StackTrack: An Automated Transactional Approach to Concurrent Memory Reclamation

Dan Alistarh [*]
MSR Cambridge
daalista@microsoft.com

Patrick Eugster [†]
Purdue University
p@cs.purdue.edu

Maurice Herlihy [‡]
Brown University
mph@cs.brown.edu

Alexander Matveev
MIT
amatveev@csail.mit.edu

Nir Shavit [§]
MIT and TAU
shanir@csail.mit.edu

Abstract

Dynamic memory reclamation is arguably the biggest open problem in concurrent data structure design: all known solutions induce high overhead, or must be customized to the specific data structure by the programmer, or both. This paper presents StackTrack, the first concurrent memory reclamation scheme that can be applied automatically by a compiler, while maintaining efficiency. StackTrack eliminates most of the expensive bookkeeping required for memory reclamation by leveraging the power of hardware transactional memory (HTM) in a new way: it tracks thread variables dynamically, and in an atomic fashion. This effectively makes all memory references visible without having threads pay the overhead of writing out this information. Our empirical results show that this new approach matches or outperforms prior, non-automated, techniques.

1. Introduction

For the time being, Moore's Law continues to make transistors smaller, but it has stopped making them faster. As a result, there is increasing pressure for shared data structures to scale under concurrency. Over the last two decades, researchers have developed efficient *non-blocking* [19] implementations for many classic data structures [13, 15, 21, 23]. This trend is being accelerated by the advent of *hardware transactional memory* (HTM) [18], recently available in commodity multiprocessors [1], which promises to simplify and speed up non-blocking data structures [8].

While such data structures are attractive for their high levels of concurrency, they make memory reclamation notoriously hard [4, 7, 10, 20, 22]. The difficulty is that in the absence of coarse-grained locks (or automatic garbage collection), it can be non-trivial to determine whether a thread has a reference to a node in the data structure hidden in a register, cache, or store buffer.

Existing schemes for *concurrent memory reclamation* fall into three rough categories. First, *quiescence-based schemes* [15, 16] reclaim memory whenever threads pass though a *quiescent* state in which no thread holds a reference to a shared node. These schemes are relatively lightweight, but their performance is significantly impacted by thread delays, which can prevent reaching quiescent states. Moreover, a thread crash can result in an unbounded amount of unreclaimed memory. *Reference counting* schemes [7, 14, 24], while easy to implement, have been observed to require expensive synchronization overhead [16]. Finally, *pointer-based* schemes, such as *hazard pointers* [22], *pass-the-buck* [20], or *drop-the-anchor* [4], explicitly mark *live* nodes (nodes accessible by other threads) which should not be de-allocated. The principal limitations of pointer-based schemes are that they must be customized to the data structure at hand, and that they add a critical *validation* step, which ensures that new memory accesses are exposed to reclaiming threads. Recent work [10] employed hardware transactional memory to speed up common components of known memory reclamation schemes. Overall, to our knowledge, all known techniques either require an often substantial implementation effort from the programmer, or induce performance overhead on data structure operations.

In this paper, we present a new technique exploiting hardware transactional memory for concurrent memory reclamation. We describe StackTrack, the first framework for concurrent memory reclamation that is both *automatic* and *efficient*. The key new insight is that we use HTM to solve the critical "invisible readers" problem: by having parts of

[*] Part of this work was performed while the author was a Postdoctoral Associate a MIT CSAIL, supported in part by NSF grant CCF-1217921, DoE ASCR grant ER26116/DE-SC0008923, and by grants from the Oracle and Intel corporations.

[†] Supported in part by DARPA grant N11AP20014 and NSF grant CNS-1117065.

[‡] Supported by NSF grant 1301924.

[§] Supported in part by NSF grants CCF-1217921 and CCF-1301926, DoE ASCR grant ER26116/DE-SC0008923, and by grants from the Oracle and Intel corporations.

EuroSys 2014 April 13 - 16 2014, Amsterdam, Netherlands

http://dx.doi.org/10.1145/2592798.2592808

the data structure operations be transactional, we ensure that reclaiming threads always perceive a *consistent view* of all variables accessed by other threads. We can then reduce memory reclamation to having the reclaiming thread scan the stack and registers of active threads for pointers to accessed memory locations.

To precisely explain how StackTrack works, we first need to define some terms. Hardware transactions are executed speculatively in processor caches. When a thread executes a transaction, it loads data into its cache, marks those cache lines as transactional, and does not write them back to memory. The data accessed by a transaction is called its *data set*. The native cache coherence protocol detects data conflicts with concurrent threads. If the transaction completes without encountering a data conflict, it *commits*, marking those lines as non-transactional, allowing them to be written back to memory. If a conflict occurs, the transaction *aborts*, and its tentative changes are discarded. A transaction can also restart because of a *capacity abort* that occurs if its data set overflows its cache, or if it runs for too long and a timer interrupt clears the cache.

Roughly speaking, StackTrack works as follows. Consider a correct non-blocking data structure implementation, and assume for now that each data structure operation is a single transaction. Before reclaiming an object, the reclaiming thread must ensure that that object is not in another thread's data set. Care must be taken to avoid the race condition where one thread adds an object to its data set at the same time another thread tries to reclaim it. This is where we exploit HTM: having data structure operations be transactional ensures that threads always observe a *consistent view* of memory. If a thread observes that an object is still live, it leaves that object alone. If a thread *correctly* observes that an object is no longer live, it reclaims it. If a thread observes that an object is no longer live, but it is still accessed inside an uncommitted transaction, then a data conflict will force that transaction to abort. In all cases, the reclaiming operation is correct. Note that the reclaiming operation does not need to be transactional.

This rough description lacks critical details. First, it is unrealistic to assume that an entire data structure operation can be placed inside a single hardware transaction because of capacity aborts. Second, it is not clear how a thread can efficiently announce that it has added an object to its data set. Methods such as hazard pointers are unappealing because they add to the transaction's cache footprint, increasing the likelihood of a capacity abort; moreover, they are not automatic.

To address the capacity problem, we introduce a technique which automatically splits the code into transactional *segments*. At compile time, the compiler injects a *split checkpoint* at the end of each basic block. At each checkpoint, the thread checks whether it should attempt to commit the current transaction and start another, or whether it should continue the current transaction for another basic block. Initially, we (optimistically) attempt to run the entire operation inside a transaction. Each time a transaction aborts, we lower the number of basic blocks in a segment, and if transactions repeatedly commit, we tentatively increase the number of basic blocks. Over time, this scheme converges to a segment length that matches the capacity of the hardware and the conflict level of the software. In the worst case, the scheme falls back to executing one operation at a time, just as in the original algorithm.

Next, we designed a novel scheme for tracking operations' data sets. Instead of having each thread explicitly announce each location it accesses, the reclaiming thread scans the *registers* (register contents) and *stack frames* of active threads, searching for references to the object to be reclaimed.[1] We batch the reclaiming thread's steps to amortize costs over the number of objects reclaimed. By contrast with pointer-based schemes, which require a memory fence each time a pointer is validated, StackTrack requires a memory fence only when a transactional segment commits.

StackTrack is designed to work with any best-effort HTM system [5], and reverts to a non-blocking, non-transactional mechanism if the HTM system fails. In this way, StackTrack uses the HTM system for good performance in the common case, and the non-transactional fallback for guaranteed progress in the worst case.

