Nested Parallelism in Transactional Memory

Kunal Agrawal Jeremy T. Fineman Jim Sukha

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
{kunal_ag, jfineman, sukhaj}@mit.edu

Abstract

This paper investigates adding transactions with nested parallelism and nested transactions to a dynamically multithreaded parallel programming language that generates only series-parallel programs. We describe XConflict, a data structure that facilitates conflict detection for a software transactional memory system which supports transactions with nested parallelism and unbounded nesting depth. For languages that use a Cilk-like work-stealing scheduler, XConflict answers concurrent conflict queries in $O(1)$ time and can be maintained efficiently. In particular, for a program with T_1 work and a span (or critical-path length) of T_∞, the running time on p processors of the program augmented with XConflict is only $O(T_1/p + pT_\infty)$.

Using XConflict, we describe CWSTM, a runtime-system design for software transactional memory which supports transactions with nested parallelism and unbounded nesting depth of transactions. The CWSTM design provides transactional memory with eager updates, eager conflict detection, strong atomicity, and lazy cleanup on aborts. In the restricted case when no transactions abort and there are no concurrent readers, CWSTM executes a transactional computation on p processors also in time $O(T_1/p + pT_\infty)$. Although this bound holds only under rather optimistic assumptions, to our knowledge, this result is the first theoretical performance bound on a TM system that supports transactions with nested parallelism which is independent of the maximum nesting depth of transactions.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming —Parallel programming; E.1 [*Data Structures*]: Distributed data structures

General Terms Algorithms, Theory

Keywords Cilk, data structure, fork-join, multithreading, nested parallel computations, series-parallel computations, transactional memory, transaction conflict detection, work stealing.

This research was supported in part by NSF Grants CNS-0615215 and CNS-0540248 and a grant from Intel Corporation.

A preliminary version of this paper also appeared in The Second ACM SIGPLAN Workshop on Transaction Computing. This workshop has no printed proceedings.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'08, February 20–23, 2008, Salt Lake City, Utah, USA.
Copyright © 2008 ACM 978-1-59593-960-9/08/0002…$5.00

1. INTRODUCTION

Transactional memory (TM) (Herlihy and Moss 1993) represents a collection of hardware and software mechanisms that help provide a transactional interface for accessing memory to programmers writing parallel code. Recently, TM has been an active area of study; for example, researchers have proposed many designs for transactional memory systems, with support for TM in hardware (e.g. (Hammond et al. 2004; Ananian et al. 2006; Moore et al. 2006)), in software (e.g., (Herlihy et al. 2003; Marathe et al. 2006; Adl-Tabatabai et al. 2006)), or hybrids of hardware and software (e.g., (Damron et al. 2006; Kumar et al. 2006)). A typical TM runtime system executes transactions optimistically, aborting and rolling back transactions that "conflict" to guarantee that transactions appear to execute atomically.

Most work on transactional memory focuses exclusively on supporting transactions in programs that use persistent threads (e.g., pthreads). TM systems for such an environment are designed assuming that transactions execute serially, since the overhead of creating or destroying a pthread naturally discourages programmers from having nested parallelism inside a transaction. Furthermore, the special case of serial transactions greatly simplifies conflict detection for TM; typically, the TM runtime detects a ***conflict*** between two distinct active transactions if they both access the same object ℓ, at least one transaction tries to write to ℓ, and both transactions are executed on different threads. This last condition is relevant for TM that supports *nested transactions*. Conceptually, when transactions execute serially, two active transactions executing on the same thread are allowed to access the same object because one must be nested inside the other.

Another way to write parallel programs, however, is to use dynamic multithreaded languages such as Cilk (Blumofe et al. 1996; Supercomputing) or NESL (Blelloch and Greiner 1996) or multithreaded libraries like Hood (Blumofe and Papadopoulos 1999). In such languages, a programmer specifies dynamic parallelism using linguistic constructs such as "fork" and "join," "spawn" and "sync," or "parallel" blocks. Dynamic multithreaded languages allow the programmer to specify a program's parallel structure abstractly, that is, permitting (but not requiring) parallelism in specific locations of the program. A runtime system (e.g., for Cilk) dynamically schedules the program on the number of processors, p, specified at execution time. If the language also permits only "properly nested" parallelism, i.e., any program execution can be represented as a "series-parallel dag" or "series-parallel parse tree" (Feng and Leiserson 1997), then a Cilk-like work-stealing scheduler executes the program in a provably efficient manner (Blumofe et al. 1996). A natural question arises: how can transactions be added to a dynamic multithreaded language such as Cilk?

To pose the problem more concretely, consider the series-parallel program shown in Figure 1, which performs parallel in-

```
PARALLELINCREMENT()
1  x ← 0
2  parallel
3     {  x ← x + 1                    }
4     {  x ← x + 10
5        parallel
6              {  x ← x + 100   }
7              {  x ← x + 1000  }    }
8  print x
```

Figure 1. A simple fork-join program that does several parallel increment of a shared variable. The **parallel** statement, similar to Dijkstra's "cobegin," allows the two following code blocks (each contained in $\{.\}$) to run in parallel. The subsequent line (line 8) executes after *both* parallel blocks complete. This program contains several races—assuming sequential consistency, valid outputs are $x \in \{1, 11, 101, 110, 111, 1001, 1010, 1011, 1101, 1110, 1111\}$.

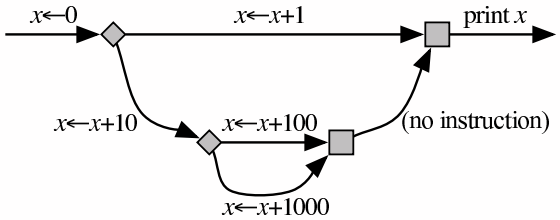

Figure 2. The series-parallel dag for the sample program given in Figure 1. Edges correspond to instructions in the program. Diamonds and squares correspond to the start and end, respectively, of parallel constructs.

crements to a shared variable. Figure 2 gives the corresponding series-parallel dag for the program. One natural way to add transactions to a series-parallel program is by wrapping segments of the program in atomic blocks, as illustrated by Figure 3. As shown, it is easy to generate transactions (e.g., X_3) with nested parallelism and nested transactions (e.g., X_4). How does a TM system execute the program in Figure 3?

This paper investigates adding transactions to a dynamic multithreaded language that generates only series-parallel programs. We focus on a provably efficient TM system that supports unbounded nesting and parallelism. That is, we want a TM system with a bound on a program's completion time that is independent of the maximum nesting depth of transactions. It turns out that TMs that perform work on every transaction commit proportional to the size of the transaction's "readset" or "writeset" cannot support an unbounded nesting depth efficiently. Generally, TM with lazy conflict detection requires work proportional to the size of the transactions readset, and TM with lazy updates require work proportional to the size of the transaction's writeset. Thus, we focus on TM with eager conflict detection and eager updates, since both require only a constant amount of work on every commit.

A key component of a TM system with nested parallelism is the conflict-detection scheme. We describe the semantics for TM with eager conflict detection for series-parallel computations with transactions. We present XConflict, a data structure that a software TM system can use to query for conflicts when implementing these semantics. For Cilk-like work-stealing schedulers, the XConflict answers concurrent queries in $O(1)$ time and can be maintained efficiently. In particular, consider a program with T_1 ***work*** and a

```
XPARALLELINCREMENT()
 1  atomic {                                   ▷ Transaction X1
 2     x ← 0
 3     parallel
 4        {  atomic {x ← x + 1}                    }    ▷ X2
 5        {  atomic {                                   ▷ X3
 6              x ← x + 10
 7              parallel
 8                 {  x ← x + 100                 }
 9                 {  atomic {x ← x + 1000}       }        ▷ X4
10           }                                        }
11  }
12  print x
```

Figure 3. The program from Figure 1 with the addition of some transactions, denoted by **atomic**$\{.\}$ blocks. The triangle denotes a comment. Since atomic blocks are not placed around *all* increments, this program still permits multiple outputs—valid outputs are 111 and 1111. The (symmetric) 1011 is excluded due to strong atomicity.

span (or critical-path length) of T_∞, (In terms of the series-parallel dag, work is the number of edges, i.e., 7 for Figure 2. Span is the longest path through the dag, i.e., 5 for Figure 2.) Then the running time on p processors of the program augmented with XConflict is only $O(T_1/p + pT_\infty)$. In comparison, with high probability, Cilk executes the program without XConflict in the asymptotically optimal time $O(T_1/p + T_\infty)$. These two bounds imply that maintaining the XConflict data structure does not asymptotically increase the running time of the program, compared to Cilk, when $\sqrt{T_1/T_\infty} \gg P$.

