Nested Parallelism in Transactional Memory

Kunal Agrawal Jeremy T. Fineman Jim Sukha
MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139, USA

Abstract

This paper describes XCilk, a runtime-system design for software transactional memory in a Cilk-like parallel programming language, which uses a work-stealing scheduler. XCilk supports transactions that themselves can contain nested parallelism and nested transactions, both of unbounded nesting depth. Thus, XCilk allows users to call a library function within a transaction, even if that function itself exploits concurrency and uses transactions. XCilk provides transactional memory with strong atomicity, eager updates, eager conflict detection and lazy cleanup on aborts.

XCilk uses a new algorithm and data structure, called XConflict, to facilitate conflict detection between transactions. Using XConflict, XCilk guarantees provably good performance for closed-nested transactions of arbitrary depth in the special case when all accesses are writes and there is no memory contention. More precisely, XCilk executes a program with work T_1 and critical-path length T_∞ in $O(T_1/p + pT_\infty)$ time on p processors if all memory accesses are writes and all concurrent paths of the XCilk program access disjoint sets of memory locations. 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.

1. INTRODUCTION

Transactional memory (TM) [17] 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. [4, 15, 20]), in software (e.g., [1, 16, 19]), or hybrids of hardware and software (e.g., [10, 18]).

Most work on transactional memory focuses exclusively on the case when each transaction executes serially. This assumption restricts the composability of parallel programs that use transactions. A parallel program cannot call a parallel library function `f` from a transaction without serializing `f`. In addition, programs written using these TM systems are constrained by the requirement that the abstract parallelism of the program follows the structure of threads created by the program.

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

```
1  cilk void foo() { atomic { b++;... } }
2  cilk void bar() { atomic { ...,b++ } }
3  cilk void proc() {
4    atomic {       // transaction X
5     b++;
6     spawn foo();  // transaction Y1
7     spawn bar();  // transaction Y2
8     b++;
9     sync;
10   }
11 }
```

Figure 1. A transaction `X` with nested parallelism. As `Y2` executes, it should conflict with `Y1` but not with `X`.

For languages such as Cilk [7, 21], however, a program's abstract parallelism is decoupled from the number of available threads. In Cilk, the programmer specifies parallelism abstractly using `spawn` statements. The Cilk runtime system uses a work-stealing to dynamically schedule the program on p processors, where p is specified when the execution of the program begins. The Cilk scheduler guarantees that a program with "work" T_1 and "critical-path" length T_∞ completes in $O(T_1/p + T_\infty)$ time.

A natural extension of TM and Cilk is to allow Cilk functions to use transactions. This extension leads to the idea of nested parallel transactions inside transactions, as shown in Figure 1. Here, a transaction `X` spawns two other methods that also contain transactions. According to Cilk semantics, the calls to `foo` and `bar` can execute in parallel. But they must execute in isolation with respect to each other. Thus, the notion of conflict detection generalizes to nested subtransactions; the calls to `foo` and `bar` conflict with each other since they write to the same memory location `b`.

Incorporating transactions into a language such as Cilk poses several challenges for conflict detection. In a Cilk program, the state information of a transaction (e.g., its readset

and writeset), can no longer be directly associated with a particular thread; thus, we cannot detect a conflict between two memory operations by simply checking whether different worker threads performed those operations. Consider a Cilk execution of `proc` from Figure 1, and suppose the transaction `X` starts executing on worker 0, which first increments `b`. When encountering the first `spawn`, worker 0 pushes the continuation of `proc` (starting immediately after line 6) onto its deque then continues executing `Y1`. While worker 0 is executing `Y1`, worker 1 may steal from worker 0 and continue executing `proc`. It immediately pushes the continuation of `proc` (starting after line 7) on the deque and executes `Y2`. When `Y2` increments `b`, there is a conflict with `Y1`, which is running in parallel with `Y2` and modifies the same variable `b`. Since `Y2` is nested inside `X`, however, `Y2` does not conflict with `X`, even though `X` also modifies `b`. Thus, we cannot directly check for conflicts by comparing worker ids. Instead, checking for conflicts requires us to be able to efficiently determine online whether one transaction is an ancestor of the other. Again, worker 3 might now steal the continuation of `proc` and execute line 8. The increment of `b` on line 8 again belongs to `X`, but it conflicts with `Y1` and `Y2` if the TM system supports strong atomicity [8].[1]

This paper describes XCilk, a runtime system design for a software TM system that supports nested parallel transactions with strong atomicity [8]. XCilk supports transactions with nested parallelism in a Cilk-like language that uses a work-stealing scheduler. XCilk performs ***eager conflict detection*** (i.e., detects conflicts as soon as they occur), thereby minimizing wasted work. XCilk uses a novel algorithm, called ***XConflict***, for fast conflict detection. In addition, XCilk performs ***eager updates*** and makes the (hopefully) common case of commits run fast at the expense of slow aborts. XCilk also supports lazy cleanup on aborts. Moreover, when a transaction X aborts, other transactions can help roll back the memory updates performed by X.