We implemented StackTrack in the C language, adding automatic memory reclamation to classic non-blocking queue, linked list, skip-list, and hash-table algorithms. These data structures were chosen to test StackTrack at different contention levels (queue vs. hash-table), and for different operation lengths (skip-list vs. queue). We ran tests on an 8-way Intel Haswell chip with 4 cores, each multiplexing 2 hardware threads. We compared the performance of StackTrack against schemes that implement quiescence, hazard pointers, and drop-the-anchor, as well as an uninstrumented implementation that did not reclaim memory at all. (Hazard pointers can be seen as an upper bound on the performance of reference-counting techniques.) The StackTrack scheme consistently outperforms hazard pointers (by up to 200%), and has an overhead of at most 50% in comparison to the uninstrumented implementation. (By contrast, in our tests, hazard pointers reduce throughput down to as little as 20%.) The quiescence technique scales as long as no threads are preempted, but has significantly decreased performance once context switches occur.

We found the main sources of overhead in StackTrack are aborts (contention aborts and capacity aborts), and the additional cost of instrumentation. Our empirical evaluation shows that the overhead caused by instrumentation and contention aborts increases linearly with the number of threads, as one would expect. On the other hand, capacity aborts in-

[1] We do assume that programs do not "hide" pointers to live objects. This is similar to the assumption made by conservative garbage collectors [2].

crease substantially once the number of threads surpasses the number of physical cores, as the onset of Hyperthreading increases cache pressure. These results suggest that Hyperthreading limits the scalability of HTM itself, but that these limits will be less apparent in future larger-scale HTM systems.

The rest of the paper is organized as follows. Section 2 describes relevant aspects of Haswell's HTM, while Section 3 gives an overview of related work. Section 4 recounts what we learned implementing concurrent memory reclamation using HTM, which may be of independent interest. Section 5.1 gives a high-level view of the system, while the technical details are presented in Sections 5.2–5.5. We discuss the properties of the framework in Section 5.6. Experimental results are presented in Section 6. Section 7 concludes with a discussion.

2. System Model and Problem Definition

In this section, we outline the system model, and overview the semantics of the Haswell TSX framework. There are n threads which communicate by reading and writing shared objects. Threads run at arbitrary speeds, can be subject to arbitrary delays, and may crash during the computation.

Threads share a correct non-blocking (lock-free) implementation of a data structure, which ensures global progress as long as *some* process takes steps. We now define the *concurrent memory reclamation* problem [20, 22]. A *node* is a memory range which can be viewed as a logical object in the data structure at a given time. The *roots* of the data structure are nodes which can be accessed by threads directly (e.g., through pointers to the data structure). A node is *accessed* at a given time if some thread holds a reference to it. A node is *reachable* at a given time if it can be accessed by following valid pointers through the data structure from one of the roots. A node is *removed* (or *unlinked*) if it is no longer reachable. A removed node may still be accessed by some thread. A node is *free* if its memory is available for reallocation. A thread is *reclaiming* if it is currently removing a node as part of the free; otherwise, the thread is *reading*. A node is *live* if it is accessed by some reading thread.

The data structure is assumed to provide a lock-free operation which removes nodes. In essence, memory reclamation schemes must transition nodes from the *removed* to the *free* state, ensuring that they are no longer live. An important technical note is that when the memory reclamation scheme is invoked, the node is *no longer reachable* in the data structure. Also, only *a single thread* may attempt to free a node (though several nodes may attempt to unlink it). These assumptions hold in most non-blocking data structures, and are standard for concurrent memory reclamation [20, 22].

Intel Transactional Synchronization Extensions (TSX) [1], define a best-effort HTM implementation, which allows the programmer to group operations into *transactions*, which are either applied atomically to memory (committed) or fail (abort). A transaction may abort for one of several reasons: *capacity*, suggesting that the transaction is too long; *contention*, suggesting that threads accessed memory concurrently; and because of hardware reasons.

3. Related Work

A significant amount of research has focused on practical solutions for concurrent memory reclamation, e.g. [7, 10, 16, 20, 22]. Hazard pointers (HP) [22] are probably the most widely-known solution. (A similar scheme, known as *pass-the-buck*, was proposed by Herlihy et al. [20].) Roughly, hazard pointers work as follows.

Each thread has a pool of pointers, known as *hazards* (or *guards* [22]), which it uses to mark objects as *live*—such objects should not be reclaimed, since they may still be accessed. Each thread also maintains a local buffer of nodes which should be reclaimed. Regularly, the thread checks, for each node in the buffer, if there are hazards by any other thread pointing to the node. If there are no such hazards, the node can be safely reclaimed. The critical interval is between the point when the thread obtains a reference to an object, and the point when the hazard pointer to the object is visible to all other threads, as the node may be reclaimed during this interval. Crucially, the thread must re-validate the pointer before proceeding, and a memory fence has to be inserted between the initial step and the validation. These additional fence instructions are necessary even on relatively strong memory models (e.g., TSO), and induce significant overhead, as can be seen in our experiments.

Another approach, e.g. [7, 14, 24], employed *reference counts* to track the accesses of each node. Each node is associated with a counter, which is incremented/decremented by threads as they access/drop references to the node. Updates need to be atomic, therefore instructions such as fetch-and-add or double-compare-and-swap (DCAS) must be employed. A significant drawback of such schemes is the performance overhead of maintaining consistent reference counts [16].

Harris [15] proposed a per-thread timestamping scheme for memory reclamation in a concurrent list, which relies on scheduler support to guarantee progress. Hart et al. [16] performed a systematic comparison of several memory management techniques, such as hazard pointers, reference counters, and *quiescence/epoch*-based reclamation [12]. (In these latter schemes, memory can only be reclaimed when all threads have passed through a *quiescent* state, in which no thread holds any reference to shared nodes. Harris's timestamping scheme [15] can be seen as an instance of quiescence-based reclamation. Epoch and quiescence-based schemes are similar, the main difference being that the former usually enforce quiescent states to occur when threads complete their operations.) Hart et al. [16] found that schemes such as hazard pointers and reference counters suffer from a performance penalty because of the additional

synchronization operations, which are required even in the case of read-only operations. By contrast, quiescence-based techniques perform better in the absence of thread delays. On the other hand, if some threads are slow (preventing a quiescent state), performance may be affected; moreover, if a thread crashes, an unbounded amount of memory might not be reclaimed.

Recently, Braginsky et al. [4] proposed an efficient scheme which builds on timestamping and on hazard pointers. In brief, each thread has a timestamp which tracks its activity, and an *anchor pointer*, which records a new reference only once every several node accesses. To recover from a thread crash, a complex technique called *freezing* [3] is used, in which threads co-operate to replace part of the data structure, restoring their ability to free memory. The key to performance is that the expensive anchor and freezing operations are used rarely, and do not affect the "fast path" of the construction. Tests show that this scheme can improve throughput significantly with respect to hazard pointers, by up to 500% for a concurrent list on an AMD Opteron machine. On the other hand, the technique is complex, and relatively difficult to implement for arbitrary data structures. To our knowledge, the only implementation currently available is for a linked list.

From the complexity standpoint, reference counters are arguably the simplest reclamation technique, and can probably be automated. (Note however that some reference counting schemes [24] impose limitations on the compatibility between the memory reclamation and the memory allocation mechanism.) However, this is also the solution with the highest performance overhead [16]. A common shortcoming of other non-blocking techniques is that they cannot be automated [22], as they require a good understanding of the underlying implementation. Specifically, for hazard pointers, the programmer must know which accesses may generate a hazard: omitting a hazard can lead to segmentation faults, whereas placing superfluous hazards induces a performance penalty. In contrast, our scheme is also non-blocking, but is *automatic*.