We describe CWSTM, a design for a software TM system with eager updates that uses the XConflict data structure. CWSTM provides strong atomicity and supports lazy cleanup on aborts (i.e., when a transaction X aborts, other transactions can help roll back the objects modified by X). The XConflict bounds translate to CWSTM in a restricted case when there are no concurrent readers (all memory accesses are treated as writes) and there are no transaction abort. If the underlying transaction-free program has T_1 work and T_∞ span, then the CWSTM executes the transactional program in time $O(T_1/p + pT_\infty)$ when run on p processors. At first glance, these bounds might seem uninteresting due to the restrictions. It is difficult, however, for any TM system to provide any nontrivial bounds on completion time in the presence of aborts, since the system might redo an arbitrary amount of work. Moreover, TM with eager conflict detection that allows more than a constant number of shared readers to an object can potentially lead to memory contention; thus, even if there are no conflicts on that object, it seems difficult to provide efficient worst-case theoretical bounds.

The remainder of this paper is organized as follows. We use a computation tree to model a computation with nested parallel transactions; Section 2 describes the computation tree and how the CWSTM runtime system maintains this computation tree online. Section 3 defines our CWSTM semantics. We show in Section 4 that a naive conflict-detection algorithm has poor worst-case performance. Section 5 describes the high-level design of CWSTM and its use of XConflict for conflict-detection. Section 6 gives an overview of the XConflict algorithm. Sections 7–10 provide details on data structures used by XConflict. Finally, Section 11 claims that XConflict, and hence CWSTM, is efficient for programs that experience no conflicts or contention.

2. CWSTM FRAMEWORK

This section presents the computation-tree framework we use to model CWSTM program executions. A program execution is modeled as an ordered tree, called the computation tree (as in (Agrawal et al. 2006)). The computation tree is not given *a priori* (i.e., from a static analysis of the program); rather, it unfolds dynamically as the program executes. Moreover, nondeterminism in the program may result in different computation trees. Constructing the computation tree on the fly as the program executes is not difficult and thus not described in full in this paper. A partial program execution corresponds to partial traversal of the computation tree.

A computation tree has two types of nodes: leaf nodes correspond to single memory operations, while internal nodes model the nested parallel control structure of the program as well as the structure of nested transactions. As in the "series-parallel parse tree" of (Feng and Leiserson 1997), an internal node is either an ***S-node*** or a ***P-node***. An S-node denotes series composition of the subtrees (i.e., the subtrees must execute in left-to-right order), whereas a P-node denotes parallel composition. Transactions are specified in a computation tree by marking a subset of S-nodes as transactions. A computation-tree node B is a descendant of a particular transaction node X if and only if B is contained in X's transaction. Consequently, a transaction Y is ***nested*** inside a transaction X if X is an ancestor of Y in the computation tree. We consider only computations that have closed-nested transactions.

In our canonical computation tree, all P-nodes have exactly 2 nontransactional S-nodes as children, while S-nodes can have an arbitrary number of children. In addition, we require that no nontransactional S-node has a child nontransactional S-node. Thus, it follows that all nontransactional S-nodes are children of P-nodes (or the root of the tree). For convenience, we treat the root of the computation tree as both a transactional and a nontransactional S-node.

We define several notations for the computation tree $\mathcal{C}$. Let $\texttt{root}(\mathcal{C})$ denote the tree's root node. For any node $B \neq \texttt{root}(\mathcal{C})$, $\texttt{parent}[B]$ denotes B's parent in $\mathcal{C}$. If B is an internal node of $\mathcal{C}$, then $\texttt{children}(B)$ is the ordered set of B's children, $\texttt{ances}(B)$ is the set of B's ancestors and $\texttt{desc}(B)$ is B's descendants. In this paper, whenever we refer to the set of ancestors or descendants of a node B, we include B in this set.

For any node $B \neq \texttt{root}(\mathcal{C})$, we define the transactional parent $\texttt{xparent}[B]$ as $Z = \texttt{parent}[B]$ if Z is a transaction or $\texttt{root}(\mathcal{C})$, and as $\texttt{xparent}[Z]$ otherwise. Similarly, we define nontransactional S-node parent $\texttt{nsParent}[B]$ as $Z = \texttt{parent}[B]$ if Z is a nontransactional S-node or $\texttt{root}(\mathcal{C})$, or $\texttt{nsParent}[Z]$ otherwise.

At any point during the computation-tree traversal, each node B in the computation tree has a ***status***, denoted by $\texttt{status}[B]$. The status can be one of `PENDING`, `PENDING_ABORT`, `COMMITTED`, or `ABORTED`. A leaf is ***complete*** if the corresponding operation has been executed. An internal node can complete only if all nodes in its subtree are complete. Thus, a complete node corresponds to the root of a subtree that will not unfold further, and hence a node is complete if and only if its status is `COMMITTED` or `ABORTED`. A node is ***active*** (having status `PENDING` or `PENDING_ABORT`) if it has any unexecuted descendants. Once a node is complete, it can never become active again.

Any execution of the computation tree has the invariant that at any time, the set of active nodes in the computation tree also forms a tree, with the leaves of this active tree being the set of ***ready*** nodes. Only a node that is ready can be traversed to "discover" a new child node. (Discovering a new child node corresponds to executing an instruction in the program: a read or write creates new leaf below the current S-node, a transactional begin creates a new transactional S-node, and a fork statement creates a new P-node along with its two S-node children.) When a ready transactional S-node completes, its parent becomes ready. When a ready nontransactional S-node Z completes, if Z's sibling nontransactional S-node is already complete, then Z's parent (which is a P-node) completes, and Z's grandparent becomes ready.

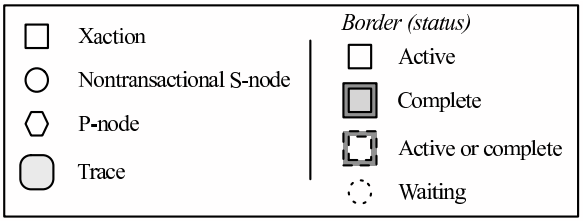

Figure 4. A legend for computation-tree figures.

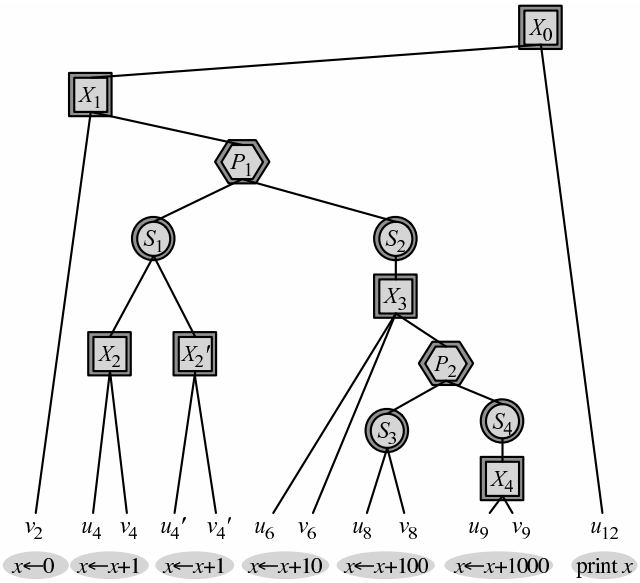

Figure 5. A computation tree for an execution of the program given by Figure 1, in which transaction X_2 aborted once and was retried as the transaction X_2'. The root X_0 does not correspond to any transaction in the program—it is just the S-node root of the tree. Each increment to x on line j of the program decomposes into two atomic memory operations: a `read` u_j, and a `write` v_j. The corresponding code is shown in a gray oval under the accesses.

Figure 5 shows the structure of the computation tree after an execution of the code from Figure 1 in which transaction X_2 aborts once. If other aborts (and retries) occur, the computation tree would have additional subtrees.

3. CWSTM SEMANTICS

This section describes CWSTM semantics, a semantics for a generic transactional memory system with nested parallel transactions and eager conflict detection. We describe these semantics operationally, in terms of a "readset" and "writeset" for each transaction. In particular, we define conflicts and describe transaction commits and aborts abstractly using readsets and writesets. Later, in Section 4, we give a simple design for a TM that provides these semantics. In Section 5, we improve the simple TM and present the the CWSTM design.

At any point during the program execution, the ***readset*** of a transaction X is the set of objects ℓ that X has "accessed". Similarly, the ***writeset*** is a set of objects ℓ that X has written to. Operationally, readsets and writesets change as follows. A transaction begins with an empty readset and empty writeset. Whenever a successful `read` of ℓ_1 occurs in a memory-operation (leaf) node u, ℓ_1 is added to $\texttt{xparent}[u]$'s readset. Similarly, whenever a successful `write` of ℓ_2 occurs in a memory-operation node u, ℓ_2 is added to $\texttt{xparent}[u]$'s readset and writeset. A `read` or a `write` to ℓ by an operation u "observes" the value associated with the write stored in the writeset of Z, where Z is the nearest transactional ancestor of u that contains ℓ in its writeset. For consistency, the writeset of the computation-tree's root contains all objects. When a transaction X commits, its readset and writeset are merged into $\texttt{xparent}[X]$'s readset and writeset, respectively.

