Transactional Acceleration of Concurrent Data Structures

Yujie Liu
Lehigh University
yul510@lehigh.edu

Tingzhe Zhou
Lehigh University
tiz214@lehigh.edu

Michael Spear
Lehigh University
spear@cse.lehigh.edu

ABSTRACT

Concurrent data structures are a fundamental building block for scalable multi-threaded programs. While Transactional Memory (TM) was originally conceived as a mechanism for simplifying the creation of concurrent data structures, modern hardware TM systems lack the progress properties needed to completely obviate traditional techniques for designing concurrent data structures, especially those requiring nonblocking progress guarantees.

In this paper, we introduce the Prefix Transaction Optimization (PTO) technique for employing hardware TM to accelerate existing concurrent data structures. Our technique consists of three stages: the creation of a prefix transaction, the mechanical optimization of the prefix transaction, and then algorithm-specific optimizations to further improve performance. We apply PTO to five nonblocking data structures, and observe speedups of up to 1.5x at one thread, and up to 3x at 8 threads.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

Keywords

Lock-Freedom, Transactional Memory, Concurrency, Synchronization

1 Introduction

Modern parallel programs rely on concurrent data structures (CDS) to achieve scalable synchronization. Over the past two decades, dozens of concurrent data structures have been proposed, providing highly scalable stacks, queues, lists, trees, hash tables, skiplists, heaps, and many other data structures.

The design of concurrent data structures is challenging, for a number of reasons. First, they must achieve good performance across a variety of workloads. An implementation ought to have low latency when accessed by a single thread. This property is valuable for applications whose threads rarely access the data structure at the same time: if latency is too high, then the programmer may instead opt to protect a sequential data structure with a coarse-grained lock. However, an implementation should also exhibit high scalability. That is, in highly concurrent workloads, threads should not impede each others' progress when they access disjoint parts of the data structure. There is typically a tension between these goals: to ensure good scalability, a greater amount of metadata manipulation is required to coordinate potential concurrent accesses to the data structure; however, the injection of metadata introduces overhead to the streamlined sequential implementations, and moreover, metadata accesses often require the use of expensive atomic synchronization primitives, such as compare-and-swap (CAS). Introducing fine-grained metadata can result in more atomic primitives per operation, and thus more latency.

Secondly, many programs expect progress guarantees from concurrent data structures. Wait-freedom [19], the strongest progress guarantee, ensures that at any point in an execution, regardless of the states of other threads, there exists a finite bound on the number of steps for any thread to complete its operation. Wait-free algorithms often require expensive synchronization among threads. The weaker guarantee of lock-freedom [20], where at any point in a program's execution there exists *some* thread that can complete its operation in a finite number of steps, has been achieved in many practical data structures. However, even in lock-free data structures, there is often significant overhead to make concurrent updates to multiple locations appear atomic (e.g., by simulating a multi-word compare-and-swap [15, 32]).

Hardware Transactional Memory (HTM) [21] was originally designed to simplify the task of creating concurrent data structures. The idea behind HTM is simple: programmers mark regions of code that ought to execute as a single, indivisible operation, and then the hardware runs these "transactions" concurrently, while tracking their memory accesses. By tracking accesses, the hardware can identify conflicting memory accesses among transactions. By also providing a buffering mechanism, the hardware can abort, roll-back, and retry some of the transactions involved in a conflict, so that each transaction appears to execute in isolation.

Unlike research HTM proposals, the first-generation HTM systems from IBM [23, 44] and Intel [22] expose significant limitations, which limit their suitability to lock-free programming. These "best effort" HTMs [7, 27] do not guarantee progress for arbitrary transactions: a transaction attempt will fail if it (a) attempts to access too many distinct locations; (b) executes for longer than a scheduler quantum of the operating system; or (c) attempts to perform an unsupported operation, such as a system call. Transaction attempts can also fail due to memory accesses that conflict with concurrent operations from transactions, or accesses that conflict with concurrent nontransactional code. This property, called "strong atomicity" [2], is a natural outcome of implementing HTM through the cache coherence protocol. It also allows for clever composition of transactional and nontransactional code [7, 8, 46].

Even if these limitations did not exist, it is unlikely that HTM could ever fully replace the best concurrent data structure implementations. As recently reported by Gramoli [13], concurrent data structures implemented directly from atomic primitives (i.e. CAS) tend to provide the best performance in comparison to those implemented by using locks or transactions.

This paper explores how HTM might still benefit the design and implementation of concurrent data structures. Specifically, we propose a methodology, called Prefix Transaction Optimization (PTO), by which HTM can be used to accelerate an existing implementation. There are three components of PTO, which vary in terms of the degree to which they can be automated, the amount of implementation-specific knowledge needed, and the potential gain. In the first step, we create a prefix transaction to execute a sequence of steps in the existing implementation, which uses HTM but may fail. In the second step, we mechanically optimize this prefix through strength reduction and elimination of corner cases [38], and other classic compiler optimizations. In the third step, we modify the original algorithm so as to introduce minimal "overhead" while affording more aggressive optimization of the prefix transaction.

PTO offers many compelling properties. It preserves the progress guarantees of the original algorithm, which is an improvement over many approaches to transactional acceleration of concurrent programs. It is also a composable technique, which can be applied at multiple levels of granularity. PTO optimizations can be linked together, and the benefits of doing so are (more or less) additive. Lastly, and most significantly, PTO can dramatically improve performance. We observe speedups of up to 1.5x at one thread, and up to 3x at 8 threads, on state-of-the-art nonblocking data structures.

2 Prefix Transaction Optimization

In this section, we present the algorithm-agnostic aspects of the Prefix Transaction Optimization (PTO) technique. Enhancements and modifications specific to a single data structure or class of data structures are discussed in Section 3.

2.1 Model

The PTO technique is applicable to concurrent objects implemented in shared memory using read/write registers and common synchronization primitives (e.g. compare-and-swap, fetch-and-add, etc). The object interface defines a set of invocable operations.

We adopt the *control flow graph* representation [37] for each operation (and its sub-operations), where a node represents a step in the algorithm and an edge represents a transition in the control flow. We assume each operation has a single *start* node. For two nodes a and b in a control flow graph, a *dominates* b if any code path from start to b includes a. A *superblock* is a connected sub-graph where all nodes are dominated by a single *entry* node. An edge from a node within a superblock to one outside is called an *exit* edge.

We assume that HTM is supported by the architecture via three instructions: `TxBegin` starts a transaction, `TxEnd` commits the transaction, and `TxAbort` causes the transaction to abort. The `TxBegin` instruction can return more than once: if a transaction cannot commit for any reason, then the effects of the transaction are undone, and control returns to the point of `TxBegin` with a return value indicating the cause of the inability to commit. A return value of `OK` indicates that the code is running as a transaction.