In a recent paper, Dragojevic et al. [10] explored the power of HTM to simplify memory reclamation. In particular, they focused on using HTM to speed up the *dynamic collect* problem, which is a common component of several known reclamation techniques. They showed that HTM enables simpler and faster implementations for dynamic collect, which in turn leads to faster memory reclamation. By contrast, we propose a new scheme which exploits the bookkeeping capabilities of HTM explicitly.

There has also been significant work on parallel / multi-threaded garbage collection, for example [11, 17]. Generic concurrent garbage collection offers a functional super-set of memory reclamation, likely to be more complex and expensive to implement, and is beyond the scope of this paper.

4. Observations

We now outline some observations made while examining the performance of concurrent memory reclamation using HTM. These observations may be of independent interest.

Extra cache lines are costly. An important performance predictor for transactional code is the number of cache lines it accesses. This is because of the generated cache traffic, but also, more importantly, because accessing too many cache lines causes the hardware transaction to be aborted due to a capacity abort. In particular, memory reclamation schemes which need to add update operations to the code will suffer a performance penalty when combined with transactions.

Transactions must be short. A consequence of the previous observation is that hardware transactions must be *short*. Transactions that access more than a small number of cache lines, filling the L1 cache, will be aborted, leading to a significant performance penalty. Moreover, a long transaction is also more likely to generate contention, which can also cause an abort. In general, short transactions are significantly more likely to commit.

Take advantage of HTM bookkeeping. Any scheme for concurrent memory reclamation must maintain, in some form, a consistent record of the locations accessed (or about to be accessed) by the active threads. A useful observation is that HTM already maintains such a consistent record for each thread. In particular, a series of accesses performed inside a hardware transaction may only read from locations that have not been freed concurrently: otherwise, the transaction is automatically aborted, since all transactional operations are guaranteed to be executed atomically.

5. The StackTrack Framework

5.1 Overview

To free an object, a reclamation algorithm must first ensure that no thread holds references to it. References to objects accessed by a thread are loaded to its stack variables and registers throughout its execution. Hence, whenever attempting to free an object, our algorithm performs a global scan which inspects the stack and registers for every thread in the system, searching for the address it wants to free. The address is also checked against information about dynamically-allocated objects.

Consistent views. The obvious problem with this idea is that each thread is constantly updating its stack variables, and its registers are not visible to the other threads. We need to expose an atomic view of the current stack and registers for the global scan operation, so that a scan cannot miss a reference. This is easy if each operation is part of a single transaction, since then the view of memory is always consistent. However, long operations might not always "fit" inside transactions. Our solution is to split each operation into a series of hardware transactions, and to expose the

Algorithm 1 StackTrack: free functions.

```
1:  procedure FREE(ctx, ptr_free)
2:      INSERT(ctx.free_set, ptr_free)
3:      if SIZE(ctx.free_set) > max_free then
4:          SCAN_AND_FREE(ctx)
5:      end if
6:  end procedure
7:
8:  procedure SCAN_AND_FREE(ctx)
9:      for all ptr_free ∈ ctx.free_set do
10:         ptr_free ← ctx.free_buf[index]
11:         is_found ← false
12:         for thread_id = 0 → max_threads do
13:             th_ctx ← activity_array[thread_id]
14:             oper_cnt_pre ← th_ctx.oper_counter
15:             start inspect:
16:             htm_cnt_pre ← th_ctx.splits
17:             if IS_IN_STACK(th_ctx, ptr_free) then
18:                 is_found ← true
19:             end if
20:             if IS_IN_REGISTERS(th_ctx, ptr_free) then
21:                 is_found ← true
22:             end if
23:             htm_cnt_post ← th_ctx.splits
24:             oper_cnt_post ← th_ctx.oper_counter
25:             if oper_cnt_pre = oper_cnt_post then
26:                 if htm_cnt_pre ≠ htm_cnt_post then
27:                     goto start inspect
28:                 end if
29:             end if
30:         end for
31:         if ¬is_found then
32:             HEAP_FREE(ptr_free)
33:             DELETE(ctx.free_set, ptr_free)
34:         end if
35:     end for
36: end procedure
37:
38:
39: function IS_IN_STACK(ctx, ptr_free)
40:     for all frame ∈ ctx.stack_frames do
41:         addr ← frame.start_addr
42:         end_addr ← frame.end_addr
43:         while addr ≤ end_addr do
44:             if PTRS_EQUAL(addr, ptr_free) then
45:                 return true
46:             end if
47:             addr ← addr + 1
48:         end while
49:     end for
50:     return false
51: end function
52:
53:
54: function IS_IN_REGISTERS(ctx, ptr_free)
55:     for all reg ∈ ctx.registers do
56:         if PTRS_EQUAL(addr, ptr_free) then
57:             return true
58:         end if
59:     end for
60:     return false
61: end function
```

current values of the registers before every commit. As a result, global scans always perceive a consistent view of the registers and the stack, for every thread.

Splitting operations. We split the operation into a series of hardware transactions by injecting "split checkpoints" into the code, which we use to manage the start and commit points of the hardware transactions. To have a reasonable number of split checkpoint calls, the algorithm injects exactly one checkpoint call per basic code block. (Recall that a basic code block is a sequence of instructions without a branch instruction.) This process is automatic, since the basic code blocks are known to the compiler.

When an operation executes, it first starts a hardware transaction, and invokes the split checkpoint function on every basic code block start. The split checkpoint counts the number of basic code blocks executed so far, and, when reaching a predefined split limit, it commits the current hardware transaction and starts a new one. When the data structure operation is completed, it commits the last hardware transaction.

To achieve good performance, it is essential to optimize the split lengths. Making them too long results in excessive hardware aborts; making them too short results in high overhead. To address this problem, we construct a dynamic split length predictor, which "learns" the behavior of each operation, and adjusts the split lengths accordingly.

Length predictor. We define a dynamic split length predictor for every possible split transaction inside each operation. For example, if we have a skip-list data structure with search, insert, and delete operations, and each operation is split into hardware transactions at some point in the execution, then each hardware transaction has a unique dynamic split length predictor.

Every dynamic split length predictor first tries to execute the split transaction with a large split length (say, 50 basic code blocks). If the hardware transaction aborts too often, then it reduces the length by one block, and restarts the

transaction. This is iterated until the right split length is found, and the hardware commits successfully. In a similar way, if the hardware transaction commits often, the predictor speculates that it may succeed with a higher split length, and increases the length by one block.

Transactional operations. In sum, the scheme takes all the regular data structure operations and makes them transactional. (In practice, operations are split into shorter hardware transactions for performance reasons, however each can be seen as a single transaction for the purposes of memory reclamation.) Operations which reclaim memory are the exception. These operations are transactional only until the point where they call the FREE() procedure, which our scheme implements. The FREE() procedure, which scans the thread stacks and registers, does not need to be transactional, and multiple instances can be active at once. Notice that, by the time the FREE() procedure is called, each target node has already been made *unreachable* by the already executed code of the corresponding data structure operation, and that at most one thread may attempt to free a specific node.

We now describe the key elements of the scheme in more detail. We first present the core mechanism, and then describe the slow path and removal batching.

