Data-Race Detection in Transactions-Everywhere Parallel Programming

by

Kai Huang

B.S. Computer Science and Engineering, B.S. Mathematics
Massachusetts Institute of Technology, June 2002

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Engineering
in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology
June 2003

Signature of Author ____
Department of Electrical Engineering and Computer Science
May 21, 2003

Certified by ____
Charles E. Leiserson
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Data-Race Detection in Transactions-Everywhere Parallel Programming

by

Kai Huang

Submitted to the Department of Electrical Engineering
and Computer Science on May 21, 2003 in partial fulfillment
of the requirements for the degree of Master of Engineering
in Electrical Engineering and Computer Science

ABSTRACT

This thesis studies how to perform dynamic data-race detection in programs using "transactions everywhere", a new methodology for shared-memory parallel programming. Since the conventional definition of a data race does not make sense in the transactions-everywhere methodology, this thesis develops a new definition based on a weak assumption about the correctness of the target program's parallel-control flow, which is made in the same spirit as the assumption underlying the conventional definition.

This thesis proves, via a reduction from the problem of 3cnf-formula satisfiability, that data-race detection in the transactions-everywhere methodology is an NP-complete problem. In view of this result, it presents an algorithm that approximately detects data races. The algorithm never reports false negatives. When a possible data race is detected, the algorithm outputs simple information that allows the programmer to efficiently resolve the root of the problem. The algorithm requires running time that is worst-case quadratic in the size of a graph representing all the scheduling constraints in the target program.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

Contents

Chapter 1

Introduction

This thesis considers the problem of data-race detection in parallel programs using "transactions everywhere", a new methodology for shared-memory parallel programming suggested by Charles E. Leiserson [26]. Transactions everywhere reduces the amount of thinking required of the programmer concerning concurrent shared-memory accesses. This type of thinking is often unintuitive and error-prone, and represents a main obstacle to the widespread adoption of shared-memory parallel programming.

This introduction first describes Cilk, the parallel-programming language in which we conduct our study. Then, it presents transactions as an alternative system to conventional locks for creating atomic sections and eliminating data races. Finally, it describes the transactions-everywhere methodology for parallel programming with transactions.

The Cilk Language. The results in this thesis apply to parallel programming using transactions everywhere, irrespective of the implementation language. For concreteness, however, we conduct our study in the context of Cilk [2, 3, 22, 13, 39], a shared-memory parallel-programming language developed by Charles E. Leiserson's research group. Cilk is *faithful* extension of C, which means that if all Cilk keywords are elided from a Cilk program, then a semantically correct serial C program, called the *serial elision*, is obtained. The three most basic Cilk keywords are `cilk`, `spawn`, and `sync`. This thesis considers programs that contain only these three extensions.

We illustrate the usage of the three Cilk keywords by an example, which we shall reuse throughout this introduction. Consider the common scenario of concurrent linked-list update, such as might arise from insertions into a shared hash table that resolves collisions using linked lists. The following is a ***Cilk function*** for inserting new data at the head of a shared singly linked list. The keyword `cilk` is placed before a function declaration or definition (in this case the definition of `list_insert`) to indicate a Cilk function. The shared variable `head` points to the head of the list, and each node of type `Node` has a `data` field and a `next` pointer.

```
cilk void list_insert( double new_data )
{
    Node *pnode = malloc( sizeof(Node) );
    pnode->data = process(new_data);
    pnode->next = head;
    head = pnode;
}
```

Cilk functions must be called by using the keyword `spawn` immediately before the function name. A Cilk function call spawns a new thread of computation to execute the new function instance, while the parent function instance continues to execute in parallel. The keyword `sync` is used as a standalone statement to synchronize all the threads spawned by a Cilk function. All Cilk function instances that have been previously spawned by the current function are guaranteed to have returned before execution continues on to the statement after the `sync`. The following segment of code demonstrates `spawn` and `sync`. It inserts two pieces of data into the shared linked list in parallel, and then prints the length of the list.

```
⋮
spawn list_insert(23.118);
spawn list_insert(23.170);
sync;
printf( "The list has length %d.", list_length() );
⋮
```

The `sync` statement guarantees that the `printf` statement only executes after both `spawn` calls have returned. Without the `sync` statement, the action of `list_length` would be unpredictably intertwined with the actions of the two calls to `list_insert`, thus causing an error. Note that the function `list_length` for counting the length of the list is a regular C function.

Cilk functions may call other Cilk functions or C functions, but C functions cannot call Cilk functions. Thus, the function `main` in a Cilk program must be a Cilk function. Also, all Cilk functions implicitly sync[1] their spawned children before returning.

In an execution of a Cilk program, a ***Cilk thread*** is defined as a maximal sequence of instructions without any parallel-control constructs, which in our case are `spawn` and `sync`. Thus, the segment of code above is divided into four serially ordered Cilk threads by the three parallel control constructs (two `spawn`s and a `sync`). Also, each spawned instance of `list_insert` comprises a thread that executes in parallel with some of the threads in the parent function instance. Figure 1.1 is a graphical view of the serial relationships among these threads.

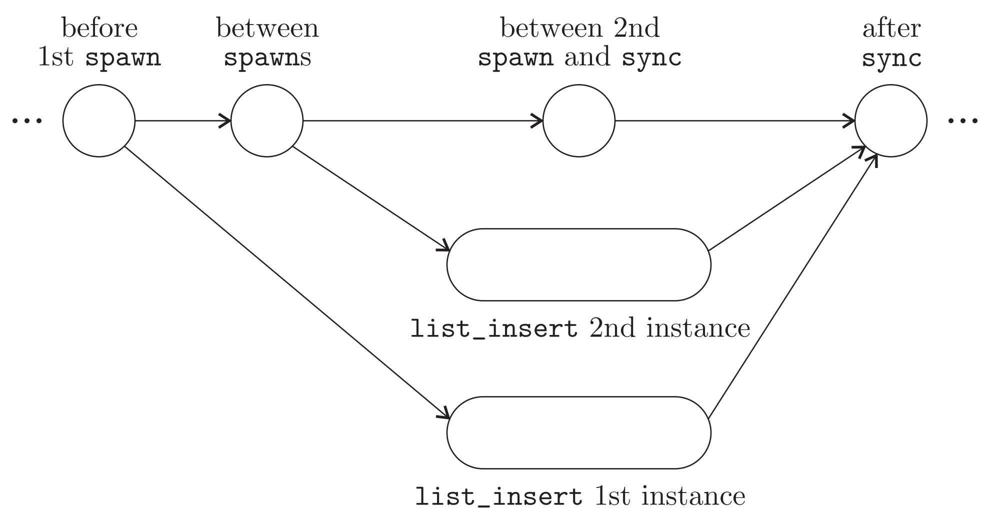

Figure 1.1: A graphical view of the threads in our code segment.

[1] We use "sync" instead of "synch" because this is the spelling of the keyword in Cilk.

Chapter 2 gives formal terminology and notation for these serial relationships, so that we can work with them mathematically. Henceforth in this thesis, when we talk about programs, functions, and threads, we shall mean Cilk programs, Cilk functions, and Cilk threads, respectively.

Now, a note about the programs and graphs that we study in this thesis is in order. First, since different inputs to a program can cause the program to behave differently in its execution, we consider detecting data races in programs that are run on fixed given inputs. This assumption is common in dynamic data-race detection. Furthermore, we assume that the control flow of a program stays the same even though its threads and transactions may be scheduled nondeterministically, because it has been proven by Robert H. B. Netzer and Barton P. Miller [32] that it is otherwise NP-hard to determine all possible executions of a parallel program. This assumption means that although a data-race detector can only build a graph from a particular execution trace, we may assume that such a graph is representative of all possible executions with respect to its control flow. Also, we are allowed to define and analyze graphs that represent programs in addition to graphs that represent executions.

Transactions versus Conventional Locks. The function `list_insert` is not correct as defined above. It contains a data race on the variable `head`. Between the two accesses to `head` made by a given instance of `list_insert`, the value of `head` could be changed by a concurrent thread (possibly another instance of `list_insert`).

The conventional solution for eliminating data-race errors is to use locks. (Throughout this thesis, we shall use the word "conventional" only to describe items related to programming using locks.) In the Cilk library, the functions for acquiring and releasing a lock are `Cilk_lock` and `Cilk_unlock`, respectively. If we use a lock `list_lock` to protect the shared linked list, then the following code is a correct version of `list_insert` using conventional locks.

```
cilk void list_insert( double new_data )
{
    Node *pnode = malloc( sizeof(Node) );
    pnode->data = process(new_data);
    Cilk_lock(list_lock);
    pnode->next = head;
    head = pnode;
    Cilk_unlock(list_lock);
}
```

The holding of `list_lock` during the two accesses to `head` guarantees that no other thread can concurrently access `head` while holding `list_lock`. Thus, if all parallel threads follow a common contract of only accessing `head` while holding `list_lock`, then the data race is eliminated.

For parallel programmers, keeping track of all the locks in use and deciding which locks should be held at any one time often becomes confusing and error-prone as the size of the program increases. This reasoning about concurrency may be simplified if a programmer uses a single global lock to protect all shared data, but programmers typically cannot just use one lock because of efficiency concerns. Locks are a preventive measure against data races, and as such, only one thread can hold a lock at any one time, while other concurrent threads that need the lock wait idly.

The concept of ***transactional memory*** [19] enables parallel programming under the logic of using a single global lock, but without a debilitating loss of efficiency. A ***transaction*** is a section of code that must be executed atomically within a single thread of computation, meaning that there can be no intervening shared-memory accesses by concurrent threads during the execution of the transaction. Transactional-memory support guarantees that the result of running a program looks as if all transactions happened atomically. For example, if we assume that the keyword `atomic` is used to indicate an atomic block to be implemented by a transaction, then the following code is a correct version of `list_insert` using transactions.

```
cilk void list_insert( double new_data )
{
    Node *pnode = malloc( sizeof(Node) );
    pnode->data = process(new_data);
    atomic {
        pnode->next = head;
        head = pnode;
    }
}
```

Transactional-memory support can be provided in the form of new machine instructions for defining transactions, which would minimally include `transaction_begin` and `transaction_end` for defining the beginning and end of a transaction. This support can be provided in hardware or simulated by software. Hardware transactional memory [19, 20] can be implemented on top of cache-consistency mechanisms already used in shared-memory computers. The rough idea is that each processor uses its own cache to store changes from transactional writes, and those changes are propagated to main memory when the transaction ends and successfully commits. Multiple processors can be attempting different transactions at the same time, and if there are no memory conflicts, then all the transactions should successfully commit. If two concurrent transactions experience a memory conflict (they both access the same shared-memory location, and at least one of the accesses is a write), then one of the transactions is aborted and retried at some later time. Software transactional memory [36, 18] simulates this mechanism in software, but the overhead per transaction is higher, making it less practical for high-performance applications.

From the programmer's point of view, using transactions is logically equivalent to using a single global lock, because all shared-memory accesses within a transaction happen without interruption from other concurrent transactions. This property eliminates many problems that arise when using multiple locks, such as priority inversion and deadlock. At the same time, transactional memory avoids the debilitating loss of efficiency that comes with using a single global lock, because the strategy of transactional memory is to greedily attempt to process multiple atomic sections concurrently, only aborting a transaction when an actual memory conflict occurs.

Transactions Everywhere. As its name suggests, ***transactions everywhere*** [26] is a methodology for parallel programming in which every instruction becomes part of a transaction. A working assumption is that hardware transactional memory provides low enough overhead per transaction to make this strategy a viable option.

Let us define some terminology for the transactions-everywhere methodology. The division points between transactions in the same thread are called ***cutpoints***. Cutpoints can be manually inserted by the programmer or automatically inserted by the compiler. To ***atomize*** a program is to divide a program up into transactions everywhere by inserting cutpoints, resulting in an ***atomized*** program, also known as an ***atomization*** of the program. An ***atomization strategy*** is a method that defines where to insert cutpoints into a program, such as that used by a compiler to automatically generate transactions everywhere.

We first go over the process when transactions everywhere are defined manually by the programmer. In this case, the starting point is the original program with no cutpoints. This starting point is in fact a valid atomization of the program itself, with each thread consisting of one transaction that spans the whole thread. Since each thread is an atomic section, we call this most conservative atomization the ***atomic-threads*** atomization of the original program.

From the atomic-threads starting point, the programmer looks to reduce potential inefficiency by cutting up large transactions, but only if doing so does not compromise correctness. For example, the function `list_insert` remains correct as long as the two accesses to `head` are in the same transaction. Since `list_insert` is a small function, the programmer may choose to leave it as one transaction. On the other hand, if the call to `process` takes a long time, then potential inefficiency exists because one instance of `list_insert` may make both of its accesses to `head` while another instance is calling `process`. This situation does not produce an error, but is nevertheless disallowed by the atomic-threads atomization. Thus, if the call to `process` takes a long time, the programmer may choose to add a cutpoint after this call to reduce inefficiency. If we assume that the keyword `cut` is used to indicate a cutpoint, then the following code is a more efficient form of `list_insert` using transactions everywhere.

```
cilk void list_insert( double new_data )
{
    Node *pnode = malloc( sizeof(Node) );
    pnode->data = process(new_data);
    cut;
    pnode->next = head;
    head = pnode;
}
```

To illustrate this atomization, we redraw Figure 1.1 with transactions instead of threads as vertices. This new graph is shown in Figure 1.2. We use dashed lines for the edges connecting consecutive transactions within the same thread.

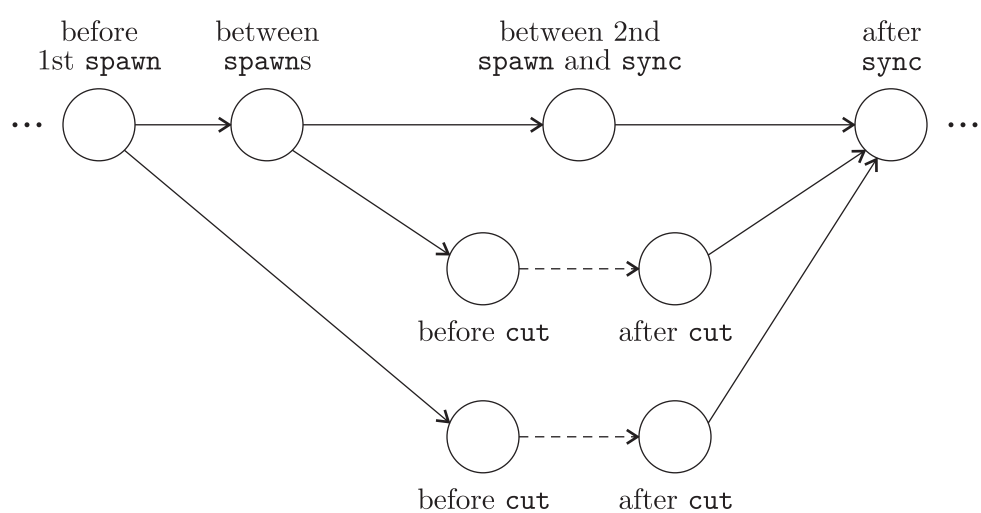

Figure 1.2: A revised version of Figure 1.1 with transactions as vertices.

Chapter 2 gives formal terminology and notation for these serial relationships, so that we can work with them mathematically.

We now consider the process when transactions everywhere are generated automatically by the compiler. In this case, the compiler uses an atomization strategy that consists of a set of heuristic rules describing where to insert cutpoints. For example, Clément Ménier has experimented with one such set of heuristic rules [28], which we shall call the *Ménier atomization strategy.* This strategy produces cutpoints at the following places:

- Parallel-programming constructs (`spawn`, `sync`).
- Calling and returning from a C function.
- The end of a loop iteration (`for`, `while`, `do` loops).
- Some other language constructs (`label`, `goto`, `break`, `continue`, `case`, `default`).

If the cutpoints inserted by the compiler produce a correct atomization (see Chapter 3 for a rigorous definition), then the programmer is spared from much thinking about concurrency issues. Thus, parallel programming becomes easier and more accessible from the average programmer's point of view. For example, applying the Ménier atomization strategy to `list_insert` produces cutpoints at the following places, marked by comments beginning with the symbol "▷".

```
cilk void list_insert( double new_data )
{
    ▷ cutpoint: C function call
    Node *pnode = malloc( sizeof(Node) );
    ▷ cutpoint: C function return

    ▷ cutpoint: C function call
    pnode->data = process(new_data);
    ▷ cutpoint: C function return

    pnode->next = head;
    head = pnode;
}
```

This atomization is correct because the two accesses to `head` appear in the same transaction.

In general, we expect that good heuristic rules should produce correct atomizations for most functions. In some cases, however, the granularity of the automatically generated transactions is too fine to preclude all data races. For example, if `list_insert` were rewritten so that the statement that sets `pnode->next` precedes the statement that sets `pnode->data`, then applying the Ménier atomization strategy produces cutpoints at the following places, once again marked by comments beginning with the symbol "▷".

```
cilk void list_insert( double new_data )
{
    ▷ cutpoint: C function call
    Node *pnode = malloc( sizeof(Node) );
    ▷ cutpoint: C function return

    pnode->next = head;

    ▷ cutpoint: C function call
    pnode->data = process(new_data);
    ▷ cutpoint: C function return

    head = pnode;
}
```

This atomization is incorrect because the call to `process` causes the two accesses to `head` to be placed in different transactions, thus resulting in a data-race error.

The above example shows that when the compiler automatically generates transactions everywhere, the atomization strategy used may be inaccurate for the particular situation. Likewise, when the programmer manually defines transactions everywhere, human error can lead to an incorrect atomization. In both cases, the programmer needs to use a data-race detector to find possible error locations, and then to adjust the transaction cutpoints if necessary. This thesis studies how to perform this task of detecting data races.

Chapter 2

Constraints in Transaction Scheduling

This chapter introduces some terminology for discussing the constraints that define the "schedules" (i.e. legal executions) of an atomized program. It also proves some preparatory lemmas about these constraints. This background material allows us in Chapters 3–4 to develop a definition for a data race in the transactions-everywhere setting, and to prove properties about data-race detection.

Section 2.1 lists the "thread constraints" and "transaction constraints" on program scheduling imposed by the serial control flow of the program, and formally defines a schedule of an atomized program. Section 2.2 gives a notion of equivalence for schedules, and defines "access constraints", which determine whether two schedules are equivalent. Section 2.3 then defines "access interactions" for atomized programs, which are the counterpart to access constraints for schedules, and "access assignments", which are the links between access constraints and access interactions.

Throughout our analysis of data-race detection, we shall use P to denote a Cilk program that has not yet been atomized, Q to denote an atomized program, and R to denote a schedule of an atomized program. Also, we shall use e to denote a thread and t to denote a transaction.

2.1 Serialization Constraints

This section formally defines the "thread constraints" and "transaction constraints" on the legal schedules of an atomized program, which are imposed by the serial control flow of the program. It also defines a "serialization graph" for visualizing these constraints. Finally, the formal definition of a schedule is derived in terms of these constraints.

Consider an atomized program Q. Since Q has a fixed control flow, we can view it as being composed of a set of n transactions, with certain serialization constraints between them. These constraints are determined by the serial order of transactions in the same thread, the serial order of threads in the same function instance, and the `spawn` and `sync` parallelization structure of the control flow. We now give names to these constraints.

A "thread constraint" of an atomized program Q is a constraint on schedules R of Q imposed by a serial relationship between two threads in the control flow of Q.

Definition 1. In an atomized program Q, a ***thread constraint*** exists from transaction t_1 to transaction t_2, denoted $\boldsymbol{t_1} \xrightarrow{\mathrm{E}} \boldsymbol{t_2}$, if

1. t_1 is the last transaction of a thread e_1 and t_2 is the first transaction of a thread e_2, and
2. the relationship between e_1 and e_2 is one of the following:
   a. e_1 immediately precedes e_2 in the same function instance, or
   b. e_1 directly precedes the spawn point of the function instance whose first thread is e_2, or

c. e_2 directly follows the sync point of the function instance whose final thread is e_1.

A "transaction constraint" of an atomized program Q is a constraint on schedules R of Q imposed by a serial relationship between two transactions in the control flow of Q.

Definition 2. In an atomized program Q, a ***transaction constraint*** exists from transaction t_1 to transaction t_2, denoted $\boldsymbol{t_1 \xrightarrow{\mathrm{T}} t_2}$, if t_1 immediately precedes t_2 in the same thread.

We can view these serialization constraints as edges in a graph with transactions as vertices.

Definition 3. The ***serialization graph*** of an atomized program Q is the graph $G = (V, E_{\mathrm{E}}, E_{\mathrm{T}})$, where V is the set of transactions, and the two types of edges are ***thread edges*** $E_{\mathrm{E}} = \{(t_1, t_2) \mid t_1 \xrightarrow{\mathrm{E}} t_2\}$ and ***transaction edges*** $E_{\mathrm{T}} = \{(t_1, t_2) \mid t_1 \xrightarrow{\mathrm{T}} t_2\}$.

Example. The following example program will be used throughout Chapters 2–3 to illustrate our definitions. The symbol "▷" denotes a comment.

```
int x1,     ▷ location ℓ1
    x2;     ▷ location ℓ2

cilk void fun1()
{
    int a;

    ▷ t1
    ⋮
    x1 = a;     ▷ write ℓ1
    ⋮

    ▷ cutpoint
    ▷ t2
    ⋮

    ▷ cutpoint
    ▷ t3
    ⋮
    a = x2;     ▷ read ℓ2
    ⋮
}

cilk void fun2()
{
    int a;

    ▷ t6
    ⋮
    x2 = a;     ▷ write ℓ2
    ⋮
}
```

```
cilk int main()
{
    int a;

    ▷ t0
    ⋮

    spawn fun1();
    ▷ t4
    ⋮
    a = x1;     ▷ read ℓ1
    ⋮

    ▷ cutpoint
    ▷ t5
    ⋮
    a = x2;     ▷ read ℓ2
    ⋮

    spawn fun2();
    ▷ t7
    ⋮

    ▷ cutpoint
    ▷ t8
    ⋮
    x1 = a;     ▷ write ℓ1
    ⋮

    sync;
    ▷ t9
    ⋮
    return 0;
}
```

This program uses two shared variables `x1` and `x2`, whose values are stored in shared-memory locations ℓ_1 and ℓ_2, respectively. The `main` function spawns two function instances (the first is an instance of `fun1`, and the second is an instance of `fun2`), and then syncs them at a later point. The cutpoints in this program are marked by comments, because they could very well be automatically generated. The transactions are $t_0, \ldots, t_9$, also marked by comments.