HTM is assumed to provide strong atomicity [2]. When a hardware transaction is running, none of its effects are visible to any concurrent code; all effects become visible atomically at `TxEnd`. During the execution of the transaction, if any concurrent transaction performs a conflicting access, the HTM will choose (at least) one transaction to abort. If any nontransactional code performs a conflicting access, the transaction will immediately abort. Upon any abort, control will return to `TxBegin`, where the program can decide whether to attempt the transaction again.

2.2 The Prefix Transaction Transformation

Given a superblock B in a control flow graph G, the Prefix Transaction Transformation (illustrated in Figure 1) is constructed by attempting to execute the superblock using a hardware transaction. If that attempt fails, then the original version of code is invoked, without the use of a transaction. More precisely:

DEFINITION 1. *Let B be a superblock of a control flow graph G. The Prefix Transaction Transformation is a function $\mathcal{T}_B(G)$ that maps G to G', a copy of G with B replaced by B', such that:*

- *T_B is a copy of T where a `TxEnd` instruction is inserted at each exit edge, and zero or more `TxAbort` instructions are inserted at any edges of T except the exit edges;*
- *S is a `TxBegin` instruction with a branch to the dominator of T_B if the return value is `OK` and a branch to the dominator of B otherwise;*
- *Let B' be the superblock dominated by S with all nodes of T_B and B included.*

Given a transformation $\mathcal{T}_B(G)$ defined in Definition 1, we say B' is the *optimized superblock*. Inside B', we say T_B is the *prefix transaction* of B, and B is the *fallback*.

The following theorems capture basic properties of the Prefix Transaction Transformation. We first prove the correctness of our transformation, by constructing a refinement mapping [1] from the transformed implementation to the original (Theorem 2). We then prove the progress guarantee of the original implementation is preserved by the transformation (Theorem 3). Finally, implied by the theorems, we observe that the program may choose to explicitly abort a transaction at any point (within the transaction) without compromising correctness or progress conditions.

THEOREM 2 (REFINEMENT). *Let G be the control flow graph of some operation of an implementation I, and let I' be the implementation with $\mathcal{T}_B(G)$ applied to I. I' refines I.*

PROOF SKETCH. The mapping of states is simply an identical function that maps the states of I' to the states of I. For a process p taking a step in I', if the step is not a transactional instruction or access, we let p take a corresponding step in I. For a `TxBegin`, `TxAbort`, or a transactional access step in I', p takes no step in I. For a `TxEnd` step in I', let k be the number of steps p has taken in-between the `TxEnd` the last `TxBegin` step, we let process p take k steps in I. □

THEOREM 3 (PROGRESS PRESERVATION). *Let G be the control flow graph of some operation of a lock-free (or wait-free) implementation I, and let I' be the implementation with $\mathcal{T}_B(G)$ applied to I. I' provides lock-free (or wait-free) progress.*

PROOF SKETCH. Suppose I is lock-free. Then some operation in I completes if process p takes a bounded number of steps. For a given configuration c of I, let k be this bound. Then at most k steps are spent in superblock B before some operation completes, and since B contains at least one step, it can be executed no more than k times before some operation completes.

Let f be the refinement mapping constructed in Theorem 2. For a configuration c' in I' where $c = f(c')$, process p can spend at

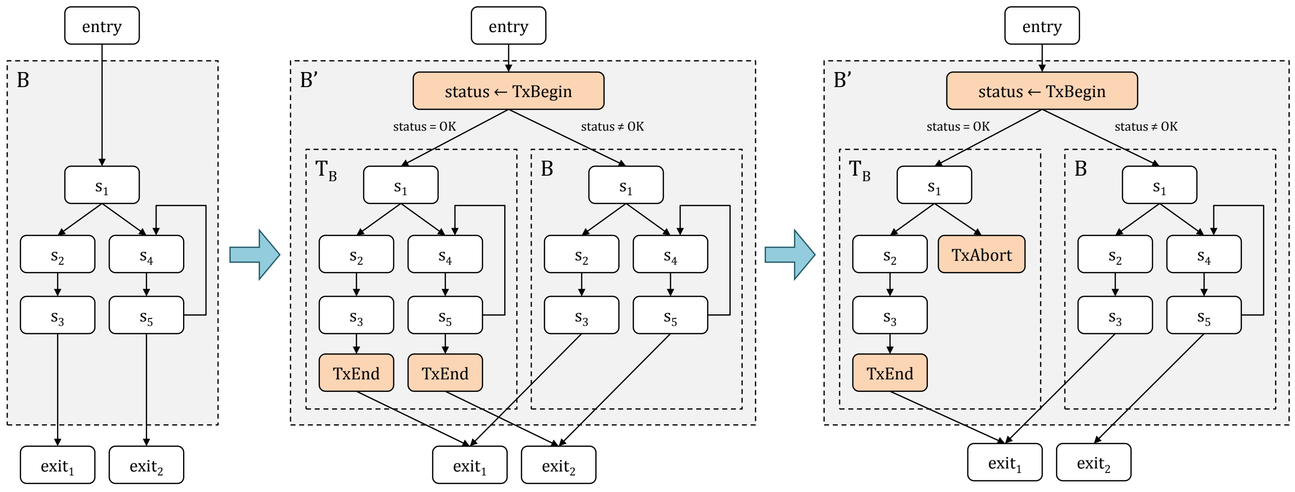


Figure 1: Prefix Transaction Transformation

most k steps in a transaction (excluding the `TxBegin`, `TxAbort` and `TxEnd` steps) before some operation completes. A committed or aborted transaction takes at most $(k + 3)$ steps including the `TxBegin`, `TxAbort` and `TxEnd` steps. In case the transaction aborts, at most $(2k + 3)$ steps are spent to execute the optimized superblock. Hence, we know in configuration c', some operation completes within $k \cdot (2k + 3)$ steps taken by process p.

Proving the preservation of wait-free progress employs the similar technique. □

2.3 Optimizing Prefix Transactions

We now turn our discussion to how to optimize the prefix transaction. We first present optimizations that can be easily identified and performed by a compiler using canonical static analyses.

Eliminating Synchronization: Correctness proofs of concurrent data structures often assume sequential consistency [26]. Implementations, in turn, must entail memory fences to enforce explicit ordering on architectures with weaker memory models.

Within a prefix transaction, memory fences can be elided, since they are subsumed by the implicit memory fences of `TxBegin` and `TxEnd` instructions, and atomic synchronization primitives, such as compare-and-swap and read-modify-write operations, can be replaced with their corresponding loads, stores, and branches.

