The Teleportation Design Pattern for Hardware Transactional Memory

Nachshon Cohen[1], Maurice Herlihy[2], Erez Petrank[3], and Elias Wald[4]

1 **Computer Science Dept., EPFL,** `nachshonc@gmail.com`
2 **Computer Science Dept., Brown University,** `mph@cs.brown.edu`
3 **Computer Science Dept., Technion,** `erez@cs.technion.ac.il`
4 **Computer Science Dept., Brown University,** `elias_wald@brown.edu`

Abstract

We identify a design pattern for concurrent data structures, called *teleportation*, that uses best-effort hardware transactional memory to speed up certain kinds of legacy concurrent data structures. Teleportation unifies and explains several existing data structure designs, and it serves as the basis for novel approaches to reducing the memory traffic associated with fine-grained locking, and with hazard pointer management for memory reclamation.

Keywords and phrases Hardware transactional memory, concurrent data structures

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

The emerging widespread availability of *hardware transactional memory* (HTM) [18] on commodity processors such as Intel's x86 [20] and IBM's Power [7] architectures provides opportunities to rethink the design and implementation of highly-concurrent data structures.

This paper identifies the *teleportation* design pattern, which uses best-effort hardware transactional memory to reduce the memory costs of certain kinds of concurrent data structures. Retrospectively, the teleportation pattern provides a common framework for describing and explaining several heretofore unrelated techniques from the literature. Here, we describe two novel applications of this design pattern: to reduce the memory overheads associated with fine-grained locking, and with hazard pointer management for memory reclamation.

In highly-concurrent data structures, a high-level operation (such as adding a value to a hash set) is implemented as a sequence of atomic state transitions, where each transition is identified with a *memory operation*. Memory operations include loads, (non sequentially-consistent) stores, memory barriers, and atomic operations such as compare-and-swap (CAS). Operations of concurrent threads can be interleaved.

This model encompasses both lock-based and lock-free data structures. For example, a thread applying an operation to a lock-based data structure might spin until the lock is free (repeated loads), acquire that lock (CAS, with implicit barrier), modify the data structure (loads and stores), and finally release the lock (store and memory barrier). A thread traversing a lock-free data structure might read fields (loads, perhaps barriers) and perform one or more CAS operations.

Sometimes, a thread's sequence of memory operations has the property that if the thread were to execute that sequence uninterruptedly, that is, without interleaving steps with other threads, then some of the memory operations could be *elided* (omitted), because later memory operations "undo" the effects of earlier ones. For example, *Lock elision* [25, 26] is a well-known technique that exploits the observation that a matching lock acquisition and release "cancel" one another, leaving the lock's memory state unchanged at the end of the critical section. If a thread executes a critical section as a best-effort hardware transaction, then it can reduce memory traffic by eliding the lock acquisition and release.

The *teleportation pattern* retrofits an existing data structure with additional speculative code paths chosen to elide memory operations. When a thread reaches a state where a speculative alternative exists, it executes that path as a best-effort hardware transaction. If that transaction fails for any reason, its effects are automatically discarded,

Within the transaction, the thread appears to execute as if it were running uninterruptedly, with no interleaved steps by other threads. From the other threads' viewpoints, the speculating thread appears to transition instantaneously from the path's initial state to its final state, hence the name "teleportation".

As discussed in the related work section, examples of mutually-canceling memory operations include lock acquisition and release [25, 26], short-lived reference counters with matching increments and decrements [11], and sequences of memory barriers that can be coalesced into a single barrier [2].

This paper makes the following contributions. First, we believe that identifying the teleportation design pattern itself is a useful addition to the lexicon, as it explains and unifies several earlier techniques from the literature, and it provides a way to organize and explain the new techniques described here. To be useful, a design pattern should help unify and explain existing designs (*e.g.*,"both lock elision and telescoping are instances of the

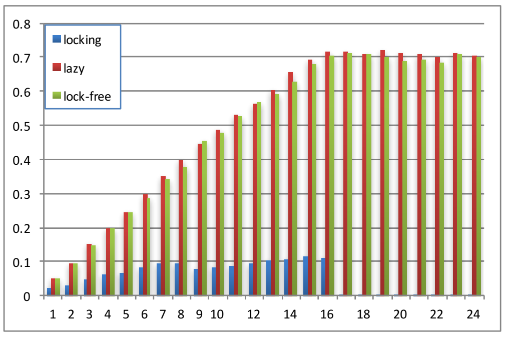

Figure 1 Throughput in Million Operations Per Second (MOPS) for list implementation without memory management.

teleportation pattern"), and it should suggest specific approaches to new problems (*e.g.*, "can we use the teleportation pattern to improve hazard pointers?"). It should be general enough to apply to disparate situations, but not so general as to be meaningless.

Next, we show how the teleportation pattern can be applied to retrofit speculative fast paths to well-known fine-grained locking algorithms such as *lock-coupling* [4] (or hand-over-hand locking). Teleporting from one lock-holding state to another avoids the memory traffic produced by locking and unlocking intermediate nodes.

Finally, we show how the teleportation pattern can be applied to retrofit speculative fast paths to hazard pointer-based memory management schemes [24]. Teleporting from one hazard pointer-publishing state to another, avoids the memory barriers needed to publish intermediate nodes' hazard pointers.

Experimental results on a 8-core Broadwell show that teleportation can substantially boost the performance of fine-grained locking and lock-free hazard pointer management. In some cases, the hazard pointer scheme with teleportation outperforms the same structure with a faster but less robust epoch-based reclamation scheme.