Algorithm 3 StackTrack: split example

```
1:  function REDBLACK_TREE_SEARCH(root, key)
2:      op_id ← 1
3:      SPLIT_INIT(ctx, op_id)
4:      SPLIT_START(ctx)
5:      node ← root
6:      while node ≠ null do
7:          SPLIT_CHECKPOINT(ctx)
8:          if node.key = key then
9:              SPLIT_CHECKPOINT(ctx)
10:             return node
11:         end if
12:         if node.key < key then
13:             SPLIT_CHECKPOINT(ctx)
14:             node ← node.left
15:         else
16:             SPLIT_CHECKPOINT(ctx)
17:             node ← node.right
18:         end if
19:     end while
20:     SPLIT_COMMIT(ctx)
21:     return null
22: end function
```

5.2 The Free Procedure

The algorithm associates a thread-local context structure, called ctx, with every thread. The context holds thread specific information, and the StackTrack uses it to inspect the current thread stack and registers. Whenever accessing the data structure, each thread registers itself into a global $activity\_array$ data structure, and deregisters when it completes its operations. Intuitively, the activity array allows each active thread to be "found" by other threads. In particular, whenever attempting to reclaim an object, a thread identifies the active threads in the system by scanning this global array, and inspects their stack and registers.

Algorithm 1 shows the structure of the FREE() procedure (lines 2-5). It adds the free requests to a local $free\_set$ buffer. When this set reaches a predefined size, it invokes the SCAN_AND_FREE() procedure which inspects the stacks and registers of the other threads, and frees the objects which are not pointed to by any reference (lines 9-35).

The IS_IN_STACK() and IS_IN_REGISTERS() functions implement the inspection of the stack and registers for a thread, respectively. For the stack, the algorithm scans the stack frames of the thread word-by-word, searching for a pointer value that is equal to the pointer to be freed ptr_free. For the registers, it traverses the registers in the thread context ctx.

The update of the stack and registers in the ctx of a thread is atomic, however, the scan of the stack and registers is not atomic. Therefore, a scan is consistent if there was no concurrent update to the stack and registers during the scan. In other words, there was no subsequent HTM successful commit of the next split segment. To achieve this, we use the split counter of the thread, which updates with every HTM split segment commit, and read it before inspecting the thread stack and registers and after we did it. If the split counter has changed, then we restart the scan of this thread. Else, the scan is consistent and we can continue to the next thread.

At first glance, a restart of a thread scan seems to be a blocking operation, that may restart infinitely. But, a restart of a thread scan means that the thread made a successful HTM commit of its current split segment. So, this means that some thread progresses. Therefore, eventually the thread will complete its current operation, in which case we are no longer required to scan its stack and registers. We identify completed operations by using the *oper_counter* of the thread, and read it before the thread scan and after. If it has changed, then we don't need to restart the scan of the thread and we continue to the next thread.

Notice that the scan may result in false positives, where a value that is not a real pointer is equal to the free pointer. However, this does not effect correctness, and such false positives are highly unlikely, since the heap pointers have specific structure and values. (We have not observed false positives during our experiments.)

Free procedure optimization . The current implementation traverses the stacks of the threads for every pointer in the free buffer. This is relatively inefficient; it is in fact possible to optimize this procedure in a similar way as for hazard pointers. We can scan the stacks only once, hashing all of the

Algorithm 2 StackTrack: split functions.

```
1:  procedure SPLIT_INIT(ctx, op_id)
2:      ctx.splits ← 0
3:      ctx.op_id ← op_id
4:      memory_fence()
5:  end procedure
6:
7:  procedure SPLIT_START(ctx)
8:      ctx.steps ← 0
9:      ctx.limit ← ctx.limits[ctx.op_id][ctx.splits]
10:     while HTM_START()≠ sucess do
11:         MANAGE_SPLIT_ABORT(ctx)
12:         ctx.steps ← 0
13:         ctx.limit ← ctx.limits[ctx.op_id][ctx.splits]
14:     end while
15: end procedure
16:
17: procedure SPLIT_CHECKPOINT(ctx)
18:     ctx.steps ← ctx.steps + 1
19:     if ctx.steps ≥ ctx.limit then
20:         SPLIT_COMMIT(ctx)
21:         SPLIT_START(ctx)
22:     end if
23: end procedure
24:
25:
26: procedure SPLIT_COMMIT(ctx)
        ▷ Expose can be omitted on final commit
27:     EXPOSE_REGISTERS(ctx)
28:     ctx.splits ← ctx.splits + 1
29:     HTM_COMMIT()
30:     MANAGE_SPLIT_COMMIT(ctx)
31: end procedure
32:
33: procedure EXPOSE_REGISTERS(ctx)
34:     for all ⟨reg_name, reg_value⟩ ∈ registers do
35:         ctx.registers[reg_name] ← reg_value
36:     end for
37: end procedure
38:
39: procedure MANAGE_SPLIT_COMMIT(ctx)
        ▷ If current split succeeded 5 times in a row, increase its limit by 1 (ctx.limits[ctx.op_id][ctx.splits] += 1 )
40: end procedure
41:
42: procedure MANAGE_SPLIT_ABORT(ctx)
        ▷ If current split failed 5 times in a row, then its limit by 1 (ctx.limits[ctx.op_id][ctx.splits] -= 1 )
43: end procedure
```

pointers in the stacks to a hash table, and then traverse the free buffer pointers and compare them to the hash table associated entries. This will result in an average constant time work per a free pointer instead of all stacks scan per such a pointer. (We note that, in our empirical evaluation, this optimization did not give a significant performance advantage, because the cost of the free procedure scan is amortized over the free calls; it executes only once per predefined number of free invocations.)

5.3 The Split Procedure

We execute each operation as a series of hardware transactions, which we call *segments*. The algorithm manages the start and commit points of segments by calls to split checkpoints, which are inserted in the code at every basic code block start. We allow each segment to have a dynamic length, by maintaining private split information for each segment. In this way, we ensure that the split dynamically adapts to the behavior of the code. In order to achieve this, we assign a unique id per operation, and a unique segment number which identifies a specific split inside an operation, such that the combination of the two uniquely defines a segment.

Algorithm 3 shows an example of applying these functions to the red-black tree search operation. (We employ the red-black tree as a running example, since it generates short code blocks, which best illustrate the instrumentation.) First, it calls the SPLIT_INIT() function with the unique operation id. It then starts the first hardware transaction by a call to the SPLIT_START() function. During the execution, the code calls to SPLIT_CHECKPOINT() on every basic code block start, and on operation finish, it commits the last hardware transaction by a call to SPLIT_COMMIT().

Algorithm 2 shows the implementation of the procedures SPLIT_INIT(), SPLIT_START(), SPLIT_CHECKPOINT(), and SPLIT_COMMIT(). These split procedures depend on a group of local variables: $ctx.op\_id$, which holds the current operation id; $ctx.splits$, which counts the current number of segments, $ctx.steps$, which counts the current number of basic code blocks, and $ctx.limit$, which defines when the checkpoint call will attempt to commit the current split. Notice that the combination of operation id and split number uniquely defines the current segment, therefore $ctx.limits[ctx.op\_id][ctx.splits]$ holds the length for the current segment.

On split init, the algorithm sets $ctx.op\_id$ to the unique operation id and the $ctx.splits$ to 0. On split start, it sets $ctx.steps$ to 0 and $ctx.limit$ to the current value of the variable $ctx.limits[ctx.op\_id][ctx.splits]$, which defines the unique length of the current segment. It then starts the hardware transaction by a call to HTM_START(). If a hardware abort occurs, the algorithm calls MANAGE_SPLIT_ABORT(),

which may change the length of the current segment, and updates the value of the $ctx.steps$ and $ctx.limit$.

On split checkpoint, it increments $ctx.steps$ by 1, and checks it against the $ctx.limit$. If the limit is reached, then it attempts to commit the current split transaction, and starts a new one. The split commit first exposes the current registers of the thread, by writing their values to $ctx.registers$, and then actually performs the hardware commit. After a successful commit, it calls the MANAGE_SPLIT_COMMIT() procedure, which may change the current split transaction limit, and then increments $ctx.splits$ by 1, to indicate a move to the next transaction split.

