Memory Management for Concurrent Data Structures on Hardware Transactional Memory

Peter Pirkelbauer
University of Alabama at Birmingham
Birmingham, AL 35294
pirkelbauer@uab.edu

Amalee Wilson
University of Alabama at Birmingham
Birmingham, AL 35294
amalee@uab.edu

Hadia Ahmed
University of Alabama at Birmingham
Birmingham, AL 35294
hadia@uab.edu

Reed Milewicz
Sandia National Laboratories
Albuquerque, NM 87123
rmilewi@sandia.gov

Abstract

Similar to fine-grained locking, and lock-free programming, memory management poses challenges to programming systems with hardware transactional memory (HTM). Thus, scalable data structures need to integrate a memory management scheme, such as hazard pointers, repeated offender, reference counting, and StackTrack.

In this paper, we revisit epochs, another popular memory management technique, and offer an interpretation for HTM systems. HTM helps avoid a common problem of quiescent techniques where delayed or failed threads prevent memory reclamation. In transactional mode, HTM allows threads to interrupt other delayed threads that prevent progress otherwise. This paper describes the design and implementation of this technique and discusses tradeoffs compared to other techniques. Experiments were conducted on Intel Haswell and Power8 architectures. The described technique is competitive with other HTM based techniques. Our results also demonstrate that under many scenarios, HTM based algorithms can be as fast as other optimized algorithms.

1. Introduction

Threads in shared memory multiprocessor systems communicate by executing reads and writes to shared data. The result of the concurrent execution of a number of operations is dependent on the threads' interleaving. Thus, programmers need to rely on synchronization mechanisms to enforce the desired interleaving of operations. The most common technique for synchronizing concurrent threads is the use of a mutual exclusion lock (mutex) [19], which blocks all contending threads except the one holding the lock, thereby guaranteeing thread-safe operations. Even for small multi-core systems, the use of blocking reduces parallelism, introduces safety hazards, and can lead to convoying effects that can seriously hurt application performance [13].

In an effort to eliminate problems of coarse grain locks, fine-grain locking and lock-free synchronization techniques have been explored. Until recently, most hardware architectures supported lock-free programming by providing single- or double-word wide atomic primitives. These primitives have been employed to implement many concurrent data structures. One challenge that most of these implementations face is determining when an object that has been removed from a data structure can actually be freed. Several solutions have been proposed to solve this problem. *Fine-grain techniques* [3, 5, 11, 20, 27, 28, 34] track the use of each pointer individually. Thus they bound the number of non-reclaimable memory locations in the presence of thread failures, but incur a high overhead as they need to interact with every pointer load. To some extent this problem can be remedied by coarsening pointer tracking [8], or by distinguishing between read and write operations [10]. *Coarse-grain techniques* work on an per-operation level, so they incur less runtime overhead. However, thread failures in the middle of an operation prevent memory reclamation, so such techniques are considered blocking [5]. Coarse grain techniques include the application of fuzzy barriers [15] to guarantee container quiescence, read-copy update mechanisms [32] that offer inexpensive reads but costly updates, and epochs (or timestamps) [12]. Debra+ is a recent epoch implementation that uses POSIX signals to abort operations that are in progress when a thread does not advance its epoch in some specified amount of time [9]. Finally, *garbage collection* [6] is a natural solution to this problem of determining when to free objects, but currently no available garbage collection technique offers all desirable properties. [30]

1.1 Contributions and Outline

In this paper we will revisit epochs and utilize hardware transactions for reading and updating shared data structures. Similar to Debra+, if a thread is delayed and prevents other threads from freeing memory, a thread can abort an ongoing operation by modifying the delayed threads epoch counter. We tested our implementation on a skip list with varying depth (from a linked list to N levels) and compare it to a fine-grain locking and non-blocking implementation [19] using available memory management techniques, as well as other HTM based implementations.

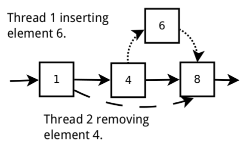

Figure 1: Linked list with Thread 1 inserting and Thread 2 removing an element

The remainder of this paper is outlined as follows. §2 provides the background of this work, §3 and §4 describe our implementation of skiplists and epoch memory management in detail, §5 evaluates our approach by comparing it against other known HTM and non-HTM techniques, §6 gives an overview of related work, and §7 concludes and offers some ideas for future work.

2. Background

2.1 Concurrent Linked-lists

Linked lists are a widely used data structure in sequential and concurrent computing. A problem of concurrent linked lists is the deletion of an element (the victim). An implementation of delete that applies just compare and exchange to update the predecessor's next pointer with the victim's next pointer would lose any update to the victim's successor that took place after the victim's pointer was read. In Fig. 1 Thread 1 inserts an element 6. It has determined that 4 is the predecessor and 8 is the successor. At the same time, Thread 2 removes element 4. It may have seen that 8 is 4's successor, thus updating 1's next element to 8. This would lose the inserted element 6. A fine-grained locking implementation may use locks on the victim and its predecessor to disallow any modification while the deletion is taking place. Note that any other reading thread can optimistically read through the affected nodes. Harris's lock-free implementation [16] marks the victim's next pointer, thereby indicating that a node has been logically deleted. When other threads find a marked node, they help remove the element from the data structure.

As a remedy to such problems, Herlihy and Moss conceived the notion of transactional memory to allow atomic updates of multiple memory locations[18]. Transactional memory is an optimistic concurrency control mechanism [23]. Executed transactions are serializable. A transaction is comprised of a sequence of instructions that are executed speculatively. If it succeeds, all updates take effect atomically. If an element that has been read in a transaction is modified by other threads, the transaction aborts and no update takes effect.

While we could execute an operation in a single transaction [2] it is often not desirable to do that when some operations are not read-only. If multiple threads execute a coarse grain transaction, only one (or maybe none) may succeed, while others get aborted due to transactional conflicts. Consider a linked list where three threads update the first, second, and third element respectively. Before they execute their respective updates, the third thread must have read the first two elements, and the second must have read the first element. An update by the first thread conflicts with the read of the other threads and possibly aborts them, unless the other threads finish their transactions before the first thread can execute an update. Note that memory management in systems with coarse grain transactions is as simple as in coarse grain locks. Any thread that successfully executes a transaction updating an object's entry point causes an abort for any other thread accessing the same object through the same entry point.

