Efficient Detection of Determinacy Races in Transactional Cilk Programs

Xie Yong

December 17, 2003

Abstract

One of the main obstacle to the widespread adoption of shared-memory parallel programming is the enforcing of atomicity in coordinating concurrent tasks. "Transactions-everywhere" [7], proposed by Leiserson et al, is a new methodology for shared-memory parallel programming with hardware supported atomicity to ease software developers in writing parallel applications. A parallel multithreaded program in Cilk could be atomized into a transactional Cilk program by partitioning the program into multiple atomic transactions. However, a parallel transactional Cilk program may behave nondeterministically due to bugs in the code. These bugs are called determinacy races, and they result from unintended interactions between transactions in accessing shared memory. We have implemented a provably efficient transactions-everywhere race detector for transactional Cilk, which we call Transactional Nondeterminator . If a transactional Cilk program runs on a given input data set has a determinacy race, our debugging tool guarantees to detect and localize the first occurancy of determinacy race.

The current implemented Transactional Nondeterminator is a serial program that is based on Tarjan's nearly linear-time least-common-ancestors algorithm for detecting ted race in series-parallel directed acyclic graphs. For a transactional Cilk program that runs T time on one processor and uses v shared-memory locations, the Transactional Nondeterminator uses $O(v)$ memoery spaces, and runs in $O(T\alpha(v,v))$, where α is Tarjan's functional inverse of Ackermann's function. We tested the Transactional Nondeterminator using a variety of Cilk program benchmarks, and the slowdown is less than 15, which we contend is an acceptable slowdown for debugging purpose.

1 Introduction

One of the main obstacle to the widespread adoption of shared-memory parallel programming is the enforcement of atomicity in coordinating concurrent tasks. When writing parallel applications in share-memory systems, one of the thinkings required of software developers is the concerning of concurrent shared-memory accesses, and this type of thinking is often unintuitive and error-prone. Typical way of enforcing atomicity is to use locking protocols, but these protocols can introduce other complications such as deadlock, priority inversion, etc. One way to avoid locks is to use lock-free data-structure. Researchers have investigated techniques for implementing lock-free concurrent data structures using software techniques, but software implementations of lock-free data structures often do not perform as well as their locking-based counterparts. Herligy and Moss [4] proposed *transactional memory* as a way to ease the writing of concurrent programs. It allows a program to read and modify multiple, disparate memory locations as a single atomic operation, and avoids

the problems related to locks. The authors also claimed the efficiency of transactional memory on a prototype based on a cache-consistency mechanism. Software based transactional memory (STM) has been proposed recently (e.g. in [10, 5, 9]), but the overhead is high.

The idea of "transaction everywhere" [7], proposed by Charles E. Leiserson, is a methodology for parallel programming in which every instruction becomes part of a transaction. Hardware transactional memory provides low enough overhead to make transactions-everywhere viable in practice. Preliminary investigation of language support has been done focusing on providing atomicity within Cilk [8], and we will also use Cilk as the base to build a data-race detector for transactions-everywhere.

In Cilk's transactions-everywhere or transactional Cilk, programs are divided into atomic transactions by specifying the begining (*Cilk_transaction_begin*) and ending point (*Cilk_transaction_end*) of a transaction, and this can be done either by the programmer explicitly or compiler automatically. We call this division process *atomization* of the program. An *atomization strategy* is a method that defines where to partition the program. One conservative atomization strategy is to make each Cilk thread an atomic transaction, but this approach is inefficient, because threads can have very long execution time, and aborting long threads incurs a lot of redundancy and overheads. Therefore, we should try to atomize the threads into smaller transactions, but only if doing so does not affect the correctness the program, otherwise determinacy races will occur.

There are several work done in the past on data race detection (e.g. [1, 3]), but they focus on a non-transactional environment. Kai did pioneering work in this area. In his thesis [6], he laid the mathematical foundation for studying the constraints on transaction scheduling, and formulated the definition of data race in transactions-everywhere setting. He also proved data-race detection is NP-complete using his definition of data race, and he proposed an algorithm that approximately detects data races.

This paper will define the determinacy race in transactional Cilk, and describes a system we call "Transactional Nondeterminator " to detect the determinacy races. The Transactional Nondeterminator takes as input a transactional Cilk program and an input data set. If determinacy races exists, the Transactional Nondeterminator will localize the bug, providing varaible name, file name and line number.

The Transactional Nondeterminator was implemented by modifying the original Cilk compiler and runtime system. Each read and write in the user's program is instrumented by the Transactional Nondeterminator 's compiler to perform checking at runtime. The Transactional Nondeterminator executes a transactional Cilk program in a serial, depth-first fashion (like C execution), but it performs checks when reads and writes occur. The core of Transactional Nondeterminator is an algorithm, which we call TERD , and it was implemented based on an efficient Disjoint-Set datastructure [2] to manage disjoint sets of elements. The Transactional Nondeterminator is tested using several Cilk application benchmarks, and the slow down compared to a serial execution of the benchmarks is within 15, which we contend is an acceptable slowdown for debugging purpose.

The remainder of the paper is organized as follows. Section 2 define the determinacy-race in transactional Cilk. Section 3 presents the TERD algorithm with proof of correctness. Section 4 describes the implementation of Transactional Nondeterminator with empirical result, and we offer some concluding remarks in Section 5 together with some discussion on furture work.

```
1: cilk void list_insert ( double new_data ) {

2:   Cilk_transaction_begin; // transaction t1 starts
3:   Node *pnode = malloc ( sizeof(Node) );
4:   Cilk_transaction_end;   // transaction t1 ends

5:   Cilk_transaction_begin; // transaction t2 starts
6:   pnode->next = head;
7:   Cilk_transaction_end;   // transaction t2 ends

8:   Cilk_transaction_begin; // transaction t3 starts
9:   pnode->data = processing ( new_data ) ;
10:  Cilk_transaction_end;   // transaction t3 ends

11:  Cilk_transaction_begin; // transaction t4 starts
12:  head = pnode;
13:  Cilk_transaction_end;   // transaction t4 ends

14:}
```

Figure 1: A sample cilk procedure which inserts a new node to a linked list. head is a pointer pointing at the first element of the list, and accesses of head are separated into two transaction, t2 and t4.