2 List-Based Sets

Pointer-and-node data structures come in many forms: skiplists, hashmaps, trees, and others. Here, for simplicity, we consider *list-based sets*, using data structures adapted from the Herlihy and Shavit textbook [19]. There are three methods: add(x) adds x to the set, returning *true* if and only if x was not already there, remove(x) removes x from the set, returning *true* if and only if x was there, and contains(x) returns *true* if and only if the set contains x. (We will discuss trees and skiplists later.)

A set is implemented as a linked list of nodes, where each node holds an item (an integer), a reference to the next node in the list, and possibly other data such as locks or flags. In addition to regular nodes that hold items in the set, we use two *sentinel* nodes, called LSentinel and RSentinel, as the first and last list elements. Sentinel nodes are never added, removed, or searched for, and their items are the minimum and maximum integer values.

We consider three distinct list implementations: the *lock-based* list, the *lazy* list, and the *lock-free* list. These three list algorithms are discussed in detail elsewhere [19].

Our *lock-based* list implementation employs *fine-grained locking* and is based on *lock coupling* [4], sometimes called *hand-over-hand locking*. Each list node has its own lock. A thread traversing the data structure first locks a node, then locks the node's successor, then releases the predecessor's lock. While fine-grained locking permits more concurrency than using a monolithic lock, it causes threads to acquire and release many locks, and restricts parallelism because threads cannot overtake one another.

By contrast, in the so-called *lazy list*, each list node has a Boolean marked field indicating whether that node's value is really in the set. Threads navigate through the list without acquiring locks. The contains() method consists of a single wait-free traversal: if it does not find a node, or finds it marked, then that item is not in the set. To add an element to the list, add() traverses the list, locks the target's predecessor, and inserts the node. The remove() method is lazy, taking two steps: first, mark the target node, *logically* removing it, and second, redirect its predecessor's next field, *physically* removing it.

Finally, the *lock-free* list replaces the lazy list's lock acquisitions with atomic compare-and-swap operations.

To establish a baseline for understanding the costs of synchronization and memory management, we ran a simple synthetic benchmark comparing the lock-based, lazy, and lock-free list implementations. We used an Intel Broadwell processor (Core D-1540) with RTM and HLE enabled, running at 2 GHz. The machine has a total of 8GB of RAM shared across eight cores, each having a 32 KB L1 cache. Hyper-threading was enabled, yielding a total of sixteen hardware threads. Threads were not pinned to cores.

When there are 8 or fewer threads, each thread has its own core. Between 9 and 16 threads, hyperthreading ensures that each thread has its own hardware context, but threads share caches. Beyond 16 threads, there are more threads than hardware contexts.

Each thread allocates nodes by calling the **new** operator, but for now nodes are not reused after being removed from the list. We use a simple "test-and-test-and-set" spin lock, where a thread spins reading the lock until it appears to be free, and then applies a compare-and-swap to try to acquire the lock. If that compare-and-swap fails, the thread temporarily releases the processor. To avoid false sharing, locks are padded so distinct locks occupy distinct cache lines.

The list implementations were compared using a simple synthetic benchmark based on the following parameters. List values range from zero to 10,000, and the list is initialized to hold approximately half of those values. All measurements are executed for one second. The number of threads varies from 1 to 24. Each time a thread calls an operation, it chooses contains() with a probability that varies from 0 to 100, and otherwise calls add() or remove() with equal probability.

All the experimental results reported here have the following structure. We ran experiments with 100%, 90%, 80%, 50%, and 0% contains() calls. Because the cost of list operations is dominated by navigation, all these mixtures produced similar curves. The graphs shown here reflect a mixture of 80% contains(), 10% add(), and 10% remove() calls. Each execution time shown in a graph represents an average over 10 executions.

Figure 1 compares the performance of the three list implementations on the benchmark *with no memory management*. This graph is a baseline allowing us to separate the costs of navigation (shown here) from the costs of memory management (shown later).

The lock-based list performs substantially worse than the other two because it acquires and releases a lock each time it traverses a node, while the others navigate without locks, acquiring locks or executing CAS operations only at the very end. Beyond 16 threads, there are more threads than hardware contexts, and fine-grained locking becomes extremely slow because some thread is always suspended while holding a lock, and while that thread is suspended, no active thread can overtake it.

In the next section, however, we will see that Figure 1, while informative, can be misleading, because none of these implementations performs any memory management.

3 Memory Management

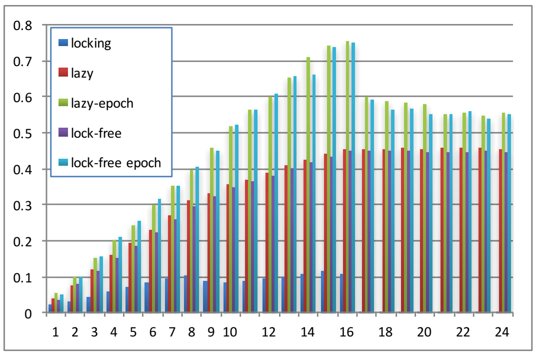

Figure 2 Throughput for list implementations with memory management

We now add memory management to the three list implementations considered in the previous section. To provide a common basis for comparison, we use the following simple structure. Each thread keeps a thread-local pool of *free* nodes available for immediate reuse, and a thread-local pool of *retired* nodes that have been removed from the list, but which may not be ready to be reinitialized and reused. Both pools are implemented as lists, but because they are thread-local, they do not require synchronization.

