A Template for Implementing Fast Lock-free Trees Using HTM

Trevor Brown
Technion, Israel
me@tbrown.pro

ABSTRACT

Algorithms that use hardware transactional memory (HTM) must provide a software-only fallback path to guarantee progress. The design of the fallback path can have a profound impact on performance. If the fallback path is allowed to run concurrently with hardware transactions, then hardware transactions must be instrumented, adding significant overhead. Otherwise, hardware transactions must wait for any processes on the fallback path, causing concurrency bottlenecks, or move to the fallback path. We introduce an approach that combines the best of both worlds. The key idea is to use three execution paths: an HTM fast path, an HTM middle path, and a software fallback path, such that the middle path can run concurrently with each of the other two. The fast path and fallback path do *not* run concurrently, so the fast path incurs no instrumentation overhead. Furthermore, fast path transactions can move to the middle path instead of waiting or moving to the software path. We demonstrate our approach by producing an accelerated version of the tree update template of Brown et al., which can be used to implement fast lock-free data structures based on down-trees. We used the accelerated template to implement two lock-free trees: a binary search tree (BST), and an (a, b)-tree (a generalization of a B-tree). Experiments show that, with 72 concurrent processes, our accelerated (a, b)-tree performs between 4.0x and 4.2x as many operations per second as an implementation obtained using the original tree update template.

1 INTRODUCTION

Concurrent data structures are crucial building blocks in multithreaded software. There are many concurrent data structures implemented using locks, but locks can be inefficient, and are not fault tolerant (since a process that crashes while holding a lock can prevent all other processes from making progress). Thus, it is often preferable to use hardware synchronization primitives like compare-and-swap (CAS) instead of locks. This enables the development of *lock-free* (or *non-blocking*) data structures, which guarantee that at least one process will always continue to make progress, even if some processes crash. However, it is notoriously difficult to implement lock-free data structures from CAS, and this has inhibited the development of advanced lock-free data structures.

One way of simplifying this task is to use a higher level synchronization primitive that can atomically access multiple locations. For example, consider a k-word compare-and-swap (k-CAS), which atomically: reads k locations, checks if they contain k expected values, and, if so, writes k new values. k-CAS is highly expressive, and it can be used in a straightforward way to implement *any* atomic operation. Moreover, it can be implemented from CAS and registers [18]. However, since k-CAS is so expressive, it is difficult to implement efficiently.

Brown et al. [7] developed a set of new primitives called LLX and SCX that are less expressive than k-CAS, but can still be used in a natural way to implement many advanced data structures. These primitives can be implemented much more efficiently than k-CAS. At a high level, LLX returns a snapshot of a node in a data structure, and after performing LLXs on one or more nodes, one can perform an SCX to atomically: change a field of one of these nodes, and *finalize* a subset of them, *only if* none of these nodes have changed since the process performed LLXs on them. Finalizing a node prevents any further changes to it, which is useful to stop processes from erroneously modifying deleted parts of the data structure. In a subsequent paper, Brown et al. used LLX and SCX to design a *tree update template* that can be followed to produce lock-free implementations of down-trees (trees in which all nodes except the root have in-degree one) with any kinds of update operations [8]. They demonstrated the use of the template by implementing a chromatic tree, which is an advanced variant of a red-black tree (a type of balanced binary search tree) that offers better scalability. The template has also been used to implement many other advanced data structures, including lists, relaxed AVL trees, relaxed (a, b)-trees, relaxed b-slack trees and weak AVL trees [6, 8, 19]. Some of these data structures are highly efficient, and would be well suited for inclusion in data structure libraries.

In this work, we study how the new hardware transactional memory (HTM) capabilities found in recent processors (e.g., by Intel and IBM) can be used to produce significantly faster implementations of the tree update template. By accelerating the tree update template, we also provide a way to accelerate all of the data structures that have been implemented with it. Since library data structures are reused many times, even minor performance improvements confer a large benefit.

HTM allows a programmer to run blocks of code in transactions, which either commit and take effect atomically, or abort and have no effect on shared memory. Although transactional memory was originally intended to *simplify* concurrent programming, researchers have since realized that HTM can also be used effectively to *improve the performance of existing concurrent code* [23, 24, 30]: Hardware transactions typically have very little overhead, so they can often be used to replace other, more expensive synchronization mechanisms. For example, instead of performing a sequence of CAS primitives, it may be faster to perform reads, if-statements and writes inside a transaction. Note that this represents a *non-standard use of HTM*: we are *not* interested in its ease of use, but, rather, in its ability to reduce synchronization costs.

Although hardware transactions are fast, it is surprisingly difficult to obtain the full performance benefit of HTM. Here, we consider

This work was performed while Trevor Brown was a student at the University of Toronto. Funding was provided by the Natural Sciences and Engineering Research Council of Canada. I would also like thank my supervisor Faith Ellen for her helpful comments on this work, and to Oracle Labs for providing access to the 72-thread Intel machine used in my experiments.

CONF 'YY, Month DD, 20YY, City, Country
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Intel's HTM, which is a *best-effort* implementation. This means it offers *no guarantee* that transactions will ever commit. Even in a single threaded system, a transaction can repeatedly abort because of internal buffer overflows, page faults, interrupts, and many other events. So, to guarantee progress, any code that uses HTM must also provide a *software fallback path* to be executed if a transaction fails. The design of the fallback path profoundly impacts the performance of HTM-based algorithms.

Allowing concurrency between two paths. Consider an operation O that is implemented using the tree update template. One natural way to use HTM to accelerate O is to use the original operation as a fallback path, and then obtain an HTM-based fast path by wrapping O in a transaction, and performing optimizations to improve performance [23]. We call this the **2-path concurrent** algorithm (*2-path con*). Since the fast path is just an optimized version of the fallback path, transactions on the fast path and fallback path can safely run concurrently. If a transaction aborts, it can either be retried on the fast path, or be executed on the fallback path. Unfortunately, supporting concurrency between the fast path and fallback path can add significant overhead on the fast path.

The first source of overhead is *instrumentation* on the fast path that manipulates the *meta-data* used by the fallback path to synchronize processes. For example, lock-free algorithms often create a *descriptor* for each update operation (so that processes can determine how to help one another make progress), and store pointers to these descriptors in Data-records, where they act as locks. The fast path must also manipulate these descriptors and pointers so that the fallback path can detect changes made by the fast path.

The second source of overhead comes from constraints imposed by algorithmic assumptions made on the fallback path. The tree update template implementation in [8] assumes that only child pointers can change, and all other fields of nodes, such as keys and values, are never changed. Changes to these other *immutable* fields must be made by replacing a node with a new copy that reflects the desired change. Because of this assumption on the fallback path, transactions on the fast path *cannot* directly change any field of a node other than its child pointers. This is because the fallback path has no mechanism to detect such a change (and may, for example, erroneously delete a node that is concurrently being modified by the fast path). Thus, just like the fallback path, the fast path must replace a node with a new copy to change its immutable fields, which can be much less efficient than changing its fields directly.

Disallowing concurrency between two paths. To avoid the overheads described above, concurrency is often *disallowed* between the fast path and fallback path. The simplest example of this approach is a technique called **transactional lock elision** (TLE) [27, 28]. TLE is used to implement an operation by wrapping its sequential code in a transaction, and falling back to acquire a global lock after a certain number of transactional attempts. At the beginning of each transaction, a process reads the state of the global lock and aborts the transaction if the lock is held (to prevent inconsistencies that might arise because the fallback path is not atomic). Once a process begins executing on the fallback path, all concurrent transactions abort, and processes wait until the fallback path is empty before retrying their transactions.

If transactions never abort, then *TLE represents the best performance we can hope to achieve*, because the fallback path introduces almost no overhead and synchronization is performed entirely by hardware. Note, however, that TLE is not lock-free. Additionally, in workloads where operations periodically run on the fallback path, performance can be very poor.

As a toy example, consider a TLE implementation of a binary search tree, with a workload consisting of insertions, deletions and *range queries*. A range query returns all of the keys in a range $[lo, hi)$. Range queries access many memory locations, and cause frequent transactional aborts due to internal processor buffer overflows (capacity limits). Thus, range queries periodically run on the fallback path, where they can lead to numerous performance problems. Since the fallback path is sequential, range queries (or any other long-running operations) cause a severe *concurrency bottleneck*, because they prevent transactions from running on the fast path while they slowly complete, serially.

One way to mitigate this bottleneck is to replace the sequential fallback path in TLE with a lock-free algorithm, and replace the global lock with a fetch-and-increment object F that counts how many operations are running on the fallback path. Instead of aborting if the lock is held, transactions on the fast path abort if F is non-zero. We call this the **2-path non-concurrent** algorithm (*2-path* $\overline{con}$). In this algorithm, if transactions on the fast path retry only a few times before moving to the fallback path, or do not wait between retries for the fallback path to become empty, then the lemming effect [15] can occur. (The lemming effect occurs when processes on the fast path rapidly fail and move to the fallback path, simply because other processes are on the fallback path.) This can cause the algorithm to run only as fast as the (much slower) fallback path. However, if transactions avoid the lemming effect by retrying many times before moving to the fallback path, and waiting between retries for the fallback path to become empty, then processes can spend most of their time *waiting*. The performance problems discussed up to this point are summarized in Figure 1.

The problem with two paths. In this paper, we study two different types of workloads: **light workloads**, in which transactions rarely run on the fallback path, and **heavy workloads**, in which transactions more frequently run on the fallback path. In light workloads, algorithms that allow concurrency between paths perform very poorly (due to high overhead) in comparison to algorithms that disallow concurrency. However, in heavy workloads, algorithms that disallow concurrency perform very poorly (since transactions on the fallback path prevent transactions from running on the fast path) in comparison to algorithms that allow concurrency between paths. Consequently, all two path algorithms have workloads that yield poor performance. Our experiments confirm this, showing surprisingly poor performance for two path algorithms in many cases.

Using three paths. We introduce a technique that simultaneously achieves high performance for both light and heavy workloads by using three paths: an HTM fast path, an HTM middle path and a non-transactional fallback path. (See the right half of Figure 1.) Each operation begins on the fast path, and moves to the middle path after it retries F times. An operation on the middle path moves to the fallback path after retrying M times on the middle path. The fast path does not manipulate any synchronization meta-data used by the

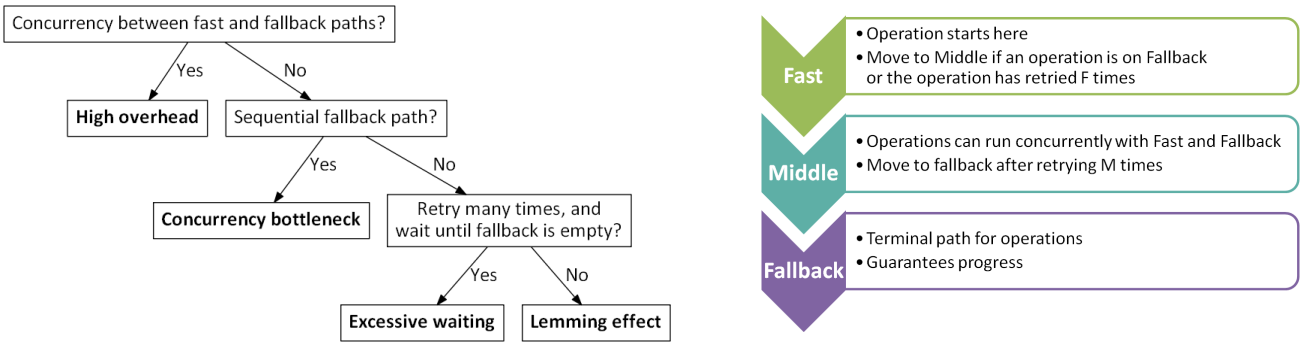

Figure 1: (Left) Performance problems affecting two-path algorithms. (Right) Using three execution paths.

fallback path, so operations on the fast path and fallback path cannot run concurrently. Thus, whenever an operation is on the fallback path, all operations on the fast path move to the middle path. The middle path manipulates the synchronization meta-data used by the fallback path, so operations on the middle path and fallback path can run concurrently. Operations on the middle path can also run concurrently with operations on the fast path (since conflicts are resolved by the HTM system). We call this the **3-path** algorithm (*3-path*).

We briefly discuss why this approach avoids the performance problems described above. Since transactions on the fast path do not run concurrently with transactions on the fallback path, transactions on the fast path run with *no instrumentation overhead*. When a transaction is on the fallback path, transactions can freely execute on the middle path, *without waiting*. The *lemming effect* does not occur, since transactions do not have to move to the fallback path simply because a transaction is on the fallback path. Furthermore, we enable a high degree of concurrency, because the fast and middle paths can run concurrently, and the middle and fallback paths can run concurrently.

We performed experiments to evaluate our new template algorithms by comparing them with the original template algorithm. In order to compare the different template algorithms, we used each algorithm to implement two data structures: a binary search tree (BST) and a relaxed (a, b)-tree. We then ran microbenchmarks to compare the performance (operations per second) of the different implementations in both light and heavy workloads. The results show that our new template algorithms offer significant performance improvements. For example, on an Intel system with 72 concurrent processes, our best implementation of the relaxed (a, b)-tree outperformed the implementation using the original template algorithm by an average of 410% over all workloads.