2 Definition of Determinacy-Race in Transactional Cilk

In this section, we examine the example of inserting nodes into a linked list, and use it to introduce the concept of correctness of a program in transactions-everywhere setting, followed by a formal definition of data-race in transactional Cilk.

2.1 Motivating Example

Conventional definition of a data race used in [1, 3] is not applicable in transactions-everywhere setting, simply because atomicity is achieved by the hardware transactional memory (HTM), and no data race will be reported using the conventional definition. So we make the assumption that if only two concurrent transactions access the shared memory, any order of execution of the two transactions is assumed to be correct. However, the access pattern of shared-memory locations between three or more transactions in transactions-everywhere could cause the execution of a program to behave non-deterministically. For example, the code in Figure 1 has a potential problem, because if two processors P_0 and P_1 execute the code concurrently, accesses to head could be interleaved, which causes inserted element to be overwritten, and it is shown in Figure 2(b). This motivates the definition of data-race in transactions-everywhere.

2.2 Definition of Data-Race

We first define some notations to express the relationship between transactions:

- transaction x is in parallel with transaction y: $x \parallel y$
- transaction x precedes transaction y: $x \prec y$

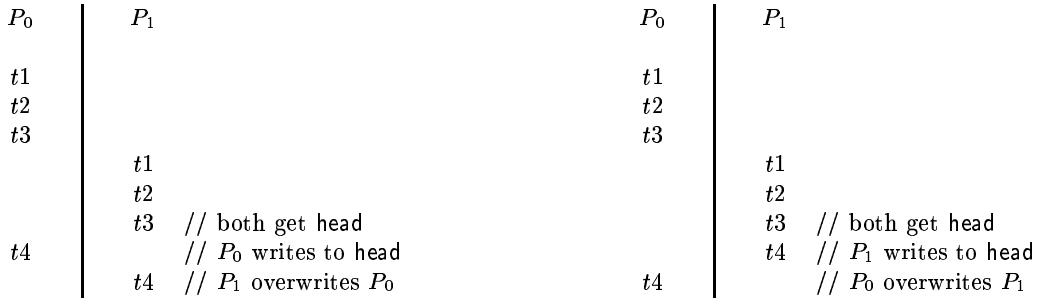

Figure 2: (a) P_0 and P_1 write to the same head, and the data written by P_0 is overwritten by P_1. **(b)** P_0 and P_1 write to the same head, and the data written by P_1 is overwritten by P_0.

Definition 1 *Transaction x precedes transaction y ($x \prec y$) if and only if one of the following is satisfied*

- *x and y are in the same procedure, and there is a path from x to y in the procedure.*
- *procedure $proc_x$ precedes procedure $proc_y$ ($proc_x \prec proc_y$), and x is in $proc_x$, and y is in $proc_y$.*

□

Definition 2 *Transaction x is in parallel with transaction y, $x \parallel y$, if and only if thread t_x is in parallel with t_y ($t_x \parallel t_y$), and x is in t_x, and y is in t_y.* □

We also define how transaction accesses memory locations based on how the instructions inside the transaction access the momery locations.

Definition 3 *Transaction x `writes` memory location l if and only if there exists an instruction i in x such that i writes l.* □

Note that if transaction x writes l, there could be multiple instructions in a transaction x that writes l, but we only consider the write once.

Definition 4 *Transaction x `reads` memory location l if and only if there exists an instruction i in x such that i `reads` l, and there does not exist instruction j in x, such that j `writes` l before i.* □

Note that, if instruction j exists in x, then any read (within x) after j will read the most recent value written by the instruction in x only, and is not affected by any other transactions' access to the shared memory location. Also, there could be multiple `read` instructions in a transaction that satisfy Definition 4, but we only consider it once.

Definition 5 *Data race in transactions-everywhere occurs if and only if there exists distinct transactions x, y, and z, such that all x, y, and z access a shared-memory location, l, and $x \parallel z$, $y \parallel z$, $x \prec y$, and also one of the following is satisfied*

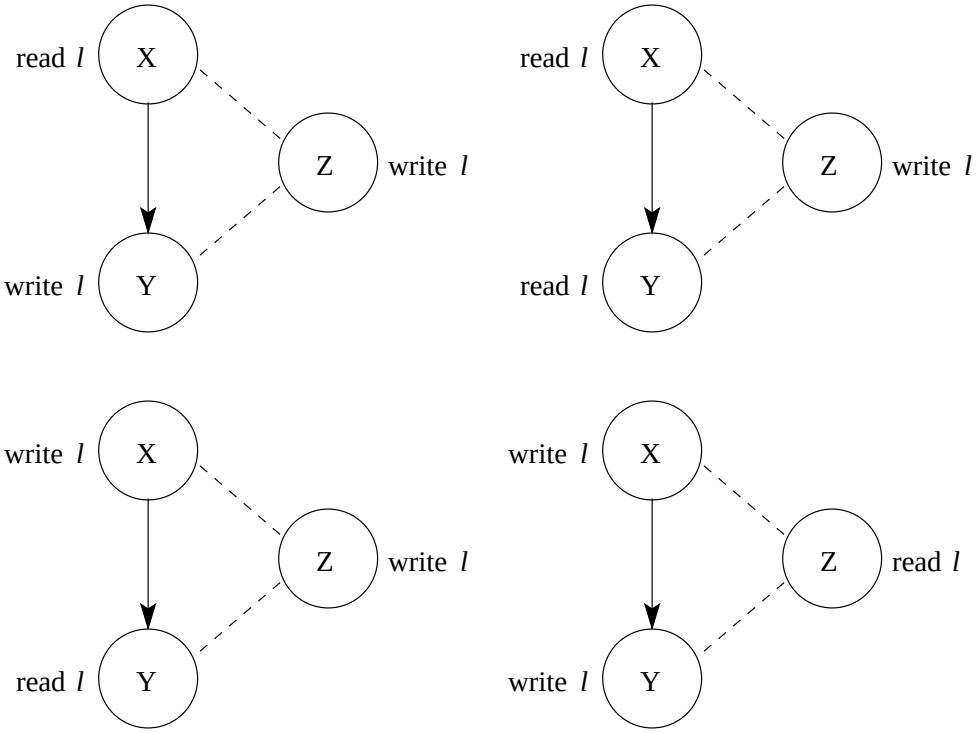

Figure 3: The four possible ways for data race to occur between transactions X, Y and Z in accessing a common shared memory location l. Each transaction is represented as a circle. The solid arrow represents precedence relationship between transactions, and the dashed line represents the relationship between two parallel transactions which access a shared memory location and at least one is a `write` .

