Hand-Over-Hand Transactions with Precise Memory Reclamation

Tingzhe Zhou
Lehigh University
tiz214@lehigh.edu

Victor Luchangco
Oracle Labs
victor.luchangco@oracle.com

Michael Spear
Lehigh University
spear@lehigh.edu

ABSTRACT

In this paper, we introduce *revocable reservations*, a transactional memory mechanism to reserve locations in one transaction and check whether they are unchanged in a subsequent transaction without preventing reserved locations from being reclaimed in the interim. We describe several implementations of revocable reservations, and show how to use revocable reservations to implement lists and trees with a transactional analog to hand-over-hand locking. Our evaluation of these data structures shows that revocable reservations allow precise and immediate reclamation within transactional data structures, without sacrificing scalability or introducing excessive latency.

CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**; **Concurrent computing methodologies**;

KEYWORDS

Transactional Memory, Concurrency, Synchronization, Data Structures, Memory Management

ACM Reference format:
Tingzhe Zhou, Victor Luchangco, and Michael Spear. 2017. Hand-Over-Hand Transactions with Precise Memory Reclamation. In *Proceedings of SPAA '17, Washington DC, USA, July 24-26, 2017,* 10 pages.
https://doi.org/10.1145/3087556.3087587

1 INTRODUCTION

Transactional memory (TM) [18] facilitates the implementation of concurrent data structures: instead of using locks, a programmer can simply designate a block of code to be executed as a transaction, and the system is responsible for executing the transaction atomically. Transactional memory implementations typically use speculation to execute multiple transactions concurrently. If concurrent transactions conflict, one or more of them is aborted and re-executed. To detect conflicts, a transaction typically maintains a "read set" of locations; the transaction is aborted if any of these locations are changed before the transaction commits.

Like critical sections in lock-based implementations, it is desirable to keep transactions as small as possible, in both time and space (i.e., duration and number of locations accessed): smaller transactions are less likely to conflict and abort. Furthermore, existing hardware support for transactional memory (HTM) [24] typically has capacity limits, forcing large transactions to fall back to slower software implementations (STM) [26].

Many operations on pointer-based data structures can be partitioned into a read-only *traversal phase* followed by an *update phase*. In the traversal phase, the operation follows a chain of nodes until it finds a node satisfying some condition. The found node is updated in the update phase. (The update phase is empty if the operation does not modify the data structure.) Some lock-based implementations reduce the size of critical sections of such operations by using *hand-over-hand locking*: An operation traverses the data structure by acquiring the lock for each node it traverses and then releasing each lock after the lock to the next node is acquired. The locks for all nodes accessed in the update phase must also be acquired, and they are not released until the end of the operation. Thus, during the traversal phase, each lock is held only while the next lock in the chain is acquired. This series of overlapping critical sections gives rise to the term hand-over-hand locking, and guarantees the atomicity of the entire operation.

Early release [17] and *elastic transactions* [13] achieve a similar effect for transactions by removing locations from a transaction's read set so that subsequent updates to the "released" locations do not cause the transaction to abort. Although they do not make transactions smaller, these techniques make transactions less likely to abort by reducing the size of their read sets. However, they cannot be applied to transactions executed by HTM, which provides no support for releasing locations.

We propose to hew more closely to hand-over-hand locking by dividing an operation's traversal phase into several read-only transactions, and executing the update phase as a single transaction. However, we cannot ensure atomicity by overlapping transactions as hand-over-hand locking overlaps critical sections. Instead, we introduce a mechanism to link consecutive transactions by "reserving" a location at the end of a transaction and checking the reservation at the beginning of the next transaction, aborting if the location has changed since the previous transaction committed. The composition of these linked transactions appears atomic because no updates are done until the final transaction.

One problem with reservations as described thus far: What happens if a reserved location is reclaimed? In that case, even reading the location in the next transaction may not be safe (e.g., it may result in a segmentation fault). We can avoid this problem by treating a reservation as a kind of hazard pointer [23], deferring the reclamation of reserved locations. Such deferral is not a significant concern for garbage-collected languages. However, in languages like C and C++, custom allocators are required, which must delay reclamation

SPAA '17, July 24-26, 2017, Washington DC, USA

ACM ISBN 978-1-4503-4593-4/17/07…$15.00
https://doi.org/10.1145/3087556.3087587

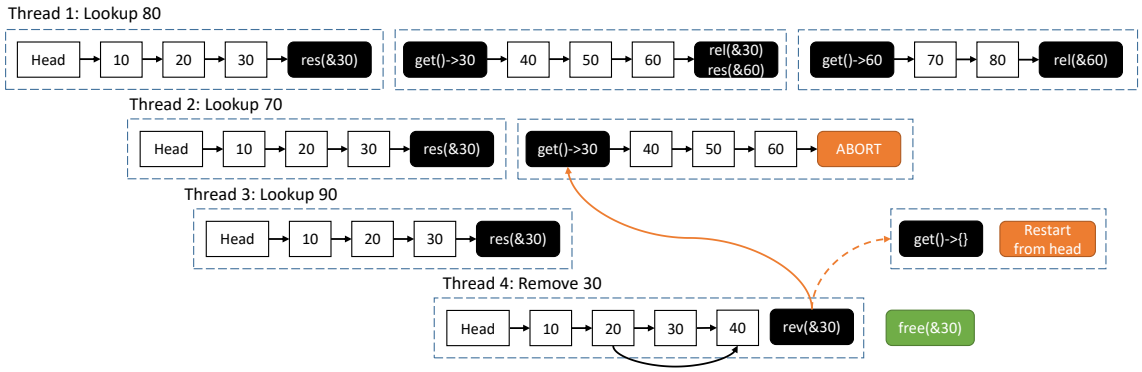


Figure 1: Concurrent operations on a linked list, with hand-over-hand transactions and revocable reservations. Time advances to the right. Dashed boxes indicate transactions, and filled black rectangles represent operations on the revocable reservation shared object ("res", "get", "rel", and "rev" correspond to reserving a value, getting a previously reserved value, releasing a reservation, and revoking all reservations for a specific value).

of some locations until all possible concurrent reads complete. It is difficult to bound the time between logical removal and physical reclamation [11], and many scalable techniques accept unbounded worst-case delay for a bounded [23] or unbounded [9] number of items. To avoid these delays, a system might fall back to complex or expensive measures when the amount of unreclaimed memory becomes too great [3, 6–8]. However, there will always remain programs whose correctness depends on memory being reclaimed immediately, hence the need for *precise* memory reclamation.