2.2 Systems with Hardware Transactional Memory

In actual HTM implementation, the size of a transaction is limited by the hardware and ranges from a few kilobytes (IBM zEC12 and IBM Power8) to multiple megabytes (IBM Blue Gene/Q). Moreover, most HTM implementations are best-effort and do not provide progress guarantees. Due to cache conflicts or operating system scheduling, a transaction can spuriously fail long before the maximum transaction size is reached. On such systems programmers are required to provide a non-transactional fall-back path to work around these limitations.

2.3 Transaction Partitions

To cope with these limitations, many HTM based algorithms divide a large transaction into several smaller partitions. Consistency-oblivious computing [4] divides transactions in a read and update partition. Xiang and Scott describe automatic partitioning for large transactions [41]. Partitioning schemes may distinguish between read and write partitions, where reading is often done outside of transactions for performance reasons. Several concurrent data structures use a variable size transaction window to dynamically adjust the transaction size to the current workload [3, 11]. However, partitioning a large transaction into several smaller steps reintroduces the memory management problem. A transaction tx that is partitioned into several smaller transactions $tx_0, tx_1, ..$ needs to secure all objects at the end of a partition tx_n that are needed to start the next transaction tx_{n+1}. Compared to hazard pointers, a transactional linked list needs to secure only a single anchor node. The anchor can be validated at the beginning of a transaction, and the transaction will fail if the node gets invalidated.

Recent work has revisited and improved fine-grain memory management techniques, such as reference counting and hazard pointers, by using hardware transactions [11]. It has been found that HTM enhances these techniques in several ways. The major benefit is that HTM makes telescoping possible. Only objects that need to be held in between two transaction partitions need to be secured [25]. Depending on the supported transaction sizes, this helps eliminate costly per object operations for hazard pointers and contention on reference counters [11].

2.4 Epoch

Many concurrent systems avoid tracking every single object and resort to coarse grain techniques such as epochs, read-copy-update (RCU) [32], and quiescent techniques. However, these techniques are considered blocking because a single delayed or failed thread can prevent memory from being freed [5].

The key idea of epochs is that each thread has its own epoch counter. When a thread starts an operation, the counter is incremented, and when the operation finishes, the counter is incremented again. Thus an even counter indicates no activity, while an odd counter indicates that an operation is in progress. After a thread has removed an object from a data structure, it reads the epoch counters from all threads. The object is tagged with those counter values. When a thread attempts to free an object, it reads the epoch counters a second time. A removed object can be actually reclaimed if the following condition holds for every thread: thread has advanced its epoch, or a thread's epoch counter indicates no active operation.

Recently Brown [9] has introduced Debra+, an epoch implementation that can abort another delayed thread through the means of POSIX signal. A thread that sees another thread delayed amidst an active operation can send a signal to that thread. The interrupted thread will execute the signal handler and abort its current operations by executing a `siglongjmp`. Debra+ does not suffer from delayed threads and is non-blocking. The application of epoch memory management to concurrent data structures can be easily automated by a source-to-source transformation system, as each entry and exit point of an operation are augmented with code that interacts with the memory manager [9].

2.5 Memory Ordering

Since we compare our approach against an implementation using relaxed memory operations, we also introduce the C++11 memory model briefly. C++11 distinguishes between data operations and synchronization operations. Memory locations subject to data races have to be of atomic type. A data race is defined as two or more concurrent memory accesses to the same memory location, where at least one of them is a write [33]. Programmers can exercise fine-grain control over memory ordering by tagging atomic operations. By default atomic operations use sequential consistency (tag `memory_order_seq_cst`) to establish a total order among them. A use case are standard hazard pointers, where a sequentially consistent operation separates the publication of a pointer from its validation. That operation needs to be globally ordered with another thread starting to scan published hazard pointers [17]. The tags `memory_order_release` / `memory_order_acquire` form a pair on the stored/loaded value. They guarantee that the reading thread sees all memory updates in the storing thread that occurred before the store tagged with release. The `memory_order_release` / `memory_order_consume` tags also form a pair on the stored/loaded value. Compared to `memory_order_acquire`, `memory_order_consume` offers weaker guarantees. It preserves order only for data that is accessed through the "consumed" value. The tag `memory_order_relaxed` does not establish an ordering relationship. Descriptions of the C++11 memory model and more subtle details are described by the C++11 standard [21], Boehm and Adve [7], and Williams [39].

Linked lists commonly use release/consume semantics to read the next pointer of a cell and `release` to insert a new element [26]. Consider the code for `insert` in the following snippet. `insert` creates a new element, and inserts it into the linked list between `prev` and `next`. The release tag on the CAS guarantees that the order between the constructor call and the CAS is preserved.

```
struct IntList { int data; std::atomic<IntList*> next; };

void insert(IntList* prev, IntList* next, int val) {
  IntList* el = new IntList(val, next);
  if (!prev−>next.CAS(next, el, std::memory_order_release, ...)) {
    // insert failed, start over
  }
}
```

Iterating through the linked list would use consume semantics. Consider the following code snippet. consume preserves dependencies, thus a reading thread can safely access the content of a linked list node through `next`, which was loaded using consume. Together with release, consume preserves the happens-before relationship object creation $\rightarrow$ data structure update $\rightarrow$ pointer read $\rightarrow$ data access through pointer.

```
auto find(int val) −> std::pair<IntList*, IntList*> {
  IntList* prev = ...
  IntList* next = prev−>next.load(std::memory_order_consume);
  while (next && next−>data < val) {
    prev = next; next = prev−>next.load(std::memory_order_consume);
  }
  return std::make_pair(prev, next);
}
```

`std::memory_order_consume` can, in principle, be implemented efficiently on most architectures. On x86 consume is a `mov` instruction, and on PowerPC systems, consume could also be implemented through a cheap `lwz` instruction[31], as long as the compiler can track dependencies. However, many current compilers treat consume similar to acquire. Acquire is a stronger primitive that also guarantees ordering when the last data access is independent from the acquired value (e.g., some other value that the releasing thread has stored). On the Power8, acquire requires a light weight barrier to be inserted after the load [1]. Fine-tuning data structures by using non-sequentially consistent operations is challenging because reasoning about possible interleavings and memory orderings is difficult.