Contributions

- We present four accelerated implementations of the tree update template of Brown et al. that explore the design space for HTM-based implementations: *2-path con*, *TLE*, *2-path* $\overline{con}$, and *3-path*.
- We highlight the importance of studying both light and heavy workloads in the HTM setting. Each serves a distinct role in evaluating algorithms: light workloads demonstrate the potential of HTM to improve performance by reducing overhead, and heavy workloads capture the performance impact of interactions between different execution paths.
- We demonstrate the effectiveness of our approach by accelerating two different lock-free data structures: an unbalanced BST, and a relaxed (a, b)-tree. Experimental results show a significant performance advantage for our accelerated implementations.

The remainder of the paper is structured as follows. The model is introduced in Section 2. Section 3 describes LLX and SCX, and the tree update template. We describe an HTM-based implementation of LLX and SCX in Section 4. In Section 5, we describe our four accelerated template implementations, and argue correctness and progress. In Section 6, we describe two data structures that we use in our experiments. Experimental results are presented in Section 7. Section 8 describes an optimization to two of our accelerated template implementations. In Section 9, we describe a way to reclaim memory more efficiently for *3-path* algorithms. In Section 10, we describe how our approach could be used to accelerate data structures that use the read-copy-update (RCU) or k-compare-and-swap primitives. Related work is surveyed in Section 11. Finally, we conclude in Section 12.

2 MODEL

We consider an asynchronous shared memory system with n processes, and Intel's implementation of HTM. Arbitrary blocks of code can be executed as transactions, which either commit (and appear to take place instantaneously) or abort (and have no effect on the contents of shared memory). A transaction is started by invoking *txBegin*, is committed by invoking *txEnd*, and can be aborted by invoking *txAbort*. Intel's implementation of HTM is best-effort, which means that the system can force transactions to abort at any time, and no transactions are ever guaranteed to commit. Each time a transaction aborts, the hardware provides a reason why the abort occurred. Two reasons are of particular interest. *Conflict* aborts occur when two processes contend on the same cache-line. Since a cache-line contains multiple machine words, *conflict* aborts can occur even if two processes never contend on the same memory location. *Capacity* aborts occur when a transaction exhausts some shared resource within the HTM system. Intuitively, this occurs when a transaction accesses too many memory locations. (In reality, *capacity* aborts also occur for a variety of complex reasons that make it difficult to predict when they will occur.)

3 BACKGROUND

The LLX **and** SCX **primitives.** The load-link extended (LLX) and store-conditional extended (SCX) primitives are multi-word generalizations of the well-known load-link (LL) and store-conditional (SC), and they have been implemented from single-word CAS [7]. LLX and SCX operate on Data-records, each of which consists of a fixed number of mutable fields (which can change), and a fixed number of immutable fields (which cannot).

LLX(r) attempts to take a snapshot of the mutable fields of a Data-record r. If it is concurrent with an SCX involving r, it may return Fail, instead. Individual fields of a Data-record can also be read directly. An SCX(V, R, fld, new) takes as its arguments a sequence V of Data-records, a subsequence R of V, a pointer fld to a mutable field of one Data-record in V, and a new value new for that field. The SCX tries to atomically store the value new in the field that fld points to and *finalize* each Data-record in R. Once a Data-record is finalized, its mutable fields cannot be changed by any subsequent SCX, and any LLX of the Data-record will return Finalized instead of a snapshot.

Before a process p invokes SCX, it must perform an LLX(r) on each Data-record r in V. For each $r \in V$, the last LLX(r) performed by p prior to the SCX is said to be *linked* to the SCX, and this linked LLX must return a snapshot of r (not Fail or Finalized). An SCX(V, R, fld, new) by a process modifies the data structure and returns True (in which case we say it *succeeds*) only if no Data-record r in V has changed since its linked LLX(r); otherwise the SCX fails and returns False. Although LLX and SCX can fail, their failures are limited in such a way that they can be used to build data structures with lock-free progress. See [7] for a more formal specification.

Observe that SCX can only change a single value in a Data-record (and finalize a sequence of Data-records) atomically. Thus, to implement an *operation* that changes multiple fields, one must create *new* Data-records that contain the desired changes, and use SCX to change *one* pointer to replace the old Data-records.

Pseudocode for the original, CAS-based implementation of LLX and SCX appears in Figure 2. Each invocation S of SCX$_O$$V$, R, fld, new starts by creating an SCX-record D, which contains all of the information necessary to perform S, and then invokes Help$D$ to perform it. The SCX-record also contains a *state* field, which initially contains the value InProgress. When S finishes, the *state* field of D will contain either Committed or Aborted depending on whether the S succeeded.

S synchronizes with other invocations of SCX$_O$ by taking a special kind of lock on each Data-record in V. These locks grant exclusive access to an *operation*, rather than to a *process*. Henceforth, we use the term *freezing* (resp., unfreezing), instead of locking (resp., unlocking), to differentiate this kind of locking from typical mutual exclusion. A Data-record u is *frozen* for S if $u.info$ points to D, and either $D.state$ Committed and $u.marked$ *true* (in which case we say u is *finalized*), or $D.state$ InProgress.

So, S freezes a Data-record u by using CAS to store a pointer to D in $u.info$ (at the freezing CAS step in Figure 2). Suppose S successfully freezes all Data-records in its V sequence. Then, S prepares to finalize each Data-record $u \in R$ by setting a *marked* bit in u (at the mark step in Figure 2). Finally, S changes fld to new, and atomically releases all locks by setting $D.state$ to Committed (at the commit step in Figure 2). Observe that setting $D.state$ to Committed has the effect of atomically finalizing all Data-records in R and unfreezing all Data-records in $V \setminus R$.

Now, suppose S was prevented from freezing some Data-record u because another invocation S' of SCX$_O$ had already frozen u (i.e., the freezing CAS step by S failed, and it saw $r.info \neq scxPtr$ at the following line). Then, S aborts by setting the $D.state$ to Aborted (at the abort step in Figure 2). This has the effect of atomically unfreezing any Data-records S had frozen. Note that, before the process that performed S can perform another invocation of SCX$_O$ with u in its V-sequence, it must perform LLXu. If u is still frozen for S' when this LLXu is performed, then the LLX will use the information stored in the SCX-record at u to *help* S' complete and unfreeze u. (SCX-records also contain another field $allFrozen$ that is used to coordinate any processes helping the SCX, ensuring that they do not make conflicting changes to the *state* field.)

The correctness argument is subtle, and we leave the details to [7], but one crucial algorithmic property is relevant to our work:

P1: Between any two changes to (the user-defined fields of) a Data-record u, a pointer to a new SCX-record (that has never before been contained in $u.info$) is stored in $u.info$.

This property is used to determine whether a Data-record has changed between the last LLX$_O$ on it and a subsequent invocation of SCX$_O$. Consider an invocation S of SCX$_O(V, R, fld, new)$ by a process p. Let u be any Data-record in V, and L be the last invocation of LLX$_O u$ by p. L reads $u.info$ and sees some value ptr. S subsequently performs a freezing CAS step to change $u.info$ from ptr to point to its SCX-record, freezing u. If this CAS succeeds, then S infers that u has not changed between the read of $u.info$ in the LLX and the freezing CAS step.

Progress properties. Specifying a progress guarantee for LLX and SCX operations is subtle, because if processes repeatedly perform LLX on Data-records that have been finalized, or repeatedly perform failed LLXs, then they may never be able to invoke SCX. In particular, it is not sufficient to simply prove that LLXs return snapshots infinitely often, since *all* of the LLXs in a sequence must return snapshots before a process can invoke SCX. To simplify the progress guarantee for LLX and SCX, we make a definition. An SCX-Update algorithm is one that performs LLXs on a sequence V of Data-records and invokes SCX(V, R, fld, new) if they all return snapshots. The progress guarantee in [7] is then stated as follows.

PROG: Suppose that (a) there is always some non-finalized Data-record reachable by following pointers from an entry point, (b) for each Data-record r, each process performs finitely many invocations of LLXr that return Finalized, and (c) processes perform infinitely many executions of SCX-Update algorithms. Then, infinitely many invocations of SCX succeed.

The tree update template. The tree update template implements lock-free updates that atomically replace an old connected subgraph R of a down-tree by a new connected subgraph N (as shown in Figure 3). Such an update can implement any change to the tree, such as an insertion into a BST or a rotation in a balanced tree. The old subgraph includes all nodes with a field to be modified. The new subgraph may have pointers to nodes in the old tree. Since every

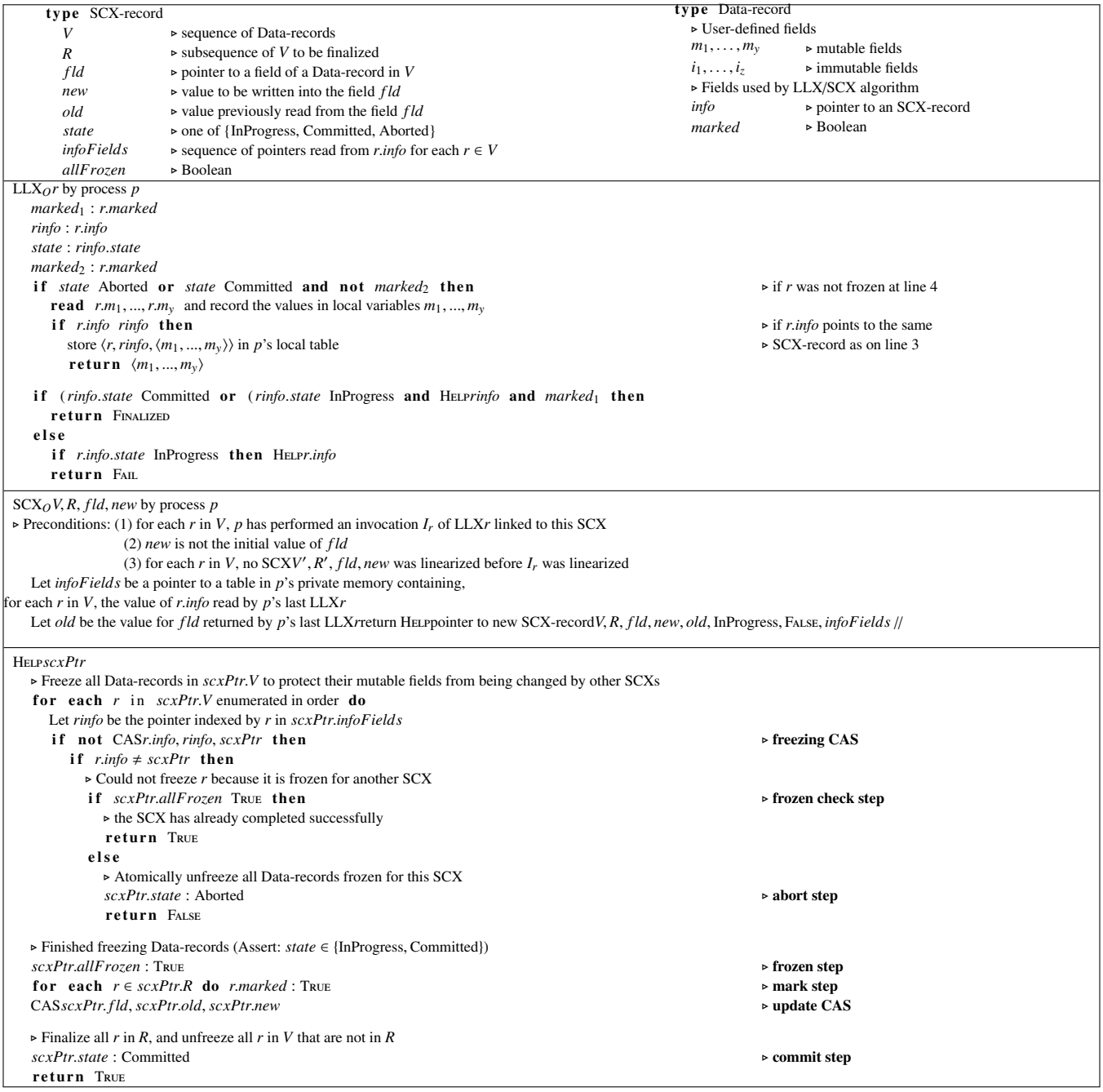

Figure 2: Data types and pseudocode for the original LLX and SCX algorithm.

node in a down-tree has indegree one, the update can be performed by changing a single child pointer of some node *parent*. However, problems could arise if a concurrent operation changes the part of the tree being updated. For example, nodes in the old subgraph, or even *parent*, could be removed from the tree before *parent*'s child pointer is changed. The template takes care of the process coordination required to prevent such problems.

Each tree node is represented by a Data-record with a fixed number of child pointers as its mutable fields. Each child pointer either points to a Data-record or contains NIL (denoted by $-\circ$ in our figures). Any other data in the node is stored in immutable fields. Thus, if an update must change some of this data, it makes a new copy of the node with the updated data. There is a Data-record *entry* which acts as the entry point to the data structure and is never deleted.

