On The Power of Hardware Transactional Memory to Simplify Memory Management

Aleksandar Dragojević
EPFL, Switzerland
aleksandar.dragojevic@epfl.ch

Maurice Herlihy
Brown University
mph@cs.brown.edu

Yossi Lev
Oracle Labs
yossi.lev@oracle.com

Mark Moir
Oracle Labs
mark.moir@oracle.com

ABSTRACT

Dynamic memory management is a significant source of complexity in the design and implementation of practical concurrent data structures. We study how hardware transactional memory (HTM) can be used to simplify and streamline memory reclamation for such data structures. We propose and evaluate several new HTM-based algorithms for the "Dynamic Collect" problem that lies at the heart of many modern memory management algorithms. We demonstrate that HTM enables simpler and faster solutions, with better memory reclamation properties, than prior approaches. Despite recent theoretical arguments that HTM provides no worst-case advantages, our results support the claim that HTM can provide significantly better common-case performance, as well as reduced conceptual complexity.

Categories and Subject Descriptors

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

General Terms

Algorithms,Design,Performance

Keywords

Transactional Memory, Synchronization, Hardware, Memory Management

1. INTRODUCTION

The Java(tm) concurrency libraries [13] provide a number of lock-free data structures that have no counterparts in C++. A key obstacle to porting them to C++ is that Java is garbage-collected, while C++ requires explicit memory management, which can be very difficult. In this paper, we explore our belief that hardware transactional memory (HTM) [12] can significantly simplify the design and implementation of common concurrent data structures and algorithms, particularly with respect to dynamic memory management.

PODC'11, June 6–8, 2011, San Jose, California, USA.

We begin with some anecdotal evidence showing that, by using HTM, we can build a lock-free FIFO queue that is superior to the state-of-the-art implementation in terms of algorithmic complexity, performance, and space requirements. This dramatic example motivates us to explore whether HTM is *fundamentally* more powerful than traditional hardware synchronization primitives for building dynamic-sized concurrent data structures. We argue that the *Dynamic Collect* problem [11] is an appropriate problem to study in exploring this question, and most of this paper focuses on that problem.

In Section 2, we precisely specify the Dynamic Collect variant on which we have focused for this work. In Section 3, we outline a variety of HTM-based Dynamic Collect algorithms that explore various tradeoffs; we present one algorithm in detail in Section 4. In Section 5, we present performance results illustrating the impact of these tradeoffs on different HTM-based implementations, as well as comparing to some implementations that do not use HTM. Broadly, our results show that with HTM it is significantly easier to design correct implementations, and that non-HTM algorithms tend to perform worse, require significantly more space, or both. We discuss how various aspects of HTM designs relate to our algorithms in Section 6, and conclude in Section 7.

1.1 Lock-free FIFO queues

The Michael-Scott queue [15] is one of the best-known and most widely used lock-free data structures. The queue is represented as a linked list of entries that are allocated dynamically as values are enqueued. Any practical implementation of this algorithm must address the question of how the memory for these entries can be reused. This is challenging because the algorithm allows a queue node to be accessed by ongoing operations even after the node has been removed from the queue.

The most straightforward approach is to have each thread keep a thread-local pool of unused entries. When a thread enqueues an item, it allocates an entry from its local pool whenever possible, and when a thread dequeues a value, it returns the dequeued entry to its own pool. Using this approach, once an entry has been allocated, that memory cannot be used for any purpose other than as a queue entry. Therefore, even in a quiescent state, when no method calls are in progress, the memory used for the queue is at least proportional to the *historical* maximal queue size, which is a significant disadvantage.

An alternative is to use a technique such as the "Repeat Offender Problem" ROP [10] or Hazard Pointers [14] to enable the Michael-Scott algorithm to reclaim memory, but this entails significant additional overhead and complexity, as discussed further below.

We have implemented a concurrent FIFO queue by enclosing

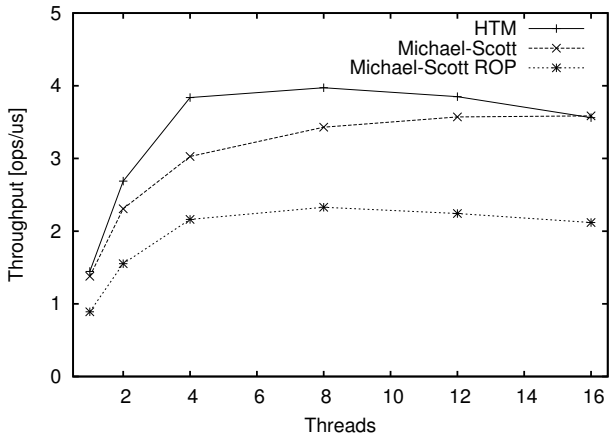

Figure 1: Queue performance

simple sequential code in hardware transactions. A successful dequeue operation frees the dequeued entry's memory to the operating system. No transaction serialized after the dequeue will see a reference to that entry, so the only danger is that a concurrent transaction may try to use it. However, if it does, it is guaranteed to abort.[1] Moreover, the HTM-based algorithm is significantly simpler than the Michael-Scott algorithm, which must deal with certain race conditions that cannot occur when operations are executed within hardware transactions, as well as with the ABA problem [15] that arises due to recycling queue nodes. To roughly quantify the difference in complexity between these algorithms, the HTM-based one would be a reasonable homework exercise for an undergraduate student, while the Michael-Scott algorithm yielded a PODC publication!

Figure 1 compares the throughput of the simple HTM-based queue and the Michael-Scott queue (with and without ROP) when a mix of enqueue and dequeue operations is performed on the queue. The experiment was performed on a 16 core Rock CPU [8]. The HTM-based algorithm can reclaim unused entries and is nonetheless up to 25% faster than the Michael-Scott queue due to its simpler code. The overheads of using ROP for reclaiming memory are significant—between 35% and 75%. This matches our intuition that HTM can enable algorithms that are better in terms of speed, simplicity *and* memory reuse.

1.2 Dynamic Collect problem

Techniques such as Hazard Pointers [14] or the "Repeat Offender Problem" (ROP) [10] can be used to enable concurrent data structures such as the Michael-Scott queue to return memory to the system when it is no longer needed. These techniques require a thread to "announce" its intention to use a reference before using it. Before a thread can free a block of memory, it must check that no other thread has announced an intention to access that block. This check amounts to performing a *collect* [2, 3, 16] over an array of announced references to ensure that the block to be freed is not potentially in use.

