Detecting Data Races in Parallel Program Executions

Robert H. B. Netzer
netzer@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

Computer Sciences Department
University of Wisconsin−Madison
1210 W. Dayton Street
Madison, Wisconsin 53706

Research supported in part by National Science Foundation grant CCR-8815928, Office of Naval Research grant N00014-89-J-1222, and a Digital Equipment Corporation External Research Grant.
Copyright © 1989,1990 Robert H. B. Netzer, Barton P. Miller.
To appear in *Languages and Compilers for Parallel Computing*, D. Gelernter, T. Gross, A. Nicolau, and D. Padua eds., MIT press, 1991; also appears in *Proceedings of the 3rd Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

Abstract

Several methods currently exist for detecting data races in an execution of a shared-memory parallel program. Although these methods address an important aspect of parallel program debugging, they do not precisely define the notion of a data race. As a result, is it not possible to precisely state which data races are detected, nor is the meaning of the reported data races always clear. Furthermore, these methods can sometimes generate false data race reports. They can determine whether a data race was exhibited during an execution, but when more than one data race is reported, only limited indication is given as to which ones are real. This paper addresses these two issues. We first present a model for reasoning about data races, and then present a two-phase approach to data race detection that attempts to validate the accuracy of each detected data race.

Our model of data races distinguishes among those data races that actually occurred during an execution (*actual* data races), those that could have occurred because of timing variations (*feasible* data races), and those that appeared to have occurred (*apparent* data races). The first phase of our two-phase approach to data race detection is similar to previous methods and detects a set of data race candidates (the apparent data races). We prove that this set always contains all actual data races, although it may contain other data races, both feasible and infeasible. Unlike previous methods, we then employ a second phase which validates the apparent data races by attempting to determine which ones are feasible. This second phase requires no more information than previous methods collect, and involves making a conservative estimate of the data dependences among the shared data to determine how these dependences may have constrained alternate orderings potentially exhibited by the execution. Each apparent data race can then be characterized as either being feasible, or as belonging to a set of apparent data races where at least one is feasible.

1. Introduction

In shared-memory parallel programs, if accesses to shared data are not properly coordinated, a *data race* can result, causing the program to behave in a way not intended by the programmer. Detecting data races in a particular execution of a parallel program is an important part of debugging. Several methods for data race detection have been developed[1, 3, 4, 9, 15, 17]. Although these methods provide valuable tools for debugging, they do not precisely define the notion of a data race. As a result, we cannot precisely state which data races are detected by these methods. In addition, false data race reports can sometimes be generated. These methods can determine whether or not a data race occurred, but when more than one data race is reported, no indication is given as to which ones are real. Failure to characterize the detected data races, and the generation of false data race reports, can make it difficult to use these methods for debugging the program and locating the cause of the data races. This paper addresses these two issues. We first present a formal model in which to reason about data races, and then outline a two-phase approach to data race detection that is discussed entirely in terms of the model. The first phase performs essentially the same type of analysis as previous methods, and detects a set of candidate data races (which we call *apparent* data races). Unlike previous methods, we then employ a second phase that validates each of the apparent data races to determine which ones are real. By providing a model for reasoning about data races, the correctness of our techniques can be convincingly argued, and the meaning of the data race reports (generated by our methods or others) is made explicit. By validating the apparent data races, the programmer can be provided with information crucial to debugging.

One purpose of adding explicit synchronization to shared-memory parallel programs is to coordinate accesses to shared data. Some programs are intended to behave deterministically, and for these programs synchronization is usually designed to force all shared-data accesses to the same location to occur in a specific order (for some given program input). When the order of two shared-memory accesses made by different processes (to the same location) is not enforced, a *race condition* is said to exist[5, 6], possibly resulting in a nondeterministic execution. In contrast, other programs are not intended to be deterministic, and for these programs synchronization is usually added to ensure that some sections of code execute as if they were atomic (i.e., to implement critical sections). For example, consider a section of code that adds up a list of shared data representing the deposits to a bank account during a certain month. If this section of code does not execute as if it were atomic (because, for example, another section of concurrently executing code is debiting the account), the computed deposit total might not be correct. A section of code is guaranteed to execute atomically if the shared variables it reads and modifies are not modified by any other concurrently executing section of code[2]. If these conditions are not met, a *data race* is said to exist. Since nondeterministic behavior can result, a data race is a special case of the more general race condition. In this paper we focus on data race detection.

To provide a mechanism for reasoning about data race detection, we present a model for representing executions of shared-memory parallel programs, on sequentially consistent processors, that use fork/join and counting semaphores. Our model distinguishes between the ordering of events that *actually* occurred during execution and the ordering that *could have* occurred. Given an actual execution of the program, we characterize alternate event orderings that the execution could have exhibited. Possible orderings include those that could still allow the original data dependences among the shared data to occur and that do not violate the semantics of the explicit synchronization primitives used by the program. An execution exhibiting such an alternate ordering is called a *feasible program execution*. The characterization of feasible program executions in general requires knowledge of which shared-data dependences (if any) were exhibited between any two events performed by the execution. Since recording this information is not practical in general, we characterize approximate information in terms of our model. We show how the information recorded by previous methods can be used to define an *approximate program execution*. We then distinguish between three types of data races. *Actual* data races are those actually exhibited during an execution, *feasible* data races are those that could have occurred because of nondeterministic timing variations, and *apparent* data races are those that appeared to have occurred from analyzing the approximate information. Previous methods detect apparent data races. We show that apparent data races are not always actual or feasible, and show how a two-phase approach can be used to detect and then validate the apparent data races. The first phase is essentially identical to previous methods and simply detects the apparent data races. We prove that each actual data race is also apparent. The approach employed by previous methods is therefore safe in the sense that no actual data races are left undetected. We also employ a second phase that classifies each apparent data race as either feasible or as belonging to a set of data races that contains at least one feasible data race. Performing such a validation provides the programmer with some information as to which of the apparent data races should be investigated for debugging.

2. Previous Data Race Detection Methods

All previous methods for dynamic data race detection operate by first instrumenting the program so that information about its execution is recorded, and then executing the program and analyzing the collected information. These methods all analyze essentially the same information about the execution, but differ mainly in how and when that information is collected and analyzed. Two approaches to this information collection and analysis have been proposed: *on-the-fly* and *post-mortem*. On-the-fly techniques[4, 9, 17] detect data races by an on-going analysis during execution that encodes information about the execution so it can be accessed quickly and discarded as it becomes obsolete. Post-mortem techniques[1, 3, 15] detect data races after execution ends by analyzing trace files that are produced during execution. Although all previous methods never fail to detect any data races actually exhibited during execution (we prove this claim in Section 7), they do not precisely locate where these data races occurred. We briefly describe the common characteristics of these methods below.

Previous methods instrument the program to collect the same information: which sections of code executed, the set of shared variables read and written by each section of code, and the relative execution order between some synchronization operations. To represent this relative ordering, a DAG is constructed (either explicitly or in an encoded form), which we call the *ordering graph*, in which each node represents an execution instance of either a synchronization operation (a *synchronization event*) or the code executed between two synchronization operations (a *computation event*)†. Edges are added from each event to the next event belonging to the same process, and between some pairs of synchronization events (belonging to different processes) to indicate the order in which the synchronization events executed. The various methods differ in the types of synchronization handled, but they all handle **fork**/**join** (in one form or another). An edge is added from each **fork** event to the first event in each child created by the **fork**, and from the last event in each child to the corresponding **join** event.

The crux of data race detection is the location of events that accessed a common shared variable (that at least one wrote) and that either did or could have executed concurrently. Finding events that accessed a common shared variable is straightforward, since the sets of shared variables read and written by each event is recorded. To determine if two events could have executed concurrently, all previous methods analyze the ordering graph. Two computation events are assumed to have potentially executed concurrently if no path in the graph connects the two events. Data races are therefore reported between pairs of events that accessed a common shared variable and that have no connecting path. However, this assumption is not always true, and causes previous methods to generate potentially many false data race reports.