To avoid this problem, we introduce *revocable reservations*, which allow threads to revoke all reservations to a specified location. A subsequent transaction that checks a reservation will see that it has been revoked and therefore not attempt to access the formerly reserved location. By leveraging features of HTM, particularly the immediacy of aborts, concurrent operations are able to revoke these reservations and immediately reclaim memory, without compromising correctness.

To see how hand-over-hand transactions work with revocable reservations, consider the execution shown in Figure 1, in which four threads perform operations on a linked-list based set using hand-over-hand transactions. Threads T_1, T_2 and T_3 invoke *Lookup* with values 80, 70 and 90 respectively. Each of these threads executes an initial transaction that traverses the first four nodes in the list (including the head), and then commits that transaction, reserving the node N with value 30. Then T_1 and T_2 start new transactions to continue traversing the list starting from the reserved node N, and T_1 commits its transaction, releasing N and reserving a new node (with value 60). Concurrently, thread T_4 invokes *Remove*(30), finding the relevant node (N) with its first transaction. Because it wants to remove and free N, it calls *Revoke*(N) before committing, which revokes the reservations of T_2 and T_3. (By this time, T_1 has already released N and reserved a different node, so it is not affected and can complete its operation.) This revocation conflicts with T_2's use of its reservation at the beginning of its second transaction, so this transaction must be aborted. T_3 begins its second traversal transaction after T_4 commits, so when T_3 checks its reservation, it finds that its reservation has been revoked, and so retries its operation from the beginning (i.e., begins traversing from the head). Note that had T_4 removed 20 or 40 rather than 30, for example, the reservations of T_2 and T_3 would not have been revoked, so they could have continued with their operations unaffected.

We present several approaches to implementing revocable reservations, which differ in both asymptotic overhead and likelihood of conflict. We implemented and evaluated these variants in the context of singly and doubly linked lists, and unbalanced binary search trees. We found performance to be competitive with the state of the art in both transactional and nonblocking data structures, without sacrificing immediate memory reclamation.

2 SPECIFICATION

A *revocable reservation* is a shared object that provides four methods, *Reserve*, *Get*, *Release*, and *Revoke*. Each of these operations takes a "reference" as a parameter. The object maintains a set of references for each thread. A thread adds and removes references from its set using *Reserve* and *Release* respectively. *Revoke* removes a reference from every thread's set. *Get* checks whether a reference is in the caller's set, returning **nil** if it is not. These methods must be called from within transactions, so they always appear atomic, and we can define their behavior precisely with a sequential specification, which appears in Listing 1.

Note that conflicts among operations on a revocable reservation always involve operations with same reference, one of which must be a call to *Revoke*. In particular, concurrent calls to *Reserve*, *Get*, and *Release* never conflict with each other.

The revocable reservation implementations we describe in the next section also provide a *Register* method, which is invoked by a thread before it invokes any other operation on the revocable reservation. Although not formally part of the specification, this operation is useful for maintaining the set of threads that may use the object, and thus for whom a set of references must be maintained.

Listing 1: Sequential specification for the revocable reservation shared object

$\mathcal{T}$ = set of all threads
$\mathcal{R}$ = set of all references

states
$refs : \mathcal{T} \to \text{Set}(\mathcal{R})$; initially $refs(t) = \{\}$ for all $t \in \mathcal{T}$

procedure Reserve($r : \mathcal{R}$) for thread t
requires $r \notin refs(t)$
$refs(t) \leftarrow refs(t) \bigcup r$

procedure Release($r : \mathcal{R}$) for thread t
requires $r \in refs(t)$
$refs(t) \leftarrow refs(t) \setminus r$

function Get($r : \mathcal{R}$) for thread t
return $\begin{cases} r & \text{if } r \in refs(t), \\ \textbf{nil} & \text{if } r \notin refs(t). \end{cases}$

procedure Revoke($r : \mathcal{R}$) for thread t
$\forall_{t' \in \mathcal{T}} : refs(t') \leftarrow refs(t') \setminus r$

Listing 2: Basic revocable reservation algorithm (RR-FA): a linked list simulates a fully associative cache of reservations. All functions are called from an active transaction.

```
   Variables ("t" subscript indicates per-thread):
     LL   : List⟨Node⟨R⟩⟩
     N_t  : Node⟨R⟩

   function Register()
1    if N_t = nil then
2      N_t ← new Node(nil)
3      LL.appendNode(N_t)

   function Reserve(r : R)
4    N_t.value ← r

   function Release()
5    N_t.value ← nil

   function Get()
6    return N_t.value

   function Revoke(r : R)
7    for n ∈ LL do
8      if n.value = r then
9        n.value ← nil
```

3 IMPLEMENTATIONS

In this section, we present two families of revocable reservation implementations. The first adheres strictly to the specification in Section 2. The second provides a relaxed guarantee, inspired by STM: a *Get* may return **nil** even when the previously reserved reference was not revoked. To simplify the presentation, the algorithms in this section only support one reservation per thread. Extending the algorithms to support per-thread sets of reserved references is straightforward.

System Model. We assume a shared memory multiprocessor with coherent caches, and a TM implementation that provides a total order on transactions. The consequence of these requirements is that all writes to any single location must be ordered, and all operations performed within a transaction must appear to execute without any interleaving of transactional operations by other threads. We do not require Strong Isolation [1, 5, 27], and thus both HTM and opaque [15] STM are compatible with our algorithms. When STM is used, it must support privatization safety [21]. This is not a requirement of our algorithms, but rather a consequence of the desire for memory to be immediately reclaimed.

3.1 Strict Implementations

Our first three implementations of revocable reservations resemble fully associative, direct-mapped, and set-associative caches. These implementations strictly adhere to the specification in Section 2. There is significant overlap among the three implementations; to save space, we only present pseudocode for the first in Listing 2.

Fully Associative Reservations. RR-FA resembles a fully associative cache: through the *Register* method, threads "own" nodes in a linked list, and each thread t leaves its node N_t in the list from the time it first performs an operation on an RR-FA-protected list until the point where t terminates. To reserve reference r, t stores r in N_t; to release, t stores **nil** in N_t. To get its reservation, t reads from N_t. To revoke reservations on reference r, a thread traverses the list, and whenever it finds that a node stores r, it writes **nil** to that node.

To support multiple reservations per thread, we would replace the *value* field with a set. Then *Reserve* would append to the set, *Release* would remove an element from the set, and *Get* would test the set for membership. *Revoke* would remove from each thread's set, potentially increasing asymptotic complexity. All methods of the revocable reservation are performed within a transaction, which simplifies coordination of threads' accesses to these sets.