While Hazard Pointers and ROP can enable memory reuse, they have memory requirements of their own. In particular, each active thread requires a separate location in which to announce a pointer it intends to access. Unless these locations themselves can be reclaimed and recycled, algorithms that use these techniques inherit another form of historical space requirement: even in a quiescent state, the memory consumption of the data structure is at least proportional to its current size plus the historical maximum number of active threads. To overcome this limitation, Michael [14] and Herlihy et al. [10] propose dynamic versions of their techniques that allow locations used for announcements to be released and recycled. As a result, these memory management techniques encompass *Dynamic Collect* [11] algorithms: a thread announces its intention to dereference a pointer by using the `Register` and/or `Update` operations, and scans for other threads' announcements using the `Collect` operation. Any fundamental limitation of Dynamic Collect algorithms is inherited by any data structure that uses these approaches to manage memory. We therefore believe that studying the impact of synchronization support on Dynamic Collect algorithms provides useful insight into the inherent limitations of the ability of non-HTM systems to support dynamic concurrent data structures.

[1]We assume "sandboxed" HTM such as in Sun's prototype multicore chip code named Rock [8], in which a transaction that dereferences an illegal reference aborts, but does not otherwise disrupt the thread (say, by causing a segmentation fault).

2. DYNAMIC COLLECT

2.1 Data types and operations

A *Collect object* uses two data types, *handle* and *value*, and supports the following operations:

- $h =$ `Register`(v): binds the value v to an unused handle h, which is returned to the caller.
- `Update`(h, v): binds value v to handle h.
- `DeRegister`(h): removes the current binding to handle h.
- `Collect`$()$: returns a set of (handle,value) pairs.

2.2 Well-formedness

We say that a handle h is *registered* to a thread t when it is returned by an invocation by t of `Register`(v) for some v, and that it is *deregistered* when `DeRegister`(h) is invoked. A thread may invoke `Update` and `DeRegister` only with a handle that has previously been registered to it, and which it has not since deregistered. Any thread may invoke `Collect` at any time if it is not currently performing another operation on the dynamic collect object.

2.3 Requirements

Following standard definitions, there is a natural partial order on operations: if the invocation event of an operation *op0* occurs after the return event of another operation *op1*, then *op0 follows op1* and *op1 precedes op0*. Otherwise, the operations are said to be *concurrent*.

A call to `Register` by thread t must return a handle that is not registered to any other thread. Together with the well-formedness requirements stated above, this implies that `Register` and `Update` operations for a given handle h are totally ordered by the *precedes* relation. Thus, if any such operations precede an operation *op*, then there is a unique "last" one of them. If this operation exists and there is no `DeRegister`(h) operation that follows it and precedes *op*, then we denote it as $lastbind(h, op)$; otherwise $lastbind(h, op)$ is not defined.

Informally, a handle-value pair (h, v) may "flicker" during a $h =$ `Register`(v) or `Update`(h, v) call: a concurrent `Collect` call may or may not return it. However, if such an operation completes *before* the invocation of a `Collect` operation, and the handle is not subsequently deregistered, then the `Collect` operation

must return a value for that handle (either v or another value v' if there is a subsequent `Update`(h, v') operation). More precisely, a `Collect` operation *cop* returns a set S of handle-value pairs such that the following conditions hold, for every handle h and value v:

- If $(h, v) \in S$, then either
  - $lastbind(h, cop)$ is defined and is a $h =$ `Register`(v) or `Update`(h, v) operation, or
  - a $h =$ `Register`(v) or `Update`(h, v) call was concurrent with *cop*.
- If $lastbind(h, cop)$ is defined and there is no `DeRegister`(h) operation that is concurrent with *cop*, then $(h, v') \in S$, for some value v'.

Note that this specification is non-deterministic: there may be multiple sets of handle-value pairs that can legitimately be returned by a `Collect` call. Furthermore, it does not preclude a `Collect` operation returning multiple pairs for the same handle h. Clients can filter out duplicates if necessary by choosing any one of the pairs for each handle in the returned set.

There are many possible small variations on this specification. For example, an alternative `Register` might return a handle without binding it to a value, or `Collect` might omit the handles and simply return a multiset of values. Nonetheless, the above specification is suitable for a variety of use cases, including for use in memory management mechanisms as discussed in Section 1. To our knowledge, these possible minor variations in the specification do not have a significant impact on our findings.

3. ALGORITHMS

Our primary goal has been to explore how the availability of HTM *in general* impacts the ease of dynamic memory management. However, we wanted to be able to implement and experiment with the algorithms using real HTM-enabled hardware, namely Sun's Rock prototype. Thus, we could not ignore certain limitations [8, 9] of Rock's HTM, such as the requirement that transactions do not perform more stores than are accommodated by the store buffer.

Despite these constraints, we have found that the availability of HTM allows us to explore many algorithmic approaches, tradeoffs and optimizations. In contrast, without HTM, it is significantly more difficult to come up with *any* correct algorithm, and there is less flexibility for variants and optimizations. In this section, to illustrate the flexibility HTM enables, we give high-level descriptions of some of the algorithmic techniques we have explored; in the next section, we describe one algorithm in more detail.

3.1 List-based Algorithms

We have developed two kinds of list-based algorithms: *hand-over-hand reference counting* (`HOHRC`) and *fast collect* (`FastCollect`); each uses a doubly-linked list with one value per node.

3.1.1 `HOHRC`

This algorithm uses a per-node reference count to "pin" a node (prevent it from being deallocated) while a `Collect` is accessing its value. `Collect` traverses the list, using short transactions to increment the reference count of a node n while atomically confirming that n's predecessor still points to it. As the predecessor has previously been pinned, this ensures that node n is still part of the `Collect` object. After incrementing n's reference count, `Collect` reads n's value non-transactionally and copies it to the result set. It then unpins n's predecessor, using a transaction to decrement its reference count, and, if it becomes zero and the predecessor's "delete marker" has been set, unlinks it from of the list and deallocates it.

`Register` allocates a new node, uses a transaction to insert it at the beginning of the list, and returns its address. `Update` non-transactionally stores the new value into the node. `DeRegister` executes a short transaction to set the delete marker of the node to be deregistered. If this transaction observes that the node's reference count is zero, it unlinks the node from the list and deallocates it after the transaction commits. Otherwise, some ongoing `Collect` has pinned the node. The last `Collect` that unpins the node will unlink it and deallocate it. Note that a given node may be continually pinned and thus never reclaimed. However, each `Collect` pins at most two nodes at a time, so the shared memory used is proportional to the number of active handles plus the number of ongoing `Collect`s.