Our design for XCilk also provides a provable performance guarantee in the special case of an XCilk program that experiences no contention on any memory accesses assuming all accesses are writes, and for which no transaction aborts. Suppose the program has T_1 work and critical-path length T_∞. Then XCilk executes the program in time $O(T_1/p + pT_\infty)$ when run on p processors. Although this result requires extremely optimistic assumptions, a straightforward scheme for supporting closed-nested transactions with parallelism may potentially increase the work of the computation by a factor proportional to maximum transaction nesting depth. In the worst case, for XCilk programs, the maximum transaction nesting depth can be $\Omega(T_1)$.

The remainder of this paper is organized as follows. XCilk uses a computation tree to model a computation with nested parallel transactions; Section 2 describes the computation tree and how the XCilk runtime system maintains this computation tree online. Section 3 describes the high-level transactional memory implementation by XCilk and its use of the XConflict conflict-detection algorithm. Section 4 gives an overview of the XConflict algorithm. Sections 5–8 provide details on data structures used by XConflict. Finally, Section 9 claims that XConflict, and hence XCilk, is efficient for programs that experience no conflicts or contention.

[1] When transactions have nested parallelism, we adopt the definition of strong atomicity which treats every individual memory operation as its own transaction which always commits.

2. THE XCilk COMPUTATION TREE

The XCilk runtime system supports computations with transactions which contain nested parallelism, and schedules these computations using work-stealing. Such computations can be represented using a ***computation tree*** [2]. This section first describes the structure of this computation tree. Then we briefly describe how the XCilk system maintains this computation tree online, and how XCilk performs work-stealing in the context of this computation tree framework.

Structure of the Computation Tree

The transactional-computation framework described in [2] represents the execution of a program as a computation tree $\mathcal{C}$. Structurally, a ***computation tree*** $\mathcal{C}$ is an ordered tree with two types of nodes: ***memory-operation nodes*** $\texttt{memOps}(\mathcal{C})$ at the leaves, and ***control nodes*** $\texttt{spNodes}(\mathcal{C})$ as internal nodes. The root of the tree is represented by $\texttt{root}(\mathcal{C})$. Each leaf node $u \in \texttt{memOps}(\mathcal{C})$ represents a single read or write of a memory location $\ell \in \mathcal{M}$. The internal nodes $\texttt{spNodes}(\mathcal{C})$ of $\mathcal{C}$ represent the parallel control structure of the computation, as well as the nested structure of transactions. In the manner of [13], each internal node $A \in \texttt{spNodes}(\mathcal{C})$ is labeled as either an S-node or P-node to capture fork/join parallelism. All the children of an S-node are executed in series from left to right, while the children of a P-node can be executed in parallel.

We define several structural notations on the computation tree $\mathcal{C}$. If A is an internal node of the tree, then $\texttt{children}(A)$ is the ordered set of A's children, $\texttt{ances}(A)$ is the set of A's ancestors and $\texttt{desc}(A)$ is A's descendants. In this paper, whenever we refer to the set of ancestors or descendants of a node A, we include A in this set.

Transactions are specified in a computation tree $\mathcal{C}$ by marking a subset of S-nodes as transactions. A computation-tree node A is a descendant of a particular transaction node X if and only if the execution of A is contained in X's transaction. A transaction Y is said to be ***nested*** inside a transaction X if $X \in \texttt{ances}(Y)$. We restrict our attention to computations with only closed-nested transactions.

Building the Computation Tree

XCilk abstractly builds the computation tree dynamically as the computation unfolds. The computation tree built by an XCilk program has a ***canonical*** form in that each P-node

has exactly two children, and both these children are nontransactional S-nodes. Figure 3 illustrates a canonical XCilk computation tree. We shall not describe the construction of the tree, since it is relatively straightforward.

As the XCilk runtime system executes a computation, it maintains the following fields for all internal nodes A that it has encountered. Each internal node also knows whether it is an S-node, a P-node, or a transaction.

- `parent`$[A]$: A's parent in the computation tree.
- `xparent`$[A]$: The deepest transactional ancestor of A.
- `nsParent`$[A]$: The deepest nontransactional S-node ancestor of A.[2]
- If A is a P-node, then it maintains pointers to both its children. If it is an S-node, it maintains a linked list of all the children it has seen so far. In addition if A is an S-node, it remembers its current active child, `activeChild`$[A]$.
- `status`$[A]$: The status is either `PENDING`, `PENDING_ABORT`, `COMMITTED` or `ABORTED`.

A node whose status is `PENDING` or `PENDING_ABORT` is considered ***active***. A node is considered ***complete*** if its status is `COMMITTED` or `ABORTED`. The XCilk runtime guarantees that a node A can become complete only after all the descendants of A have completed.

