserts, 15% deletes workload.

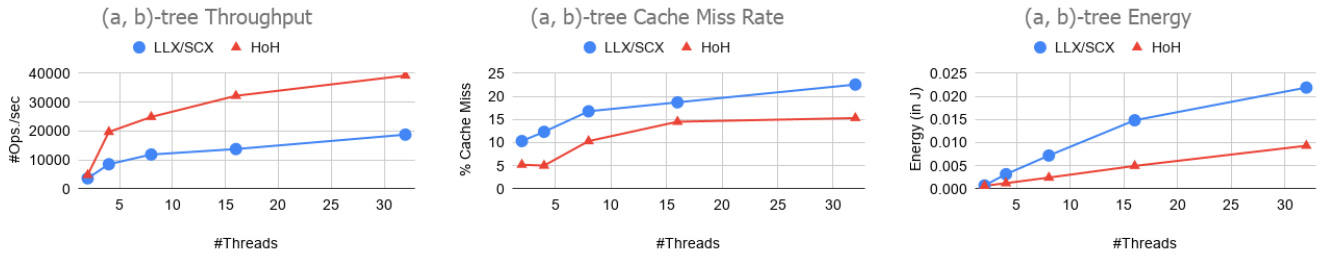


Figure 6: Throughput, Cache Miss Rate and Energy results for the (a,b)-tree, compared with our HoH-based implementations with 35% inserts, 35% deletes workload.

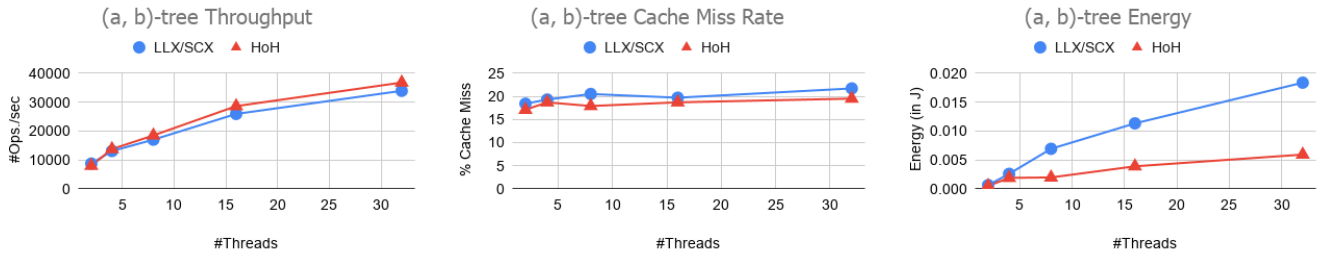


Figure 7: Throughput, Cache Miss Rate and Energy results for the (a,b)-tree, compared with our HoH-based implementations with 15% inserts, 15% deletes workload.

On every run, we set the initial size of the data structure and the key range that the threads operate on. On every iteration, each thread selects a key at random within the range.

The initial size is half of the range, and the percentage of insertions and deletions are the same, which keeps the data structure

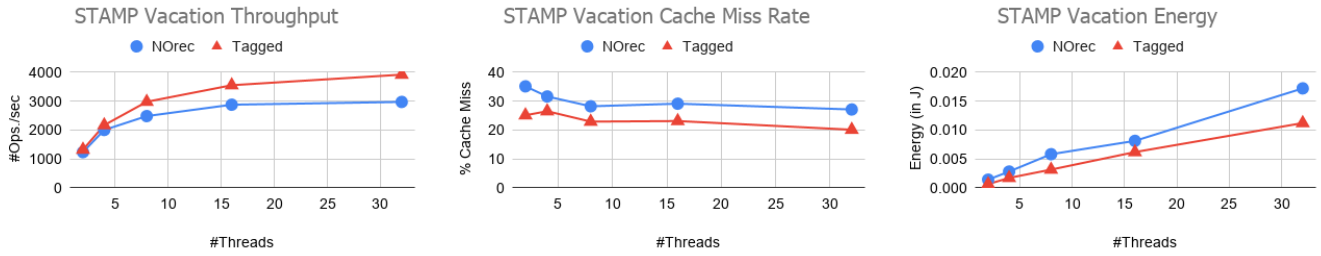


Figure 8: Throughput, Cache Miss Rate and Energy results for the STAMP vacation benchmark on NOrec, compared with our tagged implementations for -n4 -q60 -u90 -r16384 -t4096 parameter values.

size roughly constant. For the same reason, about half of the update operations return `False`.