To illustrate these false reports, consider the program fragment in Figure 1. This program creates two identical children that remove from a shared queue the lower and upper bounds of a region of a shared array to operation upon, perform some computation on that region of the array, and loop until the queue is empty. The queue initially contains records representing disjoint regions of the array. A correct execution of this program should therefore exhibit no data races, since only disjoint regions of the shared array should be accessed concurrently. However, assume that the ‘‘remove’’ operations do not properly synchronize their accesses to the shared queue. An ordering graph for one possible execution of this program is also shown (the dotted lines only illustrate the data races and are not part of the graph). In this execution, the first ‘‘remove’’ operation performed by the left child completed before the first ‘‘remove’’ performed by the right child began (the nodes are staggered horizontally to indicate this order). The first two records were therefore correctly removed, and both children operated (correctly) on disjoint regions of the array. However, during the next iteration, the ‘‘remove’’ operations actually overlapped, and the right child correctly removed the fourth record, but the left child removed the upper bound (100) from the third record and the lower bound (300) from the fourth record. The left child therefore operated (erroneously) on region [100,299] of the array.

In this graph, no paths connect any nodes of the left child with any nodes of the right child. Since both children accessed the same queue, previous methods would report four data races between the ‘‘remove’’ operations (shown by the finely dotted lines). Similarly, since both children accessed a common region of the array, a data race would also be reported between these array accesses (shown by the coarsely dotted line). The latter data race report can be misleading, however, since the accesses by the left child to region [100,299] did not, and could never, execute concurrently with the accesses to region [200,299] made by the right child. For these accesses to execute concurrently, the second ‘‘remove’’ operation performed

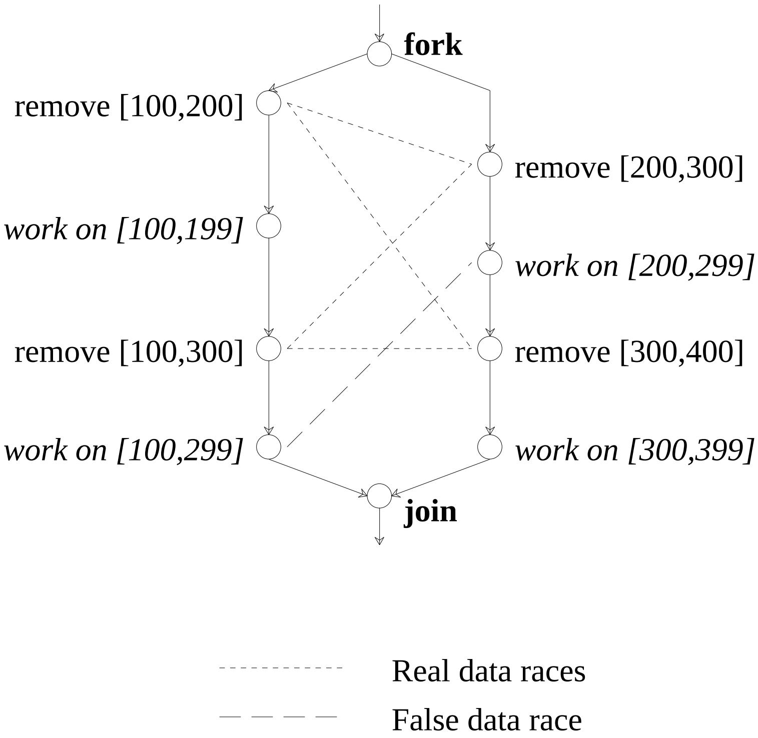
† Some methods do not actually construct a node to represent a computation event but rather represent the event by an edge connecting the two surrounding synchronization events[4, 15].

fork

```
loop
  remove (L,U)
  work on region [L,U–1]
      of shared array
while QueueNotEmpty

loop
  remove (L,U)
  work on region [L,U–1]
      of shared array
while QueueNotEmpty
```

join



Initial state of Queue:

[100,200]
[200,300]
[1,100]
[300,400]

Figure 1. Example program fragment and ordering graph (the dotted lines only illustrate the reported data races)

---

by the left child would have to execute before the second ‘‘remove’’ operation performed by the right child (with which it originally overlapped). If this would happen, the erroneous record [100,300] would not be removed by the left child, since it would no longer overlap with the other ‘‘remove’’ operation, and a different region of the array would be accessed.

If the array accesses were more complex, perhaps creating other children, there may have been many nodes in the graph representing these accesses. In such a case, many false data race reports would be generated, instead of only one. In this example, the data races are caused by lack of synchronization in the ‘‘remove’’ operations. The fact that non-disjoint regions of the array were accessed is an artifact of this missing synchronization, and does not represent a bug in the program. Reporting many false data races to the programmer, only one of which involves events that did (or could) execute concurrently, complicates the job of debugging.

False data race reports can result whenever shared variables are used (either directly or transitively) in conditional expressions or in expressions determining which shared locations are accessed (e.g., shared array subscripts). Accurate data race detection involves examining how shared data flowed through the execution and whether the execution might have changed had a different ordering occurred. This paper presents results showing how to validate the accuracy of each data race without recording additional information about the execution.

3. Program Execution Model

Before discussing data race detection, we first present a formal model to provide a mechanism for reasoning about shared-memory parallel program executions. The model contains the objects that represent a program execution (such as which statements were executed and in what order), and axioms that characterize properties those objects must possess. This model is useful as a notational device for describing behavior the execution *actually* exhibited. We later show how it can also be used to speculate on behavior that the execution *could have* exhibited (such as alternate event orderings) due to nondeterministic timing variations. Our model describes programs that use counting semaphores and the fork/join construct.

3.1. General Model

Our model is based on Lamport's theory of concurrent systems[13], which provides a formalism for reasoning about concurrent systems that does not assume the existence of atomic operations. In Lamport's formalism, a concurrent system execution is modeled as a collection of *operation executions*. Two relations on operation executions, *precedes* ($\rightarrow$) and *can causally affect* ($\dashrightarrow$), describe a system execution; $a \rightarrow b$ means that a completes before b begins (in the sense that the last action of a can affect the first action of b), and $a \dashrightarrow b$ means that some action of a precedes some action of b. We use Lamport's theory, but restrict it to the class of shared-memory parallel programs that execute on sequentially consistent processors[11].

When the underlying hardware guarantees sequential consistency, any two events that execute concurrently can affect one another (i.e., $a \not\leftrightarrow b \Leftrightarrow a \dashrightarrow b \wedge b \dashrightarrow a$).† Given sequential consistency, a single relation is sufficient to describe the temporal aspects of a system execution. For this purpose we introduce $\xrightarrow{T}$, the *temporal ordering* relation among events; $a \xrightarrow{T} b$ means that a completes before b begins, and $a \overset{T}{\not\leftrightarrow} b$ means that a and b execute concurrently (i.e, neither completes before the other begins). We should emphasize that we are defining the temporal ordering relation so it describes the order in which events *actually* executed during a particular execution; e.g., $a \overset{T}{\not\leftrightarrow} b$ means that a and b actually executed concurrently, and does not mean that a and b could have executed in any order. In Section 5 we show how to speculate on alternate temporal orderings that could have been exhibited.

In addition, we replace the $\dashrightarrow$ relation with the *transitive shared-data dependence* relation (or just *shared-data dependence* relation for brevity), $\xrightarrow{D}$. This relation shows when one event can causally affect another either because of a *direct* data dependence involving a single shared variable, or because of a chain of direct dependences involving several different variables. A direct shared-data dependence from a to b (denoted $a \xrightarrow{DD} b$) exists if a accesses a shared variable that b later accesses (where at least one access modifies the variable); we also say that a direct dependence exists if a precedes b in the same process, since data can in gen-

---

† In Lamport's terminology, we are considering the class of system executions that have global-time models. Throughout this paper, we use superscripted arrows to denote relations, and write $a \not\rightarrow b$ as a shorthand for $\neg(a \rightarrow b)$, and $a \not\leftrightarrow b$ as a shorthand for $\neg(a \rightarrow b) \wedge \neg(b \rightarrow a)$.

eral flow through non-shared variables which are local to the process. A transitive shared-data dependence ($a \xrightarrow{D} b$) exists if there is a chain of direct dependences from a to b (possibly involving events that access different shared variables); $\xrightarrow{D} = (\xrightarrow{DD})^+$, the irreflexive transitive closure of the direct shared-data dependence relation‡. This definition of data dependence is different from the standard ones[10] since we consider transitive dependences involving flow-, anti-, and output-dependences, and do not explicitly state the variables involved.