1. *x `reads` l, y `writes` l, and z `writes` l.*
2. *x `writes` l, y `reads` l, and z `writes` l.*
3. *x `reads` l, y `reads` l, and z `writes` l.*
4. *x `writes` l, y `writes` l, and z `reads` l.*

□

The four ways of how x, y and z interact with each other in accessing memory location l are shown in Figure 3. We illustrate Defition 5 using examples.

1. x `reads` l, y `writes` l, and z `writes` l: similar to the example (insert new element at head of list) shown in Figure 1, x reads pointer *head*, and y writes *head* to point to the newly inserted element, but depending when z writes *head*, the newly added element may be overwritten.
2. x `writes` l, y `reads` l, and z `writes` l: in this case, the read in y is supposed to read the value written by x, but depending on when z writes its value, y may not read the intended value.
3. x `reads` l, y `reads` l, and z `writes` l: the two reads in x and y will read different values of l depending when z writes l, but if x and y are merged into a single transaction, then they are guranteed to read the same value regardless of when z writes l.

4. x `writes` l, y `writes` l, and z `reads` l: the value z will read depends on when x and y write l, it could be l_{before}, which is the value of l before x writes, or l_x, which is the vaule x writes, or l_y, which is the value y writes. But if x and y are merged into a single transaction, then only l_{before} and l_y are possible to be read by z.

The cases in Definition 5 may or may not be bugs in real applications, but it is the race detector's responsibility to remind programmers of possible non-deterministic behavior in the program.

3 Algorithm for Data-Race Detection

This section describes the algorithm for detecting the data-race in Definition 5, followed by proof of correctness.

3.1 Transactions-Everywhere Race Detector (TERD) Algorithm

The TERD alogrithm is an extension of the SP-Bags algorithm in [3], and it is a serial algorithm. It uses the fact that any transactional Cilk program can be executed on one processor in a depth-first (C-like) fashion and conforms to the semantics of the C program that results when all `spawn`, `sync` and `transaction-begin/end` keywords are removed. As the TERD algorithm executes, it employs several data structures to determine which procedure instances have the potential to execute "in parallel" with each other, and is thereby able to check and report the first occurance of determinacy race access patterns.

The TERD algorithm maintains a count of the number of transactions, which is also the ID of the current transaction: `current-transaction-id`, and fourteen *shadow spaces* of shared memory. The following four shadow spaces capture the last read/write of memory location l in series/parallel.

- LAST-SERIAL-READ[l]: the ID of the last procedure that reads l and is supposed to be in series with the current procedure
- LAST-SERIAL-WRITE[l]: the ID of the last procedure that writes l and is supposed to be in series with the current procedure
- LAST-PARALLEL-READ[l]: the ID of the last procedure that reads l and is supposed to be in parallel with the current procedure
- LAST-PARALLEL-WRITE[l]: the ID of the last procedure that writes l and is supposed to be in parallel with the current procedure

Eight more shadow spaces are needed to record the access patterns of the memory location l.

- READ-READ-1[l], READ-READ-2[l]: the IDs of the procedures that last read l in series
- READ-WRITE-1[l], READ-WRITE-2[l]: the IDs of the procedures that last read and write l in series
- WRITE-WRITE-1[l], WRITE-WRITE-2[l]: the IDs of the procedures that write l in series
- WRITE-READ-1[l], WRITE-READ-2[l]: the IDs of the procedures that write and read l in series

Spawn procedure F:
$S_F \leftarrow$ MAKE-SET(F)
$P_F \leftarrow \emptyset$

Sync in procedure F:
$S_F \leftarrow$ UNION(S_F, P_F)
$P_F \leftarrow \emptyset$

Return from procedure F' to F:
$P_F \leftarrow$ UNION($S_F, P_{F'}$)

Transaction-Begin:
current-transaction-id ++

Figure 4: The TERD algorithm (part 1).

Furthermore, we need to keep track of whether a transaction `reads` or `writes` memory location l already. As specified in Definition 4, if a transaction already `writes` l, then further reads and writes inside the same transaction will not be evaluated, and if a transaction already `reads` l, then further reads inside the same transaction will not be evaluated either.

- TRANSACTION-ID-READ[l]: the ID of the last transaction that reads l
- TRANSACTION-ID-WRITE[l]: the ID of the last transaction that writes l

During the execution of the TERD algorithm, two "bags" of procedure ID's are maintained for every Cilk procedure on the call stack. The $S-bag$ S_F of a procedure F contains the ID's of those descendants of F's completed children that logically "precede" the currently executing thread, as well as the ID of F itself. The $P-bag$ P_F of a procedure F contains the ID's of those desendants of F's completed children that operate logically "in parallel" with the currently executing thread.

The TERD algorithm uses Disjoint-Set data structure [2] to maintain the relationship between two procedures (i.e. in series or logically in parallel), and operations for `spawn`, `sync` and `return` are shown in Figure 4. The TERD alogrithm also performs additional operations whenever one of the two actions occurs: `write` and `read`, which are shown in Figure 5 and Figure 6.