Eliminating Redundant Loads: Double-checking is a technique used in many concurrent data structures [9,35]. Implementations employ double-checking to ensure a consistent view of multiple memory locations. In the prefix transaction, a single read to a shared location suffices, since the second read will always return the same value (given the transaction does not perform a write to the location in-between the reads); any conflicting write to the location will cause the transaction to abort.

For implementations that use the atomic compare-and-swap primitive, the compare-and-swap is usually attempted after a preceding read to the location. Since in a transaction we convert the compare-and-swap to a read followed by a conditional write, the read (produced by the conversion) can coalesce with the former read.

We also observe that many search data structures [9, 45] employ a *search phase*, followed by an *update phase* that performs its writes after validating selected locations accessed in the search phase. These implementations are likely to benefit from the elimination of redundant loads enabled by our transformation.

Eliminating Redundant Stores: Nonblocking data structures often exploit *intermediate states* during an update operation to allow helping from concurrent threads. The size of intermediate states may vary from unused bits embedded in the data fields [14, 28] to complex, dynamically-allocated auxiliary structures [9, 39]. Fundamentally, these intermediate states are introduced to overcome the difficulty that traditional synchronization primitives can update only a single word at a time.

It is commonly seen in nonblocking algorithms [9,29,39,41] that operations first attempt to change several locations from a clean state to some intermediate state, and then restore them back to a clean state. Given that an update is performed within a transaction, and all stores to a location appear atomically, the temporary change to intermediate states can be eliminated. Furthermore, if dynamic memory allocations are involved to create intermediate objects, these allocations can be eliminated together with the silent stores, mitigating pressure on the shared allocator object.

In CDSs using hazard pointers [34] or reference counts [42] to manage dynamic memory, intermediate updates to the hazard lists (i.e., insertion followed by removal) or to the reference counters (i.e., increment followed by decrement) can be safely eliminated as redundant stores in the prefix transaction.

2.4 Avoiding Helping in Prefix Transactions

Although helping is the key idea behind many nonblocking concurrent data structures, it tends to increase contention among threads in some cases [16, 25, 29, 36].

When a prefix transaction observes states in which it must perform helping to make progress, it may be preferable to simply abort the transaction and switch to executing the lock-free fallback. The rationale governing such decision is twofold: First, when a prefix transaction determines to help, the situation suggests a concurrent operation is accessing locations touched by the transaction (and vice versa) and is likely to create a conflict that causes the transaction to abort. Thus, the explicit abort can serve as an ad-hoc backoff mechanism to avoid the contention in the first place. Second, if the prefix transaction is optimized (as discussed in Section 2.3) so that it does not introduce intermediate states, it can be desirable to maximally avoid helping (which introduces intermediate states) in the prefix transaction for the sake of improving total throughput.

From a pragmatic perspective, we argue that it is fairly straightforward for a concurrent data structure designer to identify the helping code paths in the algorithm, and decide whether to replace them with explicit aborts in the prefix transactions. Examples of how to make such choices are discussed in subsequent sections of the paper. On the other hand, we found that in most nonblocking algorithms, a helping code path can be defined as an unreachable sub-path in the control flow graph of a single-threaded executions. A trivial example is the code to handle a failed compare-and-swap operation. Using this definition as a heuristic, an optimizing compiler can collect information from a single-threaded profile run, and (approximately) identify the helping paths for making optimization decisions.

2.5 Recursive Optimizations

Prefix Transaction Transformation is a *local* optimization, which means it can be applied to a whole operation or to individual components (superblocks) of the operation. More importantly, the optimization can be repeatedly applied on optimized code until achieving the best performance.

The simplest example of an recursive optimization is to allow an aborted prefix transaction to retry before attempting the fallback. For instance, the following transformation attempts the same prefix transaction T_B twice before switching to the fallback:

$$\mathcal{T}_{\mathcal{B}}(\mathcal{T}_{\mathcal{B}}(G))$$

A more powerful use of recursive optimization is to compose optimizations in a hierarchical structure. Suppose that in the control flow graph G of some operation, superblock B is a sub-graph of superblock A. The following transformation:

$$\mathcal{T}_{\mathcal{B}}(\mathcal{T}_{\mathcal{A}}(G)) \text{ where } B \subset A$$

first attempts the prefix transaction T_A, and in the fallback path of T_A, the program can still benefit from the optimizations of $\mathcal{T}_{\mathcal{B}}(G)$.

Hierarchical composition has an important impact in practice: Applying the transformation on larger superblocks maximizes the opportunity for eliminating redundancy (i.e. loads, stores, and fences), but makes it harder for transactions to make progress under contention, and thus, hurts scalability. Applying the transformation on smaller superblocks facilitates making progress, but reduces the opportunity to reduce latency. Composing optimizations makes it possible to achieve low latency and high scalability at the same time. We also notice that, by Theorem 3, applying the transformation for a bounded number of times preserves the progress guarantees of the original implementation.

3 Applying Prefix Transaction Optimization

The PTO technique presented in Section 2 does not require much algorithm-specific knowledge, though a programmer with knowledge about expected common paths may insert explicit aborts to increase optimization opportunities. We now turn our attention to the technical details of applying PTO to specific concurrent data structures, including additional algorithm-specific optimizations.

3.1 Data Structures with Simple Applications

Mindicators We first consider the Mindicator data structure [28]. Like SNZI [10] and the f-array [24], the Mindicator is a static-sized tree that computes a function over a set of values, where each thread offers at most one value as an input to the function. The original Mindicator algorithm uses a marking phase to traverse from a per-thread leaf up to some point in the tree, and unmarks nodes as it traverses back to the leaf. Unlike f-Array, not all operations must traverse to the root; unlike SNZI, additional functions (min, max) are supported in addition to 0/1 saturating addition.

The application of PTO to the Mindicator did not make use of any algorithm-specific optimizations, primarily because the tree is static and hence there is no memory allocation. By applying PTO, the marking and unmarking steps could be coalesced: marking and unmarking were previously both implemented as increments to a per-node counter; with PTO, the counter is incremented once, by two. This, in turn, eliminated the downward traversal entirely. After applying PTO, we tuned the threshold for retries before PTO falls back to the lock-free slow path. A choice of three attempts yielded the best performance.