The transaction split length limit is controlled by calls to MANAGE_SPLIT_ABORT() and MANAGE_SPLIT_COMMIT(). Currently, we decrease the limit by one if we get five consecutive capacity aborts, and increase the limit by one if we get five consecutive successful commits. We found that this simple local scheme results in good performance. More complex schemes can be implemented to obtain better control and faster adjustment of the transaction split lengths, but this is a general problem that is not the focus of the current paper.

5.4 The Fallback Mechanism

Hardware aborts. Our algorithm executes data structure operations as a series of hardware transactions. Current Intel and IBM HTM implementations provide a best-effort HTM design, meaning that there is no progress guarantee for the hardware transactions: they may abort due to many hardware-related reasons, and there is no guarantee that a transaction commits.

The most likely reasons for HTM aborts are contention and cache capacity limitations. The algorithm is able to handle both of these common reasons dynamically by its automatic adjustment of the lengths of the segments. However, other (less likely) reasons for abort exist, such as unsupported instructions that simply cannot execute in a hardware transaction. For this case, if the unsupported instruction is not related to a local variable or register update (required for StackTrack correctness), then the we can handle it by committing the current hardware transaction, executing the unsupported instruction, and starting a new hardware transaction. For any other case, we are not able to use the HTM system, and must fall back to a software-only StackTrack slow-path version of the operation, described below.

Slow-path fallback. The slow-path version of the operation implements a software-only extension of the "hazard pointers" scheme. It instruments every shared memory read and write with a call to SLOW_READ() and SLOW_WRITE() procedures, which add the memory location to a reference set of the thread. In addition, it adds a call to SLOW_START() and SLOW_COMMIT() on operation start and finish, where the SLOW_COMMIT() resets the reference set of the thread to an empty set. The goal of this design is to maintain compatibility with the GCC TM compile-time automatic instrumentation, which can be used for automatic generation of the slow-path versions for data structure operations.

Algorithm 4 StackTrack: slow-path example

```
1:  function RB_TREE_SEARCH_SLOW_PATH(root, key)
2:      op_id ← 1
3:      SLOW_INIT(ctx, op_id)
4:      SLOW_START(ctx)
5:      node ← SLOW_READ(&(root))
6:      while node ≠ null do
7:          SLOW_CHECKPOINT(ctx)
8:          if SLOW_READ(ctx, &(node.key)) = key then
9:              SLOW_CHECKPOINT(ctx)
10:             return node
11:         end if
12:         if SLOW_READ(&(node.key)) < key then
13:             SLOW_CHECKPOINT(ctx)
14:             node ← SLOW_READ(&(node.left))
15:         else
16:             SLOW_CHECKPOINT(ctx)
17:             node ← SLOW_READ(&(node.right))
18:         end if
19:     end while
20:     SLOW_COMMIT(ctx)
21:     return null
22: end function
```

We now describe the fallback mechanics, and how they interact with the fast path. We generate the HTM fast-path and the software-only slow-path version for every data structure operation. In both of these paths, we inject the split checkpoint calls on every basic code block start, and use them to match the fast-path and the slow-path versions of the split segments. In this way, when a fast-path split hardware transaction constantly fails, the associated split checkpoint procedure detects it and performs a jump to the corresponding split checkpoint procedure in the slow-path. As a result, the slow-path can continue the execution in software from the point where the fast-path has failed.

Algorithm 4 shows an example of the slow-path version of the red-black tree search operation. Comparing it to the fast-path example in Algorithm 3, we see that the shared memory accesses are instrumented with calls to SLOW_READ(), which perform the reference set maintenance, and we can see that the SLOW_INIT(), SLOW_START() and SLOW_CHECKPOINT() procedure placement corresponds to the corresponding placement of split procedures on the fast path. As a result, a fast-path split checkpoint procedures can always jump to the corresponding slow-path checkpoint procedure. Algorithm 5 shows the implementation of the slow-path functions. For clarity, we omit the slow-path checkpoint policy code and the fast-path to slow-path jumping code, and show only the reference set maintenance code.

Slow-path algorithm. The fallback tracks references using a reference set for every thread. (See Algorithm 5 for pseudocode.)

On shared memory read, the SLOW_READ() call reads the location, adds it to a reference set of the thread, performs a memory fence, and reads the address again, in order to verify that it has not been changed in the meantime. If the reference has changed, it restarts the slow read operation. (Note that a restart implies that some other thread made progress.) The memory fence and subsequent re-read of the location ensure that a possible concurrent global scan operation will synchronize correctly, and detect an update to the reference set of the thread.

On shared memory write, the SLOW_WRITE() call first performs the SLOW_READ() of the location to ensure that it has been recorded in the reference set, and then performs the write. When the data structure operation completes, the SLOW_COMMIT() resets the reference set to an empty set, by removing all of the entries.

Note that this mechanism is less efficient than hazard pointers, since it treats all of the shared reads and writes as hazardous, and handles them as such. However, the high cost of the fallback is not an issue for our scheme, since it is only used as an unlikely backup.

Finally, the StackTrack global scan procedure uses a global $slow\_path\_counter$ to detect if there are threads executing the fallback. Every thread atomically increments this counter on slow-path start and decrements it on slow-path finish. The StackTrack global scan checks that this counter is 0 before the scan start. If the counter is not 0, then in addition to the inspection of the stacks and the registers, it also inspects the reference sets of the threads.

5.5 Automatic Code Transformation

To apply our scheme, the programmer must simply indicate to the compiler the functions which require reclamation, and use the StackTrack FREE() procedure to reclaim memory. The compiler then adds the HTM split logic, injecting split checkpoint calls on every basic code block start, and generates the slow-path version of the operations, which requires to instrument every shared memory read and write with a call to the slow-path fallback read and write.

At runtime, the split predictors automatically adjust the lengths of the different segments to optimize the HTM commit rate. If some segment length reduces to one basic code block, and still fails more than a predefined number of times, then the algorithm executes the matching code block from the software-only slow-path fallback.

We note that none of the actions described above require any data structure specific information. Moreover, the GCC compiler already supports automatic TM instrumentation of shared memory accesses, which can be used to generate the slow-path versions for the operations. The only significant addition to the code is the split checkpoint call per basic code block, which is performed by the compiler.

Limitations. Importantly, we note that our scheme does not cover the case where the programmer stores pointers to variables which might be accessed in the heap, and not in the threads' stack. Another issue is that of code pointer calculations which may "hide" pointer values from the free procedure that scans for them. General solutions to this problem are not known, since programmers may employ unusual techniques (for example, one may hide pointer values by XOR-ing). Conservative GC schemes have the same problem [2]. However, there are reasonable pointer arithmetic operations for array and structure accesses that our scheme supports. For example, consider an array object starting at address 0x1234 and having 20 entries of one byte each. The code may load 0x1234 to a local variable ptr, calculate 0x1234 + K in order to get access to the K-th element of this array, and then store 0x1234 + K to variable ptr. A thread attempting to free the array performs a scan for the array's starting address 0x1234 in the stacks and registers of the other threads. This scan might miss the value 0x1234 + K in the local variable ptr. The solution is to use the heap information about the allocated objects. All dynamic allocations go through the heap, therefore we hook on the allocation function (for example, in the GNU C library, we use the $\_\_malloc\_hook$ variable), and store the start address and the length of every object in a data structure supporting range queries. Then, for any two pointers, we can identify if they point to the same object by making a range query into this data structure. In this way, the FREE procedure can detect hidden pointers to arrays or structures. We note that in some settings, this information is already present in the malloc data structures.