3.2 Correctness of the TERD-Algorithm

To show that the TERD is correct, we begin by reviewing the series-parallel structure of Cilk dag, and extend it to transactional Cilk dag. Then, we analyse the canonical parse tree for transactional Cilk dag, and define subpattern $\pi(x, y)$ to be a pair of transactions x and y in series, which access

```
Read meomery location [l] by transaction T procedure F:
  if (TRANSACTION-ID-READ[l] != current-transaction-id or
        TRANSACTION-ID-WRITE[l] != current-transaction-id)
      TRANSACTION-ID-READ[l] ← current-transaction-id
      EVAL-READ (l, F)


EVAL-READ (l, F):
  ▷ test ww-r race
  if (FIND-SET(WRITE-WRITER-1[l]) is a P-bag)
      report data race exists
  if (FIND-SET(LAST-SERIAL-READ[l]) is a S-bag)
      ▷ test w-rr race
      if (FIND-SET(LAST-PARALLEL-WRITE[l]) is a P-bag
        and LAST-PARALLEL-WRITE[l] < READ-READ-1[l])
        report data race exists
      if (FIND-SET(READ-READ-1[l]) is a S-bag)
        READ-READ-1[l] ← LAST-SERIAL-READ[l]
        READ-READ-2[l] ← F
  if (FIND-SET(LAST-SERIAL-READ[l]) is a P-bag)
      LAST-SERIAL-READ[l] ← F
  if (FIND-SET(LAST-SERIAL-WRITE[l]) is a S-bag)
      ▷ test w-wr race
      if (FIND-SET(LAST-PARALLEL-WRITE[l]) is a P-bag
        and LAST-PARALLEL-WRITE[l] < WRITE-READ-1[l])
        report data race exists
      if (FIND-SET(WRITE-READ-1[l]) is a S-bag)
        WRITE-READ-1[l] ← LAST-SERIAL-WRITE[l]
        WRITE-READ-2[l] ← F
  if (FIND-SET(LAST-PARALLEL-READ[l]) is a S-bag)
      LAST-PARALLEL-READ[l] ← F
```

Figure 5: The TERD algorithm (part 2).

```
Write meomery location [l] by transaction T procedure F:
  if (TRANSACTION-ID-WRITE[l] != current-transaction-id)
       TRANSACTION-ID-WRITE[l] ← current-transaction-id
       TRANSACTION-ID-READ[l] ← current-transaction-id
       EVAL-WRITE (l, F)


EVAL-WRITE (l, F):
  ▷ test rw-w race
  if (FIND-SET(READ-WRITER-1[l]) is a P-bag)
       report data race exists
  ▷ test wr-w race
  if (FIND-SET(WRITE-READ-1[l]) is a P-bag)
       report data race exists
  ▷ test rr-w race
  if (FIND-SET(READ-READ-1[l]) is a P-bag)
       report data race exists
  if (FIND-SET(LAST-SERIAL-WRITE[l]) is a S-bag)
       ▷ test r-ww race
       if (FIND-SET(LAST-PARALLEL-READ[l]) is a P-bag
         and LAST-PARALLEL-READ[l] < WRITE-WRITE-1[l])
         report data race exists
       if (FIND-SET(WRITE-WRITE-1[l]) is a S-bag)
         WRITE-WRITE-1[l] ← LAST-SERIAL-WRITE[l]
         WRITE-WRITE-2[l] ← F
  if (FIND-SET(LAST-SERIAL-WRITE[l]) is a P-bag)
       LAST-SERIAL-WRITE[l] ← F
  if (FIND-SET(LAST-SERIAL-READ[l]) is a S-bag)
       ▷ test w-rw race
       if (FIND-SET(LAST-PARALLEL-WRITE[l]) is a P-bag
         and LAST-PARALLEL-WRITE[l] < READ-WRITE-1[l])
         report data race exists
       if (FIND-SET(READ-WRITE-1[l]) is a S-bag)
         READ-WRITE-1[l] ← LAST-SERIAL-READ[l]
         READ-WRITE-2[l] ← F
  if (FIND-SET(LAST-PARALLEL-WRITE[l]) is a S-bag)
       LAST-PARALLEL-WRITE[l] ← F
```

Figure 6: The TERD algorithm (part 3).

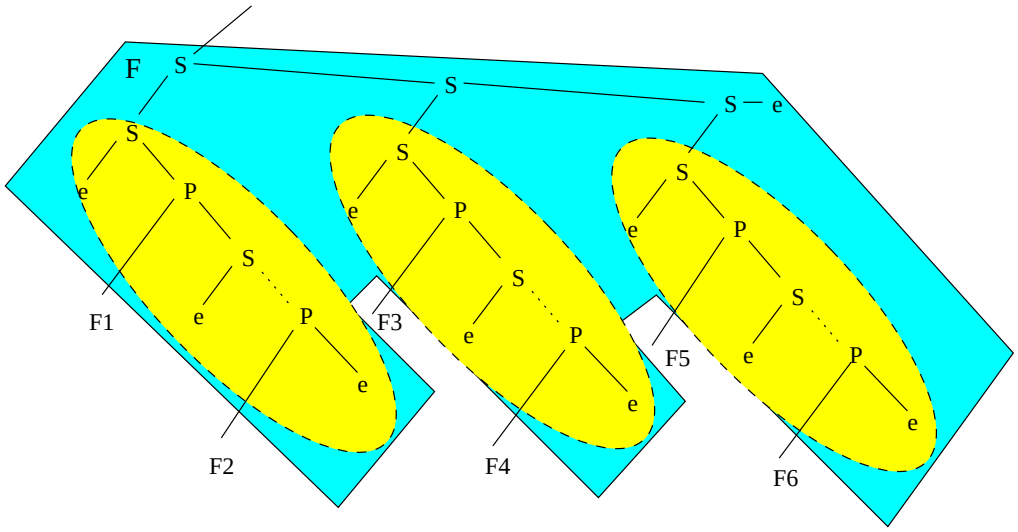

Figure 7: A sample SP parse tree.

a shared memory location l. We use subpatterns as the basic units to show some properties of the transactional Cilk dag. By using the properties, we prove that the TERD algorithm detects and report the first occurance of determinacy race if and only if it exists one inside the program.

3.2.1 Transactional Cilk Dag

Feng and Leiserson have shown the series-parallel structure of Cilk dag in [3], and we will extend it to show that transactional Cilk dag is also a series-parallel dag.