The complexity of *Revoke* is linear in the thread count. *Revoke* is also prone to low-level conflicts: From the time a revoking thread t accesses some node N_i until t's revoking transaction commits, any release or reserve by t_i will cause t's transaction to abort. On the other hand, *Reserve*, *Release*, and *Get* have $O(1)$ complexity with low constant overhead. As long as each thread's node is in a separate cache line, these methods should not experience false transaction conflicts.

Direct Mapped Reservations. RR-DM (direct mapped) replaces *LL* with an array of unsorted, doubly linked lists, and uses a hash function to assign references to these lists. Each thread is still assigned a single node, which can be present in at most one list at any time. To revoke reservations on a reference r, a thread traverses through the list in the array position to which that reference hashes, and in any node where it observes r, it writes **nil**. To reserve a node, a thread sets the value in its node, and then inserts its node into the appropriate list. To release a reservation, the thread must set its node's value to **nil**, and should remove its node from the list. As a contention-avoiding optimization, in RR-DM, a thread can delay removing the node from its list until a subsequent transaction. The *Get* method is unchanged from RR-FA.

RR-DM reduces the common-case overhead of *Revoke*, but the worst case is unchanged. The asymptotic complexity of *Reserve* and *Release* is unchanged, but the constants are higher: each now inserts or removes from a doubly linked list. More significantly, simultaneous calls to *Reserve* and *Release* from different threads can now result in transaction conflicts on one of the reservation object's lists. To reduce contention, each list begins with a sentinel node.

Set Associative Reservations. RR-SA (set associative) replaces the single array of doubly linked lists in RR-DM with A arrays. Each thread is assigned to an array via a mapping function. In *Reserve*

and *Release*, the calling thread chooses the appropriate array, and then operates as in RR-DM. *Get* is unchanged from RR-DM and RR-FA. However, *Revoke* must now traverse a list in each of the A arrays, resulting in $O(A + T)$ complexity, where T is the number of threads.

RR-SA does not change the complexity of *Reserve* or *Release*, nor does it reduce the constants in these methods relative to RR-DM. However, it does reduce the likelihood that concurrent invocations of *Reserve* and *Release* will result in transaction conflicts, since these methods are unlikely to access the same lists. The cost of this improvement is additional overhead in *Revoke*, which now must traverse A lists (though the total number of entries in those lists cannot exceed the number of threads).

Correctness. In the absence of concurrent calls to *Revoke*, the correctness of the cache-inspired implementations is straightforward: a thread writes a reference into a thread-specific location to reserve it, reads from that location to get it, and clears that location to release it. Thus the overall correctness of the algorithm reduces to ensuring that revoking r prevents subsequent calls to *Get* from returning r. In each algorithm, revocation traverses every possible location where a node might store r, and whenever that value is found, it is replaced with **nil**. Since every method is called from a transaction, there are no concurrent interleavings to complicate the argument.

3.2 Relaxed Implementations

In our relaxed implementations, after t_i reserves reference r_i, its subsequent calls to *Get* may return **nil** on account of another thread calling *Revoke*(r_j) or *Reserve*(r_j), for $r_j \neq r_i$. In exchange for this relaxation, these algorithms have smaller constant and asymptotic overhead, and less likelihood of transaction contention.

Exclusive Ownership. RR-XO is inspired by the idea of ownership in STM, and is presented in Listing 3. Again, all functions are called from a transactional context, but now a hash function provides a many-to-one mapping from memory locations to positions in an array (*OWN*) of thread identifiers. Every thread is assigned a unique identifier through *Register*, starting with the value 0. The value −1 has special meaning. To reserve a reference r, a thread writes r to its thread-local variable R_t, and writes its unique identifier ID_t into the array at the position determined by *hash*(p). *Release* is a local operation, which sets R_t to **nil** but does not update the shared array of identifiers. To revoke reference r, a thread writes −1 in the shared array in the position determined by *hash*(p). To perform a *Get* of reference r, thread t must check that ID_t is still in the table at the expected position. If so, the value in R_t is returned; otherwise **nil** is returned. To support multiple reservations per thread, R_t can be replaced with a set. Since R_t is only accessed by thread t, this does not introduce new concurrency challenges.

In RR-XO, all methods run in constant time. *Release* only accesses thread-local data, and can never cause transactions to conflict. Unlike the strict algorithms, *Reserve* must write to shared memory, and two threads cannot reserve the same reference simultaneously. *Get* retains its constant-time complexity, but it must read from shared memory to determine if R_t remains valid. In exchange for these increases in overhead, *Revoke* reduces to a single constant-time write.

Listing 3: Exclusive Owner Revocable Reservation algorithm (RR-XO): an array of thread IDs is used to indicate which thread currently holds a reservation for all references that hash to any given array entry.

```
Variables ("t" subscript indicates per-thread):
  OWN   : ℕ[]
  ID    : ℕ      // initially 0
  ID_t  : ℕ      // initially -1
  R_t   : ℛ      // initially nil

function Register()
1   if ID_t = −1 then
2       ID_t ← ID
3       ID ← ID + 1

function Reserve(r : ℛ)
4   R_t ← r
5   OWN[hash(r)] ← ID_t

function Release()
6   R_t ← nil

function Get()
7   if OWN[hash(R_t)] = ID_t then
8       return R_t
9   else
10      return nil

function Revoke(r : ℛ)
11  OWN[hash(r)] ← −1
```

Shared Ownership. RR-SO extends RR-XO similarly to how RR-SA extends RR-DM: it introduces multiple arrays of thread identifiers. Each thread is assigned to a specific array, and operates identically to RR-XO for the *Get*, *Reserve*, and *Release* methods. To revoke, a thread must write −1 to the appropriate position in each of the arrays of thread identifiers. We refer to this "shared (read) ownership" variant as RR-SO.

With A ownership arrays, RR-SO increases the complexity of *Revoke* to $O(A)$. It does not change the complexity of the other operations. The main benefit is that threads will rarely cause transaction conflicts when they reserve the same reference, since they are likely to be assigned to different ownership arrays.

Versioned Reservations. RR-V (Listing 4) uses versioning to share reservations without increasing the overhead of *Revoke*. The *OWN* array is replaced with a version array (V), which stores counters. These counters function like ownership records [10] in STM. To reserve reference r, t writes r to a thread-local field R_t, and then records the counter associated with that reference in thread-local V_t. The *Get* method checks that the value in the counter array is still V_t. To release, the thread writes **nil** to R_t, and to revoke reference r, the counter associated with r is incremented.