We define a *program execution*, P, to be a triple, $\langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$, where E is a finite set of *events*, and $\xrightarrow{T}$ and $\xrightarrow{D}$ are the relations over E described above. We refer to a given program execution, P, as an *actual program execution* when P represents an execution that the program at hand actually performed. Each event $e \in E$ represents the execution of a set of program statements, and possesses two attributes, $READ(e)$ and $WRITE(e)$, the set of shared variables read and written by those statements. A *data conflict* is said to exist between two events if one events writes a shared variable that the other reads or writes. The temporal ordering and shared-data dependence relations must satisfy the following axioms:

A1. $\xrightarrow{T}$ is an irreflexive partial order.

A2. If $a \xrightarrow{T} b \overset{T}{\not\leftrightarrow} c \xrightarrow{T} d$ then $a \xrightarrow{T} d$.

A3. If $a \xrightarrow{D} b$ then $b \overset{T}{\nrightarrow} a$.

No generality is lost by modeling each event, e, as having a unique *start time* (e_s) and *finish time* (e_f)[12]. A total ordering on the start and finish times is called a *global-time model*. Given a global-time model, the $\xrightarrow{T}$ relation is defined as follows: $a \xrightarrow{T} b$ iff $a_f < b_s$, and $a \overset{T}{\not\leftrightarrow} b$ iff $a_s < b_f \wedge b_s < a_f$. Axioms A1-A3 can also be written in terms of the start and finish times. We will occasionally employ such a view in proofs of our results.

3.2. Model Applied to Semaphores and Fork/Join

So far, the model does not describe any of the synchronization aspects of a program execution. By imposing some structure on the set of events, E, and by adding axioms that describe the semantics of synchronization operations, we extend the general model to describe programs that use counting semaphores and the fork/join construct. Other types of synchronization can be similarly accommodated.

We assume that a program execution consists of a number of processes, each of which either exists when the program execution begins or is created during execution by a *fork* operation. Similarly, a process either continues to exist until the program execution ends or until the process (and all others created by the same fork operation) is terminated by a *join* operation. The set of events belonging to process p is denoted by E_p, and therefore $E = \bigcup_p E_p$, for all

‡ The transitive shared-data dependence relation is conservative in the sense that when data flows from a to b it always shows a dependence from a to b, but also sometimes shows a dependence when in fact no data flow occurs. A more precise characterization of causality would require examining the semantics of the individual actions performed by each event.

processes p that exist during the program execution. Each process is viewed as containing a totally ordered sequence of events, and the term $e_{p,i}$ will denote the i^{th} event in the execution of process p. The following axiom describes the total ordering imposed on events belonging to the same process:

A4. $e_{p,i} \xrightarrow{\text{T}} e_{p,i+1}$ for all processes p and $1 \le i < |E_p|$

To describe the presence of synchronization operations, we distinguish between different types of events. A *synchronization event* is an instance of a semaphore operation (a *P event* or a *V event*), a fork operation (a *fork event*), or a join operation (a *join event*). The set of all P and V operations on semaphore i is denoted by $E_{P(i)}$ and $E_{V(i)}$, respectively. A *computation event* is an instance of a group of statements, belonging to the same process, that executed consecutively, none of which are synchronization operations. Any arbitrary grouping of (consecutively executed) statement instances that does not include a synchronization operation defines a computation event.

To describe the semantics of synchronization operations, we add additional axioms to our model. A fork event, $Fork_{p,i}$, is assumed to precede all events in the child processes which it creates, and all events in these child processes are assumed to precede the subsequent join event in process p, $Join_{p,i+k}$:

A5. For all child processes, c, created by each $Fork_{p,i}$ and terminated at $Join_{p,i+k}$,

$$Fork_{p,i} \xrightarrow{\text{T}} e_{c,j} \xrightarrow{\text{T}} Join_{p,i+k} \quad 1 \le j \le |E_c|$$

We assume that in any program execution the semaphore invariant[7] is always maintained. For counting semaphores, the semaphore invariant is maintained iff at each point in the execution, the number of V operations that have either completed or have begun executing is greater than or equal to the number of P operations that have completed. For each semaphore, S, this invariant can be expressed by the following axiom:

A6. For every subset of P events, $P \subseteq E_{P(S)}$,

$$|\ \{\ v \ \mid\ v \in E_{V(s)} \ \wedge\ \exists p \in P \ \ (v \xrightarrow{\text{T}} p \ \vee\ v \overset{\text{T}}{\not\longleftrightarrow} p)\ \}\ | \ \ \ge\ \ |P|.$$

The above version of axiom A6 assumes that the initial value of each semaphore is zero. An arbitrary initial value, m, for some semaphore could be described by creating an artificial process that contains m V-events that precede all other events.

3.3. Higher-Level Views

It is useful to be able to view a program execution at different levels of abstraction, since information about the execution may be collected at that level, and because sometimes it is useful to abstract irrelevant details of part of an execution into a higher-level event. We can reason about a program execution at any level of abstraction by following Lamport and defining a *higher-level view*. A higher-level view of a program execution $P = \langle E, \xrightarrow{\text{T}}, \xrightarrow{\text{D}} \rangle$ is $\mathcal{P} = \langle \mathcal{E}, \xrightarrow{\mathcal{T}}, \xrightarrow{\mathcal{D}} \rangle$ where

(H1) $\mathcal{E}$ partitions E, and $\forall e' \in \mathcal{E}$,

$$READ(e') = \bigcup_{e \in e'} READ(e), \text{ and } WRITE(e') = \bigcup_{e \in e'} WRITE(e).$$

(H2) $A \xrightarrow{\mathcal{T}} B \iff \forall a \in A,\ b \in B\ (a \xrightarrow{\mathrm{T}} b)$, and

(H3) $A \xrightarrow{\mathcal{D}} B \iff \exists a \in A,\ b \in B\ (a \xrightarrow{\mathrm{D}} b)$.

A higher-level view always obeys axioms A1-A3. Since axioms A4-A6 are defined in terms of synchronization and computation events, they are also obeyed if each higher-level event consists of either a single synchronization event from E, or only a set of computation events from E. In such a case, each event $e' \in \mathcal{E}$ inherits its type from the type of the events comprising e'. When the higher-level events are defined to partition E in this way, $\mathcal{P}$ obeys axioms A1-A6 and is then itself a program execution.

4. Representing Actual Program Executions

The model we have presented so far captures complete information about a program execution in the sense that $\xrightarrow{\mathrm{T}}$ shows the relative ordering in which any two events actually executed, and $\xrightarrow{\mathrm{D}}$ shows the actual shared-data dependences between any two events. In practice, recording such complete information is not practical, and we now discuss how to represent partial information about a program execution in our model. Our intent is not to discuss details of program instrumentation, but rather to outline one type of information that is sufficient for data race detection, and show how this information is represented in our model. We will define *approximate* counterparts to the $\xrightarrow{\mathrm{T}}$ and $\xrightarrow{\mathrm{D}}$ relations capturing partial information about a program execution that is based on the type of information previous methods record. This information can be recorded without tracing every shared-memory access and without introducing a central bottleneck into the program. The resulting approximate relations, $\xrightarrow{\hat{\mathrm{T}}}$ and $\xrightarrow{\hat{\mathrm{D}}}$, define an *approximate program execution*, $\hat{P} = \langle E, \xrightarrow{\hat{\mathrm{T}}}, \xrightarrow{\hat{\mathrm{D}}} \rangle$. In Section 7 we show how $\hat{P}$ can be used for data race validation.