Work-Stealing

An XCilk computation is scheduled using work-stealing. Consider a computation that executes on p *worker threads*, numbered $0, 1, \ldots p-1$, with worker 0 corresponding to the program's main thread. Every worker i maintains a field `running`(i), which stores the a pointer to the S-node S that i is currently *running*. Every worker i also maintains a deque of ***waiting*** S-nodes, denoted as the set `waiting`(i), representing work that can stolen. Note that P-nodes are never considered as running or waiting.

Each worker starts by working on an S-node and executes this S-node's subtree in a depth-first manner. When a worker encounters a P-node, it starts working on the left child of the P-node, and adds the right child to the bottom of the worker's ready deque. When it needs more work, it removes the S-node from the *bottom* of its deque and starts executing that S-node's subtree. If it runs out of work, it randomly picks some other worker and tries to steal the S-node from the *top* of this other worker's deque.

Initially, the root of the tree is on worker 0's deque, and all the other workers have empty deques. Worker 0 starts executing the root, and the other workers try to steal work.

Figure 3 illustrates a canonical XCilk computation tree.

3. XCilk TRANSACTIONAL MEMORY

The TM provided by XCilk has three important features:

[2] For convenience, we treat `root`(C) as both a transactional and nontransactional S-node.

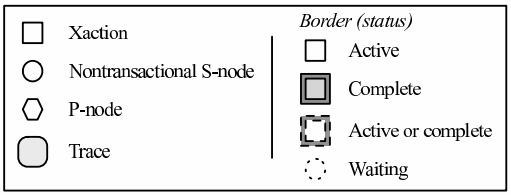

Figure 2. A legend for computation-tree figures.

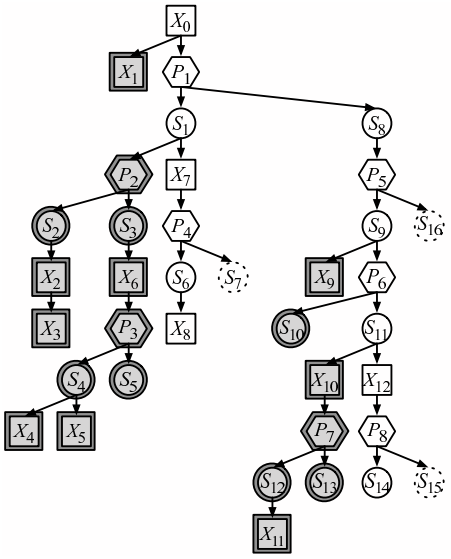

Figure 3. A sample computation tree C generated by XCilk.

- ***Eager Updates***: When transaction X writes to a memory location ℓ, ℓ is immediately updated, and the old value of the ℓ is added to X's log. This update enables fast commits and slow aborts.
- ***Eager Conflict Detection***: When a memory operation u generates a conflict, the conflict is immediately detected. Therefore, XCilk minimizes wasted work.
- ***Lazy Cleanup on Aborts***: Since XCilk performs eager updates, when a transaction X aborts, the memory locations changed by X have to be rolled back to their previous state. This rollback is not instantaneous. In XCilk, other transactions that access these memory locations can help in this cleanup.

This section describes how the XCilk handles aborts, commits, reads and writes to provide these three features.

Committing and Aborting a Transaction

The status field `status`$[X]$ for the transaction X is the location in the XCilk that indicates whether the transaction has successfully committed or aborted. To commit a transaction X, the worker running X performs an `xend` operation which simply changes `status`$[X]$ from `PENDING` to `COMMITTED`. This change represents the commit point of the transaction,

and therefore must happen atomically. Note that all of X's descendants must complete before X can attempt a commit.

Since a worker may signal an abort of a transaction running on a (possibly different) worker whose descendants have not yet completed, the XCilk protocol for a transaction abort is more involved than for a commit. Suppose worker i wishes to abort an active transaction X running on a different worker. Then worker i performs a `sigabort` operation on X. The `sigabort` is implemented as follows. First the worker i changes $\texttt{status}[X]$ from `PENDING` to `PENDING_ABORT`. Again, this change must happen atomically, since it might interfere with the commit of X. Worker i then walks X's active subtree, changing $\texttt{status}[Y]$ to `PENDING_ABORT`, for all active $Y \in \texttt{desc}(X)$.

For reasons specific to the XConflict data structure XCilk uses for conflict detection, `COMMITTED` transactions $Y \in \texttt{desc}(X)$ must be marked as `ABORTED`, since they are closed nested inside X. (Section 7 describes how to optimize this operation a little, avoiding a walk of the entire completed subtree of X.) XConflict guarantees that other transactions can no longer detect conflicts with X anytime after XCilk changes the status of a transaction X to `ABORTED`.

When the worker j executing X "discovers" that X's status has changed to `PENDING_ABORT`, or when X aborts itself, all of X's descendants have completed. Thus, j performs an `xabort` operation that atomically changes $\texttt{status}[X]$ from `PENDING_ABORT` to `ABORTED`.