Figure 2.1 shows the serialization graph of our example program. The solid lines are thread edges and the dashed lines are transaction edges. The vertices of this graph are the transactions $t_0, \ldots, t_9$. This diagram only labels the vertices with their transaction subscripts, for ease of reading. We shall maintain this practice throughout the remainder of this thesis.

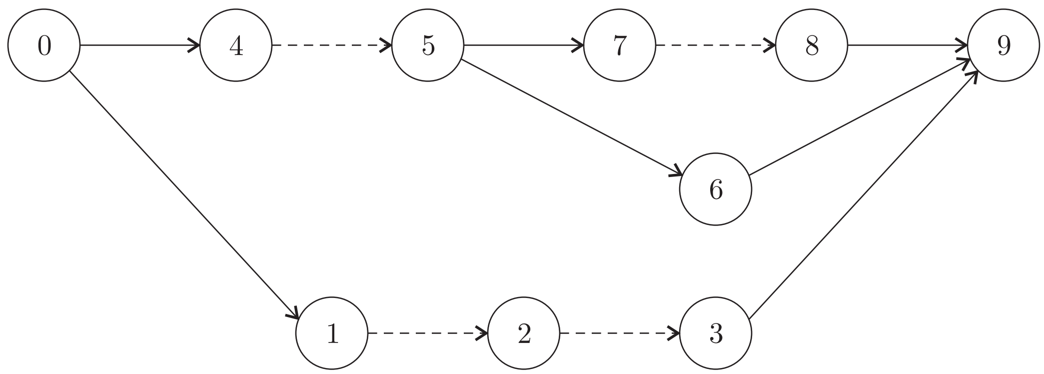

Figure 2.1: The serialization graph of our example program.

Our example program generates this runtime graph as follows. For now, ignore the accesses to shared-memory locations. The three function instances are represented by the three rows of vertices (the middle row has only one transaction). The top row is the program's `main` function. It first spawns function `fun1` consisting of transactions t_1, t_2, and t_3, and then spawns function `fun2` consisting of the single transaction t_6. The spawned functions are synced before transaction t_9. $\Diamond$

Now that we have specified the constraints in scheduling the transactions of Q, we can formalize our understanding of a schedule.

Definition 4. A ***schedule*** R of an atomized program Q is a total (linear) ordering $\prec_{\boldsymbol{R}}$ on the transactions of Q that satisfies all the thread and transaction constraints of Q. That is, for any two transactions t_1 and t_2, if $t_1 \stackrel{\mathrm{E}}{\to} t_2$ or $t_1 \stackrel{\mathrm{T}}{\to} t_2$, then $t_1 \prec_R t_2$.

Example. Figure 2.2 shows one possible schedule R_{ex} of our example program from Figure 2.1. The diagram is drawn with time moving from left to right (earlier transactions appear to the left of later transactions). Once again, this diagram labels the vertices with their transaction subscripts. The ordering of the transactions in this schedule is $t_0 \prec_{R_{ex}} t_1 \prec_{R_{ex}} t_4 \prec_{R_{ex}} t_2 \prec_{R_{ex}} t_5 \prec_{R_{ex}} t_7 \prec_{R_{ex}} t_8 \prec_{R_{ex}} t_6 \prec_{R_{ex}} t_3 \prec_{R_{ex}} t_9$. $\Diamond$

2.2 Access Constraints

This section addresses the question of which schedules of an atomized program are equivalent in the sense of exhibiting the same behavior. We first define a notion of equivalence for schedules derived from the same program. This definition suggests a new type of constraint, called an

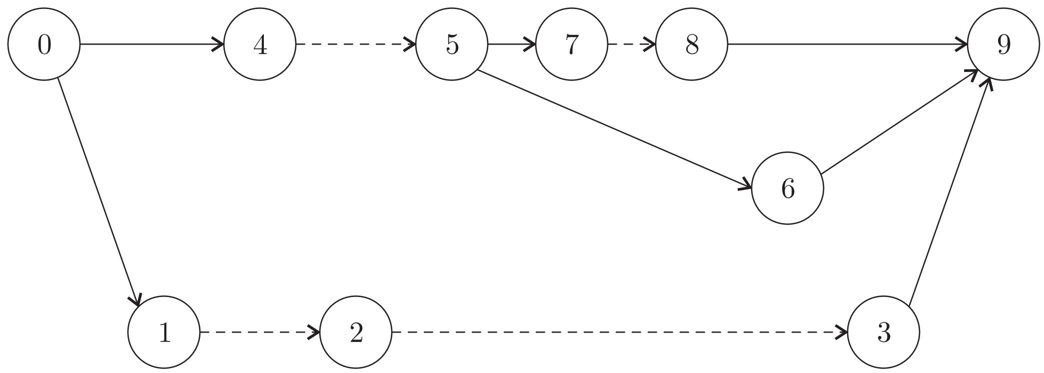

Figure 2.2: One possible schedule of our example program.

"access constraint". We prove that access constraints accurately determine when two schedules are equivalent. Finally, we define a "schedule graph" for visualizing all the constraints in a schedule.

Our first step is to define a precise notion of equivalence.

Definition 5. Let Q and Q' be (possibly identical) atomizations of a program P, and let R and R' be schedules of Q and Q', respectively. We say that R and R' are ***equivalent*** if for each shared-memory location ℓ,

1. corresponding writes of ℓ in R and R' occur in the same order, and
2. corresponding reads of ℓ in R and R' receive values written by corresponding writes.

We should note two subtleties of Definition 5. One point is that it is not enough for corresponding reads to receive the same value; they must in fact receive the value written by the same corresponding writes. The other point is that whole sets of accesses consisting of a write and all its subsequent reads cannot be reordered. These subtleties exist because we want a definition of equivalence that facilitates dynamic data-race detection.

The following theorem shows that our notion of equivalence gives strong guarantees for identical behavior between two equivalent schedules.

Theorem 1. *Let Q and Q' be (possibly identical) atomizations of a program P, and let R and R' be schedules of Q and Q', respectively. If R and R' are equivalent, then for each instruction I in P, the following invariants hold: Before and after the corresponding executions of I in R and R',*

1. *all local variables whose scopes include I have the same values in R and R', and*
2. *all shared-memory locations accessed by I (if any) have the same values in R and R'.*

Proof. Let $\langle I_0, \ldots, I_{k-1} \rangle$ be the instructions of P in the order that they occur in R. We shall prove the invariants by strong induction on this ordering of the instructions.

First, observe that if the invariants are met before an instruction I executes, then they continue to hold true after I executes. Consequently, we only need to concentrate on proving that the invariants hold before an instruction executes.

For the base case, we note that the first instruction in any schedule must necessarily be the first instruction of the `main` function instance of P. Therefore, I_0 is the first instruction in both R and R'. Before I_0 executes, all the local variables and shared-memory locations are identical, since no instructions have been executed yet.

For the inductive step, assume that the invariants hold for all instructions before some instruction I_i in R. The value of a local variable created before I_i was written by the same previous instruction I that serially precedes I_i in both schedules, because the serial control flow is determined by P and is identical in both schedules. By the inductive hypothesis, we know that the value of this local variable is the same in R and R' after I executes.

Now, consider the value of a shared-memory location ℓ accessed by I_i. If I_i writes ℓ, then condition 1 in Definition 5 dictates that the previous instruction I to write ℓ must be the same in both schedules. Similarly, if I_i reads ℓ, then condition 2 in Definition 5 guarantees that the previous instruction I to write ℓ is the same in both schedules. Applying the inductive hypothesis to I tells us that the value in ℓ was the same in both R and R' after I executed. ☐

We now introduce a third type of constraint called an "access constraint", which is imposed by parallel accesses to shared memory. Access constraints determine how a schedule R may be reordered into an equivalent schedule R', or alternatively, whether two schedules R and R' are equivalent. Unlike thread and transaction constraints, access constraints are defined for a particular schedule R as opposed to an atomized program Q.

Definition 6. In a schedule R of an atomized program Q, an ***access constraint*** exists from transaction t_1 to transaction t_2, denoted $\boldsymbol{t_1 \xrightarrow{\mathrm{A}} t_2}$, if

1. t_1 and t_2 are in parallel in the control flow of Q,
2. there exists a shared-memory location ℓ such that t_1 and t_2 both access ℓ, and at least one of them writes ℓ, and
3. t_1 appears before t_2 in R (i.e. $t_1 \prec_R t_2$).

We can think about the constraints on how a schedule R can be reordered in terms of a graph with transactions as vertices and constraints as edges.

Definition 7. The ***schedule graph*** of a schedule R of an atomized program Q is the graph $G = (V, E_{\mathrm{E}}, E_{\mathrm{T}}, E_{\mathrm{A}})$, where V is the set of transactions, and the three types of edges are thread edges $E_{\mathrm{E}} = \{(t_1, t_2) \mid t_1 \xrightarrow{\mathrm{E}} t_2\}$, transaction edges $E_{\mathrm{T}} = \{(t_1, t_2) \mid t_1 \xrightarrow{\mathrm{T}} t_2\}$, and ***access constraint edges*** $E_{\mathrm{A}} = \{(t_1, t_2) \mid t_1 \xrightarrow{\mathrm{A}} t_2\}$.

Example. Figure 2.3 shows the schedule graph of our example schedule from Figure 2.2. The access constraint edges are drawn with dotted lines.

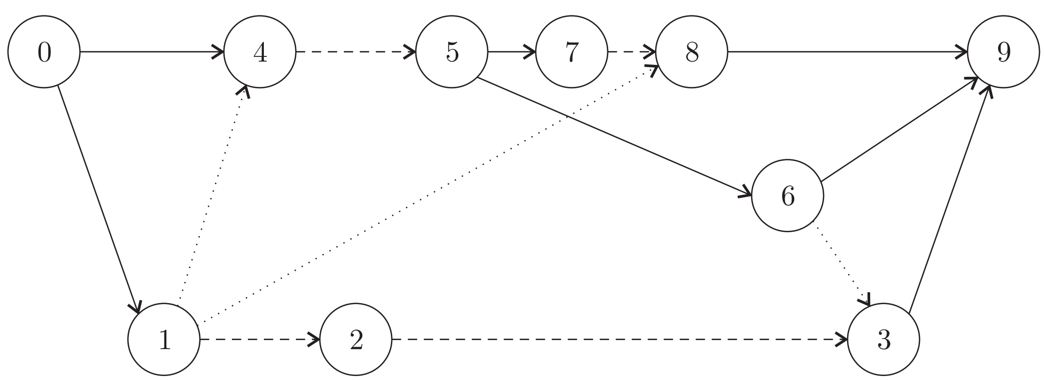

Figure 2.3: The schedule graph of our example schedule from Figure 2.2.

The shared-memory accesses in our example program are as follows:

- location ℓ_1 is written by transactions t_1 and t_8, and read by transaction t_4;
- location ℓ_2 is written by transaction t_6, and read by transactions t_3 and t_5.

The accesses to ℓ_1 generate the two access constraints $t_1 \xrightarrow{\text{A}} t_4$ and $t_1 \xrightarrow{\text{A}} t_8$. There is no access constraint from t_4 to t_8 even though they both access ℓ_1, because they are not in parallel in the control flow of our example program. The multiple accesses to ℓ_2 only generate the single access constraint $t_6 \xrightarrow{\text{A}} t_3$. There is no access constraint from t_5 to t_6 even though they both access ℓ_2, because they are not in parallel, and there is no access constraint from t_5 to t_3 even though they both access ℓ_2, because neither of them writes ℓ_2. $\Diamond$

The following lemma proves that access constraints accurately determine equivalence.

Lemma 2. *Two schedules R and R' of an atomized program Q are equivalent if and only if they have the same set of access constraints.*

Proof. Forward direction. Let $t_1 \xrightarrow{\text{A}} t_2$ be an arbitrary access constraint of R. We shall show that R' also has this access constraint. Conditions 1 and 2 in Definition 6 do not depend on the particular schedule, so they apply to R' as well. All that remains to be shown is condition 3, which requires that $t_1 \prec_{R'} t_2$. We consider three cases.

Case 1: t_1 and t_2 both write ℓ. Since t_1 and t_2 both write ℓ, condition 1 in Definition 5 dictates that if $t_1 \prec_R t_2$, then it must be that $t_1 \prec_{R'} t_2$ as well.

Case 2: t_1 writes ℓ and t_2 only reads ℓ. Let t_3 be the last transaction before t_2 to write ℓ. It must be the same transaction in both R and R' because they are equivalent. If $t_1 = t_3$, then certainly $t_1 \prec_{R'} t_2$. If $t_1 \neq t_3$, then $t_1 \prec_R t_3$, since t_1 also writes ℓ but is not the last transaction to do so before t_2. Thus, the ordering of the three transactions in R is $t_1 \prec_R t_3 \prec_R t_2$. Condition 1 in Definition 5 dictates that $t_1 \prec_{R'} t_3$ as well, and condition 2 dictates that $t_3 \prec_{R'} t_2$ as well, so we conclude that $t_1 \prec_{R'} t_2$.

Case 3: t_1 only reads ℓ and t_2 writes ℓ. Let t_3 be the last transaction before t_1 that writes ℓ. It must be the same transaction in both R and R' because they are equivalent. Thus, the ordering of the three transactions in R is $t_3 \prec_R t_1 \prec_R t_2$. Now, if $t_2 \prec_{R'} t_1$, then we must also have $t_2 \prec_{R'} t_3$, because t_3 is the last transaction to write ℓ before t_1. But, having $t_2 \prec_{R'} t_3$ contradicts condition 1 in Definition 5, so we conclude that $t_1 \prec_{R'} t_2$.

Since the preceding arguments apply to any access constraint $t_1 \xrightarrow{\text{A}} t_2$ of R, we see that R' has all the access constraints of R. By a symmetrical argument, R also has all the access constraints of R'. Thus, R and R' must have the same set of access constraints.

Backward direction. We shall prove the contrapositive, which says that if R and R' are not equivalent, then they do not have the same set of access constraints.

If R and R' are not equivalent due to condition 1 in Definition 5 not being satisfied, then let t_1 and t_2 be transactions that both write a shared-memory location ℓ, and such that $t_1 \prec_R t_2$ and $t_2 \prec_{R'} t_1$. Then, the access constraint $t_1 \xrightarrow{\text{A}} t_2$ exists for R but not for R', so R and R' do not have the same set of access constraints.

If R and R' are not equivalent due to condition 2 in Definition 5 not being satisfied, then let t_1 be a transaction that reads a shared-memory location ℓ, but for which the previous transaction to write ℓ was t_2 in R and t_3 in R'. We consider three possibilities. First, if $t_1 \prec_R t_3$, then the access constraint $t_1 \xrightarrow{\text{A}} t_3$ exists for R but not for R'. Second, if $t_1 \prec_{R'} t_2$, then the access constraint $t_1 \xrightarrow{\text{A}} t_2$ exists for R' but not for R. Finally, if both t_2 and t_3 appear before t_1 in both R and R', then the ordering of the three transactions in R must be $t_3 \prec_R t_2 \prec_R t_1$ (because t_2 is the last transaction to write ℓ before t_1), while the ordering in R' must be $t_2 \prec_{R'} t_3 \prec_{R'} t_1$. But then, $t_3 \xrightarrow{\text{A}} t_2$ is an access constraint that exists for R but not for R'. Thus, in all cases, R and R' do not have the same set of access constraints. $\square$

2.3 Access Interactions and Access Assignments

Ultimately, we wish to prove properties about atomized programs as a whole, and not just particular schedules. In preparation for doing so, this section defines "access interactions", which are the counterpart for an atomized program Q to what access constraints are for a schedule of Q, and the "interaction graph", which is the counterpart to the schedule graph. Finally, this section answers the question of which subsets of all possible access constraints of an atomized program Q can be extended to a scheduling of Q. In doing so, it introduces "access assignments", which represent the link between access interactions and access constraints.

An "access interaction" of an atomized program Q is a bidirectional (symmetric) relation between two transactions of Q that share an access constraint in any schedule.

Definition 8. In an atomized program Q, an ***access interaction*** exists between transaction t_1 and transaction t_2, denoted $\boldsymbol{t_1 \overset{A}{\leftrightarrow} t_2}$ (or $\boldsymbol{t_2 \overset{A}{\leftrightarrow} t_1}$), if

1. t_1 and t_2 are in parallel in the control flow of Q, and
2. there exists a shared-memory location ℓ such that t_1 and t_2 both access ℓ, and at least one of them writes ℓ.

Conditions 1 and 2 in Definition 8 are the same as conditions 1 and 2 in Definition 6. If there is an access interaction between two transactions t_1 and t_2, then in any schedule R of Q, there is an access constraint of either $t_1 \overset{A}{\to} t_2$ or $t_2 \overset{A}{\to} t_1$, depending on how t_1 and t_2 are ordered with respect to each other in R.

We can view all the serialization constraints and access interactions of Q in one graph.

Definition 9. The ***interaction graph*** of an atomized program Q is the graph $G = (V, E_{\mathrm{E}}, E_{\mathrm{T}}, E_{\mathrm{A}})$, where V is the set of transactions of Q, and the three types of edges are thread edges $E_{\mathrm{E}} = \{(t_1, t_2) \mid t_1 \overset{E}{\to} t_2\}$, transaction edges $E_{\mathrm{T}} = \{(t_1, t_2) \mid t_1 \overset{T}{\to} t_2\}$, and ***access interaction edges*** $E_{\mathrm{A}} = \{\{t_1, t_2\} \mid t_1 \overset{A}{\leftrightarrow} t_2\}$.

In this thesis, we shall sometimes refer simply to an "access edge" when it is clear by context whether we mean an access constraint edge or an access interaction edge.

Example. Figure 2.4 shows the interaction graph of our example program.

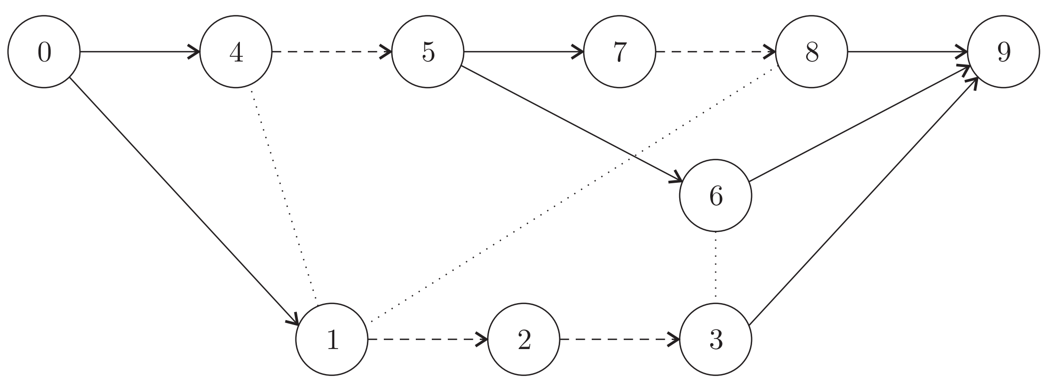

Figure 2.4: The interaction graph of our example program.

The access interaction edges $t_1 \overset{A}{\leftrightarrow} t_4$, $t_1 \overset{A}{\leftrightarrow} t_8$, and $t_3 \overset{A}{\leftrightarrow} t_6$ are drawn as dotted lines without arrow heads. The reader should refer to the text following Figure 2.3 for a discussion of the shared-memory accesses in our example program, and why certain pairs generate access edges.

Comparing this diagram with Figure 2.3, even though the transactions are named and positioned differently, we can see that the access constraints in Figure 2.3 are simply the access interactions in this diagram with directions selected based on the particular schedule. $\Diamond$

Although each bidirectional access interaction of Q corresponds to an access constraint in one direction or the other in a schedule R, not every assignment of directions to access interactions (thus turning them into access constraints) produces a legal schedule. The final question we address in this section is that of which subsets of all the possible access constraints can be extended to a schedule.

Definition 10. An ***access assignment*** A of an atomized program Q is a subset of all the possible access constraints, as indicated by the access interactions of Q. For example, if Q has an access interaction $t_1 \overset{\mathrm{A}}{\leftrightarrow} t_2$, then A may contain neither, either, or both of the access constraints $t_1 \xrightarrow{\mathrm{A}} t_2$ and $t_2 \xrightarrow{\mathrm{A}} t_1$. We say that an access assignment A is ***realizable*** if there exists a schedule R of Q such that A is a subset of the access constraints of R. In such a case, we say that R ***realizes*** A.

We can view an access assignment A of an atomized program Q, along with the serialization constraints of Q, as a graph.

Definition 11. The ***assignment graph*** of an access assignment A of an atomized program Q is the graph $G = (V, E_{\mathrm{E}}, E_{\mathrm{T}}, A)$, where V is the set of transactions, and the three types of edges are thread edges $E_{\mathrm{E}} = \{(t_1, t_2) \mid t_1 \xrightarrow{\mathrm{E}} t_2\}$, transaction edges $E_{\mathrm{T}} = \{(t_1, t_2) \mid t_1 \xrightarrow{\mathrm{T}} t_2\}$, and access interaction edges from A.

Example. According to Figure 2.4, the set of all possible access constraints for our example program is $\{(t_1, t_4), (t_4, t_1), (t_1, t_8), (t_8, t_1), (t_3, t_6), (t_6, t_3)\}$. Any subset of this set constitutes an access assignment of our example program. One particular access assignment is $\{(t_1, t_4), (t_6, t_3)\}$. Its assignment graph is shown in Figure 2.5. $\Diamond$

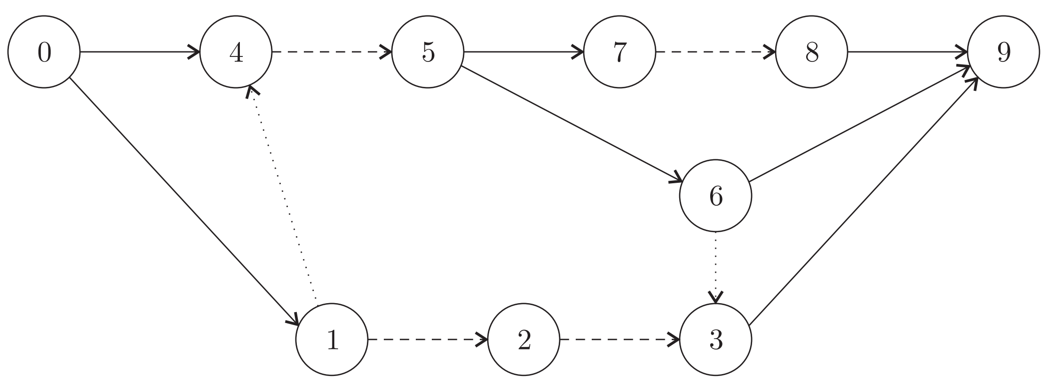

Figure 2.5: The assignment graph of one particular access assignment.