When a thread needs to allocate a node to add to the list, it first tries to take a node from its free pool. If the free pool is empty, it inspects the retired pool to determine whether there are any nodes that can be moved to the free pool. If not, it calls the C++ **new** operator to allocate a new node.

For the lock-coupling list, memory management is simple: as soon as a node is removed from the list, it can be placed directly in the free pool, bypassing the retired pool.

For the lazy and lock-free lists, however, a node cannot be reused immediately after it is retired because another thread, performing a lock-free traversal, may be about to read from that node, presenting a so-called *read hazard*.

Hazard pointers [24] provide a non-blocking way to address this problem. When a thread reads the address of the node it is about to traverse, it *publishes* that address, called a *hazard pointer*, in a shared array. It then executes a *memory barrier*, ensuring that its hazard pointer becomes immediately visible to the other threads. It rereads the address, and proceeds if the address is unchanged, As long as the hazard pointer is stored in the shared array, a thread that has retired that node will not reinitialize and reuse it. The memory barrier is the most expensive step of this protocol.

For the lazy and lock-free lists with hazard pointers, each thread requires two hazard pointers, one for the current node, and one for the previous node. To avoid false sharing, shared hazard pointers are padded so that hazard pointers for different threads reside on different cache lines.

For purposes of comparison, we also test an *epoch-based* reclamation scheme [12]. The threads share a global *epoch count* to determine when no hazardous references exist to a retired node in a limbo list. Each time a thread starts an operation that might create a hazardous reference, it observes the current epoch. If all processes have reached the current epoch, the global epoch can be advanced, and all nodes that were retired two epochs ago can safely be recycled. A thread starting such an operation blocks only if its own free list is empty, but there are retired nodes to be reclaimed when trailing threads catch up to the current epoch. Each thread needs to keep track of retired pools the last three epochs only. Careful signaling ensures that quiescent threads do not inhibit reclamation, but, as noted, an active thread that encounters a delay inside an operation can prevent every other thread from reclaiming memory.

Figure 2 shows the same benchmark for the same list implementations, except now with memory management. The lazy and lock-free lists with hazard pointers have become significantly slower because of the need to execute a memory barrier each time a node is traversed, an operation almost as expensive as acquiring a lock. The lock-coupling implementation is still slower than all, but it is significantly simpler than the other methods that require non-trivial memory management code. Epoch-based reclamation does not significantly slow down these data structures (but of course the "lock-free" list is no longer lock-free).

4 Teleportation

Teleportation for lock-based data structures works as follows: a thread holding the lock for a node launches a hardware transaction that traverses multiple successor nodes, acquires the lock only for the last node reached, and then releases the lock on the starting node before it commits. (For lock-free and lazy algorithms, the same observations hold, except replacing lock acquisition and release with hazard pointer publication.)

Compared to simple fine-grained locking, teleportation substantially reduces the number of lock acquisitions (or hazard pointer memory barriers) needed for traversals. Moreover, if a hardware transaction that attempts to teleport across too many nodes fails, the thread can adaptively try teleporting across fewer nodes, and in the limit, it reverts to conventional fine-grained locking manipulation.

4.1 Intel RTM

Although teleportation can be adapted to multiple HTM schemes, we focus here on Intel's *transactional synchronization extension* (TSX) [20], which supports hardware transactions through the *restricted transactional memory* (RTM) interface.

When a hardware transaction begins execution, the architectural state is checkpointed. Memory operations performed inside a transaction are tracked and buffered by the hardware and become visible to other threads only after the transaction successfully commits. The hardware detects memory access conflicts with other threads, both transactional and non-transactional. If a conflict is detected, one transaction *aborts*: its buffered updates are discarded and its state is restored.

In the RTM interface, a thread starts a hardware transaction by calling `xbegin`. If the call returns `XBEGIN_STARTED`, then the caller is executing inside a hardware transaction. Any other return value means that the caller's last transaction aborted. While inside a transaction, calling `xend()` attempts to commit the transaction. If the attempt succeeds, control continues immediately after the `xend()` call in the normal way. If the attempt fails, then control jumps to the return from the `xbegin` call, this time returning a code other than `XBEGIN_STARTED`. While inside a transaction, calling `xabort()` aborts the transaction. A thread can test whether it is in a transaction by calling `xtest()`.

4.2 Teleportation Algorithm

The heart of the teleportation algorithm is the `teleport()` method shown in Figure 3. (For brevity, we focus on lock-based teleportation.) This method takes three arguments: a

```
 1 Node* teleport(Node* start,
 2                T v,
 3                ThreadLocal* threadLocal) {
 4   Node* pred = start;
 5   Node* curr = start->next;;
 6   int  retries  = 10;
 7   while (--retries) {
 8     int  distance  = 0;
 9     if  (xbegin() == _XBEGIN_STARTED) {
10       while (curr->value < v) {
11         pred  = curr;
12         curr  = curr->next;
13         if  (distance++>threadLocal->teleportLimit)
14           break;
15       }
16       pred->lock();
17       start ->unlock();
18       _xend();
19       threadLocal->teleportLimit++;
20       return pred;
21     } else {
22       threadLocal->teleportLimit =
23             threadLocal->teleportLimit/2
24       if(threadLocal->teleportLimit<THRESHOLD)
25         threadLocal->teleportLimit=THRESHOLD;
26     }
27   }
28   curr->lock();
29   start ->unlock();
30   return curr;
31 };
```

Figure 3 Lock-based List: the teleport () method

starting node, the target value being sought, and a structure holding thread-local variables. The arguments must satisfy the following preconditions: the starting node must be locked by the caller, and the starting node's successor's value must be less than the target value. When the method returns, the starting node is unlocked, the result node whose address is returned is locked, and the result node's value is greater than the starting node's value, but less than the target value.