In XCilk, the rollback of memory locations on abort occur lazily, and thus is decoupled from an `xabort` operation. Once the status of a transaction X changes to `ABORTED`, other transactions which try to access the same memory location modified by X may help with cleanup for that location.

Handling Reads and Writes

XCilk performs eager conflict detection, meaning that it checks for conflicts and successfully completes a `read` or `write` instruction only if the operation does not cause a conflict. Conceptually, in XCilk, a successful `read` instruction from a memory location ℓ issued by a transaction X adds the location ℓ to X's ***readset***. Similarly, a successful `write` instruction adds ℓ to X's ***writeset***. At any point in time, let $\texttt{readers}(\ell)$ and $\texttt{writers}(\ell)$ be the sets of *active* transactions X that have memory location ℓ in its readset or writeset, respectively.[3] Then, using the computation tree, we define conflicts as follows:

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

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

The XCilk runtime maintains the invariant that a program execution is always conflict-free, according to Definition 1. XCilk supports closed-nested transactions; when a transaction X commits, it conceptually merges its readset and writeset into that of its transactional parent, $\texttt{xparent}[X]$, and when X aborts, it discards its readset and writeset. 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 [2].[4] As shown in [2], prefix race-freedom and serializability are equivalent if one can safely "ignore" the effects aborted transactions.[5]

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 memory location 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 $\texttt{writers}(\ell) \subset \texttt{ances}(Y)$.*
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 $Q = \texttt{lastReaders}(\ell)$ denote the set of readers $Q \subseteq \texttt{desc}(\texttt{lowest}(\texttt{writers}(\ell)))$ 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.[6]

TM Using Access Stacks

Using Lemma 1, one simple way TM can perform conflict detection is to explicitly store all readers and writers of a location ℓ on an access stack, denoted by $\texttt{accessStack}(\ell)$. Conceptually, every element on the stack is either a single

[3] We assume the writeset is always a subset of the readset.

[4] The proof is a special case of the proof for the operational model described in [2], without any open-nested transactions.

[5] Note that this equivalence may not hold in TM systems with explicit `retry` constructs that are visible to the programmer.

[6] Note that being conflict free implies strong atomicity (generalized to transactions with nested parallelism). Suppose a transaction X_1 nested inside its parent X_2 has written to a memory location ℓ. If the parent X_2 tries to read or write to ℓ directly, in parallel with X_1, then this access generates a conflict.

transaction $X \in \texttt{writers}(\ell)$, or a set Q of transactions, with $Q \subseteq \texttt{readers}(\ell)$. Every $\texttt{accessStack}(\ell)$ maintains the invariants that for any transaction Y on the stack (or $Y \in Q$ where Q is a list of readers on the stack), every transaction below Y in the stack is an ancestor of Y, and no two adjacent entries on the stack are both lists of readers. Also, $\texttt{accessStack}(\ell)$ always satisfies the property that either the top entry of the stack is $\texttt{lowest}(\texttt{writers}(\ell))$, or the top is $\texttt{lastReaders}(\ell)$ and the second-from-top entry of the stack is $\texttt{lowest}(\texttt{writers}(\ell))$.

To handle aborts, every access-stack entry for ℓ corresponding to a write also has a pointer to the old value of ℓ before the write. Assuming that every `read` or `write` instruction to a location ℓ appears to happen atomically (e.g., because it holds a lock on $\texttt{accessStack}(\ell)$), one can use access stacks to implement TM.

XCilk does not use this simple TM directly, however, because it does not allow for fast commits and lazy cleanup on aborts. Suppose $Y = \texttt{lowest}(\texttt{writers}(\ell))$ is at the top of the stack. Then, when Y commits, $\texttt{xparent}[Y]$ becomes $\texttt{lowest}(\texttt{writers}(\ell))$. For the top of the stack to always be either $\texttt{lowest}(\texttt{writers}(\ell))$ or $\texttt{lastReaders}(\ell)$, we must update the top of all stacks for all locations accessed by Y. Similarly, on an abort of Y, the TM system would have to clean up all the memory locations accessed by Y *before* any other transaction can access these memory locations.

Lazy Access Stacks

In XCilk, therefore, we use a *lazy access stack*, in which the top of $\texttt{accessStack}(\ell)$ does not change immediately on the commit or abort of a transaction. Instead, we use data structures to detect conflicts in a more sophisticated manner. At any point in time, if the program execution has been conflict-free up until that point, then a new read operation v does not generate a conflict with a transaction X that writes to ℓ if and only if

1. some transactional ancestor of X is `ABORTED`, or
2. the X's closest active transactional ancestor is also an ancestor of v.

XCilk performs conflict detection by using the XConflict data structure to answer this query.[7]

Similarly, a write operation w does not generate a conflict if and only if one of the above conditions holds true for all transactions Y satisfying $\texttt{status}[Y] \neq \texttt{ABORTED}$ in reader lists which appear above X in $\texttt{accessStack}(\ell)$.