Now we may answer the question of which access assignments are realizable.

Lemma 3. *An access assignment A of an atomized program Q is realizable if and only if its assignment graph G does not contain any cycles.*

Proof. Forward direction. We shall prove the contrapositive, which says that if G contains a cycle, then A is not realizable. Let the cycle in G be $t_0 \xrightarrow{?} t_1 \xrightarrow{?} \cdots \xrightarrow{?} t_{k-1} \xrightarrow{?} t_0$, where each pair of consecutive transactions has a constraint between them of one of the three types (thread, transaction, access). Since G is a subgraph of the schedule graph of any schedule R that realizes A, any such schedule R must also satisfy all the constraints in the cycle. In particular, we must have $t_0 \prec_R t_1 \prec_R \cdots \prec_R t_{k-1} \prec_R t_0$, which is impossible since t_0 cannot precede itself in R.

Backward direction. If G does not contain any cycles, then we can construct a schedule R that realizes A using a topological-sort algorithm (for example, see [6], pages 549–551). $\square$

Chapter 3

The Definition of a Data Race

This chapter defines a data race in the transactions-everywhere environment and proves conditions for the existence of a data race. Section 3.1 first discusses the difference between data-race detection in the transactions-everywhere setting and in the conventional locks setting. This discussion leads us to the definition of a data race in the transactions-everywhere methodology. The definition is based on the basic assumption of "correct parallelization", which informally says that the target program's spawn-and-sync structure is correct. This assumption is made in the same spirit as the assumption that underlies the conventional definition of a data race. Section 3.2 proceeds to determine necessary and sufficient conditions (collectively called a "race assignment") for an atomized program to contain a data race. These conditions are easier to work with than the primitive definition of a data race, and they find their application in Chapters 4–5.

3.1 Discussion and Definition

This section establishes the definition of a data race in the transactions-everywhere methodology. We begin by discussing why the conventional definition of a data race does not make sense when using transactions everywhere, especially when the transactions are automatically generated. We then explore the primary assumption behind the conventional definition, and develop an assumption, called "correct parallelization", that can be made in the same spirit (albeit weaker) when using transactions everywhere. It turns out that this assumption is difficult to apply directly, so this section uses it to prove that the atomic-threads atomization of the target program is guaranteed to be correct. This last fact serves as the basis for the definition of a data race.

Detecting data races in the transactions-everywhere setting is much different than the corresponding task in the conventional locks setting. Conventional data-race detectors find places in a parallel program where two concurrent threads access the same memory location without holding a common lock, and at least one of the accesses is a write. If the same algorithm were used on an atomized program, it would report no data races. The reason is that using transactions is logically equivalent to using a single global lock, and using transactions everywhere is analogous to executing all instructions, in particular all memory accesses, while holding the common global lock.

The absence of conventional data races does not ensure that an atomization is guaranteed to be accurate. From the point of view of data-race detection, the key difference between locks and transactions everywhere is that locks carry the programmer's certification of correctness. Locks do not actually eliminate data races, because the locked sections can still be scheduled in different ways that lead to different answers. Rather, locks carry the assertion that the programmer has thought about all the possible orderings of the sections protected by the same lock, and has concluded that there is no error, even though the possibility for nondeterminism exists.

Since transactions everywhere do not carry any certification by the programmer (especially when they are automatically generated), it would seem that all potential data-race errors must still be reported, leading to the same amount of thinking required of the programmer as when using locks. This reasoning is incorrect, however. If we make just one basic assumption about the programmer's understanding of the program, then we can avoid reporting many data races, because they most likely do not lead to errors.

The basic assumption we make is that the program has "correct parallelization", which informally says that the program's spawn-and-sync structure, as the programmer has written it, does not need to be changed to make the program correct. The only thing that may need to be adjusted is the atomization.

Definition 12. We say a program has ***correct parallelization*** if there exists some atomization of the program such that all of its possible schedules exhibit correct behavior, as determined by the programmer's intent.

Because our definition is based on the programmer's intent, it does not require the atomized program to behave deterministically, either internally or externally (see [33] for definitions of these terms). However, we do restrict the meaning of correct behavior to eliminate efficiency concerns due to differences in atomization or schedule.

The assumption that a program has correct parallelization is made in the same spirit as the assumption made by conventional data-race detectors that concurrent memory accesses with a common lock are safe. In the locks environment, the assumption is that the programmer thought about correctness when he or she used the functions `Cilk_lock` and `Cilk_unlock`. In the transactions-everywhere environment, the assumption is that the programmer thought about correctness when he or she used the keywords `spawn` and `sync`.

The assumption of correct parallelization is difficult to use directly, because while it promises the existence of a correct atomization, it gives us no information about what that atomization may be. In order to check whether a given atomization is correct, we wish to be able to compare it against something concrete that we know to be correct. Such a standard is provided by the following theorem.

Theorem 4. *If a program P has correct parallelization, then the atomic-threads atomization Q^* of P is guaranteed to be correct.*

Proof. Since P has correct parallelization, there exists some atomized version Q of P such that all schedules of Q exhibit correct behavior. It suffices for us to show that every schedule of Q^* is equivalent to some schedule of Q, because then the possible behaviors of Q^* would be a subset of the possible behaviors of Q, all of which we know to be correct.

We map a schedule R^* of Q^* into an equivalent schedule R of Q simply by dividing each thread in R^* into its constituent transactions in Q, while keeping all the instructions in their original order. Since R^* and R represent executions of the same instructions in the same order, they are equivalent. We also need to check that R is a legal schedule of Q. Since R^* is a legal schedule of Q^*, it satisfies all the thread constraints of Q^*, which are the same as those of Q, so R satisfies the thread constraints of Q. Also, since each thread of R^* is divided into transactions of Q without changing the order of those constituent transactions in Q, we see that R also satisfies all the transaction constraints of Q. □

Theorem 4 gives us an important tool with which to build a race detector. Given a program with correct parallelization, we always know of one concrete atomization that is guaranteed to be correct (albeit possibly inefficient). Our strategy shall be to have the data-race detector check whether all the possible behaviors of a given atomization of a program are also possible behaviors of the atomic-threads atomization of the same program.

Definition 13. In the transactions-everywhere methodology, let P be a program with correct parallelization, let Q be an atomized program derived from P, and let Q^* be the atomic-threads atomization of P. We say Q contains a ***data race*** if there exists a schedule of Q that is not equivalent to any schedule of Q^*.

Just as data races in the conventional locks setting are not necessarily errors, data races in the transactions-everywhere setting are also not always errors. Instead, we can consider Definition 13 to be a good heuristic for when to report a possible data-race error.

3.2 Race Assignments

This section identifies conditions under which an atomized program contains a data race, in hopes of developing efficient algorithms to search for these conditions. We first determine conditions (collectively named a "thread cycle") for when a particular schedule of an atomized program causes a data race. Then, we extend this result to find conditions (collectively named a "race assignment") for when an atomized program contains a data race.

Our first step is to determine conditions for when a particular schedule causes a data race. Consider a schedule R of an atomization Q of a program P, whose atomic-threads atomization is Q^*. Since we want to know whether R is equivalent to some schedule of Q^*, we can think of the access constraints of R as inducing an access assignment A on Q^*. Remember that the transactions of Q^* are just the threads of P.

Definition 14. The ***induced access assignment*** by R on Q^* is the access assignment A of Q^* that contains an access constraint $e_1 \xrightarrow{\text{A}} e_2$ whenever there exists an access constraint $t_1 \xrightarrow{\text{A}} t_2$ in R from some transaction t_1 in thread e_1 to some transaction t_2 in thread e_2.

Example. Recall our example schedule from the Figure 2.2 and its schedule graph from Figure 2.3. The induced access assignment by this schedule on the atomic-threads version of our example program is shown in Figure 3.1.

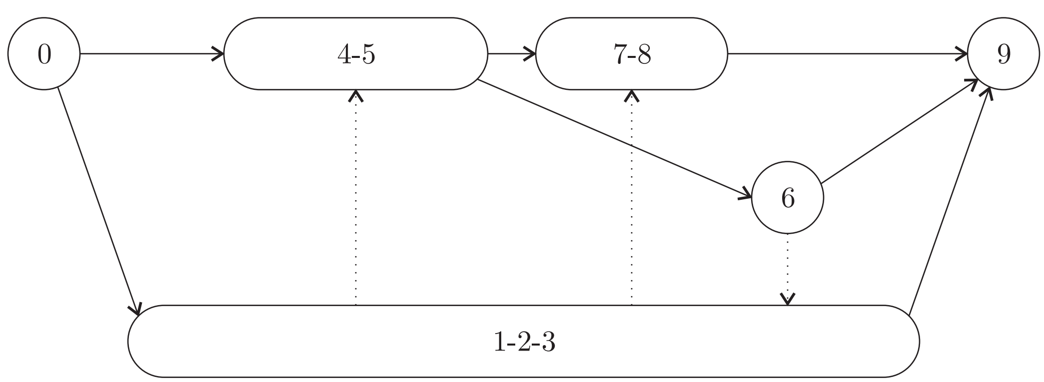

Figure 3.1: The induced access assignment by our example schedule from Figure 2.2.

The transactions of the atomic-threads atomization are simply the threads of the program. Each access constraint edge between two transactions in Figure 2.3 translates into an access constraint edge between those transactions' respective threads in Figure 3.1. Although our example does not show it, the reader should note that if a schedule graph were to contain multiple access constraint edges from transactions belonging to one thread to transactions belonging to another thread, then all those access constraint edges would collapse into one access constraint edge between the two threads in the assignment graph of the induced access assignment. $\Diamond$

The following lemma ties the concept of the induced access assignment to our problem. In the following proof, we extend our "$\prec_R$" notation for transaction precedence in a schedule R in the natural way to denote instruction precedence in R as well.

Lemma 5. *Let R be a schedule of an atomization Q of a program P, whose atomic-threads atomization is Q^*. Then, R is equivalent to some schedule of Q^* if and only if the induced access assignment A by R on Q^* is realizable.*

Proof. Forward direction. Let R^* be a schedule of Q^* that is equivalent to R. We shall prove that R^* realizes A. Let $e_1 \xrightarrow{\text{A}} e_2$ be any access constraint in A. We need to show that $e_1 \prec_{R^*} e_2$, so that R^* also has $e_1 \xrightarrow{\text{A}} e_2$ as an access constraint. Definition 14 implies that an access constraint $t_1 \xrightarrow{\text{A}} t_2$ exists in R from some transaction t_1 in thread e_1 to some transaction t_2 in thread e_2. Then, Definition 6 implies that t_1 and t_2 both access a shared-memory location ℓ, and at least one of them writes ℓ. We consider three cases.

Case 1: t_1 and t_2 both write ℓ. Let I_1 and I_2 be instructions in t_1 and t_2, respectively, that write ℓ. We know that $I_1 \prec_R I_2$. Now, if $e_2 \prec_{R^*} e_1$, then we would have $I_2 \prec_{R^*} I_1$, causing R and R^* not to be equivalent, which is a contradiction. Thus, we must have $e_1 \prec_{R^*} e_2$.

Case 2: t_1 writes ℓ and t_2 only reads ℓ. Let I_1 be an instruction in t_1 that writes ℓ, and let I_2 be an instruction in t_2 that reads ℓ. Furthermore, let I be the last instruction before I_2 that writes ℓ. We know that $I_1 \prec_R I$ because I_1 also writes ℓ but is not the last instruction before I_2 to do so. Now, if $e_2 \prec_{R^*} e_1$, then we would have $I \prec_{R^*} I_1$, causing R and R^* not to be equivalent, which is a contradiction. Thus, we must have $e_1 \prec_{R^*} e_2$.

Case 3: t_1 only reads ℓ and t_2 writes ℓ. Let I_1 be an instruction in t_1 that reads ℓ, and let I_2 be an instruction in t_2 that writes ℓ. Furthermore, let I be the last instruction before I_1 that writes ℓ. We know that $I \prec_R I_2$ because $I \prec_R I_1$ and $I_1 \prec_R I_2$. Now, if $e_2 \prec_{R^*} e_1$, then we would have $I_2 \prec_{R^*} I$, causing R and R^* not to be equivalent, which is a contradiction. Thus, we must have $e_1 \prec_{R^*} e_2$.

Since the preceding arguments apply to any access constraint $t_1 \xrightarrow{\text{A}} t_2$ in A, we conclude that R^* has all the access constraints of A, which proves that R^* indeed realizes A.

Backward direction. Let R^* be a schedule of Q^* that realizes A. We shall prove that R is equivalent to R^*. First, construct a schedule R' of Q by dividing up the threads of R^* into their constituent transactions in Q, while maintaining the same order for all the instructions. Then, R' satisfies the thread constraints of Q because they are the same as those for Q^*, and R' satisfies the transaction constraints of Q because the threads in R^* are divided up without any reordering of the transactions. Thus, R' is a legal schedule of Q.

Now, we show that R is equivalent to R', which in turn is equivalent to R^*. That R' and R^* are equivalent follows from the fact that both represent executions of the same instructions in the same order. To show that R and R' are equivalent, consider any access constraint $t_1 \xrightarrow{\text{A}} t_2$ of R. Let t_1 belong to thread e_1 and t_2 belong to thread e_2. By Definition 14, R^* must have $e_1 \xrightarrow{\text{A}} e_2$ as an access constraint, which means that we must have $e_1 \prec_{R^*} e_2$. Then, because R' is derived from R^* without any reordering of instructions, it must be true that in R', all the transactions of e_1 appear consecutively before all the transactions of e_2 appear consecutively. In particular, $t_1 \prec_{R'} t_2$, so that R' also has $t_1 \xrightarrow{\text{A}} t_2$ as an access constraint. Since this reasoning applies to any access constraint of R, we see that R' must have all the access constraints of R. In addition, R' cannot have any other access constraints, because the number of access constraints for a schedule of Q is constant (equal to the number of access interactions of Q). Thus, R and R' have the same set of access constraints, and by Lemma 2, they must therefore be equivalent. □

The following definition and lemma restate the result in Lemma 5 without reference to the induced access assignment.

Definition 15. A ***thread cycle*** in a schedule graph, interaction graph, or assignment graph is a series of pairs of transactions $\langle (t_0, t'_1), (t_1, t'_2), \ldots, (t_{k-2}, t'_{k-1}), (t_{k-1}, t'_0) \rangle$ such that

1. for each $i \in \{0, \ldots, k-1\}$, there is a thread or access edge from t_i to t'_{i+1}, where index arithmetic is performed modulo k, and
2. for each $i \in \{0, \ldots, k-1\}$, t'_i and t_i belong to the same thread (and may be the same transaction).

Example. Recall the example access assignment $\{(t_1, t_4), (t_6, t_3)\}$ of our example program, and its assignment graph from Figure 2.5. This assignment graph contains the thread cycle $\langle (t_1, t_4), (t_5, t_6), (t_6, t_3) \rangle$, which is shown in Figure 3.2. Note that each pair of transactions in the thread cycle represents either a thread or access constraint edge ($t_1 \xrightarrow{\text{A}} t_4$, $t_5 \xrightarrow{\text{E}} t_6$, $t_6 \xrightarrow{\text{A}} t_3$). Also, note that the second transaction of one pair and the first transaction of the next pair are in the same thread (t_4 and t_5, t_6 and t_6, t_3 and t_1). ◊

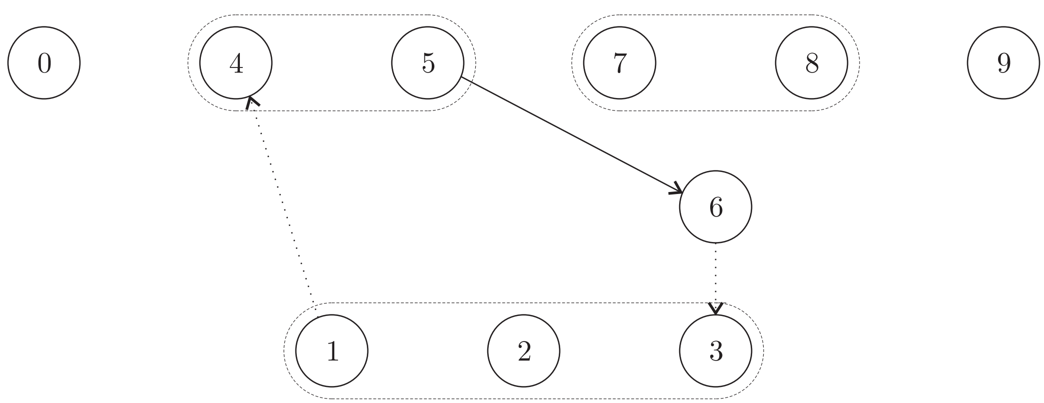

Figure 3.2: A thread cycle in the assignment graph from Figure 2.5.

Lemma 6. *Let G be the schedule graph of a schedule R of an atomization Q of a program P. Then, R is equivalent to some schedule of the atomic-threads atomization of P if and only if G does not contain any thread cycles.*

Proof. Let Q^* be the atomic-threads atomization of P. Lemma 5 tells us that R is equivalent to some schedule of Q^* if and only if the induced access assignment A by R on Q^* is realizable. Furthermore, by Lemma 3, we know that A is realizable if and only if its assignment graph G_A does not contain any cycles. Thus, all we need to prove is that G_A does not contain any cycles if and only if G does not contain any thread cycles, or equivalently, that G_A contains a cycle if and only if G contains a thread cycle.

Below, we prove that G_A contains a cycle if and only if G contains a thread cycle.

Forward direction. If G_A contains a cycle c, then c is of the form $e_0 \xrightarrow{?} e_1 \xrightarrow{?} \cdots \xrightarrow{?} e_{k-1} \xrightarrow{?} e_0$, where any two consecutive threads are connected by either a thread constraint or an access constraint (there are no transaction constraints in the atomic-threads atomization). For every thread edge $e_i \xrightarrow{\mathrm{E}} e_{i+1}$ in c, there is a corresponding thread edge $t_i \xrightarrow{\mathrm{E}} t'_{i+1}$ in G, where t_i is the last transaction of e_i and t'_{i+1} is the first transaction of e_{i+1}. The reason is that Q and Q^*, both being atomizations of the same program P, have the same thread constraints. Also, for every access constraint edge $e_i \xrightarrow{\mathrm{A}} e_{i+1}$ in c, Definition 14 guarantees the existence of an access constraint edge $t_i \xrightarrow{\mathrm{A}} t'_{i+1}$ in G such that t_i belongs to e_i and t'_{i+1} belongs to e_{i+1}. Thus, G contains the thread cycle $C = \big\langle (t_0, t'_1), (t_1, t'_2), \ldots, (t_{k-2}, t'_{k-1}), (t_{k-1}, t'_0) \big\rangle$.

Backward direction. If G contains a thread cycle C, then let $C = \langle (t_0, t'_1), (t_1, t'_2), \ldots, (t_{k-2}, t'_{k-1}), (t_{k-1}, t'_0) \rangle$, and furthermore, let e_i denote the thread containing t'_i and t_i for each i. For every thread edge $t_i \xrightarrow{\mathrm{E}} t'_{i+1}$ in c, there is a corresponding thread edge $e_i \xrightarrow{\mathrm{E}} e_{i+1}$ in G_A. The reason is that Q and Q^*, both being atomizations of the same program P, have the same thread constraints. Also, for every access constraint edge $t_i \xrightarrow{\mathrm{A}} t'_{i+1}$ in C, Definition 14 tells us that there is a resulting access constraint edge $e_i \xrightarrow{\mathrm{A}} e_{i+1}$ in G_A. Thus, G_A contains a cycle c of the form $e_0 \xrightarrow{?} e_1 \xrightarrow{?} \cdots \xrightarrow{?} e_{k-1} \xrightarrow{?} e_0$, where any two consecutive threads are connected by either a thread constraint or an access constraint. □

With Lemma 6, we are ready to extend our analysis from schedules to atomized programs. We can now derive exact conditions for when an atomized program contains a data race.

Definition 16. A ***race assignment*** A of an atomized program Q is an access assignment of Q whose assignment graph contains a thread cycle but does not contain any cycles.

Example. Once again, recall the example access assignment $\{(t_1, t_4), (t_6, t_3)\}$ of our example program, and its assignment graph from Figure 2.5. The previous example showed that this assignment graph contains a thread cycle. When we examine Figure 2.5, however, we find that it does not contain any cycles. Therefore, the example access assignment $\{(t_1, t_4), (t_6, t_3)\}$ is in fact a race assignment of our example program. ◊

The following theorem shows that the existence of a race assignment is a necessary and sufficient condition for the existence of a data race in an atomized program.

Theorem 7. *An atomized program contains a data race if and only if it has a race assignment.*

Proof. Throughout this proof, let Q be the atomized program referred to by the theorem, and let Q^* be the atomic-threads atomization of the program that Q is derived from.

Forward direction. If Q contains a data race, then by Definition 13, there exists a schedule R of Q that is not equivalent to any schedule of Q^*. Then by Lemma 6, the schedule graph G of R must contain a thread cycle. Let A be the set of access constraint edges appearing in one thread cycle. I claim that A is a race assignment of Q. The condition that the assignment graph G_A of A contain a thread cycle is satisfied by construction (we picked the elements of A to make the cycle). Also, we know that A is realizable (by R), which by Lemma 3 tells us that G_A does not contain any cycles, so this condition is satisfied as well.

Backward direction. Let A be a race assignment of Q, and let G_A be its assignment graph. Since A does not contain any cycles, we know by Lemma 3 that it is realizable. Let R be a schedule of Q that realizes A, and let G be the schedule graph of R. Since R realizes A, we know that G is a supergraph of G_A, which implies that G also contains the thread cycle that exists in G_A. Finally, by Lemma 6, we deduce that R is not equivalent to any schedule of Q^*, from which we conclude that Q contains a data race. $\square$

Chapter 4

NP-Completeness of Data-Race Detection

This chapter gives strong evidence that dynamic data-race detection in the transactions-everywhere methodology is intractable in general. It proves that even given the interaction graph of the target atomized program, searching for data races is an NP-complete problem in general[1]. In particular, the NP-hardness of data-race detection is proved via a polynomial-time reduction from the problem of 3cnf-formula satisfiability (*3SAT*) to the problem of data-race detection.