The thread-local teleportLimit variable is the maximum distance the method will try to teleport the lock. (In our experiments, we chose an initial value of 256, but the results were insensitive to this choice.) The method initializes pred to the start node, and curr to the start node's successor (Lines 4–5). It then enters a loop. At the start of each iteration, it starts a hardware transaction (Line 7), which advances pred and curr while curr−>value is less than the target value (Lines 10–15), stopping early if more than teleportLimit nodes are traversed (Line 14). On leaving the loop, the starting node is unlocked and pred is locked, teleporting the lock from one to the other (Lines 16–17). If the transaction commits (Line 18), then the limit for the next transaction is increased by 1 (Line 19), and the method returns pred.

If the commit fails, or if the transaction explicitly aborts, then the teleportation distance is halved, and the outer loop is resumed (Line 21). However, we do not let the distance fall below a threshold (we chose 20[1]), because short teleportation distances harm performance, especially in the tree algorithm. When too many failures occur (more than 10), the algorithm reverts to standard fine-grained locking (Lines 28–30).

As noted, this method follows an *additive increase, multiplicative decrease* (AIMD) strategy: the teleportation limit is incremented by 1 upon each successful commit, and halved upon each abort. We experimented with several other adaptive policies without noticing a dramatic difference. Overall, AIMD proved slightly better than the others.

Because atomic operations such as compare-and-swap are typically not permitted inside hardware transactions, The spinlock's lock() method calls xtest() to check whether it is in a transaction. If so, and if the lock is taken, then the method explicitly aborts its transaction. If, instead, the lock is free, it simply writes the value LOCKED and returns. Otherwise, if the caller is not in a transaction, then it executes a simple "test-and-test-and-set" lock as before.

When teleport() traverses a node, it reads the node's value and next fields, but not the node's spinLock field. (These fields are cache-aligned to avoid false sharing.) This omission means that one teleporting thread can overtake another, which is impossible with conventional fine-grained locking.

The teleport() methods for the lazy and lock-free lists are similar, omitting hazard pointer publication instead of locking.

4.3 How to teleport()

The lock-based remove() method, shown in Figure 4, sets pred to the local node (as discussed above) and enters a loop (Lines 5–19). If the pred node's successor's value is greater than the target value, then the value is not present in the list, so the method returns *false* (Line 6). If the pred node's successor's value is equal to the target value, then we lock the successor and unlink it from the list (Line 9). Finally, if the pred node's successor's value is less than the target value, we call teleport() to advance pred, and resume the loop (Line 16).

The tree-based set's range query method uses lock coupling. The range query method takes a pair (k_1, k_2) and returns the set of all items in the set between k_1 and k_2. Range queries are difficult for lists, but conceptually easy for unbalanced binary trees. Lock-coupling provides an easy snapshot view of the data structure, but it performs poorly because it acquires a lock for each traversed node, significantly reducing concurrency. Teleportation improves performance, while maintaining the simplicity and the functionality of the lock-coupling algorithm. The teleporting thread checks the traversed nodes' locks and aborts if any is locked. Since threads do not bypass locked nodes, locking a node freezes its subtree, allowing a snapshot of the key range.

The range query method starts by searching for the lowest-level node whose subtree contains all elements in the range. This search employs lock-coupling with teleportation. Once the lock of the subtree's root is acquired, the method gathered the nodes of the

[1] The exact value of constants used in our teleportation code had little effect on overall performance.

```
bool remove(T v, ThreadLocal* threadLocal) {
  Node* pred = &LSentinel;
  bool result = false;
  int retryCount = 0;
  while (true) {
    if (pred->next->value > v) {
      result = false;
      break;
    } else if (pred->next->value == v) {
      Node *nodeToDelete = pred->next;
      nodeToDelete->lock();
      pred->next = nodeToDelete->next;
      free(nodeToDelete);
      result = true;
      break;
    } else {
      pred = teleport(pred, v, threadLocal);
    }
  }
  pred->unlock();
  return result;
}
```

Figure 4 Lock-based list: remove() with teleportation.

subtree that are in the desired range one by one, making sure that previous operations have completed and released the nodes' locks. Finally, the subtree's root lock is released.

The teleportation algorithm for the range queries is similar to the list teleportation algorithm in Figure 3, except that it checks the visited node's lock and aborts if the lock is already held. In addition, it goes to the right if the current key is smaller than the maximum, to the left if the current key is larger, and otherwise it acquires the target's lock and commits.

4.4 An Optimization

One standard disadvantage of using fine-grained locking with a tree is that the lock of the tree root becomes a bottleneck, serializing operations that could otherwise be executed in parallel. To avoid high contention on the root, we add a simple, yet effective, optimization. A precondition to the teleportation method is that the node it receives is locked. Our optimization lets each thread have an artificial local node pointing to the sentinel first node of the data structure, e.g., the root of the tree. The artificial local node is never reclaimed and is always locked by the thread using it. A thread starts a search from its local node, and teleports from there. This eliminates contention on the first node and has several disadvantages. If two operations teleport on separate routes from the tree root node, they will not conflict. But even for a list, since the teleportation length varies between threads, there are multiple target nodes for the first teleportation, and contention is reduced. To see that correctness is preserved, note that the node returned by the first teleportation is either the sentinel node or a node reachable from it.