Programmer-defined transactional regions. Using StackTrack does not preclude the programmer from defining her own transactional regions in the code. In particular, the split procedure adapts to this case by ensuring that a split is never performed during a user-defined transaction. To ensure correctness, the split procedure does have to insert the necessary register expose operations at the end of the user-defined transaction. This may increase the abort rate of the user-defined transactions. At the same time, since the HTM is best-effort, the programmer must still provide a non-transactional backup execution alternative for the transactional code. This backup is not provided by StackTrack.

5.6 Correctness

Progress. The fallback mechanism ensures that data structure operations remain non-blocking even in the case when not all instructions can be executed by the hardware transactional memory system. (In particular, the HTM may constantly fail if the original code uses external libraries or I/O operations.) It is straightforward to check that the backup is non-blocking. Execution of the slow-path fallback with other concurrent HTM split transactions does not affect correctness, since the specification ensures that HTM aborts

Algorithm 5 StackTrack: slow-path functions.

```
1:  procedure SLOW_INIT(ctx, op_id)
        ▷ Slow-path checkpoint policy code
2:  end procedure
3:
4:  procedure SLOW_START(ctx)
        ▷ Slow-path checkpoint policy code
5:  end procedure
6:
7:  procedure SLOW_READ(ctx, addr)
8:      step 1: ptr_value ← LOAD(addr)
9:      ADD(ctx.refs_set, ptr_value)
10:     memory_fence()
11:     if ptr_value = LOAD(addr) then
12:         return ptr_value
13:     else
14:         REMOVE(ctx.refs_set, ptr_value)
15:         goto step 1
16:     end if
17: end procedure
18:
19: procedure SLOW_WRITE(ctx, addr, new_value)
20:     SLOW_READ(ctx, addr)
21:     STORE(addr, new_value)
22: end procedure
23:
24: procedure SLOW_CHECKPOINT(ctx)
        ▷ Slow-path checkpoint policy code
25: end procedure
26:
27: procedure SLOW_COMMIT(ctx)
28:     for all ptr_value ∈ ctx.refs_set do
29:         REMOVE(ctx.refs_set, ptr_value)
30:     end for
31: end procedure
```

immediately on write-after-read, write-after-write or read-after-write conflict with non-speculative code [1].

Safety. To check that no live node is freed, we examine the code of the reclaim operation. A node is freed only if the reclaiming thread scans the registers and stack frames of active threads and does not see a reference to the node; if there are threads on the slow path, then the reclaiming thread also checks the reference metadata for the threads. Assume for contradiction that there still exists a reading thread with a reference to the free node.

We have two cases: if the reading thread is on the fast path, then it must have accessed the node as part of its current hardware transaction. (Otherwise, the reference to the node would have been seen by the reclaiming thread.) However, the reclaiming thread already successfully *unlinked* the node from the data structure, since it is executing the FREE() operation. Therefore, the current transaction of the reading thread has an inconsistent view of memory, and *will be aborted*. Crucially, the reading thread cannot obtain another reference to the reclaimed node after the abort, since, by definition, this node is *no longer reachable* in the data structure.

The second case is when the thread holding the reference is on the slow path. In this case, notice that, since the collecting thread does not see any references to the node, the reading thread must be between the point where it obtains the reference to the node for the first time, but *before* it exposes this reference through the fence instruction. The algorithm specifies that the reading thread must then *validate* the pointer before using it, checking that it still has the same value. However, the reclaiming thread has unlinked the node from the data structure. There are two possibilities: first, the reading thread's validation sees a changed pointer, in which case the thread re-starts its read operation.

Second, the reading thread's validation can see *the same* pointer value. (This can occur if the node is reclaimed, then re-allocated and again linked to from the parent node.) This is an instance of the well-known *ABA problem* [22]. However, notice that, for the thread, this execution is exactly the same as if the data structure pointer pointed to the second "version" of the node throughout its call to SLOW_READ(). Therefore, the thread can safely proceed with its execution. (Note that this scenario cannot occur on the transactional fast path, since any transaction experiencing this switch would be automatically aborted.)

Finally, notice that the execution of the slow-path fallback with concurrent HTM segments does not affect correctness, since the specification ensures that hardware transactions immediately abort on conflict with non-speculative code [1]. Also, our scheme is not affected by cyclic references in the data structure (such as a doubly-linked list), since the reclamation algorithm only tracks references per thread. Once a node is in unlinked state and is no longer referenced, it can be freed.

6. Performance Evaluation

We implemented the StackTrack scheme in C, and integrated it into a set of classic non-blocking data structures. In the following, we first examine search data structures, i.e. the Harris lock-free list [15] and the Fraser-Harris lock-free skip-list [13]. Then, we show results for data structures with short operations, i.e. the Michael-Scott lock-free queue [23] and a lock-free hash-table based on the Harris lock-free list [15]. We chose these data structures to explore the performance of the scheme under high contention (queue), low contention (hash-table), long operations (list, skip list), and for short operations (queue).

We instantiate these data structures with standard parameter values. (We found that significantly increasing the number of nodes in the data structure or the percentage of mutations increases the base cost of the implementation, partially hiding the cost of memory reclamation.)

Experimental setup. We executed the benchmarks on Intel 8-way Haswell chip with 2 cores, each multiplexing 2 hardware threads (HyperThreading). This chip supports *TSX*, Transactional Synchronization Extensions [1], a best-effort HTM implementation. We use the *RTM* interface to interact with the HTM. Currently, 4 is the highest number of cores available for a system supporting the Haswell HTM. The limited concurrency implies that our scalability results should be interpreted with a grain of salt. However, they are a good illustration for the potential of our scheme.

Approaches. The memory reclamation algorithms we benchmarked are the following:

Original : The original implementation of the lock-free concurrent algorithm, without any memory reclamation, and without HTM. This represents the best performance that the data structure could achieve. (We also tested lock-free versions employing HTM, and obtained similar results.)

Epoch : Epoch-based memory reclamation [12, 16]. Every thread has a local timestamp, which it updates with every operation start and finish. Before reclaiming a node, the *free* procedure checks that all of the threads made progress, by taking a snapshot of these timestamps and waiting for their progress (or change).

Hazards : The hazard pointers implementation of [22]. We manually add the code that defines, updates and verifies the hazard pointers to the data-structure implementations.

DTA : The recent *drop-the-anchor* scheme [4], loosely based on hazard pointers. It is known to achieve better performance than hazard pointers by eliding hazards, and by utilizing a wait-free freezing procedure. On the other hand, this scheme is not automatic, as it requires the programmer to implement the complex wait-free freezing manually. This requires a good understanding of the concurrent data-structure internals. Moreover, to our knowledge, the only data structure for which the specifics of this memory reclamation mechanism are known is the linked list [4]. Therefore, we only present the implementation results for the case of the linked list.

StackTrack : Implementation of our StackTrack scheme. We start the execution with initial hardware transactions length set to 50, and the algorithm performs automatic adjustment of these lengths according to the prediction algorithm described in Section 5.3.

Note that StackTrack, Epoch, and reference counting are the only schemes which could be applied automatically to existing code. Hazard pointers require manual coding of the hazard update checks, which depend on the specific data-structure implementation. Whether hazard pointers can be implemented automatically is an open problem [22]. We implement hazard pointers since they perform better than reference counting schemes, which update shared counters on every shared read. Also, hazard pointers allow us to better understand how code instrumentation affects the overall performance.