4.1. Approximate Temporal Ordering

As described in Section 2, previous data race detection methods record the temporal ordering among only some synchronization events. For example, the order among **fork** and **join** operations and their child processes is recorded, but the relative order of operations belonging to the different child processes is not. Recording incomplete ordering information is desirable because the required instrumentation can be embedded into the implementation of the synchronization operations without introducing additional synchronization. Not introducing additional synchronization ensures that the instrumentation will not create a central bottleneck which could reduce the amount of parallelism achievable by the program. We assume that the program is instrumented to record such incomplete ordering information, and that the ordering is represented by constructing a graph, called the *temporal ordering graph*, similar to the ordering graphs used by previous methods. For every event, e, this graph contains two nodes†, $\mathbf{e_s}$ and $\mathbf{e_f}$ (corresponding to the start and finish of e), and an edge from $\mathbf{e_s}$ and $\mathbf{e_f}$. The graph defines an

† In practice, it is not always necessary to actually construct two nodes per event, but we use such a representation here since it conceptually follows our model.

approximate temporal ordering relation, $\xrightarrow{\hat{T}}$, as follows: $a \xrightarrow{\hat{T}} b$ iff there is a path from $\mathbf{a_f}$ to $\mathbf{b_s}$, $b \xrightarrow{\hat{T}} a$ iff there is a path from $\mathbf{b_f}$ to $\mathbf{a_s}$, and $a \overset{\hat{T}}{\longleftrightarrow}\!\!\!\!\!\!/\;\;\; b$ otherwise. We assume the program instrumentation constructs a temporal ordering graph that gives $\xrightarrow{\hat{T}}$ the following properties:

(1) If $a \xrightarrow{\hat{T}} b$ then $a \xrightarrow{T} b$, and

(2) If $a \overset{\hat{T}}{\longleftrightarrow}\!\!\!\!\!\!/\;\;\; b$ then the explicit synchronization performed by the execution did not prevent a and b from executing concurrently.

The first property states that the ordering of events given by $\xrightarrow{\hat{T}}$ must be consistent with the order in which they actually executed (i.e., $\xrightarrow{\hat{T}} \subseteq \xrightarrow{T}$). The second property means that any linear ordering of the graph is a global-time model defining a temporal ordering that obeys the synchronization axioms (i.e., axioms A4-A6). Recall that $\xrightarrow{T}$ was defined to represent the actual order in which events executed; $a \overset{T}{\longleftrightarrow}\!\!\!\!\!\!/\;\;\; b$ means that a and b actually overlapped during execution. Since $\xrightarrow{\hat{T}}$ is an approximation of $\xrightarrow{T}$ (it is a subset), $a \overset{\hat{T}}{\longleftrightarrow}\!\!\!\!\!\!/\;\;\; b$ does not (necessarily) mean that a and b actually overlapped. Instead, it means that a and b were not forced to occur in a certain order by *explicit* synchronization (the graph does not contain enough information to determine their actual execution order). As illustrated in Figure 1, such events may nonetheless be ordered. The goal of data race validation (discussed in Section 7) is determining which events that are not ordered by $\xrightarrow{\hat{T}}$ could have indeed executed concurrently.

For programs using **fork**/**join**, the ordering graphs constructed by previous methods satisfy the above two properties. To accommodate semaphore synchronization, edges can be added to the temporal ordering graph by recording the order of all operations on a given semaphore. Such an ordering can be recorded without introducing additional synchronization into the program (as mentioned above for **fork**/**join** operations). To reflect this ordering in the graph, an edge can be drawn from each semaphore operation (on a given semaphore) to the next operation on the same semaphore. Since $\xrightarrow{\hat{T}}$ only needs to obey the synchronization axioms, other approaches for adding edges (that result in more events being unordered by $\xrightarrow{\hat{T}}$, allowing more data races to be detected) are possible. For example, previous methods that handle semaphores[3, 4, 15] construct edges only from a **V** operation to the **P** operation it allowed to proceed. More sophisticated approaches have also been investigated[8].

4.2. Approximate Shared-Data Dependences

Determining the actual shared-data dependences exhibited by an execution would in general require the relative order of all shared memory accesses to be recorded. However, in Section 2 we mentioned that previous methods record the *READ* and *WRITE* sets for each computation event. By using only these sets and the approximate temporal ordering, the actual shared-data dependences can be conservatively estimated. The *approximate shared-data dependence* relation, $\xrightarrow{\hat{D}}$, is defined by speculating on what the actual shared-data dependences might have been. Consider two events, a and b, that both access a common shared variable (where at least one access is a modification). If $a \xrightarrow{\hat{T}} b$, then there is a direct shared-data dependence from a

to b. When $a \overset{\hat{T}}{\not\longleftrightarrow} b$, the direction of any direct dependence cannot be determined (since the actual temporal ordering between a and b is not known), and we make the conservative assumption that a direct dependence exists from a to b and from b to a. This assumption is conservative since it will always include the actual direct dependences, although it may introduce a dependence from b to a when in fact the only dependence was from a to b. The $\xrightarrow{\hat{D}}$ relation is then defined as the irreflexive transitive closure (see Section 3.1) of this approximation of the direct dependences. As we will see, for data race validation $\xrightarrow{\hat{D}}$ only needs to be computed between events a and b when $a \overset{\hat{T}}{\not\longleftrightarrow} b$.

5. Characterizing Alternate Temporal Orderings

An actual program execution describes aspects of how the program *actually* performed, and does not contain any information regarding what the program might have done. For example, $\xrightarrow{T}$ is defined to represent the *actual* temporal ordering in which any two events were performed. In a given program execution, the temporal ordering between some events is not always enforced by (explicit or implicit) synchronization, but sometimes occurs by chance. It is possible that another execution of the program could perform exactly the same events, but exhibit a different temporal ordering among these events. In this section we characterize such alternate temporal orderings that an actual program execution, P, *could have* exhibited because of nondeterministic timing variations. To determine how much the temporal ordering of P can be disturbed without affecting the events performed, we consider the shared-data dependences exhibited by P. Any execution exhibiting these same dependences is capable of performing the same events. We later use this characterization of alternate orderings to distinguish between data races actually exhibited by an execution and data races that could have been exhibited.

For a given execution of a program, consider a particular view of the execution (called a *single-access* view) in which each computation event is defined to comprise at most one shared-memory access. The program execution describing this view, P_S, shows the data dependences among the individual shared-memory accesses made by the execution. These *single-access* shared-data dependences uniquely characterize the events performed. Since the execution outcome of each statement instance depends only upon the values of the variables it reads[14], the single-access dependences uniquely determine the program state at each step in each process.† Any temporal ordering that could still allow these data dependences to occur (and that would not violate the semantics of the synchronization operations) is an ordering the execution could have exhibited. Therefore, any other single-access program execution, $P_S{}'$, possessing the same events and (single-access) shared-data dependences as P_S, represents an execution the program could actually exhibit, regardless of how its temporal ordering differs from that of P_S.

---

† For this statement to hold, interactions with the external environment must be modeled as shared-data dependences.

Similarly, this result also holds for higher-level views of the program execution. In higher-level views, computation events can consist of many shared-memory accesses. In the following theorem we show that any higher-level program execution possessing the same events and (higher-level) shared-data dependences as P describes an execution the program could actually exhibit, regardless of how its temporal ordering differs from that of P. We call a program execution that could actually be exhibited a *feasible program execution*. The following theorem gives sufficient conditions for a program execution to be feasible.

Theorem 5.1.

Let $P = \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$ be an actual program execution. $P' = \langle E, \xrightarrow{T'}, \xrightarrow{D'} \rangle$ is a feasible program execution if

(F1) P' is a valid program execution (axioms A1-A6 are satisfied), and

(F2) $\xrightarrow{D'} \; = \; \xrightarrow{D}$.

Proof.

We will use the result mentioned above that any single-access program execution possessing the same (single-access) shared-data dependences as those that actually occurred represents an execution the program could exhibit[14]. This theorem extends the result to higher-level program executions. Since computation events in higher-level program executions can consist of more than one shared-memory access, there may be more than one single-access program execution for which P, the actual program execution, is a higher-level view. Therefore, given a higher-level view, we do not always know which shared-data dependences actually occurred at the single-access level. To show that P' is a feasible program execution, we must show that it is a higher-level view of a single-access program execution possessing the actual single-access dependences. However, since these dependences are not known, we will show that the shared-data dependences exhibited by *each* single-access program execution described by P are also exhibited by *some* single-access execution described by P'. We will then be guaranteed that, no matter which single-access shared-data dependences were exhibited during P, an execution capable of exhibiting those same dependences is described by P'.