4.5 Performance Measurements

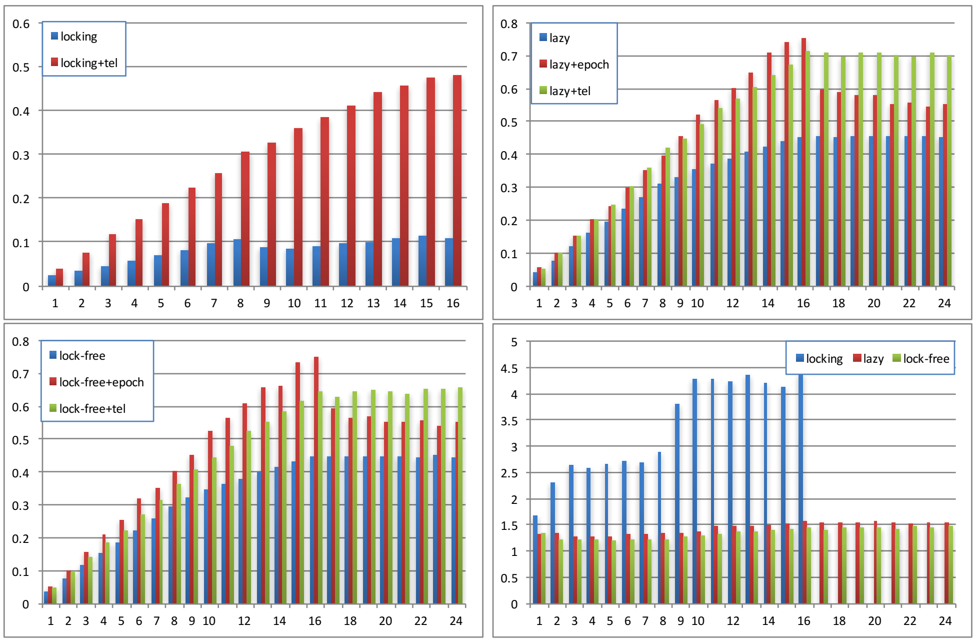

Figure 5 Effects of teleportation on various lists. Graphs (a)-(c) show throughput in million operations per second, higher is better. Graph (d) shows speedup, higher means teleportation is more effective.

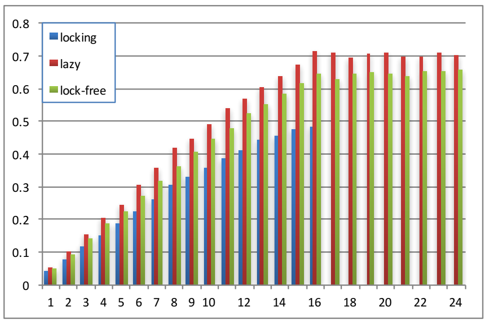

Figure 6 Throughput with teleportation

Figure 5a shows how teleportation improves the performance of the lock-based list implementation. For a single thread, teleportation yields an approximate 1.5-fold speedup, increasing to a 4-fold speedup at 16 threads.[2]

Figure 5b compares lazy list with various memory management schemes. It compares the throughput for (1) the lazy list with hazard pointers, (2) the lazy list with epoch-based management, and (3) the lazy list with hazard pointers and teleportation. Teleportation speeds up lazy list with hazard pointers by 33% – 64%. Teleportation with hazard pointers is just as fast as epoch-based reclamation, while eliminating the epoch-based scheme's

[2] Fine-grained locking becomes very slow beyond 16 threads, but fine-grained locking with teleportation works well beyond this limit because teleporting threads can overtake one another.

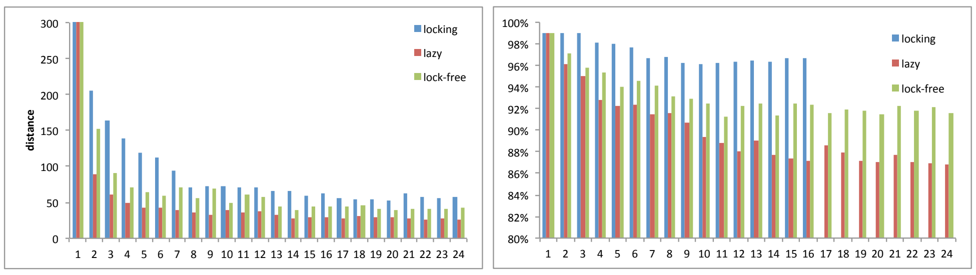

Figure 7 Average teleportation distance (L) and commit rates (R)

vulnerability to thread delay. It is slower by 6% for a single thread, faster by 0% – 5% for 2 – 8 threads, slower by 1% – 9% for 9 – 16 threads, and faster by 18% – 30% for 17 – 24 threads. We also experimented with adding teleportation to the epoch-based scheme, but we found that epoch-based performance is unaffected by teleportation because it does not perform memory barriers between most transitions.

Figure 5c compares lock-free list with various memory management schemes. It compares the throughput for (1) the lock-free list with hazard pointers, (2) the lock-free list with epoch-based management, and (3) the lock-free list with hazard pointers and teleportation. Teleportation speeds up the lock-free list with hazard pointers by 21% – 48%. When compared against epoch-based reclamation, teleportation is slightly slower at low thread counts (5% – 15%), and slightly faster (7% – 21%) above 16 threads.

Figure 5 provides a clear picture of the teleportation speedups in the above measurements. First, when applied to improve the performance of hand-over-hand locking; second, when applied to the hazard pointers with lazy list; and third, when applied to the hazard pointers scheme for the lock-free list.