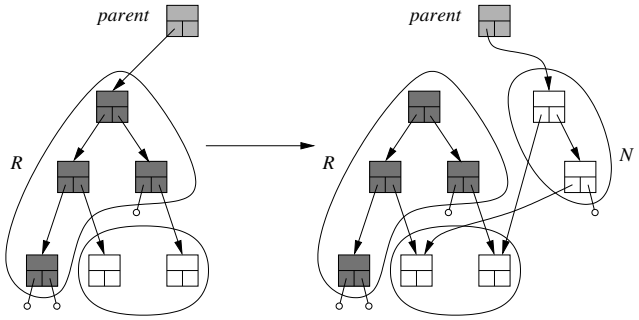

Figure 3: Example of the tree update template.

At a high level, an update that follows the template proceeds in two phases: the *search phase* and the *update phase*. In the search phase, the update searches for a location where it should occur. Then, in the update phase, the update performs LLXs on a connected subgraph of nodes in the tree, including *parent* and the set R of nodes to be removed from the tree. Next, it decides whether the tree should be modified, and, if so, creates a new subgraph of nodes and performs an SCX that atomically changes a child pointer, as shown in Figure 3, and finalizes any nodes in R. See [8] for further details.

4 HTM-BASED LLX AND SCX

In this section, we describe an HTM-based implementation of LLX and SCX. This implementation is used by our first accelerated template implementation, *2-path con*, which is described in Section 5.

In the following, we use SCX_O and LLX_O to refer to the original lock-free implementation of LLX and SCX. We give an implementation of SCX that uses an HTM-based fast path called SCX_{HTM}, and SCX_O as its fallback path. Hardware transactions are instrumented so they can run concurrently with processes executing SCX_O. This algorithm guarantees lock-freedom and achieves a high degree of concurrency. Pseudocode appears in Figure 4. At a high level, an SCX_{HTM} by a process p starts a transaction, then attempts to perform a highly optimized version of SCX_O. Each time a transaction executed by p aborts, control jumps to the onAbort label, at the beginning of the SCX procedure. If a process *explicitly* aborts a transaction at line 19, then SCX returns FALSE at line 4. Each process has a budget *AttemptLimit* that specifies how many times it will attempt hardware transactions before it will fall back to executing SCX_O.

In SCX_O, SCX-records are used (1) to facilitate helping, and (2) to lock Data-records and detect changes to them. In particular, SCX_O guarantees the following property. **P1**: between any two changes to (the user-defined fields of) a Data-record u, a new SCX-record pointer is stored in $u.info$. However, SCX_{HTM} does not create SCX-records. In a transactional setting, helping causes unnecessary aborts, since executing a transaction that performs the same work as a running transaction will cause at least one (and probably both) to abort. Helping in transactions is also *not necessary* to guarantee progress, since progress is guaranteed by the fallback path. So, to preserve property P1, we give each process p a *tagged sequence number* $tseq_p$ that contains the process name, a sequence number, and a *tag* bit. The *tag* bit is the least significant bit. On modern systems where pointers are word aligned, the least significant bit in a pointer is always zero. Thus, the tag bit allows a process to distinguish between a tagged sequence number and a pointer. In SCX_{HTM}, instead of having p create a new SCX-record and store pointers to it in Data-records to lock them, p increments its sequence number in $tseq_p$ and stores $tseq_p$ in Data-records. Since no writes performed by a transaction T can be seen until it commits, it never actually needs to hold any locks. Thus, every value of $tseq_p$ stored in a Data-record represents an unlocked value, and writing $tseq_p$ represents p locking and immediately unlocking a node.

After storing $tseq_p$ in each $r \in V$, SCX_{HTM} finalizes each $r \in R$ by setting $r.marked$: TRUE (mimicking the behaviour of SCX_O). Then, it stores *new* in the field pointed to by fld, and commits. Note that eliminating the creation of SCX-records on the fast path also eliminates the need to *reclaim* any created SCX-records, which further reduces overhead.

The SCX_{HTM} algorithm also necessitates a small change to LLX_O, to handle tagged sequence numbers. An invocation of $\text{LLX}_O(r)$ reads a pointer *rinfo* to an SCX-record, follows *rinfo* to read one of its fields, and uses the value it reads to determine whether r is locked. However, *rinfo* may now contains a tagged sequence number, instead of a pointer to an SCX-record. So, in our modified algorithm, which we call LLX_{HTM}, before a process tries to follow *rinfo*, it first checks whether *rinfo* is a tagged sequence number, and, if so, behaves as if r is unlocked. The code for LLX_{HTM} appears in Figure 8.

4.1 Correctness and Progress

The high-level idea is to show that one can start with LLX_O and SCX_O, and obtain our HTM-based implementation by applying a sequence of transformations. Intuitively, these transformations preserve the semantics of SCX and maintain backwards compatibility with SCX_O so that the transformed versions can be run concurrently with invocations of SCX_O. More formally, for each execution of a transformed algorithm, there is an execution of the original algorithm in which: the same operations are performed, they are linearized in the same order, and they return the same results. For each transformation, we sketch the correctness and progress argument, since the transformations are simple and a formal proof would be overly pedantic.

Adding transactions. For the first transformation, we replaced the invocation of HELP in SCX_O with the body of the HELP function, and wrapped the code in a transaction. Since the fast path simply executes the fallback path algorithm in a transaction, the correctness of the resulting algorithm is immediate from the correctness of the original LLX and SCX algorithm.

We also observe that it is not necessary to commit a transaction that sets the *state* of its SCX-record to Aborted and returns FALSE. The only effect that committing such a transaction would have on shared memory is changing some of the *info* fields of Data-records in its V sequence to point to its SCX-record. In SCX_O, *info* fields serve two purposes. First, they provide pointers to an SCX-record while its SCX is in progress (so it can be helped). Second, they act as locks that grant exclusive access to an SCX_O, and allow an invocation of SCX_O to determine whether any user-defined fields of

```
1  Private variable for process p: attempts_p, tagseq_p
2  SCX(V, R, fld, new) by process p
3  onAbort:                                   ▷ jump here on transaction abort
4    if we jumped here after an explicit abort then return False
5    if attempts_p < AttemptLimit then
6      attempts_p : attempts_p  1
7      retval : SCX_HTM(V, R, fld, new) ▷ Fast
8    else
9      retval : SCX_O(V, R, fld, new) ▷ Fallback
10   if retval then attempts_p : 0
11   return retval
```

```
12 SCX_HTM(V, R, fld, new) by process p
13   Let infoFields be a pointer to a table in p's private memory containing,
         for each r in V, the value of r.info read by p's last LLX(r)
14   Let old be the value for fld returned by p's last LLX(r)
15   Begin hardware transaction
16   tagseq_p := tagseq_p + 2^⌈log n⌉
17   for each r ∈ V do
18     Let rinfo be the pointer indexed by r in infoFields
19     if r.info ≠ rinfo then Abort hardware transaction (explicitly)
20   for each r ∈ V do r.info : tagseq_p
21   for each r ∈ R do r.marked : True
22   write new to the field pointed to by fld
23   Commit hardware transaction
24   return True
```

Figure 4: Final HTM-based implementation of SCX.

```
1  SCX_1(V, R, fld, new) by process p
2    Let infoFields be a pointer to a table in p's private memory containing,
3  for each r in V, the value of r.info read by p's last LLX(r)
4    Let old be the value for fld returned by p's last LLX(r)
5    Begin hardware transaction
6    scxPtr : pointer to new SCX-record V, R, fld, new, old, InProgress, False, infoFields )
7    ▷ Freeze all Data-records in scxPtr.V to protect their mutable fields from being changed by other SCXs
8    for each r in scxPtr.V enumerated in order do
9      Let rinfo be the pointer indexed by r in scxPtr.infoFields
10     if not CAS(r.info, rinfo, scxPtr) then                    ▷ freezing CAS
11       if r.info ≠ scxPtr then
12         ▷ Could not freeze r because it is frozen for another SCX
13         if scxPtr.allFrozen True then                         ▷ frozen check step
14           ▷ the SCX has already completed successfully
15           Commit hardware transaction
16           return True
17         else Abort hardware transaction (explicitly)
18   ▷ Finished freezing Data-records (Assert: state ∈ {InProgress, Committed})
19   scxPtr.allFrozen : True                                     ▷ frozen step
20   for each r ∈ scxPtr.R do r.marked : True                    ▷ mark step
21   CAS(scxPtr.fld, scxPtr.old, scxPtr.new)                     ▷ update CAS
22   ▷ Finalize all r in R, and unfreeze all r in V that are not in R
23   scxPtr.state : Committed                                    ▷ commit step
24   Commit hardware transaction
25   return True
```

Figure 5: Transforming SCX_O: after adding transactions.

a Data-record r have changed since its linked LLX(r) (using property P1). However, since the effects of a transaction are not visible until it has already committed, a transaction no longer needs help by the time it modified any *info* field. And, since an SCX_O that sets the *state* of its SCX-record to Aborted does not change any user-defined field of a Data-record, these changes to *info* fields are not needed to preserve property P1. The only consequence of changing these *info* fields is that other invocations of SCX_O might needlessly fail and return False, as well. So, instead of setting *state* Aborted and committing, we *explicitly abort* the transaction and return False. Figure 5 shows the result of this transformation: SCX_1. (Note that aborting transactions does not affect correctness—only progress.)

Of course, we must provide a fallback code path in order to guarantee progress. Figure 6 shows how SCX_1 (the fast path) and SCX_O (the fallback path) are used together to implement lock-free SCX. In order to decide when each code path should be executed, we give each process p a private variable $attempts_p$ that contains the number of times p has attempted a hardware transaction since it last performed an SCX_1 or SCX_O that succeeded (i.e., returned True). The SCX procedure checks whether $attempts_p$ is less than a (positive) threshold *AttemptLimit*. If so, p increments $attempts_p$ and invokes SCX_1 to execute a transaction on the fast path. If not, p invokes SCX_O (to guarantee progress). Whenever p returns True from an invocation of SCX_1 or SCX_O, it resets its budget $attempts_p$ to zero, so it will execute on the fast path in its next SCX. Each time a transaction executed by p aborts, control jumps to the onAbort label, at the beginning of the SCX procedure. If a process explicitly aborts a transaction it is executing (at line 17 in SCX_1), then control jumps to the onAbort label, *and* the SCX returns False at the next line.

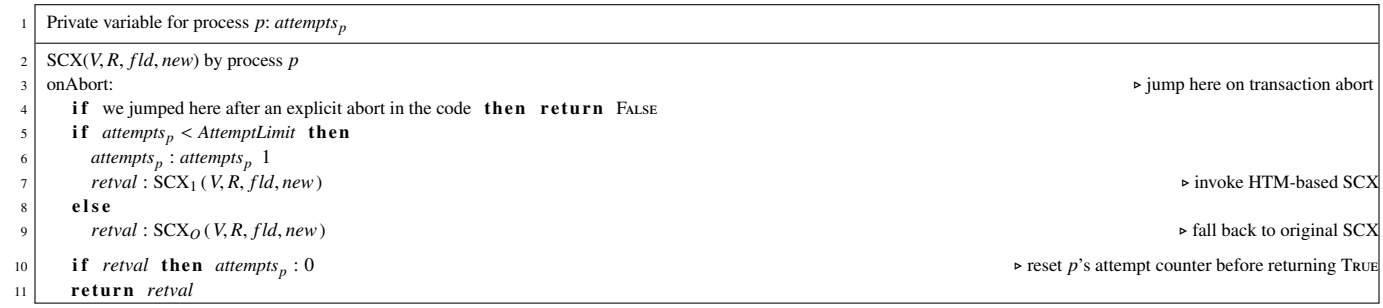

Figure 6: How the HTM-based SCX_1 is used to provide lock-free SCX.

Progress. It is proved in [7] that PROG is satisfied by LLX_O and SCX_O. We argue that PROG is satisfied by the implementation of LLX and SCX in Figure 6. To obtain a contradiction, suppose the antecedent of PROG holds, but only finitely many invocations of SCX return True. Then, after some time t, no invocation of SCX returns True.

Case 1: Suppose processes take infinitely many steps in transactions. By inspection of the code, each transaction is wait-free, and SCX returns True immediately after a transaction commits. Since no transaction commits after t, there must be infinitely many aborts. However, each process can perform at most *AttemptLimit* aborts since the last time it performed an invocation of SCX that returned True. So, only finitely many aborts can occur after t—a contradiction.

Case 2: Suppose processes take only finitely many steps in transactions. Then, processes take only finitely many steps in SCX_1. It follows that, after some time t', no process takes a step in SCX_1. Therefore, in the suffix of the execution after t', processes only take steps in SCX_O and LLX_O. However, since LLX_O and SCX_O satisfy PROG, infinitely many invocations of SCX must succeed after t', which is a contradiction.

Eliminating *most* accesses to fields of SCX-records created on the fast path. In LLX_O and SCX_O, helping is needed to guarantee progress, because otherwise, an invocation of SCX_O that crashes while one or more Data-records are frozen for it could cause every invocation of LLX_O to return Fail (which, in turn, could prevent processes from performing the necessary linked invocations of LLX_O to invoke SCX_O). However, as we mentioned above, since transactions are atomic, a process cannot see any of their writes (including the contents of any SCX-record they create and publish pointers to) until they have committed, at which point they no longer need help. Thus, it is not necessary to help transactions in SCX_1.[1]

In fact, it is easy to see that processes will not help any SCX-record created by a transaction in SCX_1. Observe that each transaction in SCX_1 sets the *state* of its SCX-record to Committed before committing. Consequently, if an invocation of LLX_O reads *r.info* and obtains a pointer *rinfo* to an SCX-record created by a transaction in SCX_1, then *rinfo* has *state* Committed. Therefore, by inspection of the code, LLX_O will not invoke Help(*rinfo*).