The single-access program executions that are described by P is given by the set $\{ P_S = \langle E_S, \xrightarrow{T_S}, \xrightarrow{D_S} \rangle \mid P$ is a higher-level view of $P_S \}$, and the single-access executions described by P' is given by $\{ P_S' = \langle E_S, \xrightarrow{T_S'}, \xrightarrow{D_S'} \rangle \mid P'$ is a higher-level view of $P_S' \}$. We must prove that each $\xrightarrow{D_S}$ is equal to some $\xrightarrow{D_S'}$. We first show that for any pair of higher-level events, we can always find some $\xrightarrow{D_S'}$ exhibiting the same shared-data dependences as any $\xrightarrow{D_S}$ among the lower-level events comprising these events, and then show that this guarantees some $\xrightarrow{D_S'}$ exists exhibiting the same dependences as any $\xrightarrow{D_S}$ among *all* the lower-level events comprising the actual program execution (which shows $\xrightarrow{D_S} \; = \; \xrightarrow{D_S'}$).

First, consider any P_S, and its (single-access) shared-data dependences among the lower-level events $a_S \in a$ and $b_S \in b$ comprising any two higher-level events a and b. We now show a P_S' exists exhibiting these same dependences. Since each lower-level event comprises at most one shared-memory access, it suffices to show that some P_S' exists such that $b_S \overset{T_S'}{\not\longrightarrow} a_S$ whenever $a_S \xrightarrow{D_S} b_S$, and $a_S \overset{T_S'}{\not\longrightarrow} b_S$ whenever $b_S \xrightarrow{D_S} a_S$, for any $a_S \in a$ and $b_S \in b$.

Case (1): $a \xrightarrow{T} b$ and $a \xrightarrow{T'} b$. In this case, $\xrightarrow{D_S}$ can only contain shared-data dependences from some a_S to some b_S, and all $P_S{}'$ have $a_S \xrightarrow{T'} b_S$ for all $a_S \in a$ and $b_S \in b$.

Case (2): $a \xrightarrow{T} b$ and $a \overset{T'}{\not\longleftrightarrow} b$. As with case (1), $\xrightarrow{D_S}$ can only contain shared-data dependences from some a_S to some b_S. Some $P_S{}'$ must exist in which $b_S \overset{T_S{}'}{\not\longrightarrow} a_S$ for all $a_S \in a$ and $b_S \in b$, since otherwise $b_S \xrightarrow{T_S{}'} a_S$ for all $a_S \in a$ and $b_S \in b$ would imply $b \xrightarrow{T'} a$, contradicting the assumption $a \overset{T'}{\not\longleftrightarrow} b$.

Case (3): $a \xrightarrow{T} b$ and $b \xrightarrow{T'} a$. In this case, $\xrightarrow{D_S}$ can contain no shared-data dependences between any a_S and b_S (or else P' would violate A3).

Case (4): $a \overset{T}{\not\longleftrightarrow} b$ and $a \xrightarrow{T'} b$. Since $\xrightarrow{D_S}$ can contain shared-data dependences only from some a_S to some b_S (or else P' would violate A3), this case is analogous to case (1).

Case (5): $a \overset{T}{\not\longleftrightarrow} b$ and $a \overset{T'}{\not\longleftrightarrow} b$. In this case, $\xrightarrow{D_S}$ can contain shared-data dependences in both directions between the a_S and b_S, and since the set of single-access program executions described by P' contains all possible temporal orderings among the a_S and b_S that cause $a \overset{T'}{\not\longleftrightarrow} b$, some $P_S{}'$ clearly exists with the desired properties.

Finally, we show that each $\xrightarrow{D_S}$ equals some $\xrightarrow{D_S{}'}$. Notice that when there are events a and b that overlap, P' describes more than one single-access program execution. These single-access program executions contain all possible (legal) temporal orderings among the lower-level events $a_S \in a$ and $b_S \in b$ that cause a and b to overlap. The set of all single-access program executions described by P' can be constructed by choosing, for each pair of higher-level events a and b, one such temporal ordering among the lower-level events comprising a and b. We showed above that for any $\xrightarrow{D_S}$, some $\xrightarrow{D_S{}'}$ exists exhibiting the same shared-data dependences among the lower-level events comprising any pair of higher-level events. Using this result, we can always find a $\xrightarrow{D_S{}'}$ exhibiting the same shared-data dependences as any $\xrightarrow{D_S}$ among *all* the lower-level events by independently considering each pair of higher-level events. Therefore, each $\xrightarrow{D_S}$ is equal to some $\xrightarrow{D_S{}'}$, which proves the theorem. ■

Given an actual program execution, P, we are not attempting to predict the behavior of the program had different shared-data dependences occurred. Instead, the above theorem characterizes different program executions (performing the same events as P) that we can *guarantee* the program is capable of exhibiting. Indeed, there may be program executions that violate the above conditions but nevertheless perform the same events. However, characterizing such program executions requires analyzing the semantics of the program itself, to determine what effects different shared-data dependences would have on P.

6. Definition of Data Race

We can now characterize different types of data races in terms of our model. We distinguish between an *actual data race*, which is a data race exhibited during an actual program execution, and a *feasible data race*, which is a data race that could have been exhibited because of timing variations. We also characterize the *apparent data races*, those data races detected by searching the ordering graph for data-conflicting events that are not ordered by the graph (which are the data races reported by previous methods). As discussed in the next section, the problem of data race validation is determining which apparent data races are feasible.

Definition 6.1

A *data race under* $\xrightarrow{T}$ exists between a and b iff

(DR1) a data conflict exists between a and b, and

(DR2) $a \overset{T}{\not\longleftrightarrow} b$.

Definition 6.2

An *actual data race* exists between a and b iff

(AR1) $P = \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$ is an actual program execution, and

(AR2) a data race under $\xrightarrow{T}$ exists between a and b.

Definition 6.3

A *feasible data race* exists between a and b iff

(FR1) there exists some feasible program execution, $P' = \langle E, \xrightarrow{T'}, \xrightarrow{D'} \rangle$, and

(FR2) a data race under $\xrightarrow{T'}$ exists between a and b.

Definition 6.4

An *apparent data race* exists between a and b iff

(AP1) $\hat{P} = \langle E, \xrightarrow{\hat{T}}, \xrightarrow{\hat{D}} \rangle$ is an approximate program execution, and

(AP2) a data race under $\xrightarrow{\hat{T}}$ exists between a and b.

7. Detecting Data Races

We now present our two-phase approach to data race detection. In the first phase, the apparent data races are located by using the approximate information collected about the execution to construct and then analyze the temporal ordering graph. This first phase performs the same type of analysis as previous data race detection methods. Unlike previous methods, we then employ a second phase to validate each apparent data race by attempting to determine whether or not the race is feasible. This determination is made by first augmenting the temporal ordering graph with additional edges representing a conservative estimate of the shared-data dependences, and then analyzing the resulting graph for cycles. Such a two-phase approach has the advantage that approximate information (such as that recorded by previous methods) can be used, but the programmer can still be provided with information regarding the feasibility of the reported data races. Throughout the remainder of this section, $\hat{P} = \langle E, \xrightarrow{\hat{T}}, \xrightarrow{\hat{D}} \rangle$ will denote an approximate program execution, and $P = \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$ will denote the actual program execution (which is unknown).

7.1. Phase I: Detecting Apparent Data Races

The first phase of our data race detection method identifies the apparent data races. The apparent data races are located by first constructing the temporal ordering graph and then searching the graph for pairs of data-conflicting events, *a* and *b*, whose nodes have no connecting path (implying that $a \overset{\hat{T}}{\not\longleftrightarrow} b$). In general, these data races include all actual data races, plus additional races, both feasible and infeasible. This phase cannot distinguish among these types of races since doing so would require knowledge of the complete temporal ordering.