This description assumes that values stored by `Update` operations fit into a word that can be written and read by a native machine instruction, a significant advantage when `Update` operations are frequent. This advantage stems from the fact that the storage for a given handle does not move during the lifetime of the handle. The array-based algorithms described below depend on the ability to move the storage for a handle, thus requiring `Update` operations to use transactions to confirm the location of the storage.

The main disadvantage of the `HOHRC` algorithm is the expensive `Collect` operation, which updates each list node twice, increasing the cost of `Collect`, causing significant memory coherence traffic, and causing transactions used by `Collect` operations to conflict with each other. Telescoping (Section 3.4) reduces these effects, but cannot fully eliminate them.

3.1.2 `FastCollect`

This algorithm aims to improve `Collect` performance when `DeRegister` operations are infrequent. It uses the same `Register` and `Update` operations as `HOHRC`. However, it avoids `HOHRC`'s main disadvantage by dispensing with the reference counts: `DeRegister` uses a transaction to atomically unlink a node n and increment a shared deregister counter dc, and deallocates n immediately afterwards. `Collect` traverses the list using transactions to atomically read the current value of dc and the next node in the list. If dc has changed since the start of the `Collect`, the `Collect` is restarted from the beginning.

The main disadvantage of `FastCollect` is that `Collect`s can be prevented from making any progress by concurrent `DeRegister`s. A variety of practical approaches can be used to address this problem, such as adding a mode in which `DeRegister` operations add nodes to a to-be-freed list that is freed by a `Collect` operation after it completes. Again, HTM makes it straightforward to integrate such variants.

3.2 Array-based Algorithms

Our array-based algorithms can be categorized based on how they: (1) manage memory, (2) register new handles, and (3) compact (move elements inside) the array.

3.2.1 Managing memory

Our array-based algorithms are either *static* or *dynamic*. The static ones *do not solve* the Dynamic Collect problem: they assume a known bound on the number of handles to be registered, and do not attempt to deallocate unused space. We use these algorithms as a stepping stone towards truly dynamic ones, and to isolate the algorithmic and performance issues related to registering and dereg-

istering handles, collecting only from registered handles, etc., from the issues related to reclaiming unused memory.

The dynamic array-based algorithms can replace the current array with a new one of a different size, employing a level of indirection to identify the current array. We double the array when it is full, that is, when every *slot* (array entry) is in use for a registered handle; and halve it when it is 25% full. This way we avoid excessive resizing while keeping space usage proportional to the number of registered handles.

To resize the array, the algorithms allocate a new one, and install it as the "new" array. The values are then copied from the current array to the new, and the new array is made current. These steps may be performed in cooperation with other threads. While there are small differences in how our algorithms achieve this, the detailed description in the next section is representative.

3.2.2 Registering

The `Register` operation can either *search* for an empty slot or *append* a new element after the last used slot in the array (for example, using the `count` variable in the next section).

3.2.3 Compacting

To reduce or avoid fragmentation, the array can be "compacted" by moving elements within it. Our algorithms either perform *no* compaction, or do so on each *resize* or each `DeRegister`.

Compacting requires slots to be moved by threads other than their owners. This creates a race between one thread moving a slot and another performing a `Update` to it, requiring synchronization between these threads. This is explained in detail in the next section.

Algorithms that compact on resize move slots only when the array is resized. Elements are copied into consecutive locations in the new array.

Algorithms that perform compaction with every `DeRegister` operation use a transaction to move the last used slot into the space used by the slot being deregistered, atomically updating other data such as a count of the number of registered slots, and data used to associate the handle of the moved slot with the memory location in which it is stored. `Collect` operations must access the elements in the array from the last towards the first to avoid missing an element that is moved by a concurrent `DeRegister`. This may lead to multiple values being returned by the same `Collect` operation for the same handle, which is allowed by the specification, whereas missing a handle is not.

3.2.4 Algorithms

Many combinations of these design choices yield meaningful algorithms. We have implemented some of the most interesting choices, naming the algorithms according to the choices for memory management, registering, and compacting, yielding the following algorithms: `ArrayStatSearchNo`, `ArrayStatAppendDereg`, `ArrayDynSearchResize`, and `ArrayDynAppendDereg`.

3.3 Baseline algorithms

We have also implemented two non-HTM-based Collect algorithms for comparison. The Static baseline algorithm uses a fixed-sized array, with threads mapped statically to slots in it (`Register` and `DeRegister` are no-ops). `Update` operations by a thread write directly to the thread's slot, and `Collect` simply scans the entire array and returns the set of non-null values seen. Recall that such static algorithms do not solve the Dynamic Collect problem: we use them merely to put the performance of dynamic algorithms in context.

The Dynamic baseline (Algorithm 2 from [11]) uses a doubly linked list of nodes whose forward pointers are augmented with reference counts *for the pointed-to node*. `Register` searches for a free node, incrementing forward pointer counters on the way. If none is found, a new one is added to the end of the list. The address of the node that is found or added is returned as the handle. `Update` uses the handle to store the value directly into the registered node. `DeRegister` of node n decrements forward pointer counters in all nodes preceding n. If any of the counters reaches zero, the node pointed to by the associated forward pointer is unlinked and deallocated. `Collect` traverses the list, incrementing the forward pointer counters. After reaching the end of the list, it goes back in the opposite direction decreasing the counters, unlinking and deallocating nodes pointed to by forward pointers with zero reference counts.

3.4 Telescoping

The `HOHRC` algorithm can be improved by observing that the net effect of several traversal steps executed in sequence (without activity by other threads) is to increment the reference count of the last node accessed and decrement that of the first because the reference count of each of the intermediate nodes is incremented and subsequently decremented. By combining these steps into a single transaction, we not only amortize the cost of starting and committing a transaction over multiple steps down the list, but we also avoid modifying the intermediate nodes, thereby improving cache behavior. This is safe because the intermediate nodes are accessed inside a transaction that ensures not only that the first node accessed in a transaction is still in the list, but also that the pointers between this node and subsequent ones are intact. We call this technique *telescoping* and the number of nodes accessed in each transaction the *step size*.

The telescoping technique is also applicable to other algorithms. For example, in the `FastCollect` algorithm, each transaction could read dc once, and then access a number of list nodes, thereby amortizing the cost of starting and committing a transaction and of reading dc.