[1] In fact, helping transactions would be *actively harmful*, since performing the same modifications to shared memory as an in-flight transaction *will cause it to abort*. This leads to very poor performance, in practice.

Since LLX_O never invokes Help(*rinfo*) for any *rinfo* created by a transaction in SCX_1, most fields of an SCX-record created by a transaction are accessed only by the process that created the SCX-record. The only field that is accessed by other processes is the *state* field (which is accessed in LLX_O). Therefore, it suffices for a transaction in SCX_1 to initialize only the *state* field of its SCX-record. As we will see, any accesses to the other fields can simply be eliminated or replaced with locally available information.

Using this knowledge, we transform SCX_1 in Figure 5 into a new procedure called SCX_2 in Figure 7. First, instead of initializing the entire SCX-record when we create a new SCX-record at line 6 in SCX_1, we initialize only the *state* field. We then change any steps that read fields of the SCX-record (lines 8, 9, 13, 20 and 21 in SCX_1) to use locally available information, instead.

Next, we eliminate the *frozen step* at line 19 in SCX_1, which changes the $allFrozen$ field of the SCX-record. Recall that $allFrozen$ is used by SCX_O to prevent helpers from making conflicting changes to the *state* field of its SCX-record. When a *freezing CAS* fails in an invocation S of SCX_O (at line 27 of Help in Figure 2), it indicates that either S will fail due to contention, or another process had already helped S to complete successfully. The $allFrozen$ bit allows a process to distinguish between these two cases. Specifically, it is proved in [7] that a process will see $allFrozen$ True at line 27 of Help if and only if another process already helped S complete and set $allFrozen$: True. However, since we have argued that processes never help transactions (and, in fact, no other process can even *access* the SCX-record until the transaction that created it has committed), $allFrozen$ is always False at the corresponding step (line 13) in SCX_1. This observation allows us to eliminate the entire *if* branch at line 13 in SCX_1.

Clearly, this transformation preserves PROG. Note that SCX_2 (and each of the subsequent transformed variants) is used in the same way as SCX_1: Simply replace SCX_1 in Figure 6 with SCX_2.

Completely eliminating accesses to fields of SCX-records created on the fast path. We now describe a transformation that completely eliminates all accesses to the *state* fields of SCX-records created by transactions in SCX_2 (i.e., the last remaining accesses by transactions to fields of SCX-records).

We transform SCX_2 into a new procedure SCX_3, which appears in Figure 8. First, the commit step in SCX_2 is eliminated. Whereas in SCX_2, we stored a pointer to the SCX-record in *r.info* for each $r \in V$ at line 11, we store a *tagged pointer* to the SCX-record at line 29 in SCX_3. A tagged pointer is simply a pointer that has its least significant bit set to one. Note that, on modern systems where

```
1  Private variable for process p: attempts_p
2  SCX_2(V, R, fld, new) by process p
3      Let infoFields be a pointer to a table in p's private memory containing,
4  for each r in V, the value of r.info read by p's last LLX(r)
5      Let old be the value for fld returned by p's last LLX(r)
6      Begin hardware transaction
7      scxPtr : pointer to new SCX-record(−, −, −, −, −, InProgress, −, −)
8      ▷ Freeze all Data-records in V to protect their mutable fields from being changed by other SCXs
9      for each r in V enumerated in order do
10         Let rinfo be the pointer indexed by r in infoFields
11         if not CAS(r.info, rinfo, scxPtr) then                          ▷ freezing CAS
12             if r.info ≠ scxPtr then Abort hardware transaction (explicitly)
13     ▷ Finished freezing Data-records
14     for each r ∈ R do r.marked : True      ▷ Finalize each r ∈ R        ▷ mark step
15     CAS(fld, old, new)                                                  ▷ update CAS
16     scxPtr.state : Committed                                            ▷ commit step
17     Commit hardware transaction
18     return True
```

Figure 7: Transforming SCX_O: after eliminating most accesses to fields of SCX-records created on the fast path.

```
1  Private variable for process p: attempts_p
2  LLX_HTM(r) by process p ▷ Precondition: r ≠ Nil.
3      marked_1 : r.marked                                   ▷ order of lines 2–5 matters
4      rinfo : r.info
5  *   state : rinfo & 1 ? Committed : rinfo.state           ▷ if rinfo is tagged, take state to be Committed
6      marked_2 : r.marked
7      if state Aborted or (state Committed and not marked_2) then   ▷ if r was not frozen at line 4
8          read r.m_1, ..., r.m_y and record the values in local variables m_1, ..., m_y
9          if r.info rinfo then                              ▷ if r.info points to the same SCX-record as on line 3
10             store ⟨r, rinfo, ⟨m_1, ..., m_y⟩⟩ in p's local table
11             return ⟨m_1, ..., m_y⟩
12 *   state_2 : rinfo & 1 ? Committed : rinfo.state         ▷ if rinfo is tagged, take state_2 to be Committed
13 *   if (state_2 Committed or (state_2 InProgress and Help(rinfo))) and marked_1 then
14         return Finalized
15     else
16 *       rinfo_2 : r.info
17 *       state_3 : rinfo_2 & 1 ? Committed : rinfo_2.state  ▷ if rinfo_2 is tagged, take state_3 to be Committed
18 *       if state_3 InProgress then Help(rinfo_2)
19         return Fail
20 SCX_3(V, R, fld, new) by process p
21     Let infoFields be a pointer to a table in p's private memory containing,
22 for each r in V, the value of r.info read by p's last LLX(r)
23     Let old be the value for fld returned by p's last LLX(r)
24     Begin hardware transaction
25     scxPtr : pointer to new SCX-record(−, −, −, −, −, −, −, −)
26     ▷ Freeze all Data-records in V to protect their mutable fields from being changed by other SCXs
27     for each r in V enumerated in order do
28         Let rinfo be the pointer indexed by r in infoFields
29         if not CAS(r.info, rinfo, scxPtr & 1) then                  ▷ freezing CAS
30             if r.info ≠ scxPtr & 1) then Abort hardware transaction (explicitly)
31     ▷ Finished freezing Data-records
32     for each r ∈ R do r.marked : True      ▷ Finalize each r ∈ R    ▷ mark step
33     CAS(fld, old, new)                                              ▷ update CAS
34     Commit hardware transaction
35     return True
```

Figure 8: Transforming SCX_O: after completely eliminating accesses to fields of SCX-records created on the fast path.

pointers are word aligned, the least significant bit in a pointer to an SCX-record will be zero. Thus, the least significant bit in a tagged pointer allows processes to distinguish between a tagged pointer (which is stored in *r.info* by a transaction) from a regular pointer (which is stored in *r.info* by an invocation of SCX_O). Line 30 in SCX_3 is also updated to check for a tagged pointer in *r.info*.

In order to deal with tagged pointers, we transform LLX_O into new procedure called LLX_{HTM}, that is used instead of LLX_O from here on. Any time an invocation of LLX_O would follow a pointer that

was read from an *info* field *r.info*, LLX_{HTM} first checks whether the value *rinfo* read from the *info* field is a pointer or a tagged pointer. If it is a pointer, then LLX_{HTM} proceeds exactly as in LLX_O. However, if *rinfo* is a tagged pointer, then LLX_{HTM} proceeds as if it had seen an SCX-record with *state* Committed (i.e., whose SCX has already returned TRUE). We explain why this is correct. If *rinfo* contains a tagged pointer, then it was written by a transaction T that committed (since it changed shared memory) at line 34 in SCX_3, just before returning TRUE. Observe that, in SCX_2, the *state* of the SCX-record is set to Committed just before TRUE is returned. In other words, if not for this transformation, T would have set the *state* of its SCX-record to Committed. So, clearly it is correct to treat *rinfo* as if it were an SCX-record with *state* Committed.

Since this transformation simply changes the *representation* of an SCX-record D with *state* Committed that is created by a transaction (and does not change how the algorithm behaves when it encounters D), it preserves PROG.

Eliminating the creation of SCX-records on the fast path. Since transactions in SCX_3 are not helped, we would like to eliminate the *creation* of SCX-records in transactions, altogether. However, since SCX-records are used as part of the *freezing* mechanism in SCX_O on the fallback path, we cannot simply eliminate the steps that freeze Data-records, or else transactions on the fast path will not synchronize with SCX_O operations on the fallback path. Consider an invocation S of SCX_O by a process p that creates an SCX-record D, and an invocation L of LLX(r) linked to S. When S uses CAS to freeze r (by changing *r.info* from the value seen by L to D), it interprets the success of the CAS to mean that r has not changed since L (relying on property P1). If a transaction in SCX_3 changes r without changing *r.info* (to a new value that has never before appeared in *r.info*), then it would violate P1, rendering this interpretation invalid. Thus, transactions in $\text{SCX}_3(V, R, fld, new)$ must change *r.info* to a new value, for each $r \in V$.

We transform SCX_3 into a new procedure SCX_r, which appears in Figure 9. We now explain what a transaction T in an invocation S of SCX_4 by a process p does instead of creating an SCX-record and using it to freeze Data-records. We give each process p a *tagged sequence number* $tseq_p$, which consists of three bit fields: a tag-bit, a process name, and a sequence number. The tag-bit, which is the least significant bit, is always one. This tag-bit distinguishes tagged sequence numbers from pointers to SCX-records (similar to tagged pointers, above). The process name field of $tseq_p$ contains p. The sequence number is a non-negative integer that is initially zero. Instead of creating a new SCX-record (at line 25 in SCX_3), S increments the sequence number field of $tseq_p$. Then, instead of storing a pointer to an SCX-record in *r.info* for each $r \in V$ (at line 29 in SCX_3), T stores $tseq_p$. (Line 30 is also changed accordingly.) The combination of the process name and sequence number bit fields ensure that whenever T stores $tseq_p$ in an *info* field, it is storing a value that has never previously been contained in that field.[2]

[2]Technically, with a finite word size it is possible for a sequence number to overflow and wrap around, potentially causing P1 to be violated. On modern systems with a 64-bit word size, we suggest representing a tagged sequence number using 1 tag-bit, 15 bits for the process name (allowing up to 32,768 concurrent processes) and 48 bits for the sequence number. In order for a sequence number to experience wraparound, a *single process* must then perform 2^{48} operations. According to experimental measurements for several common data structures on high performance systems, this would take at least a decade of continuous updates. Moreover, if wraparound is still a concern, one can replace the freezing CAS steps in SCX_O with double-wide CAS instructions (available on most modern systems) which atomically operate on 128-bits.

Observe that LLX_{HTM} does not require any further modification to work with tagged sequence numbers, since it distinguishes between tagged sequence numbers and SCX-records using the tag-bit (the exact same way it distinguished between tagged pointers and pointers to SCX-records). Moreover, it remains correct to treat tagged sequence numbers as if they are SCX-records with *state* Committed (for the same reason it was correct to treat tagged pointers that way). Progress is preserved for the same reason as it was in the previous transformation: we are simply changing the *representation* of SCX-records with *state* Committed that are created by transactions.

Note that this transformation eliminates not only the *creation* of SCX-records, but also the need to *reclaim* those SCX-records. Thus, it can lead to significant performance improvements.

Simple optimizations. Since any code executed inside a transaction is atomic, we are free to replace atomic synchronization primitives inside a transaction with sequential code, and reorder the transaction's steps in any way that does not change its sequential behaviour. We now describe how to transform SCX_4 by performing two simple optimizations.

For the first optimization, we replace each invocation of CAS(x, o, n) with sequential code: **if** $x \quad 0$ **then** $x : n, result :$ TRUE **else** $result :$ FALSE. If the CAS is part of a condition for an if-statement, then we execute this code just before the if-statement, and replace the invocation of CAS with $result$. We then eliminate any *dead code* that cannot be executed. Figure 10 shows the transformed procedure, SCX_5.

More concretely, in place of the CAS at line 11 in SCX_4, we do the following. First, we check whether *r.info* $\quad$ *rinfo*. If so, we set *r.info* : $tseq_p$ and continue to the next iteration of the loop. Suppose not. If we were naively transforming the code, then the next step would be to check whether *r.info* contains $tseq_p$. However, p is the only process that can write $tseq_p$, and it only writes $tseq_p$ just before continuing to the next iteration. Thus, *r.info* cannot possibly contain $tseq_p$ in this case, which makes it unnecessary to check whether *r.info* $\quad tseq_p$. Therefore, we execute the *else*-case, and explicitly abort the transaction. Observe that, if SCX_5 is used to replace SCX_1 in Figure 6, then this explicit abort will cause SCX to return FALSE (right after it jumps to the onAbort label). In place of the CAS at line 16 in SCX_4, we can simply check whether fld contains *old* and, if so, write *new* into fld.