We evaluated the performance of tagging for varying contention workloads for all data structures: between 80%/20% to 20%/80% reads versus updates. The results are consistent across the range, although the gains from tagging are more pronounced in settings with higher update rate. We present the results on a workload with 35% inserts, 35% deletes & 30% contains and 15% inserts, 15% deletes & 70% contains. Specifically, we tested tagging for a range of well-known concurrent data structure implementations in the simulator, including the Harris-Michael lock-free linked list [11, 12, 14], (a,b) tree [30] and NOrec STM [28] on the STAMP Vacation benchmark [33].

Implementation details. For the hand-over-hand tagging approach to work, we map each data structure node to a unique cache line, avoiding false sharing. Note that we could modify MemTags such that remove tag would not untag a cache line until all nodes mapped to it are untagged; however, this would lead to more frequent invalidations due to what is essentially false sharing.

Discussion. The results demonstrate that under moderate-to-high update rates, tagging can improve throughput significantly. MemTags can speed up highly optimized linked list by 10 to 30%, balanced search trees by up to 2× versus the optimized implementations based on LLX/SCX [8]. In practice, we observed that the impact of cache line eviction for reasonable sizes of the data structure was negligible.

Since only update methods of data structures are affected by tagging, we depict the results for scenarios with more write operations. For read operations (without contention), the implementations remain the same and the throughput is same as in the original structures. Closer examination of the execution logs reveals that the overhead of spurious invalidations is negligible (< 1%), while the performance benefits of tagging are directly correlated to reduced cache traffic.

7 CONCLUSIONS

While many systems providing hardware support for concurrency, such as TM, have been originally introduced with non-blocking data structure applications in mind [1], such methods have gradually shifted towards providing broader support for other forms of concurrency. In this paper, we have re-visited the question of efficient architectural support for non-blocking semantics, and investigated the power of hardware-data-structure co-design by proposing a method called MemTags, and investigating its impact on concurrent search data structures.

Our results suggest that MemTags induce a new, non-trivial point in the space of trade-offs between usability, expressivity and performance for concurrency abstractions. For instance, for balanced search trees, MemTags are the only technique that is able to simultaneously avoid atomically validating an entire root-to-leaf path *and* support updates via a single atomic pointer change. (The (a,b)-tree implemented with LLX/SCX must perform *finalizing*, in addition to changing a pointer, to obtain atomicity.) Our results suggest that MemTags can be used to significantly increase performance for search data structures. Additionally, MemTags may simplify the design of some data structures. For example, one could pair a design that uses HoH tagging with a simple slow path that uses a global lock to ensure atomicity (similar to transactional lock elision). Of note, rather than requiring completely new designs, MemTags serve as an efficient fast-path for existing non-blocking data structure mechanisms.

REFERENCES

[1] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, (New York, NY, USA), pp. 289–300, ACM, 1993.

[2] I. Corporation, "Intel architecture instruction set extensions programming reference, chapter 8," 2013.

[3] D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii," in *Distributed Computing*, pp. 194–208, Springer, 2006.

[4] R. Rajwar and J. R. Goodman, "Speculative lock elision: Enabling highly concurrent multithreaded execution," in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, (Washington, DC, USA), pp. 294–305, IEEE Computer Society, 2001.

[5] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear, "Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 39–52, ACM, 2011.

[6] T. David, R. Guerraoui, and V. Trigonakis, "Asynchronized concurrency: The secret to scaling concurrent search data structures," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 631–644, ACM, 2015.

[7] R. Guerraoui and V. Trigonakis, "Optimistic concurrency with OPTIK," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, (New York, NY, USA), pp. 18:1–18:12, ACM, 2016.

[8] T. Brown, F. Ellen, and E. Ruppert, "Pragmatic primitives for non-blocking data structures," in *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013*, pp. 13–22, 2013.