In RR-V, all operations have constant overhead. *Reserve* no longer writes to shared memory, and thus should not cause transaction conflicts. The use of version numbers allows any number of threads to simultaneously reserve the same reference, unlike the limits of 1 and A in RR-XO and RR-SO, respectively. *Revoke* is still $O(1)$, but must read and write to shared memory, instead of writing a constant.

Correctness. Due to the use of hash functions, a revoke of r_1 could invalidate a reservation of r_2 if r_1 and r_2 hash to the same

Listing 4: Version-based Revocable Reservation algorithm (RR-V): an array of integers is used to coordinate revocation and reservation operations.

```
   Variables ("t" subscript indicates per-thread):
     V     : ℕ[]
     ID    : ℕ      // initially 0
     ID_t  : ℕ      // initially -1
     R_t   : ℛ      // initially nil
     V_t   : ℕ

   function Register()
1    if ID_t = −1 then
2      ID_t ← ID
3      ID ← ID + 1

   function Reserve(r : ℛ)
4    R_t ← r
5    V_t ← V[hash(r)]

   function Release()
6    R_t ← nil

   function Get()
7    if V[hash(R_t)] = V_t then
8      return R_t
9    else
10     return nil

   function Revoke(r : ℛ)
11   V[hash(r)] ← V[hash(r)] + 1
```

position in *OWN* or *V*. Still, we can reason about correctness in the absence of hash conflicts. In the absence of concurrency, the correctness of the three implementations follows immediately from the implementation. As in the previous subsection, the correctness of a call to *Get* in the face of a concurrent call to *Revoke* requires discussion. Calls to *Revoke* do not overwrite t's reference in R_t. However, since every call to *Get* that returns reference r accesses the metadata (in *OWN* or *V*) associated with r, and revoking r writes that metadata, the TM implementation will ensure that a conflict manifests, and a revoked r will not be used. Furthermore, RR-XO and RR-SO limit the ability of threads to concurrently hold a reservation on a reference. This limitation affects progress, but not correctness: if t_i has reserves reference r, and then thread t_j also reserves r, such that ID_i is no longer in *OWN*, t_i will mistake t_j's call to *Reserve* for a call to *Revoke*, but will not return an incorrect value.

4 USING REVOCABLE RESERVATIONS

In this section, we briefly sketch three concurrent data structures that use revocable reservations: a singly linked list, a doubly linked list, and an unbalanced binary search tree.

4.1 Singly Linked List

Listing 5 presents a concurrent singly linked list implementation that uses revocable reservations. The behavior of this code corresponds to the illustration in Figure 1. We represent the common functionality of the *Insert*, *Lookup*, and *Remove* functions as the *Apply* function (lines 1–18). *Apply* takes a search key and two functions, which are run when the key is found, or is not found, respectively. Lines 19–39 provide implementations of the two functions suitable for *Lookup*, *Insert*, and *Remove* operations.

Initially, the thread does not have a reservation, and the *Get* on line 3 returns **nil**. In this case, the thread will start from the head

Listing 5: Singly linked list with revocable reservations. Nodes consist of a value and a next pointer, and the head initially points to a sentinel node.

```
   Variables:
     RR    : RevocableReservation
     head  : Node⟨T⟩          // Head of the list
     W     : ℕ                // Nodes to visit per transaction

   function Apply(key : T, λ_found, λ_notfound)
1    while true do
2      transaction
         // Initialize
3        (prev, i) ← (RR.Get(), 0)
4        if prev = nil then
5          (prev, i) ← (head, scatter(W))
         // Traverse
6        curr ← prev.next
7        while curr ≠ nil ∧ curr.val < key ∧ i < W do
8          (prev, curr, i) ← (curr, curr.next, i + 1)
         // Match
9        if curr ≠ nil ∧ curr.val = key then
10         res ← λ_found(prev, curr)
11         RR.Release()
12         return res
         // No Match
13       if curr = nil ∨ curr.val > key then
14         res ← λ_notfound(prev, curr)
15         RR.Release()
16         return res
         // Next Window
17       RR.Release()
18       RR.Reserve(curr)

   function Lookup(key : T)
19   λ_found ← function (prev : Node, curr : Node)
20     return true
21   λ_notfound ← function (prev : Node, curr : Node)
22     return false
23   return Apply(key, λ_found, λ_notfound)

   function Insert(key : T)
24   λ_found ← function (prev : Node, curr : Node)
25     return false
26   λ_notfound ← function (prev : Node, curr : Node)
27     n ← new Node(key)
28     n.next ← curr
29     prev.next ← n
30     return true
31   return Apply(key, λ_found, λ_notfound)

   function Remove(key : T)
32   λ_found ← function (prev : Node, curr : Node)
33     prev.next ← curr.next
34     RR.Revoke(curr)
35     delete(curr)
36     return true
37   λ_notfound ← function (prev : Node, curr : Node)
38     return false
39   return Apply(key, λ_found, λ_notfound)
```

of the list, and will begin traversing forward. Typically, a thread will access a maximum of W nodes per iteration of the `while` loop. To reduce contention on revocable reservation metadata, it may be desirable to have threads shorten their initial traversal; this is achieved by the `scatter` function, which ensures that each thread's first traversal in *Apply* will be of some number between 1 and W nodes (subsequent transactions will traverse up to W nodes). During traversal, the counter i tracks the remaining nodes before a transaction must commit and start a new window. This provides the hand-over-hand behavior we desire, ensuring that traversals

that progress far into the list do not conflict with modifications to nodes at the beginning of the list.

When there are no concurrent *Remove* operations, a thread will reserve its current position on line 18, commit its transaction, and then immediately get that position on line 3. If a concurrent *Remove* invalidates the traversing thread's start position (via *Revoke*), then the traversal either (a) aborts, and then discovers that its reservation is now **nil**, or (b) is between transactions, and will discover that its reservation has become **nil**. In either case, the thread will restart its search from the head of the list (line 5).

4.2 Doubly Linked List

Due to space constraints, we do not present full pseudocode for the doubly linked list algorithm; it is not significantly different from the singly linked list. We add a "previous" pointer to each node in the list, and during insertions and removals, we must set both the next and previous pointers; since updates are performed transactionally, the code to achieve this behavior is identical to a sequential doubly linked list. The only substantiative change relates to the removal code. In the singly linked list, the previous and current nodes are needed when unlinking a node, but in the doubly linked list, the current node suffices: its predecessor and successor are both reachable from it. This affords an optimization: rather than perform the unlinking and revoking operations from within `Apply`, a *Remove* that finds a node with matching key can reserve the node, commit the transaction, and then use a new transaction to perform both the unlinking step and the *Revoke*. If this transaction discovers that its reservation has been invalided, then it must mean that a concurrent transaction removed the same node, in which case the operation can return **false**: it can appear to happen immediately after the concurrent *Remove* operation.