Mounds We also applied PTO to the Mound [29], a heap-like data structure that implements a priority queue. Like the Mindicator, the Mound is a tree-shaped data structure. However, it is a tree of sorted lists, where each list is only modified at its head. We did not choose the Mound because it is the best nonblocking heap or priority queue. We chose it instead for the value it adds when evaluating PTO. Specifically, the Mound employs double-compare-and-swap (DCAS) and double-compare-single-swap (DCSS) operations throughout its implementation, to perform atomic updates on up to two locations. This afforded an opportunity to evaluate the impact of applying PTO locally, e.g., to individual DCAS and DCSS operations.

In the Mound, insertion consists of a search, followed by a double-compare-single-swap (DCSS), which is implemented in software through a sequence of CAS instructions. Removal entails performing a CAS to remove the top of the heap, and then several DCAS operations to restore invariants at the root and then on its children, recursively. Insertions can barely benefit from PTO, because they are streamlined and contention-free already: the heap itself is a static tree, obviating memory management overheads, and the insertion entails a log-log-depth traversal and just one simulated DCSS. Similarly, employing PTO on the entire removal operation is not effective at any level of concurrency, since all concurrent removals contend at the top of the heap. However, it is profitable to use PTO on a sub-operation of insert and removal, namely the DCAS/DCSS operations.

Skip Lists Lock-free skip lists [12] are a widely used search data structure to implement concurrent maps and sets. In the skip list algorithm, an update operation first locates the predecessor and successor nodes of a given key value, and then uses a sequence of compare-and-swap operations to link/unlink the nodes into the hierarchy of lists.

We experimentally determined that local application of PTO was the only promising technique. We proceeded to apply PTO only to the insert and remove operations. In an insert operation, we use a prefix transaction to update the next pointers of the predecessors. Similarly, in a remove operation, we attempt to mark the deleted node's next pointers using a single transaction, instead of performing individual compare-and-swap operations.

3.2 Nonblocking Binary Search Trees

We now discuss our experience with applying PTO to the nonblocking binary search tree (BST) algorithm created by Ellen et al. [9]. The algorithm implements a set object with insert, remove, and lookup operations. To achieve lock-freedom, the algorithm employs a "marking" technique to coordinate concurrent updates to the BST. During an insert or a remove operation, the thread first traverses down the tree (the search phase) to locate an appropriate position to perform the update. Then in the update phase, the thread allocates an operation descriptor (Info record) that contains sufficient information to allow helping from other threads. The descriptor is installed at nodes involved in the update, using compare-and-swap operations: one node is marked in an insertion and two are marked in a removal. An operation linearizes if it successfully marks all nodes involved in the update. Upon completion, some of the nodes are restored to a clean state.

We identify two opportunities to apply PTO in the binary search tree algorithm. The first is to put the entire update operation inside a transaction. The second is to use a prefix transaction to execute the update phase, leaving the search phase out of the transaction. In both choices, we can eliminate the allocation of the descriptor for an insert operation, because the node is restored to a clean state at the end of the transaction. For a remove operation, since the algorithm does *not* restore one of the updated nodes to a clean state, we cannot safely eliminate its descriptor. However, we can use a unique, statically-allocated dummy descriptor in place of a dynamically allocated one: When all updates are performed in a transaction, there is no need for helping if the transaction commits, and the dummy descriptor is simply ignored by subsequent operations.

3.3 Dynamic-Sized Hash Tables

The final data structure we studied is a nonblocking resizable hash table [30]. The algorithm employs a "freezable set" abstraction to achieve nonblocking size adjustments. In the hash table, each bucket is a pointer to a freezable set object, which is implemented as an unsorted array of elements. All updates to the array are performed via copy-on-write, that is, by creating an updated version of the array to replace the old one, and then using a compare-and-swap on the bucket pointer.

A straightforward application of PTO on the hash table appears barely helpful, since the algorithm is streamlined: In the common case, an insert or a remove operation on the hash table consists of a single allocation and an uncontended compare-and-swap on the bucket pointer. To improve performance, we changed the algorithm by removing the copy-on-write within transactions.

The idea of our optimization is to perform *speculative in-place writes* to the array objects, so that allocations could be avoided in the common case. When making this change, we attached a counter to the bucket pointers, so that a transactional update could increment the counter and modify the bucket in place. Unfortunately, this can affect the correctness of a concurrent lookup to the bucket. To prevent errors, we degrade the progress of lookups from wait-free to lock-free, by requiring lookups to double-check the bucket pointer counter after they search the bucket.

4 Evaluation

In this section, we evaluate the effectiveness of PTO in accelerating concurrent data structures. We consider five data structures as discussed in Section 3, which affords us the ability to look at the various aspects of PTO in detail.

4.1 Microbenchmarks

We use three microbenchmarks in our experiments:

setbench evaluates of set implementations which support insert, remove, and lookup operations. Each thread repeatedly invokes a lookup or an update operation (with equal chance of being an insert or a remove) with some random value within range.

pqbench evaluates priority queue implementations where each thread repeatedly invokes a push with some random value or a pop; the pop returns a null value if the queue is empty.

mbench evaluates Mindicator objects where each thread repeatedly invokes an arrive operation with some random value, followed by a depart operation.

All experiments were conducted on an machine equipped with an Intel Core i7-4770 CPU running at 3.40GHz, with 8 GB of RAM. The i7-4770 supports Intel's Restricted Transactional Memory (rtm) interface. There are 4 cores, each 2-way multi-threaded, for a total of 8 hardware threads. The software stack included Ubuntu 14.04.1 and GCC 4.8.2. All experiments were run in 32-bit mode, and data points are the average of 5 trials.

4.2 Latency and Scalability Improvement

Figure 2(a) contrasts the performance of the PTO Mindicator with the original lock-free implementation. We also compare to a version in which the Mindicator is protected by a coarse-grained lock, and the transactional lock elision (TLE) [40] is employed to allow concurrency. In the experiment, threads repeatedly insert and then remove a randomly-chosen value; this ensures that some operations must traverse to the top of the tree. We configured the Mindicator as a binary tree with 64 leaves, and used the default mapping, where threads were assigned to leaves from left to right.

There are two important trends: First, we see that at a single thread, PTO provides latency that is nearly as good as TLE, which does not have marking, unmarking, or helping phases. Thus we can conclude that PTO can provide near-optimal single-thread performance. Second, we see that whereas TLE scales poorly, due to its locking fallback, PTO scales comparably to the original lock-free code. Thus in all cases, PTO is on par with the best performing algorithm. Furthermore, beyond 4 threads we see that PTO scales better than the lock-free code. This is a natural consequence of the workload: when using random keys, as the number of threads increases, the likelihood that any thread must traverse to the root decreases. As fewer threads traverse to the root, the likelihood of conflicts for any thread also decreases, and the prefix transaction becomes more likely to succeed.