We present code describing how to manipulate lazy access stacks in Figure 4, assuming for simplicity that all memory accesses behave as `write` instructions.[8] Incorporating readers into the access stacks is more complicated, but conceptually similar. Note that for a conflict due to "nontransactional" code running in parallel with a transaction, line 14 in Figure 4 must choose to abort the transaction.

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

    ▷ Set X to be the top writer on the stack.
 4  X ← accessStack(ℓ).TOP()
 5  if (X = Z)
 6     then goto line 23

 7  result ← XCONFLICT-ORACLE(X, u)
 8  if result is no conflict due to abort
 9     then accessStack(ℓ).UNLOCK()
10          CLEANUP(ℓ)
11          return RETRY

12  if result is no conflict due to ancestor
13     then goto line 20
    ▷ Otherwise, we have a conflict with transaction result.B
14  if choose to abort self
15     then accessStack(ℓ).UNLOCK()
16          return XABORT
17     else accessStack(ℓ).UNLOCK()
18          sigabort(result.B)
19          return RETRY

    ▷ Log the access
20  ADDTOWRITESET(Z, ℓ)
21  LOGVALUE(Z, ℓ)
22  accessStack(ℓ).PUSH(Z)

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

Figure 4. Code for a transaction Z which performs a memory access to location ℓ, assuming all accesses are treated as writes. $\text{ACCESS}(u, \ell)$ returns `XABORT` if Z should abort, `RETRY` if the access should be retried, or `SUCCESS` if the memory operation succeeded.

[7] Recall that we consider X to be an ancestor of itself. Thus, Case 1 includes the case when X itself is `ABORTED`. Also, since the root of the computation tree is always an active transaction, Case 2 includes the case where the top-level transaction containing X has committed.

[8] The access code in Figure 4 assumes XCilk locks the access stack before every access to a memory location. It is possible to optimize this operation such that a transaction X only needs to lock the stack on the first access to the location. We create an *accessor* field for the stack which is changed using a CAS operation. A transaction X that is the accessor of ℓ conceptually owns the stack lock on ℓ, and can only release if it commits or aborts, or if one its nested transactions steals the lock from X. This scheme can be extended to support strong atomicity by setting the accessor fields to be nontransactional S-nodes as well.

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

Figure 5. Code for cleaning up an aborted transaction from the top of accessStack(ℓ), assuming all accesses are writes.

```
XCONFLICT-ORACLE(X, u)
   ▷ For any X ∈ nodes(C) and any u ∈ memOps(C)
     with parent[u] = running(i) on some worker i.
1  if ∃Y ∈ ances(X) such that status[Y] = ABORTED
2     then return no conflict due to abort
3     else Y′ ← closest active transactional ancestor of X
4          if Y′ ∈ ances(u)
5             then return no conflict due to ancestor
6             else BSet ← (ances(Y′) − ances(LCA(Y′, u))).
7                  return any transaction B in BSet
```

Figure 6. Query answered by XConflict data structure. In line 6, if Y' is not an ancestor of Z, then XConflict returns any transaction B on the path between Y' and LCA(Y', Z); since B and Z are both active, they must run in parallel.

The code for a memory access calls a cleanup subroutine, shown in Figure 5. The code also assumes the existence of a function that can answer the query described in Figure 6. In Section 4, we describe the details of XConflict, the data structure for answering this query.

Note that while the ACCESS method is running, transactions running on different workers from u 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.

4. XCilk CONFLICT DETECTION

This section describes the high-level ***XConflict*** scheme for conflict detection in XCilk. 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 worker. 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. In particular, all our data structures allow queries without locks. Updates on a data structure whose elements belong to a single trace are also performed without locks because these updates are performed by a single worker. We do, however, globally lock on updates to data structures whose elements are traces. Since these 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 runtime system.

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 [6] SP-hybrid algorithm for series-parallel maintenance (later improved in [14]). 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 [6, 14]. 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 $U \subseteq$ nodes$(\mathcal{C})$ 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 worker in a depth-first manner. Whenever a steal occurs, a trace U splits into three traces U_0, U_1, and U_2 (i.e., $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{U_0, U_1, U_2\} - \{U\}$), and a worker steals the right subtree of a P-node $P \in U$. 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 worker performing the steal begins work on *only the right subtree* of P, both subtrees become new traces. Figure 7 gives an example of traces resulting from a steal. The left and right subtrees of the *highest uncompleted* P-node P_1 are the roots of two new traces, U_1 and U_2.

Traces in XCilk satisfy the following properties.