Conversely, hardware transactions processing offers the benefit of coarse grain synchronization at the beginning and the end of a transaction. Any read within a transaction does not require memory ordering barriers (when update operations are performed within transactions). This greatly simplifies the design of algorithms. In terms of performance, reading and updating linked data structures can be performed more cheaply within a transaction. However, this benefit may be countered by transactional overhead at the start and commit time.

3. Implementation

This section describes our skip list implementation for HTM systems followed by a description of our epoch implementation.

3.1 Skip list

In order to test memory managers, we implemented an HTM based skip list in C++ enhanced with low-level operations accessing the HTM hardware. A skip list is a probabilistic data structure that consists of multiple layers, where each layer is organized as linked list. When an object is inserted, the data structure randomly determines in how many layers an object participates. The likelihood that an object participates in the next layer decreases by a predetermined factor. Both lock-free and fine-grained locking algorithms for skip lists are known [19]. The main algorithmic difference between these two is that the lock-free version, similar to a list implementation, marks nodes to be deleted and then relies on helping by other threads that happen to read through the same nodes. In contrast, the fine-grained locking based skip list uses a lock-free `find` method to identify all predecessor and successor nodes of an operation. Then an update operation acquires locks for all predecessors and successors (or the victim in case of `remove`) in descending order. Once all locks have been acquired, the data structure can be modified. While a thread updates the data structure, any other thread that needs to update the same nodes is blocked. When it eventually acquires the locks, it tests whether the modification invalidated its operation, in which case the thread restarts the operation. Note that `find` ignores locks and continues reading through these nodes. Interestingly enough, this property renders fine-grained memory management techniques unsuitable. Consider a thread t_1 reading a node a, which has a successor b. Before it accesses its data, t_1 secures a by, for example, publishing its pointer. Then two other thread remove a and b from the list. Since the next field of a has not been updated, t_0 may "secure" b at a time when it has been released. The use of HTM helps solve this problem.

We chose to derive our implementation from the fine-grain locking skip list. The rationale for this choice was that HTM allows us to replace the critical section with a transaction that modifies the data structure atomically. We implemented two update operations (`insert`, `erase`) and one operation that queries whether an element is in the data structure (`contains`). Update operations are executed in two phases. (1) find finds a node with a given value, or if that node is not present, its immediate successors, and all predecessor nodes (2) execute updates the data structure.

A skip list node contains the data, the number of levels, and a pointer to an array containing pointers to the successor nodes. The successor nodes are manipulated in a single transaction and are in a valid state until the node gets deleted. In this case, the link at level zero is set to null.

find: The find operation is further split into several smaller transactions to minimize conflicts with concurrent updates and to deal with capacity aborts. Like Dragojević et al.[11] we use dynamically sized transactions. The transaction size is computed as rolling average over the last four successful transactions. When a transaction repeatedly fails, we reduce the transaction size by half. When the last four transaction sizes were the same, we double the transaction size. Instead of dealing with the actual memory footprint, we count the transaction size in terms of reading the number of next pointers. Thus, the minimum transaction size is two (so find can make progress) and the maximum (also the starting point) has been empirically determined for a given hardware by executing a single operation on a skip list that does not use memory management and stores integers. `find` traverses a skip list until it reaches its transaction limits. It secures all immediate predecessors at a higher level and an anchor node. The first step in the follow-up transaction validates that the anchor node has not been removed in the meantime. This is done by reading its successor node at the lowest level. If that is null, the node has been deleted and the operation is restarted.

Executing find in a transaction serves several purposes. With a fine-grained memory management scheme, we get the benefits of telescoping. Only nodes that are held in between transactions need to be secured. While inside of the transaction, we can rely on conflict aborts to stop a transaction when another thread has modified it. With coarse-grain memory management, the transaction allows other threads to abort an operation that seemingly makes no progress. In addition, the transaction allows us to use cheap relaxed reads of interlinked data structures.

execute: before an update operation begins, threads carry out non-transactional code, such as creating a new node and interlinking its next pointers to the expected successors. A update transaction validates that the predecessor and successor (or victim) nodes produced by `find` have not changed. The last part of the execution updates the data structure. We do not divide this update any further, though this could be achieved through descriptors. Instead we rely on hardware capabilities to handle transactions of this size.

The use of transactions greatly simplifies the algorithm design because a node is either fully part of a data structure, or it is not part of the data structure. The fine-grained locking algorithm achieves this by using two boolean flags, which are redundant in an HTM environment.

If the validation fails, the operation is restarted. Here, both implementations allow for a minor improvement. Instead of restarting the operation from the beginning, the operation could restart with the first valid predecessor node that has more levels than the victim (or inserted) node.

fallback path: When the execution of an operation in a transaction consistently fails, the skip list falls back to a single global lock implementation. Due to variable size transactions, the fall back path will be rarely taken.

4. Memory Management Design and Implementation

In this section we describe the implementation of epochs for HTM systems.

4.1 Epochs for HTM systems

We derive epochs for HTM systems from a standard epoch implementation. Standard epochs increment a counter at the beginning and at the end of a transaction. Other threads only read a thread's epoch counter and determine whether an epoch is active, and if a thread has made progress since a node was removed from the data structure.

In epochs for HTM systems, we allow other threads to update the epoch counter, thereby terminating a thread's operation. We describe the necessary modifications below.

```
void begin_epoch() {
  // the first time a thread is executed
  if (!epochdata)
    epochdata = announceThread();

  size_t epochval = epochdata->epoch.load(relaxed) + 1;
  epochdata->epoch.store(epochval, relaxed);
  atomic_thread_fence(seq_cst);
}

void end_epoch() {
  size_t epochval = epochdata->epoch.load(relaxed);

  // exiting an operation
  if (isActive(epochval))
    epochdata->epoch.CAS(epochval, epochval+1, relaxed, relaxed);

  if (timeToCollect(epochval)) release_memory();
}
```

The first time a thread starts an epoch, `begin_epoch` creates a new entry, initializes the epoch counter to 0, and enters it into a global data structure. `epochdata` is a thread local pointer that points to a thread's epoch data. The thread increments its epoch counter and uses a sequentially consistent barrier to globally order this operation with other threads that read that counter when they release memory. The code of `begin_epoch` is the same as for epochs without HTM.

Exiting an epoch increments the counter. Conversely to epochs without HTM, the epoch counter can be modified by other threads. A CAS guarantees that a thread does not increment an epoch counter that has been modified by another threads' cancellation. The CAS uses relaxed consistency. It is ordered with the preceding transaction at the transaction boundary. The C++ language guarantees that the update eventually becomes visible to other threads. Thus, using `relaxed` may delay, but not indefinitely prevent, freeing of nodes.