The appeal of this optimization is that it avoids costly calls to *Revoke* from within traversing transactions. However, in our relaxed implementations, it is not correct: If *Get* returns **nil**, it may mean that an unrelated *Revoke* (RR-V) or even a concurrent *Reserve* (RR-XO and RR-SO) has incorrectly invalided the reservation. For these algorithms, if the final call to *Get* in a *Remove* operation returns **nil**, we must retry the entire operation.

4.3 Unbalanced Binary Search Tree

Lastly, we discuss an unbalanced binary search tree that uses revocable reservations. We focus on an internal tree, and again employ a sentinel node at the root, to simplify the case where the first element reached in a traversal is the target of a removal operation. Our implementation does not save parent pointers in tree nodes, but each node does store whether it is the left or right child of its parent.

Since the tree is not balanced, the *Lookup* and *Insert* operations are nearly identical to corresponding singly linked list code: a *Lookup* traverses until it finds its target node, and an *Insert* traverses until it either finds the value it is trying to insert, or it reaches a **nil** node, at which point it adds its value. Neither of these operations needs to revoke reservations, and both need only one reservation during their hand-over-hand traversals.

The complexity of the algorithm is in the *Remove* operation. If the value to be removed is in a leaf node, or if it is in a node that only has one child, then it can be removed in the same manner as in a singly linked list. With regard to the reservation mechanism, these code paths need only revoke reservations on one node: the node to be removed. Revoking the node to remove is necessary, since another thread's search may have reserved the node that is being deleted. However, we need not revoke the node's parent or its child (if any): If a concurrent *Apply* reserved the parent, then the removal of the node cannot invalidate the *Apply*'s most recent traversal, and the removal will be detected when the operation starts its next transaction. Similarly, if the concurrent *Apply* reserved the child, then it must not be searching for the removed value, and the removal cannot have affected the success of the next transaction issued by the operation, because the subtree rooted at the child has not changed.

When the value to be removed is in a node with two children, we must find a node to swap into its place. We choose the leftmost descendant of the node's right child for the swap. If that node is a leaf, it is removed from the tree by writing **nil** to its parent's left child. Otherwise, it is removed from the tree by promoting its right child to its parent's left child.

When we remove a value stored in a node with two children, we do not extract the node holding the value from the tree. Instead, we overwrite its value and then extract the leftmost descendant node of the right child from the tree. Clearly, the node that is extracted must be revoked. However, the fact that its value moves upward in the tree means that concurrent operations that have performed a *Reserve* anywhere on the path from $curr$ to $leftmost$ may be invalid. Let v be the value to delete, and l be the value of the leftmost descendant of the right child. l will be written into v's node using a transaction, so there is never a time where it could appear that l is not in the tree. However, suppose a concurrent thread t_C is performing an operation with a value of l. If t_C has traversed to a point between the node holding v and the node holding l, and has reserved that node, then when it begins its next transaction, it will continue searching from a point below the point where l has been moved, and thus it will incorrectly return that l is not in the tree.

A sufficient condition to remedy this situation is for the thread removing v to perform a *Revoke* on every node in the path from v to l. The revocation causes concurrent threads on that path to restart their traversal from the root of the tree, thereby ensuring that they do not resume from a point that has become invalid.

5 EVALUATION

To evaluate the performance of revocable reservations, we conducted a series of stress-test microbenchmarks. Our evaluation platform is a 4-core/8-thread Intel Core i7-4770 CPU running at 3.40GHz. This CPU supports Intel's TSX extensions for HTM [19]. It has 8 GB of RAM and runs a Linux 4.3 kernel. We used the TM support in the GCC 5.3.1 compiler. Results are the average of 5 trials. Across all experiments, variance was below 3%.

We evaluate four data structures: singly and doubly linked lists and internal and external unbalanced binary search trees. We consider the six revocable reservation implementations, and an implementation in which every data structure operation is performed within a single hardware transaction. GCC's language-level support for HTM falls back to a serial mode after hardware transactions

fail twice. For the lists, this policy is adequate, but for the trees, we changed the number to 8, as it improved the performance of all implementations.

In the singly linked list experiments, we compare against a lock-free list based on the work of Harris [16] and Michael [22]. We provide two versions of the list: one that never reclaims memory, and one that uses hazard pointers [23]. The former approximates the best-case performance of an epoch-based allocator or garbage collector, but has no bounds on memory overheads. The latter is significantly more expensive, but can bound memory consumption more tightly. Our implementations adhere to the C++11 standard. With hazard pointers, performance is best when threads only reclaim after 64 deletions, so we report that result.

Our list experiments also include a transactional hazard pointer implementation. Algorithmically, operations are identical to those in Listing 5, except that memory reclamation is deferred via hazard pointers, instead of being performed immediately. We also provide a reference-counted version of this implementation. These algorithms most closely resemble work by Liu et al. [20], and benefit from only accessing hazard pointers once per internal transaction.

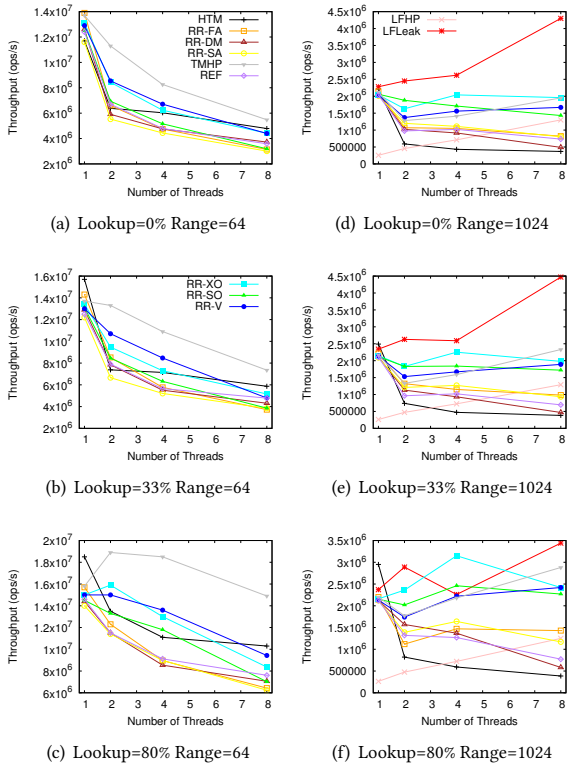
We consider both internal and external trees. This affords an opportunity to compare against an existing lock-free unbalanced search tree [25], taken from SynchroBench [14]. Note that this algorithm leaks memory.