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

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

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

PROPERTY 3. *Trace boundaries occur at P-nodes; either both children of the P-node and the node itself belong to*

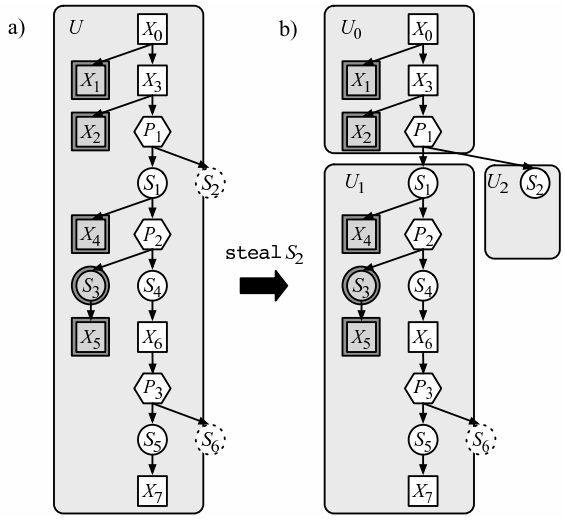

Figure 7. Traces of a computation tree $\mathcal{C}$ (a) before and (b) after a worker `steal` action. Before the `steal`, only one worker 0 is executing ($\texttt{running}(0) = X_7$), with $\texttt{waiting}(0) = \{S_2, S_6\}$. Since S_2 is at the top of 0's deque, when worker 1 performs a successful steal, we get $\texttt{running}(1) = S_2$.

different traces, or all three nodes belong to the same trace. All children of an S-node, however, belong to the same trace.

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 $\texttt{parent}[\texttt{head}[U']] \in U$.[9] 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 if a descendant trace U' is active, than U' is a descendant of *all* active nodes in U. The proof (omitted due to space constraints) relies on the fact that traces execute serially in a depth-first manner.

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

XConflict Algorithm

XCilk instruments memory accesses, testing for conflicts on each memory access by making performing queries of XConflict data structures. In particular, XConflict must test whether a recorded access by node A conflicts with the current access by node u. Suppose that A does not have an aborted ancestor. Then recall that in Figure 6, a conflict occurs if only if the nearest uncommitted transactional ancestor of A is *not* an ancestor of u.

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

```
XCONFLICT(A, u)
    ▷ For any A ∈ nodes(C) and any u ∈ memOps(C)
      with parent[u] = running(i) on some worker i.

    ▷ Test for simple base cases
 1  if trace(A) = trace(u)
 2     then return no conflict
 3  if some ancestor transaction of A is aborted
 4     then return no conflict due to abort

 5  Let X be the nearest transactional ancestor of A
       belonging to an active trace.
 6  if X = null
 7     then return no conflict  ▷ committed at top level
 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 due to 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 due to ancestor
17             else return conflict with Z
```

Figure 8. Pseudocode for the XConflict algorithm.

A straightforward algorithm for conflict detection finds the nearest uncommitted transactional ancestor of A and determines whether this 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 A; it does, however, still determine whether that transaction is an ancestor of u. In particular, let Z be the nearest uncommitted transactional ancestor of A, 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.

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 8. lines 1–4 handle the simple base cases. If A and u belong to the same trace, they are executed by a single worker, so there is no conflict. If A is aborted, there is also no conflict.

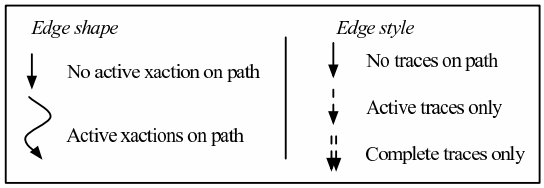

Figure 9. The definition of arrows used to represent paths in Figures 10, 11 and 12.

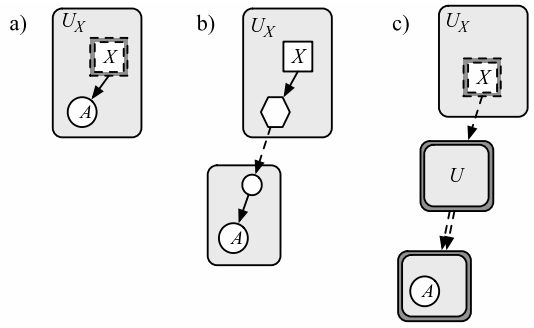

Figure 10. The three possible scenarios in which X is the nearest transactional ancestor of A that belongs to an active trace. Arrows represent paths between nodes (i.e., many nodes are omitted): see Figures 2 and 9 for definitions. In both (a) and (b), A belongs to an active trace. In (a), $\texttt{xparent}[A]$ belongs to the same active trace as A. In (b), $\texttt{xparent}[A]$ belongs to an ancestor trace of $\mathsf{trace}(A)$. In (c), A belongs to a complete trace, U is the highest completed ancestor trace of A, and X is the $\texttt{xparent}[U]$.