Since the proof is long, it is divided up into four sections. The presentation follows a top-down approach. Section 4.1 is an outline of the complete proof. It references definitions and lemmas in Sections 4.2–4.4 to fill in major parts of the proof.

In order to understand this chapter, the reader should be familiar with the theory of NP-completeness and the problem of 3cnf-formula satisfiability. For a good treatment of these topics, see [37], Sections 7.1–7.4 (pages 225–260).

4.1 Proof Outline

There are two parts to proving the NP-completeness of data-race detection. The first part, proving that the problem of data-race detection belongs in NP, is done in this section. The second part, proving that all problems in NP are polynomial-time reducible to the problem of data-race detection, is outlined in this section, with the bulk of the work to be filled in by Sections 4.2–4.4.

Following standard practice, we first state our problem as a "language" in the computability and complexity sense. The language for our problem is

$$\begin{aligned} \mathit{DATARACE} = \{\, G \mid G &\text{ is the interaction graph of an atomized} \\ &\text{program } Q \text{ that contains a data race.}\} \end{aligned}$$

We do not worry about the exact details of how G is encoded, as any reasonable encoding (having length polynomial in the number of vertices and edges of G) suffices for the purpose of proving NP-completeness.

Here is the main theorem of this chapter and its proof outline.

[1] This chapter does not discuss the obvious problem of a given target program never halting, and thus the dynamic data-race detector never halting. This problem is shared by all data-race detectors that need to run the target program to completion. It is a well documented fact that the halting problem is undecidable (for example, see [37], pages 172–173), and so dynamic data-race detection is indeed intractable in this sense.

Theorem 8. *DATARACE is* NP*-complete.*

Proof. First, we apply Theorem 7 from the previous chapter to redescribe the set *DATARACE* in a form that is easier to work with. Theorem 7 tells us that an atomized program Q contains a data race if and only if it has a race assignment. Consequently, we can write

$$\begin{aligned} \mathit{DATARACE} = \{\, G \mid\ & G \text{ is the interaction graph of an atomized} \\ & \text{program } Q \text{ that has a race assignment.}\} \end{aligned}$$

Recall that a race assignment A is an access assignment whose assignment graph G_A (which is a subgraph of G) contains a thread cycle but does not contain any cycles. We shall use this definition of *DATARACE* for the remainder of this proof. We shall talk in terms of the existence of a race assignment instead of the existence of a data race.

Now, there are two required conditions for *DATARACE* to be NP-complete.

Part 1: DATARACE $\in$ NP*.*

We shall describe a verifier V for checking that a given graph G belongs in *DATARACE*. If $G \in \mathit{DATARACE}$, then its program Q has a race assignment. Let A be a minimal race assignment of Q, by which we mean that no proper subset of A constitutes a race assignment. Furthermore, let the assignment graph of A be G_A. By Definition 16, G_A must contain a thread cycle. Let C be a shortest thread cycle in G_A. We shall use this thread cycle C as the certificate for G in the input to V. Note that if A is minimal, then any thread cycle in the assignment graph of A must contain all the elements of A as access constraint edges, for otherwise, only those access constraint edges that appear in one thread cycle would constitute a race assignment smaller than A. Thus, A is simply the set of all access constraint edges in C, meaning that when we provide C to V, we are also implicitly providing A to V.

The thread cycle C cannot visit the same thread twice, because if it did visit some thread e twice, then we can obtain a shorter thread cycle by removing the portion of C from the first occurrence of e to the last occurrence of e, which would contradict the fact that C is a shortest thread cycle in G_A. Consequently, the size of our certificate C is polynomial (in fact at most linear) in the size of G.

The verifier V checks whether $G \in \mathit{DATARACE}$ using the following steps[2]:

1. Extract A from C by selecting all the access constraint edges.
2. Check that A is a subset of the possible access constraint edges, as defined by the access interaction edges in G.
3. Combine A and G to compute G_A.
4. Check that all the edges of C are contained in G_A.
5. Check that C is a thread cycle.
6. Check that G_A does not contain any cycles (for example, by running a depth-first search from the first vertex of G_A).

If all the checks pass, then V declares $G \in \mathit{DATARACE}$. Note that each of these steps can be completed in polynomial time in the sizes of G and C.

If the interaction graph G of an atomized program Q indeed belongs in *DATARACE*, then by the reasoning in the first paragraph, we see that there exists a certificate C which would prove to V that $G \in \mathit{DATARACE}$. On the other hand, if V accepts an input G and C, then there must

[2]In our problem context, the verifier V does not need to check that G is a valid interaction graph (although this can be done in polynomial time if desired), because we are only concerned with graphs G that are generated from the completed execution of some atomized program Q.

exist a race assignment A of Q (V actually computes it and checks that it meets the conditions for a race assignment), and so $G \in \mathit{DATARACE}$. This reasoning shows that V is a valid verifier for *DATARACE*.

Since there exists a verifier that checks membership in *DATARACE* in polynomial time using a polynomial-size certificate, we conclude that $\mathit{DATARACE} \in \mathrm{NP}$.

Part 2: All languages in NP *are polynomial-time reducible to DATARACE.*

We shall exhibit a polynomial-time reduction from *3SAT* to *DATARACE*. Such a reduction suffices to prove that all languages in NP are polynomial-time reducible to *DATARACE* because it is already known that the same fact is true for *3SAT*.

The mapping reduction that we use is the function F to be defined in Section 4.2. This function can be computed in polynomial time in the size of the input using the steps laid out in Section 4.2. Now let ϕ be any 3cnf boolean formula, let $G = F(\phi)$, and let Q be any atomized program having G as its interaction graph. In order to show that F is a mapping reduction, we need to prove that ϕ has a satisfying assignment if and only if Q has a race assignment.

Forward direction. This direction is proved by Theorem 9 of Section 4.3.

Backward direction. This direction is proved by Theorem 13 of Section 4.4. □

4.2 Reduction Function

This section defines a function F from the set of 3cnf boolean formulas to the set of interaction graphs of atomized programs. We define F by describing a procedure for taking a 3cnf formula ϕ and constructing the corresponding interaction graph $G = F(\phi)$.

At various points in our construction, we shall be spawning an arbitrary number of paths (function instances) from a single vertex (transaction), as illustrated in Figure 4.1(a), where a parent vertex labeled "P" spawns multiple child vertices labeled "C".

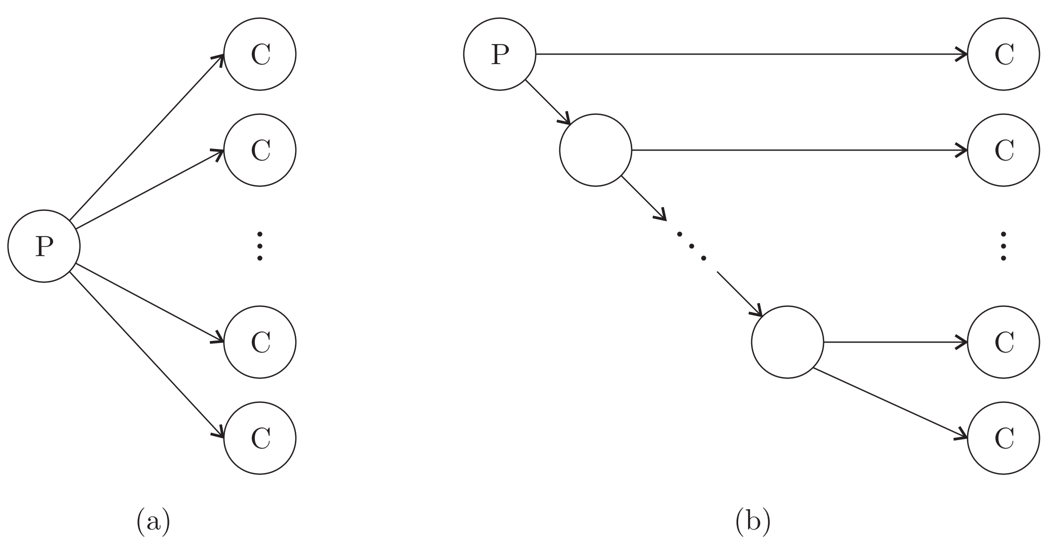

Figure 4.1: The meaning of having one parent vertex spawn multiple child vertices.

However, Cilk allows only one function instance to be spawned at a time. Thus, when our construction calls for the behavior in Figure 4.1(a), what we shall mean is that a loop is used to spawn off one child in each iteration. The resulting graph (including the vertices that would otherwise be hidden) is shown in Figure 4.1(b).

Now we are ready to discuss the process for constructing the interaction graph $G = F(\phi)$ from the 3cnf boolean formula ϕ.

Step 1. Before looking at the formula ϕ, we first begin the construction of the graph G with a base configuration of five vertices, as shown in Figure 4.2. (We continue our practice from previous chapters of labeling vertices with their transaction subscripts.) The base configuration includes the first transaction t_f and last transaction t_g of G. There are also two paths from t_f to t_g, one going through a thread with two transactions t_p followed by t_r, and the other going through a thread with one transaction t_q.

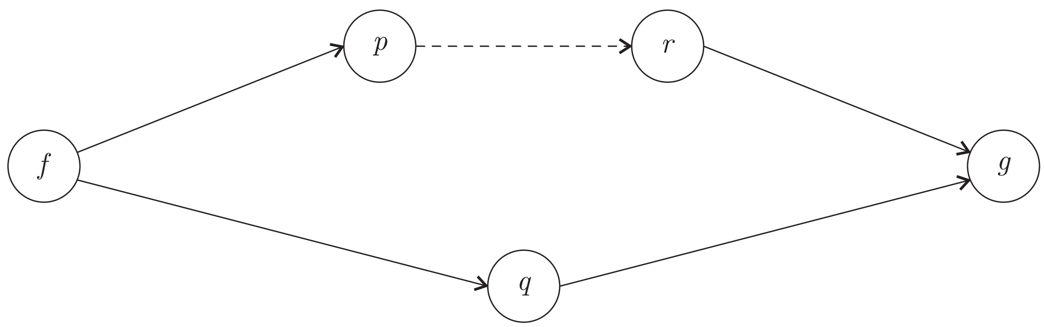

Figure 4.2: The base configuration.

Later in the construction, we shall add more vertices to G in other paths going from t_f to t_g. However, the thread containing t_p and t_r shall remain the only thread in G that consists of more than one transaction.

Step 2. Now, we look at the formula ϕ. Let the variables of ϕ be $x_1, \ldots, x_m$. For each variable x_i, we add a pair of "variable vertices" t_{x_i} and $t_{\overline{x_i}}$ to G, representing the two literals of x_i.

Let the clauses of ϕ be $y_1, \ldots, y_n$. We shall refer to the literal positions within ϕ by the clause number followed by one of the letters a (first), b (second), or c (third). For example, literal position $2c$ refers to the third literal position in clause y_2. For each clause y_j, we add three "clause vertices" t_{ja}, t_{jb}, and t_{jc} to G, representing the three literal positions in y_j.

Example. Consider the example 3cnf formula

$$\phi_{ex} = (x_1 \vee x_2 \vee x_2) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee x_3 \vee \overline{x_3}).$$

Since its variables are x_1, x_2, and x_3, the variable vertices added to $G_{ex} = F(\phi_{ex})$ are t_{x_1}, $t_{\overline{x_1}}$, t_{x_2}, $t_{\overline{x_2}}$, t_{x_3}, and $t_{\overline{x_3}}$. Since it has three clauses, the clause vertices added to G_{ex} are t_{1a}, t_{1b}, t_{1c}, t_{2a}, t_{2b}, t_{2c}, t_{3a}, t_{3b}, and t_{3c}. We shall continue to use this example throughout the remainder of our construction. $\Diamond$

Step 3. For each literal x_i (or $\overline{x_i}$), we use a "gadget" to connect together the variable vertex t_{x_i} (or $t_{\overline{x_i}}$) with the clause vertices that represent the literal positions in ϕ that hold x_i (or $\overline{x_i}$). The goal is to make sure that all the clause vertices appear in parallel, and that they are all direct ancestors of t_{x_i} (or $t_{\overline{x_i}}$). We consider three cases.

In the general case, if a literal x_i (or $\overline{x_i}$) appears in two or more literal positions in ϕ, then the gadget we use consists of a helper vertex, the variable vertex t_{x_i} (or $t_{\overline{x_i}}$), and all the clause vertices that represent literal positions that hold x_i (or $\overline{x_i}$). The connections are simple. The helper vertex spawns off all the clause vertices in parallel, and then they are all synced at t_{x_i} (or $t_{\overline{x_i}}$).

Example. Figure 4.3 shows the gadget for the literal x_2 in our example formula ϕ_{ex}. This literal appears in literal positions $1b$, $1c$, and $2b$. Therefore, in the gadget for x_2, a helper vertex (unlabeled) spawns the clause vertices t_{1b}, t_{1c}, and t_{2b} in parallel, and then these three vertices are subsequently synced at t_{x_2}. ♢

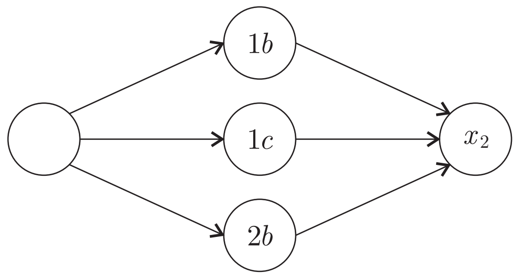

Figure 4.3: The gadget for x_2, which demonstrates the general case.

There are two special cases for our gadget. If a literal x_i (or $\overline{x_i}$) only appears in one literal position in ϕ, then we cannot simply have a helper vertex that spawns the single clause vertex, because each spawn must create at least two children. Instead, we let a helper vertex spawn the clause vertex and a dummy vertex, which are then synced together at t_{x_i} (or $t_{\overline{x_i}}$). Finally, if a literal x_i (or $\overline{x_i}$) does not appear in ϕ at all, then no helper vertex is needed, and the whole gadget for x_i (or $\overline{x_i}$) consists only of the single vertex t_{x_i} (or $t_{\overline{x_i}}$).

Example. Figure 4.4(a) shows the gadget for the literal $\overline{x_1}$ in our example formula ϕ_{ex}, which only appears in literal position $2a$. Since the literal appears in only one literal position, a dummy vertex is placed in parallel with the single clause vertex. The helper and dummy vertices are unlabeled. Figure 4.4(b) shows the gadget for the literal $\overline{x_2}$ in ϕ_{ex}, which does not appear in any literal positions. ♢

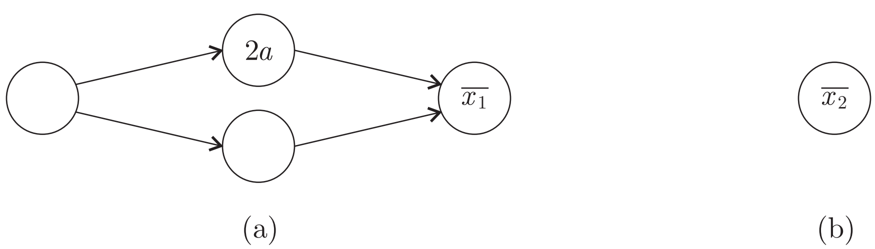

Figure 4.4: The gadgets for $\overline{x_1}$ and $\overline{x_2}$, which demonstrate the special cases.

Step 4. Next, we connect the gadgets for each of the literals to the base configuration from Step 1. The connections are simple. Each gadget receives its own path from the first vertex t_f to the last vertex t_g. Thus, all the gadgets appear in parallel in G. This step completes the serialization subgraph of G.

Example. Figure 4.5 shows the serialization subgraph of $G_{ex} = F(\phi_{ex})$. In addition to the two paths from the base configuration, there is a path from t_f to t_g for each of the six gadgets. ♢

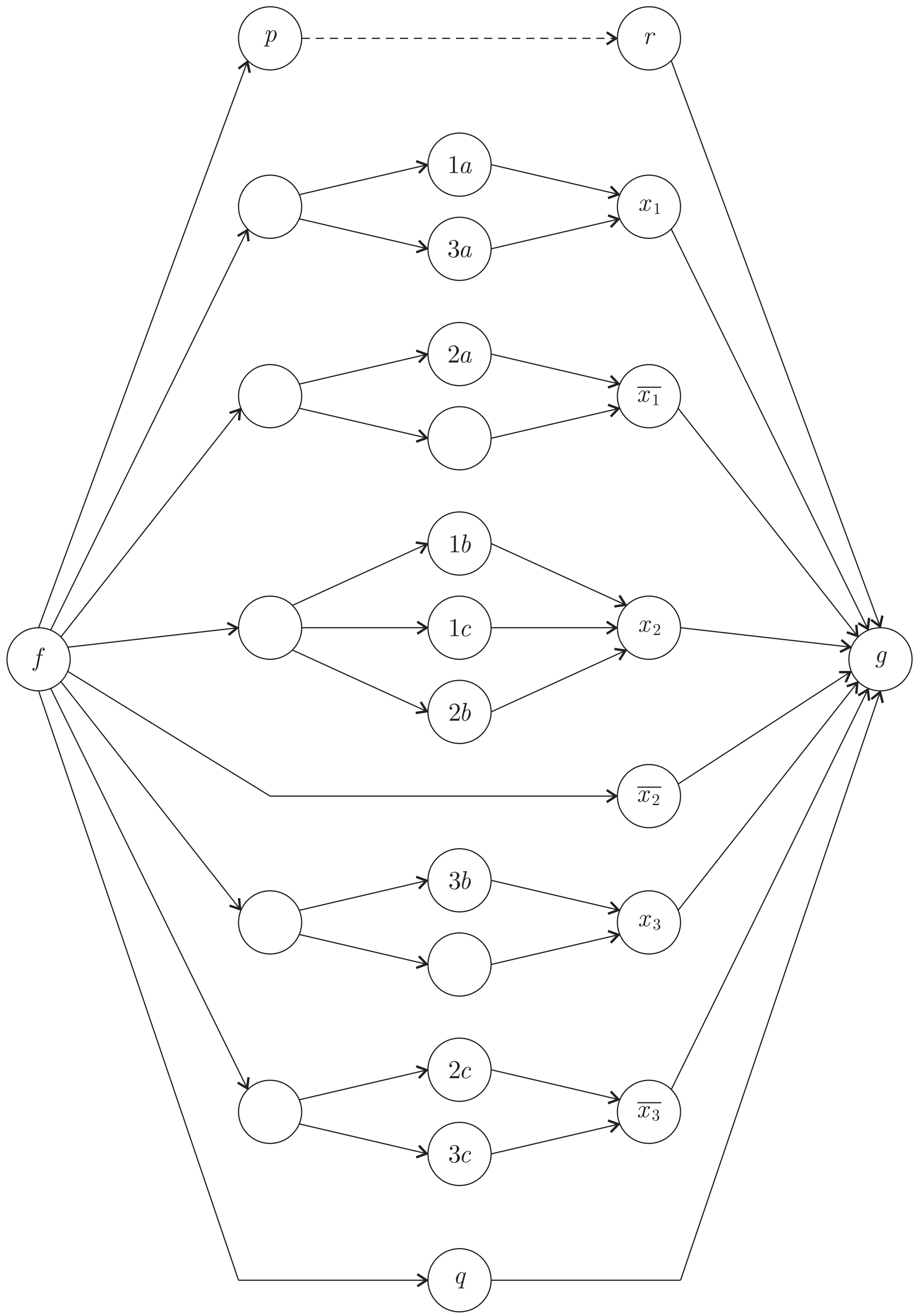

Figure 4.5: The serialization subgraph of G_{ex}.

Step 5. Finally, we add to G the following access interaction edges:

1. t_p is connected to both of t_{x_1} and $t_{\overline{x_1}}$.
2. For each $i \in \{1, \ldots, m-1\}$, both of t_{x_i} and $t_{\overline{x_i}}$ are connected to both of $t_{x_{i+1}}$ and $t_{\overline{x_{i+1}}}$.
3. t_q is connected to both of t_{x_m} and $t_{\overline{x_m}}$.
4. t_q is connected to all three of t_{na}, t_{nb}, and t_{nc}.
5. For each $j \in \{1, \ldots, n-1\}$, all three of t_{ja}, t_{jb}, and t_{jc} are connected to all three of $t_{(j+1)a}$, $t_{(j+1)b}$, and $t_{(j+1)c}$.
6. t_r is connected to all three of t_{1a}, t_{1b}, and t_{1c}.

This completes the construction of G.