Figure 6 compares the benchmark performances for the three list implementations, with memory management and with teleportation. The lazy implementation performs best for low thread counts - it is faster than the lock-free implementation by 7% – 13%. The locking implementation does not perform as well as lazy or lock-free lists, but it is easier to develop, maintain, and extend.

4.6 Transaction Measurements

Figure 7 presents the average distance (number of nodes) a lock or hazard pointer is teleported by each list algorithm, and Figure 7 shows the transaction commit rates. The lock-free implementation is able to teleport around 50 nodes and its commit rate is about 95%. The lazy implementation is able to teleport approximately 30 nodes at a transaction and the commit rate is around 88%. The locking implementation has the highest commit rate, around 96, and average teleportation distance around 70.

Although only small percentage of transactions abort, it is instructive to examine why. When a transaction aborts in RTM, it stores a condition code that indicates (more-or-less) why it aborted. In our benchmarks, we observed four kinds of abort codes.

- A *capacity* abort occurs if a transaction's data set overflows the L1 cache for any reason.
- A *conflict* abort occurs if one transaction writes to a location read or written by another transaction.
- An *explicit* abort occurs when the software calls xabort(). Here, this kind of abort means that a transaction tried to lock a lock already held by another thread.
- An *unknown* abort is any other kind of abort.

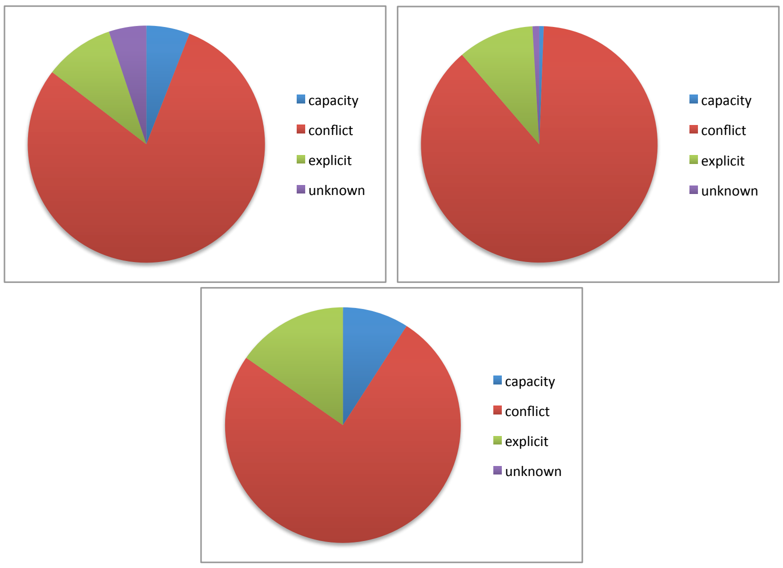

Figure 8 Why transactions abort: 4, 8, and 12 threads

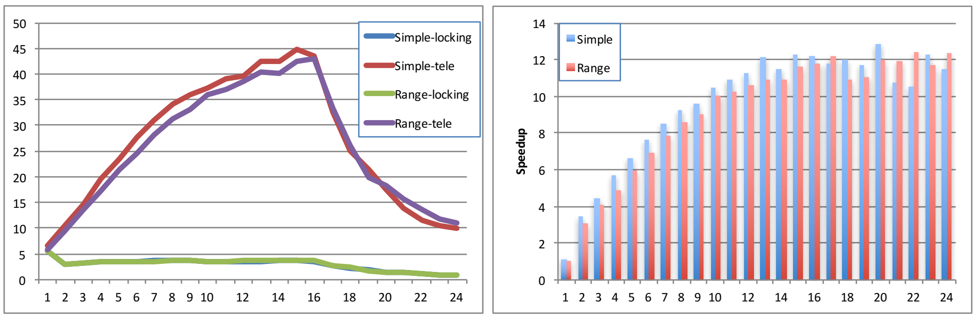

Figure 9 Throughput and speedup for binary tree with range queries.

Figure 8 shows the distribution of abort codes for the lock-based list teleportation, for 4, 8, and 12 threads. Most aborts are conflict and explicit aborts, which is a good result, meaning that aborts are caused mostly by real conflicts and not by hardware limitations or inaccuracies.

In addition to the linked list, we also implemented a skiplist based on the lazy list with teleportation, and a binary tree with range queries based on lock-coupling. For both implementations we used a micro-benchmark with a typical workload of 10% insertions, 10% deletions, and 80% contain operations. For the binary tree we further added a range workload, where the contain operations are replaced by range queries of length 10. In Figure 9 we report execution times for the binary tree with range queries support, implemented with the lock-coupling paradigm. It can be seen that the lock-coupling algorithm offers no scalability. However, with teleportation, excellent scalability is obtained. In Figure 9 we present the speedup of teleportation over the standard lock-coupling algorithm. With a single thread there is almost no improvement, but for two threads and up the speedup is 3.1 – 12.

Finally, in Figure 10 and Figure 10 we consider the speedup and execution time of a lazy skip list with and without hazard pointers teleportation. The teleportation speeds up the lazy implementation by 2% – 20%.