We discovered two sources of sensitivity when running the experiments. First, there is a relationship between the number of threads and the optimal transaction window size (variable W in Listing 5). We determined the best window size for each thread count and data structure, and used these values. Second, the choice of memory allocator had a significant impact on scalability. This is a known issue with TM [2]. We performed each experiment with three allocators: the Linux system allocator, Hoard [4] and jemalloc [12]. In our charts, we report performance for whatever allocator was most stable: Hoard for the lists, jemalloc for the trees. We also observed significant improvements for all implementations when memory allocation and reclamation was performed outside of transactions. This suggests TM-aware allocators as a future research topic.

5.1 Linked Lists

Figure 2 presents a singly linked list benchmark. In each experiment, we pre-populate the list to a 50% filled state, and then perform 1M operations per thread. We vary the key range (6-bit or 10-bit) and operation mix (0, 33, or 80% lookups, with the remaining operations split evenly among inserts and removes). In the 6-bit experiments, we omit results for the lock-free implementations: hazard pointers (LFHP) perform the worst, and the lock-free list without any memory reclamation (LFLeak) performs best, but neither scales.

With small key ranges, transactions cannot scale: any list modification operation is likely to access a location that has recently been read by a concurrent transaction. However, hand-over-hand transactions exceed the baseline single-transaction implementation (HTM) in most cases, especially when lookups do not dominate. The experiments also show that when transactions are small, the cost of *Revoke* is important: the implementations with $O(1)$ *Revoke* (RR-XO, RR-SO, and RR-V) perform significantly better than those with $O(T)$ overhead (RR-FA, RR-DM, RR-SA).



(a) Lookup=0% Range=64 (d) Lookup=0% Range=1024

(b) Lookup=33% Range=64 (e) Lookup=33% Range=1024

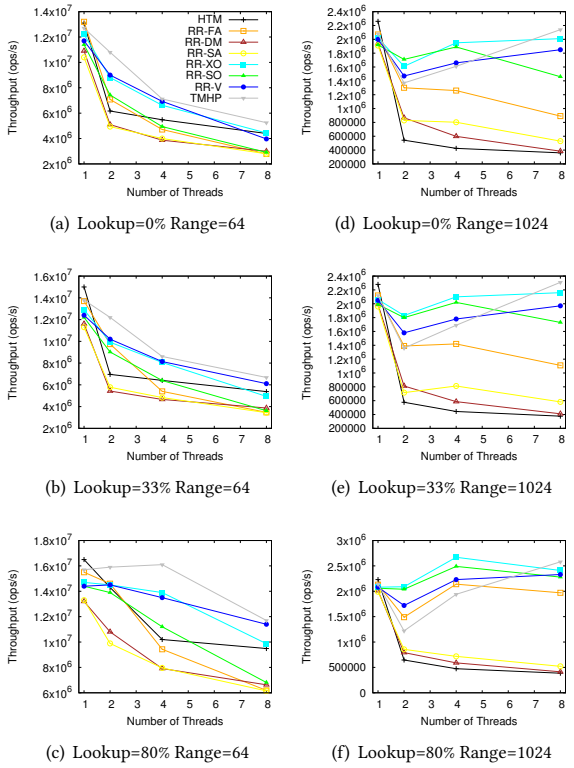
(c) Lookup=80% Range=64 (f) Lookup=80% Range=1024

Figure 2: Singly linked list microbenchmark

Even with `scatter` optimizations, transactional reference counting (REF) performs poorly. This is despite optimizations that put reference counts in separate cache lines, read only for the first and last node of each transaction. Additionally, for small lists we can quantify the cost of precise reclamation by contrasting performance with transactional hazard pointers (TMHP). By deferring reclamation and performing reclamation in batches, reclamation exhibits more locality with allocator metadata, and hence has lower latency. Furthermore, since TMHP does not write to shared reservation metadata, it scales better for small key ranges.

With 10-bit key ranges, the bottlenecks in many of our reservation-based algorithms decrease. *Revoke* represents a smaller fraction of total execution time, and the overall performance of each reservation implementation becomes a function of the propensity for *Reserve* and *Release* to cause conflicts: RR-DM performs worst, since threads must insert and remove their nodes from doubly linked lists, and RR-SA performance varies between RR-DM and RR-FA, depending on the value of A (in the charts, $A = 8$).

For high lookup ratios, the relaxed implementations perform best. They avoid the per-access overheads of the leaky list, and their costs relative to transactional hazard pointers are amortized over a longer transaction execution. Interestingly, this workload experiences the longest reclamation delays for the hazard pointer and epoch-based reclamation strategies. Revocable reservations eliminate all reclamation delay while delivering good performance.



(a) Lookup=0% Range=64 (d) Lookup=0% Range=1024

(b) Lookup=33% Range=64 (e) Lookup=33% Range=1024

(c) Lookup=80% Range=64 (f) Lookup=80% Range=1024

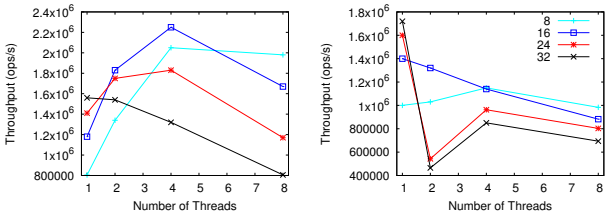
Figure 3: Doubly linked list microbenchmark

Figure 3 presents results for the doubly linked list. We no longer report reference counting, since it performs poorly, and we do not report lock-free doubly linked list performance: the only known algorithms make use of simulated multi-word compare-and-swap, and perform significantly worse than the lock-free singly linked list. The main difference between the two list algorithms is that *Remove* operations can unlink in a separate transaction from the one that finds the target node. This is beneficial for scaling, since the writing transaction is smaller. It also reduces conflicts in the reservation mechanism: if a call to *Revoke* aborts due to conflicts with a concurrent *Reserve*, the enclosing transaction retries immediately, without performing a traversal.

Overall, the doubly linked list trends are similar to the singly linked list. We observe a slightly smaller gap between the reservation mechanisms and TMHP, suggesting that the separate unlink-and-revoke transaction reduces conflicts and contention within the reservation mechanism. The remaining differences stem from TMHP's ability to batch its deferred reclamation.