Lemma 6 *A transactional Cilk parallel control-flow dag is a series-parallel (SP) dag.*

Proof. Given the Cilk dag $G = (V, E)$, where V contains all the Cilk threads, and V consists of spawn and sync edges. After atomization, the transactional Cilk dag $G' = (V', E')$, where V' contains all the transactions, and V' consists of spawn, sync, and continual edges which are between Cilk transactions within the same Cilk thread. Notice that, the difference between G and G' is that a vertex v in V may be partitioned into a set of vertexes $v'_{0..m-1}$ in V', where m is the number of transactions inside a Cilk thread, and $v'_{0..m-1}$ are connected by continual edges in series. This process is consistent with the recursive definition of SP dag, which means G' is also a SP dag. □

As presented in [3], a SP dag can be represented by a binary parse tree, and we will view the structure of Cilk transactions dag as a binary parse tree. The creation process of a SP parse tree for Cilk transactions follows the way how transactional Cilk dag is created. Firstly, we construct the Cilk SP parse tree in the same way as specified in [3] (a sample is shown in Figure 7, and then transform each thread into a tree of transactions. Note that, since transaction in the same thread are always in series, the tree only consists of S-nodes and transactions, and nothing else. The transformation is shown in Figure 8. In the remainder of the paper, we refer SP dag as the transactional Cilk SP dag.

We also show that the properties of Cilk dag's parse tree can be extended to transactional Cilk dag's parse tree as well.

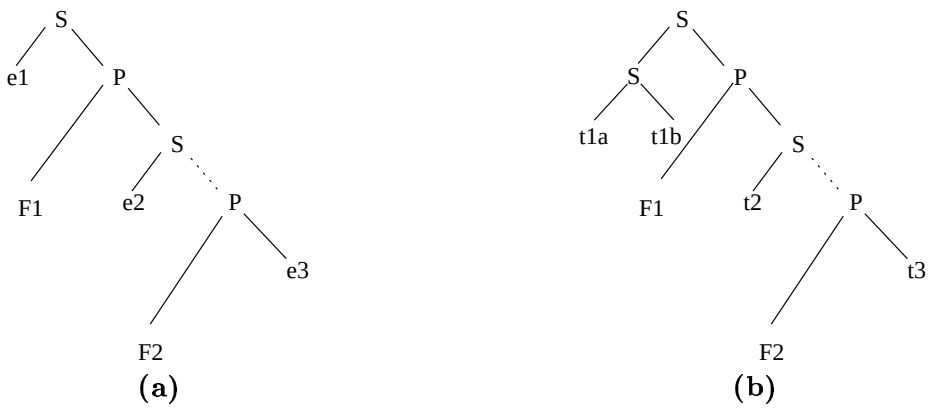

Figure 8: (a) Part of a Cilk SP parse tree. **(b)** The corresponding transactional Cilk parse tree of **(a)** after transformation (atomization). Thread $e1$ is transformed into a tree of two transactions $t1a$ and $t1b$ with a S-nodes as the root. Thread $e2$ corresponds to transaction $t2$, and thread $e3$ corresponds to transaction $t3$.

Lemma 7 *Let t_1 and t_2 be distinct transactions in a transactional Cilk dag, and let LCA(t_1, t_2) be their least common ancestor in a parse tree for the dag. Then, $t_1 \parallel t_2$ if and only if LCA(t_1, t_2) is a P-node.*

Proof. See proof of Lemma 4 in [3].

Corollary 8 *Let t_1 and t_2 be distinct transactions in a transactional Cilk dag, and let LCA(t_1, t_2) be their least common ancestor in a parse tree for the dag. Then, $t_1 \prec t_2$ if and only if LCA(t_1, t_2) is a S-node, and t_1 is in the left subtree of LCA(t_1, t_2) , and t_2 is in the right subtree of LCA(t_1, t_2) .* □

The TERD algorithm takes advantage of relationships among transactions that can be derived from the serial, depth-first execution order of the dag.

Definition 9 *In a serial, depth-first execution of a transactional Cilk program, transaction t_1 is executed before transaction t_2 is denoted as $t_1 < t_2$.* □

Lemma 10 *Suppose that three transactions t_1, t_2 and t_3 execute in order in a serial, depth-first execution of a transactional Cilk dag (i.e. $t_1 < t_2 < t_3$), and suppose that $t_1 \prec t_2$, and $t_1 \parallel t_3$. Then, we have $t_2 \parallel t_3$.*

Proof. Assume for the purpose of contradiction that $t_2 \prec t_3$. Then, since $t_1 \prec t_2$, we have $t_1 \prec t_3$ by transitivity, contradicting the assumption that $t_1 \parallel t_3$. □

Lemma 11 *Suppose that three transactions t_1, t_2 and t_3 execute in order in a serial, depth-first execution of a transactional Cilk dag (i.e. $t_1 < t_2 < t_3$), and suppose that $t_1 \parallel t_2$, and $t_2 \parallel t_3$. Then, we have $t_1 \parallel t_3$.*

Proof. Consider the parse tree of the transactional Cilk dag with t_1, t_2 and t_3. Let $a_1 = LCA(t_1, t_2)$ and $a_2 = LCA(t_2, t_3)$. Lemma 7 implies that both a_1 and a_2 are P-nodes. Because $t_1 < t_2 < t_3$, one can show that either a_1 or a_2 is the least common ancestor of t_1 and t_3, and since both t_1 and t_2 are P-nodes, it follows from Lemma 7 that $t_1 \parallel t_3$. □

Lemma 12 *Suppose that three transactions t_1, t_1' and t_2 execute in order in a serial, depth-first execution of a transactional Cilk dag (i.e. $t_1 < t_1' < t_2$), and suppose that $t_1 \parallel t_1'$, and $t_1 \prec t_2$. Then, we have $t_1' \prec t_2$.*

Proof. Assume for the purpose of contradiction that $t_1' \parallel t_2$ (t_2 cannot be preceding t_1 because $t_1 < t_2$). Then, since $t_1 \parallel t_1'$, Lemma 11 implies that $t_1 \parallel t_2$, contradicting the assumption that $t_1 \prec t_2$. □

Similar to the SP-bag algorithm in [3], we define a mapping h of transactions or nodes in the parse tree to procedures in the spawn tree to be the *procedurification* function for the parse tree. This function is used in the next lemma to relate the S-nodes and P-nodes in the parse tree to procedure ID's in the S-bags and P-bags during the execution of the TERD algorithm.