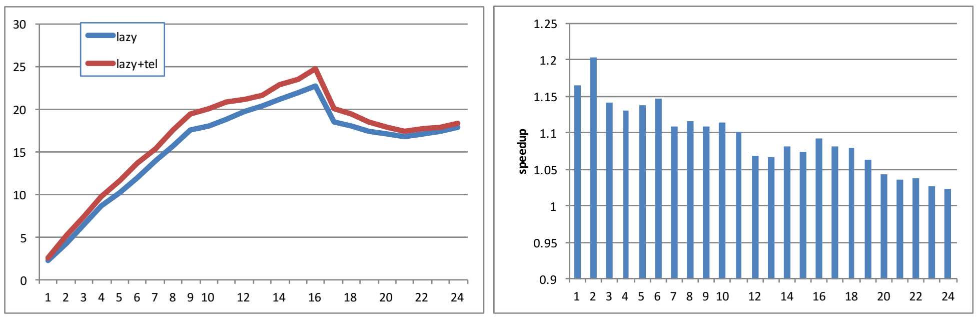

Figure 10 Throughput and speedup for skip list.

5 Correctness

Although a formal model of teleportation is beyond the scope of this paper, we can sketch what such a treatment would look like. There are multiple *threads*, where each thread is a *state machine* that undergoes a sequence of state *transitions*. The threads communicate through a (passive) shared *memory*. Each thread transition is associated with a *memory operation* such as a load, store, memory barrier, or read-modify-write operation. Such a model can be precisely captured, for example, using I/O Automata [21] or similar formalisms.

We write a thread state transition as $(s, m) \to (s', m')$, where s is a thread state and m is the corresponding memory state. An *execution* is a sequence of interleaved thread transitions.

Assume we are given a *base* system $\mathcal{B}$ encompassing threads $T_0, \ldots, T_n$ and memory M. From $\mathcal{B}$ we derive a *teleported* system $\mathcal{T}$ encompassing the same number of threads $T'_0, \ldots, T'_n$ and memory M. A map f carries each state of T'_i to a state of T_i such that

- every transition in $\mathcal{B}$ has a matching transition in $\mathcal{T}$: for every transition $(s, m) \to (s', m')$ of $\mathcal{B}$ there is a transition $(\hat{s}, m) \to (\hat{s}', m')$ of $\mathcal{T}$ such that $f(\hat{s}) = s$ and $f(\hat{s}') = s'$.
- every transition of $\mathcal{T}$ corresponds to some atomic sequence of transitions of $\mathcal{B}$: if $(s, m) \to (s', m')$ is a transition of $\mathcal{T}$, then there is a sequence of transitions of $\mathcal{B}$, $(s_i, m_i) \to (s_{i+1}, m_{i+1})$, $0 \le i \le k$, such that $f(s) = s_0, m = m_0$ and $f(s') = s_k, m' = m_k$.

From these conditions, it is not difficult to show that every execution of the teleported system $\mathcal{T}$ maps to an execution of the base system $\mathcal{B}$.

Informally, this model just says that checking correctness for a teleportation design requires checking that the thread and memory states in $\mathcal{T}$ after teleportation match the final memory states after an uninterrupted single-thread execution of $\mathcal{B}$.

6 Related Work

As noted, *Hardware lock elision* (HLE) [25, 26] is an early example of the teleportation pattern. In HLE, a critical section is first executed speculatively, as a hardware transaction, and if the speculation fails, it is re-executed using a lock.

Another prior example of teleportation is *telescoping*, proposed by Dragojevic et al. [11] Here, hardware transactions are used to traverse a reference-counted list without the cost of incrementing and decrementing intermediate reference counts.

StackTrack [2] (arguably) incorporates the teleportation pattern within a more complex design. It provides a compiler-based alternative to hazard pointers, in which hardware transactions are overlaid on the basic blocks of non-transactional programs. On commit, each such transaction publishes a snapshot of its register contents and local variables, effectively announcing which memory locations that thread may be referencing.

There are a number of related techniques that resemble our use of teleportation in some aspects, but either do not retrofit a viable prior design, or do not exist to minimize memory traffic.

Makreshanski *et al.* [22] examine (among other issues) the use of hardware transactions to simplify the conceptual complexity of lock-free B-trees by judicious introduction of multi-word CAS operations. Here, the speculative path executes the multi-word CAS as a hardware transaction, while the non-speculative fallback path uses a complex locking protocol. Although this does not really retrofit speculative short-cuts to prior design, and the overall goal is simplicity, not efficiency gain by reducing memory traffic.

BulkCompiler [1] proposes a novel hardware architecture where a compiler automatically parcels Java applications into hardware transactions (called "chunks"). Lock acquisition and release instructions within chunks can be replaced by memory reads and writes. Instead of retrofitting an existing design, BulkCompiler requires specialized hardware and a specialized compiler, while teleportation targets commodity platforms and compilers. BulkCompiler is monolithic, operating automatically on entire applications.

Hand-over-hand locking, also known as lock coupling, is due to Bayer and Schkolnick [4]. Lock-free linked-list algorithms are due to Valois [28], Harris [14], and Heller *et al.* [16].

Hazard pointers are due to Michael [24]. Other approaches to memory management for highly-concurrent data structures include *pass the buck* [17], *drop the anchor* [5], epoch-based techniques [14], and reference counting [10, 13, 27]. Hart *et al.* [15] give a comprehensive survey of the costs associated with these approaches.

Several recent studies aimed at reducing the costs of hazard pointers. The *Optimistic Access* scheme [9, 8] reduces the cost of memory barriers and hazard pointers by letting the threads optimistically read memory without setting hazard pointers. This scheme provides a mechanism that makes a thread aware of reading reclaimed memory and allow it to restart its operation. The optimistic access scheme can be applied automatically in a garbage-collection-like manner, and it fully preserves lock-freedom. *Debra* [6] extends epoch-based reclamation by interrupting delayed threads and making them restart. Optimistic access and Debra require the algorithm to be presented in a special form that allows restarting the operation when needed.