Suppose A is not aborted, and A 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 10. 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 $\texttt{xparent}[A]$. Case (c) involves first finding U, the highest completed ancestor trace of $\mathsf{trace}(A)$, then performing a simple looking of $\texttt{xparent}[U]$. Section 7 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 11, 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 11 (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 12. 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

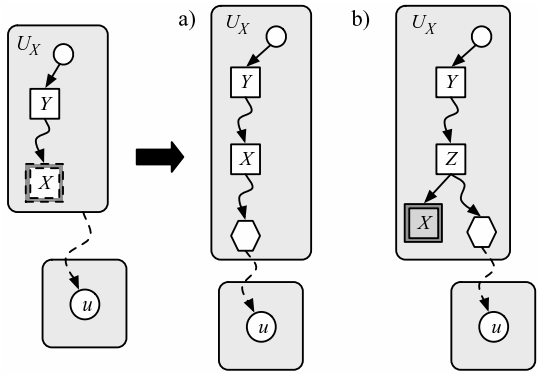

Figure 11. The possible scenarios in which the highest active transaction Y in U_X is an ancestor of X, and U_X is an ancestor of u (i.e., line 11 of Figure 8 returns true). Arrows represent paths between nodes (i.e., many nodes are omitted): see Figures 2 and 9 for definitions. The block arrow shows implication from the left side to either (a) or (b).

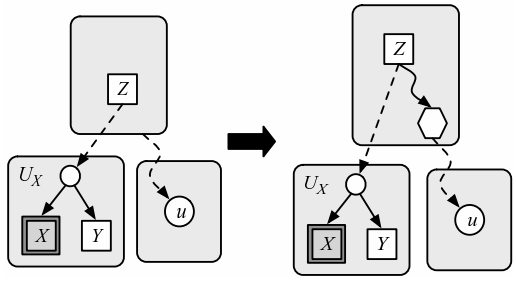

Figure 12. The scenario in which the highest active transaction Y in U_X is not an ancestor of X, and $Z = \texttt{xparent}[U_X]$ is an ancestor of u (i.e., line 15 of Figure 8 returns true). The block arrow shows implication from the left side to the situation on the right.

to Z. Thus, if $\mathsf{trace}(Z)$ is an ancestor of u, we conclude that A has committed to an ancestor of u, as shown in Figure 12.

Section 5 describes how to find the trace containing a particular computation-tree node (i.e., computing $\mathsf{trace}(A)$). Section 6 describes how to maintain the highest active transaction of any trace (used in line 9). Section 7 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 ($\texttt{xparent}[A]$) is trivial. Section 8 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 A be a node in the computation tree, and let u be a currently executing memory access. Suppose that A does not have an aborted ancestor. Then* $\text{XCONFLICT}(A, u)$

reports a conflict if and only if the nearest (deepest) active transactional ancestor of A is an ancestor of u.

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

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.

Let Z be the nearest active transactional ancestor of A. 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 A belonging to an active trace (line 5). The nearest active ancestor of A 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 11). 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 A, and XConflict performs the correct test in lines 15–17. □

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

5. TRACE MAINTENANCE

This section describes how to maintain trace membership for each node in the computation tree $\mathcal{C}$ subject to queries $\mathsf{trace}(A)$ for any $A \in \mathcal{C}$. 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 [6, 14].

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 as the program executes using a disjoint-sets data structure [9, Chapter 21]. Since traces execute on a single worker, we do not lock the data structure on update (UNION) operations. Bags are merged in a way similar to the SP-bags [13, 14] used by the local tier of SP-hybrid [6, 14]. 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 [6, 14]. 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 6 and 8), the global lock is released.

To query what trace a node A 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 [6, 14]. Merging bags takes $O(1)$ amortized time, but an optimization [14] gives a technique that improves UNIONs to worst-case $O(1)$ time whenever the amortization might adversely increase the program's critical path.

6. HIGHEST ACTIVE TRANSACTION

This section describes how to maintain the highest active transaction in a trace, used in line 9 of Figure 8. In particular, XConflict finds highest active transaction in $O(1)$ time.

For each nontransactional S-node S, we have a field $\mathit{nextx}[S]$ 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 $\mathit{nextx}[S]$ is defined to be the nearest, active descendant transaction of S in U. Otherwise, $\mathit{nextx}[S] = \texttt{null}$.

Finding the highest active transaction is simply a call to $\mathit{nextx}[\texttt{head}[U]]$, 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 $\mathit{nextx}[S]$. If $\mathit{nextx}[S] = \texttt{null}$, then set $\mathit{nextx}[S] \leftarrow X$. Otherwise, do nothing. Whenever completing a transaction X, check $\mathit{nextx}[S]$. If $\mathit{nextx}[S] = X$, then set $\mathit{nextx}[S] \leftarrow \texttt{null}$. Otherwise, do nothing. Whenever encountering a nontransactional S-node S'. If $\mathit{nextx}[S] = \texttt{null}$, 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 XCilk 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*$[S] \leftarrow$ `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 5).

7. 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 8, illustrated by U in Figure 10 (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 5, 6, and 8). Note that unlike the data structures in Sections 5, 6, and 8, the updates to supertraces do not occur when steals occur. To prove good performance (in Section 9), 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 [6].