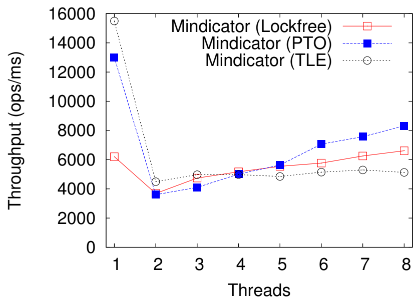
Figure 2(b) shows performance for a workload with an even mix of insert and removeMin operations on the Mound, using random keys. Using PTO, we were able to replace up to five CAS operations with a single transaction for each of the DCAS and DCSS operations. We encapsulated the DCAS in a function, and tuned the retry parameter once, ultimately settling on a value of four. This value was used for all DCASes, whether at the (high contention) root of the Mound, or at leaves.

The main benefit of PTO for the Mound was in removing latency from each DCAS. This result is similar to the finding of Yoo et al. [46], that coarsening atomic regions via hardware transactions can amortize some of the costs of atomicity. In terms of concurrent data structure design, the lesson is that thinking in terms of DCAS and other simpler primitives remains useful: assuming the availability of DCAS allowed the Mound designers to split removeMin into multiple atomic operations, thereby limiting the duration of contention on the Mound root.

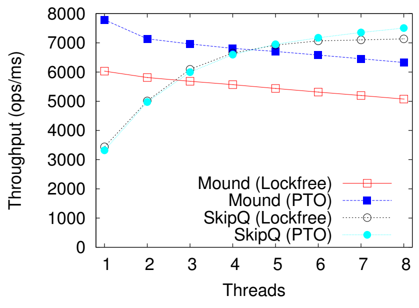
4.3 Impacts on Relative Performance

We next turn our attention to skiplists. We evaluate skiplists in two settings: as a search data structure (Figure 3) and as a priority queue (Figure 2(b)).

We began with Gramoli's skiplist implementation [13]. To create a skiplist priority queue, we employed a modified version of the

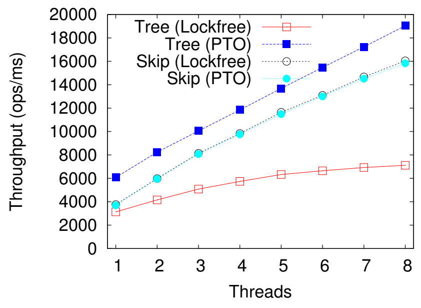


(a) Mindicators

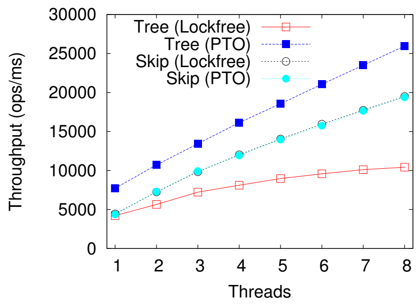


(b) Priority Queues

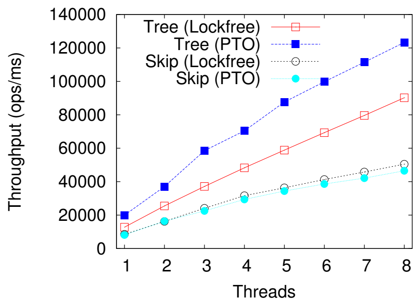
Figure 2: Mindicator and Priority Queue Microbenchmarks



(a) Lookup=0% Range=512



(b) Lookup=34% Range=512



(c) Lookup=100% Range=512

Figure 3: Logarithmic Search Data Structure Microbenchmark

Lotan-Shavit technique [31], and made it linearizable by disallowing a pop operation from traversing through an marked node.

While we expected to observe a similar decrease in latency to the Mound, due to the reduced latency for coarsened atomic operations, such benefit did not manifest. There are two drivers of this result. First, the main source of latency is not silo maintenance, but accessing locations that are not in the cache, during the traversal stage. According to the criteria in [5], the skiplist implementation is already close to optimal with respect to concurrency. Thus at one thread, there was little to gain. The second impediment to speedup at higher thread counts is that as a silo maintenance operation traverses the silo, it becomes increasingly prone to conflicts with concurrent readers. Intel TSX employs a requester-wins [3] conflict detection strategy, and thus any read to the write set of an PTO operation causes the PTO operation to fail.

4.4 Additive Benefits in Recursive PTO

PTO is a compositional technique, and can be used to optimize an entire operation, as well as a portion of its fallback path. To assess this property, we evaluated the nonblocking BST created by Ellen et al. [9]. We transliterated the code from Java to C++, replacing `volatile` variables with sequentially consistent `std::atomic` variables. We also employed an epoch-based memory reclamation policy, to ensure that locations were not reclaimed while a concurrent thread held a reference to them.

We identified two applications of PTO to the BST, which we refer to as PTO1 and PTO2. In PTO1, the entire insert, remove, and lookup operations are transformed using PTO. By optimizing the lookup phase, we are able to remove code that double-checks the values of reads. We were also able to replace sequentially consistent `std::atomic` accesses with relaxed accesses, which may avoid processor and compiler fences on some architectures.

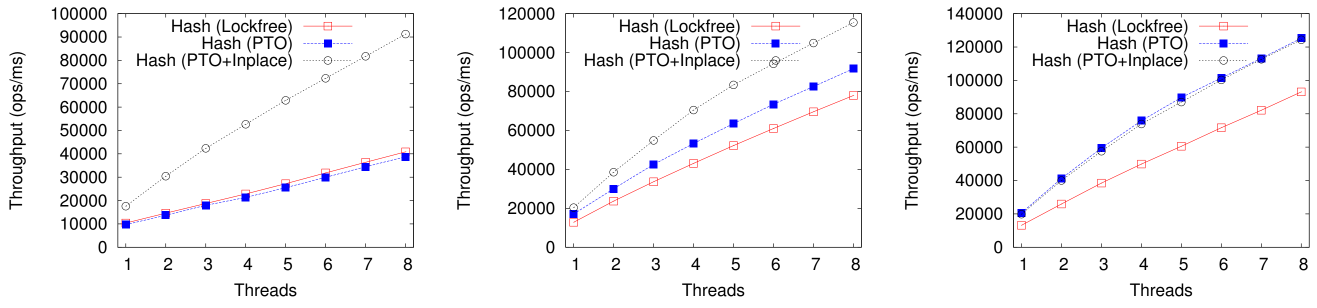
As Figure 5(a) shows, PTO1 results in more than 75% higher throughput at low thread counts. In contrast, PTO2 only optimizes the update phase of the insert and remove operations. While it also offers an improvement at all thread counts, the effect is much less at low levels of concurrency, where search overhead dominates, but much higher as concurrency increases. The improvement at higher thread counts is a consequence of a smaller contention window: since the traversal is not part of the hardware transaction, there are fewer opportunities to conflict with concurrent transactions. However, the lookup phase no longer runs in a hardware transaction, and thus must incur the overheads of double-checking and fences.