In fact, it is not necessary to check whether fld contains *old*, because the transaction will have aborted if fld was changed after *old* was read from it. We explain why. Let S be an invocation of SCX_5 (in Figure 10) by a process p, and let r be the Data-record that contains fld. Suppose S executes line 16 in SCX_5, where it checks whether $fld \quad old$. Before invoking S, p performs an invocation L of LLX(r) linked to S. Subsequently, p reads *old* while performing S. After that, p freezes r while performing S. If r changes after L, and before p executes line 11, then p will see *r.info* $\neq$ *rinfo* when it executes line 11 (by property P1, which has been preserved by our transformations). Consequently, p will fail to freeze r, and S will perform an explicit abort and return FALSE, so it will *not* reach line 16, which contradicts our assumption (so this case is impossible). On

```
1  Private variable for process p: tseq_p
2  SCX_4(V, R, fld, new) by process p
3      Let infoFields be a pointer to a table in p's private memory containing,
4  for each r in V, the value of r.info read by p's last LLX(r)
5      Let old be the value for fld returned by p's last LLX(r)
6      Begin hardware transaction
7      tseq_p := tseq_p + 2^⌈log n⌉                          ▷ increment p's tagged sequence number
8      ▷ Freeze all Data-records in V to protect their mutable fields from being changed by other SCXs
9      for each r in V enumerated in order do
10         Let rinfo be the pointer indexed by r in infoFields
11         if not CAS(r.info, rinfo, tseq_p) then            ▷ freezing CAS
12             if r.info ≠ tseq_p then Abort hardware transaction (explicitly)
13     ▷ Finished freezing Data-records
14     ▷ Finalize each r ∈ R, update fld, and unfreeze all r ∈(V \ R)
15     for each r ∈ R do r.marked : True                     ▷ mark step
16     CAS(fld, old, new)                                    ▷ update CAS
17     Commit hardware transaction
18     return True
```

Figure 9: Transforming SCX_O: after eliminating SCX-record creation on the fast path.

```
1  Private variable for process p: tseq_p
2  SCX_5(V, R, fld, new) by process p
3      Let infoFields be a pointer to a table in p's private memory containing,
4  for each r in V, the value of r.info read by p's last LLX(r)
5      Let old be the value for fld returned by p's last LLX(r)
6      Begin hardware transaction
7      tseq_p := tseq_p + 2^⌈log n⌉                          ▷ increment p's tagged sequence number
8      ▷ Freeze all Data-records in V to protect their mutable fields from being changed by other SCXs
9      for each r in V enumerated in order do
10         Let rinfo be the pointer indexed by r in infoFields
11         if r.info rinfo then r.info : tseq_p
12         else Abort hardware transaction (explicitly)
13     ▷ Finished freezing Data-records
14     ▷ Finalize each r ∈ R, update fld, and unfreeze all r ∈(V \ R)
15     for each r ∈ R do r.marked : True                     ▷ mark step
16     if fld old then fld : new
17     Commit hardware transaction
18     return True
```

Figure 10: Transforming SCX_O: after replacing CAS with sequential code and optimizing.

the other hand, if r changes after p executes line 11, and before p executes line 16, then the transaction will abort due to a data conflict (detected by the HTM system). Therefore, when p executes line 16, fld must contain old.

For the second optimization, we split the loop in Figure 10 into two. The first loop contains all of the steps that check whether *r.info* *rinfo*, and the second loop contains all of the steps that set *r.info* : $tseq_p$. This way, all of the writes to *r.info* occur after all of the reads and if-statements. The advantage of delaying writes for as long as possible in a transaction is that it reduces the probability of the transaction causing other transactions to abort. As a minor point, whereas the loop in SCX_O iterated over the elements of the sequence V in a particular order to guarantee progress, it is not necessary to do so here, since progress is guaranteed by the fallback path, not the fast path. This final transformation yields the code in Figure 4. Clearly, it does not affect correctness or progress.

5 ACCELERATED TEMPLATE IMPLEMENTATIONS

The *2-path con* algorithm. We now use our HTM-based LLX and SCX to obtain an HTM-based implementation of a template operation O. The fallback path for O is simply a lock-free implementation of O using LLX_O and SCX_O. The fast path for O starts a transaction, then performs the same code as the fallback path, except that it uses the HTM-based LLX and SCX. Since the *entire operation* is performed inside a transaction, we can optimize the invocations of SCX_{HTM} that are performed by O as follows. Lines 15 and 23 can be eliminated, since SCX_{HTM} is already running inside a large transaction. Additionally, lines 17-19 can be eliminated, since the transaction will abort due to a data conflict if *r.info* changes after it is read in the (preceding) linked invocation of LLX(r), and before the transaction commits. The proof of correctness and progress for *2-path con* follows immediately from the proof of the original template and the proof of the HTM-based LLX and SCX implementation.

Note that it is not necessary to perform the entire operation in a single transaction. In Section 8, we describe a modification that

```
1  Private variable for process p: tseq_p
2  SCX_HTM(V, R, fld, new) by process p
3      Let infoFields be a pointer to a table in p's private memory containing,
4  for each r in V, the value of r.info read by p's last LLX(r)
5      Let old be the value for fld returned by p's last LLX(r)
6      Begin hardware transaction
7      tseq_p := tseq_p + 2^⌈log n⌉                          ▷ increment p's tagged sequence number
8      for each r ∈ V do                                     ▷ abort if any r ∈ V has changed since the linked LLX(r)
9          Let rinfo be the pointer indexed by r in infoFields
10         if r.info ≠ rinfo then Abort hardware transaction (explicitly)
11     for each r ∈ V do r.info : tseq_p                     ▷ change r.info to a new value, for each r ∈ V
12     for each r ∈ R do r.marked : True                     ▷ mark each r ∈ R (so it will be finalized)
13     write new to the field pointed to by fld              ▷ perform the update
14     Commit hardware transaction
15     return True
```

Figure 11: Final implementation of SCX_{HTM}.

allows a read-only *searching* prefix of the operation to be performed before the transaction begins.

The *TLE* algorithm. To obtain a TLE implementation of an operation O, we simply take *sequential code* for O and wrap it in a transaction on the fast path. The fallback path acquires and releases a global lock instead of starting and committing a transaction, but otherwise executes the same code as the fast path. To prevent the fast path and fallback path from running concurrently, transactions on the fast path start by reading the lock state and aborting if it is held. An operation attempts to run on the fast path up to *AttemptLimit* times (waiting for the lock to be free before each attempt) before resorting to the fallback path. The correctness of TLE is trivial. Note, however, that TLE only satisfies deadlock-freedom (not lock-freedom).

The *2-path* $\overline{con}$ algorithm. We can improve concurrency on the fallback path and guarantee lock-freedom by using a lock-free algorithm on the fallback path, and a global fetch-and-increment object F instead of a global lock. Consider an operation O implemented with the tree update template. We describe a *2-path* $\overline{con}$ implementation of O. The fallback path increments F, then executes the lock-free tree update template implementation of O, and finally decrements F. The fast path executes *sequential code* for O in a transaction. To prevent the fast path and fallback path from running concurrently, transactions on the fast path start by reading F and aborting if it is nonzero. An operation attempts to run on the fast path up to *AttemptLimit* times (waiting for F to become zero before each attempt) before resorting to the fallback path.

Recall that operations implemented using the tree update template can only change a single pointer atomically (and can perform multiple changes atomically only by creating a connected set of new nodes that reflect the desired changes). Thus, each operation on the fallback path simply creates new nodes and changes a single pointer (and assumes that all other operations also behave this way). However, since the fast path and fallback path do not run concurrently, the fallback path does *not* impose this requirement on the fast path. Consequently, the fast path can make (multiple) direct changes to nodes. Unfortunately, as we described above, this algorithm can still suffer from concurrency bottlenecks.

The *3-path* algorithm. One can think of the *3-path* algorithm as a kind of hybrid between the *2-path con* and *2-path* $\overline{con}$ algorithms that obtains their benefits while avoiding their downsides. Consider an operation O implemented with the tree update template. We describe a 3-path implementation of O. As in *2-path* $\overline{con}$, there is a global fetch-and-increment object F, and the fast path executes *seqeuential code* for O in a transaction. The middle path and fallback path behave like the fast path and fallback path in the *2-path con* algorithm, respectively. Each time an operation begins (resp., stops) executing on the fallback path, it increments (resp., decrements) F. (If the scalability of fetch-and-increment is of concern, then a *scalable nonzero indicator* object [17] can be used, instead. Alternatively, one could use a *counter* object, which is weaker and can be implemented using only registers.) This prevents the fast and fallback paths from running concurrently. As we described above, operations begin on the fast path, and move to the middle path after *FastLimit* attempts, or if they see $F \neq 0$. Operations move from the middle path to the fallback path after *MiddleLimit* attempts. Note that an operation never waits for the fallback path to become empty—it simply moves to the middle path.

Since the fast path and fallback path do not run concurrently, the fallback path does not impose any overhead on the fast path, except checking if F 0 (offering low overhead for light workloads). Additionally, when there are operations running on the fallback path, hardware transactions can continue to run on the middle path (offering high concurrency for heavy workloads).

Correctness and progress for *3-path*. The correctness argument is straightforward. The goal is to prove that all template operations are linearizable, regardless of which path they execute on. Recall that the fallback path and middle path behave like the fast path and fallback path in *2-path con*. It follows that, if there are no operations on the fast path, then the correctness of operations on the middle path and fallback path is immediate from the correctness of *2-path con*. Of course, whenever there is an operation executing on the fallback path, no operation can run on the fast path. Since operations on the fast path and middle path run in transactions, they are atomic, and any conflicts between the fast path and middle path are handled automatically by the HTM system. Therefore, template operations are linearizable.

The progress argument for *3-path* relies on three assumptions.

A1. The sequential code for an operation executed on the fast path must terminate after a finite number of steps if it is run on a static tree (which does not change during the operation).

A2. In an operation executed on the middle path or fallback path, the search phase must terminate after a finite number of steps if it is run on a static tree.

A3. In an operation executed on the middle path or fallback path, the update phase can modify only a finite number of nodes.

We give a simple proof that *3-path* satisfies lock-freedom. To obtain a contradiction, suppose there is an execution in which after some time t, some process takes infinitely many steps, but no operation terminates. Thus, the tree does not change after t. We first argue that no process takes infinitely many steps in a transaction T. If T occurs on the fast path, then A1 guarantees it will terminate. If T occurs on the middle path, then A2 and A3 guarantee that it will terminate. Therefore, eventually, processes only take steps on the fallback path. Progress then follows from the fact that the original tree update template implementation (our fallback path) is lock-free.

6 EXAMPLE DATA STRUCTURES

We used two data structures to study the performance of our accelerated template implementations: an unbalanced BST, and a relaxed (a, b)-tree. The BST is similar to the chromatic tree in [8], but with no rebalancing. The relaxed (a, b)-tree is a concurrency-friendly generalization of a B-tree that is based on the work of Jacobson and Larsen [20]. In this section, we give a more detailed description of these data structures, and give additional details on their *3-path* implementations.

Each data structure implements the ordered dictionary ADT, which stores a set of keys, and associates each key with a value. An ordered dictionary offers four operations: INSERT(key, $value$), DELETE(key), SEARCH(key) and RANGEQUERY(lo, hi).

Both data structures are *leaf-oriented* (also called *external*), which means that all of the keys in the dictionary are stored in the leaves of the tree, and internal nodes contain *routing* keys which simply direct searches to the appropriate leaf. This is in contrast to *node-oriented* or *internal* trees, in which internal nodes also contain keys in the set.

6.1 Unbalanced BST

Fallback path. The fallback path consists of a lock-free implementation of the operations in Figure 12 using the (original) tree update template. As required by the template, these operations change child pointers, but do not change the key or value fields of nodes directly. Instead, to replace a node's key or value, the node is replaced by a new copy. If key is not already in the tree, then *Insert*(key, $value$) inserts a new leaf and internal node. Otherwise, *Insert*(key, $value$) replaces the leaf containing key with a new leaf that contains the updated value. *Delete*(key) replaces the leaf l being deleted and its parent with a new copy of the sibling of l.

It may seem strange that *Delete* creates a new copy of the deleted leaf's sibling, instead of simply reusing the existing sibling (which is not changed by the deletion). This comes from a requirement of the tree update template: each invocation of SCX(V, R, fld, new) must change the field fld to a value that it has *never previously contained*. This requirement is motivated by a particularly tricky aspect of lock-free programming: avoiding the *ABA problem*. The ABA problem occurs when a process p reads a memory location x and sees value A, then performs a CAS on x to change it from A to C, and *interprets* the success of this CAS to mean that x has not changed between when p read x and performed the CAS on it. In reality, after p read x and before it performed the CAS, another process q may have changed x to B, and then back to A, rendering p's interpretation invalid. In practice, the ABA problem can result in data structure operations being applied multiple times, or lost altogether. The ABA problem cannot occur if each successful CAS on a field stores a value that has never previously been contained in the field (since, then, q cannot change x from B back to A). So, in the template, the ABA problem is avoided by having each operation use SCX to store *a pointer to a newly created node* (which cannot have previously been contained in any field).

Middle path. The middle path is the same as the fallback path, except that each operation is performed in a large transaction, and the HTM-based implementation of LLX and SCX is used instead of the original implementation.