Example. Figure 4.6 shows the access interaction edges[3] in $G_{ex} = F(\phi_{ex})$. Following the rules laid out in Step 5, t_p is connected to both of t_{x_1} and $t_{\overline{x_1}}$, both of which are connected to both of t_{x_2} and $t_{\overline{x_2}}$, both of which are connected to both of t_{x_3} and $t_{\overline{x_3}}$, both of which are connected to t_q, which is connected to all three of t_{3a}, t_{3b}, and t_{3c}, all of which are connected to all three of t_{2a}, t_{2b}, and t_{2c}, all of which are connected to all three of t_{1a}, t_{1b}, and t_{1c}, all of which are connected to t_r. ◊

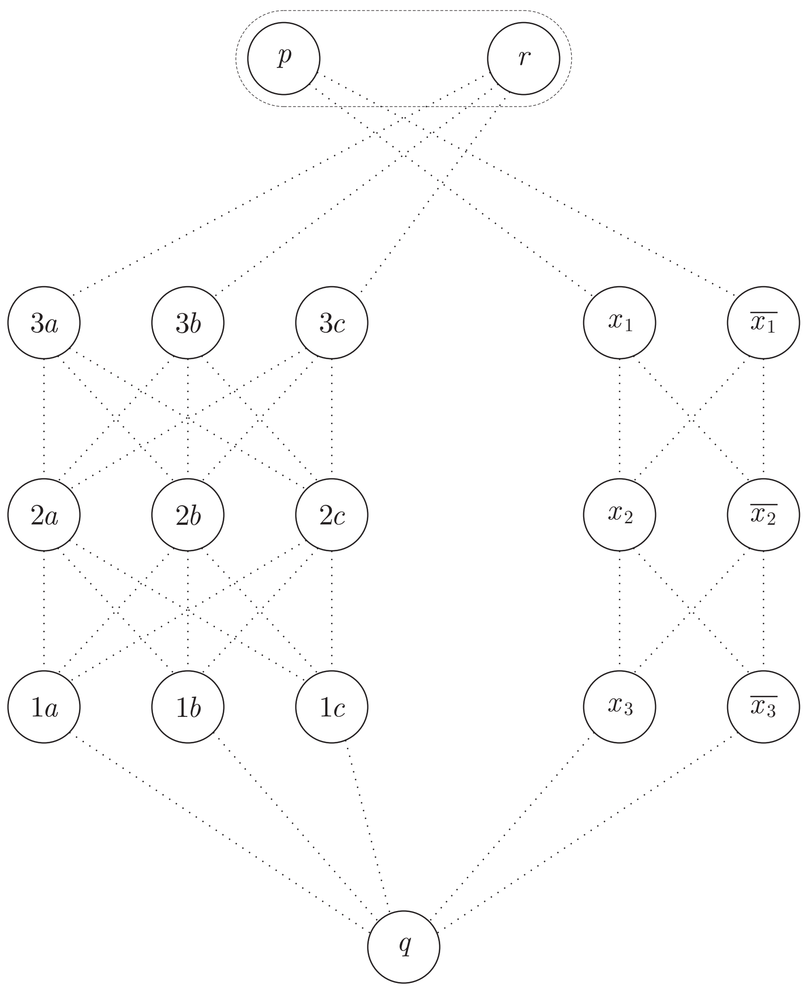

Figure 4.6: The access interaction edges in G_{ex}.

[3]For clarity in this diagram, we omit the serialization edges.

4.3 Forward Direction

This section proves the forward direction in our reduction from *3SAT* to *DATARACE*.

In this section, let F be the function defined in Section 4.2, let ϕ be any 3cnf boolean formula, let $G = F(\phi)$, and let Q be any atomized program having G as its interaction graph.

Theorem 9. *If ϕ has a satisfying assignment, then Q has a race assignment.*

Proof. If ϕ has a satisfying assignment, then let α be such an assignment. We shall construct a race assignment A for Q. The set A consists of all the edges in a thread cycle C (comprising only access constraint edges) selected from G according to the race assignment α.

Let the variables of ϕ be $x_1, \ldots, x_m$, and let its clauses be $y_1, \ldots, y_n$. Without reference to α, the form of C is as follows. It begins at t_p, then goes to either t_{x_1} or $t_{\overline{x_1}}$, then goes to either t_{x_2} or $t_{\overline{x_2}}$, then continues on in this fashion until it reaches either t_{x_m} or $t_{\overline{x_m}}$, then goes to t_q, then goes to one of t_{na}, t_{nb}, or t_{nc}, then goes to one of $t_{(n-1)a}$, $t_{(n-1)b}$, or $t_{(n-1)c}$, then continues on in this fashion until it reaches one of t_{1a}, t_{1b}, or t_{1c}, then finishes by going to t_r.

Notice that for each variable x_i, C goes through exactly one of t_{x_i} or $t_{\overline{x_i}}$. We select which one according to α as follows. If $x_i = $ FALSE in α, then we let C go through t_{x_i}; otherwise ($x_i = $ TRUE), we let C go through $t_{\overline{x_i}}$.

Similarly, notice that for each clause y_j, C goes through exactly one of t_{ja}, t_{jb}, or t_{jc}. We select which one according to α as follows. Since α is a satisfying assignment, every clause in ϕ must be satisfied under α. In particular, for clause y_j, at least one of the three literals in positions ja, jb, and jc must be TRUE. We let C go through a clause vertex that corresponds to a literal position whose literal is TRUE in α. If more than one literal position meets this condition, then we select arbitrarily among them. This completes the selection of C.

Recall that A is simply the set of all edges in C. We shall now prove that A is indeed a race assignment of Q. Certainly the assignment graph G_A of A contains a thread cycle, because by construction it contains the thread cycle C. All that remains is to prove that G_A does not contain any cycles.

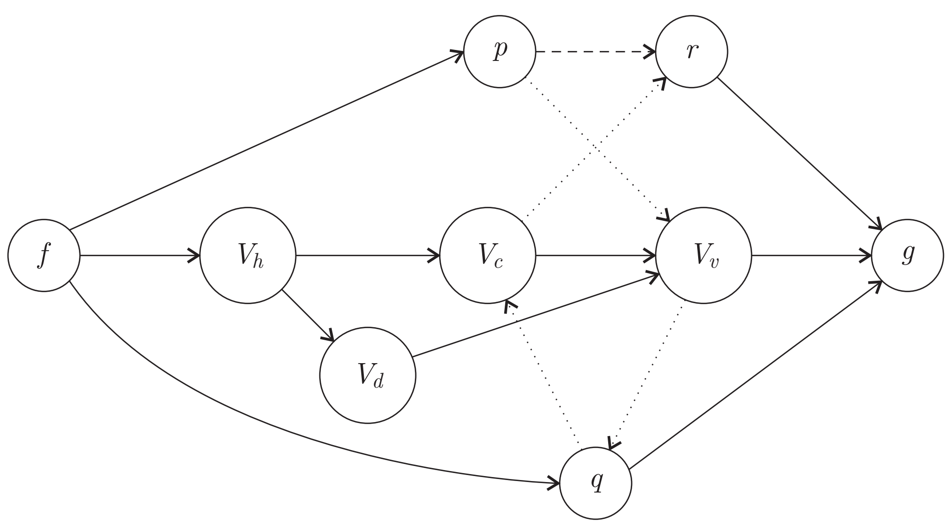

Figure 4.7: The abstracted structure of G_A.

Let V_v, V_c, V_h, and V_d be the sets of variable vertices, clause vertices, helper vertices, and dummy vertices in G_A, respectively. Figure 4.7 shows the abstracted structure of G_A, with these four sets drawn as larger circles. An edge is drawn from one set to another if that type of edge exists from some member of the first set to some member of the second set. We shall use Figure 4.7 to aid in proving that G_A does not contain any cycles.

We shall exhaustively eliminate all possibilities for the existence of a cycle in G_A. First, consider the possibility of a cycle that resides completely within V_v. We note that the vertices in V_v are connected only by the access constraint edges in A. Furthermore, according to the form of C, the access constraint edges in V_v form a path going from either t_{x_1} or $t_{\overline{x_1}}$ to either t_{x_m} or $t_{\overline{x_m}}$ that never revisits a vertex. Thus, a cycle cannot exist that resides completely in V_v. A similar argument shows that a cycle cannot exist that resides completely in V_c. Also, a cycle cannot exist that resides completely within V_h or V_d, because there are no edges connecting two members of either of these two sets.

We now know that if there is a cycle in G_A, then it must traverse the edges shown in Figure 4.7. In this graph, the only nontrivial strongly connected component is the subgraph containing V_c, V_v, and t_q, shown in Figure 4.8. Thus, this subgraph is the only possible form of a cycle in G_A.

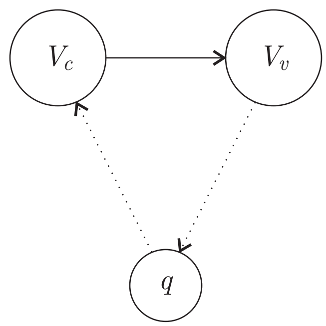

Figure 4.8: The only possible form of a cycle in G_A.

For the purpose of deriving a contradiction, assume that a cycle does exist in the form of Figure 4.8. Then, this cycle must contain a thread edge from a vertex t_{ja}, t_{jb}, or t_{jc} in V_c to a vertex t_{x_i} or $t_{\overline{x_i}}$ in V_v. Without loss of generality, assume that the edge is from a vertex t_{ja} to a vertex t_{x_i} (the argument is the same for the other cases).

First, note that for t_{ja} and t_{x_i} to be part of the cycle, they both must also be incident on access constraint edges in G_A, which means that they both must have been vertices that we selected for inclusion in C. Since we selected t_{ja} for inclusion in C, the literal in position ja in ϕ must be TRUE in α. Moreover, since a thread edge exists from t_{ja} to t_{x_i}, we know from the definition of F that x_i is in fact the literal in position ja, and so we find that $x_i =$ TRUE in α. On the other hand, since we selected t_{x_i} for inclusion in C, we know that $x_i =$ FALSE in α. This contradiction means that a cycle cannot exist in the form of Figure 4.8.

We have now eliminated all possibilities for a cycle in G_A. Since G_A contains the thread cycle C but does not contain any cycles, we conclude that A is indeed a race assignment for Q. $\square$

4.4 Backward Direction

This section proves the backward direction in our reduction from *3SAT* to *DATARACE*.

In this section, let F be the function defined in Section 4.2, let ϕ be any 3cnf boolean formula, let $G = F(\phi)$, and let Q be any atomized program having G as its interaction graph.

Lemma 10. *If Q has a race assignment A with assignment graph G_A, then any thread cycle C in G_A must be of the form $\langle \ldots, (t_{i-1}, t_r), (t_p, t'_{i+1}), \ldots \rangle$. That is, C must go through the thread containing t_p and t_r, and furthermore it must be that t_r is the end of one edge in C and t_p is the start of the next edge in C.*

Proof. Let e_{pr} be the thread containing t_p and t_r. By the definition of F, every thread other than e_{pr} in G_A consists of only a single transaction. If C does not go through e_{pr}, or if C uses only one of the two transactions t_p and t_r whenever it visits e_{pr}, then the edges of C form a (transaction) cycle in G_A, which contradicts the fact that A is a race assignment. Thus, C must go through e_{pr}, and must use both transactions t_p and t_r. Furthermore, suppose that when C goes through e_{pr}, it is always the case that t_p is the end of one edge in C and t_r is the start of the next edge in C. Then, the edges of C combined with the transaction edge $t_p \xrightarrow{\mathrm{T}} t_r$ form a cycle in G_A, which again contradicts the fact that A is a race assignment. Thus, when C goes through e_{pr}, it must be that t_r is the end of one edge in C and t_p is the start of the next edge in C. $\square$

Using what we learned from Lemma 10, and because thread cycles can be written down with any edge as the starting point, we shall from now on consider any thread cycle contained in the assignment graph of a race assignment of Q to have the form $\langle (t_p, t'_1), \ldots, (t_{k-1}, t_r) \rangle$. That is, we shall consider any such thread cycle to start at t_p and end at t_r.

Also, from now on, let V_v, V_c, V_h, and V_d denote the sets of variable vertices, clause vertices, helper vertices, and dummy vertices in G, respectively. Figure 4.9 shows the abstracted structure of the thread and access edges in G, with these four sets drawn as larger circles. An edge is drawn from one set to another if that type of edge exists from some member of the first set to some member of the second set. We shall use Figure 4.9 to aid in the following proofs.

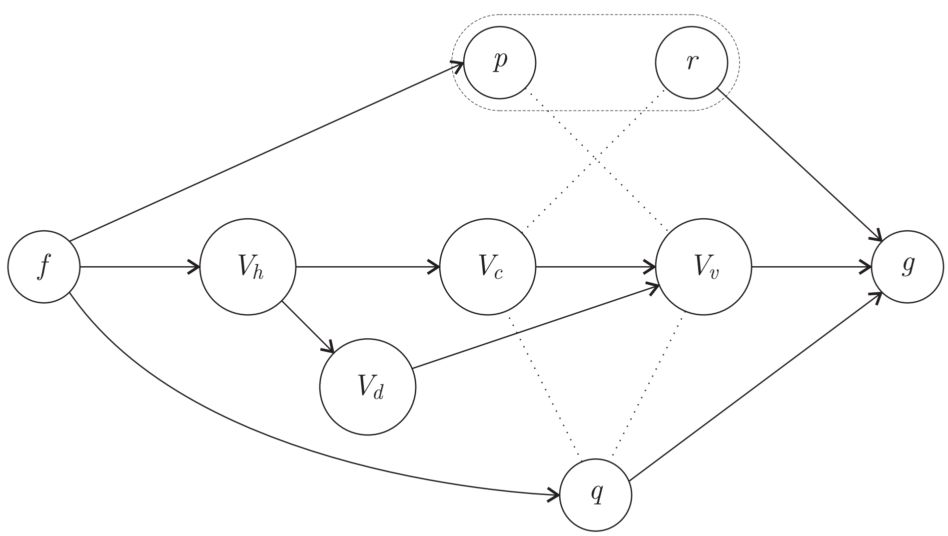

Figure 4.9: The abstracted structure of the thread and access edges in G.

Lemma 11. *If Q has a race assignment A with assignment graph G_A, then any shortest thread cycle C in G_A must start at t_p, then go to V_v, then go to t_q, then go to V_c, then end at t_r.*

Proof. We make two initial observations. The first observation is that C cannot visit the same thread twice. The reason is that if it did visit some thread e twice, then we could obtain a shorter thread cycle by removing the portion of C between the first and last visits to e, which would contradict the fact that C is a shortest thread cycle in G_A.

The second observation is that, if at any point C visits a vertex in V_v, then it must visit t_q at a later point. We can see this by examining Figure 4.9, which shows the abstracted structure of the thread and access edges in G. These are all the edges in G that can appear in a thread cycle. As can be seen from this diagram, the only vertices that can be reached from V_v, without going through t_q, are t_p and t_g. However, we know by Lemma 10 (and the ensuing paragraph) that C must end at t_r, so C must go through t_q at a later point in order to reach t_r.

With these two observations, we are now ready to prove that C does take the path claimed. Please refer to Figure 4.9 while reading the following steps of reasoning:

1. First, by Lemma 10 (and the ensuing paragraph), we know that C starts at t_p.
2. From t_p, C can only go to V_v.
3. From V_v, C can go to t_p, t_g, or t_q.
   a. If C goes to t_p, then it visits t_p twice, which is illegal by our first observation.
   b. If C goes to t_g, then it hits a dead end.
   c. Thus, C must go to t_q.
4. From t_q, C can go to t_g, V_v, or V_c.
   a. If C goes to t_g, then it hits a dead end.
   b. If C goes to V_v, then by our second observation, it must later visit t_q for a second time, which is illegal by our first observation.
   c. Thus, C must go to V_c.
5. From V_c, C can go to t_q, V_v, or t_r.
   a. If C goes to t_q, then it visits t_q twice, which is illegal by our first observation.
   b. If C goes to V_v, then by our second observation, it must later visit t_q for a second time, which is illegal by our first observation.
   c. Thus, C must go to t_r.
6. At t_r, C can continue on or C can end.
   a. If C continues on, then at the end it must return to t_r for a second time, which is illegal by our first observation.
   b. Thus, C must end.

This proves that C must start at t_p, then go to V_v, then go to t_q, then go to V_c, then end at t_r. □

In the following lemma, we draw some corollaries from Lemma 11.

Lemma 12. *If Q has a race assignment A with assignment graph G_A, then for any shortest thread cycle C in G_A, the following properties hold:*

1. *C is composed entirely of access constraint edges,*
2. *for each variable x_i, C goes through at least one of t_{x_i} or $t_{\overline{x_i}}$, and*
3. *for each clause y_j, C goes through at least one of t_{ja}, t_{jb}, or t_{jc}.*

Proof. For the first property, we know from Lemma 11 that C must be of the form $t_p \to V_v \to t_q \to V_c \to t_r$. By the definition of F, there can only be access edges from t_p to V_v, within V_v, from V_v to t_q, from t_q to V_c, within V_c, and from V_c to t_r.

For the second property, since C enters V_v from t_p and exits to t_q using only access edges, we can deduce from Step 5 in the definition of F (see Figure 4.6) that C must go through at least one of t_{x_i} or $t_{\overline{x_i}}$ for each x_i.

Similarly, for the third property, since C enters V_c from t_q and exits to t_r using only access edges, we can deduce from Step 5 in the definition of F that C must go through at least one of t_{ja}, t_{jb}, or t_{jc} for each y_j. □

Finally, we are ready to prove the backward direction in our reduction.

Theorem 13. *If Q has a race assignment, then ϕ has a satisfying assignment.*

Proof. If Q has a race assignment, then let A be such an assignment, let G_A be the assignment graph of A, and let C be a shortest thread cycle in G_A. We shall construct a satisfying assignment α for ϕ using C.

By Lemma 12, we know that for each variable x_i, C must go through at least one of t_{x_i} or $t_{\overline{x_i}}$. If C goes through only t_{x_i}, then we set $x_i = \text{FALSE}$ in α. If C goes through only $t_{\overline{x_i}}$, then we set $x_i = \text{TRUE}$ in α. Finally, if C goes through both t_{x_i} and $t_{\overline{x_i}}$, then we choose arbitrarily whether $x_i = \text{TRUE}$ or $x_i = \text{FALSE}$. As a result of our construction of α, if a literal x_i (or $\overline{x_i}$) is FALSE in α, then C must go through t_{x_i} (or $t_{\overline{x_i}}$). We shall use this key observation later.

We need to prove that this assignment α indeed satisfies ϕ. By Lemma 12, we know that for each clause y_j, C goes through at least one of t_{ja}, t_{jb}, or t_{jc}. I claim this clause vertex that C goes through (if more than one, then select one arbitrarily) corresponds to a literal position that holds a literal that is TRUE in α, and thus clause y_j is satisfied under α. If this claim is true, then the theorem follows, since every clause y_j would then be satisfied under α, showing that α is a satisfying assignment for ϕ.

We now prove the claim. Without loss of generality, say that C goes through t_{ja} (the argument is the same for t_{jb} or t_{jc}). Also without loss of generality, say that literal position ja happens to hold a literal x_i (the argument is the same for $\overline{x_i}$). For the sake of deriving a contradiction, assume that $x_i = \text{FALSE}$ in α. Then, by the key observation above, C must go through t_{x_i}. Thus, we know that C goes through both t_{x_i} and t_{ja}. Combining this knowledge with Lemma 11, which tells us that the form of C must be $t_p \to V_v \to t_q \to V_c \to t_r$, we find that a portion of C must be a (transaction) path from t_{x_i} to t_{ja}. Finally, since literal position ja holds x_i, we know by the definition of F that there is a thread edge $t_{ja} \xrightarrow{\text{E}} t_{x_i}$. But this edge combined with the portion of C that goes from t_{x_i} to t_{ja} forms a cycle in G_A, contradicting the fact that A is a race assignment. Thus, our assumption that $x_i = \text{FALSE}$ in α must be incorrect. We conclude that $x_i = \text{TRUE}$ in α and that clause y_j is indeed satisfied under α by literal x_i in position ja. $\square$

Chapter 5

An Algorithm for Approximately Detecting Data Races

We learned in Chapter 4 that detecting data races in atomized programs is an NP-complete problem in general. This fact leads us to search for efficient algorithms that approximately detect data races. This chapter presents one such algorithm.

Since the complete algorithm is long, it is broken into three parts $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$. Part $\mathcal{A}$, presented in Section 5.1, involves instrumentation of the target atomized program for recording its serialization structure and shared-memory accesses. Part $\mathcal{B}$, presented in Section 5.2, computes the least common ancestors (LCA) structure [1, 35, 21] of the threads and finds the access interactions. Part $\mathcal{C}$, presented in Section 5.3, uses the information prepared in the first two parts to search for a possible data race.

Section 5.4 states and proves an exact set of conditions under which part $\mathcal{C}$ of the algorithm reports a data race, and also proves that the algorithm never reports any false negatives. Then, it gives an intuitive explanation of the algorithm and discuss how the programmer should use the reported information about a possible data race. Finally, it combines the analyses from Sections 5.1–5.3 to show that the total running time of the algorithm is worst-case quadratic in the size of the target program's interaction graph.

5.1 Part $\mathcal{A}$: Serialization Structure and Shared-Memory Accesses

Part $\mathcal{A}$ of the algorithm involves instrumentation of the target atomized program for recording its serialization structure and shared-memory accesses. This section first describes the variables used in this part of the algorithm. Next, it provides pseudocode with text explanations. For ease of understanding, the pseudocode shown is slightly inaccurate, with corrections noted afterward. Then, this section analyzes the running time of this part of the algorithm. Finally, it gives brief arguments about correctness.

Variables. Below is a list of all the variables used in part $\mathcal{A}$, excluding local variables that are only defined for a very small section of code. The variables are categorized into input, output, and local variables. Within each category, the variables are further divided according to the data they contain. Each variable in the list is accompanied by a short text description. A description in paragraph form follows the list.

In the array lengths below, p denotes the number of function instances in the target program,

m the number of threads, n the number of transactions, and L the number of shared-memory locations. Correspondingly, in the algorithm, function instances are numbered 0 to $p-1$, threads numbered 0 to $m-1$, transactions numbered 0 to $n-1$, and shared-memory locations numbered 0 to $L-1$. These letters shall denote the same quantities throughout the remainder of this chapter. Also, the constants TRUE, FALSE, and NONE are used throughout the algorithm. Since everything is numbered starting from 0, the constant NONE is defined as -1. In many situations, this definition eliminates the need to check for NONE as a special case.

★ Input variables: *Not applicable.*

★ Output variables:

- information about threads:
  - m ▷ number of threads
  - *edge-e1*$[0 \,..\, m-1]$ ▷ first thread constraint
  - *edge-e2*$[0 \,..\, m-1]$ ▷ second thread constraint

- information about transactions:
  - n ▷ number of transactions
  - *thr*$[0 \,..\, n-1]$ ▷ enclosing thread
  - *reads*$[0 \,..\, n-1]$ ▷ list of memory locations read
  - *writes*$[0 \,..\, n-1]$ ▷ list of memory locations written

- information about shared-memory locations:
  - *readers*$[0 \,..\, L-1]$ ▷ list of transactions that read a location
  - *writers*$[0 \,..\, L-1]$ ▷ list of transactions that write a location

★ Local variables:

- information about function instances:
  - p_{new} ▷ next unused function instance index
  - *parent*$[0 \,..\, p-1]$ ▷ parent function instance
  - *spawned*$[0 \,..\, p-1]$ ▷ list of spawned function instances
  - *next-tran*$[0 \,..\, p-1]$ ▷ next transaction to be executed (defined when executing a ▷ spawned child to remember where to return to)
  - *final-thr*$[0 \,..\, p-1]$ ▷ final thread (only defined when done)

- information about threads:
  - m_{new} ▷ next unused thread index

- information about transactions:
  - n_{new} ▷ next unused transaction index
  - *fun*$[0 \,..\, n-1]$ ▷ enclosing function instance

- information about the current transaction:
  - t_{curr} ▷ currently executing transaction
  - *is-new-thread* ▷ TRUE if current transaction is start of a new thread
  - *reads-temp* ▷ list of memory locations read by t_{curr} (repeats allowed)
  - *writes-temp* ▷ list of memory locations written by t_{curr} (repeats allowed)

Since part $\mathcal{A}$ is the first part of the algorithm, there are no input variables from earlier parts. The output variables that are recorded by part $\mathcal{A}$ for use in parts $\mathcal{B}$ and $\mathcal{C}$ are as follows.