[9] T. Skare and C. Kozyrakis, "Early release: Friend or foe," in *Workshop on Transactional Memory Workloads*, vol. 77, 2006.

[10] S. K. Haider, W. Hasenplaugh, and D. Alistarh, "Lease/release: Architectural support for scaling contended data structures," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, (New York, NY, USA), pp. 17:1–17:12, ACM, 2016.

[11] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the International Conference on Distributed Computing (DISC)*, pp. 300–314, 2001.

[12] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pp. 73–82, ACM, 2002.

[13] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. S. III, and N. Shavit, "A lazy concurrent list-based set algorithm," in *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS 2005), Revised Selected Papers* (J. H. Anderson, G. Prencipe, and R. Wattenhofer, eds.), vol. 3974 of *Lecture Notes in Computer Science*, pp. 3–16, Springer, 2006.

[14] M. Herlihy and N. Shavit, *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[15] K. Fraser, "Practical lock-freedom," Tech. Rep. UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, Feb. 2004.

[16] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit, "A simple optimistic skiplist algorithm," in *Proceedings of the 14th international conference on Structural information and communication complexity*, SIROCCO'07, (Berlin, Heidelberg), pp. 124–138, Springer-Verlag, 2007.

[17] D. Lea, 2007. http://java.sun.com/javase/6/docs/api/java/util/concurrent/ConcurrentSkipListMap.html.

[18] M. Fomitchev and E. Ruppert, "Lock-free linked lists and skip lists," in *Proceedings of the 23rd annual ACM symposium on Principles of Distributed Computing (PODC' 04)*, (New York, NY, USA), pp. 50–59, ACM Press, 2004.

[19] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, "Non-blocking binary search trees," in *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, (New York, NY, USA), pp. 131–140, ACM, 2010.

[20] A. Natarajan and N. Mittal, "Fast concurrent lock-free binary search trees," in *ACM SIGPLAN Notices*, vol. 49, pp. 317–328, ACM, 2014.

[21] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, (New York, NY, USA), pp. 267–275, ACM, 1996.

[22] R. K. Treiber, "Systems programming: Coping with parallelism," Tech. Rep. RJ 5118, IBM Almaden Research Center, 1986.

[23] N. Shavit and D. Touitou, "Elimination trees and the construction of pools and stacks: preliminary version," in *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pp. 54–63, ACM, 1995.

[24] I. Lotan and N. Shavit, "Skiplist-based concurrent priority queues," in *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pp. 263–268, IEEE, 2000.

[25] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit, "The spraylist: A scalable relaxed priority queue," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, (New York, NY, USA), pp. 11–20, ACM, 2015.

[26] H. Rihani, P. Sanders, and R. Dementiev, "Brief announcement: Multiqueues: Simple relaxed concurrent priority queues," in *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, (New York, NY, USA), pp. 80–82, ACM, 2015.

[27] R. Bayer and M. Schkolnick, "Readings in database systems," ch. Concurrency of Operations on B-trees, pp. 129–139, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988.

[28] L. Dalessandro, M. F. Spear, and M. L. Scott, "NOrec: streamlining STM by abolishing ownership records," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, pp. 67–78, 2010.

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

[30] T. Brown, F. Ellen, and E. Ruppert, "A general technique for non-blocking trees," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pp. 329–342, 2014.

[31] T. Brown, *Techniques for Constructing Efficient Lock-free Data Structures*. PhD thesis, University of Toronto, 2017.

[32] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pp. 1–12, IEEE, 2010.

[33] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: stanford transactional applications for multi-processing," in *IISWC*, pp. 35–46, IEEE Computer Society, 2008.

[34] O. Ostrovsky and A. Morrison, "Scaling concurrent queues by using HTM to profit from failed atomic operations," in *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020* (R. Gupta and X. Shen, eds.), pp. 89–101, ACM, 2020.

[35] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari, "Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 144–157, ACM, 2015.

[36] N. Cohen, M. Herlihy, E. Petrank, and E. Wald, "The teleportation design pattern for hardware transactional memory," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 95, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[37] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st ed., 2011.

[38] R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, (New York, NY, USA), p. 175–184, Association for Computing Machinery, 2008.