Lemma 13 *Consider the execution of the TERD algorithm on a given transactional Cilk dag. Let h be the procedurification mapping the canonical parse tree for the dag to procedures in the spawn tree. Suppose thread t_1 is executed before thread t_2, and let $a = LCA(t_1, t_2)$ be their least common ancestor in the parse tree. If a is an S-node, then the procedure ID for $h(t_1)$ belongs to the S-bag of $h(a)$ when t_2 is executed. Similaly, if a is a P-node, then the procedure ID for $h(t_1)$ belongs to the P-bag of $h(a)$ when t_2 is executed.*

Proof. Given the difference between transactions and threads is covered by the procedurification function, the proof is the same as the proof in Lemma 8 of [3]. □

Corollary 14 *Consider the execution of the TERD algorithm on a given transactional Cilk dag. Let h be the procedurification mapping the canonical parse tree for the dag to procedures in the spawn tree. Suppose transaction t_1 is executed before transaction t_2, and let $a = LCA(t_1, t_2)$ be their least common ancestor in the parse tree. If a is an S-node, then the procedure ID for $h(t_1)$ belongs to the S-bag of $h(a)$ when t_2 is executed. Similaly, if a is a P-node, then the procedure ID for $h(t_1)$ belongs to the P-bag of $h(a)$ when t_2 is executed.*

Proof. Combining Lemma 7, Corollary 8, and Lemma 13. □

3.2.2 Sub-pattern

As described in Defintion 5, the pattern of determinacy race contains a pair of transactions in series (i.e. $x \prec y$) and the third transaction, z, which is in parallel with both x and y. We introduce the defintion of sub-pattern to describe the pair of transactions in series.

Definition 15 *In a transactional Cilk program P, x and y are distinct transactions accesing shared memory location l, $x \prec y$ and $x < y$, a sub-pattern $\pi(x, y)$ is of one of the following four kinds:*

- *rw: x **reads** l and y **writes** l.*
- *wr: x **writes** l and y **reads** l.*

- *rr: x **reads** l and y **reads** l.*
- *ww: x **writes** l and y **writes** l.*

□

Definition 16 *Sub-patterns $\pi(x, y)$ and $\pi(x', y')$ accessing shared location l, are distinct if and only if either $x \neq x$ or $y \neq y'$.* □

Definition 17 *Transaction x has the same kind of access of memory location l as transaction y (both **read** l or both **write** l), denoted as $x \equiv_l y$.* □

Definition 18 *Two distinct sub-patterns $\pi(x, y)$ and $\pi(x', y')$ accessing shared memory location l, are of the same kind, i.e. $\pi(x, y) \equiv_l \pi(x', y')$, if and only if $x \equiv_l x'$ and $y \equiv_l y'$.* □

We will use sub-pattern as a basic unit in the transactional Cilk dag for the following discussion about the relationships between sub-partterns and transactions, and between sub-patterns as well.

Definition 19 *Given $x < y < z$, and all three transactions access shared memory location l, the sub-pattern $\pi(x, y)$ is in parallel with transaction z, i.e. $\pi(x, y) \parallel z$ if and only if $x \parallel z$.* □

Definition 20 *A sub-pattern $\pi(x, y)$ is executed before a transaction z, i.e. $\pi(x, y) < z$, if and only if $y < z$.* □

Definition 21 *A sub-pattern $\pi(x, y)$ precedes a transaction z, i.e. $\pi(x, y) \prec z$, if and only if $\pi(x, y) < z$ and $x \prec z$.* □

Definition 22 *A sub-pattern $\pi(x, y)$ is executed before another sub-pattern $\pi(x', y')$, i.e. $\pi(x, y) < \pi(x', y')$, if and only if $x < x'$ and $y \leq y'$.* □

Definition 23 *A sub-pattern $\pi(x, y)$ precedes another sub-pattern $\pi(x', y')$, i.e. $\pi(x, y) \prec \pi(x', y')$, if and only if $\pi(x, y) < \pi(x', y')$ and $x \prec y'$.* □

We will use the properties of sub-patterns to show some lemmas, which will be used to prove the correctness of the TERD algorithm.

Lemma 24 *Suppose that four transactions x, x', y and z execute in order in a serial, depth-first execution of a transactional Cilk, i.e. $x < x' < y < z$. Also suppose that the sub-pattern $\pi(x, y)$ is in parallel with z, i.e. $\pi(x, y) \parallel z$, and $x \parallel x'$, and $x \equiv_l x'$. Then, we have sub-pattern $\pi(x', y) \parallel z$.*

Proof. Since $x \parallel x'$, and $x \prec y$, Lemma 12 implies that $x' \prec y$. Also because $x \equiv_l x'$, the sub-pattern $\pi(x', y)$ have the same property as $\pi(x, y)$, which is in parallel with z. □

Lemma 24 shows that under the conditions specified, we can safely replace x for x' without missing any possible determinacy race. An illustration of Lemma 24 is shown in Figure 9.

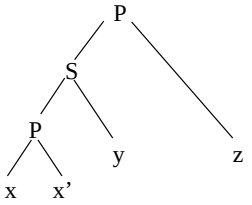

Figure 9: Illustration of Lemma 24.

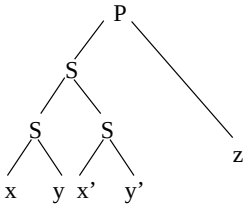

Figure 10: Illustration of Lemma 25.