A transactional memory with eager conflict detection must test for conflict before performing each `read` or `write`. An access is unsuccessful if it generates a transactional conflict. TM systems with serial, closed-nested transactions report conflicts when two active transactions on different threads are accessing the same object ℓ, and one of those accesses is a write. Thus, only a single active transaction is allowed to contain ℓ in its writeset at one time. For CWSTM semantics, we generalize this definition of conflict in a straightforward manner. At any point in time, let $\texttt{readers}(\ell)$ and $\texttt{writers}(\ell)$ be the sets of *active* transactions that have object ℓ in their readsets or writesets, respectively. Then, we define conflicts as follows:

DEFINITION 1. *At any point in time, a memory operation v generates a **conflict** if*

1. *v reads object ℓ, and $\exists X \in \texttt{writers}(\ell)$ such that $X \notin \texttt{ances}(v)$, or*
2. *v writes to object ℓ, and $\exists X \in \texttt{readers}(\ell)$ such that $X \notin \texttt{ances}(v)$.*

If there is such a transaction X, then we say that v conflicts with X. If v belongs to the transaction X', then we say that X and X' conflict with each other.

If a memory operation v would cause a conflict between $X = \texttt{xparent}[v]$ and another transaction X', then v triggers an abort of either X or X' (or both). Say X is aborted. An abort of a transaction X changes $\texttt{status}[X]$ from `PENDING` to `PENDING_ABORT`, and also changes the status of any `PENDING` (nested) transaction Y in the subtree of X to `PENDING_ABORT`. In general, a `PENDING_ABORT` transaction X that is also ready can only complete by changing its status to `ABORTED`. Conceptually, when a transaction X is `ABORTED`, CWSTM semantics discards X's writeset and readset. Since X is no longer active after this action occurs, the action also conceptually removes X from $\texttt{readers}(\ell)$ and $\texttt{writers}(\ell)$ for all objects ℓ. Note that in CWSTM, if v causes a conflict, and the runtime chooses to abort $X' \neq \texttt{xparent}[v]$, then the conflict is not fully resolved until $\texttt{status}[X']$ has changed to `ABORTED`.

Consider a computation subtree rooted at a transaction X with $\texttt{status}[X] = \texttt{PENDING}$. Since we allow only closed-nested transactions, if every child of X has completed, CWSTM can commit X, i.e., change X's status from `PENDING` to `COMMITTED`, and merge X's readset and writeset into those of $\texttt{xparent}[X]$.

Code Example

We can now describe how the CWSTM semantics constrain the possible outputs of the program in Figure 3. Since parallelism is allowed in transactions, we must consider the scoping of atomicity. In particular, the $x \leftarrow x+1$ in line 4 and the code block in lines 6–9 must appear as though one executes entirely before the other. If the **atomic** statements in lines 4 and 5 were removed, then these two blocks could interleave arbitrarily, even though the entire procedure is protected by an **atomic** statement in line 1. Basically, the atomicity applies only when comparing two blocks of code belonging to different transactions (protected by different atomic statements), not parallel blocks within the same transaction (protected by the same atomic statement).

Conflict as stated in Definition 1 naturally enforces strong atomicity (Blundell et al. 2006). Strong atomicity implies that although line 8 is not atomic, it cannot perform its write between line 9's read and write. In terms of the computation tree in Figure 5, after u_9 performs a `read` of x, it adds x to the readset of X_4; thus, after u_9 occurs but before X_4 commits, if v_8 tries to write to x, it will cause a conflict with X_4. We can, however, have line 8 read x, line 9 read and write x and commit, and then line 8 write x. This interleaving can occur because when u_8 happens, it adds x to the readset of X_3, and u_9 and v_9 can subsequently happen because they are both descendants of X_3 in the computation tree. This behavior means that the increment of 1000 can be "lost" (by being overwritten) but the increment of 100 cannot. Another way of describing strong atomicity is that each memory operation is viewed as a transaction.

Semantic Guarantees

The CWSTM semantics maintains the invariant that a program execution is always conflict-free, according to Definition 1. One can show that when transactions have nested parallel transactions, TM with eager conflict detection according to Definition 1 satisfies the transactional-memory model of *prefix race-freedom* defined in (Agrawal et al. 2006).[1] As shown in (Agrawal et al. 2006), prefix race-freedom and serializability are equivalent if one can safely "ignore" the effects aborted transactions. Note that this equivalence may not hold in TM systems with explicit `retry` constructs that are visible to the programmer.

Definition 1 directly implies the following lemma about a conflict-free execution.

LEMMA 1. *For a conflict-free execution, the following invariants hold for any object ℓ:*

1. *All transactions $X \in \texttt{writers}(\ell)$ fall along a single root-to-leaf path in $\mathcal{C}$. Let $\texttt{lowest}(\texttt{writers}(\ell))$ denote the unique transaction $Y \in \texttt{writers}(\ell)$ such that $\texttt{writers}(\ell) \subseteq \texttt{ances}(Y)$.*
2. *All transactions $X \in \texttt{readers}(\ell)$ are either along the root-to-leaf path induced by the writers or are descendants of the $\texttt{lowest}(\texttt{writers}(\ell))$.*

We use Lemma 1 to argue that one can check for conflicts for a memory operation u by looking at one writer and only a small number of readers. Since all the transactions fall on a single root-to-leaf path, by Lemma 1, Invariant 1, the transaction $\texttt{lowest}(\texttt{writers}(\ell))$ belongs to $\texttt{writers}(\ell)$ and is a descendant of all transactions in $\texttt{writers}(\ell)$. Similarly, let $Q = \texttt{lastReaders}(\ell) \subseteq \texttt{desc}(\texttt{lowest}(\texttt{writers}(\ell)))$ denote the set of readers implied by Invariant 2. If a memory operation u tries to read ℓ, abstractly, there is no conflict exactly if and only if $\texttt{lowest}(\texttt{writers}(\ell))$ is an ancestor of u. Similarly, when u tries to write to ℓ, by Invariant 2, there is no conflict if for all $Z \in \texttt{lastReaders}(\ell)$, Z is an ancestor of u.

4. A NAIVE TM

The CWSTM semantics described in Section 3 suggest a design for a TM system that supports transactions with nested parallelism. In particular, Lemma 1 suggests that for each object ℓ, the TM can maintain an active writing transaction $\texttt{lowest}(\texttt{writers}(\ell))$ and some active reading transactions $\texttt{lastReaders}(\ell)$. This scheme

[1] The proof is a special case of the proof for the operational model described in (Agrawal et al. 2006), without any open-nested transactions.

allows transactions accessing ℓ to test for conflicts against these transactions. This section focuses on a straightforward data structure, called an "access stack," used to maintain these values. We show that an access stack yields a TM with poor worst-case performance, even assuming the rest of the TM system incurs no overhead. The CWSTM design uses a lazy variant of the access stack, described in Section 5, that has much better performance.

The ***access stack*** for an object ℓ is a stack containing the active transactions that have written to ℓ and sets of active transactions that have read from ℓ. The order of transactions on the stack is consistent with the ancestry of transactions in the computation tree. The writing transaction $\texttt{lowest}(\texttt{writers}(\ell))$ is either on top (first item to pop) of the stack, or is the next element on the stack. If the writer is not on the top of the stack, then $\texttt{lastReaders}(\ell)$ is. No two consecutive elements are sets of readers.

The access stack is maintained as follows, locking the relevant stack on all memory access to guarantee atomicity. Consider (a memory operation whose transactional parent is) a transaction X that successfully reads ℓ. If the top of the stack contains a set of readers, then X is added to that set, assuming it is not already there. If the top of the stack is a writer other than X, then $\{X\}$ is added to the top of the stack. Similarly, if X successfully writes ℓ, then X is pushed onto the top of the stack if it not already there.

Whenever a transaction X commits, for each ℓ in X's readset, X is removed from the top of ℓ's access stack and replaced with $\texttt{xparent}[X]$ (in a fashion that ensures there are no duplicated transactions). This action mimics the commit semantics from Section 3: when a transaction X commits, the objects in its readset and writeset are moved to $\texttt{xparent}[X]$'s readset and writeset, respectively. If instead X aborts, then X is popped from each relevant object's access stack. To facilitate rollback on aborts, every access-stack entry corresponding to a write stores the old value before the write.[2]

Maintaining the access stack has poor worst-case performance because the work required on the commit of transaction X is proportional to the size X's readset. If the original program (without transactions) had work T_1, then this implementation might require work $\Omega(dT_1)$, where d is the maximum nesting depth of transactions. In particular, consider the following code snippet:

```
void f(int i) {
  if (i >= 1) {  atomic { x[i]++; f(i-1); }   }
}
```

A call of `f(d)` generates a serial chain of nested transactions, each incrementing a different place in the array `x`. When the transaction at nesting depth j commits, it updates $d-j$ access stacks for a total of $\Theta(d^2)$ access-stack updates. The work of the original program (without transactions), however, is only $\Theta(d)$.

In general, this asymptotic blowup can occur if a TM system with nested transactions must perform work proportional to the size of a transaction's readset or writeset on every commit. For example, a TM system that validates every transaction due to lazy conflict detection for reads exhibits this problem. Similarly, a TM system that copies data on commit due to lazy object updates also has this issue.