Threadscan [3] mitigates the cost of hazard pointers by avoiding memory barriers and interrupting all threads to gather their hazard pointers or their local pointers when needed. Threadscan requires substantial support from the runtime system and the compiler. The teleportation scheme presented in this paper does not require any special algorithmic form and it does not require compiler support.

Read-copy-update [23] is a memory-management mechanism optimized for low update rates, but does not permit fine-grained concurrency among updating threads.

7 Conclusion

State teleportation exploits best-effort hardware transactional memory (HTM) to improve of memory management for highly-concurrent data structures. Because modern HTMs do not provide progress guarantees, teleportation is designed to degrade gracefully if transaction commit rates decline, either because of contention or because the platform does not support HTM. In the limit, the data structures revert to their underlying memory management schemes.

In this paper we illustrated the use of teleportation for improving the performance of various data structures, including lock-based and lock-free lists, skiplists, and trees. Teleportation successfully lowers the costs associated with multiple lock acquisition and with memory barriers required by hazard pointers.

Naturally, there are limitations to be addressed by future work. Teleportation, like hazard pointers, must currently be applied by hand, just like the underlying locking or hazard pointers mechanisms. A future challenge is to automate this process.

7.0.0.1 Acknowledgements

The authors are grateful to Tim Harris for comments.

References

1 W. Ahn, S. Qi, M. Nicolaides, J. Torrellas, J.-W. Lee, X. Fang, S. Midkiff, and David Wong. Bulkcompiler: High-performance sequential consistency through cooperative compiler and hardware support. In *MICRO*, MICRO 42, pages 133–144, 2009. URL: `http://doi.acm.org/10.1145/1669112.1669131`, `doi:10.1145/1669112.1669131`.

2 Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. Stacktrack: An automated transactional approach to concurrent memory reclamation. In *EuroSys*, EuroSys '14, pages 25:1–25:14, 2014. `doi:10.1145/2592798.2592808`.

3 Dan Alistarh, William M Leiserson, Alexander Matveev, and Nir Shavit. Threadscan: Automatic and scalable memory reclamation. In *SPAA*, pages 123–132, 2015.

4 R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta Informatica*, 9:1–21, 1977.

5 Anastasia Braginsky, Alex Kogan, and Erez Petrank. Drop the anchor: Lightweight memory management for non-blocking data structures. In *SPAA*, SPAA '13, pages 33–42, 2013. URL: `http://doi.acm.org/10.1145/2486159.2486184`, `doi:10.1145/2486159.2486184`.

6 Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *PODC*, PODC '15, pages 261–270, 2015.

7 Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust architectural support for transactional memory in the power architecture. In *ISCA*, ISCA '13, pages 225–236, 2013. URL: `http://doi.acm.org/10.1145/2485922.2485942`.

8 Nachshon Cohen and Erez Petrank. Automatic memory reclamation for lock-free data structures. In *OOPSLA*, OOPSLA '15, pages 260–279, 2015.

9 Nachshon Cohen and Erez Petrank. Efficient memory management for lock-free data structures with optimistic access. In *SPAA*, SPAA '15, pages 254–263, 2015.

10 David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele, Jr. Lock-free reference counting. In *PODC*, PODC '01, pages 190–199, 2001.

11 Aleksandar Dragojevic, Maurice Herlihy, Yossi Lev, and Mark Moir. On the power of hardware transactional memory to simplify memory management. In *PODC*, pages 99–108, 2011.

12 Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, February 2004. URL: `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf`.

13 Anders Gidenstam, Marina Papatriantafilou, Hakan Sundell, and Philippas Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *TPDS*, 20(8):1173–1187, 2009. `doi:http://doi.ieeecomputersociety.org/10.1109/TPDS.2008.167`.

14 Tim Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, pages 300—314, 2001.

15 Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67(12):1270–1285, December 2007. URL: `http://dx.doi.org/10.1016/j.jpdc.2007.04.010`, `doi:10.1016/j.jpdc.2007.04.010`.

16 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, pages 3–16, 2005.

17 Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *DISC*, DISC '02, pages 339–353, 2002.

18 Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

19 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, March 2008. URL: `http://www.worldcat.org/isbn/0123705916`.

20 Intel Corporation. Transactional Synchronization in Haswell. Retrieved from `http:/http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/`, 8 September 2012.

21 Nancy Lynch and Mark Tuttle. An Introduction to Input/Output automata. Technical Memo MIT/LCS/TM-373, Massachusetts Institute of Technology, November 1988.

22 Darko Makreshanski, Justin Levandoski, and Ryan Stutsman. To lock, swap, or elide: On the interplay of hardware transactional memory and lock-free indexing. In *VLDB*, volume 8, No. 11, 2015. URL: `https://www.microsoft.com/en-us/research/publication/to-lock-swap-or-elide-on-the-interplay-of-hardware-transactional-memory-and-lock-free-indexi`

23 P.E. McKenney and J.D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *PDCS*, pages 509–518, 1998.

24 Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15:491–504, June 2004. `doi:10.1109/TPDS.2004.8`.

25 R. Rajwar and J. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *MICRO*, pages 294–305, 2001. URL: `citeseer.nj.nec.com/rajwar01speculative.html`.

26 R. Rajwar and J. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS*, pages 5–17, 2002.

27 Hakan Sundell. Wait-free reference counting and memory management. In *IPDPS*, IPDPS '05, pages 24.2–, 2005.

28 J. Valois. Lock-free linked lists using compare-and-swap. In *PODC*, pages 214–222, 1995. URL: `http://citeseer.nj.nec.com/valois95lockfree.html`.