The best choice of step size depends on several factors. Larger step sizes allow fixed transaction costs to be amortized over more steps, but make transactions more vulnerable to abort, depending on the algorithm and limitations of the HTM. In our experiments, we were could not use step sizes greater than 32, which is the size of Rock's store buffer, because each step performs at least one store (to record a value in the result set). Because different step sizes perform best at different contention levels, we developed a simple mechanism for adapting the step size based on the abort rate. This mechanism bases its decisions on the success or failure of the most recent 8 transactions. However, in order to avoid excessive resizing, only transaction attempts since the last resize are relevant to the decision.

Our mechanism maintains a counter that records the difference between the number of commits and the number of aborts amongst the relevant transactions. The counter is maintained by using an 8-bit vector to record the results of the recent transactions, allowing us to "age out" the contribution of the oldest transaction and update the difference counter accordingly. If the value of the counter is higher than 6 after a commit, we double the step size. If it is below -2 after an abort, we halve the step size. These thresholds were determined experimentally.

4. THE `ArrayDynAppendDereg` ALGORITHM

In this section, we present the `ArrayDynAppendDereg` algorithm in more detail. Figure 2 gives pseudocode for the algorithm,

```
1  public struct slot_t {
2    val_t val;
3    slot_t **slot_ref;
4  }
5
6  # shared data
7  slot_t array[] = new slot_t[MIN_SIZE];
8  int capacity = MIN_SIZE;  // MIN_SIZE >= 1
9  int count = 0;
10 slot_t array_new[] = NULL;
11 int capacity_new;
12 int copied;
13
14 bool copying() {
15   return array_new != NULL;
16 }
17
18 public slot_t **Register(val_t val) {
19   slot_t **slot_ref = new (slot_t *);
20   action_t action = NOTHING;
21   while(action != DONE) {
22     atomic {
23       if(!copying()) {
24         if(count < capacity) {
25           append(slot_ref, val);
26           action = DONE;
27         } else {
28           count_l = count;
29           action = GROW;
30         }
31       } else {
32         if(count < capacity && count <
               capacity_new) {
33           append(slot_ref, val);
34           action = DONE;
35         } else
36           action = HELP;
37       }
38     }
39     if(action == GROW) attempt_resize(count_l,
           count_l);
40     else if(action == HELP) help_copy();
41   }
42   return slot_ref;
43 }
44
45 public void Deregister(slot_t **slot_ref) {
46   action_t action = HELP;
47   while(action != DONE) {
48     atomic {
49       count_l = count;
50       capacity_l = capacity;
51       if (count_l*4 == capacity_l && count_l*2
             >= MIN_SIZE)
52         action = SHRINK;
53       else if(!copying()) {
54         count = count_l-1;
55         **slot_ref = array[count];
56         *(array[count].slot_ref) = *slot_ref;
57         action = DONE;
58       }
59     }
60     if(action == SHRINK) {
61       attempt_resize(count_l, capacity_l);
62       action = HELP;
63     } else if(action == HELP) help_copy();
64   }
65   delete slot_ref;
66 }
67
68 void append(slot_t **slot_ref, val_t val) {
69   array[count] = slot_t(val, slot_ref);
70   *slot_ref = &(array[count]);
71   count = count + 1;
72 }
73
74 public void Update(slot_t **slot_ref, val_t
       val) {
75   atomic {
76     (*slot_ref)->val = val;
77   }
78 }
79
80 public void Collect(vector_t ret) {
81   help_copy();
82   int i = count - 1;
83   while(i >= 0) {
84     atomic {
85       if(i >= count)
86         i = count - 1;
87       if(i >= 0) {
88         ret.add(array[i].val);
89         i = i - 1;
90       }
91     }
92   }
93 }
94
95 void attempt_resize(int count_l, int
       capacity_l) {
96   slot_t array_tmp[] = new slot_t[count_l*2];
97   bool free_tmp = true;
98   atomic {
99     if(!copying() && count == count_l &&
           capacity == capacity_l) {
100      array_new = array_tmp;
101      capacity_new = count_l*2;
102      copied = 0;
103      free_tmp = false;
104    }
105  }
106  if(free_tmp) delete[] array_tmp;
107  help_copy();
108 }
109
110 void help_copy() {
111   while(copying()) help_copy_one();
112 }
113
114 void help_copy_one() {
115   slot_t array_to_free[] = NULL;
116   atomic {
117     if(copying()) {
118       if(copied < count) {
119         array_new[copied] = array[copied];
120         *(array_new[copied].slot_ref) = &
               array_new[copied];
121         copied = copied + 1;
122       } else {
123         array_to_free = array;
124         array = array_new;
125         capacity = capacity_new;
126         array_new = NULL;
127       }
128     }
129   }
130   if(array_to_free != NULL) delete[]
         array_to_free;
131 }
```

Figure 2: Pseudocode for the `ArrayDynAppendDereg` algorithm.

using C++-like notation: we use * for declaring and dereferencing pointers, -> for accessing a field of a structure through a pointer, and & for taking the address of a variable. We use `public` to denote all functions and types that are parts of the object interface.

4.1 Dynamic array and resizing mechanism

`ArrayDynAppendDereg` uses a dynamic array of "slots"; each slot can store one value that has been associated with one handle The current array and the number of slots in it are identified by `array` and `capacity`, respectively. To resize the array, a thread allocates a new array (line 96), then atomically stores a pointer to it in `array_new` and its size (in slots) in `capacity_new`, and sets `copied` to zero to indicate that no slots have yet been copied from the old array to the new (lines 100-102). The thread then calls `help_copy`, which copies slots individually from the old array to the new (lines 119–121), and finally makes the new array current and sets `array_new` back to `null` to facilitate subsequent resizing (lines 124–126). Other threads calling `help_copy` may also participate; the one that makes the new array current deallocates the old array (line 130).

The value associated with a handle can be moved, either during resizing or if it is the last value in the current array and is moved to replace a slot being deregistered. To facilitate moving of values, each handle has an associated "slot reference", which points to its associated slot, and the slot has a pointer back to this slot reference. This way, when a value is moved, the slot's pointer to the slot reference can be used to update the slot reference so that it points to the new slot (lines 56 and 120), thus allowing subsequent `Update` operations for that handle to determine its new location.