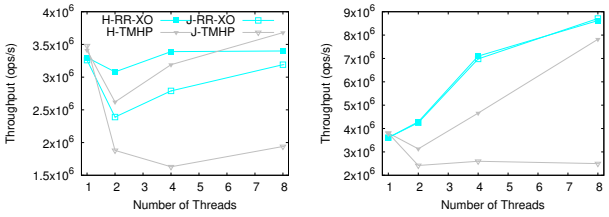
5.2 Window Size

Figure 4 shows the impact of window size on our algorithms. We highlight RR-FA and RR-XO, which are representative of the strict and relaxed techniques. The experiments use 10-bit keys and a 33% lookup ratio. On the one hand, smaller windows are less likely to



(a) RR-XO (b) RR-FA

Figure 4: Impact of window size



(a) Lookup=0% Range=512 (b) Lookup=98% Range=512

Figure 5: Impact of allocator

result in transactional conflicts. On the other, smaller windows increase latency, since there are overheads at each transaction boundary. In addition, for RR-XO, scattering the initial window size is an important optimization, since threads will otherwise conflict when reserving nodes.

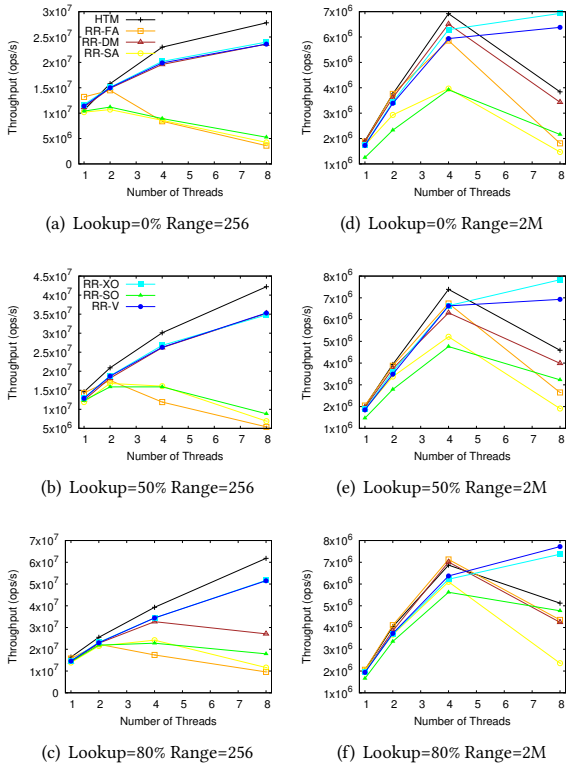
At one thread, there are no conflicts, and all transactions fit within the hardware capacity, even with a window size of 32. The advantage of a large window diminishes rapidly, especially in RR-FA, where revocation is more likely to cause false conflicts with calls to *Reserve* and *Release*. Since the likelihood of conflicts increases with the thread count, higher counts favor smaller windows. In addition, at 8 threads, each hardware transaction's capacity is effectively halved, since our CPU has four two-way threaded cores. Up to 4 threads, a window size of 16 is best. At 8 threads, the balance tips in favor of a window size of 8.

This experiment suggests future work in dynamic tuning of the window size. Doing so will entail hand-crafting the transactions, instead of using GCC TM support: GCC TM does not expose the fact of an abort, or its cause, to the programmer. Without that information, it is not possible to implement a data structure-specific tuning mechanism.

5.3 The Impact of Allocator Algorithm

To illustrate the impact of the memory allocator on performance, Figure 5 contrasts the performance of transactional hazard pointers (TMHP) with the RR-XO algorithm for a doubly linked list. We conduct two experiments, with 0% and 98% lookup ratios, on a list holding 9-bit keys. Curves prefixed with "J-" use jemalloc, and curves prefixed with "H-" use Hoard.

This case is the most extreme that we observed in all of our experiments: TMHP exposes a pathological behavior in jemalloc,



(a) Lookup=0% Range=256 (d) Lookup=0% Range=2M

(b) Lookup=50% Range=256 (e) Lookup=50% Range=2M

(c) Lookup=80% Range=256 (f) Lookup=80% Range=2M

Figure 6: Internal binary search tree microbenchmark

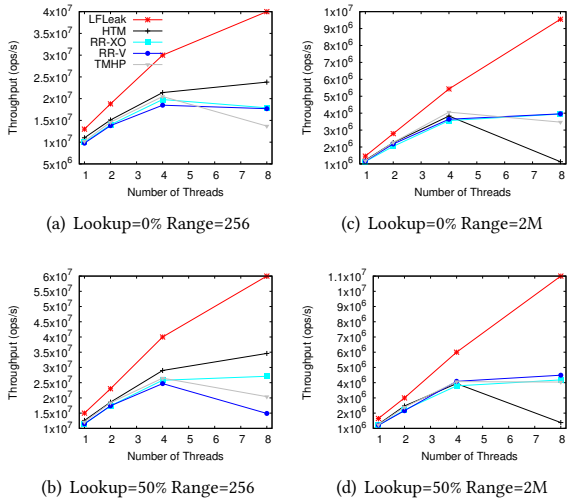
resulting in poor scalability. This is especially peculiar at the 98% lookup ratio, where memory allocation and deallocation are rare. The impact of the allocator was roughly the same for our six implementations of revocable reservations on the doubly linked list.

5.4 Unbalanced Search Trees

We now turn our attention to binary search trees. When a tree has logarithmic depth, performing operations within a single transaction should deliver good performance: even with 2M entries in the tree, the average traversal should only touch about 22 nodes, and should fit in the hardware cache. Thus the potential for reservations to improve performance is diminished. However, since the trees are not balanced, occasional large traversals are possible. Without hand-over-hand transactions, these traversals are likely to exceed cache size, and cause program-wide serialization of transactions.

Figure 6 presents an internal tree microbenchmark, which has mixed (0, 50, or 80%) lookups. We now consider 8-bit and 21-bit keys. In each experiment, the data structure is pre-populated with random keys to reach a 50% fill rate. We are not aware of internal trees that use hazard pointers, and the lock-free tree in SynchroBench is an external tree, so we can only compare our six algorithms against an algorithm where each data structure operation is a single transaction (HTM).