To adapt this mechanism for transactions, we introduce functions for validation and cancellation. The validation is called from a transactional context, and cancellation allows other threads to terminate a transaction.

```
void validate_transaction() {
  if (!(epochdata->epoch.load(relaxed) & 1))
    tx::abort();
}

void cancel_transaction(size_t id, size_t currepoch) {
  epoch_t* epochptr = lookup_thread(id);
  epochptr->epoch.compare_exchange_strong(currepoch, currepoch+1);
}
```

`validate_transaction` reads the thread's epoch counter and loads it into the transaction's read set. If the epoch counter was incremented by another thread before the transaction started executing, `validate_transaction` aborts the transaction immediately. If some other thread increments the the epoch counter while the transaction is executing, HTM detects the conflict and aborts the transaction. Since other threads only cancel a transaction (making the epoch counter even) it is sufficient to test whether the epoch is still set active.

```
// use of validate_transaction
if (tx::begin()) { // starts a transaction
  ...
  validate_transaction();
  ...
  tx::end();
}
else { // abort handler
  // retries or enters lock-based fall back path
}
```

`cancel_transaction` receives two arguments `id` and `currepoch`, indicating the id and epoch of the thread to be canceled. After looking up the thread's epoch record the transaction is canceled by a CAS incrementing the epoch counter and setting it to the next inactive value. If multiple threads attempt to cancel a thread's transaction, only the first thread succeeds in modifying the counter value. Other threads' CAS will fail thereby indicating that an operation is no longer active. Either the delayed thread has finished its operation meanwhile, or some thread has canceled an ongoing operation. The nodes that the thread id potentially accessed can be released.

Object reclamation: When an object is removed from a data structure it is enqueued in a list of objects whose deallocation is pending. Objects in this list will be tagged with a timestamp indicating when the object was removed from the data structure and a survival count. The timestamp is represented by a vector of epoch counters, $ts = (e_{t0}, \ldots, e_{t1})$. Initially, an object is untagged. When the number of untagged objects reaches a certain threshold τ, where $\tau >= |threads|$, a thread collects a new timestamp by reading all other thread's current epoch counters and associates the timestamp with yet untagged objects. The collected timestamp is also used to identify objects that can be freed. An object can be freed, if the comparison of its tag with the current timestamp shows that either every thread has advanced since the object was tagged, or a thread has been quiescent at the time when the object was tagged, $\forall_{0 \leq i < n},\ ts_i > tag(obj)_i \lor even(tag(obj)_i)$. If an object cannot be reclaimed, its survival count is incremented. When the survival count reaches a threshold σ, then all threads whose epoch counter prevent deallocation will be canceled. Thus, a thread will hold at most $\tau\ \sigma$ objects whose deallocation is pending.

5. Evaluation

We evaluate our implementation on two different hardware platforms: Intel Haswell and Power8. Table 1 gives an overview of the systems and their HTM capabilities [29].

	Haswell	Power8
Cores	1x4	2x10
Threads/core	2	8
Conflict-detection granularity	64 bytes	128 bytes
Transactional load capacity	4 MB	8 KB
Transactional store capacity	22 KB	8 KB
Level 1 data cache	32 KB, 8-way	64 KB, 8-way
Level 2 data cache	256 KB	512 KB, 8-way

Table 1: Overview of test systems

Comparing the two architectures, the Intel Haswell's transaction capacity is significantly larger than the Power8's. The Power8 system is a two socket system, where each socket has 10 cores, and each of the cores supports up to eight threads in hardware. In contrast the Haswell system has a single socket with four hyperthreading enabled cores. Under these parameters we expect a higher abort count on the Power8 system. We expect a higher number of transaction collisions and capacity aborts due to the higher number of supported threads and the smaller size of supported transactions. Both systems are best effort implementations and do not provide progress guarantees. Even when executed in isolation, a transaction may fail due to cache size limitation, cache associativity limitations, or context switches, for example, induced by page faults. Under a best case scenario, the Haswell can update up to 352 independent memory locations. Due to its smaller store capacity and larger cacheline size, the Power8 can update up to 64 independent memory locations. Depending on the HTM implementation, worst case sizes can be dramatically smaller. The Power8 also offers suspended transactions to allow the reading of shared data that can introduce order to transactions or to reduce the pressure on transaction size. We have not yet explored the use of that feature.

5.1 Performance Comparison

We tested the following scenarios on Power8 and Intel architectures. The code was compiled with gcc-4.8 and we used -O2 -march=native (-mcpu=native on Power8) for code optimizations. Each test was run with n operations. To initialize the data structure, $n/10$ insert operations were executed. Then we timed the execution of 45% insert and 45% alternating erase operations. We used unique elements for insert and passed existing elements to erase, so that any operation would succeed. Consequently, the data structure contained $\frac{total\ operations}{10}$ elements after the initial insert phase. Each test was run 10 times, and the average was reported after the best and worst timing were removed.

We compared several algorithms, including *epochs / HTM*, *reference counting / HTM*, *publish and scan / HTM*, and a *StackTrack* implementation.

Publish and scan / HTM publishes all hazard pointers at the end of each find partition to shared memory. The size of the find partition was dynamically determined (§ 3.1). Pointer publishing needs to be part of a transaction and therefore the number of published pointers reduces the transaction size slightly. However, due to the Power8's large cache line, publishing 16 pointers would just take one independent HTM storage location. Memory scans of published pointers are a fairly cheap operation, implemented by a sequentially consistent barrier followed by relaxed memory reads from the published pointer lists.

StackTrack publishes the location of its references to shared objects. The references are already stack allocated, thus the amount of published pointers is reduced to one. While this reduces transactional memory footprint on most systems, it also requires that scans through other thread's published references are run inside a transaction to avoid accessing a terminated thread's stack.

Reference Counting for HTM systems: Our implementation follows the Dragojević [11] reference counting implementation. HTM simplifies reference counting significantly, because a thread can read a pointer to an object and update the object's reference counter within a transaction. In order to prevent conflicts due to false sharing, we placed the reference counter on its own cacheline (i.e., separate from the next pointers). On one side this reduces transaction aborts due to false sharing, but on the other side, this increases the memory size of a transaction and consequently results in smaller transaction partitions or more conflict aborts.