HTM behavior. The performance results are best understood in relation to the behavior of the Haswell HTM as the number of threads increases. In particular, we identified three regimes. In the *parallel* regime, the number of threads is at most the number of cores, i.e. 1–4 threads. Here, the HTM scales well, as the number of aborts is relatively low. For 5–8 threads, we enter the *hardware multiplexing* regime, where HyperThreading starts to execute two hardware threads per core. There is less real parallelism in the hardware; in particular, pairs of hardware threads share the same L1 cache. As a result, the number of *capacity aborts* increases by orders of magnitude in this range (see Figure 3), which affects performance. Finally, in the range of 9–16 threads, we have *software multiplexing*, as threads are preempted. Contention effects become more prevalent, and scalability is further affected.

It is important to note that *capacity aborts* are the main source of overhead once the number of threads surpasses the number of cores, since multiple threads are sharing the same cache. Capacity aborts are likely to be less of an issue once machines with HTM and more cores become available.

Scan behavior. The cost of scanning the stacks and registers of all the threads is amortized over the total number of operations. We found that the cost of the global scan becomes negligible for a thread when it executes once per every 10 free memory calls of this thread. Deeper analysis shows that the average stack depth inspected, increases linearly with the number of threads. We add that a scan does not always need to consider all threads: in particular, threads which are not currently performing data structure operations are skipped.

Split predictor. We use a simple split predictor, that decrements/increments by -1/+1 each segment length based on HTM commit abort/success ratios. The converge time of this predictor is relatively slow, and it is able to achieve a good performance after 2 seconds on average. We execute each run for 10 seconds, and report the average throughput per sec, which includes the simple split predictor penalties.

6.1 List and Skip-List Benchmarks

Figure 1 shows the results for the lock-free list and the lock-free skip-list. In brief, StackTrack scales well until the number of threads surpasses the number of cores, and shows performance degradation above 4 and above 8 threads, as a consequence of HyperThreading (above 4 threads), and of thread context-switches (above 8 threads).

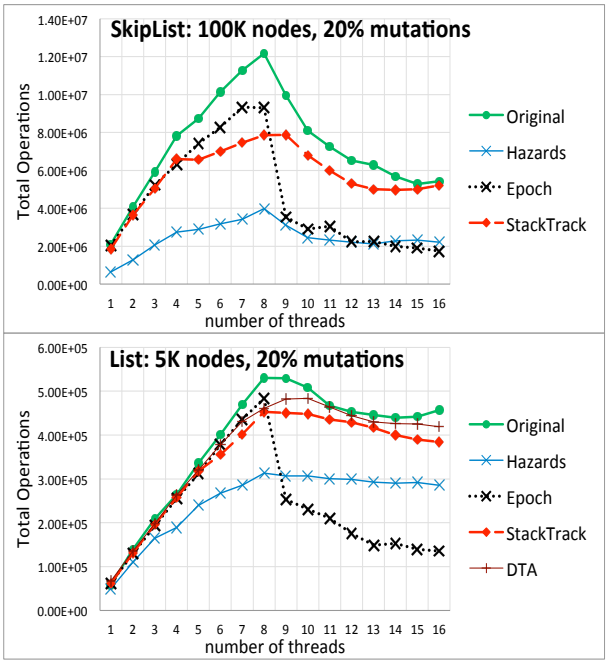

Figure 1. Throughput results for a lock-free list and a lock-free skip-list. The X-axis denotes the number of threads, and the Y-axis denotes operations per second.

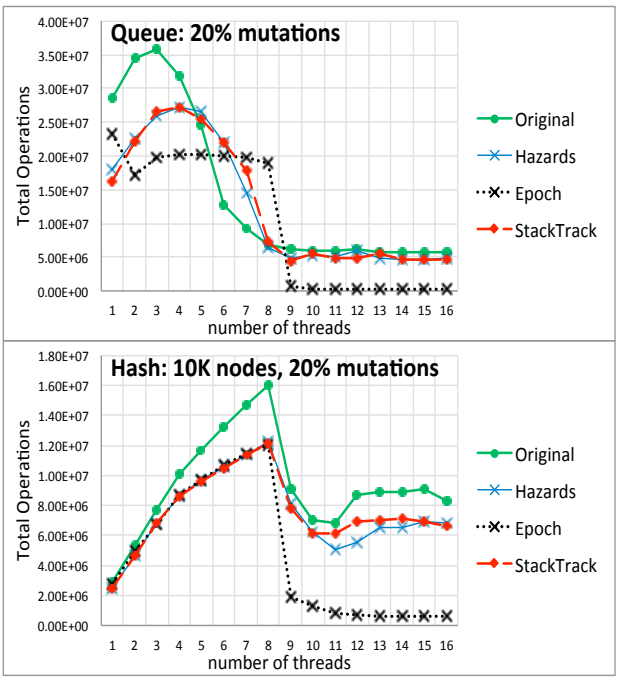

Figure 2. Throughput results for a lock-free queue and a lock-free hash-table. The X-axis denotes the number of threads, and the Y-axis denotes operations per second.

More precisely, the results for 1–4 threads illustrate the overhead of the different schemes relative to the original implementation. Hazard pointers add the most overhead, since they require a memory fence per hazard pointer update, which happens every time the operation moves to the next node in the data-structure it traverses. A memory fence is an expensive synchronization operation; its performance depends on the architecture, but in general it stops the next memory operation from progressing until the store buffer of the processor drains completely. The cost of these instructions on different architectures has been analyzed recently by David et al. [6]. Epoch is the most light-weight, since it only requires a thread local timestamp update per operation start and finish. The overhead of StackTrack is close to that of the Epoch scheme. This may seem surprising, since StackTrack adds a split checkpoint function call per basic code block. However, notice that most of the time these calls simply increment a local counter, and only perform the split operation when the local counter reaches the threshold. Hence, we avoid synchronization operations or memory fences per every call.

In the 5–8 threads range, there is a performance penalty for the StackTrack scheme, because of the Intel Haswell HyperThreading mechanism: pairs of threads share the same L1 cache, which causes the HTM to incure more capacity aborts. Figure 3 illustrates the HTM contention and capacity abort rates for the lock-free list. As expected, contention increases linearly; however, there is a significant increase in capacity aborts above 4 threads.

Another source of the StackTrack overhead is the automatic adjustment of the lengths of the split hardware transactions lengths. In Figure 4 we can see the average HTM lengths the algorithm converged to, and the average number of splits it performed per operation. Higher thread counts induce more aborts, therefore the system converges towards shorter transactions, which increase the commit rate.

Above 8 threads, the system starts to perform thread context switches, which introduces thread delays. As a result, all the algorithms receive an additional penalty. Here, the blocking Epoch scheme incurs a drastic decrease in performance, since it is must wait for all threads to progress, including the ones that are preempted. The advantage of non-blocking memory reclamation algorithms is apparent in this scenario.

6.2 Queue and Hash-Table Benchmarks

Figure 2 shows results for a lock-free queue and a lock-free hash-table. Similarly to the list and the skip-list benchmarks, we can see the effect of HyperThreading above 4 threads, and the effect of the context-switches above 8 threads. Also, the Epoch scheme sees drastic performance degradation above 8 threads, while StackTrack and hazard pointers maintain stable performance.

The queue benchmark represents a highly contended execution, since every thread contends on the head and tail pointers. The hash-table is the least contended, since threads

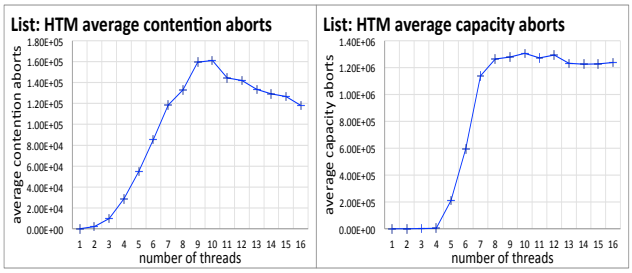

Figure 3. HTM contention and capacity aborts for the list benchmark. The average is taken per transactional segment.