An interesting observation is that HTM makes it trivial to implement a minor variant on this algorithm that is optimized for `Update` performance at the cost of higher overhead for `Collect` operations. The idea is to store the value associated with a handle together with the slot reference for that handle, rather than in the array slot to which it points. This way, slot references do not move, even if their associated array slots are compacted. Therefore, a `Update` operation can store its value directly and without using a transaction (at least for some common cases; see Section 5.1), rather than through a level of indirection using a transaction. The downside of this choice is that `Collect` operations must now use a transaction to dereference the pointer in each array slot in order to access the associated value. Depending on anticipated workload, this may be an appropriate choice. We have not implemented this variant.

4.2 Operations

`Register` is very simple in the common case: it allocates a new slot reference (line 19) and calls `append` to store the new value and a pointer to the new slot reference in the first unused slot, updates the slot reference to point to the new slot, and increments `count` so that a subsequent `Register` operation will use the next slot (lines 69–71). The `DeRegister` operation is also straightforward in the common case: it copies the last used slot to the slot being deregistered, updates the slot reference for the moved slot to point to its new location (line 56), decrements `count` to make the last used slot available again (line 54), and frees the slot reference associated with the deregistered handle (line 65).

The more complicated cases for the `Register` and `DeRegister` operations arise because of resizing the array. First, if a new array is being installed (lines 23 and 53), these operations call `help_copy` to ensure that the new array become current before trying again. (There is one exception in `Register` that we discuss later.) Furthermore, these operations perform additional steps in order to keep the space used by the array proportional to the number of registered slots (while ensuring a non-zero minimum number of slots). Specifically, we maintain the following invariant: $max(\texttt{count}, MIN\_SIZE) \leq \texttt{capacity} \leq 4 * \texttt{count}$. `Register` checks if there is still room in the current array (line 24), and if not initiates an attempt to grow the array (line 39). Similarly, `DeRegister` checks if decrementing the number of slots used would violate the invariant, and if so initiates an attempt to shrink the array (line 61). Both procedures pass the values of `count` and `capacity` seen in the transaction to the `attempt_resize` procedure. The resize attempt is abandoned if either of these variables differs from the previously observed value (line 99), as this indicates that either there is no longer any risk of violating the above invariant, or the array has already been resized. Furthermore, if copying is already in progress, the resize attempt is abandoned, and the thread helps to complete the current resizing if necessary (lines 99 and 107).

The new array size chosen when resizing—whether shrinking or growing—is twice the value of `count`; this way, after a successful resize, `count` is in the middle of the range of values that satisfy the above invariant, so further resizing will occur only when the number of registered handles either halves or doubles.

Interestingly, a `Register` operation can complete even while resizing is in progress, provided there is enough space for the newly registered element in both the old and new arrays (line 32). This is because the same transaction that determines that the last element has been copied (line 118) also installs the new array. Thus, if a `Register` operation succeeds in claiming a slot in the current array during resizing, it is guaranteed that the slot will be copied to the new array before the new array becomes current.

It remains to describe the `Collect` operation. It would be trivial to satisfy the requirements of the `Collect` by reading values from all registered slots in one hardware transaction. However, this is not practical: existing HTM implementations do not support transactions of unbounded size, and even if they did, we would be attempting to read many locations in a transaction that would conflict with any concurrent `Update` operation, causing excessive aborts. Therefore, the algorithm presented in Figure 2 reads only a single slot in each hardware transaction. As discussed in Section 3.4, the `Collect` could copy more than one slot per transaction to reduce the overheads of starting and committing the transactions. That is, lines 87–90 can be executed multiple times in the same transaction; we experiment with different "step sizes" (number of elements copied within each transaction) in the next section.

Even using multiple transactions, `Collect` is quite simple, but there are some subtle issues. First, for the reason explained in Section 3.2, `Collect` reads slots in reverse order.

Second, a `Collect` operation can proceed despite concurrent resizing and compacting. This is because any slot that is continually registered during the `Collect` either stays at the same index in the current array, or moves to a lower one due to a concurrent `DeRegister` operation. Thus, because `Collect` reads from the slot of the current array for each index below the value of `count` observed at the beginning of the `Collect` in *reverse* order, a slot will not be missed even if it is moved to a new array during the `Collect`.

However, there is one exception. If a resize were in progress when a `Collect` operation began, then the `Collect` could copy a value from a slot that had already been copied to the new array. An `Update` operation could have updated such a slot in the new array before the `Collect` operation began, but the `Collect` would fail to return the new value, which would be incorrect. To eliminate this possibility, a `Collect` operation begins by calling `help_copy`, which ensures that there is no copy in progress be-

fore it returns. While Collect may still miss Updates performed during *subsequent* resizes, these Updates are concurrent with the Collect operation, and therefore the specification allows them to be missed.

Finally, we note that Collect checks that the index from which it is about to read is still valid (lines 85), "advancing" the index down to count-1 if not, to avoid reading deregistered slots.

4.3 Impact of HTM on algorithm complexity

Transactions make it easy to maintain simple invariants. For example, capacity always contains the size of array, and similarly for capacity_new and array_new. Access-after-free errors are easily avoided as we only deallocate arrays that are not referenced by either array or array_new, and accesses to slots in arrays are always performed inside transactions that confirm that the array is identified by either of these variables. Similarly, ensuring that a handle's slot reference always points to its slot makes it easy to move slot data without the risk of an Update operation accessing the old location. Without hardware transactions, these simple relationships cannot be maintained, as variables involved in them must be updated individually, which significantly complicates the algorithm.

We included the optimization of allowing a Register operation to complete despite ongoing resizing in order to illustrate the power of HTM to facilitate such changes. In our experience, nonblocking algorithms designed using only traditional hardware support for synchronization are delicate and inflexible, making such optimizations infeasible or too complicated. For the sake of simplicity, we have foregone a number of other optimizations that would be similarly straightforward.

5. EXPERIMENTAL RESULTS

In this section, we present results of experiments performed on a 16-core Rock system [8]. We used the libumem malloc implementation. Each graph point represents the average of 10 runs. Unless stated otherwise, we show the results for the best telescoping step size and indicate it in the graphs. We used several microbenchmarks to evaluate different aspects of the algorithms.

5.1 Update latency

We measured the latency of Update operations at about 215ns for the ArrayStatAppendDereg, ArrayDynSearchResize, and ArrayDynAppendDereg algorithms, and about 135ns for the remaining algorithms. This is explained by the fact that the remaining algorithms all perform Update operations directly to an address determined by the handle, whereas the algorithms named above all require a level of indirection through the handle to determine the address to write.