To further reduce the contention on reference counters, we randomize the length of the first transaction partition within the range of 2 and the maximum transaction length. This increases the likelihood that threads searching through the same data structure region find different anchor points.

A function that is partitioned into several smaller transactions $tx_0, tx_1, ..$ needs to secure all objects at the end of a transaction tx_n that are needed to execute the next transaction tx_{n+1}. The function that modifies the reference counters compares the set of secured objects sec_n with the set identified by the previous transaction sec_{n-1}. Reference counters for objects in sec_n and not in sec_{n-1} will be incremented, and reference counters for objects in sec_{n-1} and not in sec_n will be decremented. Reference counters of objects in both sets remain unmodified.

A thread that removes an object from a data structure unlinks the object from the data structure. The object can be released as soon as its reference count reaches zero.

Other techniques: To establish a point of reference with traditional skip list implementations, we also tested a fine-grain locking implementation using epochs as memory management and a lock-free algorithm using hazard pointers. Lastly, we establish a base case by using an HTM algorithm with a memory manager that does not free, so it does not incur reclamation overhead, though it requires more memory.

The reclamation's frequency of the epoch schemes was adjusted so that the number of reclamations was similar, but not lower, than the number of StackTrack's reclamations.

Linked list: We used a sorted linked list (i.e., a skip list with a single level) and timed the execution of 100000 operations. We varied the number of threads between 1 and threads supported in hardware. In a first scenario, we generated data so that the insert locations are disjoint after the initial insert phase, with the goal to produce operations with low contention. Since this test uses a linked-list, threads that operate in later sections of the list, could be subject to conflict aborts when they read through an earlier section where another thread performs update operations. The results for the Power8 and Haswell architectures are displayed in Fig. 2 and Fig. 3 (diagrams on the right side). On the Power8, the fine-grain locking implementation with epochs was clearly fastest. On the Haswell, the fine-grain locking variant and the three HTM based techniques (i.e., publish and scan, StackTrack, and epochs) performed about the same. The HTM implementation without collection gives a rough best case scenario for the use of HTM. HTM based implementation significantly slowed down beyond the number of physical cores. The lockfree algorithm incurred high overhead with low thread counts, but after reaching the number of physical cores, the performance was similar to HTM techniques. Reference counting using hazard pointers came in last.

Under another scenario, we generated data so that operations either insert at the end or remove an element from the beginning of the list. This scheme should generate contention, but due non-determinism it is not guaranteed that several runs produce the same amount of contention. Moreover, once maximum scalability is reached, techniques that have more overhead often reduce contention. The results for the Power8 and Intel architectures are displayed in Fig. 2 and Fig. 3 (diagrams on the left side). The relative results are similar to the non-contentious case.

Skip list: We tested a skip list implementation under the same two scenarios outlined for the linked list test. Tests executed 4M operations in total. On the Intel, the maximum size of the skip list was 32 levels (the likelihood of levels decreases with the power of 2), while on the Power8 system the number of skip list levels were limited to 16 (the likelihood of levels decreases with the Power of 8) to account for the smaller HTM capacity.

Similar to linked lists, the three HTM techniques (epochs / HTM, publish and scan, and StackTrack) and the fine-grained locking container perform best. Only in the test case that induces contention on data structure accesses, the fine-grained locking skip list does not scale beyond five threads on the Power system. The HTM based approaches continue improving until the number of cores is reached. Up to 40 threads StackTrack and epochs / HTM perform similarly. Beyond that point, StackTrack scales better.

We also measured the impact of the reclamation technique on the average number of steps (reading of the pointer to the successor node) executed per transaction. Fig. 6 shows the average number of steps for two scenarios. The left side scenario shows the measurements obtained from our linked list test on the Intel Haswell. The different approaches exhibit identical transaction lengths. The exception is reference counting, whose average number of steps is significantly smaller. Under contention, the transaction length decreases. The transaction length of reference counting decreases rapidly and reaches the minimum of two with four threads.

The right side in Fig. 6 shows the transaction lengths obtained for skip lists on a Power8 system. Under no contention, no memory management executes the largest number of steps, followed by the epochs. StackTrack and publish and scan's execute slightly fewer steps in a transaction (1.7). Reference counting achieves significantly fewer steps. With increasing number of threads, the average transaction length decreases, but the relative performance differences remain. We also measured the transaction length of skip lists on the Haswell. In a single threaded execution all techniques (including reference counting) achieved a transaction length of 79. With four threads, only the transaction length of reference counting started to decrease (58), while other systems' transaction lengths remained stable.

5.2 Discussion

In this section we compare the use of different memory management techniques, including HTM versions of hazard pointers and reference counting; techniques that rely on HTM, including StackTrack and Epochs / HTM; and an epoch based technique for non-transactional memory. Table 2 provides an overview of our findings.

All techniques except epochs without HTM support are non-blocking, meaning a single delayed thread cannot prevent freeing objects. However, Debra+ is an epoch based scheme that uses signals to abort operations of delayed threads and enables epochs to become non-blocking. Hazard pointers, reference counting, and StackTrack are fine grained techniques that secure individual objects from being reclaimed, while epochs are a coarse grain mechanism that operate on a per-operation level.

Hazard pointers and Stacktrack need to scan other thread's secured pointers to determine whether an object can be safely released. If the number of threads and hazard pointers are known, the cost of scanning other thread's pointers can be amortized over a large number of objects, $O(threads * obj * log(threads * obj))$, yielding amortized constant time. In a similar fashion, epochs collect other threads' epoch counters. Thus the cost of scanning also depends on the number of threads in a system, but is independent from the number of secured objects. The collection of other threads' epochs should be amortized over a large number of reclaimed objects ($O(threads)$). Reference counting operates on a per-object basis, thus it is independent from the number of threads in the system.

Due to current hardware limitations of transactions, techniques that have a lower memory footprint are less likely to experience capacity aborts. In an execution without contention, a lower footprint means that an operation can amortize the cost of transaction start and commit through longer but fewer transactions. Traditional hazard pointers maintain a list of objects that can be safely accessed. This hazard pointer list is written before a transaction partition finishes. Thus hazard pointers incur a small memory overhead compared to other methods. StackTrack improves on that by manipulating the list of objects directly, thereby eliminating both the memory needed to store hazard pointers and the cost of copying those objects to shared memory. Reference counting incurs memory overhead for storing the counter. In our implementation the counter is located on a separate and non-adjacent cache line in order to minimize conflict aborts. On the Power8 system, the increased demands on memory caused capacity aborts. Epochs incur a small overhead in that they need to read the thread's epoch counter in every transaction. Epochs without TM incur only the cost of incrementing a counter at the beginning and end of an operation. In addition to that, Debra+ needs to save the execution context for every operation and requires clean-up code.