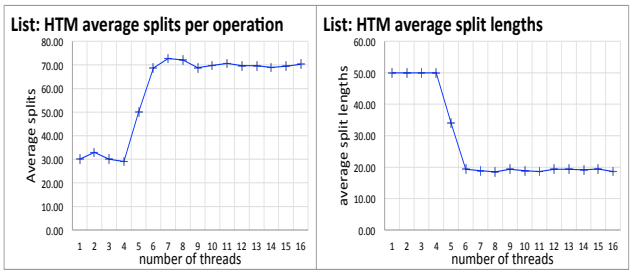

Figure 4. HTM average number of splits per operation, and average length of the split hardware transaction.

are evenly distributed by the hash function. This difference is reflected in the benchmark results.

The hash-table algorithm scales almost linearly for 1–8 threads, since the data structure experiences limited contention. By contrast, for the queue benchmark, the original algorithm scales only for 1–3 threads, and then its performance decreases. For the queue, StackTrack and hazard pointers techniques scale slightly better, and are able to outperform the original queue for 4–8 threads. The reason for this is the "over-throttle" effect, which the original queue encounters due to excessive contention. The memory reclamation schemes slow down the implementation, and as a side-effect reduce the contention on the head and tail pointers of the queue, which improves overall performance. (This effect has been previously described and studied in [9].)

We note that StackTrack and hazard pointers perform similarly for the queue and the hash-table benchmarks. This is because the operations are very short, so adding a small number of hazards pointers or a small number of split checkpoints has approximately the same overhead. The difference becomes more significant in favor of our scheme when operations are longer, as seen in the skip-list benchmark.

6.3 Slow-Path Analysis

Figure 5 shows skip-list benchmark results for the slow-path fallback. In the graph, *StackTrack-0* indicates an experiment with 0% slow-path fallbacks—all operations execute using split hardware transactions. We use this case as an ideal

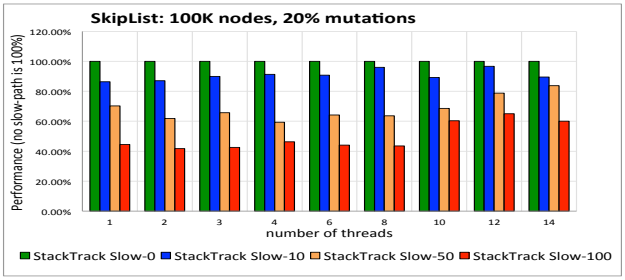

Figure 5. The effect of slow-path fallbacks in the skip-list benchmark. We give results for different percentages of slow-paths: 0%, 10%, 50% and 100% of the total operations.

100% performance reference point, and show the relative performance of the StackTrack 10, 50, and 100, which execute 10%, 50%, and 100% slow-path fallbacks respectively.

Slow-paths induce significant slow-down, of up to 60% of the original performance for 100% slow-path fallbacks. On the other hand, most of the operations would succeed in hardware, since the split predictor is able to adjust and make split hardware transactions smaller or larger. We expect the slow-path fallback to occur rarely, and the likely expected slow-down is upper bounded by the 10% slow-path case.

The slow-path penalty is less prominent for higher thread counts; there are higher penalties for concurrency and contention when there are many threads executing at the same time, which "hides" the cost of the slow-path fallback.

7. Discussion and Future Work

The results in the previous section suggest that StackTrack can provide automatic memory reclamation while maintaining low overhead. Its performance is superior to that of hazard pointers (by up to 200%), and outperforms the epoch-based technique if threads may be preempted. On the other hand, StackTrack may reduce the throughput of the data structure by at most 50%.

The overhead due to instrumentation and contention grows roughly linearly with the number of threads, while the number of capacity aborts appears to grow exponentially once the number of threads surpasses the number of cores. This suggests that a significant fraction of the overhead is caused by limitations of the current hardware, which might be mitigated in future HTM systems. In sum, these results lead us to believe that our scheme has the potential to scale well on HTM systems with higher numbers of cores.

The main limitation of our scheme is its reliance on HTM. We believe that this technology will become more prevalent, and will provide better scalability. In fact, StackTrack provides a good illustration of the power of HTM to simplify concurrent data structure design. (While StackTrack can also be executed using software transactional memory, hardware support is essential for performance.)

An interesting aspect of our framework is the automatic splitting of transactions into short segments, whose commit rate is superior to that of long transactions. Our experience suggests that this technique may be useful beyond memory reclamation, and that it is possible to automate the segmentation to optimize performance. Improving the automatic segmentation technique is an intriguing topic for future work.

We have certainly not explored the whole potential of HTM for memory reclamation. The currently available HTM system has several limitations, both in terms of number of cores and capacity. The capabilities and performance of HTM will undoubtedly improve in the future, and so will the performance of our scheme.

Acknowledgments. We would like to thank Alex Kogan for useful suggestions, and the shepherd and anonymous reviewers for their careful consideration of our work.

References

[1] Intel architecture instruction set extensions programming reference, chapter 8, 2013.

[2] H.-J. Boehm. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 197–206, New York, NY, USA, 1993. ACM. doi: 10.1145/155090.155109.

[3] A. Braginsky and E. Petrank. Locality-conscious lock-free linked lists. In *Proceedings of the 12th International Conference on Distributed Computing and Networking (ICDCN)*, pages 107–118, 2011.

[4] A. Braginsky, A. Kogan, and E. Petrank. Drop the anchor: lightweight memory management for non-blocking data structures. In *Proceedings of the 25th ACM symposium on Parallelism in algorithms and architectures*, SPAA '13, pages 33–42, New York, NY, USA, 2013. ACM.

[5] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 39–52. ACM, 2011.

[6] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, pages 33–48, New York, NY, USA, 2013. ACM. doi: 10.1145/2517349.2522714.

[7] D. Detlefs, P. A. Martin, M. Moir, and G. L. S. Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002.

[8] D. Dice, Y. Lev, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 325–334, 2010.

[9] D. Dice, D. Hendler, and I. Mirsky. Lightweight contention management for efficient compare-and-swap operations. In *Proceedings of the 19th International Conference on Parallel Processing (Euro-Par)*, pages 595–606, 2013.

[10] A. Dragojevic, M. Herlihy, Y. Lev, and M. Moir. On the power of hardware transactional memory to simplify memory management. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 99–108, 2011.

[11] C. H. Flood, D. Detlefs, N. Shavit, and X. Zhang. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the USENIX Java Virtual Machine Research and Technology Symposium*, 2001.

[12] K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, Feb. 2004.

[13] K. Fraser and T. L. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), 2007.

[14] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Trans. Parallel Distrib. Syst.*, 20(8): 1173–1187, 2009.

[15] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the International Conference on Distributed Computing (DISC)*, pages 300–314, 2001.

[16] 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, 2007.

[17] M. Herlihy and J. E. B. Moss. Lock-free garbage collection for multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 3(3): 304–311, 1992.

[18] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, pages 289–300, 1993.

[19] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. ISBN 0123705916, 9780123705914.

[20] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of the 16th International Conference on Distributed Computing (DISC)*, pages 339–353, 2002.

[21] D. Lea. Java concurrency package, 2005. Available at http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/.

[22] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15 (6):491–504, 2004.

[23] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275, 1996.

[24] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 214–222, 1995.