Although this seems like a significant difference, we believe that in many workloads of interest, Update operations will account for a small fraction of application runtime. Furthermore, the ability of some of the algorithms to perform Update operations using naked store instructions depends on the values being stored fitting within a single machine word, as in our experiments. For larger values, synchronization (HTM-based or not) would be needed to prevent Collect from returning partial values, which would largely close the gap in Update performance. Also, as discussed later, it is straightforward to reduce the frequency of DeRegister and Register operations in workloads in which they are invoked frequently enough to dominate performance. Therefore, the rest of our evaluation concentrates on Collect performance.

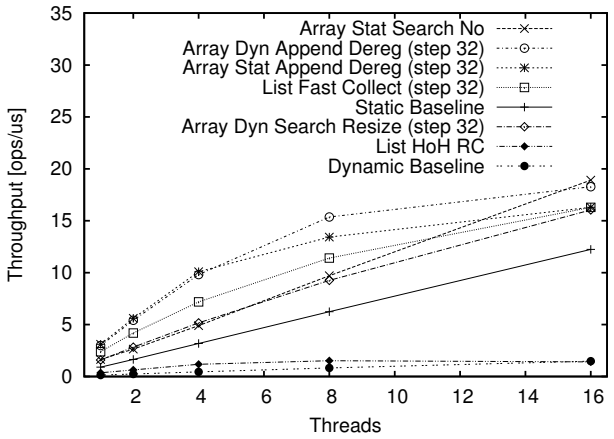

Figure 3: Collect-dominated

5.2 Collect-dominated benchmark

In this benchmark, threads randomly perform operations, with the following distribution: Collect 90%, Update 8%, Register 1%, DeRegister 1%. Each thread t maintains a queue of at most n_t slots, where the n's are chosen to spread a total of 64 evenly amongst the threads used. Before measurement begins, the threads register a total of 32 slots, spread evenly between them.

A thread ignores Register operations when its queue is full and ignores DeRegister and Update operations when the queue is empty. Otherwise, for a Register operation, a thread registers a new slot and adds it to its queue; for a DeRegister operation, it removes a slot from its queue and deregisters it; and for an Update operation, the thread stores to the least recently used slot in the thread's queue.

As shown in Figure 3, the Dynamic baseline and HOHRC performed significantly worse than all other algorithms due to poor cache performance caused by modifying each node in the list while traversing it. These two algorithms were similarly outperformed by large margins in all experiments involving Collect operations and are therefore omitted from the rest of our results to allow easier comparison of the remaining algorithms.

ArrayDynAppendDereg and ArrayStatAppendDereg perform best up to 8 threads. They outperform even the Static baseline because its Collect traverses the entire array, which is on average only half full, whereas the Append algorithms scan only registered slots.

With higher thread counts, the Collect transactions restart more often due to higher contention, and the Append-Dereg algorithms become slower than ArrayStatSearchNo with 16 threads (recall that this algorithm does not solve the Dynamic Collect problem). Even so, both algorithms are consistently among the best. The two Append-Dereg algorithms have roughly the same performance up to 4 threads, but diverge slightly thereafter. Upon further investigation, we determined that this difference is caused by idiosyncrasies of Rock's microTLB, not algorithmic differences. Similar experiences were reported in [9].

5.3 Collect-Update benchmark

This benchmark evaluates Collect performance under contention from concurrent Updates. One thread performs Collects while 15 others execute Updates (Figure 4). Update threads perform Update operations no more often than the *update period*, which we vary in order to control contention. Before measurement

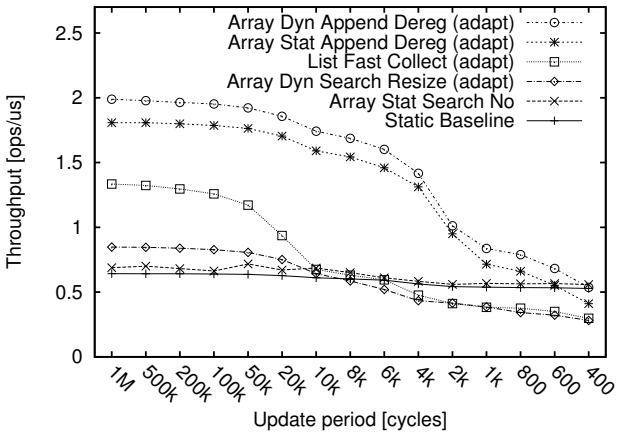

Figure 4: Collect-Update

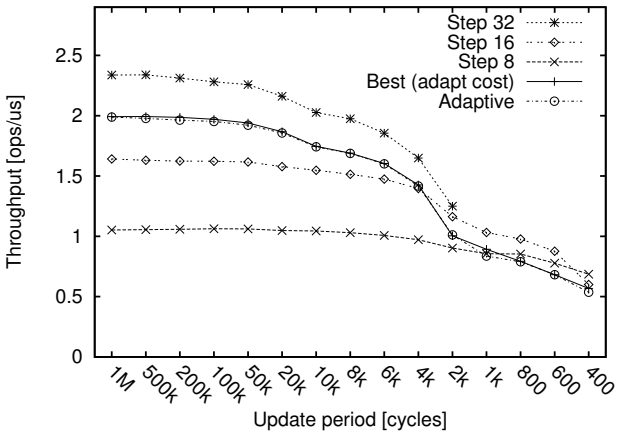

Figure 5: Adapting step size for `ArrayDynAppendDereg`

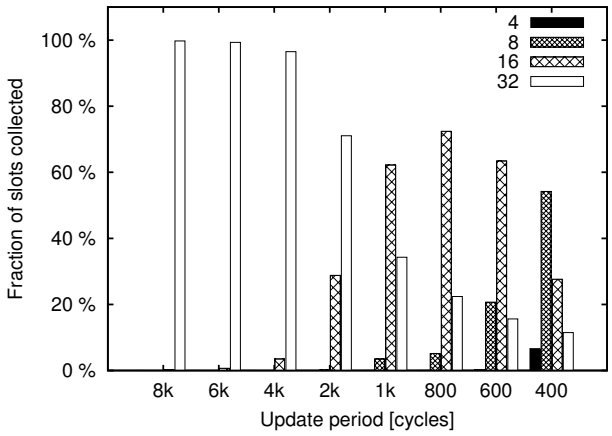

Figure 6: Step size distribution for `ArrayDynAppendDereg`

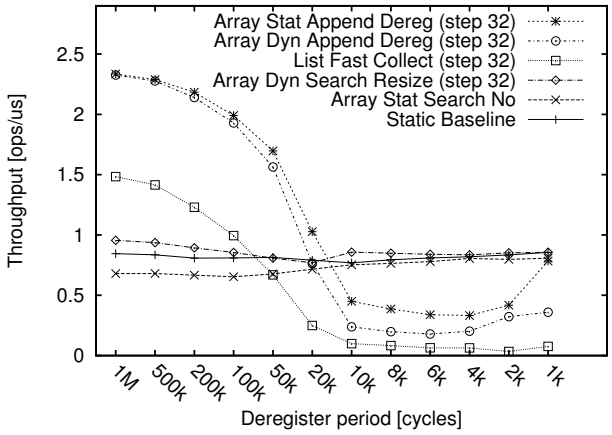

Figure 7: Collect-(De)Register

begins, the `Update` threads register a total of 64 handles. Each `Update` thread uses the same one of its handles for all operations; the rest of the handles are unused; we register them only to keep the total number of registered slots in this experiment independent of the number of threads.