In PTO1+PTO2, we employ PTO1, and then use PTO2 within the fallback path. To fall back all the way to the original lock-free algorithm, an operation must first fail 2 times in PTO1, and then 16 times in PTO2. This composition achieves close to the best of both approaches. Even more remarkably, the composition of PTO+PTO2 boosts the BST performance to a constant factor higher than the skiplist set. As Figure 3 shows, the optimized BST provides the same scalability as the skiplist, but with lower latency.

4.5 Fast Speculative Inplace Updates

We ported the hash table from Java to C++, again using an epoch-based memory reclamation policy. We applied PTO to each of the insert, lookup, and remove operations, and then performed algorithm-specific optimizations to eliminate copy-on-write.

The simple application of PTO does little to benefit updates, since their overhead is dominated by the cost of allocating a new bucket, copying the old bucket's values, and applying the corresponding insert or removal. However, lookup operations show a



(a) Lookup=0% Range=64K (b) Lookup=80% Range=64K (c) Lookup=100% Range=64K

Figure 4: Hash Table Microbenchmark

decrease in latency. When PTO is applied to the lookup, all interaction with the epoch-based reclaimer can be elided. This eliminates two memory fences and two stores. Given the streamlined code path, there is a noticeable impact on latency.

Figure 4 presents the performance improvement for this optimization. In a write-only workload, we observe more than 2x speedup at 8 threads, and 1.8x speedup at one thread. The improvement is a consequence of the elimination of copying, and reduced interaction with the allocator. Since the allocator can require system calls, and its metadata can present a bottleneck, the benefits increase at higher thread counts.

4.6 What Makes PTO Fast?

Our evaluation shows some dramatic improvements in performance, particularly for the BST and hash table. However, it also shows some more modest gains, and fails to improve the skiplist at all. While the methodology does simplify the task of accelerating a concurrent data structure, it is still beneficial to be able to analyze an algorithm and predict whether it will benefit from PTO.

Generalizing our above experiments, we believe that there are four principal sources of latency that PTO can eliminate:

Memory Fences: Figure 5(b) and (c) present additional results for the Mound and BST, showing the impact when we did not elide memory fences within hardware transactions. For both the Mound, where the placement of fences was hand-optimized, and the BST, where the placement of fences mirrored their placement in the equivalent (and necessarily conservative) Java code, we see that the elimination of fences contributed significantly to savings in latency. For the Mound, the impact of removing fences was the sole source of improvement. For the BST, fences were a component of a suite of techniques that decreased latency.

Double-Checking Reads: Double-checked reads introduce two costs: not only do they add more instructions to the operation, but they also introduce branches, for when the check fails. In Figure 5(c), we break down sources of reduced latency for the BST write-only experiment. While fence removal plays a significant role, the baseline improvement comes without it. At low thread counts, where the entire operation is enclosed in a transaction, the credit is largely due to eliminating double-checking of reads.

Redundant Stores: In the Mindicator and Mound, the process of marking and unmarking nodes during an update or DCAS creates unnecessary work. Eliminating this work was the primary driver of improved latency in the Mound.

Allocation: The ability to replace copy-on-write in the hash table was single-handedly responsible for more than 2x speedup on the write-dominated workload. This improvement directly followed from the reduced interaction with the allocator. A similar benefit arose in the BST, where we were able to avoid allocating descriptors, but did not affect the Mound, where descriptors are reused from one operation to the next.

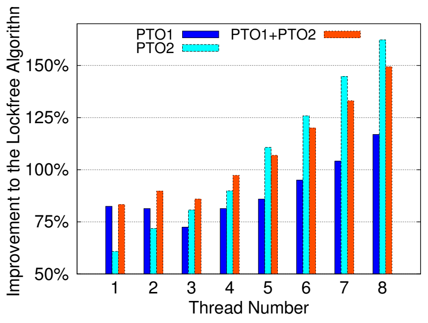
5 Rethinking Concurrent Data Structure Design and Implementation

We highlight two implications of PTO on concurrent data structure design.

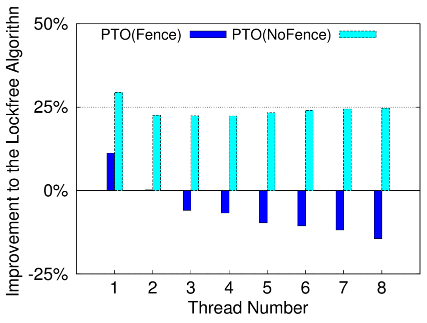
Optimization on Strengthened Invariants: We first observe that the use of a hardware transaction can strengthen some of the invariants of the original data structure. The most straightforward example is that within a hardware transaction, the intermediate states of an operation are not visible to other threads. In many nonblocking data structures, operation descriptors are installed by the operation to indicate that a certain objects are involved in an operation, and those descriptors are removed during the clean-up phase of the operation, after its linearization point. In many algorithms, it will be possible to avoid not only the installation and removal of descriptors, but also their allocation and deallocation.

Similarly, some algorithms employ hazard pointers to prevent objects from being made unreachable during critical periods in a method's execution. When the method is executed within a hardware transaction, there is an invariant that memory accessed by the transaction will not change due to external events. Thus it is not possible for an object accessed by a transaction T to become unreachable before T commits. While T must respect the hazard pointers reserved by concurrent (non-transactional) threads, T need not guard locations via hazard pointers during its own operation. In an analogous manner, hardware transactions do not need to update memory management epochs [12, 33]. This latter case clearly cannot be handled by the compiler, since epochs are represented with monotonically increasing counters. For short operations, such as those on hash tables, epoch operations and their corresponding memory fences can be a significant contributor to latency; for read-only operations, epochs can again be a significant cost, due to their introduction of memory fences.

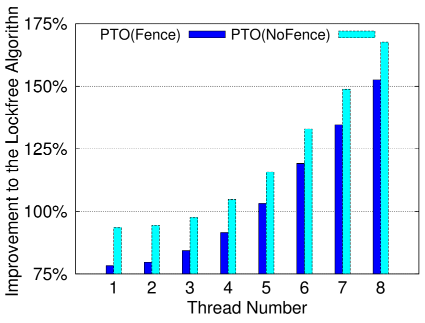
Progress vs. Optimization Trade-off: A more aggressive opportunity lies in weakening the progress guarantees of the original algorithm to increase the opportunity for fast-path optimization. There exist algorithms [16, 30] in which read-only lookup operations are wait-free. Reducing the progress of lookups to lock-free



(a) Composition of PTO on a Binary Search Tree



(b) Fence Elimination on Mound



(c) Fence Elimination on Binary Search Tree

Figure 5: Effectiveness of Specific Factors in PTO

can have non-local benefits by increasing the opportunity to optimize the PTO fastpath of inserts and removals.