At any point during program execution, a completed trace $U \in Q$ belongs to a ***supertrace*** $K = \mathsf{strace}(U) \subseteq Q$. 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 8), 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 [9, 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 $J(\mathcal{C})$).[10] If `head`$[U']$ is `ABORTED`, then we skip U'. If `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., $A \in U$, $B \in \texttt{children}(A)$, $B \in U' \neq U$), we set that root of the child trace (B) 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 $U \in Q$, let $K = \mathsf{strace}(U)$, and let $U' =$ rep$[K]$. Exactly one of the following cases holds.*

1. *Either* `head`$[U']$ *is* `ABORTED`*, or*
2. `head`$[U']$ *is* `COMMITTED` *and* $\mathsf{trace}(\texttt{parent}[\texttt{head}[U']])$ *is active.*

8. 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 8) and a "global" ancestor query of two different traces (lines 11 and 15 of Figure 8). 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). Many details are omitted due to space constraints.

Local ancestor queries

XCilk executes a trace on a single worker, and each trace is executed in depth-first order. We thus view a trace execution as a depth-first execution of a computation (sub)tree (or a depth-first tree walk).

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 descendents have finished). (This same labeling appears in depth-first search in [9, 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

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

not work for ancestor queries between traces. Instead, we keep two total orders on the traces dynamically using order-maintenance data structures [5, 11]. 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 5 and 6. Since our global ancestor-query data structure resembles the global series-parallel-maintenance data structure in SP-hybrid [6], 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 $O(1)$.

9. PERFORMANCE CLAIMS

The following theorem bounds the running time of an XCilk 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 [14], and we omit the details due to space constraints. 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 worker being stolen from is making progress on the original computation. (That is, whenever stealable, a worker 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 worker holding the lock makes any progress.

The following theorem states the running time of an XCilk 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 [14]).

THEOREM 5. *Consider an XCilk 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 or memory contention 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.*

For illustration, consider a program where all concurrent paths access disjoint sets of memory locations. 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.

Another 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 to have better constants hidden in the big-O.

The XCilk design we describe does not provide any reasonable performance guarantees when we allow multiple readers. Even in the case where transactions do not conflict with each other, and when concurrent read operations never wait to acquire an access stack lock, it appears that write operation may check for conflicts against potentially many readers in a reader list (some of which might 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 a memory location. It is part of future work to improve the XCilk design and analysis in the presence of multiple readers.

10. CONCLUSIONS

The XCilk model presented in Section 3 describes one approach for implementing a software transactional memory system that supports transactions with nested fork-join parallelism. XCilk design was guided by a few major goals.

- Supporting nested transactions of arbitrary 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 XCilk guarantees provably good completion time in the case when there are no aborts or contention, and all accesses are treated as writes.

XCilk support eager updates and eager conflict detection in order to achieve to these design goals. A TM system with lazy updates has to write back its changes when a transaction commits. Similarly, a TM system with lazy conflict detection must validate a transaction upon its commit. In both cases, the commit of a transaction X may perform work proportional to the memory footprint of X. Therefore, operations performed by an X that are nested k-deep may have to be written back/validated k times (once for each ancestor of X), potentially increasing the work k-fold.

XCilk also has a problem with memory footprint. Due to lazy cleanup on aborts, and fast commits, the access stack for a memory location may grow and require space proportional to the number of accesses to that location. Also, access stacks may contain pointers to transaction logs that persist long after the transactions 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.

It would be interesting to see if XCilk-like mechanisms are useful for high-performance languages like Fortress [3] and X10 [12]. 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.

Finally, we do not claim that XCilk's design is particularly efficient as currently written. For example, one might be able to implement transactions with nested parallelism more efficiently or simply by limiting the nesting depth. We would like to implement the system in the Cilk runtime system to evaluate its practical performance and explore ways to optimize the implementation.

References

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. 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.

[2] K. Agrawal, C. E. Leiserson, and J. 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*.

[3] E. Allen, D. Chase, J. Hllett, V. Luchango, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The fortress language specification, version 1.0 β. Technical report, Sun Microsystems, Inc., March 2007.

[4] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. *IEEE Micro*, 26(1):59–69, Jan. 2006.

[5] 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.

[6] M. A. Bender, J. T. Fineman, S. Gilbert, and C. 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.

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 25 1996.

[8] C. Blundell, E. C. Lewis, and M. M. K. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), Nov 2006.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, second edition, 2001.

[10] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006.

[11] 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.

[12] K. Ebcioglu, V. Saraswat, and V. 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)*.

[13] M. Feng and C. 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.

[14] J. 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.

[15] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. 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.

[16] M. Herlihy, V. Luchangco, M. Moir, and W. 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.

[17] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 289–300, 2003.

[18] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, March 2006.

[19] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. 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.

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

[21] Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science. *Cilk 5.4.2.3 Reference Manual*, Apr. 2006.