Lemma 25 *Suppose that two sub-patterns $\pi(x, y)$ and $\pi(x', y')$, and a transaction z execute in order in a serial, depth-first execution of a transactional Cilk, i.e. $\pi(x, y) < \pi(x', y') < z$. Also suppose that the sub-pattern $\pi(x, y)$ is in parallel with z, i.e. $\pi(x, y) \parallel z$, and the two sub-patterns are of the same kind with respect to memory location l, and they are in series, i.e. $\pi(x, y) \equiv_l \pi(x', y')$ and $\pi(x, y) \prec \pi(x', y')$. Then, we have sub-pattern $\pi(x', y') \parallel z$.*

Proof. The proof is similar to the proof of Lemma 10, except that we treat a sub-pattern as a basic unit. Assume for the purpose of contradiction that $\pi(x', y') \prec z$. Then, since $\pi(x, y) \prec \pi(x', y')$, we have $\pi(x, y) \prec z$, contradicting the assumption that $\pi(x, y) \parallel z$. □

Lemma 25 shows that under the conditions specified, we can safely replace sub-pattern $\pi(x, y)$ for $\pi(x', y')$ without missing any possible determinacy race. An illustration of Lemma 25 is shown in Figure 10.

Lemma 26 *Suppose that two transactions z and z', and a sub-pattern $\pi(x, y)$ execute in order in a serial, depth-first execution of a transactional Cilk, i.e. $z < z' < \pi(x, y)$. Also suppose that the z is in parallel with sub-pattern $\pi(x, y)$, i.e. $z \parallel \pi(x, y)$, and the two transactions are of the same kind with respect to memory location l, and they are in series, i.e. $z \equiv_l z'$ and $z \prec z'$. Then, we have $z \parallel \pi(x, y)$.*

Proof. The proof is similar to the proof of Lemma 10, except we treat a sub-pattern as a basic unit. Assume for the purpose of contradiction that $z' \prec \pi(x, y)$. Then, since $z \prec z'$, we have $z \prec \pi(x, y)$, contradicting the assumption that $z \parallel \pi(x, y)$. □

Lemma 26 shows that under the conditions specified, we can safely replace transaction z for z' without missing any possible determinacy race. An illustration of Lemma 26 is shown in Figure 11.

Lemma 27 *Suppose that two transactions z and x', and a sub-pattern $\pi(x, y)$ execute in the order that $z < x < x' < y$ in a serial, depth-first execution of a transactional Cilk. Also suppose that*

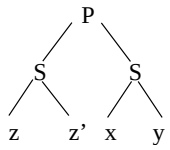

Figure 11: Illustration of Lemma 26.

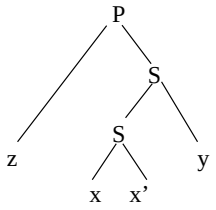

Figure 12: Illustration of Lemma 27.

the z is in parallel with sub-pattern $\pi(x, y)$, i.e. $z \parallel \pi(x, y)$, and the x and x' are of the same kind with respect to memory location l, and they are in parallel, i.e. $x \equiv_l x'$ and $x \parallel x'$. Then, we have $z \parallel \pi(x', y)$.

Proof. Since $x \parallel x'$ and $x \prec y$ because of $\pi(x, y)$, Lemma 12 implies that $x' \prec y$. Also as $x \equiv_l x'$, we can replace x for x', and the least-common ancestor of z and y remains to be a P-node, which means $\pi(x', y)$ is still in parallel with z. □

Lemma 27 shows that under the conditions specified, we can safely replace transaction x for x' without missing any possible determinacy race. An illustration of Lemma 27 is shown in Figure 12.

3.2.3 TERD

As described in Defintion 5, four memory access patterns of the three transactions x, y, and z need to be detected, and due to the asymmetry of the positions of the three transactions, there are eight patterns to be detected in a serial, depth-first execution of a given transactional Cilk program. We charaterize the patterns into two groups:

1. $\pi(x, y) - z$: in which $\pi(x, y) < z$
2. $z - \pi(x, y)$: in which $z < \pi(x, y)$

Theorem 28 *The TERD detects a determinacy race in a transactional Cilk program if and only if a determinacy race exits.*

Proof. Given a transactional Cilk program P, x, y and z are distinct transactions in P, and they access shared memory location l. h is the procedurification function mapping transaction to the procedure which it resides in.

($\Rightarrow$) Suppose TERD detects a determinacy race at transaction t accessing l, W.L.O.G, t can be z in the cases 1-4, and t can be y in the cases 5-8, then one of the following eight must be satisfied: (the order $x < y < z$ for case 1-4 is guaranteed by the way how the sub-patterns are updated, i.e. $x < y$; the order $z < x < y$ for case 5-8 is guaranteed by the check of z is executed before x)

1. $rw - w$ $(\pi(x,y) - z)$: x `reads` l, y `writes` l, z `writes` l, and $x \prec y$. $h(x)$ is in P-bag of $h(z)$.
2. $wr - w$ $(\pi(x,y) - z)$: x `writes` l, y `reads` l, z `writes` l, and $x \prec y$. $h(x)$ is in P-bag of $h(z)$.
3. $rr - w$ $(\pi(x,y) - z)$: x `reads` l, y `reads` l, z `writes` l, and $x \prec y$. $h(x)$ is in P-bag of $h(z)$.
4. $ww - r$ $(\pi(x,y) - z)$: x `writes` l, y `writes` l, z `reads` l, and $x \prec y$. $h(x)$ is in P-bag of $h(z)$.
5. $w - rw$ $(z - \pi(x,y))$: x `reads` l, y `writes` l, z `writes` l, and $x \prec y$. $h(z)$ is in P-bag of $h(y)$.
6. $w - rw$ $(z - \pi(x,y))$: x `writes` l, y `reads` l, z `writes` l, and $x \prec y$. $h(z)$ is in P-bag of $h(y)$.
7. $w - rr$ $(z - \pi(x,y))$: x `reads` l, y `reads` l, z `writes` l, and $x \prec y$. $h(z)$ is in P-bag of $h(y)$.
8. $r - ww$ $(z - \pi(x,y))$: x `writes` l, y `writes` l, z `reads` l, and $x \prec y$. $h(z)$ is in P-bag of $h(y)$.