In the small (8-bit) key range experiment, our best implementations use a large window at low thread counts, and the entire operation fits in a single transaction. Thus differences relative to



(a) Lookup=0% Range=256 (c) Lookup=0% Range=2M

(b) Lookup=50% Range=256 (d) Lookup=50% Range=2M

Figure 7: External binary search tree microbenchmark

the HTM curve at one thread indicate the cost of reservations, and differences at higher thread counts reveal bottlenecks or contention due to the reservation mechanism. As in the list curves, we see that the relaxed implementations RR-XO and RR-V offer the best performance: all overheads are constant, and threads are unlikely to reserve the same nodes.

For the large key range experiment, there is an inflection point after 4 threads, due to hardware multithreading. The HTM algorithm exceeds the cache, and serializes. In contrast, reservation-based algorithms can use a smaller window size and complete. However, only RR-XO and RR-V scale well. The cause of poor performance in the other algorithms is the overhead of calling *Revoke* for multiple references. Recall that in the tree, a removal must revoke reservations on the path between the found node and the node whose value will be swapped. In RR-SA and RR-SO, each call to *Revoke* visits A locations ($A = 8$). In RR-DM, k *Revoke* operations result in $O(k)$ unique accesses.

Lastly, Figure 7 presents the performance of an external binary search tree that uses revocable reservations. We also include results for a nonblocking external tree [25] that leaks memory (LFLeak), and a tree that uses hand-over-hand transactions and hazard pointers (TMHP).

With no memory reclamation overheads, no overheads on transaction boundaries, and a highly optimized lock-free implementation, LFLeak performs significantly better at all thread levels, and scales linearly. The difference between LFLeak and the other algorithms makes it difficult to observe their performance, so we omitted the weaker-performing reservation algorithms. The best reservation algorithms, RR-XO and RR-V, exhibit the same relationship to HTM as in the internal tree experiments. In addition, we see that the performance of TMHP is almost indistinguishable from these algorithms. In the absence of multiple calls to *Revoke*, the other reservation algorithms performed better than in the internal tree, though still below RR-XO and RR-V.

6 CONCLUSIONS AND FUTURE WORK

This paper introduces revocable reservations, which allow concurrent data structure designers to make use of hand-over-hand transactions and still immediately reclaim memory. We presented six implementations of revocable reservations, three of which allow for spurious revocation of reserved references. We also presented concurrent list and tree data structures that employ revocable reservations and hand-over-hand transactions. We found that the relaxed algorithms performed best, often enabling hand-over-hand transactions to provide better scalability and resilience than coarse-grained transactions, with minimal cost relative to hazard pointer implementations that sacrifice immediate reclamation. However, our revocable reservations did not outperform hand-crafted lock-free data structures that do not bound memory overheads.

Our experiments showed the value of adjusting the number of locations accessed per transaction. We used the thread count as a heuristic, but contention would be a better metric. We plan to explore techniques wherein language-level transactions can reliably and safely expose abort counts and abort causes to the programmer, to enable such optimization in a standards-compliant way. We also found that our concurrent data structures were sensitive to the choice of allocator, with unpredictable and occasionally pathological behaviors. These findings suggest that there is still opportunity to improve on the state of the art in memory allocation, possibly by considering TM-aware allocation strategies and algorithms.

Overall, we found the use of revocable reservations to be straightforward, and their application to lists and unbalanced trees did not require much data structure redesign. Based on this experience, we believe they will be a valuable technique for other concurrent data structures, such as balanced trees and hash tables, for which existing scalable algorithms rely on deferred memory reclamation.

Acknowledgments

We thank Dave Dice and Tim Harris for their advice and guidance during the conduct of this research. At Lehigh University, this work was supported in part by the National Science Foundation under Grant CAREER-1253362, and through financial support from Oracle. 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.

REFERENCES

[1] Martín Abadi, Tim Harris, and Mojtaba Mehrara. 2009. Transactional Memory with Strong Atomicity Using Off-the-Shelf Memory Protection Hardware. In *Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming*. Raleigh, NC.

[2] Alexandro Baldassin, Edson Borin, and Guido Araujo. 2015. Performance Implications of Dynamic Memory Allocators on Transactional Memory Systems. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming*. San Francisco, CA.

[3] Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. 2016. Fast and Robust Memory Reclamation for Concurrent Data Structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. Asilomar State Beach, CA.

[4] Emery Berger, Kathryn McKinley, Robert Blumofe, and Paul Wilson. 2000. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*. Cambridge, MA.

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

[6] Anastasia Braginsky, Alex Kogan, and Erez Petrank. 2013. Drop the Anchor: Lightweight Memory Management for Non-Blocking Data Structures. In *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures*. Montreal, Quebec, Canada.

[7] Trevor Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There has to be a Better Way. In *Proceedings of the 34th ACM Symposium on Principles of Distributed Computing*. Portland, OR.

[8] Nachshon Cohen and Erez Petrank. 2015. Efficient Memory Management for Lock-Free Data Structures with Optimistic Access. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*. Portland, OR.

[9] Mathieu Desnoyers, Paul McKenney, Alan Stern, Michel Dagenais, and Jonathan Walpole. 2012. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems* 23, 2 (2012), 375–382.

[10] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*. Stockholm, Sweden.

[11] Aleksandar Dragojevic, Maurice Herlihy, Yossi Lev, and Mark Moir. 2011. 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.

[12] Jason Evans. 2017. jemalloc memory allocator. (2017). http://http://jemalloc.net/.

[13] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. 2017. Elastic Transactions. *J. Parallel and Distrib. Comput.* 100 (Feb. 2017), 103–127.

[14] Vincent Gramoli. 2015. 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.

[15] Rachid Guerraoui and Michal Kapalka. 2008. On the Correctness of Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*. Salt Lake City, UT.

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

[17] Maurice P. Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. 2003. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*. Boston, MA.

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

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

[20] Yujie Liu, Tingzhe Zhou, and Michael Spear. 2015. Transactional Acceleration of Concurrent Data Structures. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*. Portland, OR.

[21] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard Hudson, Bratin Saha, and Adam Welc. 2008. Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*. Munich, Germany.

[22] Maged Michael. 2002. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*. Winnipeg, Manitoba, Canada.

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

[24] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. 2015. 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*. Portland, OR.

[25] Aravind Natarajan and Neeraj Mittal. 2014. Fast concurrent lock-free binary search. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. Orlando, FL.

[26] Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*. Ottawa, ON, Canada.

[27] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Kate Moore, and Bratin Saha. 2007. Enforcing Isolation and Ordering in STM. In *Proceedings of the 2007 ACM Conference on Programming Language Design and Implementation*. San Diego, CA.