In the hash table case, we see a PTO insertion or removal can modify the array in-place, as long as it increments the counter within its hardware transaction. Doing so ensures that concurrent lookups will not miss a value concurrently removed and inserted, at the cost of the operation retrying when there is concurrency. If concurrency between modifications and lookups is rare, or if modifications are, themselves, frequent, the optimization may outweigh the added overhead (and reduced progress guarantees) of the modified set. Modifications of this technique can be applied to algorithms that use copy-on-write, marking, descriptors, simulated DCAS, and indirection-based versioning of data.

In summary, we see significant potential to (re)design concurrent data structures to be PTO-friendly. If the prefix succeeds with high probability, then common costs, especially those related to memory management (reference counts, hazard pointers, epochs, indirection), become less significant. A slow-path that bears these costs, coupled with an unencumbered fast-path, may provide a "sweet spot" for algorithm designers. When these techniques cease to be performance bottlenecks, they may be employed to more rapidly develop novel concurrent data structures.

6 Related Work

While there are a number of high-performance nonblocking data structures, there are few methodologies for creating or accelerating them. Herlihy presented a universal construction for creating nonblocking data structures [18], but its emphasis was progress, not performance. Subsequent improvements [11] have increased the practicality of universal constructions, particularly for wait-free data structures. However, these techniques were designed before HTM became available in commodity microprocessors, and their focus on progress still results in overhead relative to the best ad-hoc nonblocking data structure designs. Similarly, Petrank et al. [25, 43] have created methodologies for making lock-free data structures wait-free, but without eliminating the overheads of the baseline lock-free data structure.

A variety of combining techniques have gained prominence for their ability to accelerate concurrent data structures [17]. Unfortunately, these techniques do not perform well on search data structures [17], and they sacrifice nonblocking progress. In contrast, our technique can perform well on search structures, and it preserves the original progress guarantees.

Neelakantam et al. were among the first to use HTM to optimize existing software [38]. Their focus was not on concurrency, but rather on speculative optimization of a program trace. As in our work, their system replaced unlikely code paths with explicit transactional aborts. Our work builds upon these ideas by introducing the notions of progress and composition to their transformation, and by extending it to concurrent code. We also add the option of algorithm-specific optimizations, instead of limiting to automatic compiler transformations.

Dice et al. were the first to analyze the impact of a real HTM system on concurrent data structures [7]. In their work, they showed that many concurrent applications could be simplified by attempting to execute operations in HTM, and then falling back to a single global lock if the HTM operation did not succeed.

Perhaps most notably, our work demonstrates that the fallback path is a significant design consideration. Early work on hardware lock elision [40] suggested that a locking fallback would suffice. Calciu et al. showed that optimizations to the lock-based fallback path could have significant impact on throughput [4]. Similarly, hybrid TM researchers have embraced the need for an intermediate point between HTM execution and serialized fallback.Our work does the same for concurrent data structures, demonstrating that the progress guarantees of the fallback path can play a significant role in overall system throughput.

Recently, Dice et al. pointed out several subtle pitfalls [6] in the lazy subscription technique [4]. We believe these issues do not apply to the PTO-accelerated data structures investigated in this paper. The intuition is that in TLE, the execution of fallback code assumes mutual exclusion. Speculative transactions inherit this assumption, since they run the same code as the fallback. Any violation of the mutual exclusion condition (even when observed by a speculative transaction) may lead the program to reach a state that would not be reachable in the original implementation. In PTO, the situation is nuanced. A prefix transaction coalesces multiple steps of the original implementation into a single super-step. Given that the original implementation is nonblocking, no step assumes mutual exclusion, and hence PTO will never cause the implementation to reach a state that was not reachable in the original code. However, this situation can lead to more aborts.

Yoo et al. studied the same Intel HTM implementation as us, applying it to high-performance-computing applications [46]. Like Dice et al., they employed HTM in a more ad-hoc fashion to a number of applications. They identified certain techniques, like transactional coarsening, that are captured by PTO. They also presented valuable guidelines for users of Intel's HTM, such as the importance of tuning retry parameters, and the possibility of different behavior for read-only and writing hardware transactions. As we saw in Section 4, this latter difference plays a significant role in the PTO BST and hash table algorithms.

Our techniques bear a complex relationship with the concept of asynchronized concurrency [5]. Certainly ASCY explains why it is so difficult to accelerate the skiplist. ASCY also provides insight into the PTO-accelerated BST: we essentially remove from the transactional fast-path those operations that are in opposition to the ASCY principles. However, in the case of the hash table, we see that sometimes it is beneficial to reduce the ASCY compliance of the original data structure in order to accelerate the transactional prefix. A weak reduction in ASCY (double-checking lookups) was able to deliver substantial improvement by obviating copy-on-write.

7 Conclusions and Future Work

In this paper, we introduced a methodology for accelerating concurrent data structures by using transactional memory. Our technique involves creating a fast-path transaction that succeeds or fails in bounded time, and a set of optimizations that can be applied to that fast-path to eliminate latency. In evaluation on five data structures[1], we saw performance benefits ranging from 50% to 3x for the hash table and binary search tree. Even when the methodology did not improve performance, we did not observe any significant slowdowns.

Apart from performance, our methodology offers many other benefits: It relies upon, and hence confirms the value of, strongly atomic hardware transactions. It preserves nonblocking progress, despite the absence of progress guarantees for current hardware transactional memory. Our technique is oblivious to the capacity of the underlying HTM. Lastly, it is both local and compositional. This last point is crucial, as it allows data structure designers to use existing mechanisms, such as lock-free DCAS, and then transactionally accelerate them.

Among the many future directions this work encourages, we highlight two. The first is for hardware designers, who we hope will see this work as an encouragement to reduce the latency of HTM boundary operations. As HTM becomes cheaper, we believe that PTO will become even more profitable, especially for DCAS replacement and other small transactions.

Secondly, we believe that the concurrent data structure community may want to re-think its approach to data structure design. For example, the accelerated BST dramatically outperforms the skiplist, even though the un-accelerated skiplist has been shown, repeatedly, to be among the most efficient and scalable lock-free ordered sets. PTO makes costly operations, such as copy-on-write, descriptor allocation, helping, and marking, inexpensive. This, in turn, encourages the design of nonblocking data structures with slower slow-paths, as long as they afford faster fast-paths.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants CAREER-1253362 and CCF-1218530. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

8 References

[1] M. Abadi and L. Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, pages 253–284, May 1991.