5. CWSTM OVERVIEW

This section describes our CWSTM design for a transactional-memory system with nested parallel transactions and eager updates and eager conflict detection. We first describe how CWSTM updates the computation-tree-node statuses on commits and aborts. We then give an overview of the conflict-detection mechanism, deferring details of the XConflict data structure to later sections. The conflict-detection mechanism includes a "lazy access stack," improving on the shortcoming of the access stack from Section 4. Finally, we describe properties of the Cilk-like work-stealing scheduler that CWSTM uses. The XConflict data structure requires such a scheduler for its performance and correctness.

CWSTM explicitly builds the internal nodes of the computation tree (i.e., leaf nodes for memory operations are omitted). Each node maintains a status field which in most cases, explicitly represents the node's status (`PENDING_ABORT`, `PENDING`, `COMMITTED`, or `ABORTED`), and changes in a straightforward fashion. For example, when a transaction X commits, CWSTM atomically changes $\texttt{status}[X]$ from `PENDING` to `COMMITTED`.

Since a transaction may signal an abort of a transaction running on a (possibly different) processor whose descendants have not yet completed, aborting transactions is more involved. When an active transaction X aborts itself (possibly because of a conflict) it simply atomically updates $\texttt{status}[X] \leftarrow \texttt{ABORTED}$. We refer to this type of update as an `xabort`. Alternatively, suppose a processor p_i wishes to abort X even though p_i is not currently executing X. First, p_i atomically changes $\texttt{status}[X]$ from `PENDING` to `PENDING_ABORT`. Then p_i walks X's active subtree, changing $\texttt{status}[Y] \leftarrow \texttt{PENDING_ABORT}$ atomically for each active $Y \in \texttt{desc}(X)$. Notice that p_i never changes any status to `ABORTED`—only the processor running a transaction Y is allowed to perform that update. When X "discovers" that its status has changed to `PENDING_ABORT`, it has no active descendants (otherwise, X cannot be ready, and hence X cannot be executing). Then, X simply performs an `xabort` on itself.

For reasons specific to XConflict, the data structure the CWSTM design uses for conflict detection, during an abort of X, some of X's *`COMMITTED`* descendants Y also have their status field changed to `ABORTED`. Our conflict-detection algorithm uses these updates to more quickly determine that a memory operation does not conflict with Y, since Y has an `ABORTED` ancestor X. Section 9 describes when these updates occur.

In CWSTM, the rollback of objects on abort occur lazily, and thus is decoupled from an `xabort` operation. Once the status of a transaction X changes to `ABORTED`, other transactions that try to access an object modified by X help with cleanup for that object.

Conflict Detection and the Lazy Access Stack

We now discuss conflict detection. The key observation that allows us to avoid explicit maintenance of active readers and writers (or transaction readsets and writesets) is the following alternate conflict definition.

DEFINITION 2. *Consider a (possibly inactive) transaction X that has written to ℓ and a new memory operation v that reads from or writes to ℓ. Then v does* not *conflict with X if and only if*

1. *some transactional ancestor of X has aborted, or*
2. *X's nearest active transactional ancestor is an ancestor of v.*

The case when X has read from ℓ and v writes to ℓ is analogous.

This definition is equivalent to Definition 1 because X's nearest active transactional ancestor logically belongs to $\texttt{writers}(\ell)$ if X doesn't have an aborted ancestor.

Definition 2 suggests a conflict-detection algorithm that does not require maintaining $\texttt{lowest}(\texttt{writers}(\ell))$ and the normal access stack. In particular, let X be the last node that has successfully written to ℓ. Then when u accesses ℓ, test for conflict by finding X's nearest active transactional ancestor Y and determining whether Y is an ancestor of u. Figure 6 gives pseudocode for this test. Note that CWSTM does not actually implement this query as given—instead, it uses an equivalent, but more efficient query, described in Section 6.

[2] This value can either be stored in the stack itself, or in a log per transaction.

```
XCONFLICT-ORACLE(X, u)
    ▷ For any node X and active memory operation u
1   if ∃Z ∈ ances(X) such that status[Z] = ABORTED
2       then return "no conflict: X aborted"

3   Y ← closest active transactional ancestor of X
4   if Y ∈ ances(u)
5       then return "no conflict: X committed to u's ancestor"
6       else pick a transaction B in (ances(Y) − ances(LCA(Y, u)))
7            return "conflict with B"
```

Figure 6. Pseudocode for a conflict-detection query suggested by Definition 2. Many subroutine (e.g., line 3) details are omitted (and in fact do not have efficient implementations). The `LCA` function returns the least common ancestor of two nodes in the computation tree.

To maintain the most recent successful write (and reads), facilitating the necessary conflict queries, CWSTM uses a ***lazy access stack***. The structure of the lazy access stack is somewhat different from the simple access stack given in Section 4. An object ℓ's lazy stack stores (possibly complete) transactions that have written to ℓ and sets of transactions that have read from ℓ, but now these stack entries are ordered chronologically by access. The top of the stack holds the last writer or the last readers. We have the invariant that if a transaction X on the stack has aborted, then all transactions located above X on the stack (later chronologically) also have aborted ancestors, and thus represent deprecated values. The main difference in maintenance is that the lazy access stack is not updated on transactional commit (thus ignoring the merge of a transaction's readset and writeset into its parent's). On memory operations, new transactions are added to the access stack in the same way as described in Section 4.

Figure 7 gives pseudocode for an instrumentation of each memory access, assuming for simplicity that all memory accesses behave as `write` instructions.[3] Incorporating readers into the access stacks is more complicated, but conceptually similar. If a memory access u does not belong to an aborting transaction, then it is allowed to proceed. First, we test for conflict with the last writer in lines 4–5. If the last writer has aborted (or has an aborted ancestor), handled in lines 6–9, then the access stack should be cleaned up by calling CLEANUP. (This auxiliary procedure, given in Figure 8, rolls back the value of the topmost aborted transaction on ℓ's access stack.) Since there is no new conflict, after CLEANUP, the access should be retried. If, on the other hand, there is a conflict between u and an active transaction (lines 10–16), then either `xparent`$[u]$ must abort or the conflicting transaction (B) must abort. Finally, if there are no conflicts, then the access is successful. The access stack is updated as necessary (lines 18–20), and the access is performed.

Note that while u is running the ACCESS method, concurrent transactions (that access ℓ) can continue to commit or abort. The commit or abort of such a transaction can eliminate a conflict with u, but never create a new conflict with u. Thus, concurrent changes may introduce spurious aborts, but do not affect correctness.

The CWSTM Scheduler

XConflict relies on a Cilk-like work-stealing scheduler for efficiency and correctness. The main idea of a work stealing is that when a processor completes its own work, it "steals" work from a different ***victim*** processor. Conceptually, the entire (unexpanded) computation tree is initially "owned" by a single processor. A processor traverses the current subtree that it owns, and only that subtree that it owns. As this processor "discovers" P-nodes it executes one of its nontransactional S-node children, and the other child can subsequently be stolen by a thief processor. Whenever a processor p_i has no work (does not own a subtree), it steals a subtree T rooted at such a nontransactional S-node. Thus, p_i now owns and traverses the subtree T.

When we say a work-stealing scheduler is 'Cilk-like," as required by XConflict, we mean that it has the following two prop-

[3] It is possible to reduce locking on the access stack, but we do not describe that optimization in this paper.

```
ACCESS(u, ℓ)
1   Z ← xparent[u]
2   if status[Z] = PENDING_ABORT return XABORT
    ▷ Otherwise Z is active
3   accessStack(ℓ).LOCK()

    ▷ Set X to be the last writer.
4   X ← accessStack(ℓ).TOP()
5   result ← XCONFLICT-ORACLE(X, u)

6   if result is "no conflict: X aborted"
7       then accessStack(ℓ).UNLOCK()
8            CLEANUP(ℓ)                ▷ Rollback some values
9            return RETRY        ▷ The access should be retried

10  if result indicates a conflict with transaction B
11      then if choose to abort self
12               then accessStack(ℓ).UNLOCK()
13                    return XABORT
14               else accessStack(ℓ).UNLOCK()
15                    signal an abort of B
16                    return RETRY

    ▷ Otherwise, there is no conflict: X is an ancestor of Z
17  if Z ≠ X                          ▷ Z's first access to ℓ
18      then ▷ Log the access
19           LOGVALUE(Z, ℓ)
20           accessStack(ℓ).PUSH(Z)

    ▷ Actually perform the write operation
21  Perform the write
22  accessStack(ℓ).UNLOCK()
23  return SUCCESS
```

Figure 7. Pseudocode instrumenting an access by u to an object ℓ, assuming that all accesses are writes. ACCESS(u,ℓ) returns `XABORT` if Z should abort, `RETRY` if the access should be retried, or `SUCCESS` if the memory operation succeeded.

```
CLEANUP(ℓ)
1   accessStack(ℓ).LOCK()
2   X ← accessStack(ℓ).TOP()
3   if ∃Z ∈ ances(X) such that status[Z] = ABORTED
4       then RESTOREVALUE(X, ℓ) ▷ Restore ℓ from X's log
5            accessStack(ℓ).POP()
6   accessStack(ℓ).UNLOCK()
```

Figure 8. Code for cleaning up an aborted transaction from the top of `accessStack`(ℓ), assuming all accesses are writes. If the last writer has an aborted ancestor, it should be rolled back.

erties. First, a processor executes its computation subtree in a left-to-right fashion. Second, whenever a thief processor steals work from a victim processor, it steals the right subtree from the highest P-node in the victim's subtree that has work available.