Because the apparent data races are detected using an approximate temporal ordering, not all apparent data races are always actual or feasible. Figure 1 showed an example of an apparent data race that was not feasible. However, we now prove that each actual data race is also an apparent data race. The naive method of simply reporting all apparent data races to the user (which is the approach of previous methods) is therefore safe in the sense that no actual data races are left undetected.

Theorem 7.1.

Every actual data race is also an apparent data race.

Proof.

If there is an actual data race between *a* and *b*, then $a \overset{T}{\not\longleftrightarrow} b$. To show that there is an apparent data race between *a* and *b*, we must show that $a \overset{\hat{T}}{\not\longleftrightarrow} b$. By definition, the temporal ordering graph is constructed so that $a \overset{\hat{T}}{\longrightarrow} b \Rightarrow a \overset{T}{\longrightarrow} b$ (see Section 3). We must show that this implies $a \overset{T}{\not\longleftrightarrow} b \Rightarrow a \overset{\hat{T}}{\not\longleftrightarrow} b$. Consider that the contrapositive of the assumption is $a \overset{T}{\not\longrightarrow} b \Rightarrow a \overset{\hat{T}}{\not\longrightarrow} b$, which is equivalent to $a \overset{T}{\longrightarrow} b \vee a \overset{\hat{T}}{\not\longrightarrow} b$, or $a \overset{T}{\longrightarrow} b \vee b \overset{\hat{T}}{\longrightarrow} a \vee a \overset{\hat{T}}{\not\longleftrightarrow} b$. But if $a \overset{\hat{T}}{\longrightarrow} b \Rightarrow a \overset{T}{\longrightarrow} b$, then this becomes $a \overset{T}{\longrightarrow} b \vee b \overset{T}{\longrightarrow} a \vee a \overset{\hat{T}}{\not\longleftrightarrow} b$, or $\neg(a \overset{T}{\not\longleftrightarrow} b) \vee a \overset{\hat{T}}{\not\longleftrightarrow} b$, which is equivalent to $a \overset{T}{\not\longleftrightarrow} b \Rightarrow a \overset{\hat{T}}{\not\longleftrightarrow} b$. ■

Note that the proof of Theorem 7.1 does not make use of the specifics of how the temporal ordering graph is constructed. Indeed, any approximate temporal ordering, $\overset{\hat{T}}{\longrightarrow}$, with the property $a \overset{\hat{T}}{\longrightarrow} b \Rightarrow a \overset{T}{\longrightarrow} b$ is sufficient to allow all actual data races to be detected as apparent data races. However, the more exhaustively the program is traced, the more accurately the apparent data races can be validated, as is shown below. As we will also show below, apparent data races have the property that the presence of an apparent data race implies that there is a feasible data race somewhere in the program execution, implying that when no actual data races occur, no apparent data races will be reported.

7.2. Phase II: Validating Apparent Data Races

The first phase of our data race detection method locates a set of apparent data races. We now outline the second phase, which validates each apparent data race by attempting to determine whether or not the race is feasible. This determination is made by first augmenting the temporal ordering graph with edges representing a conservative estimate of the actual shared-data dependences, and then searching the augmented graph for certain types of cycles. Each apparent data race can be characterized either as being feasible, or as belonging to a set of apparent data races where at least one is feasible.

To show that an apparent data race between a and b is feasible, we must guarantee that some feasible program execution, $P' = \langle E, \xrightarrow{T'}, \xrightarrow{D} \rangle$, exists such that $a \not\xleftrightarrow{T'} b$. To determine the feasibility of a program execution requires knowledge of $\xrightarrow{D}$, the shared-data dependences exhibited by the observed execution. When only an approximate program execution is available, however, the exact shared-data dependences are not known. By using the conservative estimate of these dependences, $\xrightarrow{\hat{D}}$, we can guarantee that some feasible program executions must exist. We augment the temporal ordering graph with edges, called *shared-data dependence edges*, representing this conservative estimate. Let G be the temporal ordering graph. We construct the *augmented temporal ordering graph*, G_{AUG}, by augmenting G with edges that ensure there is a path from $\mathbf{a_s}$ to $\mathbf{b_f}$ whenever $a \xrightarrow{\hat{D}} b$. These edges ensure that any possible shared-data dependence from a to b would be allowed to occur in certain program executions defined by G_{AUG} (shown below in the proof of Theorem 7.2). If $a \xrightarrow{\hat{T}} b$, then a path from $\mathbf{a_s}$ to $\mathbf{b_f}$ already exists. Edges are therefore added only when $a \not\xleftrightarrow{\hat{T}} b$. In this case, edges are added from $\mathbf{a_s}$ to $\mathbf{b_f}$ and from $\mathbf{b_s}$ to $\mathbf{a_f}$ if there is a data conflict between a and b, or if a has a data conflict with some other event c that also has a data conflict with b and $a \not\xleftrightarrow{\hat{T}} c \not\xleftrightarrow{\hat{T}} b$.

In general, G_{AUG} may contain cycles, due to the conservative approximation made about the actual shared-data dependences. By classifying the apparent data races into those that participate in cycles and those that do not, some apparent data races can be guaranteed to be feasible. We say that two events, a and b, are *tangled* if either $\mathbf{a_s}$ and $\mathbf{b_f}$, or $\mathbf{b_s}$ and $\mathbf{a_f}$, belong to the same strongly connected component† of G_{AUG}. A *tangled data race* is an apparent data race between two tangled events. Each strongly connected component defines a set of tangled data races, called a *tangle*. We now show (in Theorem 7.2) that any apparent data race between two events that are not tangled is guaranteed to be feasible. We then show (in Theorem 7.3) that in each tangle, at least one of the apparent data races is guaranteed to be feasible.

Lemma 7.1.

For a given execution, assume $P = \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$ and $\hat{P} = \langle E, \xrightarrow{\hat{T}}, \xrightarrow{\hat{D}} \rangle$ are the associated complete and approximate program executions. Let G be the temporal ordering graph defining $\xrightarrow{\hat{T}}$, and let G_{AUG-D} be G augmented with edges representing the actual shared-data dependences, $\xrightarrow{D}$. Any linear ordering of the nodes of G_{AUG-D} is a global-time model that defines a temporal ordering relation, $\xrightarrow{T'}$, such that $P' = \langle E, \xrightarrow{T'}, \xrightarrow{D} \rangle$ is a feasible program execution.

Proof.

We introduce G_{AUG-D} as a device for showing that certain feasible program executions must exist. G_{AUG-D} is identical to G_{AUG}, except that edges representing shared-data depen-

† A strongly connected component has the property that there is path from every node in the component to every other node, but no path from a node in one component to a node in another component and back.

dences that did not actually occur do not appear (they were conservatively added to G_{AUG} so that no actual shared-data dependences were missed). Even though we do not have enough information to construct G_{AUG-D}, it nonetheless exists, and in Theorems 7.2 and 7.3 we prove that it must possess certain properties. In this Lemma, we prove that any linear ordering of the nodes of this graph can be used to define a feasible program execution. Any such linear ordering defines a temporal ordering that satisfies the conditions for feasibility. The shared-data dependence constraint (axiom A3) is satisfied since G_{AUG-D} contains shared-data dependence edges representing the actual shared-data dependences. The synchronization constraints (axioms A4-A6) are satisfied since G_{AUG-D} contains at least as many edges as G, and by definition, any linear ordering of G obeys axioms A4-A6.

Let L be any linear ordering of the nodes of G_{AUG-D}, and let $\xrightarrow{T'}$ be the temporal ordering defined by L. We first show that $\xrightarrow{T'}$ satisfies axioms A1-A6:

A1. The $\xrightarrow{T'}$ relation is irreflexive, since if $a \xrightarrow{T'} a$ for some event a, then $\mathbf{a_f}$ would have to appear before $\mathbf{a_s}$ in L, which is not possible, since by definition G contains an edge from $\mathbf{b_s}$ to $\mathbf{b_f}$ for every event b. By the same argument, the $\xrightarrow{T'}$ relation is asymmetric. The $\xrightarrow{T'}$ relation is transitive, since if $a \xrightarrow{T'} b \wedge b \xrightarrow{T'} c$, then $\mathbf{a_f}$ appears before $\mathbf{b_s}$ in L, and $\mathbf{b_f}$ appears before $\mathbf{c_s}$. Since an edge exists from $\mathbf{b_s}$ to $\mathbf{b_f}$ for every event b, it follows that $\mathbf{a_f}$ appears before $\mathbf{c_s}$ in L, implying that $a \xrightarrow{T'} c$.