Fast path. The fast path is a sequential implementation of the BST, where each operation is executed in a transaction. Figure 13 shows the insertion and deletion operations on the fast path. Unlike on the fallback path, operations on the fast path directly modify the keys and values of nodes, and, hence, can avoid creating nodes in some situations. If key is already in the tree, then *Insert*(key, $value$) directly changes the value of the leaf that contains key. Otherwise, *Insert*(key, $value$) creates a new leaf and internal node and attaches them to the tree. *Delete*(key) changes a pointer to remove the leaf containing key and its parent from the tree.

How the fast path improves performance. The first major performance improvement on the fast path comes from a reduction in node creation. Each invocation of *Insert*(key, $value'$) that sees key in the tree can avoid creating a new node by writing $value'$ directly into the node that already contains key. In contrast, a new node had to be created on the middle path, since the middle path runs concurrently with the fallback path, which assumes that the keys and values of nodes do not change. Additionally, each invocation of *Delete* that sees key in the tree can avoid creating a new copy of the sibling of the deleted leaf. This optimization was not possible on the middle path, because the fallback path assumes that each successful operation writes a pointer to a newly created node. The second major improvement comes from the fact that reads and writes suffice where invocations of LLX and SCX were needed on the other paths.

6.2 Relaxed (a,b)-tree

The relaxed (a, b)-tree [20] is a generalization of a B-tree. Larsen introduced the relaxed (a, b)-tree as a sequential data structure that was well suited to fine-grained locking. Internal nodes contain up to $b - 1$ *routing* keys, and have one more child pointer than the number of keys. Leaves contain up to b key-value pairs (which are in the dictionary). (Values may be pointers to large data objects.) The *degree* of an internal node (resp. leaf) is the number of pointers (resp. keys) it contains. When there are no ongoing updates (insertions and deletions) in a relaxed (a, b)-tree, all leaves have the same depth, and nodes have *degree* at least a and at most b, where $b \geq 2a - 1$. Maintaining this balance condition requires rebalancing steps similar to the *splits* and *joins* of B-trees. (See [20] for further details on the rebalancing steps.)

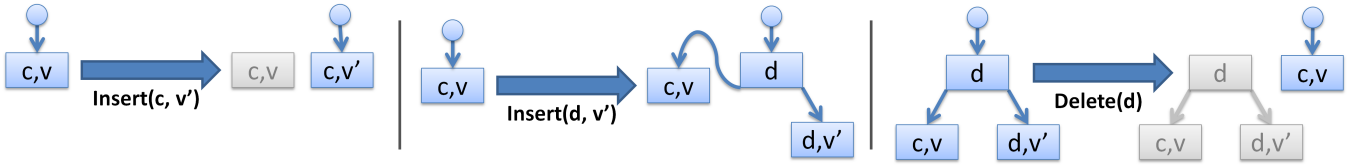

Figure 12: Fallback path operations for the unbalanced BST.

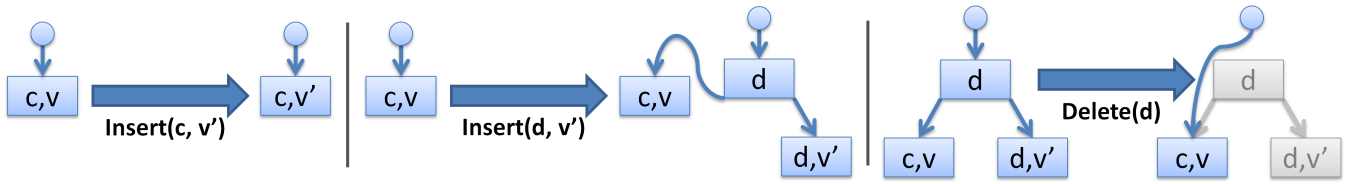

Figure 13: Fast path operations for the unbalanced BST. (*Insert*(d, v') is the same as on the fallback path.)

Fallback path. The fallback path consists of a lock-free implementation of the relaxed (a, b)-tree operations using the (original) tree update template. If *key* is in the tree, then *Insert*(*key*, *value*) replaces the leaf containing *key* with a new copy that contains (*key*, *value*). Suppose *key* is not in the tree. Then, *Insert* finds the leaf u where the key should be inserted. If u is not full (has degree less than b), then it is replaced with a new copy that contains (*key*, *value*). Otherwise, u is replaced by a subtree of three new nodes: one parent and two children. The two new children evenly share the key-value pairs of u and (*key*, *value*). The new parent p contains only a single routing key and two pointers (to the two new children), and is *tagged*, which indicates that the subtree rooted at p is too tall, and rebalancing should be performed to shrink its height. *Delete*(*key*) replaces the leaf containing *key* with a new copy *new* that has *key* deleted. If the degree of *new* is smaller than a, then rebalancing must be performed.

Middle path. This path is obtained from the fallback path the same way as in the unbalanced BST.

Fast path. The fast path is a sequential implementation of a relaxed (a, b)-tree whose operations are executed inside transactions. Like the external BST, the major performance improvement over the middle path comes from the facts that (1) operations create fewer nodes, and (2) reads and writes suffice where LLX and SCX were needed on the other paths. In particular, *Insert*(*key*, *value*) and *Delete*(*key*) simply directly modify the keys and values of leaves, instead of creating new nodes, except in the case of an *Insert* into a full node u. In that case, two new nodes are created: a parent and a sibling for u. (Recall that this case resulted in the creation of three new nodes on the fallback path and middle path.) Note that reducing node creation is more impactful for the relaxed (a, b)-tree than for the unbalanced BST, since nodes are much larger.

As a minor point, we found that it was faster in practice to perform rebalancing steps by creating new nodes, and simply replacing the old nodes with the new nodes that reflect the desired change (instead of rebalancing by directly changing the keys, values and pointers of nodes).

7 EXPERIMENTS

We used two different Intel systems for our experiments: a dual-socket 12-core E7-4830 v3 with hyperthreading for a total of 48 hardware threads (running Ubuntu 14.04LTS), and a dual-socket 18-core E5-2699 v3 with hyperthreading for a total of 72 hardware threads (running Ubuntu 15.04). Each machine had 128GB of RAM. We used the scalable thread-caching allocator (tcmalloc) from the Google perftools library. All code was compiled on GCC 4.8+ with arguments `-std=c++0x -O2 -mcx16`. (Using the higher optimization level `-O3` did not significantly improve performance for any algorithm, and decreased performance for some algorithms.) On both machines, we *pinned* threads such that we saturate one socket before scheduling any threads on the other.

Data structure parameters.. Recall that nodes in the relaxed a, b-tree contain up to b keys, and, when there are no ongoing updates, they contain at least a keys (where $b \geq 2a - 1$). In our experiments, we fix a 6 and b 16. With b 16, each node occupies four consecutive cache lines. Since $b \geq 2a - 1$, with b 16, we must have $a \leq 8$. We chose to make a slightly smaller than 8 in order to exploit a performance tradeoff: a smaller minimum degree may slightly increase depth, but decreases the number of rebalancing steps that are needed to maintain balance.

Template implementations studied. We implemented each of the data structures with four different template implementations: *3-path*, *2-path con*, *TLE* and the original template implementation, which we call *Non-HTM*. (*2-path* $\overline{con}$ is omitted, since it performed similarly to TLE, and cluttered the graphs.) The *2-path con* and *TLE* implementations perform up to 20 attempts on the fast path before resorting to the fallback path. *3-path* performs up to 10 attempts (each) on the fast path and middle path. We implemented memory reclamation using DEBRA [5], an epoch based reclamation scheme. A more efficient way to reclaim memory for *3-path* is proposed in Section 9

7.1 Light vs. Heavy workloads

Methodology. We study two workloads: in **light**, n processes perform updates (50% insertion and 50% deletion), and in **heavy**, $n - 1$

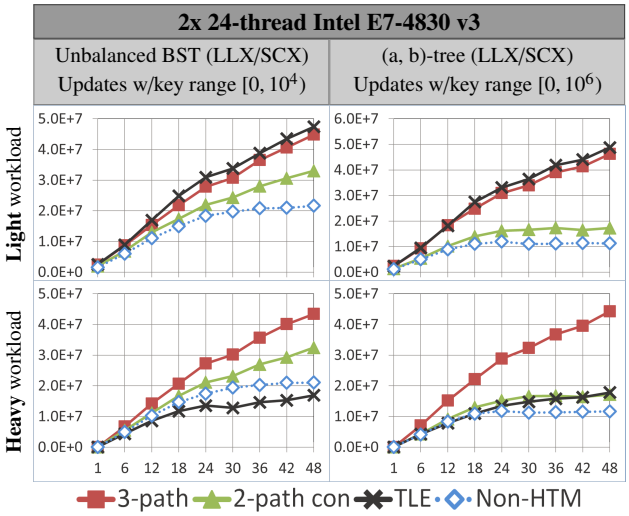

Figure 14: Results (48-thread system) showing throughput (operations per second) vs. concurrent processes.

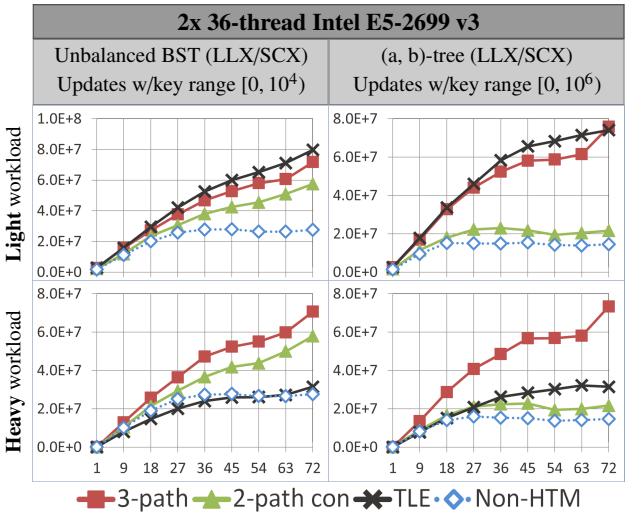

Figure 15: Results (72 thread system) showing throughput (operations per second) vs. concurrent processes.

processes perform updates, and one thread performs 100% range queries (RQs). For each workload and data structure implementation, and a variety of thread counts, we perform a set of five randomized trials. In each trial, n processes perform either updates or RQs (as appropriate for the workload) for one second, and counted the number of completed operations. Updates are performed on keys drawn uniformly randomly from a fixed key range $[0, K)$. RQs are performed on ranges $[lo, lo\ \ s)$ where lo is uniformly random in $[0, K)$ and s is chosen, according to a probability distribution described below, from $[1, 1000]$ for the BST and $[1, 10000]$ for the (a, b)-tree. (We found that nodes in the (a, b)-tree contained approximately 10 keys, on average, so the respective maximum values of s for the BST and (a, b)-tree resulted in range queries returning keys from approximately the same number of nodes in both data structures.) To ensure that we are measuring steady-state performance, at the start of each trial, the data structure is prefilled by having threads perform 50% insertions and 50% deletions on uniform keys until the data structure contains approximately half of the keys in $[0, K)$.

We verified the correctness of each data structure after each trial by computing *key-sum hashes*. Each thread maintains the sum of all keys it successfully inserts, minus the sum of all keys it successfully deletes. At the end of the trial, the total of these sums over all threads must match the sum of keys in the tree.

Probability distribution of s. We chose the probability distribution of s to produce many small RQs, and a smaller number of very large ones. To achieve this, we chose s to be $\lfloor x^2 S \rfloor$ 1, where x is a uniform real number in $[0, 1)$, and S 1000 for the BST and S 10000 for the (a, b)-tree. By squaring x, we bias the uniform distribution towards zero, creating a larger number of small RQs.

Results. We briefly discuss the results from the 48 thread machine, which appear in Figure 14. The BST and the relaxed (a, b)-tree behave fairly similarly. Since the (a, b)-tree has large nodes, it benefits much more from a low-overhead fast path (in *TLE* or *3-path*) which can avoid creating new nodes during updates. In the light workloads, *3-path* performs significantly better than *2-path con* (which has more overhead) and approximately as well as *TLE*. On average, the *3-path* algorithms completed 2.1x as many operations as their *non-HTM* counterparts (and with 48 concurrent processes, this increases to 3.0x, on average). In the heavy workloads, *3-path* significantly outperforms *TLE* (completing 2.0x as many operations, on average), which suffers from *excessive waiting*. Interestingly, *3-path* is also significantly faster than *2-path con* in the heavy workloads. This is because, even though RQs are always being performed, some RQs can succeed on the fast path, so many update operations can still run on the fast path in *3-path*, where they incur much less overhead (than they would in *2-path con*).

Results from the 72-thread machine appear in Figure 15. There, *3-path* shows an even larger performance advantage over *Non-HTM*.

7.2 Code path usage and abort rates

To gain further insight into the behaviour of our accelerated template implementations, we gathered some additional metrics about the experiments described above. Here, we only describe results from the 48-thread Intel machine. (Results from the 72-thread Intel machine were similar.)

Operations completed on each path. We started by measuring how often operations completed successfully on each execution path. This revealed that operations almost always completed on the fast path. Broadly, over all thread counts, the minimum number of operations completed on the fast path in any trial was 86%, and the average over all trials was 97%.