6. CWSTM CONFLICT DETECTION

This section describes the high-level ***XConflict*** scheme for conflict detection in CWSTM. As the computation tree dynamically unfolds during an execution, our algorithm dynamically divides the computation tree into "traces," where each trace consists of memory operations (and internal nodes) that execute on the same processor. Our algorithm uses several data structures that organize either traces, or nodes and transactions contained in a single trace. This section describes traces and gives a high-level algorithm for conflict detection.

By dividing the computation tree into traces, we reduce the cost of locking on shared data structures. Updates and queries on a data structure whose elements belong to a single trace are also performed without locks because these updates are performed by a single processor. Data structures whose elements are traces also support queries in constant time without locks. These data structures are, however, shared among all processors, and therefore require a global lock on updates. Since the traces are created only on steals, however, we can bound the number of traces by $O(pT_\infty)$—the number of steals performed by the Cilk-like work-stealing runtime system. Therefore, the number of updates on these data structure can be bounded similarly.

The technique of splitting the computation into traces and having two types of data structures–"global" data structures whose elements are traces and "local" data structures whose elements belong to a single trace–appears in Bender et al.'s (Bender et al. 2004) SP-hybrid algorithm for series-parallel maintenance (later improved in (Fineman 2005)). Our traces differ slightly, and our data structures are a little more complicated, but the analysis technique is similar.

Trace Definition and Properties

XConflict assigns computation-tree nodes to ***traces*** in the essentially the same fashion as the SP-hybrid data structure described in (Bender et al. 2004; Fineman 2005). We briefly describe the structure of traces here. Since our computation tree has a slightly different canonical form from the canonical Cilk parse tree use for SP-hybrid, XConflict simplifies the trace structure slightly by merging some traces together.

Formally, each trace U is a disjoint subset of nodes of the (*a posteriori*) computation tree. We let $\mathcal{Q}$ denote the set of all traces. $\mathcal{Q}$ partitions the nodes of the computation tree $\mathcal{C}$. Initially, the entire computation belongs to a single trace. As the program executes, traces dynamically split into multiple traces whenever steals occur.

A trace itself executes on a single processor in a depth-first manner. Whenever a steal occurs and a processor steals the right subtree of a P-node $P \in U$, the trace U splits into three traces U_0, U_1, and U_2 (i.e., $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{U_0, U_1, U_2\} - \{U\}$). Each of the left and right subtrees of P become traces U_1 and U_2, respectively. The trace U_0 consists of those nodes remaining after P's subtrees are removed from U. Notice that although the processor performing the steal begins work on *only the right subtree* of P, both subtrees become new traces. Figure 9 gives an example of traces resulting from a steal. The left and right children of the *highest uncompleted* P-node P_1 (both these nodes are nontransactional S-nodes in our canonical tree) are the roots of two new traces, U_1 and U_2.

Traces in CWSTM satisfy the following properties.

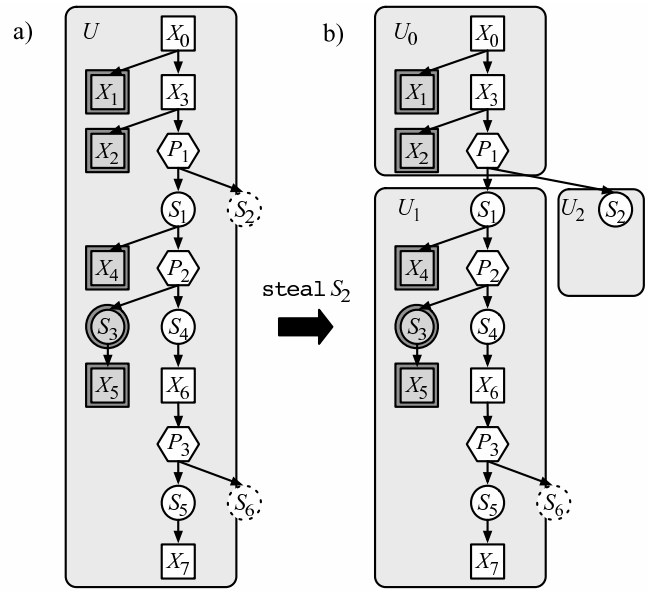

Figure 9. Traces of a computation tree (a) before and (b) after a `steal` action. Before the `steal`, only one processor is executing the subtree, but S_2 and S_6 are ready. After the steal, the subtree rooted at the highest ready S-node (S_2) is executed by the thief. The subtree rooted at S_1, on the other hand, is still owned and executed by the victim processor.

PROPERTY 1. *Every trace $U \in \mathcal{Q}$ has a well-defined **head** nontransactional S-node $S = \texttt{head}[U] \in U$ such that for all nodes $B \in U$, we have $S \in \texttt{ances}(B)$.*

For a trace $U \in \mathcal{Q}$, we use $\texttt{xparent}[U]$ as a shorthand for $\texttt{xparent}[\texttt{head}[U]]$. We similarly define $\texttt{nsParent}[U]$.

PROPERTY 2. *The computation-tree nodes of a trace $U \in \mathcal{Q}$ form a tree rooted at $S = \texttt{head}[U]$.*

PROPERTY 3. *Trace boundaries occur at P-nodes; either both children of the P-node and the node itself belong to different traces, or all three nodes belong to the same trace. All children of an S-node, however, belong to the same trace.*

PROPERTY 4. *Trace boundaries occur at "highest" P-nodes. That is, suppose a P-node P has a stolen child (i.e., P and its children belong to different traces). Consider all ancestor P-nodes P' of P such that P is in the left subtree of P'. Then P' must have a stolen child (i.e., P and ancestor P' belong to different traces).*

The last property follows from the Cilk-like work stealing.

The partition $\mathcal{Q}$ of nodes in the computation tree $\mathcal{C}$ induces a tree of traces $J(\mathcal{C})$ as follows. For any traces $U, U' \in \mathcal{Q}$, there is an edge $(U, U') \in J(\mathcal{C})$ if and only if $\texttt{parent}[\texttt{head}[U']] \in U$.[4] The properties of traces and the fact that traces partition $\mathcal{C}$ into disjoint subtrees together imply that $J(\mathcal{C})$ is also a tree.

We say that a trace U is ***active*** if and only if $\texttt{head}[U]$ is active. The following lemma states that if a descendant trace U' is active, then U' is a descendant of *all* active nodes in U. The proof relies on the fact that traces execute serially in a depth-first (or equivalently, left-to-right) manner.

[4] The function $\texttt{parent}[]$ refers to the parent in the computation tree $\mathcal{C}$, not in the trace tree $J(\mathcal{C})$.

LEMMA 2. *Consider active traces $U, U' \in Q$, with $U \neq U'$. Let $D \in U'$ be an active node, and suppose $D \in \texttt{desc}(\texttt{head}[U])$ (i.e., U' is a descendant trace of U). Then for any active node $B \in U$, we have $B \in \texttt{ances}(D)$.*

PROOF. Since traces execute on a single processor in a depth-first manner, only a single head-to-leaf path of each trace can be active. Thus, if a descendant trace U' is active, it must be the descendant of some node along that path. In particular, we claim that U' is a descendant of the leaf, and hence it is a descendant of *all* active nodes as the lemma states. This claim follows from Property 3, because both children of the *active* P-node on the trace boundary must belong to different traces. □

XConflict Algorithm

Recall that CWSTM instruments memory accesses, testing for conflicts on each memory access by performing queries of XConflict data structures. In particular, XConflict must test whether a recorded access by node B conflicts with the current access by node u. Suppose that B does not have an aborted ancestor. Then recall Definition 2 states that a conflict occurs if only if the nearest uncommitted transactional ancestor of B is *not* an ancestor of u.

A straightforward algorithm (given in Figure 6) for conflict detection finds the nearest uncommitted transactional ancestor of B and determines whether this node is an ancestor of u. Maintaining such a data structure subject to parallel updates is costly (in terms of locking overheads).

XConflict performs a slightly simpler query that takes advantages of traces. XConflict does not explicitly find the nearest uncommitted transactional ancestor of B; it does, however, still determine whether that transaction is an ancestor of u. In particular, let Z be the nearest uncommitted transactional ancestor of B, and let U_Z be the trace that contains Z. Then XConflict finds U_Z (without necessarily finding Z). Testing whether U_Z is an ancestor of u is sufficient to determine whether Z is an ancestor of u. Note that XConflict does not lock on any queries. Many of the subroutines (described in later sections) need only perform simple ABA tests to see if anything changed between the start and end of the query.