First, part $\mathcal{A}$ keeps counts of the number of threads and the number of transactions in the program, and outputs them in the variables m and n, respectively. An array *thr* records transaction membership in threads, so that for each transaction t, $thr[t]$ is the index of the thread that t belongs to. Note that t denotes the transaction index.

The serialization graph is recorded as follows. The outgoing thread edges are stored in two arrays *edge-e1* and *edge-e2*. Two arrays suffice because any thread can have at most two outgoing thread edges. For example, if a thread e has exactly one outgoing thread edge to thread e_1, then $\mathit{edge\text{-}e1}[e] = e_1$ and $\mathit{edge\text{-}e2}[e] = \text{NONE}$. The transaction edges are stored implicitly. By the algorithm's method of numbering transactions, an earlier transaction t_1 in a thread always receives a smaller index than a later transaction t_2 in the same thread. Thus, we know there is a path from t_1 to t_2 consisting of transaction edges if the transactions belong to the same thread (i.e. $thr[t_1] = thr[t_2]$) and $t_1 < t_2$.

Two arrays *readers* and *writers* are kept for recording which transactions read or write a location. Each element $\mathit{writers}[\ell]$ is the head of a linked list of indices of transactions that write ℓ. The array *readers* is similarly defined, except that if a transaction both reads and writes a location ℓ, then the transaction is only recorded in $\mathit{writers}[\ell]$.

The arrays *reads* and *writes* are the dual of the arrays *readers* and *writers*. Each element $\mathit{writes}[t]$ is the head of a linked list of shared-memory locations that are written by transaction t. The array *reads* is similarly defined, except that if a transaction t both reads and writes a shared-memory location, then the location is only recorded in $\mathit{writes}[t]$.

The variables local to part $\mathcal{A}$ are as follows.

Three counters p_{new}, m_{new}, and n_{new} are employed to keep track of the next unused function instance index, thread index, and transaction index, respectively, for the purpose of allocating new indices. Variables m_{new} and n_{new} also serve the additional purpose of counting the numbers of threads and transactions, respectively, so that these counts can be output in variables m and n. An array *fun* records transaction membership in function instances, so that for each transaction t, $\mathit{fun}[t]$ is the index of the function instance that t belongs to.

For each function instance f, we keep track of the following information. The value $\mathit{parent}[f]$ is the index of the parent function instance that spawned f, with $\mathit{parent}[f] = \text{NONE}$ if f is the `main` function instance. The value $\mathit{spawned}[f]$ is the head of a linked list of function instances that were spawned by f and have not yet been synced. The value $\mathit{next\text{-}tran}[f]$ is used to remember the index of the next transaction to be executed after a spawn point in f, during the execution of that spawned function instance (and its descendants). Finally, when f returns, $\mathit{final\text{-}thr}[f]$ records the index of the final thread in f.

Throughout part $\mathcal{A}$ of the algorithm, t_{curr} is always set to the index of the currently executing transaction. The boolean variable *is-new-thread* is used to indicate whether the current transaction is the first transaction in a new thread. The linked lists *reads-temp* and *writes-temp* record the shared-memory locations that have been read and written by t_{curr} so far, with repeats allowed. Thus, each time that a shared-memory location ℓ is read or written by t_{curr}, ℓ is inserted into *reads-temp* or *writes-temp*, respectively.

Pseudocode and Explanation. In the instrumentation of the target program, we need to add code that runs during the start of the program, during function spawn, return, and sync points, during transaction begin points, during accesses to shared-memory locations, and during transaction end points. The pseudocode for each situation is provided below. Although the pseudocode is reasonably self-explanatory, a text explanation follows each piece of pseudocode for completeness.

The instrumented program is to be run in serial fashion on one processor, so that the transactions and threads are executed in the order that they would appear in the serial elision of the target program. Specifically, whenever a function instance f_{parent} spawns a function instance f_{child}, the spawned instance f_{child} always executes first. Only when f_{child} is completely finished and has returned does the next transaction after the spawn point in f_{parent} execute.

Program Start:

1 $p_{new} \leftarrow 0$
2 $m_{new} \leftarrow 0$
3 $n_{new} \leftarrow 0$

4 **for** $\ell \leftarrow 0$ to $L-1$
5   List-Empty($readers[\ell]$)
6   List-Empty($writers[\ell]$)

7 $t_{curr} \leftarrow n_{new}$++
8 $thr[t_{curr}] \leftarrow m_{new}$++
9 $fun[t_{curr}] \leftarrow p_{new}$++
10 List-Empty($spawned[\,fun[t_{curr}]\,]$)

11 $parent[\,fun[t_{curr}]\,] \leftarrow$ NONE

12 $is\text{-}new\text{-}thread \leftarrow$ TRUE

The above code is executed at the start of the target program. Lines 1–3 initialize the counters p_{new}, m_{new}, and n_{new}. Lines 4–6 initialize the arrays *readers* and *writers*, so that all the lists are initially empty. Lines 7–9 allocate indices for the current (first) transaction t_{curr} and its associated thread and function instance. (For any variable v, the notation v++ is used to indicate that v is to be incremented at the end of the current statement.) Line 10 initializes the *spawned* list of the current function instance $fun[t_{curr}]$, which is something that needs to be done at the start of every new function instance. Line 11 sets the *parent* of the current function instance $fun[t_{curr}]$ to NONE, because $fun[t_{curr}]$ is the `main` function instance of the program. Finally, line 12 sets *is-new-thread* to TRUE because a new thread is starting.

Function Spawn:

$t_{next} \leftarrow n_{new}$++
$thr[t_{next}] \leftarrow m_{new}$++
$fun[t_{next}] \leftarrow fun[t_{curr}]$

$t_{child} \leftarrow n_{new}$++
$thr[t_{child}] \leftarrow m_{new}$++
$fun[t_{child}] \leftarrow p_{new}$++
List-Empty($spawned[\,fun[t_{child}]\,]$)

$edge\text{-}e1[\,thr[t_{curr}]\,] \leftarrow thr[t_{next}]$
$edge\text{-}e2[\,thr[t_{curr}]\,] \leftarrow thr[t_{child}]$

10 $\mathit{parent}[\,\mathit{fun}[t_{\mathit{child}}]\,] \leftarrow \mathit{fun}[t_{\mathit{curr}}]$
11 LIST-INSERT($\mathit{spawned}[\,\mathit{fun}[t_{\mathit{curr}}]\,]$, $\mathit{fun}[t_{\mathit{child}}]$)

12 $\mathit{next\text{-}tran}[\,\mathit{fun}[t_{\mathit{curr}}]\,] \leftarrow t_{\mathit{next}}$
13 $t_{\mathit{curr}} \leftarrow t_{\mathit{child}}$
14 $\mathit{is\text{-}new\text{-}thread} \leftarrow$ TRUE

The above code is executed at function spawn points. Lines 1–3 create the next transaction t_{next} following the spawn point and its associated thread. The function instance of t_{next} is the same as the current function instance $\mathit{fun}[t_{curr}]$. Lines 4–6 create the first transaction t_{child} of the child function instance, and also its associated thread and function instance. Since the child function instance $\mathit{fun}[t_{child}]$ is new, line 7 initializes its *spawned* list. Lines 8–9 record the thread edges from the current thread $\mathit{thr}[t_{curr}]$ (before the spawn point) to the two new threads created, one being the next thread $\mathit{thr}[t_{next}]$ in the current function instance and the other being the first thread $\mathit{thr}[t_{child}]$ of the child function instance. Line 10 sets the current function instance $\mathit{fun}[t_{curr}]$ to be the parent of the child function instance $\mathit{fun}[t_{child}]$, and line 11 reciprocally adds the child function instance to the *spawned* list of the current function instance. Line 12 saves the index of the next transaction t_{next} to be executed in the current function instance, so that we know where to return to when the child function instance is finished. Finally, line 13 changes the value of t_{curr} to the index of the first transaction t_{child} in the child function instance, in preparation for its execution. Line 14 notes that we are beginning a new thread.

FUNCTION RETURN:

1 $\mathit{final\text{-}thr}[\,\mathit{fun}[t_{\mathit{curr}}]\,] \leftarrow \mathit{thr}[t_{\mathit{curr}}]$

2 **if** $\mathit{parent}[\,\mathit{fun}[t_{\mathit{curr}}]\,] =$ NONE
3 $\quad \mathit{edge\text{-}e1}[\,\mathit{thr}[t_{\mathit{curr}}]\,] \leftarrow$ NONE
4 $\quad \mathit{edge\text{-}e2}[\,\mathit{thr}[t_{\mathit{curr}}]\,] \leftarrow$ NONE
5 $\quad m \leftarrow m_{\mathit{new}}$
6 $\quad n \leftarrow n_{\mathit{new}}$
7 **else**
8 $\quad t_{\mathit{curr}} \leftarrow \mathit{next\text{-}tran}[\,\mathit{parent}[\,\mathit{fun}[t_{\mathit{curr}}]\,]\,]$
9 $\quad \mathit{is\text{-}new\text{-}thread} \leftarrow$ TRUE

The above code is executed when a function instance returns. Since we are at the end of the current function instance, line 1 records the final thread of the current function instance $\mathit{fun}[t_{curr}]$ to be the current thread $\mathit{thr}[t_{curr}]$. Line 2 tests to see if the current function instance is the `main` function instance of the program (only the `main` function instance has its parent equal to NONE). If so, then we are finished running the target program. To wrap up, lines 3–4 set the outgoing thread edges of the current thread $\mathit{thr}[t_{curr}]$ (which is the last thread of the program) to NONE, and lines 5–6 set the output variables m and n to the final values of the counters m_{new} and n_{new}, respectively. Else, if we are not at the end of the `main` function instance, then line 8 changes the value of t_{curr} to the index of the next transaction in the parent function instance $\mathit{parent}[\,\mathit{fun}[t_{curr}]\,]$, in preparation for its execution. Recall that the setup operations for the next transaction in the parent function instance were done when the current function instance was spawned. Finally, line 9 notes that we are beginning a new thread.

Function Sync:

1 $t_{next} \leftarrow n_{new}$++
2 *thr*$[t_{next}] \leftarrow m_{new}$++
3 *fun*$[t_{next}] \leftarrow$ *fun*$[t_{curr}]$

4 *edge-e1*[*thr*$[t_{curr}]$] $\leftarrow$ *thr*$[t_{next}]$
5 *edge-e2*[*thr*$[t_{curr}]$] $\leftarrow$ NONE

6 **for** each $f_{child} \in$ *spawned*[*fun*$[t_{curr}]$]
7   *edge-e1*[*final-thr*$[f_{child}]$] $\leftarrow$ *thr*$[t_{next}]$
8   *edge-e2*[*final-thr*$[f_{child}]$] $\leftarrow$ NONE
9 List-Empty(*spawned*[*fun*$[t_{curr}]$])

10 $t_{curr} \leftarrow t_{next}$
11 *is-new-thread* $\leftarrow$ TRUE

The above code is executed at function sync points. Lines 1–3 create the next transaction t_{next} following the sync point and its associated thread. The function instance of t_{next} is the same as the current function instance *fun*$[t_{curr}]$. Then, the task is to record thread edges for all the threads that end at this sync point. These include the current thread, as well as the final threads of all the child function instances that were previously spawned by the current function instance, but which have not yet been synced. Line 4 records the outgoing thread edge from the current thread *thr*$[t_{curr}]$ to the next thread *thr*$[t_{next}]$, and line 5 notes that the current thread does not have a second outgoing thread edge. Lines 6–8 do the same for the final threads *final-thr*$[f_{child}]$ of each previously spawned child function instance f_{child}. Line 9 empties the *spawned* list of the current function instance *fun*$[t_{curr}]$, because all previously spawned child function instances have now been synced. Finally, line 10 changes the value of t_{curr} to the index of the next transaction t_{next} in preparation for its execution, and line 11 notes that we are beginning a new thread.

Transaction Begin:

1 **if** *is-new-thread* = TRUE
2   *is-new-thread* $\leftarrow$ FALSE
3 **else**
4     $t_{prev} \leftarrow t_{curr}$
5     $t_{curr} \leftarrow n_{new}$++
6   *thr*$[t_{curr}] \leftarrow$ *thr*$[t_{prev}]$
7   *fun*$[t_{curr}] \leftarrow$ *fun*$[t_{prev}]$

8 List-Empty(*reads-temp*)
9 List-Empty(*writes-temp*)

The above code is executed at the beginning of a transaction. Line 1 checks to see if the current transaction is the start of a new thread. If so, then the setup operations for creating the current transaction have already been done, in which case line 2 resets *is-new-thread* to FALSE. Else, if the current transaction is not the start of a new thread, then the current transaction has not yet been created, and t_{curr} actually holds the index of the previous transaction. In this case, line 4 saves the index of the previous transaction as t_{prev}, and lines 5–7 create the current transaction t_{curr}

and set its thread and function instance to be the same as those of t_{prev}. Lines 8–9 empty the lists *reads-temp* and *writes-temp* in preparation for recording the shared-memory accesses made by the current transaction t_{curr}.

READ LOCATION ℓ:

```
1   LIST-INSERT( reads-temp, ℓ )
```

WRITE LOCATION ℓ:

```
1   LIST-INSERT( writes-temp, ℓ )
```

One of the two lines of code above is executed when a shared-memory location is read or written by the current transaction, to record the access in *reads-temp* or *writes-temp*, respectively. Recall that repeats are allowed in *reads-temp* and *writes-temp*.

TRANSACTION END:

```
 1   LIST-EMPTY( reads[t_curr] )
 2   LIST-EMPTY( writes[t_curr] )

 3   for ℓ ← 0 to L − 1
 4           processed[ℓ] ← FALSE

 5   for each ℓ ∈ writes-temp
 6           if processed[ℓ] = FALSE
 7                   LIST-INSERT( writes[t_curr], ℓ )
 8                   LIST-INSERT( writers[ℓ], t_curr )
 9                   processed[ℓ] ← TRUE

10   for each ℓ ∈ reads-temp
11           if processed[ℓ] = FALSE
12                   LIST-INSERT( reads[t_curr], ℓ )
13                   LIST-INSERT( readers[ℓ], t_curr )
14                   processed[ℓ] ← TRUE
```

The above code is executed when a transaction ends. Lines 1–2 initialize the *reads* and *writes* lists for the current transaction. Lines 3–4 initialize a temporary array *processed*$[\,0 \mathinner{..} L-1\,]$ of boolean values, which are used to keep track of which shared-memory locations ℓ have already had accesses transferred from *reads-temp* and *writes-temp* (which allow repeats) to *reads*, *writes*, *readers*, and *writers* (which do not allow repeats). Line 5 iterates through *writes-temp*, which stores all the writes to shared memory that were made by the current transaction t_{curr}, and if a location ℓ has not yet been processed, then lines 7–8 insert ℓ into the *writes* list of t_{curr}, and also insert t_{curr} into the *writers* list of ℓ. Lines 10–14 accomplish the same task as lines 5–9, but for the current transaction's reads to shared memory. Note that the array *processed* is not cleared between lines 5–9 and lines 10–14, so that if the current transaction both reads and writes a shared-memory location ℓ, then only a write would be recorded.

Notes. Now some notes are in order. The pseudocode presented above for part $\mathcal{A}$ of the algorithm is actually not completely accurate. It was presented in the above form for ease of explanation and understanding. Two changes need to be made, however, in order to make part $\mathcal{A}$ correct and efficient.

First, note that the values p, m, and n are not available at the beginning of the algorithm, so arrays with these sizes cannot actually be allocated. There are two simple solutions to this dilemma. One solution is to run the target program once for the purpose of determining these values, and then to allocate the arrays and run the instrumented program again as described above. The other solution is simply to use dynamic arrays (for example, see [6], pages 416–425).

Second, note that the number of shared-memory locations L is likely a large number, possibly much larger than the number of shared-memory locations that are actually used. In order not to have the running time and space usage depend on L, we can implement the arrays *readers*, *writers*, and *processed* as dynamic perfect hash tables [7]. This change replaces lines 4–6 in the pseudocode for "Program Start" and lines 3–4 in the pseudocode for "Transaction End" with constant-time hash table initialization statements. This change also makes the running time both amortized and expected.

Running Time. Let us analyze the running time for each piece of pseudocode in turn, and then combine the results. Recall that p is the number of function instances, m is the number of threads, and n is the number of transactions. Also, let us define α to be the number of accesses to shared-memory locations.

- Program Start — This piece of code executes only once. If *readers* and *writers* are implemented as dynamic perfect hash tables, as mentioned in the notes above, then all the operations in this piece of code take constant time. Thus, this code uses a total of $\Theta(1)$ time.
- Function Spawn — This piece of code executes a total of $\Theta(p)$ times. The operations in this piece of code all take constant time. Thus, this piece of code uses a total of $\Theta(p)$ time.
- Function Return — This piece of code executes a total of $\Theta(p)$ times. The operations in this piece of code all take constant time. Thus, this piece of code uses a total of $\Theta(p)$ time.
- Function Sync — Note that there is no requirement for each function sync be associated with one or more function spawns[1]. As a result, we can only say that this piece of code executes $O(m)$ times. The only portion of this code that does not take constant time is the **for** loop at line 6. Over all the times that this piece of code executes, the **for** loop at line 6 iterates a total of $\Theta(p)$ times, since each spawned function instance is synced exactly once. The code inside the **for** loop takes constant time, so all the iterations take a total of $\Theta(p)$ time. Thus, this piece of code uses a total of $O(m) + \Theta(p) = O(m)$ time.
- Transaction Begin — This piece of code executes $\Theta(n)$ times. The operations in this piece of code all take constant time. Thus, this piece of code uses a total of $\Theta(n)$ time.
- Read Location and Write Location — One of these two pieces of code is executed for each shared-memory access. Thus, these pieces of code use a total of $\Theta(\alpha)$ time.
- Transaction End — This piece of code executes $\Theta(n)$ times. Over all the times that this piece of code executes, the **for** loops at lines 5 and 10 iterate a total of $\Theta(\alpha)$ times, since *reads-temp* and *writes-temp* simply record all the accesses to shared-memory locations. All the operations outside and inside the **for** loops take constant time (remember that lines 3–4 now take constant time since *processed* is implemented as a dynamic perfect hash table). Thus, this piece of code uses a total of $\Theta(n + \alpha)$ time.

Finally, we must not forget that in addition to the execution of our code, the original target program must also execute. Let τ be the running time of the uninstrumented target program. Since p, m,

[1] If every sync were to be associated with at least one spawn, then it would be straightforward to prove that $p \le m < 3p$, which would imply that $m = \Theta(p)$. However, we do not make this assumption.

n, and α are all $O(\tau)$, the total running time for all the code inserted by part $\mathcal{A}$ is $O(\tau)$. Thus, the total running time for part $\mathcal{A}$ of the algorithm is simply $O(\tau) + \tau = \Theta(\tau)$.

Correctness. We shall argue that all the output variables in part $\mathcal{A}$ have correct values. We shall be brief, because the pseudocode in this part of the algorithm is fairly self-explanatory.

First, m_{new} and n_{new} are initialized to 0 and incremented each time a new thread or transaction is created, respectively, so they are correct counts for the numbers of threads and transactions, respectively. At the end, their values are copied into m and n, so m and n are correct.

The values in *thr* are correct because whenever the current transaction t_{curr} is the start of a new thread, we allocate a new thread index for $thr[t_{curr}]$, and whenever the current transaction is a new transaction within the same thread, then we simply copy the *thr* value from the previous transaction (line 6 in the code for "TRANSACTION BEGIN").

For *edge-e1* and *edge-e2*, we consider four exhaustive cases. A thread e can only end at a spawn point, a sync point, the return point of a spawned function instance, or the return point of the `main` function instance. If e ends at a spawn point, then lines 8–9 in the code for "FUNCTION SPAWN" correctly set *edge-e1*$[e]$ and *edge-e2*$[e]$. If e ends at a sync point, then lines 4–5 in the code for "FUNCTION SYNC" correctly set *edge-e1*$[e]$ and *edge-e2*$[e]$. If e ends at the return point of a spawned function instance, then that function instance must have an associated sync point in its parent function instance. Thus, lines 7–8 in the code for "FUNCTION SYNC" correctly set *edge-e1*$[e]$ and *edge-e2*$[e]$. Finally, if e ends at the return point of the `main` function instance, then lines 3–4 in the code for "FUNCTION RETURN" correctly set *edge-e1*$[e]$ and *edge-e2*$[e]$.

Whenever an access is made to shared memory, it is recorded into *reads-temp* or *writes-temp* by the code for "READ LOCATION" and "WRITE LOCATION". Then, at the end of a transaction, this information is transferred to *reads*, *writes*, *readers*, and *writers*, with no repeats because of the use of *processed* (lines 5–14 in the code for "TRANSACTION END"). Also, *reads-temp* and *writes-temp* are cleared at the beginning of every transaction (lines 8–9 in the code for "TRANSACTION BEGIN"), so accesses in one transaction do not get mixed with accesses in other transactions. Thus, the arrays *reads*, *writes*, *readers*, and *writers* hold the correct data at the end of part $\mathcal{A}$ of the algorithm.

5.2 Part $\mathcal{B}$: LCA and Access Interactions

Part $\mathcal{B}$ of the algorithm uses the information recorded in part $\mathcal{A}$ to compute the least common ancestors (LCA) structure of the target program and the access interaction edges. Following the same presentation plan as the previous section, this section first describes the variables used in this part of the algorithm. Next, it provides pseudocode followed by a text explanation. Then, it analyzes the running time. Finally, it gives brief arguments about correctness.

Variables. Below are the variables used in part $\mathcal{B}$ of the algorithm, excluding local dummy variables. A description in paragraph form follows.