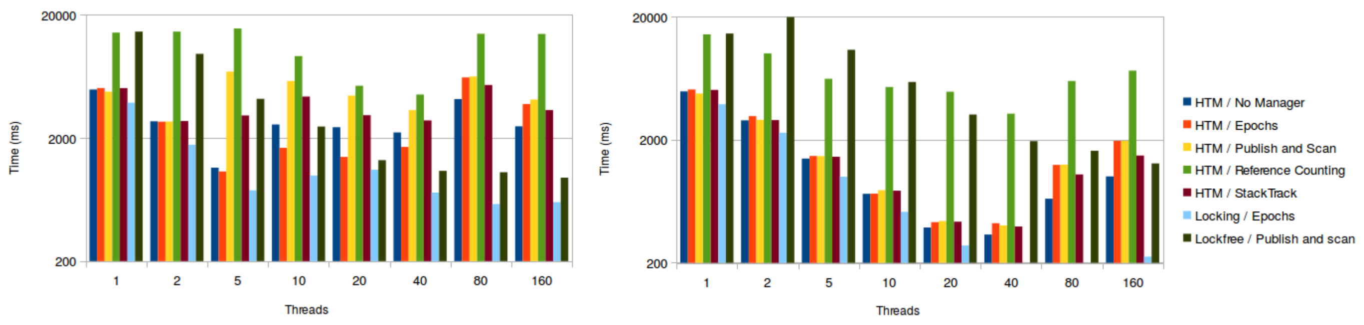

Figure 2: Execution time, Power8, sorted linked list, data induces conflicts (left) vs disjoint data (right), 100K operations

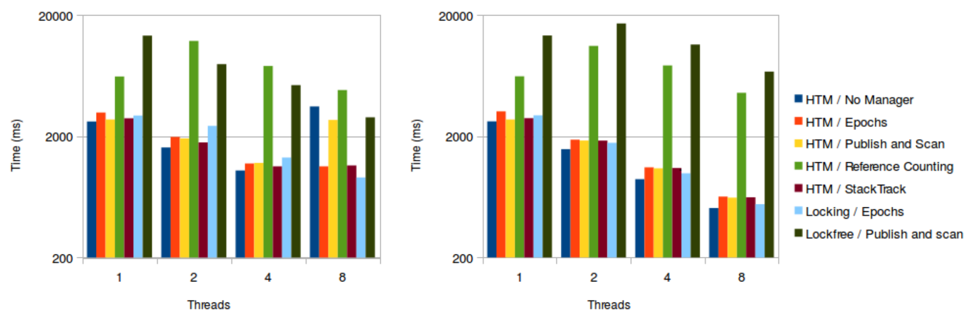

Figure 3: Execution time, Haswell, sorted linked list, data induces conflicts (left) vs disjoint data (right), 100K operations

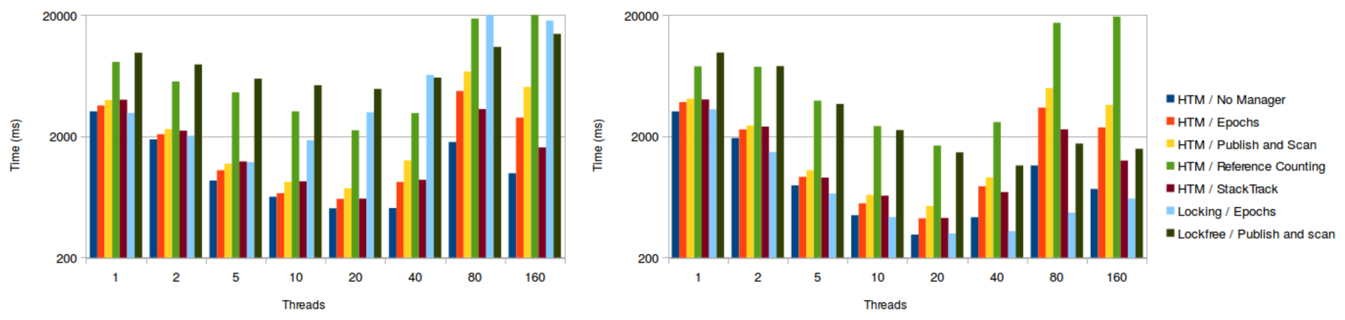

Figure 4: Power8, skip list, 16 levels, data induces conflicts (left) vs disjoint data (right), 4M operations

Mixed-mode programming refers to the ability to utilize the method in a non-transactional context. One use case is a variation to the skip list algorithm's restart sequence. The default algorithm restarts an insert or erase operation from the beginning when the corresponding predecessor or successor nodes have been modified. This could be optimized by finding a predecessor that is a valid node in the data structure. To this end, we could read the predecessor list identified by `find` starting at the levels that a node supports. The first valid node would be a good point for restarting `find`. Fine-grain systems secure objects on a per-object level, thus those objects can be safely accessed in non-transactional code. Epoch / HTM cannot do that because accessing an object outside a transaction could not be aborted by other threads, which would render the technique blocking. Epochs and Debra+ do not use transactions, though they could if transactional execution promises additional benefits. One such benefit would be the elimination of costly memory barriers inside of a transaction. With hardware expected to support larger transaction sizes, the positive performance effects of transactions on relaxed memory architectures may become more pronounced. Hazard pointers can be implemented portably without resorting to specialized operating system or hardware support. This makes them a good choice for systems that require a lock-free fallback path. StackTrack and Epochs / HTM require that architectures support HTM, while Debra+ requires that operating systems support sending signals and `siglongjmp`.

As the performance experiments indicate, most methods perform reasonably well. (We have not tested Debra+, but according to their description, the only overhead incurred in addition to epochs is saving the processor state for a long return). Most HTM based techniques significantly outperform a non-blocking publish and scan (e.g., hazard pointers). Reference counting did not deliver very good results. We attribute that to smaller transaction sizes.