The XCONFLICT algorithm is given by pseudocode in Figure 10. lines 1–4 handle the simple base cases. If B and u belong to the same trace, they are executed by a single processor, so there is no conflict. If B is aborted, there is also no conflict.

Suppose B is not aborted and that B and u belong to different traces. XCONFLICT first finds X, the nearest transactional ancestor of A that belongs to an active trace, in line 5. The possible locations of X in the computation tree are shown in Figure 12. Let $U_X = \mathsf{trace}(X)$. Notice that U_X is active, but X may be active or inactive. For cases (a) or (b), we find X with a simple lookup of $\texttt{xparent}[B]$. Case (c) involves first finding U, the highest completed ancestor trace of $\mathsf{trace}(B)$, then performing a simple lookup of $\texttt{xparent}[U]$. Section 9 describes how to find the highest completed ancestor trace.

Line 9 finds Y, the highest active transaction in U_X. If Y exists and is an ancestor of X, as shown in the left of Figure 13, then XCONFLICT is in the case given by lines 11–13. If U_X is an ancestor of u, we conclude that A has committed to an ancestor of u. Figure 13 (a) and (b) show the possible scenarios where U_X is an ancestor of u: either X is an ancestor u, or X has committed to some transaction Z that is an ancestor of u.

Suppose instead that Y is not an ancestor of X (or that Y does not exist), as shown in the left of Figure 14. Then XCONFLICT follows the case given in lines 15–17. Let Z be the transactional parent of U_X. Since X has no active transactional ancestor in U_X, it follows that X has committed to Z. Thus, if $\mathsf{trace}(Z)$ is an ancestor of u,

```
XCONFLICT(B,u)
    ▷ For any computation-tree node B and any
      active memory-operation u

    ▷ Test for simple base cases
 1  if trace(B) = trace(u)
 2     then return "no conflict"
 3  if some ancestor transaction of B is aborted
 4     then return "no conflict: B aborted"

 5  Let X be the nearest transactional ancestor of B
       belonging to an active trace.
 6  if X = null                          ▷ committed at top level
 7     then return "no conflict: B committed to root"
 8  U_X ← trace(X)

 9  Let Y be the highest active transaction in U_X

10  if Y ≠ null and Y is an ancestor of X
11     then if U_X is an ancestor of u
12             then return "no conflict: B committed
                                  to u's ancestor"
13             else  return "conflict with Y"
14     else  Z ← xparent[U_X]
15           if Z = null or trace(Z) is an ancestor of u
16             then return "no conflict: B committed
                                  to u's ancestor"
17             else  return "conflict with Z"
```

Figure 10. Pseudocode for the XConflict algorithm.

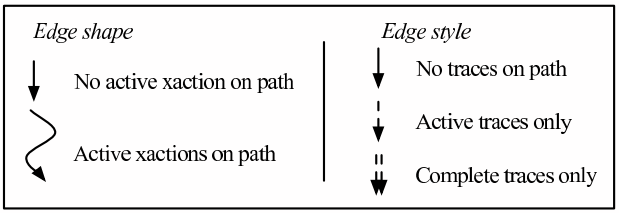

Figure 11. The definition of arrows used to represent paths in Figures 12, 13 and 14.

we conclude that A has committed to an ancestor of u, as shown in Figure 14.

Section 7 describes how to find the trace containing a particular computation-tree node (i.e., computing $\mathsf{trace}(B)$). Section 8 describes how to maintain the highest active transaction of any trace (used in line 9). Section 9 describes how to find the highest completed ancestor trace of a trace (used for line 5), or find an aborted ancestor trace (line 3). Computing the transactional parent of any node in the computation tree ($\texttt{xparent}[B]$) is trivial. Section 10 describes a data structure for performing ancestor queries within a trace (line 10), and a data structure for performing ancestor queries between traces (lines 11 and 15).

The following theorem states that XConflict is correct.

THEOREM 3. *Let B be a node in the computation tree, and let u be a currently executing memory access. Suppose that B does not have an aborted ancestor. Then* XCONFLICT(B,u) *reports a conflict if and only if the nearest (deepest) active transactional ancestor of B is an ancestor of u.*

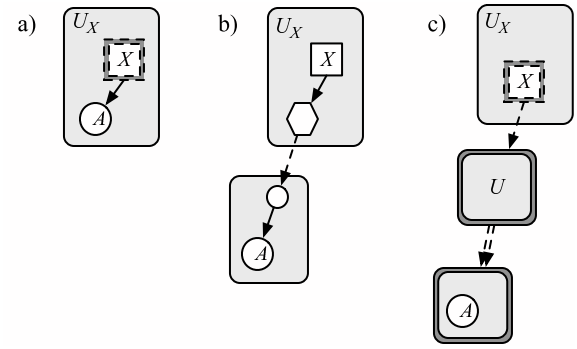

Figure 12. The three possible scenarios in which X is the nearest transactional ancestor of B that belongs to an active trace. Arrows represent paths between nodes (i.e., many nodes are omitted): see Figures 4 and 11 for definitions. In both (a) and (b), B belongs to an active trace. In (a), $\texttt{xparent}[B]$ belongs to the same active trace as B. In (b), $\texttt{xparent}[B]$ belongs to an ancestor trace of $\mathsf{trace}(B)$. In (c), B belongs to a complete trace, U is the highest completed ancestor trace of B, and X is the $\texttt{xparent}[U]$.

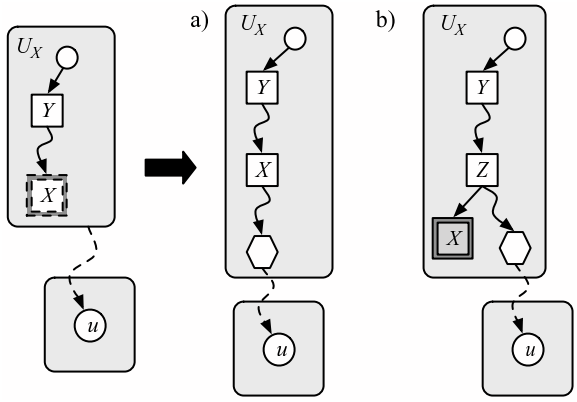

Figure 13. The possible scenarios in which the highest active transaction Y in U_X is an ancestor of X, and U_X is an ancestor of u (i.e., line 11 of Figure 10 returns true). Arrows represent paths between nodes (i.e., many nodes are omitted): see Figures 4 and 11 for definitions. The block arrow shows implication from the left side to either (a) or (b).

PROOF. If B has an aborted ancestor, then XCONFLICT properly returns no conflict.

Let Z be the nearest active transactional ancestor of B. Let U_Z be the trace containing Z; since Z is active, U_Z is active. Lemma 2 states that U_Z is an ancestor of u if and only if Z is an ancestor of u. It remains to show that XConflict finds U_Z.

XConflict first finds X, the nearest transactional ancestor of B belonging to an active trace (line 5). The nearest active ancestor of B must be X or an ancestor of X. Let U_X be the trace containing X, and let Y be the highest active transaction in U_X. If Y is an ancestor of X, then either $Z = X$, or Z is an ancestor of X and a descendant of Y (as shown in Figure 13). Thus, XConflict performs the correct test in lines 11–13.

Suppose instead that Y, the *highest* active transaction in U_X, is not an ancestor of X. Then no active transaction in U_X is an ancestor of X. Let Z be the transactional ancestor of U_X. Since U_X is active, Z must be active. Thus, Z is the nearest uncommitted transactional ancestor of B, and XConflict performs the correct test in lines 15–17.

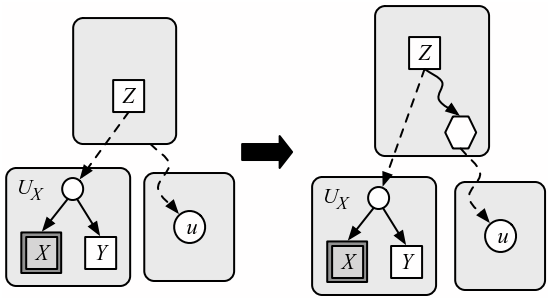

Figure 14. The scenario in which the highest active transaction Y in U_X is not an ancestor of X, and $Z = \texttt{xparent}[U_X]$ is an ancestor of u (i.e., line 15 of Figure 10 returns true). The block arrow shows implication from the left side to the situation on the right.

In the above explanation, we assume that no XConflict data-structural changes occur concurrently with a query. The case of concurrent updates is a bit more complicated and omitted from this proof. The main idea for proving correctness subject to concurrent updates is as follows. Even when trace splits occur, if a conflict exists, XCONFLICT has pointers to traces that exhibit the conflict. Similarly, if XCONFLICT acquires pointers to a transaction (Y or Z) deemed to be active, that transaction was active at the start of the XCONFLICT execution.