A2. Assume a, b, c, d exist such that $a \xrightarrow{T'} b \not\xleftrightarrow{T'} c \xrightarrow{T'} d$. By the definition of $\xrightarrow{T'}$, L must contain the nodes $\mathbf{a_f}$ $\mathbf{b_s}$ $\mathbf{c_f}$ $\mathbf{d_s}$, in this order, implying that $a \xrightarrow{T'} d$.

A3. Since G_{AUG-D} contains an edge from $\mathbf{a_s}$ to $\mathbf{b_f}$ whenever $a \xrightarrow{D} b$, $\mathbf{a_s}$ will precede $\mathbf{b_f}$ in L, implying that $b \not\xrightarrow{T'} a$.

A4-A6. Because G_{AUG-D} contains no fewer edges than G, axioms A4 through A6 are satisfied since, by definition, any linear ordering of G obeys these axioms.

Since $\xrightarrow{T'}$ satisfies axioms A1-A6, $P' = \langle E, \xrightarrow{T'}, \xrightarrow{D} \rangle$ is a feasible program execution. ■

Theorem 7.2.

If there is an apparent data race between a and b, and a and b are not tangled, then the data race is feasible.

Proof.

We first show that G_{AUG} contains no path from $\mathbf{a_f}$ to $\mathbf{b_s}$, and no path from $\mathbf{b_f}$ to $\mathbf{a_s}$, and then show that this implies the apparent data race between a and b is feasible.

To show that G_{AUG} contains no path from $\mathbf{a_f}$ to $\mathbf{b_s}$, and no path from $\mathbf{b_f}$ to $\mathbf{a_s}$, we will establish a contradiction by assuming there is a path from $\mathbf{a_f}$ to $\mathbf{b_s}$, or a path from $\mathbf{b_f}$ to $\mathbf{a_s}$. Since a and b are not tangled, only one such path can exist. Assume the path from $\mathbf{a_f}$ to $\mathbf{b_s}$ exists. Since an apparent data race exists between a and b, G_{AUG} contains shared-data dependence edges from $\mathbf{a_s}$ to $\mathbf{b_f}$, and from $\mathbf{b_s}$ to $\mathbf{a_f}$. But these edges create the path $\mathbf{a_f}$ $\mathbf{b_s}$ $\mathbf{a_f}$ in G_{AUG}, implying that $\mathbf{a_f}$ and $\mathbf{b_s}$ belong to the strongly connected component, which cannot be true since a and b are not tangled. Therefore, there can be no path from $\mathbf{a_f}$ to $\mathbf{b_s}$, and no path from $\mathbf{b_f}$ to $\mathbf{a_s}$.

We finally show that the apparent data race between a and b is feasible. Consider the graph G_{AUG-D}, constructed by augmenting the temporal ordering graph, G, with edges representing the shared-data dependences that were actually exhibited by the program execution (see the proof of Lemma 7.1). This graph contains no more edges than G_{AUG}, since the edges in G_{AUG} represent the conservative estimate of the actual shared-data dependences. Since G_{AUG} cannot contain a path from $\mathbf{a_f}$ to $\mathbf{b_s}$, or a path from $\mathbf{b_f}$ to $\mathbf{a_s}$, G_{AUG-D} cannot contain such paths either. There is thus a linear ordering of the nodes of G_{AUG-D} in which $\mathbf{a_s}$ appears before $\mathbf{b_f}$, and $\mathbf{b_s}$ appears before $\mathbf{a_f}$. This linear ordering is a global-time model that defines a temporal ordering, $\xrightarrow{T'}$, such that $a \not\xleftrightarrow{T'} b$. By Lemma 7.1, $P' = \langle E, \xrightarrow{T'}, \xrightarrow{D} \rangle$ is a feasible program execution. Therefore, the apparent data race between a and b is feasible. ■

The above theorem shows that the apparent data races between events that are not tangled are guaranteed to be feasible. Not all of the remaining apparent data races are infeasible, however. We now show that, in each tangle, at least one tangled data race is guaranteed to be feasible. Without more precise knowledge of the actual shared-data dependences (or without examining the semantics of the program execution), we cannot determine exactly which tangled data races are feasible.

Lemma 7.2.

Let G be a temporal ordering graph, let G_{AUG} be G augmented with edges representing the conservative estimate of the actual shared-data dependences, and let G_{AUG-D} be G augmented with edges representing the actual shared-data dependences (see the proof of Lemma 7.1). Assume T is a set of tangled events defined by a strongly connected component of G_{AUG}. Then there exists two events $a, b \in T$ such that an apparent data race exists between a and b, and no path from $\mathbf{a_f}$ to $\mathbf{b_s}$, or from $\mathbf{b_f}$ to $\mathbf{a_s}$, exists in G_{AUG-D}.

Proof.

Let $\mathbf{T}$ be the set of nodes in G_{AUG-D} representing the events in T. To establish a contradiction, assume that for all events $a, b \in T$ such that there is an apparent data race between a and b, there is either a path from $\mathbf{a_f}$ to $\mathbf{b_s}$, or a path from $\mathbf{b_f}$ to $\mathbf{a_s}$, in G_{AUG-D}. Since a path from $\mathbf{a_f}$ to $\mathbf{b_s}$ and a path from $\mathbf{b_f}$ to $\mathbf{a_s}$ cannot both exist (G_{AUG-D} is acyclic), assume that the path from $\mathbf{a_f}$ to $\mathbf{b_s}$ exists. Since there is an apparent data race between a and b, no such path exists in G. The path in G_{AUG-D} must therefore contain at least one shared-data dependence edge, which cannot emanate from $\mathbf{a_f}$. This path must contain nodes for two events, c and d, such that there is a path from $\mathbf{a_f}$ to $\mathbf{c_s}$, a shared-data dependence edge from $\mathbf{c_s}$ to $\mathbf{d_f}$, and a path from $\mathbf{d_f}$ to $\mathbf{b_s}$. Such a path implies that $a \xrightarrow{T} c$ and $d \xrightarrow{T} b$. Furthermore, c and d must belong to T, since $\mathbf{T}$ contains a strongly connected component.

The shared-data dependence edge from $\mathbf{c_s}$ to $\mathbf{d_f}$ exists either because there is a data conflict between c and d (and therefore also an apparent data race), or because c data conflicts with some other event that data conflicts with d (a *transitive* data conflict). Assume that the edge exists because of an apparent data race between c and d. Since c and d belong to T, our contradiction assumption implies that there must be a path from $\mathbf{c_f}$ to $\mathbf{d_s}$. By applying the above argument to c and d, we conclude that the path from $\mathbf{c_f}$ to $\mathbf{d_s}$ must contain nodes for two events, $e, f \in T$, such that there is a path from $\mathbf{c_f}$ to $\mathbf{e_s}$, a shared-data dependence edge from $\mathbf{e_s}$ to

$\mathbf{f_f}$, and a path from $\mathbf{f_f}$ to $\mathbf{d_s}$. Such a path implies that $c \xrightarrow{T} e$ and $f \xrightarrow{T} d$. Since $a \xrightarrow{T} c$ and $d \xrightarrow{T} b$, the events e and f must be different than c and d. By inductively applying the above argument, we find that we always need two more events, x and y, belonging to T, that are different than all other events in T. Since T is finite, we eventually arrive at a contradiction.

If the shared-data dependence edge exists from $\mathbf{c_s}$ to $\mathbf{d_f}$ because of a transitive data conflict between c and d, event c must participate in an apparent data race with some event e that has a (possibly transitive) data conflict with d. By applying an argument similar to the one above to c and e, we also arrive at a contradiction. Therefore, two events, $a, b \in T$, must exist such that there is an apparent data race between a and b, and there is no path from $\mathbf{a_f}$ to $\mathbf{b_s}$, and no path from $\mathbf{b_f}$ to $\mathbf{a_s}$, in G_{AUG-D}. ■

Theorem 7.3.