In each trial that we performed with 48 concurrent threads, at least 96% of operations completed on the fast path, *even in the workloads with RQs*. Recall that RQs are the operations most likely to run on the fallback path, and they are only performed by a single thread, so they make up a relatively small fraction of the total operations performed in a trial. In fact, our measurements showed that the number of

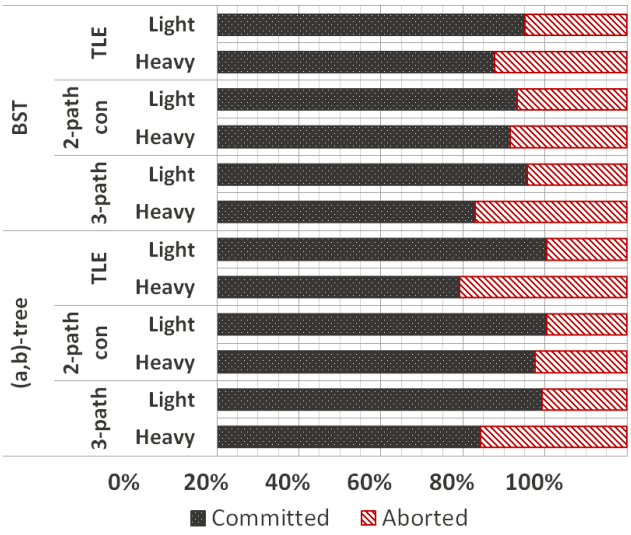

Figure 16: How many transactions commit vs. how many abort in our experiments on the 48-thread machine.

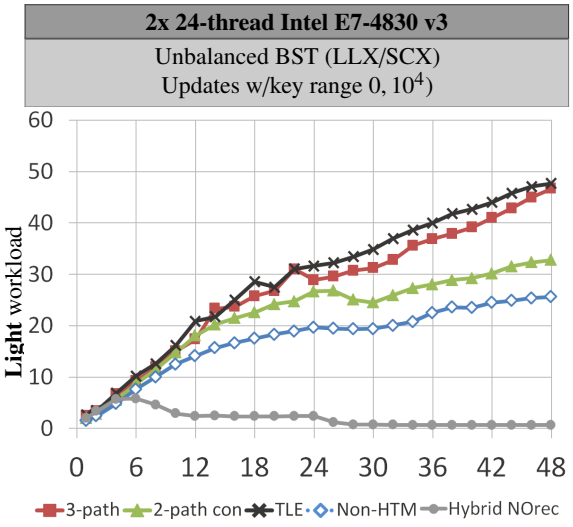

Figure 17: Results showing throughput (operations per second) versus number of processes for an unbalanced BST implemented with different tree update template algorithms, and with the hybrid TM algorithm *Hybrid NOrec*.

operations which completed on the fallback path was never more than a fraction of one percent in our trials with 48 concurrent threads.

In light of this, it might be somewhat surprising that the performance of TLE was so much worse in heavy workloads than light ones. However, the cost of serializing threads is high, and this cost is compounded by the fact that the operations which complete on the fallback path are often long-running. Of course, in workloads where more operations run on the fallback path, the advantage of improving concurrency between paths would be even greater.

Commit/abort rates. We also measured how many transactions committed and how many aborted, on each execution path, in each of our trials. Figure 16 summarizes the average commit/abort rates for each data structure, template implementation and workload. Since nearly all operations completed on the fast path, we decided not to distinguish between the commit/abort rate on the fast path and the commit/abort rate on the middle path.

7.3 Comparing with hybrid transactional memory

Hybrid transactional memory (hybrid TM) combines hardware and software transactions to hide the limitations of HTM and guarantee progress. This offers an alternative way of using HTM to implement concurrent data structures. Note, however, that state of the art hybrid TMs use locks. So, **they cannot be used to implement lock-free data structures.** Regardless, to get an idea of how such implementations would perform, relative to our accelerated template implementations, we implemented the unbalanced BST using Hybrid NOrec, which is arguably the fastest hybrid TM implementation with readily available code [29].

If we were to use a precompiled library implementation of Hybrid NOrec, then the unbalanced BST algorithm would have to perform a library function call for *each read and write to shared memory*, which would incur significant overhead. So, we directly compiled the code for Hybrid NOrec into the code for the BST, allowing the compiler to inline the Hybrid NOrec functions for reading and writing from shared memory into our BST code, eliminating this overhead. Of course, if one intended to use hybrid TM in practice (and not in a research prototype), one would use a precompiled library, with all of the requisite overhead. Thus, the following results are quite charitable towards hybrid TMs.

We implemented the BST using Hybrid NOrec by wrapping sequential code for the BST operations in transactions, and manually replacing each read from (resp., write to) shared memory with a read (resp., write) operation provided by Hybrid NOrec. Figure 17 compares the performance of the resulting implementation to the other BST implementations discussed in Section 7.

The BST implemented with Hybrid NOrec performs relatively well with up to six processes. However, beyond six processes, it experiences severe negative scaling. The negative scaling occurs because Hybrid NOrec increments a global counter in each updating transaction (i.e., each transaction that performs at least one write). This global contention hotspot in updating transactions causes many transactions to abort, simply because they contend on the global counter (and not because they conflict on any data in the tree). However, even without this bottleneck, Hybrid NOrec would still perform poorly in heavy workloads, since it incurs very high instrumentation overhead for software transactions (which must acquire locks, perform repeated validation of read-sets, maintain numerous auxiliary data structures for read-sets and write-sets, and so on). Note that this problem is not unique to Hybrid NOrec, as every hybrid TM must use a software TM as its fallback path in order to guarantee progress. In contrast, in our template implementations, the software-only fallback path is a fast lock-free algorithm.

8 MODIFICATIONS FOR PERFORMING SEARCHES OUTSIDE OF TRANSACTIONS

In this section, we describe how the *3-path* implementations of the unbalanced BST and relaxed (a, b)-tree can be modified so that each operation attempt on the fast path or middle path performs its search phase *before* starting a transaction (and only performs its update phase in a transaction). (The same technique also applies to the *2-path con* implementations.) First, note that the lock-free search procedure for each of these data structures is actually a standard, sequential search procedure. Consequently, a simple sequential search procedure will return the correct result, regardless of whether it is performed inside a transaction. (Generally, whenever we produce a *3-path* implementation starting from a lock-free fallback path, we will have access to a correct non-transactional search procedure.)

The difficulty is that, when an operation starts a transaction and performs its update phase, it may be working on a part of the tree that was deleted by another operation. One can imagine an operation O_d that deletes an entire subtree, and an operation O_i that inserts a node into that subtree. If the search phase of O_i is performed, then O_d is performed, then the update phase of O_i is performed, then O_i may erroneously insert a node into the deleted subtree.

We fix this problem as follows. Whenever an operation O on the fast path or middle path removes a node from the tree, it sets a *marked* bit in the node (just like operations on the fallback path do). Whenever O first accesses a node u in its transaction, it checks whether u has its *marked* bit set, and, if so, aborts immediately. This way, O's transaction will commit only if every node that it accessed is in the tree.

We found that this modification yielded small performance improvements (on the order of 5-10%) in our experiments. The reason this improves performance is that fewer memory locations are tracked by the HTM system, which results in fewer capacity aborts. We briefly discuss why the performance benefit is small in our experiments. The relaxed (a, b)-tree has a very small height, because it is balanced, and its nodes contain many keys. The BST also has a fairly small height (although it is considerably taller than the relaxed (a, b)-tree), because processes in our experiments perform insertions and deletions on uniformly random keys, which leads to trees of logarithmic height with high probability. So, in each case, the sequence of nodes visited by searches is relatively small, and is fairly unlikely to cause capacity aborts.

The performance benefit associated with this modification will be greater for data structures, operations or workloads in which an operation's search phase will access a large number of nodes. Additionally, IBM's HTM implementation in their POWER8 processors is far more prone to capacity aborts than Intel's implementation, since a transaction *will* abort if it accesses more than 64 different cache lines [26]. (In contrast, in Intel's implementation, a transaction can potentially commit after accessing tens of thousands of cache lines.) Thus, this modification could lead to significantly better performance on POWER8 processors.

9 MORE EFFICIENT MEMORY RECLAMATION ON THE FAST PATH

For the data structures presented in the paper, we implemented memory reclamation using an epoch based reclamation scheme called DEBRA [5]. This reclamation scheme is designed to reclaim memory for lock-free data structures, which are notoriously difficult to reclaim memory for. Since processes do not lock nodes before accessing them, one cannot simply invoke `free()` to release a node's memory back to the operating system as soon as the node is removed from the data structure. This is because a process can always be poised to access the node just after it is freed. The penalty for accessing a freed node is a program crash (due to a segmentation fault). Thus, reclamation schemes like DEBRA implement special mechanisms to determine when it is safe to free a node that has been removed from the data structure.

However, advanced memory reclamation schemes become unnecessary if *all* accesses to nodes are performed inside transactions. With Intel's HTM, accessing freed memory inside a transaction cannot cause a segmentation fault and crash the program. Instead, the transaction simply aborts. (Note, however, that this is not true for IBM's transactional memory implementation in their POWER8 processors.) Consider a graph-based data structure whose operations are performed entirely in transactions. In such a data structure, deleting and immediately freeing a node will simply cause any concurrent transaction that accesses the node (after it is freed) to abort. This is because removing the node will change a pointer that was traversed during any concurrent search that reached the node. Consequently, in such a data structure, reclaiming memory is as easy as invoking `free()` immediately after a node is removed.

In our three path algorithms, the fast path can only run concurrently with the middle path (but not the fallback path). Thus, if every operation on the fast path or middle path runs entirely inside a transaction, then memory can be reclaimed on the fast path simply by using `free()` immediately after removing a node inside a transaction. Our performance experiments did *not* implement this optimization, but doing so would likely further improve the performance of the three path algorithms.

10 OTHER USES FOR THE *3-PATH* APPROACH

10.1 Accelerating data structures that use read-copy-update (RCU)

In this section, we sketch a *3-path* algorithm for an ordered dictionary implemented with a node-oriented unbalanced BST that uses the RCU synchronization primitives. The intention is for this to serve as an example of how one might use the *3-path* approach to accelerate a data structure that uses RCU.

RCU is both a programming paradigm and a set of synchronization primitives. The paradigm organizes operations into a search/reader phase and an (optional) update phase. In the update phase, all modifications are made on a *new copy* of the data, and the old data is atomically replaced with the new copy. In this work, we are interested in the *RCU primitives* (rather than the paradigm).

Semantics of RCU primitives and their uses. The basic RCU synchronization primitives are *rcu_begin*, *rcu_end* and *rcu_wait* [12]. Operations invoke *rcu_begin* and *rcu_end* at the beginning and end of the search phase, respectively. The interval between an invocation of *rcu_begin* and the next invocation of *rcu_end* by the same operation is called a *read-side critical section*. An invocation of *rcu_wait* blocks until all read-side critical sections that started before the

invocation of *rcu_wait* have ended. One common use of *rcu_wait* is to wait, after a node has been deleted, until no readers can have a pointer to it, so that it can safely be freed. It is possible to use RCU as the sole synchronization mechanism for an algorithm if one is satisfied with allowing many concurrent readers, but only a single updater at a time. If multiple concurrent updaters are required, then another synchronization mechanism, such as fine-grained locks, must also be used. However, one must be careful when using locks with RCU, since locks cannot be acquired inside a read-side critical section without risking deadlock.

The CITRUS data structure. We consider how one might accelerate a node-oriented BST called CITRUS [2], which uses the RCU primitives, and fine-grained locking, to synchronize between threads. First, we briefly describe the implementation of CITRUS. At a high level, RCU is used to allow operations to search without locking, and fine-grained locking is used to allow multiple updaters to proceed concurrently.

The main challenge in the implementation of CITRUS is to prevent race conditions between searches (which do not acquire locks) and deletions. When an internal node u with two children is deleted in an internal BST, its key is replaced by its successor's key, and the successor (which is a leaf) is then deleted. This case must be handled carefully, or else the following can happen. Consider concurrent invocations D of *Delete*(*key*) and S of *Search*(key'), where key' is the successor of *key*. Suppose S traverses past the node u containing *key*, and then D replaces u's key by key', and deletes the node containing key'. The search will then be unable to find key', even though it has been in the tree throughout the entire search. To avoid this problem in CITRUS, rather than changing the key of u directly, D replaces u with a new copy that contains key'. After replacing u, D invokes *rcu_wait* to wait for any ongoing searches to finish, before finally deleting the leaf containing key'. The primary sources of overhead in this algorithm are invocations of *rcu_wait*, and lock acquisition costs.

Fallback path. The fallback path uses the implementation of CITRUS in [2] (additionally incrementing and decrementing the gobal fetch-and-add object F, as described in Section 5).

Middle path. The middle path is obtained from the fallback path by wrapping each fallback path operation in a transaction and optimizing the resulting code. The most significant optimization comes from an observation that the invocation of *rcu_wait* in *Delete* is unnecessary since transactions make the operation atomic. Invocations of *rcu_wait* are the dominating performance bottleneck in CITRUS, so this optimization greatly improves performance. A smaller improvement comes from the fact that transactions can avoid acquiring locks. Transactions on the middle path must ensure that all objects they access are not locked by other operations (on the fallback path), or else they might modify objects locked by operations on the fallback path. However, it is not necessary for transaction to actually *acquire* locks. Instead, it suffices for a transaction to simply *read* the lock state for all objects it accesses (before accessing them) and ensure that they are not held by another process. This is because transactions *subscribe* to each memory location they access, and, if the value of the location (in this case, the lock state) changes, then the transaction will abort.