□

Note that XCONFLICT may return some spurious conflicts if transactions complete during the course of a query.

7. TRACE MAINTENANCE

This section describes how to maintain trace membership for each node B in the computation tree, subject to queries $\mathsf{trace}(B)$. The queries take $O(1)$ time in the worst case. We give the main idea of the scheme here for completeness, but we omit details as they are similar to the local-tier of SP-hybrid (Bender et al. 2004; Fineman 2005).

To support trace membership queries, XConflict organizes computation-tree nodes belonging to a single trace as follows. Nodes are associated with their nearest nontransactional S-node ancestor. These S-nodes are grouped into sets, called "trace bags." Each bag b has a pointer to a trace, denoted $\texttt{traceField}[b]$, which must be maintained efficiently. A trace may contain many trace bags.

Bags are merged dynamically in a way similar to the SP-bags (Feng and Leiserson 1997) in the local tier of SP-hybrid (Bender et al. 2004; Fineman 2005) using a disjoint-sets data structure (Cormen et al. 2001, Chapter 21). Since traces execute on a single processor, we do not lock the data structure on update (UNION) operations. The difference in our setting is that we use only one kind of bag (instead of two in SP-bags).

When steals occur, a global lock is acquired, and then a trace is split into multiple traces, as in the global tier of SP-hybrid (Bender et al. 2004; Fineman 2005). The difference in our setting is that traces split into three traces (instead of five in SP-hybrid). It turns out that trace splits can be done in $O(1)$ worst-case time by simply moving a constant number of bags. When the trace constant-time split completes (including the split work in Sections 8 and 10), the global lock is released.

To query what trace a node B belongs to, we perform the operations $\texttt{traceField}[\text{FIND-BAG}(\texttt{nsParent}[B])]$. These queries (in particular, FIND-BAG) take $O(1)$ worst-case time as in SP-

hybrid (Bender et al. 2004; Fineman 2005). Merging bags uses an UNION operation and takes $O(1)$ amortized time, but an optimization (Fineman 2005) gives a technique that improves UNIONs to worst-case $O(1)$ time whenever the amortization might adversely increase the program's critical path.

8. HIGHEST ACTIVE TRANSACTION

This section describes how XConflict finds the highest active transaction in a trace, used in line 9 of Figure 10 in $O(1)$ time.

For each nontransactional S-node S, we have a field $nextx[S]$ that stores a pointer to the nearest active descendant transaction of S. Maintaining this field for *all* S-nodes is expensive, so instead we maintain it only for some S-nodes as follows. Let $S \in U$ be an active nontransactional S-node such that either $S = \texttt{head}[U]$, or S is the left child of a P-node and S's nearest S-node ancestor (which is always a grandparent) is a transaction. Then $nextx[S]$ is defined to be the nearest, active descendant transaction of S in U. Otherwise, $nextx[S] = \texttt{null}$.

Finding the highest active transaction simply entails a call to $nextx[\texttt{head}[U]]$, which takes $O(1)$ time. The complication is maintaining the *nextx* values, especially subject to dynamic trace splits.

To maintain *nextx*, we keep a stack of S-nodes in U for which *nextx* is defined. Initially push $\texttt{head}[U]$ onto the stack. For each of the following scenarios, let S be the S-node on the top of the stack. Whenever encountering a transactional S-node X, check $nextx[S]$. If $nextx[S] = \texttt{null}$, then set $nextx[S] \leftarrow X$. Otherwise, do nothing. Whenever completing a transaction X, check $nextx[S]$. If $nextx[S] = X$, then set $nextx[S] \leftarrow \texttt{null}$. Otherwise, do nothing. Whenever encountering a nontransactional S-node S'. If $nextx[S] = \texttt{null}$, do nothing. Otherwise, push S' onto the stack. Whenever completing a nontransactional S-node S', pop S' from the stack if it is on top of the stack.

Finally, XConflict maintains these *nextx* values even subject to trace splits. Consider a split of trace U into three traces U_1, U_2, and U_3, rooted at S, S_1, and S_2, respectively. Since CWSTM steals from the highest P-node in the computation tree, S_1 must be the highest, active, nontransactional S-node descendant of S that is the left child of a P-node. Thus, either S_1 is the second S-node on U's stack, or S_1 is not on U's stack.

If S_1 is on U's stack, then $nextx[S]$ is defined to be an ancestor of S, and we leave it as such. Moreover, since S_1 is on the stack, $nextx[S_1]$ is defined appropriately. Simply split the stack into two just below S to adjust the data structure to the new traces. Suppose instead that S_1 is not on U's stack. Then the $nextx[S]$ may be a descendant of S_1 (or it is undefined). Set $nextx[S_1] \leftarrow nextx[S]$ and $nextx[S] \leftarrow \texttt{null}$. Then split the stack below S, and prepend S_1 at the top of its stack. The necessary stack splitting takes $O(1)$ worst-case time. This splitting occurs while holding the global lock acquired during the steal (as in Section 7).

9. SUPERTRACES

This section describes XConflict's data structure to find the highest completed ancestor trace of a given trace (used as a subprocedure for line 5 in Figure 10, illustrated by U in Figure 12 (c)). To facilitate these queries, XConflict groups traces together into "supertraces." Grouping traces into supertraces also facilitates faster aborts—when aborting a transaction in trace U, we need only abort some of the supertrace children of U, not the entire subtree in $\mathcal{C}$. This section also provides some details on performing the abort.

All update operations on supertraces take place while holding the same global lock acquired during the steal (as in Sections 7, 8, and 10). Note that unlike the data structures in Sections 7, 8, and 10, the updates to supertraces do not occur when steals occur. To prove good performance (in Section 11), we use the fact that the number of supertrace-update operations is asymptotically identical to the number of steals. This amortization is similar to the "global tier" of SP-hybrid (Bender et al. 2004).

At any point during program execution, a completed trace $U \in \mathcal{Q}$ belongs to a ***supertrace*** $K = \mathsf{strace}(U) \subseteq \mathcal{Q}$. In particular, the traces in K form a tree rooted at some ***representative trace*** $rep[K]$, which is an ancestor of all traces in K. Our structure of supertraces is such that either $rep[\mathsf{strace}(U)]$ is the highest completed ancestor trace of U (i.e., as used by line 5 in Figure 10), or U has an aborted ancestor. We prove this claim in Lemma 4 after describing how to maintain supertraces.

Supertraces are implemented using a disjoint-sets data structure (Cormen et al. 2001, Chapter 21). In particular, we use Gabow and Tarjan's data structure that supports MAKE-SET, FIND (implementing $\mathsf{strace}(U)$), and UNION operations, all in $O(1)$ amortized time when unions are restricted to a tree structure (as they are in our case).

When a trace U is created, we create an empty supertrace for U (so $\mathsf{strace}(U) = \emptyset$). When the trace completes (i.e., at a `join` operation), we acquire the global lock. We then add U to U's supertrace (giving $\mathsf{strace}(U) = \{U\}$). Next, we consider all child traces U' of U (in the tree of traces $J(\mathcal{C})$).[5] If $\texttt{head}[U']$ is `ABORTED`, then we skip U'. If $\texttt{head}[U']$ is `COMMITTED`, we merge the two supertraces with $\text{UNION}(\mathsf{strace}(U), \mathsf{strace}(U'))$. Thus, for U' (and all relevant descendants), $rep[\mathsf{strace}(U')] = rep[\mathsf{strace}(U)] = U$. Once these updates complete, the global lock is released. Later, U's supertrace may be merged with its parents, thereby updating $rep[\mathsf{strace}(U)]$.

A naive algorithm to abort a transaction X must walk the entire computation subtree rooted at X, changing all of X's `COMMITTED` descendants to `ABORTED`. Instead, we only walk the subtree rooted at X in U, not $\mathcal{C}$. Whenever hitting a trace boundary (i.e., $B \in U$, $D \in \texttt{children}(B)$, $D \in U' \neq U$), we set that root of the child trace (D) to be aborted and do not continue into its descendants. Thus, we enforce all descendants of B have a supertrace with an aborted representative.

The following lemma (proof omitted) states that either the representative of U's supertrace is the highest completed ancestor trace of U, or U has an aborted ancestor.

LEMMA 4. *For any completed trace $U \in \mathcal{Q}$, let $K = \mathsf{strace}(U)$, and let $U' = rep[K]$. Exactly one of the following cases holds.*

1. *Either $\texttt{head}[U']$ is `ABORTED`, or*
2. *$\texttt{head}[U']$ is `COMMITTED`, $\mathsf{trace}(\texttt{parent}[\texttt{head}[U']])$ is active, and there is no `ABORTED` transaction between $\texttt{head}[U]$ and $\texttt{head}[U']$.*

10. ANCESTOR QUERIES

This section describes how XConflict performs ancestor queries. XConflict performs a "local" ancestor query of two nodes belonging to the same trace (line 10 of Figure 10) and a "global" ancestor query of two different traces (lines 11 and 15 of Figure 10). Both of these queries can be performed in $O(1)$ worst-case time. The global lock is acquired only on updates to the global data structure, which occurs on trace splits (i.e., steals).