★ Input variables:

- information about threads:
  - m ▷ number of threads
  - *edge-e1*$[0 .. m-1]$ ▷ first thread constraint
  - *edge-e2*$[0 .. m-1]$ ▷ second thread constraint

- information about transactions:
  n ▷ number of transactions
  $thr[0 .. n-1]$ ▷ enclosing thread
  $reads[0 .. n-1]$ ▷ list of memory locations read
  $writes[0 .. n-1]$ ▷ list of memory locations written

- information about shared-memory locations:
  $readers[0 .. L-1]$ ▷ list of transactions that read a location
  $writers[0 .. L-1]$ ▷ list of transactions that write a location

★ Output variables:

- information about threads:
  LCA data structure ▷ data structure used by the LCA algorithm
  $thr\text{-}edges\text{-}a[0 .. m-1]$ ▷ adjacency list of threads with which this thread has an
  ▷ access interaction; the value is the last transaction in a
  ▷ give thread with which some transaction in this thread
  ▷ has an access interaction

- information about transactions:
  $tran\text{-}edges\text{-}a[0 .. n-1]$ ▷ adjacency list of threads with whose transactions this
  ▷ transaction has an access interaction

★ Local variables:

- information about threads:
  $thr\text{-}edges\text{-}a\text{-}table[0 .. m-1][0 .. m-1]$ ▷ adjacency matrix of access interactions;
  ▷ the value is the last transaction in the
  ▷ second thread that shares an access
  ▷ interaction with the first thread

- information about transactions:
  $tran\text{-}edges\text{-}a\text{-}table[0 .. n-1][0 .. m-1]$ ▷ adjacency matrix of access interactions;
  ▷ the value is TRUE if the first transaction
  ▷ has an access interaction with some
  ▷ transaction in the second thread

The input variables to part $\mathcal{B}$ of the algorithm are simply all the output variables from part $\mathcal{A}$, which have already been described in the previous section. The output variables of part $\mathcal{B}$ are as follows.

One output item is a data structure containing the LCA information for the threads in the target program. Many algorithms have been devised that use linear time to precompute an LCA data structure which allows finding the LCA of any two vertices in constant time [1, 35, 21]. For concreteness, we shall assume that we are using the algorithm in [1].

Also, we output two arrays of adjacency lists of access interaction edges, in forms that are the most convenient for part $\mathcal{C}$ of the algorithm. The array *tran-edges-a* holds adjacency lists from transactions to threads. For each transaction t, *tran-edges-a*$[t]$ is a linked list of all the threads with which t has access interactions. That is, a thread $e_1 \in$ *tran-edges-a*$[t]$ if and only if e_1 contains a transaction t_1 such that $t \overset{\mathrm{A}}{\leftrightarrow} t_1$.

The array *thr-edges-a* holds adjacency lists from threads to threads, but with a slight twist. For any two threads e and e_1, if there is an access interaction between some transaction in e and some transaction in e_1, then the last transaction in e_1 that has an access interaction with some transaction in e appears in *thr-edges-a*$[e]$. In other words, let t_1 be a transaction in e_1; then $t_1 \in$ *thr-edges-a*$[e]$ if and only if t_1 has an access interaction with some transaction in e, but no later transaction in e_1 has an access interaction with any transaction in e.

Finally, the local variables *tran-edges-a-table* and *thr-edges-a-table* are adjacency matrices with the same information as in *tran-edges-a* and *thr-edges-a*, respectively. Part $\mathcal{B}$ of the algorithm uses these local variables as intermediate steps in computing *tran-edges-a* and *thr-edges-a*.

Pseudocode and Explanation. In part $\mathcal{B}$ of the algorithm, we first precompute the LCA data structure. Then, we find the access interactions and store them into the adjacency matrices *tran-edges-a-table* and *thr-edges-a-table*. Finally, we convert the information in *tran-edges-a-table* and *thr-edges-a-table* into the arrays *tran-edges-a* and *thr-edges-a* of adjacency lists. The pseudocode is provided below, and the text explanation follows.

```
COMPUTE-LCA-AND-EDGES-A():

LCA-PRECOMPUTE( edge-e1, edge-e2, m )

for t_i ← 0 to n − 1
    for e_j ← 0 to m − 1
        tran-edges-a-table[t_i][e_j] ← FALSE

for e_i ← 0 to m − 1
    for e_j ← 0 to m − 1
        thr-edges-a-table[e_i][e_j] ← NONE

for t_i ← 0 to n − 1
    for each ℓ ∈ writes[t_i]
        for each t_j ∈ writers[ℓ]
            CHECK-FOR-EDGE-A( t_i, t_j )
        for each t_j ∈ readers[ℓ]
            CHECK-FOR-EDGE-A( t_i, t_j )
    for each ℓ ∈ reads[t_i]
        for each t_j ∈ writers[ℓ]
            CHECK-FOR-EDGE-A( t_i, t_j )

for t_i ← 0 to n − 1
    LIST-EMPTY( tran-edges-a[t_i] )
    for e_j ← 0 to m − 1
        if tran-edges-a-table[t_i][e_j] = TRUE
            LIST-INSERT( tran-edges-a[t_i], e_j )

for e_i ← 0 to m − 1
    LIST-EMPTY( thr-edges-a[e_i] )
    for e_j ← 0 to m − 1
        if thr-edges-a-table[e_i][e_j] ≠ NONE
            LIST-INSERT( thr-edges-a[e_i], thr-edges-a-table[e_i][e_j] )
```

Check-For-Edge-a(t_i, t_j):

1 **if** $t_i < t_j$ **and** LCA-Find($thr[t_i]$, $thr[t_j]$) $\notin \{thr[t_i], thr[t_j]\}$
2 $\quad$ *tran-edges-a-table*$[t_i][\,thr[t_j]\,] \leftarrow$ TRUE
3 $\quad$ *tran-edges-a-table*$[t_j][\,thr[t_i]\,] \leftarrow$ TRUE
4 $\quad$ **if** *thr-edges-a-table*$[\,thr[t_i]\,][\,thr[t_j]\,] < t_j$
5 $\quad\quad$ *thr-edges-a-table*$[\,thr[t_i]\,][\,thr[t_j]\,] \leftarrow t_j$
6 $\quad$ **if** *thr-edges-a-table*$[\,thr[t_j]\,][\,thr[t_i]\,] < t_i$
7 $\quad\quad$ *thr-edges-a-table*$[\,thr[t_j]\,][\,thr[t_i]\,] \leftarrow t_i$

In the above pseudocode, Compute-LCA-and-Edges-a is the main function, and the subroutine Check-For-Edge-a has been abstracted out simply because it needs to be done in three places. In the main code, line 1 precomputes the LCA data structure. Lines 2–4 initialize *tran-edges-a-table* and lines 5–7 initialize *thr-edges-a-table*.

Lines 8–16 perform the task of finding the access interactions and putting this information into *tran-edges-a-table* and *thr-edges-a-table*. Line 8 iterates through all the transactions t_i. For each t_i, we need to find all the transactions t_j with which t_i has an access interaction. Transactions t_i and t_j only share an access interaction if t_i writes a location ℓ (line 9) and t_j either writes ℓ (line 10) or reads ℓ (line 12), or else if t_i reads a location ℓ (line 14) and t_j writes ℓ (line 15). In each of these cases, we call the subroutine Check-For-Edge-a to make two additional checks.

In the subroutine, line 1 checks that $t_i < t_j$, which eliminates redundancy, and line 1 also checks that the LCA of the threads to which t_i and t_j belong is not equal to either of those threads, which makes sure that t_i and t_j appear in parallel in the serial control flow of the target program. If these checks pass, then there is an access interaction between t_i and t_j, so lines 2–3 record this information into *tran-edges-a-table*, and lines 4–7 record this information into *thr-edges-a-table*. Note that the checks in lines 4 and 6 ensure that when all the access interactions have been processed, only the latest transactions that represent the end of an access interaction between two threads are recorded into *thr-edges-a-table*.

Back in Compute-LCA-and-Edges-a, lines 17–21 move information from *tran-edges-a-table* to *tran-edges-a*. Line 17 iterates through all the transactions t_i. For each t_i, line 18 initializes its adjacency list *tran-edges-a*$[t_i]$, and lines 19–21 iterate through all the threads and insert the appropriate threads into *tran-edges-a*$[t_i]$. Similarly, lines 22–26 move information from *thr-edges-a-table* to *thr-edges-a*. Line 22 iterates through all the threads e_i. For each e_i, line 23 initializes its adjacency list *thr-edges-a*$[e_i]$, and lines 24–26 iterate through all the threads and insert the appropriate transactions in the appropriate threads into *thr-edges-a*$[e_i]$.

Running Time. In Compute-LCA-and-Edges-a, line 1 requires $\Theta(m)$ time using the algorithm in [1]. Lines 2–4 use $\Theta(nm)$ time and lines 5–7 use $\Theta(m^2)$ time. Now we consider lines 8–16. Since the *reads* and *writes* lists do no have any repeats, every time either of the **for** loops at lines 9 and 14 iterates, it must be with a different value of t_i or ℓ. This shows that these two **for** loops iterate a total of $O(\alpha)$ times (recall that α is the total number of accesses to shared-memory locations). Within each iteration of the loop at line 9, we know that the *readers* and *writers* lists do not have any repeats, so that the **for** loops at lines 10 and 12 iterate a total of $O(n)$ times. Similarly, within each iteration of the loop at line 14, the **for** loop at line 15 iterates a total of $O(n)$ times. Together, we find that the subroutine Check-For-Edge-a is called a grand total of $O(\alpha) \cdot O(n) = O(n\alpha)$ times. The operations in the subroutine all take constant time per call, so the total time required for lines 8–16 (remembering that line 8 must iterate $\Theta(n)$ times) is $\Theta(n)+O(n\alpha) = O(n\alpha)$. Finally, lines 17–21 use $\Theta(nm)$ time and lines 22–26 use $\Theta(m^2)$ time.

Adding up all the sections of code, we find that part $\mathcal{B}$ of the algorithm uses a total running time of $\Theta(m) + \Theta(nm) + \Theta(m^2) + O(n\alpha) + \Theta(nm) + \Theta(m^2) = O(n\alpha + nm)$.

Correctness. We shall argue briefly that the output variables for part $\mathcal{B}$ of the algorithm have correct values at the end of part $\mathcal{B}$. The LCA data structure we use comes from [1], and its correctness is proved in that paper.

For *tran-edges-a*, assume t_1 belongs to e_1 and $t \overset{A}{\leftrightarrow} t_1$. We shall show that $e_1 \in$ *tran-edges-a*$[t]$. First, we examine what happens in lines 8–16 of Compute-LCA-and-Edges-A. By Definition 8, t and t_1 are in parallel, and both t and t_1 access some shared-memory location ℓ^*, with at least one of them writing ℓ^*. In each of the three cases determined by which of t and t_1 write ℓ^*, it is clear that the subroutine Check-For-Edge-A is called in one of lines 11, 13, or 16 with t_i and t_j equal to the smaller and larger, respectively, of t and t_1. Within the subroutine, the checks in line 1 pass, and one of lines 2 or 3 sets *tran-edges-a-table*$[t][e_1]$ to TRUE. Later, in lines 17–21 of Compute-LCA-and-Edges-A, there is an iteration of the inner loop with t_i and e_j equal t and e_1, at which point the check in line 20 passes and line 21 adds e_1 to *tran-edges-a*$[t]$, as desired.

In the reverse direction, assume that at the end of part $\mathcal{B}$, $e_1 \in$ *tran-edges-a*$[t]$. Then e_1 could only have been added to *tran-edges-a*$[t]$ in line 21 of Compute-LCA-and-Edges-A, which means the check in line 20 must have passed, so that *tran-edges-a-table*$[t][e_1] =$ TRUE. This, in turn, could only have been set by one of lines 2 or 3 in the subroutine Check-For-Edge-A. This subroutine was called in one of lines 11, 13, or 16 of Compute-LCA-and-Edges-A, and in each case, there must exist some t_1 such that *thr*$[t_1] = e_1$, t and t_1 are in parallel, and both t and t_1 access the same memory location ℓ^*, with at least one of them writing ℓ^*. Thus, the conditions are satisfied for the existence of an access interaction $t \overset{A}{\leftrightarrow} t_1$ between t and some transaction in e_1.

The argument for the correctness of *thr-edges-a* is almost the same as that for *tran-edges-a*, so we shall not write it out. We only note the single major difference. For threads e and e_1, lines 4–7 in the subroutine Check-For-Edge-A may be executed many times, once for each access interaction edge between some transaction in e and some transaction in e_1. The checks in lines 4 and 6 of the subroutine ensure that, at the end of part $\mathcal{B}$, *thr-edges-a-table*$[e][e_1]$ contains the last transaction in e_1 that shares an access interaction with some transaction in e.

5.3 Part $\mathcal{C}$: Searching for a Possible Data Race

Part $\mathcal{C}$ of the algorithm uses the data gathered and computed in the first two parts to search for a possible data race. Following the same presentation plan as the previous two sections, this section first describes the variables used in this part of the algorithm. Then, it provides pseudocode followed by a text explanation. Finally, it analyzes the running time. Arguments about correctness are given in Section 5.4.

Variables. Below are the variables used in part $\mathcal{C}$ of the algorithm, excluding local dummy variables. A description in paragraph form follows.

★ Input variables:

• information about threads:
  m ▷ number of threads

LCA data structure ▷ data structure used by the LCA algorithm
edge-e1[0 .. $m - 1$] ▷ first thread constraint
edge-e2[0 .. $m - 1$] ▷ second thread constraint
thr-edges-a[0 .. $m - 1$] ▷ adjacency list of threads with which this thread has an ▷ access interaction; the value is the last transaction in a ▷ give thread with which some transaction in this thread ▷ has an access interaction

- information about transactions:
  n ▷ number of transactions
  thr[0 .. $n - 1$] ▷ enclosing thread
  tran-edges-a[0 .. $n - 1$] ▷ adjacency list of threads with whose transactions this ▷ transaction has an access interaction

★ Output variables: *Not applicable.*

★ Local variables:

- information for breadth-first search:
  need-visit ▷ queue of threads that need visiting
  discovered[0 .. $m - 1$] ▷ keeps track of threads that have already been discovered

The input variables to part $\mathcal{C}$ of the algorithm are output variables from parts $\mathcal{A}$ and $\mathcal{B}$, so they have already been described in the previous two sections. There are no output variables since this is the final part of the algorithm.

The local variable *need-visit* is a queue used to keep track of threads that have been discovered but not yet visited in a breadth-first search of the threads in the interaction graph. The array *discovered* is used to record which threads have previously been discovered and enqueued into *need-visit*, so as to prevent visiting the same thread multiple times during the search.

Also, as shown below, part $\mathcal{C}$ of the algorithm returns a pair (t_i, t_j) of transactions when it finds a possible data race. Thus, we define a new constant NO-RACE-FOUND = (NONE, NONE) for the return value when no possible data race is found.

Pseudocode and Explanation. Part $\mathcal{C}$ of the algorithm runs n breath-first searches on the threads of the target program, starting from each of the n transactions. In each breadth-first search, the goal is to start from a transaction t_s and find a thread cycle that leads back to a later transaction t_j in the same thread as t_s. Moreover, the threads in the cycle must all be in parallel with the starting thread. The pseudocode is provided below, and the text explanation follows.

FIND-POSSIBLE-DATA-RACE():

```
for t_s ← 0 to n − 1

    for e_i ← 0 to m − 1
        discovered[e_i] ← FALSE
    discovered[ thr[t_s] ] ← TRUE
```

```
5          QUEUE-EMPTY( need-visit )
6          for each e_i ∈ tran-edges-a[t_s]
7               discovered[e_i] ← TRUE
8               QUEUE-ENQ( need-visit, e_i )

9          while QUEUE-IS-EMPTY( need-visit ) = FALSE
10              e_i ← QUEUE-DEQ( need-visit )
11              if LCA-FIND( e_i, thr[t_s] ) ∉ {e_i, thr[t_s]}

12                   for each t_j ∈ thr-edges-a[e_i]
13                        if thr[t_s] = thr[t_j] and t_s < t_j
14                             return (t_s, t_j)
15                        if discovered[ thr[t_j] ] = FALSE
16                             discovered[ thr[t_j] ] ← TRUE
17                             QUEUE-ENQ( need-visit, thr[t_j] )

18                   for each e_j ∈ {edge-e1[e_i], edge-e2[e_i]}
19                        if discovered[e_j] = FALSE
20                             discovered[e_j] ← TRUE
21                             QUEUE-ENQ( need-visit, e_j )

22    return NO-RACE-FOUND
```

The above pseudocode works as follows. The loop at line 1 iterates through all transactions t_s. Within the loop, in lines 2–21, t_s becomes the starting point for a breadth-first search on the threads of the target program. Lines 2–3 initialize the array *discovered* used to keep track of which threads have been discovered in the search, and line 4 marks the thread $thr[t_s]$ of the starting transaction as having been discovered. Line 5 initializes the queue *need-visit* used to keep track of threads that have been discovered but not yet visited. Lines 6–8 mark all the threads e_i that can be reached from the starting transaction t_s via access interaction edges as having been discovered and enqueues them into *need-visit* to be processed during the search. (We do not worry about thread edges here because we want all the threads that we search through to be in parallel with the starting thread.)

The setup for the breath-first search is now complete. Lines 9–21 repeatedly take threads off the queue and process them. The **while** loop at line 9 checks if there are still threads that need visiting. If so, then line 10 dequeues the next thread e_i to be processed. Line 11 checks to make sure that e_i is in parallel with the starting thread $thr[t_s]$.

At this point, lines 12–17 expand the breath-first search along the access interaction edges adjacent to e_i, while lines 18–21 expand the search along thread edges outgoing from e_i. The **for** loop at line 12 iterates through all the threads that can be reached from e_i via access constraint edges. For each such thread $thr[t_j]$, line 13 checks to see if it is the starting thread $thr[t_s]$, and if so, whether a thread cycle can be completed that ends at a transaction t_j which appears later in the starting thread than the starting transaction t_s. If such a thread cycle can be made, then a possible data race is reported, in the form of the pair of transactions (t_s, t_j). Also for each thread $thr[t_j]$, line 15 checks whether it has previously been discovered, and if not, lines 16–18 mark it as having been discovered and enqueue it into *need-visit* for future processing. Line 18 iterates through the threads e_j that can be reached from e_i via thread edges (there can be at most two), and for each such thread, lines 19–21 perform the same thread discovery procedure as lines 15–17.

Finally, if all n breadth-first searches beginning from the n transactions do not find a possible data race, then line 22 indicates this by returning NO-RACE-FOUND.

Running Time. The main **for** loop at line 1 iterates $\Theta(n)$ times. Within each iteration of this main loop is a breadth-first search of the threads. This breadth-first search is different in two ways from a regular breadth-first search of the threads. One difference is that it starts from a transaction (not a thread), and as the first step only considers threads that are reachable from the starting transaction via access interaction edges (not also thread edges). The other difference is that the search does not expand when it visits a thread that is not in parallel with the starting thread. Note that both of these differences can only cut down on the time required for a breadth-first search, but never increase it. Thus, we can bound the time required for our breadth-first search by the time required for a regular breadth-first search of the threads. If we let μ denote the number of pairs of threads that have access interactions between some of their transactions, then the time required for a regular breadth-first search of the threads is $\mathrm{O}(m + \mu)$. Thus, the time required for our breadth-first search is $\mathrm{O}(m + \mu)$ as well. Then, over all the iterations of the main loop, the total time required for part $\mathcal{C}$ of the algorithm is $\Theta(n) \cdot \mathrm{O}(m + \mu) = \mathrm{O}(nm + n\mu)$.

5.4 Correctness and Analysis

This section first states and proves exact conditions under which part $\mathcal{C}$ of the algorithm reports a data race. Next, it proves that the algorithm never reports any false negatives. Then, it gives an intuitive explanation of the algorithm and discusses how the programmer should use the reported information about a possible data race. Finally, it combines the analyses from Sections 5.1–5.3 to show that the total running time of the algorithm is worst-case quadratic in the size of the target program's interaction graph.

The following theorem gives the conditions under which a possible data race is reported.

Theorem 14. *Part $\mathcal{C}$ of the algorithm reports a possible data race if and only if the target program's interaction graph contains a thread cycle $C = \langle (t_0, t'_1), \ldots, (t_{k-1}, t'_0) \rangle$ such that*

1. *every thread in the cycle is in parallel with the thread containing t_0 (and t'_0), and*
2. *the starting transaction t_0 appears before the ending transaction t'_0 in their common thread.*

Proof. Forward direction. Say that part $\mathcal{C}$ of the algorithm reports a possible data race in the pair of transactions (t, t'). Looking at the pseudocode for FIND-POSSIBLE-DATA-RACE, this must have happened during the iteration of the main **for** loop (line 1) in which $t_s = t$. During this iteration, a breadth-first search is performed with t as the starting point.

Note that all threads visited during the breadth-first search are reachable from t via some thread path. Furthermore, due to the check in line 11, only threads that are in parallel with the thread containing t are expanded during the search.

Now consider the iteration of the **while** loop at line 9, and within it the iteration of the **for** loop at line 12, during which (t, t') is returned. Note that $t_j = t'$ during this iteration. From the previous paragraph, we know that the value of e_i is some thread that can be reached from t via a thread path consisting of threads that are all in parallel with the thread containing t. From line 12, we know that there is an access edge connecting e_i to t'. Finally, from line 13, we know that t and t' belong to the same thread and that t appears before t' in their common thread. Thus, the thread path from t to e_i combined with the access edge between e_i and t' forms a thread cycle C (starting at t and ending at t') in the target program's interaction graph that satisfies both conditions in the theorem statement.

Backward direction. Say that the interaction graph of the target program contains a thread cycle $C = \langle(t_0, t'_1), \ldots, (t_{k-1}, t'_0)\rangle$ that meets the two conditions in the theorem statement. Furthermore, let us assume that t'_0 is the last transaction in its thread to share an access edge with any transaction in the thread containing t_{k-1}, because otherwise we could change the last edge in C to make a different thread cycle that does meet this criterion.

If part $\mathcal{C}$ of the algorithm reports a possible data race that is different from (t_0, t'_0), then we are done. However, if part $\mathcal{C}$ does not report any other data race, then we shall prove that part $\mathcal{C}$ must report a possible data race in the pair of transactions (t_0, t'_0).