6. Related Work

Fine-grained techniques bound the number of objects that cannot be freed despite a thread failure or thread delay. Reference counting [14, 28, 34, 36, 39] uses counters to keep track of the number

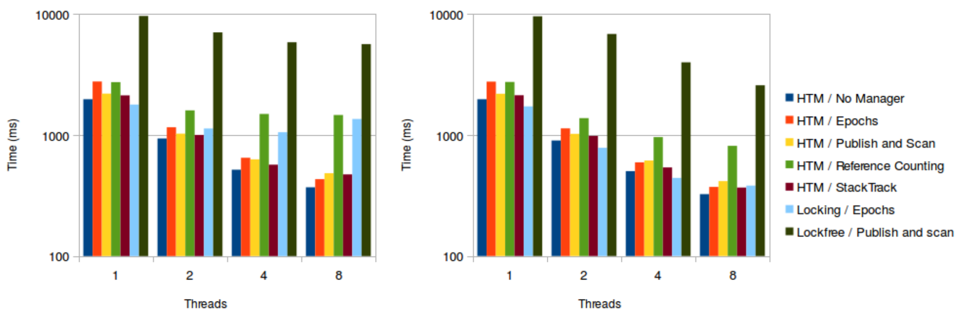

Figure 5: Haswell, skip list, 32 levels, data induces conflicts (left) vs disjoint data (right), 4M operations

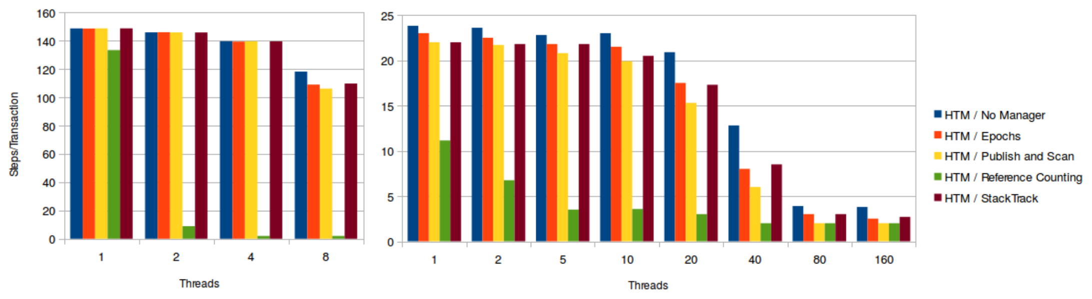

Figure 6: Average number of steps / transaction size. left: Haswell, sorted linked list, disjoint data, 100K operations. right: Power8, skip list (16 levels), disjoint data, 4M operations.

	Hazard Pointers	Reference Counting	StackTrack	Epochs	Epochs / HTM
non-blocking	✓	✓	✓	✗ (Debra+ ✓)	✓
granularity	fine	fine	fine	coarse	coarse
reclamation overhead	high	low	high	medium	medium
transactional memory footprint	high	very high	low	none	low
mixed-mode programming	✓	✓	✓	n/a	✗
relaxed read / writes	✓	✓	✓	✗	✓
portability	✓	✓	HTM	(Debra+ OS support)	HTM
performance	good (HTM)	poor	good	good	good

Table 2: Comparison of different memory management techniques

of threads that are currently holding pointers to an object. Reference counting incurs overhead on every pointer load in form of a read-modify-write operation that manipulates the object's counter. Publish and scan techniques, such as Hazard pointers[27] or repeated offenders[20] publish the object references that a thread currently holds. Before freeing an object, threads scan other threads for their published pointers, and delay freeing of an object if it is still in use. These techniques incur a runtime overhead in the form of a sequentially consistent barrier when a pointer is read [17]. Balmau et al. present a fast-path/slow-path technique, where the slow path uses hazard pointers. The slow path avoids expensive barriers by deferring object reclamation and using auxiliary processes that ensure the global visibility of published objects. Dragojević et al. demonstrate the benefits of HTM hardware for dynamic collect algorithms [11]. Cohen and Petrank [10] delay freeing reclaimed memory and allow optimistic reads, where invalid reads can be recognized and lead to a restart. StackTrack[3] is an HTM technique that exposes pointers to objects on the stack instead of using hazard pointers. This reduces memory overhead, a precious resource in transactional context, and runtime, because HTM eliminates the need for expensive barriers.

Coarse-grain techniques include the application of fuzzy barriers [15] to guarantee container quiescence, read-copy update mechanisms [32], that offer inexpensive reads, but costly updates, and epochs (or timestamps) [12]. Epochs incur overhead per each container operation. Coarse grain techniques are probe to thread delay and failures. Debra+ is a recent epoch implementation that uses POSIX signals to abort operations that are in progress and where a thread did not advance its epoch in some specified amount of time[9].

Drop-the-anchor [8] is a combination of timestamps and hazard pointers. In case of a thread failure or delay, the state of that thread is reconstructed through anchor points. Drop-the-anchor is a data structure specific technique that currently is implemented only for linked lists.

Finally, garbage collection techniques[6] also exist, though thus far non available garbage collection technique offers all desirable properties [30].

Several research projects investigate how to employ hardware transactional memory support for concurrent data structures. Most existing methods partition an operation into several segments. Timnat and Petrank [35] propose normalized lock-free data structures that combine operations into read writes and write partitions. Similar partitioning schemes were explored for transactions. Consistency-oblivious computing [4] divides transactions in a read and update partition. Xiang and Scott describe automatic partitioning for large transactions [41]. The authors also present red-black trees and evaluated their approach on an IBM EC12 system [40]. Wang et al. describe a skip list implementation where transactions were used to implement the data structure updates [38]. Li et al. explored HTM-based optimization in the context of Cuckoo hashing [24]. Several studies explore programming models where HTM helps elide locks for shared memory updates in parallel systems [22, 23, 37].

7. Conclusion and Future Work

In this paper we have presented an extension to epochs for managing memory for data structures on hardware transactional memory systems. The use of transactions makes a data structure operation cancelable by other threads as long as the fallback path is not needed. We have tested our approach against other existing memory management techniques for HTM and conventional systems. Epochs / HTM performs well when compared to other memory management techniques.

We plan to incorporate epochs / HTM into other data structures commonly used in concurrent systems. Similar to other work [11], this paper assumes that an HTM based solution is able to make progress. The tested systems do not offer this guarantee, thus the integration of a fallback method is critically important for a realistic software system.