Let G_{AUG} be an augmented temporal ordering graph, and let T be the set of tangled events defined by some strongly connected component of G_{AUG}. At least one of the apparent data races in T is feasible.

Proof.

Let G_{AUG-D} be the temporal ordering graph augmented with edges representing the actual shared-data dependences (see the proof of Lemma 7.1). By Lemma 7.2, there exists two events $a, b \in T$ such that there is an apparent data race between a and b, and there is no path from $\mathbf{a_f}$ to $\mathbf{b_s}$, and no path from $\mathbf{b_f}$ to $\mathbf{a_s}$, in G_{AUG-D}. By the argument at the end of the proof of Theorem 7.2, there is a feasible program execution, $P' = \langle E, \xrightarrow{T'}, \xrightarrow{D} \rangle$, such that $a \overset{T'}{\not\longleftrightarrow} b$, showing that the apparent data race between a and b is feasible. Therefore, at least one of the tangled data races is feasible. ■

For each tangle, the above theorem guarantees that at least one tangled data race in the tangle is always feasible. As illustrated in Section 2, however, not all of the tangled data races are always feasible. An infeasible tangled data race exists only when the outcome of one tangled data race *affects* another tangled data race. A data race between a and b can affect a data race between c and d if (1) a or b modifies a shared variable, V, and (2) either the shared locations accessed by c or d, or the presence of c or d, depend upon V. The presence of c or d can depend upon V if the outcome of some conditional statement depends upon V, and the outcome might either delay the execution of c or d, or cause c or d to not execute at all. This notion is similar to the *hides* relation of Allen and Padua[1]. A future paper will describe how to employ these ideas to locate tangled data races that can be guaranteed feasible.

8. Conclusion

This paper has addressed two issues regarding data race detection. We first presented a formal model for reasoning about data races, and then presented a two-phase approach to data race detection that validates the accuracy of each detected data race. Our model distinguished among the data races that actually occurred (actual data races), that could have occurred (feasible data races), and that appeared to have occurred (apparent data races). Such a model allowed us to characterize the type of data races detected by previous methods, and to develop and argue the correctness of our two-phase approach. The first phase of this approach is essentially identical to previous methods and detects the apparent data races. We proved that all actual data races are detected by this phase. Unlike previous methods, we then employed a second phase that validates the apparent data races. This phase augments the temporal ordering graph with edges representing a conservative estimate of the shared-data dependences. An apparent data race is validated by determining whether the events involved in the race belong to the same strongly connected component. We proved that each apparent data race involving two events belonging to different strongly connected components (or none at all) is feasible, and in each set of races belonging to a strongly connected component, at least one is feasible.

We are currently investigating several issues related to this work. First, we are developing more precise analyses for locating those apparent data races that are feasible. As mentioned in Section 7, tangled data races are infeasible only when one tangled data race affects the outcome (or the existence) of another. By examining when one event can affect another, the notion of a feasible program execution can be extended to characterize what an execution could have done had different shared-data dependences occurred. Using this extended notion of feasibility, certain tangled data races can be shown to be feasible. Second, we are examining different classes of feasible data races. We have proven that the problem of detecting all feasible data races is NP-hard[16] (even when the complete program execution is known). However, certain classes of feasible data races can be efficiently detected. Third, we are developing techniques for providing efficient data race detection in practice. These techniques include efficient program instrumentation, and algorithms for actually constructing, augmenting and analyzing the temporal ordering graph. For example, it is not necessary to model each event with two nodes in the temporal ordering graph. By appropriately modeling the end of one event as the start of the next event (in the same process), only one node per event is required. We are also investigating techniques for efficiently recording the *READ* and *WRITE* sets for each computation event. In addition, even though we presented a two-phase scheme, data race validation does not necessarily require a post-mortem approach. It may be possible to perform the validation phase on-the-fly. Finally, the ideas presented in this paper can be applied to shared-memory parallel programs that use synchronization primitives other than semaphores, such as event variables, barriers, or rendezvous. To gain practical experience with these ideas, we are currently incorporating them into a parallel program debugger[3, 15] under development at the University of Wisconsin–Madison.

Acknowledgements

This research was supported in part by National Science Foundation grant CCR-8815928, Office of Naval Research grant N00014-89-J-1222, and a Digital Equipment Corporation External Research Grant.

References

1. Allen, T. R. and D. A. Padua, ‘‘Debugging Fortran on a Shared Memory Machine,’’ *Proc. of Intl. Conf. on Parallel Processing*, pp. 721-727 St. Charles, IL, (Aug. 1987).

2. Bernstein, A. J., ‘‘Analysis of Programs for Parallel Processing,’’ *IEEE Trans. on Electronic Computers* **EC-15**(5) pp. 757-763 (Oct. 1966).

3. Choi, J.-D., B. P. Miller, and R. H. B. Netzer, ''Techniques for Debugging Parallel Programs with Flowback Analysis,'' *Comp. Sci. Dept. Tech. Rep. #786*, Univ. of Wisconsin-Madison, (Aug. 1988).

4. Dinning, A. and E. Schonberg, ''An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection,'' *Proc. of ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 1-10 Seattle, WA, (Mar. 1990).

5. Emrath, P. A. and D. A. Padua, ''Automatic Detection Of Nondeterminacy in Parallel Programs,'' *Proc. of the Workshop on Parallel and Distributed Debugging*, pp. 89-99 Madison, WI, (May 1988). Also *SIGPLAN Notices* **24**(1) (Jan. 1989).

6. Emrath, P. A., S. Ghosh, and D. A. Padua, ''Event Synchronization Analysis for Debugging Parallel Programs,'' *Supercomputing '89*, pp. 580-588 Reno,NV, (Nov. 1989).

7. Habermann, A. N., ''Synchronization of Communicating Processes,'' *Communications of the ACM* **12**(3) pp. 171-176 (Mar. 1972).

8. Helmbold, D. P., C. E. McDowell, and J.-Z. Wang, ''Analyzing Traces with Anonymous Synchronization,'' *Proc. of Intl. Conf. on Parallel Processing*, St. Charles, IL, (Aug. 1990).

9. Hood, R., K. Kennedy, and J. Mellor-Crummey, ''Parallel Program Debugging with On-the-fly Anomaly Detection,'' *Supercomputing '90*, New York, NY, (Nov. 1990).

10. Kuck, D. J., R. H. Kuhn, B. Leasure, D. A. Padua, and M. Wolfe, ''Dependence Graphs and Compiler Optimizations,'' *Conf. Record of the 8th ACM Symp. on Principles of Programming Languages*, pp. 207-218 Williamsburg, VA, (Jan. 1981).

11. Lamport, L., ''How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs,'' *IEEE Trans. on Computers* **C-28**(9) pp. 690-691 (Sep. 1979).

12. Lamport, L., ''Interprocess Communication,'' *SRI Technical Report*, (Mar. 1985).

13. Lamport, L., ''The Mutual Exclusion Problem: Part I — A Theory of Interprocess Communication,'' *Journal of the ACM* **33**(2) pp. 313-326 (Apr. 1986).

14. Mellor-Crummey, J. M., ''Debugging and Analysis of Large-Scale Parallel Programs,'' *Ph.D. Thesis, also Comp. Sci. Dept. Tech. Rep. 312*, Univ. of Rochester, (Sep. 1989).

15. Miller, B. P. and J.-D. Choi, ''A Mechanism for Efficient Debugging of Parallel Programs,'' *Proc. of the Conf. on Programming Language Design and Implementation*, pp. 135-144 Atlanta, GA, (June 1988). Also *SIGPLAN Notices* **23**(7) (July 1988).

16. Netzer, R. H. B. and B. P. Miller, ''On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions,'' *Proc. of Intl. Conf. on Parallel Processing*, St. Charles, IL, (Aug. 1990).

17. Nudler, I. and L. Rudolph, ''Tools for the Efficient Development of Efficient Parallel Programs,'' *Proc. of 1st Israeli Conf. on Computer System Engineering*, (1988).