At some point in the main **for** loop at line 1, we have $t_s = t_0$. Now let the thread containing t_{k-1} be e. Since part of the thread cycle C is a thread path from t_0 to e consisting of threads that are all in parallel with the thread containing t, we know that e is processed at some point during the breadth-first search starting from t. At this point, $e_i = e$ in the **while** loop at line 9. Within this iteration of the **while** loop, at some point $t_j = t'_0$ in the **for** loop at line 12, because there is an access edge from transaction t_{k-1} in e to transaction t'_0, and t'_0 is the latest transaction in its thread for which this is true (note that the edge from t_{k-1} to t'_0 cannot be a thread edge because then e would not be in parallel with the thread containing t). It is at this time that the checks in line 13 pass and line 14 returns (t_0, t'_0). □

With the following theorem, we prove that the algorithm never reports any false negatives. (A false negative occurs when the algorithm reports NO-RACE-FOUND, but in fact the target program contains a data race.)

Theorem 15. *If the target program contains a data race, then part $\mathcal{C}$ of the algorithm reports a possible data race.*

Proof. If the target program contains a data race, then it must have a race assignment A. Let G_A be the assignment graph of A, and let $C = \langle(t_0, t'_1), \ldots, (t_{k-1}, t'_0)\rangle$ be a thread cycle in G_A with the fewest number of access edges. It must be true that for some i, transaction t_i appears before t'_i in their common thread. The reason is that, otherwise, there would be a cycle in G_A of the form $t_0 \to t'_1 \rightsquigarrow t_1 \to \cdots \rightsquigarrow t_{k-1} \to t'_0 \rightsquigarrow t_0$, where every path $t'_i \rightsquigarrow t_i$ consists simply of transaction edges within their common thread, and this cycle would contradict the fact that A is a race assignment. Now, since the thread cycle C can be rotated to start at any thread, we can assume without loss of generality that t_0 appears before t'_0 in their common thread e_0.

At this point, we consider three cases. We shall prove that only the first case can occur.

Case 1: All the threads in C are in parallel with e_0. In this case, the proof is completed by the backward direction of Theorem 14.

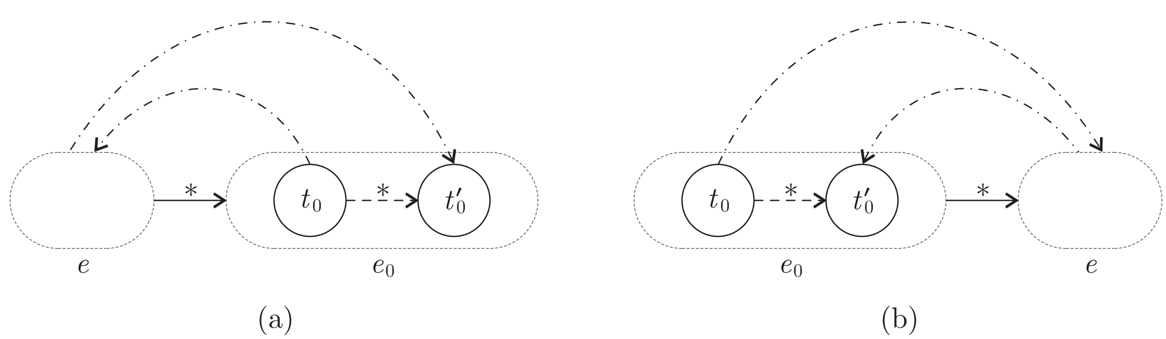

Figure 5.1: Diagrams of cases 2 and 3.

Case 2: There exists a thread e in C that is a serial ancestor of e_0. This situation is shown in Figure 5.1(a), where the dotted-and-dashed lines represent thread paths that combine to make up C, and an edge marked with an asterisk represents a path consisting of one or more edges of that type. In this case, there is a thread path from t_0 to e that is a portion of C, and a serial path from e to e_0 using only thread edges. These two paths together form a thread cycle C', which I claim has fewer access edges than C, thus contradicting the fact that C is a thread cycle with the fewest number of access edges.

The reason that C' has fewer access edges than C is as follows. Note that C' is obtained from C by removing the portion of C going from e to t'_0 and adding the serial path from e to e_0. Adding the serial path clearly does not add any access edges, so if we can prove that the path from e to t'_0 contains at least one access edge, then we are done. Note that t'_0 cannot be the first transaction in e_0 because there is at least one earlier transaction t_0. Thus, there can be no thread edge that ends at t'_0. This proves that the last edge in the path from e to t'_0, which is an edge ending at t'_0, must be an access edge.

Case 3: There exists a thread e in C that is a serial descendent of e_0. This situation is shown in Figure 5.1(b). The proof is similar to that of Case 2. We consider the thread cycle C' consisting of the portion of C going from e to t'_0 and the serial path from e_0 to e. Note that C' is obtained from C by removing the portion of C going from t_0 to e and adding the serial path from e_0 to e. Adding the serial path does not add any access edges, while removing the thread path from t_0 to e removes at least one access edge, because the first edge in this thread path is outgoing from t_0, and cannot be a thread edge because t_0 is not the last transaction in e_0. Thus, C' contains fewer access edges than C, contradicting the fact that C is a thread cycle with the fewest number of access edges. $\square$

Theorem 14 gives precise conditions for when a possible data race is reported and Theorem 15 tells us that there are no false negatives, but we still need an intuitive explanation of the algorithm, and a clarification on how to use the output information to help eliminate possible data races.

The intuition for the algorithm is as follows. We know from Chapter 4 that we (most likely) cannot efficiently detect for race assignments, which can be thought of as thread cycles in the interaction graph of the target program that do not also generate a (transaction) cycle. Thus, we are instead going to search for thread cycles that have a "reasonable chance" of not also generating a cycle. We accomplish this by eliminating from consideration all thread cycles $C = \langle (t_0, t'_1), \ldots, (t_{k-1}, t'_0) \rangle$ that can be immediately turned into a cycle of the form $t_0 \to t'_1 \rightsquigarrow t_1 \to \cdots \rightsquigarrow t_{k-1} \to t'_0 \rightsquigarrow t_0$ simply by connecting together each t'_i and t_i (if they are not already the same transaction) using transaction edges in their common thread. This is by far the most common reason for a thread cycle to also generate a cycle.

In order for a thread cycle C not to directly cause a cycle as described in the previous paragraph, some thread in C must break the potential underlying cycle by having t_i precede t'_i. Without loss of generality, we can assume t_0 precedes t'_0 in their common thread e_0. This reasoning provides the motivation for condition 2 in the statement of Theorem 14. We can think of the possible data race being exhibited as follows. If all the transactions in C are scheduled in the order that they appear in C, starting with t_0 and ending with t'_0, then this schedule would not be equivalent to any schedule of the atomic-threads version of the target program. The reason is that in any equivalent schedule of the atomic-threads atomization, due to the sequence of constraints in C, e_0 must appear before all the other threads in C as well as appear after all the other threads in C, which is clearly impossible.

This explanation of how a possible data race can be exhibited leads naturally to condition 1 in the statement of Theorem 14. We add the condition that all threads in C must be in parallel with e_0 because otherwise, the possible data race would not be able to be exhibited in the manner described above. Still, we cannot eliminate possible data races simply by intuition, so we need the support given by cases 2 and 3 in the proof of Theorem 15. The proof demonstrates that in the cases shown in Figure 5.1, the algorithm would find some other thread cycle C' that contains only a subset of the access constraint edges in C. In these situations, C' is the "root cause" of the problem demonstrated by C, and the algorithm keeps searching until it finds the thread cycle C' before reporting the problem.

Finally, we shall address how the reported information (t_0, t'_0) about C should be used to eliminate the possible data race. The programmer should ask himself or herself whether the correctness of the shared-memory accesses in t'_0 depends on the assumption that there are no intervening shared-memory accesses by other threads since t_0. If the correctness does depend on this assumption, then the programmer should use constructs in the program (which should be provided by the compiler) to force t_0 and t'_0 to be placed in the same transaction. Intuitively, this change eliminates the possibility that t_0 and t'_0 might be scheduled with intervening transactions that compromise correctness, while mathematically, this change means that e_0 no longer prevents C from directly causing a cycle. If, however, the correctness of t'_0 does not depend on shared-memory accesses made in t_0, then the programmer should use constructs in the program (which should be provided by the data-race detection system) to indicate that this possible data race is not an error and should no longer be reported.

Overall Running Time. We conclude by adding up the running times from the three parts of the algorithm to obtain the overall running time. From Section 5.1, part $\mathcal{A}$ of the algorithm requires $\Theta(\tau)$ time, where τ is the running time of the uninstrumented target program. From Section 5.2, part $\mathcal{B}$ of the algorithm requires $O(n\alpha + nm)$ time, where n is the number of transactions, m is the number of threads, and α is the number of accesses to shared-memory locations. From Section 5.3, part $\mathcal{C}$ of the algorithm requires $O(nm + n\mu)$ time, where μ is the number of pairs of threads that have access interactions between some of their transactions. Thus, the overall running time of the whole algorithm is $\Theta(\tau) + O(n\alpha + nm) + O(nm + n\mu) = O(\tau + n\alpha + nm + n\mu)$. Discounting the time required for executing the uninstrumented target program, the running time of the algorithm is worst-case quadratic in the size of the interaction graph.

Chapter 6

Conclusion

This conclusion first summarizes what I have accomplished in this thesis, and then offers some suggestions for future work.

First, I laid out a mathematical foundation for studying the constraints on transaction scheduling. I began by defining thread and transaction constraints, which are the constraints on legal schedules of an atomized program imposed by the serial control flow of the program. I then defined a notion of equivalence for schedules of the same program, from which I derived access constraints, which are the constraints that determine whether two schedules are equivalent. I extended the concept of access constraints for schedules to access interactions for atomized programs. Finally, I defined access assignments, which represent the link between access interactions and access constraints, and discussed when they are realizable. This foundation of terminology and basic facts about the constraints on transaction scheduling can be reused in future studies of the transactions-everywhere methodology, even on topics not directly related to data-race detection.

I formulated the definition of a data race in the transactions-everywhere setting. An atomized program is defined to contain a data race if there exists a schedule of it that is not equivalent to any schedule of the atomic-threads atomization. This definition is based on the weak assumption that the program has correct parallelization, which is made in the same spirit as the assumption that underlies the conventional definition of a data race. The weakness of the assumption causes the definition of a data race to become more complicated than the conventional definition. The structure of a data race also becomes more complicated, as a data race in the transactions-everywhere methodology can involve an arbitrary number of threads and transactions.

I defined race assignments and proved that the existence of a race assignment is a necessary and sufficient condition for an atomized program to contain a data race. I then used race assignments to prove, via a polynomial-time reduction from the problem of 3cnf-formula satisfiability, that data-race detection is an NP-complete problem in general. This result suggests that data races most likely cannot be detected efficiently, and that algorithms should be sought which approximately detect data races.

Finally, I presented an algorithm that approximately detects data races. The algorithm has three parts. Part $\mathcal{A}$ records the serial control flow and shared-memory accesses of the target program by instrumenting it at key points; part $\mathcal{B}$ computes the least common ancestors structure of the threads and finds the access interactions; and part $\mathcal{C}$ uses the data prepared in the first two parts to search for a possible data race. I stated and proved exact conditions under which a possible data race is reported, and showed that the algorithm never reports any false negatives. Intuitively, the algorithm searches for a thread cycle in the target program's interaction graph that does not immediately lead to a cycle, and within that thread cycle, a thread that is a primary cause of the possible data race. When it finds a possible data race, the algorithm reports two transactions that may have their correctness compromised by intervening transactions in the schedule. The programmer should then examine whether or not the two transactions have a logical data dependency. If so, then he or she would indicate to the compiler that the two transactions should be merged into the same transaction, and if not, then he or she would indicate to the data-race detector that there is in fact no error. The algorithm requires running time that is worst-case quadratic in the size of the target program's interaction graph.

At this point, I conclude by offering some suggestions for future work. The issue of how to handle data-race detection in the transactions-everywhere methodology has by no means been fully studied and solved. One basic question is whether there is a better definition of a data race that would allow data races to be efficiently detected in general, while still accurately capturing real-life data-race errors and not making extra assumptions about program correctness. Another question is whether other common techniques for detecting data races, in particular static analysis and dynamic online detection, can be fruitfully applied in the transactions-everywhere setting.

Implementation of the proposed algorithm is necessary to test its feasibility in practice. Two important factors that need to be evaluated are the rate that data races appear in programs using transactions everywhere and the rate that false positives appear. Experimentation with different heuristic rules for determining cutpoints may lead to more accurate atomization strategies for compilers that automatically generate transactions everywhere. Also, it may be possible to lower the rate of false positives by designing a better algorithm.

Related Work

Charles E. Leiserson and many graduate students in his research group designed and implemented Cilk [3, 39, 13, 22], a parallel-programming language that is a faithful extension of the C language. Mingdong Feng, Guang-Ien Cheng, Leiserson, and others developed the Nondeterminator [11, 5, 4, 38], a data-race detector for Cilk programs using conventional locks.

Other researchers have studied data-race detection in many contexts. Static data-race detectors [30] analyze the target program without executing it. Dynamic race detectors study the behavior of the target program through its execution. The latter can be categorized into those that perform detection as the target program runs [8, 9, 11, 34, 27, 12], and those that study a completed execution trace of the target program [29, 15, 10, 31].

Maurice Herlihy and J. Eliot B. Moss suggested hardware transactional memory [19], a hardware mechanism for supporting shared-memory parallel programming. Thomas F. Knight and Morry Katz presented similar ideas in the context of functional languages [24, 23]. Nir Shavit, Dan Touitou, Herlihy, Victor Luchangco, Mark Moir and others have attempted software transactional memory [36, 18], which provides transactional-memory support in software.

Leslie Lamport proposed lock-free data structures [25], and Herlihy and Moss suggested wait-free programming [16, 17], both of which are ways to implement highly concurrent shared data structures utilizing atomic instructions that are weaker than transactions.

Leiserson suggested transactions everywhere [26], a methodology for parallel programming on systems with HTM support that further eases the programmer's job beyond the benefits provided by HTM. Clément Ménier applied the transactions-everywhere methodology to sample Cilk programs and collected some statistics on the resulting transactions [28].

Bibliography

[1] Michael A. Bender and Martin Farach-Colton. "The LCA Problem Revisited". *Latin American Theoretial INformatics*, pages 88–94. April 2000. `http://www.cs.sunysb.edu/~bender/pub/lca.ps`.

[2] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently*. Ph.D. thesis, MIT Department of Electrical Engineering and Computer Science. Cambridge, Massachusetts, September 1995. `ftp://theory.lcs.mit.edu/pub/cilk/rdb-phdthesis.ps.Z`.

[3] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. "Cilk: An Efficient Multithreaded Runtime System". *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, pages 207–216. Santa Barbara, California, July 1995. `ftp://theory.lcs.mit.edu/pub/cilk/PPoPP95.ps.Z`.

[4] Guang-Ien Cheng. *Algorithms for Data-Race Detection in Multithreaded Programs.* Master's thesis, MIT Department of Electrical Engineering and Computer Science. Cambridge, Massachusetts, June 1998. `ftp://theory.lcs.mit.edu/pub/cilk/cheng-thesis.ps.gz`.

[5] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. "Detecting Data Races in Cilk Programs That Use Locks". *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*, pages 298–309. Puerto Vallarta, Mexico, June 1998. `ftp://theory.lcs.mit.edu/pub/cilk/brelly.ps.gz`.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. MIT Press, Cambridge, Massachusetts, 2001. `http://mitpress.mit.edu/algorithms/`.

[7] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. "Dynamic Perfect Hashing: Upper and Lower Bounds". *SIAM Journal on Computing*, 23(4):738-761. August, 1994. `ftp://www.mpi-sb.mpg.de/~mehlhorn/ftp/DynamicPerfectHashing.ps`.

[8] Anne Dinning and Edith Schonberg. "An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection". *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, pages 1–10. Seattle, Washington, March 1990.

[9] Anne Dinning and Edith Schonberg. "Detecting Access Anomalies in Programs with Critical Sections". *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96. ACM Press, May 1991.

[10] Perry A. Emrath, Sanjoy Ghosh, and David A. Padua. "Event Synchronization Analysis for Debugging Parallel Programs". *Proceedings of Supercomputing '89*, pages 580–588. Reno, Nevada, November, 1989. `http://www.csrd.uiuc.edu/reports/839.ps.gz`.

[11] Mingdong Feng and Charles E. Leiserson. "Efficient Detection of Determinacy Races in Cilk Programs". *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '97)*, pages 1–11. Newport, Rhode Island, June 1997. `ftp://theory.lcs.mit.edu/pub/cilk/spbags.ps.gz`.

[12] Yaacov Fenster. *Detecting Parallel Access Anomalies*. Master's thesis, Hebrew University. March 1998.

[13] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. "The Implementation of the Cilk-5 Multithreaded Language". *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223. Montreal, Quebec, Canada, June 1998. `ftp://theory.lcs.mit.edu/pub/cilk/cilk5.ps.gz`.

[14] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of* NP-*Completeness*. W. H. Freeman and Company, 1979.

[15] David P. Helmhold, Charles E. McDowell, and Jian-Zhong Wang. "Analyzing Traces with Anonymous Synchronization". *Proceedings of the 1990 International Conference on Parallel Processing*, pages II:70–77. St. Charles, Illinois, August 1990. `ftp://ftp.cse.ucsc.edu/pub/tr/ucsc-crl-89-42.ps.Z`.

[16] Maurice Herlihy. "A Methodology for Implementing Highly Concurrent Data Objects". *ACM Transactions on Programming Languages and Systems*, 15(5):745–770. November 1993. `http://www.cs.brown.edu/people/mph/Herlihy93/herlihy93methodology.pdf`.

[17] Maurice Herlihy. "Wait-Free Synchronization". *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991. `http://www.cs.brown.edu/people/mph/Herlihy91/p124-herlihy.pdf`.

[18] Maurice Herlihy, Victor Luchangco, and Mark Moir. *Obstruction-Free Software Transactional Memory for Supporting Dynamic Data Structures*. Unpublished manuscript. October 2002.

[19] Maurice Herlihy and J. Eliot B. Moss. "Transactional Memory: Architectural Support for Lock-Free Data Structures". *Proceedings of the 1993 International Symposium on Computer Architecture*, pages 289–300. San Diego, California, May 1993. `http://www.cs.brown.edu/people/mph/HerlihyM93/herlihy93transactional.pdf`.

[20] Maurice Herlihy and J. Eliot B. Moss. *Transactional Support for Lock-Free Data Structures (Technical Report 92/07)*. Digital Cambridge Research Lab. Cambridge, Massachusetts, December 1992.

[21] Dov Harel and Robert E. Tarjan. "Fast Algorithms for Finding Nearest Common Ancestors". *SIAM Journal on Computing*, 13(2):338–355. May 1984.

[22] Christopher F. Joerg. *The Cilk System for Parallel Multithreaded Computing*. Ph.D. thesis, MIT Department of Electrical Engineering and Computer Science. Cambridge, Massachusetts, January 1996. `ftp://theory.lcs.mit.edu/pub/cilk/joerg-phd-thesis.ps.gz`.

[23] Morry J. Katz. *ParaTran: A Transparent, Transaction Based Runtime Mechanism for Parallel Execution of Scheme*. Master's thesis, MIT Department of Electrical Engineering and Computer Science. Cambridge, Massachusetts, June 1986.

[24] Thomas F. Knight. "An Architecture for Mostly Functional Languages". *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 88-93. August 1986.

[25] Leslie Lamport. "Concurrent Reading and Writing". *Communications of the ACM*, 20(11):806–811. 1977.

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

[27] John Mellor-Crummey. "On-the-Fly Detection of Data Races for Programs with Nested Fork-Join Parallelism". *Proceedings of Supercomputing '91*, pages 24–33. Albuquerque, New Mexico, November 1991.

[28] Clément Ménier. *Atomicity in Parallel Programming*. Unpublished manuscript. Supercomputer Technologies Research Group, MIT Laboratory for Computer Science. Cambridge, Massachusetts, September 2002.

[29] Barton P. Miller and Jong-Deok Choi. "A Mechanism for Efficient Debugging of Parallel Programs". *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 135–144. Atlanta, Georgia, June 1988.

[30] Greg Nelson, K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Extended Static Checking home page. `http://www.research.compaq.com/SRC/esc/Esc.html`.

[31] Robert H. B. Netzer and Sanjoy Ghosh. "Efficient Race Condition Detection for Shared-Memory Programs with Post/Wait Synchronization". *Proceedings of the 1992 International Conference on Parallel Processing*. August 1992. `ftp://ftp.cs.wisc.edu/tech-reports/reports/92/tr1084.ps.Z`.

[32] Robert H. B. Netzer and Barton P. Miller. "On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions". *Proceedings of the 1990 International Conference on Parallel Processing*, pages II:93–97. St. Charles, Illinois, August 1990. `ftp://grilled.cs.wisc.edu/technical_papers/complexity.ps.Z`.

[33] Robert H. B. Netzer and Barton P. Miller. "What Are Race Conditions?" *ACM Letters on Programming Languages and Systems*, 1(1):74–88. March 1992. `http://www.cs.umd.edu/projects/syschat/raceConditions.pdf`.

[34] Itzhak Nudler and Larry Rudolph. "Tools for the Efficient Development of Efficient Parallel Programs". *Proceedings of the First Israeli Conference on Computer Systems Engineering*. May 1986.

[35] Baruch Schieber and Uzi Vishkin. "On Finding Lowest Common Ancestors: Simplification and Parallelization". *SIAM Journal on Computing*, 17(6):1253–1262. December 1988.

[36] Nir Shavit and Dan Touitou. "Software Transactional Memory". *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213. Ottawa, Ontario, Canada, 1995. `http://theory.lcs.mit.edu/~shanir/stm.ps`.

[37] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, Boston, Massachusetts, 1997. `http://www-math.mit.edu/~sipser/book.html`.

[38] Andrew F. Stark. *Debugging Multithreaded Programs that Incorporate User-Level Locking*. Master's thesis, MIT Department of Electrical Engineering and Computer Science. Cambridge, Massachusetts, May 1998. `ftp://theory.lcs.mit.edu/pub/cilk/astark-thesis.ps.gz`.

[39] Supercomputing Technologies Group, MIT Laboratory for Computer Science. *Cilk 5.3.2 Reference Manual*. Cambridge, Massachusetts, November 2001. `http://supertech.lcs.mit.edu/cilk/manual-5.3.2.pdf`.