In Group 1 (cases 1-4), $h(x)$ is in P-bag of $h(z)$ means $x \parallel z$ by Corollary 14, and since $x \prec y$ and $x < y < z$, we have $y \parallel z$, so determinacy race exists. Similarly for Group 2 (case 5-8), $h(z)$ is in P-bag of $h(y)$ means $z \parallel y$, and since $x \prec y$, and $z < x < y$, then $z \parallel y$, so determinacy race exists.

($\Leftarrow$) We will show that if a program contains a determinacy race on a location l, then the TERD will detect and report it. We seperate into two groups:

- Group 1 $\pi(x,y) - z$: when z is executed, it will check the sub-pattern stored inside the shadow spaces. If the procedurification of READ-READ-1[l], READ-WRITE-1[l], WRITE-READ-1[l], and WRITE-WRITE-1[l] are in the P-bag of z, so all group 1 cases 1-4 will be covered. We also need to show the update of the shadow spaces guarantees no race will be missed. Lemma 24 implies that updating of LAST-SERIAL-READ[l] and LAST-SERIAL-WRITE[l] will not miss any races, because TERD uses LAST-SERIAL-READ[l] and LAST-SERIAL-WRITE[l] and the current transaction to update the record of sub-patterns. Lemma 25 implies that updating of sub-patterns (READ-READ-1[l], READ-READ-2[l], ... , WRITE-WRITE-2[l],) will not miss any determinacy race.
- Group 2 $z - \pi(x,y)$: when y is executed, it will check whether the sub-pattern exists between LAST-SERIAL-READ[l] and LAST-SERIAL-WRITE[l] and the current transaction, and Lemma 27 guarantees that the updating of LAST-SERIAL-READ[l] and LAST-SERIAL-WRITE[l] will not miss any races. If the sub-pattern exists, then TERD will examine LAST-PARALLEL-READ[l] and LAST-PARALLEL-WRITE[l] and see whether its procedurification is in the P-bag of the current transaction. Also it will check LAST-PARALLEL-READ[l] and LAST-PARALLEL-WRITE[l] are executed before LAST-SERIAL-READ[l] and LAST-SERIAL-WRITE[l], and this guarantees the execution order of $z < x < y$. Note that if $x < z$, it is not a determinacy race, and TERD will not report it.

□

4 The Transactional Nondeterminator

This section presents the implementation of the Transactional Nondeterminator . We discuss how the Transactional Nondeterminator implements the TERD by modifying the Cilk compiler

Program	Original (seconds)	Transactional Nondeterminator (seconds)	Slowdown
`fib(30)`	3.1	9.6	3.21
`C.K.(5,8)`	2.2	31.2	14.18
`L.U.(512x512)`	1.1	10.6	9.63

Figure 13: The performance of example Cilk programs. Time is measured in seconds.

and runtime system. Empirical data from a variety of benchmark Cilk programs shows that the Transactional Nondeterminator typically runs in less than 15 times the execution time of the original optimized program.

The first phase of checking a user's transactional Cilk program is to run the code through the Cilk compiler with an option that turns on determinacy-race detection. This compiler option produces object code with calls to the Transactional Nondeterminator's runtime system for every read and write of shared memory, as well as the begin and end of a transaction. In addition, the compiler inserts hooks that allow the Transactional Nondeterminator's runtime system to perform actions for every spawn, sync and return operations.

At runtime, before it starts execution, the Transactional Nondeterminator will set up the shadow spaces. We use Unix memory-mapping primitive `mmap()` to fix the starting address of each shadow space so that the shadow space address can be obtained quickly from the corresponding shared-memory address. It also initializes the disjoint-set data structure.

During execution of the user program, the Transactional Nondeterminator performs the TERD algorithm, modified slightly to optimize performance. If the compiler can determine that a memory reference is to a nonshared memory region, such as local variable whose address is never computed, no determancy-race check necessary, because no determinacy-race is possible.

We have measured the performance of the Transactional Nondeterminator on multiple manually designed test cases as well as the benchmark program distributed in the Cilk package. The test is done on a 500-megahertz SUN Ultrasparc with the Solaris 5.8. Figure 13 shows some of the performance figures. The results show that the slowdown is at most 15 times compared to the serial execution of the programs.

5 Conclusion and Future Work

In conclusion, we have defined what a determinacy race in transactions-everywhere setting, and proposed an algorithm to detect it in transactional Cilk. We have implemented a version in the Cilk runtime system and modified the Cilk compiler "cilk2c" to support this. We have tested using manual test cases as well as the benchmark programs included in the Cilk package, and empirical result shows the the slowdown is at most 15 compared to a serial execution of the program.

We also found several open problems arising out of our work. Some programs may intentionally contain nondeterminism, and they might want to specify wildcards to certain memory locations or some Cilk procedures, so how does Transactional Nondeterminator tolerate intended nondeterminism while still catching unintensional determinacy race? How to incorporate more language features like *inlet*? How to parallelize the Transactional Nondeterminator? How to measure the performance of transactional Cilk program? Also, linear time algorithm could be used in TERD algorithm for maintaining the relationship between procedures, which is left for future work.

References

[1] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith Randall, and Andrew F. Stark. Detecting data races in cilk programs that use locks. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Puerto Vallarta, Mexico, June 28–July 2 1998.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw Hill, 1990.

[3] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. Submitted for publication. Available at `ftp://theory.lcs.mit.edu/pub/cilk/spbags.ps.gz`, January 1997.

[4] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of theTwentiethAnnual International Symposium on Computer Architecture*, 1993.

[5] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM Press, 2003.

[6] Huang Kai. Data-race detection in transactions-everywhere parallel programming. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2003.

[7] Charles E. Leiserson. Personal communication. Cambridge, Massachusetts, 2002.

[8] Charles E. Leiserson and Bradley C. Kuszmaul. Transactions everywhere. Available on the Internet from `http://web.mit.edu/sma/events/symposium/2003_symposium/papers/cs/CS028.pdf`, 2003.

[9] V. Luchangco M. Herlihy and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems*, May 2003.

[10] Nir Shavit and Dan Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.