Acknowledgments

We thank the anonymous referees for their feedback and suggestions for improvements.

References

[1] C/C++11 mappings to processors, 2011. www.cl.cam.ac.uk/ pes20/cpp/cpp0xmappings.html.

[2] Brief transactional memory GCC tutorial, 2012. pmarlier.free.fr/gcc-tm-tut.html.

[3] D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, and N. Shavit. Stacktrack: An automated transactional approach to concurrent memory reclamation. In *Proceedings of Eurosys 2014*. ACM, 2014.

[4] H. Avni and B. Kuszmaul. Improving HTM scaling with consistency-oblivious programming. In *Proceedings of Transact 2014*, 2014.

[5] O. Balmau, R. Guerraoui, M. Herlihy, and I. Zablotchi. Fast and robust memory reclamation for concurrent data structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 349–359, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4210-0. doi: 10.1145/2935764.2935790.

[6] H. Boehm. Bounding space usage of conservative garbage collectors. In *Proceeedings of the 2002 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2002.

[7] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 68–78, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375591.

[8] A. Braginsky, A. Kogan, and E. Petrank. Drop the anchor: Lightweight memory management for non-blocking data structures. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 33–42, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1572-2. doi: 10.1145/2486159.2486184.

[9] T. A. Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 261–270, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3617-8. doi: 10.1145/2767386.2767436.

[10] N. Cohen and E. Petrank. Efficient memory management for lock-free data structures with optimistic access. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 254–263, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3588-1. doi: 10.1145/2755573.2755579.

[11] A. Dragojević, M. Herlihy, Y. Lev, and M. Moir. On the power of hardware transactional memory to simplify memory management. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '11, pages 99–108, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0719-2. doi: 10.1145/1993806.1993821.

[12] K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, feb 2004.

[13] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), may 2007. ISSN 0734-2071. doi: 10.1145/1233307.1233309.

[14] A. Gidenstam, M. Papatriantafilou, and P. Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. In *ISPAN 2005: Proceedings, 8th International Symposium on Parallel Architectures, Algorithms and Networks*, 2005.

[15] R. Gupta. The fuzzy barrier: A mechanism for high speed synchronization of processors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS III, pages 54–63, New York, NY, USA, 1989. ACM. ISBN 0-89791-300-0. doi: 10.1145/70082.68187.

[16] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC '01*, pages 300–314, London, UK, 2001. Springer-Verlag. ISBN 3-540-42605-1.

[17] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67(12):1270–1285, Dec. 2007. ISSN 0743-7315. doi: 10.1016/j.jpdc.2007.04.010.

[18] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993. ISSN 0163-5964. doi: 10.1145/173682.165164.

[19] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, revised 1st edition edition, 2012. ISBN 0123705916, 9780123705914.

[20] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, 2005. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/1062247.1062249.

[21] ISO/IEC 14882 International Standard. *Programming Language C++*. JTC1/SC22/WG21 - The C++ Standards Committee, 2011.

[22] A. Kleen. Scaling existing lock-based applications with lock elision. *Commun. ACM*, 57(3):52–56, Mar. 2014. ISSN 0001-0782. doi: 10.1145/2576793.

[23] M. Kunaseth, R. K. Kalia, A. Nakano, P. Vashishta, D. F. Richards, and J. N. Glosli. Performance characteristics of hardware transactional memory for molecular dynamics application on bluegene/q: To-

ward efficient multithreading strategies for large-scale scientific applications. In *Proceedings of the 14th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing*, 2013.

[24] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 27:1–27:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2704-6.

[25] Y. Liu, T. Zhou, and M. Spear. Transactional acceleration of concurrent data structures. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 244–253, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3588-1. doi: 10.1145/2755573.2755598. URL `http://doi.acm.org/10.1145/2755573.2755598`.

[26] P. E. McKenney, T. Riegel, and J. Preshing. Towards implementation and use of memory_order_consume. Technical Report N4036, JTC1/SC22/WG21 C++ Standards Committee, March 2014.

[27] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004. ISSN 1045-9219. doi: http://dx.doi.org/10.1109/TPDS.2004.8.

[28] M. M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, Rochester, NY, 1995.

[29] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari. Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8. *SIGARCH Comput. Archit. News*, 43(3):144–157, June 2015. ISSN 0163-5964. doi: 10.1145/2872887.2750403.

[30] E. Petrank. Can parallel data structures rely on automatic memory managers?, 2012. Joint keynote talk for the 2012 International Symposium on Memory Management and the Workshop on Memory Systems Performance and Correctness.

[31] J. Preshing. The purpose of memory_order_consume in C++11, 2014. preshing.com/20140709/the-purpose-of-memory_order_consume-in-cpp11/.

[32] D. Sarma and P. E. McKenney. Making rcu safe for deep sub-millisecond response realtime applications. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 32–32, Berkeley, CA, USA, 2004. USENIX Association.

[33] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, nov 1997. ISSN 0734-2071. doi: 10.1145/265924.265927.

[34] H. Sundell. Wait-free reference counting and memory management. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 24b–24b, April 2005.

[35] S. Timnat and E. Petrank. A practical wait-free simulation for lock-free data structures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 357–368, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2656-8. doi: 10.1145/2555243.2555261.

[36] J. D. Valois. Lock-free linked lists using compare-and-swap. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-710-3. doi: http://doi.acm.org/10.1145/224964.224988.

[37] M. D. Wang, M. Burcea, L. Li, S. Sharifymoghaddam, G. Steffan, and C. Amza. Exploring the performance and programmability design space of hardware transactional memory. In *Proceedings of the 9th ACM SIGPLAN Workshop on Transactional Computing*, Transact '14, 2014.

[38] Z. Wang, H. Qian, H. Chen, and J. Li. Opportunities and pitfalls of multi-core scaling using hardware transaction memory. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, APSys '13, pages 3:1–3:7, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2316-1. doi: 10.1145/2500727.2500745.

[39] A. Williams. *C++ concurrency in action: practical multithreading*. Manning Publ., Shelter Island, NY, 2012.

[40] L. Xiang and M. Scott. Composable partitioned transactions. In *Proceedings of the 2012 5th Workshop on the Theory of Transactional Memory*, 2013.

[41] L. Xiang and M. L. Scott. Software partitioning of hardware transactions. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 76–86, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3205-7. doi: 10.1145/2688500.2688506.