The performance of Static baseline and `ArrayStatSearchNo`—whose `Collect` operations do not use transactions—are affected only slightly by more frequent `Update`s, due to an increase in cache misses. For the other algorithms, performance degrades more significantly because the `Collect` transactions abort more often with higher contention. The two Append-Dereg variants perform best for all update periods except 400 cycles. Even at this point, the Append-Dereg algorithms perform only slightly worse than Static baseline and `ArrayStatSearchNo`, which do not solve the Dynamic Collect problem, and easily outperform all algorithms that do. Thus, the Append-Dereg algorithms are the clear winners for this benchmark.

Figure 5 examines the need for and effectiveness of an adaptive step size using `ArrayDynAppendDereg`. The tradeoff discussed in Section 3.4 is apparent: larger step sizes result in lower overhead for successful transactions, but larger transactions are more likely to abort as contention increases. `Collect` operations that use step size of 32 do not complete for update periods less than 2000 cycles.

Each point on the "Best (adapt cost)" curve shows the throughput for the best-performing step size for that threading level with the *collection* (but not use) of data to enable adaptive step size. It shows overhead of between 20% and 30% for collecting this data. These overheads could be reduced or eliminated with simple hardware support to track recent transaction attempts. The adaptive algorithm performs close to "Best (adapt cost)" for all update periods, showing that it chooses the step size effectively and that most of its overhead stems from collecting the additional data. In some cases, the adaptive algorithm even performs better than "Best (adapt cost)", because it manages to commit a fraction of transactions with larger step size. Figure 6 shows the fraction of slots collected using common step sizes, and confirms that the adaptive algorithm is effective in finding the best step size to use.

5.4 Collect-(De)Register

Next, we evaluate the performance of `Collect` under contention from concurrent `Register`s and `DeRegister`s (Figure 7). One thread executes `Collect`s, while 15 others execute `Register`-`DeRegister` pairs with delays between them. We call the delay between the start of a `DeRegister` and the start of the following `Register` operation the *register period*, and the delay between the start of the `Register` and the start of the following `DeRegister` operation the *deregister period*. We fix the register period to 20,000 cycles and vary the deregister period. Initially, the total number of

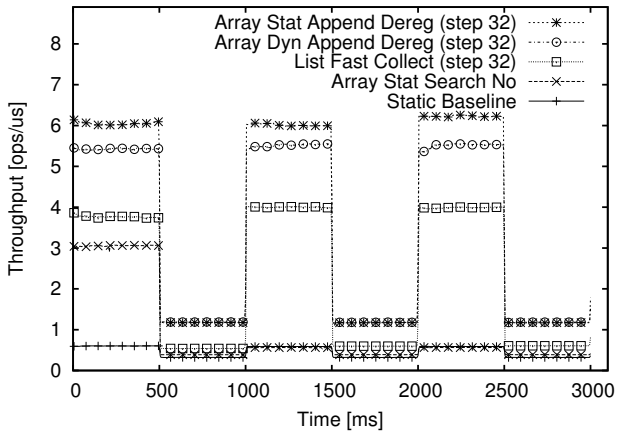

Figure 8: Collect performance with varying number of registered slots

registered slots is 64, evenly spread across the register/deregister threads. The threads start the experiment by first deregistering a slot, so the total number of registered slots is always at most 64.

With long deregister periods, algorithms in which `Collect` traverses only the registered slots perform best. The others perform worse because they either traverse all slots (Static baseline) or frequently traverse more slots than are registered due to infrequent compaction (`ArrayDynSearchResize`) or no compaction (`ArrayStatSearchNo`).

As the deregister period decreases, the algorithms that perform best with large deregister periods begin to degrade due to increased abort rates because of more frequent `Register` and `DeRegister` operations. The performance of `ArrayStatAppendDereg` and `ArrayDynAppendDereg` degrade significantly due to higher abort rates. `FastCollect` also degrades significantly, due to increased `DeRegister` frequency resulting in increased contention on the deregister count, causing `Collect` operations to start over from the beginning.

Interestingly, several of the algorithms exhibit noticeable performance *gains* at the shortest deregister periods. This is because a shorter deregister period results in fewer handles being registered at a time, so `Collect` operations are both shorter *and* less likely to conflict with a concurrent `DeRegister` operation.

For applications that perform frequent `Register` and `DeRegister` operations, it may make sense to *defer* deregistering handles, allowing them to be reused by subsequent `Register` operations. This could improve performance of many of the algorithms for such workloads, particularly `FastCollect`, because `Collect`s conflicts on the deregister count would be less frequent and thus cause fewer aborts.

5.5 Varying the number of registered slots

Finally, we examine `Collect` performance as the number of registered slots varies. One thread performs `Collect`s, while 15 others perform `Update`s with update period of 20,000 cycles. Initially, the number of registered slots is 16. The experiment proceeds in phases, with the `Update` threads alternately increasing and decreasing the number of registered handles every 500ms.

Figure 8 shows the throughput of the `Collect` operations for 3 seconds. As expected, the performance of `Collect` operations for Static Baseline is not significantly affected by the number of registered slots. The throughput varies slightly because the `Collect` copies less data when there are fewer registered slots. `ArrayStatSearchNo` initially performs significantly better than the Static baseline. However, when the number of registered slots increases (at 500ms, for example), its performance degrades significantly and becomes similar to that of Static baseline. Furthermore, it does not improve when the number of registered slots decreases at 1s because `Collect` traverses the maximum *historical* number of registered slots. The performance of Append-Dereg algorithms and `FastCollect` is also reduced at 500ms because the number of registered slots increases. However, when it decreases again at 1s, the performance of both algorithms becomes as good as it was initially. This clearly shows the benefit of algorithms that adapt to the number of registered slots. The best performing algorithms in this experiment are, again, the two Append-Dereg variants.