Fast path. The fast path is a sequential implementation of a node-oriented BST whose operations are executed in transactions. As in the other 3-path algorithms, each transaction starts by reading F, and aborts if it is nonzero. This prevents operations on the fast path and fallback path from running concurrently. There are two main differences between fast path and the middle path. First, the fast path does not invoke *rcu_begin* and *rcu_end*. These invocations are unnecessary, because operations on the fast path can run concurrently *only* with other operations on the fast path or middle path, and *neither* path depends on RCU for its correctness. (However, the middle path *must* invoke these operations, because it runs concurrently with the fallback path, which relies on RCU.) The second difference is that the fast path does not need to read the lock state for any objects. Any conflicts between operations on the fast and middle path are resolved directly by the HTM system.

10.2 Accelerating data structures that use k-CAS

In this section, we sketch a *3-path* algorithm for an ordered dictionary implemented with a singly-linked list that uses the k-CAS synchronization primitive. The intention is for this to serve as an example of how one might use the *3-path* approach to accelerate a data structure that uses k-CAS.

A k-CAS operation takes, as its arguments, k memory locations, expected values and new values, and atomically: reads the memory locations and, if they contain their expected values, writes new values into each of them. k-CAS has been implemented from single-word CAS [18]. We briefly describe this k-CAS implementation. At a high level, a k-CAS creates a descriptor object that describes the k-CAS operation, then uses CAS to store a pointer to this descriptor in each memory location that it operates on. Then, it uses CAS to change each memory location to contain its new value. While a k-CAS is in progress, some fields may contain pointers to descriptor objects, instead of their regular values. Consequently, reading a memory location becomes more complicated: it requires reading the location, then testing whether it contains a pointer to a descriptor object, and, if so, helping the k-CAS operation that it represents, before finally returning a value.

Fallback path. The fallback path consists of the lock-free singly-linked list in [30]. At a high level, each operation on the fallback path consists of a search phase, optionally followed by an update phase, which is performed using k-CAS.

Middle path. Since the search phase in a linked list can be extremely long, and is likely to cause a transaction to abort (due to capacity limitations), the middle path was obtained by wrapping *only the update phase* of each fallback path operation in a transaction, and optimizing the resulting code. The main optimization on the middle path comes from replacing the software implementation of k-CAS with straightforward implementation from HTM (using the approach in [30]). This HTM-based implementation performs the entire k-CAS atomically, so it does not need to create a descriptor, or store pointers to descriptors at nodes.

Fast path. The fast path is a sequential implementation in which the *update phase* of each operation is wrapped in a transaction. The main optimization on the fast path comes from the fact that, since there are no concurrent operations on the fallback path, there are no k-CAS descriptors in shared memory. Consequently, operations on

the fast path do not need to check whether any values they read from shared memory are actually pointers to k-CAS descriptors, which can significantly reduce overhead.

Preventing fast/fallback concurrency. Observe that our fast path optimization (to avoid checking whether any values that are read are actually pointers to k-CAS descriptors) is correct only if the search phase in the fast path does not run concurrently with the update phase of any operation on the fallback path. For each of the other data structures we described, each operation runs entirely inside a single transaction. Thus, for these data structures, it suffices to verify that the global fetch-and-add object F is zero at the beginning of each transaction to guarantee that operations on the fast path do not run concurrently with operations on the fallback path. However, this is not sufficient for the list, since only the update phase of each operation executes inside a transaction. So, we need some extra mechanism to ensure that the fast path does not run concurrently with the fallback path.

If it is not important for the algorithm to be lock-free, then one can simply use a fast form of group mutual exclusion that allows many operations on the fast path, or many operations on the fallback path, but not both. Otherwise, one can solve this problem by splitting the traversal into many small transactions, and verifying that F is zero at the beginning of each. If F ever becomes non-zero, then some transaction will abort, and the enclosing operation will also abort.

11 RELATED WORK

Hybrid TMs share some similarities to our work, since they all feature multiple execution paths. The first hybrid TM algorithms allowed HTM and STM transactions to run concurrently [11, 21]. Hybrid NOrec [10] and Reduced hardware NOrec [25] are hybrid TMs that both use global locks on the fallback path, eliminating any concurrency. We discuss two additional hybrid TMs, Phased TM [22] (PhTM) and Invyswell [9], in more detail.

PhTM alternates between five *phases*: HTM-only, STM-only, concurrent HTM/STM, and two global locking phases. Roughly speaking, PhTM's HTM-only phase corresponds to our uninstrumented fast path, and its concurrent HTM/STM phase corresponds to our middle HTM and fallback paths. However, their STM-only phase (which allows no concurrent hardware transactions) and global locking phases (which allow no concurrency) have no analogue in our approach. In heavy workloads, PhTM must oscillate between its HTM-only and concurrent HTM/STM phases to maximize the performance benefit it gets from HTM. When changing phases, PhTM typically waits until all in-progress transactions complete before allowing transactions to begin in the new mode. Thus, after a phase change has begun, and before the next phase has begun, there is a window during which new transactions must wait (reducing performance). One can also think of our three path approach as proceeding in two phases: one with concurrent fast/middle transactions and one with concurrent middle/fallback transactions. However, in our approach, "phase changes" do not introduce any waiting, and there is always concurrency between two execution paths.

Invyswell is closest to our three path approach. At a high level, it features an HTM middle path and STM slow path that can run concurrently (sometimes), and an HTM fast path that can run concurrently with the middle path (sometimes) but not the slow path, and two global locking fallback paths (that prevent any concurrency). Invyswell is more complicated than our approach, and has numerous restrictions on when transactions can run concurrently. Our three path methodology does not have these restrictions. The HTM fast path also uses an optimization called lazy subscription. It has been shown that lazy subscription can cause opacity to be violated, which can lead to data corruption or program crashes [13].

Hybrid TM is very general, and it pays for its generality with high overhead. Consequently, data structure designers can extract far better performance for library code by using more specialized techniques. Additionally, we stress that state of the art hybrid TMs use locks, so they cannot be used in lock-free data structures.

Different options for concurrency have recently begun to be explored in the context of TLE. Refined TLE [14] and Amalgamated TLE [1] both improve the concurrency of TLE when a process is on the fallback path by allowing HTM transactions to run concurrently with a *single process* on the fallback path. Both of these approaches still serialize processes on the fallback path. They also use locks, so they cannot be used to produce lock-free data structures.

Timnat, Herlihy and Petrank [30] proposed using a strong synchronization primitive called *multiword compare-and-swap* (k-CAS) to obtain fast HTM algorithms. They showed how to take an algorithm implemented using k-CAS and produce a two-path implementation that allows concurrency between the fast and fallback paths. One of their approaches used a lock-free implementation of k-CAS on the fallback path, and an HTM-based implementation of k-CAS on the fast path. They also experimented with two-path implementations that do not allow concurrency between paths, and found that allowing concurrency between the fast path and fallback path introduced significiteant overhead. Makreshanski, Levandoski and Stutsman [24] also independently proposed using HTM-based k-CAS in the context of databases.

Liu, Zhou and Spear [23] proposed a methodology for accelerating concurrent data structures using HTM, and demonstrated it on several lock-free data structures. Their methodology uses an HTM-based fast path and a non-transactional fallback path. The fast path implementation of an operation is obtained by encapsulating part (or all) of the operation in a transaction, and then applying sequential optimizations to the transactional code to improve performance. Since the optimizations do not change the code's logic, the resulting fast path implements the same logic as the fallback path, so both paths can run concurrently. Consequently, the fallback path imposes overhead on the fast path.

Some of the optimizations presented in that paper are similar to some optimizations in our HTM-based implementation of LLX and SCX. For instance, when they applied their methodology to the lock-free unbalanced BST of Ellen et al. [16], they observed that helping can be avoided on the fast path, and that the descriptors which are normally created to facilitate helping can be replaced by a small number of statically allocated descriptors. However, they did not give details on exactly how these optimizations work, and did not give correctness arguments for them. In contrast, our optimizations are applied to a more complex algorithm, and are proved correct.

Multiversion concurrency control (MVCC) is another way to implement range queries efficiently [3, 4]. At a high level, it involves maintaining multiple copies of data to allow read-only transactions to see a consistent view of memory and serialize even in the presence of concurrent modifications. However, our approach could also be applied to operations that *modify* a range of keys, so it is more general than MVCC.

12 CONCLUDING REMARKS

In this work, we explored the design space for HTM-based implementations of the tree update template of Brown et al. and presented four accelerated implementations. We discussed performance issues affecting HTM-based algorithms with two execution paths, and developed an approach that avoids them by using three paths. We used our template implementations to accelerate two different lock-free data structures, and performed experiments that showed significant performance improvements over several different workloads. This makes our implementations an attractive option for producing fast concurrent data structures for inclusion in libraries, where performance is critical.

Our accelerated data structures each perform an entire operation inside a single transaction (except on the fallback code path, where no transactions are used). We discussed how one can improve efficiency by performing the read-only *searching* part of an operation non-transactionally, and simply using a transaction to perform any modifications to the data structure. Our *3-path* approach may also have other uses. As an example, we sketched an accelerated *3-path* implementation of a node-oriented BST that uses the read-copy-update (RCU) synchronization primitives. We suspect that a similar approach could be used to accelerate other data structures that use RCU. Additionally, we described how one might produce a *3-path* implementation of a lock-free algorithm that uses the *k*-CAS synchronization primitive.

REFERENCES

[1] Y. Afek, A. Matveev, O. R. Moll, and N. Shavit. Amalgamated lock-elision. In *Distributed Computing: 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 309–324. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[2] M. Arbel and H. Attiya. Concurrent updates with rcu: search tree as an example. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 196–205. ACM, 2014.

[3] H. Attiya and E. Hillel. A single-version stm that is multi-versioned permissive. *Theory of Computing Systems*, 51(4):425–446, 2012.

[4] P. A. Bernstein and N. Goodman. Multiversion concurrency control-theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[5] T. Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 261–270, 2015.

[6] T. Brown. *Techniques for Constructing Efficient Data Structures*. PhD thesis, University of Toronto, 2017.

[7] T. Brown, F. Ellen, and E. Ruppert. Pragmatic primitives for non-blocking data structures. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 13–22, 2013.

[8] T. Brown, F. Ellen, and E. Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 329–342, 2014.

[9] I. Calciu, J. Gottschlich, T. Shpeisman, G. Pokam, and M. Herlihy. Invyswell: a hybrid transactional memory for haswell's restricted transactional memory. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 187–200. ACM, 2014.

[10] 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 Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 39–52, New York, NY, USA, 2011. ACM.

[11] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 336–346, New York, NY, USA, 2006. ACM.

[12] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. User-level implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):375–382, 2012.

[13] D. Dice, T. Harris, A. Kogan, Y. Lev, and M. Moir. Pitfalls of lazy subscription. In *Proceedings of the 6th Workshop on the Theory of Transactional Memory, Paris, France*, 2014.

[14] D. Dice, A. Kogan, and Y. Lev. Refined transactional lock elision. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 19:1–19:12, New York, NY, USA, 2016. ACM.

[15] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 157–168, New York, NY, USA, 2009.

[16] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 131–140, 2010. Full version available as Technical Report CSE-2010-04, York University.

[17] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. Snzi: Scalable nonzero indicators. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 13–22. ACM, 2007.

[18] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 265–279, 2002.

[19] M. He and M. Li. Deletion without rebalancing in non-blocking binary search trees. In *Proceedings of the 20th International Conference on Principles of Distributed Systems*, 2016.

[20] L. Jacobsen and K. S. Larsen. Variants of (a, b)-trees with relaxed balance. *Int. J. Found. Comput. Sci.*, 12(4):455–478, 2001.

[21] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 209–220, New York, NY, USA, 2006. ACM.

[22] Y. Lev, M. Moir, and D. Nussbaum. Phtm: Phased transactional memory. In *Workshop on Transactional Computing (Transact)*, 2007.

[23] Y. Liu, T. Zhou, and M. Spear. Transactional acceleration of concurrent data structures. In *Proc. of 27th ACM Sym. on Parallelism in Algorithms and Arch.*, SPAA '15, pages 244–253, New York, NY, USA, 2015. ACM.

[24] D. Makreshanski, J. Levandoski, and R. Stutsman. To lock, swap, or elide: on the interplay of hardware transactional memory and lock-free indexing. *Proceedings of the VLDB Endowment*, 8(11):1298–1309, 2015.

[25] A. Matveev and N. Shavit. Reduced hardware norec: A safe and scalable hybrid transactional memory. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 59–71, New York, NY, USA, 2015. ACM.

[26] A. T. Nguyen. *Investigation of Hardware Transactional Memory*. PhD thesis, Massachusetts Institute of Technology, 2015.

[27] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 294–305. IEEE Computer Society, 2001.

[28] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 5–17, New York, NY, USA, 2002. ACM.

[29] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing hybrid transactional memory: The importance of nonspeculative operations. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 53–64. ACM, 2011.

[30] S. Timnat, M. Herlihy, and E. Petrank. A practical transactional memory interface. In *Euro-Par 2015: Parallel Processing*, pages 387–401. Springer, 2015.