Local ancestor queries

CWSTM executes a trace on a single processor, and each trace is executed in depth-first (e.g.., left-to-right) order. We thus view a trace execution as a depth-first execution of a computation (sub)tree (or a depth-first tree walk).

[5] Maintaining a list of all child traces is not difficult. We keep a linked list for each node in the trace tree and add to it whenever a trace splits.

To perform ancestor queries on a depth-first walk of a tree, we associate with each tree node u the ***discovery time*** $d[u]$, indicating when u is first visited (i.e., before visiting any of u's children), and the ***finish time*** $f[u]$, indicating when u is last visited (i.e., when all of u's descendants have finished). (This same labeling appears in depth-first search in (Cormen et al. 2001, Section 22.3).) These timestamps are sufficient to perform ancestor queries in constant time.

In the context of XConflict, we simply need associate a "time" counter with each trace. Whenever a trace splits, this counter's value is copied to the new traces.

Global ancestor queries

Since the computation tree does not execute in a depth-first manner, the same discovery/finish time approach does not work for ancestor queries between traces. Instead, we keep two total orders on the traces dynamically using order-maintenance data structures (Dietz and Sleator 1987; Bender et al. 2002). These two orders give us enough information to query the ancestor-descendant relationship between two nodes in the tree of traces. These total orders are updated while holding the global lock acquired during the steal, as in Sections 7 and 8. Since our global ancestor-query data structure resembles the global series-parallel-maintenance data structure in SP-hybrid (Bender et al. 2004), we omit the details of the data structure. As in SP-hybrid, each query has a worst-case cost of $O(1)$, and trace splits have an amortized cost of $O(1)$.

Note that correctness of the global ancestor queries relies on Property 4—the Cilk-like work-stealing property that a thief processor steals a subtree from the highest available P-node owned by the victim.

11. PERFORMANCE CLAIMS

The section bounds the running time of an CWSTM program in the absence of conflicts. The bound includes the time to check for conflicts *assuming that all accesses are writes* and to maintain the relevant data structures. Checking for conflicts with multiple readers, however, increases the runtime. Additionally, aborts add more work to the computation. Those slowdowns are not included in the analysis.

The proof technique here is similar to the proof of performance of SP-hybrid in (Fineman 2005), and we omit the details of the proof. The key insight in this analysis technique is to amortize the cost of updates of global-lock-protected data structures against the number of steals. One important feature of XConflict's "global" data structures is that they have $O(\#\text{ of traces})$ total update cost. Another is that whenever a steal attempt occurs, the processor being stolen from is making progress on the original computation. (That is, whenever stealable, a processor performs only $O(1)$ additional work for each step of the original computation.) The proof makes the pessimistic assumption that while the global lock is held, only the processor holding the lock makes any progress.

The following theorem states the running time of an CWSTM program under nice conditions. We give bounds for both Cilk's normal randomized work-stealing scheduler, and for a round-robin work-stealing scheduler (as in (Fineman 2005)).

THEOREM 5. *Consider an CWSTM program with T_1 work and critical-path length T_∞ in which all memory accesses are writes. Suppose the program, augmented with XConflict, is executed on p and that no transaction aborts occur.*

1. *When using a randomized work-stealing scheduler, the program runs in $O(T_1/p + p(T_\infty + \lg(1/\varepsilon)))$ time with probability at least $1-\varepsilon$, for any $\varepsilon > 0$,*
2. *When using a round-robin work-stealing scheduler, the program runs in $O(T_1/p + pT_\infty)$ worst-case time.*

One way of viewing these bounds is as the overhead of XConflict algorithm itself. These bounds nearly match those of a Cilk program without XConflict's conflict detection. The only difference is that the T_∞ term is multiplied by a factor of p. In most cases, we expect $pT_\infty \ll T_1/p$, so these bound represents only constant-factor overheads beyond optimal. We would also expect the first bound (using the randomized scheduler) to have better constants hidden in the big-O.

These XConflict bounds translate to bounds on completion time of an CWSTM program under optimistic conditions. For illustration, consider a program where all concurrent paths access disjoint sets of memory. The overhead of maintaining the XConflict data structures is $O(T_1/p + pT_\infty)$. Each memory access queries the XConflict data structure at most once. Since each query requires only $O(1)$ time, the entire program runs in $O(T_1/p + pT_\infty)$ time.

The CWSTM design we describe does not provide any reasonable performance guarantees when we allow multiple readers. There are two reasons for this problem. First, concurrent reads to an object may contend on the access stack to that object. Second, even in the case where concurrent read operations never wait to acquire an access stack lock, it appears that write operation may need to check for conflicts against potentially many readers in a reader list (some of which may have already committed). Therefore, a write operation is no longer a constant time operation, and it seems the work of the computation might increase proportional to the number of parallel readers to an object. It is part of future work to improve the CWSTM design and analysis in the presence of multiple readers.

12. CONCLUSIONS

The CWSTM model presented in Section 5 describes one approach for implementing a software transactional memory system that supports transactions with nested fork-join parallelism. CWSTM design was guided by a few major goals.

- Supporting nested transactions of unbounded depth.
- Small overhead when there are no aborts.
- Avoid asymptotically increasing the work or the critical path of the computation too much.

We believe that we have achieved these goals to some extent, since the CWSTM guarantees provably good completion time in the case when there are no aborts, and there are no concurrent readers (or all accesses are treated as writes).

We believe that supporting unbounded nesting depth in transactions is important for composability of programs. If a function is called from inside a transaction, the caller should not have to worry about how many transactions are nested inside the function call. However, it may be true that the common case is transactions of small depth. In this case, a simpler design like the one described in Section 4 might be sufficient in the common case, at the expense of a slowdown in the case when the nesting depth is large. It is difficult, however, to conclude what the common case is, since there are currently few examples of programs with nested parallel transactions. An important part of future research would be to write series-parallel programs with nested transactions to understand what the common case is, and what one should optimize for.

CWSTM is a TM system design and has not been implemented yet. As described in this paper, each memory access may potentially require multiple data structure queries. CWSTM also may have a large memory footprint. Due to lazy cleanup on aborts, and fast commits, the access stack for an object may grow and require space proportional to the number of accesses to that object. Also, access stacks may contain pointers to transaction logs that persist long after the transactions are committed or aborted. Thus, a computation's memory footprint can become quite large. In

practice, implementing a separate, concurrent thread for "garbage-collection" of metadata may help. As part of future work, we would like to implement the system in the Cilk runtime system to evaluate its practical performance and explore ways to optimize the implementation.

It would be interesting to see if CWSTM-like mechanisms are useful for high-performance languages like Fortress (Allen et al. 2007) and X10 (Ebcioglu et al. 2005). Both these languages support transactions and fork-join parallelism. The language specification for Fortress also permits nested parallel transactions. These are richer languages than Cilk, however, and may require more complicated mechanisms to support nested parallel transactions.

References

Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 26–37, Jun 2006.

Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Memory models for open-nested transactions. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC)*, October 2006. In conjunction with *International Conference on Architectutal Support for Programming Languages and Operating Systems*.

Eric Allen, David Chase, Joe Hllett, Victor Luchango, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification, version 1.0 β. Technical report, Sun Microsystems, Inc., March 2007.

C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. *IEEE Micro*, 26 (1):59–69, January 2006. URL `http://supertech.csail.mit.edu/papers/xaction.pdf`.

M. A. Bender, R. Cole, E. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the European Symposium on Algorithms (ESA)*, pages 152–164, 2002.

Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 133–144, Barcelona, Spain, June 2004.

Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *ICFP*, pages 213–225, 1996.

Robert D. Blumofe and Dionisios Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. Technical Report, University of Texas at Austin, 1999.

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 25 1996.

Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), Nov 2006.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, second edition, 2001.

Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006.

P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the Symposium on Theory of Computing*, pages 365–372, New York City, May 1987.

Kemal Ebcioglu, Vijay Saraswat, and Vivek Sarkar. X10: an experimental language for high productivity programming of scalable systems. In *Proceedings of Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, 2005. In conjunction with *Symposium on High Performance Computer Architecture (HPCA)*.

Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, Newport, Rhode Island, June22–25 1997.

Jeremy T. Fineman. Provably good race detection that runs in parallel. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 2005.

Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, page 102, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2143-6.

Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 289–300, 1993. doi: http://doi.acm.org/10.1145/165123.165164. URL `http://www.cs.brown.edu/people/mph/isca2.ps`.

Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, 2003. ISBN 1-58113-708-7. doi: http://doi.acm.org/10.1145/872035.872048.

Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, March 2006.

Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of nonblocking software transactional memory. In *Proceedings of the Workshop of Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, June 2006.

K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, Feb 2006.

Supercomputing. *Cilk 5.4.2.3 Reference Manual*. Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science, April 2006. URL `http://supertech.csail.mit.edu/cilk/manual-5.4.2.3.pdf`.