6. DISCUSSION

In this section, we discuss how various details about the HTM implementation affect our algorithms. A similar discussion appears in [7], so here we only briefly mention specific issues that are relevant to the algorithms in this paper. First, as noted in Section 1, our algorithms rely on *sandboxing* [7].

Rock's memory consistency model is like TSO [17], with transactions being treated as both loads and stores, similarly to the way atomic instructions such as CAS are treated. This model is sufficient to support our algorithms without additional memory barriers. Depending on the memory consistency model implemented by other (future) architectures that support HTM, additional memory barriers may be required.

Some of our algorithms depend on the ability to eventually commit at least some small transactions, which Rock does not guarantee. We hope that future HTMs will make such guarantees (see [1], for example). It is not difficult to modify our algorithms so that they do not need this guarantee, at the cost of making the algorithms block occasionally, by using the TLE technique [6]. The key idea is to make all transactions read a lock variable and confirm that it is not held before proceeding. This way, if a transaction fails repeatedly, its effects can be applied nontransactionally while holding the lock. Note that, in the absence of any guarantees for completion of transactions, the lock would have to be acquired using nontransactional synchronization, for example using compare-and-swap (CAS).

Some of our algorithms perform concurrent transactional and nontransactional accesses to the same variable. That is, they rely on *strong atomicity* [5]. This dependence could be avoided, at the cost of some additional overhead and code complexity, by replacing such nontransactional accesses with short transactions.

The observations of the two previous paragraphs together imply that, in order to support our algorithms, HTM must provide either strong atomicity or guarantees that certain "small" transactions will eventually commit (at least in the absence of contention). The best algorithms can be built with HTMs that provide *both* features.

Some of our algorithms, including the one presented in detail in Section 4, have been complicated somewhat by our efforts to avoid memory allocation within transactions. Because we have used standard malloc implementations that use instructions such as CAS, which are not supported in transactions on Rock, modifications that naturally belong in one transaction had to be split into multiple transactions; this complicates control flow as well as synchronization. We emphasize that this complication is due to a combination of idiosyncrasies of Rock's HTM and not using a TM-aware allocator. It is *not* a fundamental limitation of HTM in general. Nonetheless, future HTMs could simplify software by allowing the use of instructions such as CAS in transactions.

As discussed in Section 4, Rock's bounded transactions did not

impose much complexity on our algorithms, because we would want to avoid large transactions even if unbounded transactions were supported. However, in some cases we believe we could have achieved better performance if we could use larger transactions. In other cases, feedback about transaction failure reasons appeared to indicate a store buffer overflow, but it was difficult to understand why, even examining assembly code in detail. This experience suggests that support for larger or unbounded transactions may still make programming easier. Relatedly, the ability to capture more detailed information about a transaction and the reason it failed would be helpful too.

7. CONCLUDING REMARKS

We have shown that hardware transactional memory (HTM) can facilitate dynamic sized concurrent data structures that are superior in terms of simplicity, flexibility, performance, and space usage, compared to those that do not use HTM. Our results add to a growing body of practical evidence that HTM has the potential to make effective concurrent programming significantly easier [7, 8].

Known limitations of Rock have complicated our algorithms to some extent. We expect future HTM implementations to have fewer limitations, making it even easier to achieve similar results.

Some have argued that HTM provides little or no inherent benefits for concurrent data structures. However, the theoretical results on which these arguments rest focus on issues of secondary practical importance: strong progress properties such as wait-freedom, worst-case time complexity under extreme contention, and fully asynchronous models that preclude common practical techniques such as back-off.

In particular, Attiya and Hendler [4] have used such results to argue that HTM does not help to solve the "adaptive collect" problem. Despite the similarity in names, this problem is fundamentally different from the Dynamic Collect problem, and seems to have little bearing on practical issues of dynamic memory management. Specifically, because it does not allow threads to dynamically allocate and release handles, the problem *specification* implies knowledge of the number of threads in the system, and time and space complexity that depend on this number. We therefore believe that studying (variants of) our Dynamic Collect problem is more likely to provide useful insight into the impact of HTM on practical concurrent programming, especially with respect to dynamic memory management. Finally, Attiya and Hendler acknowledge that their results have no bearing on the critical issue of programming complexity, whereas our results demonstrate that HTM indeed facilitates simpler and more flexible algorithm designs.

References

[1] Advanced Micro Devices. Advanced synchronization facility proposed architectural specification, March 2009. Publication # 45432, Revision: 2.1.

[2] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, September 1993.

[3] H. Attiya, A. Fouren, and E. Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, April 2002.

[4] H. Attiya and D. Hendler. Time and space lower bounds for implementations using k-cas. *IEEE Transactions on Parallel and Distributed Systems*, February 2010.

[5] C. Blundell, E. C. Lewis, and M. Martin. Deconstructing Transactions: The Subtleties of Atomicity. In *the 4th Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2005.

[6] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaum. Applications of the adaptive transactional memory test platform. Transact 2008 workshop. `http://research.sun.com/scalable/pubs/TRANSACT2008-ATMTP-Apps.pdf`.

[7] D. Dice, Y. Lev, V. J. Marathe, M. Moir, D. Nussbaum, and M. Oleszewski. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, 2010.

[8] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, 2009.

[9] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski. Early experience with a commercial hardware transactional memory implementation. Technical Report TR-2009-180, Sun Microsystems Laboratories, 2009.

[10] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems*, May 2005.

[11] M. Herlihy, V. Luchangco, and M. Moir. Space and time adaptive non-blocking algorithms. *Electronic Notes in Theoretical Computer Science*, April 2003.

[12] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*, ISCA '93, 1993.

[13] JSR166: Concurrency Utilities. http://gee.cs.oswego.edu/dl/concurrency-interest/.

[14] M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, June 2004.

[15] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th annual ACM symposium on Principles of distributed computing*, PODC '96, 1996.

[16] M. Saks, N. Shavit, and H. Woll. Optimal time randomized consensus making resilient algorithms fast in practice. In *Proceedings of the 2nd annual ACM-SIAM symposium on Discrete algorithms*, SODA '91, 1991.

[17] Sparc International, Inc. The SPARC architecture manual, version 8, 1991.