[2] C. Blundell, E. C. Lewis, and M. M. K. Martin. Subtleties of Transactional Memory Atomicity Semantics. *Computer Architecture Letters*, 5(2), Nov. 2006.

[3] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance Pathologies in Hardware Transactional Memory. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, June 2007.

[4] I. Calciu, J. Gottschlich, T. Shpeisman, G. Pokam, and M. Herlihy. Invyswell: A Hybrid Transactional Memory for Haswell's Restricted Transactional Memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, AB, Canada, Aug. 2014.

[5] T. David, R. Guerraoui, T. Che, and V. Trigonakis. Designing ASCY-compliant Concurrent Search Data Structures. Technical Report EPFL-REPORT-203822, Ecole Polytechnique Federale de Lausanne, 2014.

[6] D. Dice, T. Harris, A. Kogan, Y. Lev, and M. Moir. Pitfalls of Lazy Subscription. In *Proceedings of the 6th Workshop on the Theory of Transactional Memory*, Paris, France, July 2014.

[7] D. Dice, Y. Lev, V. Marathe, M. Moir, M. Olszewski, and D. Nussbaum. Simplifying Concurrent Algorithms by Exploiting Hardware TM. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[8] A. Dragojevic, M. Herlihy, Y. Lev, and M. Moir. On The Power of Hardware Transactional Memory to Simplify Memory Management. In *Proceedings of the 30th ACM Symposium on Principles of Distributed Computing*, San Jose, CA, June 2011.

[9] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking Binary Search Trees. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing*, Zurich, Switzerland, July 2010.

[10] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable NonZero Indicators. In *Proceedings of the Twenty-Sixth ACM Symposium on Principles of Distributed Computing*, Portland, OR, Aug. 2007.

[11] P. Fatourou and N. D. Kallimanis. A Highly-Efficient Wait-Free Universal Construction. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, San Jose, CA, June 2011.

[12] K. Fraser. *Practical Lock-Freedom*. PhD thesis, King's College, University of Cambridge, Sept. 2003.

[13] V. Gramoli. More Than You Ever Wanted to Know about Synchronization. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming*, San Francisco, CA, Feb. 2015.

[14] T. Harris. A Pragmatic Implementation of Non-Blocking Linked Lists. In *Proceedings of the 15th International Symposium on Distributed Computing*, Lisbon, Portugal, Oct. 2001.

[15] T. Harris, K. Fraser, and I. Pratt. A Practical Multi-word Compare-and-Swap Operation. In *Proceedings of the 16th International Conference on Distributed Computing*, Toulouse, France, Oct. 2002.

[16] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. Scherer, and N. Shavit. A Lazy Concurrent List-Based Set Algorithm. In *Proceedings of the 9th international conference on Principles of Distributed Systems*, Pisa, Italy, Dec. 2006.

[1]The source code from this evaluation is available on Github at mfs409/nonblocking.

[17] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat Combining and the Synchronization-Parallelism Tradeoff. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[18] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Structures. In *Proceedings of the Second ACM Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, Mar. 1990.

[19] M. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[20] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.

[21] M. P. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[22] Intel Corporation. Intel Architecture Instruction Set Extensions Programming (Chapter 8: Transactional Synchronization Extensions). Feb. 2012.

[23] C. Jacobi, T. Slegel, and D. Greiner. Transactional Memory Architecture and Implementation for IBM System Z. In *Proceedings of the 45th International Symposium On Microarchitecture*, Vancouver, BC, Canada, Dec. 2012.

[24] P. Jayanti. f-arrays: Implementation and applications. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, Monterey, California, July 2002.

[25] A. Kogan and E. Petrank. A Methodology for Creating Fast Wait-Free Data Structures. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, New Orleans, LA, Feb. 2012.

[26] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):241–248, Sept. 1979.

[27] Y. Lev and J.-W. Maessen. Split Hardware Transactions: True Nesting of Transactions Using Best-Effort Hardware Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[28] Y. Liu, V. Luchangco, and M. Spear. Mindicators: A Scalable Approach to Quiescence. In *Proceedings of 33rd International Conference on Distributed Computing Systems*, Philadelphia, PA, July 2013.

[29] Y. Liu and M. Spear. Mounds: Array-Based Concurrent Priority Queues. In *Proceedings of the 41st International Conference on Parallel Processing*, Pittsburgh, PA, Sept. 2012.

[30] Y. Liu, K. Zhang, and M. Spear. Dynamic-Sized Nonblocking Hash Tables. In *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing*, Paris, France, July 2014.

[31] I. Lotan and N. Shavit. Skiplist-Based Concurrent Priority Queues. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, Cancun, Mexico, May 2000.

[32] V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the 15th ACM Symposium on Parallel Algorithms and Architectures*, San Diego, CA, June 2003.

[33] P. E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

[34] M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[35] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, May 1996.

[36] A. Morrison and Y. Afek. Fast Concurrent Queues for x86 Processors. In *Proceedings of the 18th ACM Symposium on Principles and Practice of Parallel Programming*, Shenzhen, China, Feb. 2013.

[37] S. S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann, 1997.

[38] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware Atomicity for Reliable Software Speculation. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, June 2007.

[39] A. Prokopec, N. Bronson, P. Bagwell, and M. Odersky. Concurrent Tries with Efficient Non-Blocking Snapshots. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming*, Feb. 2012.

[40] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th IEEE/ACM International Symposium on Microarchitecture*, Austin, TX, Dec. 2001.

[41] N. Shafiei. Non-blocking Patricia Tries with Replace Operations. In *Proceedings of 33rd International Conference on Distributed Computing Systems*, Philadelphia, PA, July 2013.

[42] H. Sundell and P. Tsigas. Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems. *Journal of Parallel and Distributed Computing*, 65:609–627, May 2005.

[43] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank. Wait-Free Linked-Lists. In *Proceedings of the 16th International Conference on Principles of Distributed Systems*, Rome, Italy, Dec. 2012.

[44] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Minneapolis, MN, Sept. 2012.

[45] L. Xiang and M. L. Scott. Compiler Aided Manual Speculation for High Performance Concurrent Data Structures. In *Proceedings of the 18th ACM Symposium on Principles and Practice of Parallel Programming*, Shenzhen, China, Feb. 2013.

[46] R. Yoo, C. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High Performance Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, CO, Nov. 2013.