Improving the Accuracy of Data Race Detection

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

Abstract

For shared-memory parallel programs that use explicit synchronization, *data race* detection is an important part of debugging. A data race exists when concurrently executing sections of code access common shared variables. In programs intended to be data race free, they are sources of nondeterminism usually considered bugs. Previous methods for detecting data races in executions of parallel programs can determine when races occurred, but can report many data races that are artifacts of others and not direct manifestations of program bugs. Artifacts exist because some races can cause others and can also make false races appear real. Such artifacts can overwhelm the programmer with information irrelevant for debugging. This paper presents results showing how to identify non-artifact data races by *validation* and *ordering*.

Data race validation attempts to determine which races involve events that either did execute concurrently or could have (called *feasible data races*). We show how each detected race can either be guaranteed feasible, or when insufficient information is available, sets of races can be identified within which at least one is guaranteed feasible. Data race ordering attempts to identify races that did not occur only as a result of others. Data races can be partitioned so that it is known whether a race in one partition may have affected a race in another. The *first* partitions are guaranteed to contain at least one feasible data race that is not an artifact of any kind. By combining validation and ordering, the programmer can be directed to those data races that should be investigated first for debugging.

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.

1. Introduction

In shared-memory parallel programs, programmers often coordinate access to shared data by using explicit synchronization to implement critical sections, which are intended to execute as if they were atomic. A section of code executes atomically if the shared variables it reads and modifies are not modified by any other concurrently executing code. If the program's synchronization ever fails to ensure this atomicity, it can behave unexpectedly. A *data race* exists when two sections of code both execute concurrently and access common shared variables (and at least one is modified). Even though programs containing critical sections are often intended to be nondeterministic, a data race is a source of nondeterminism that is usually considered to be a manifestation of a program bug. In this paper we present techniques that aid the programmer in debugging the cause of data races exhibited by an execution of a shared-memory parallel program. We show how to locate those data races that are *direct manifestations* of program bugs, instead of artifacts of other data races.

Data race reports generated by most existing methods[1, 3, 4, 6, 8] can include potentially many artifacts, which can overwhelm the programmer with irrelevant information. Such data race artifacts stem from two sources. First, even when two events in an execution are not forced to occur in a specific order by explicit synchronization, it still might be impossible for them to execute concurrently. Data dependences among shared data accessed by another data race may order these two events, preventing them from ever executing concurrently and constituting a data race. Second, since a data race between two events may have resulted in the atomicity failure of either of the events, subsequent program behavior different from any data-race-free execution of the program may result. Subsequent data races may therefore only be artifacts of the earlier data races that caused atomicity to fail. Existing methods simply report data races among any two events that are not ordered by explicit synchronization. These methods can determine whether or not at least one data race occurred, but when more than one data race is reported, little indication is given as to which data races are directly caused by program bugs, and which are artifacts of other data races. In this paper we present results that show how to locate data races that are not artifacts of others.

To address the first cause of data race artifacts, we extend our previous work[7] on *validating* the data races detected by existing methods. Existing methods operate by instrumenting the program so that information about its execution is recorded, such as the sets of shared variables read and written by each event (an event represents all code executed between two consecutive synchronization operations in some process), and the relative order in which some events execute. Since the ordering among only some events is recorded, it is not always known which events actually executed concurrently. These methods assume that if two events were not forced to occur in a specific order by explicit synchronization, then they could have executed concurrently. Data race reports containing artifacts are a consequence of not considering how inter-process interactions involving shared data constrain these alternate orderings.

Data race validation attempts to determine which data races either occurred during the observed execution or had the potential of occurring because of timing variations (i.e., data races involving events that either did or could have actually executed concurrently). Our previous work is extended by analyzing how the events performed during execution affected each other, allowing the effects of an alternate ordering on the program's behavior to be determined. Under an alternate ordering, shared variable accesses may occur in a different order, possibly changing the outcome of the execution. The discrimination power of our technique depends on the accuracy of information about how events affected each other. We show that this information can be approximated to varying degrees of accuracy, by performing analysis on the execution-time trace data collected by existing methods, and perhaps additionally by performing a static analysis of the program.

To address the second cause of data race artifacts, we present new results showing how to *order* the data races to locate those that could not have been affected by atomicity failures caused by previous data races. Since the events involved in a data race may execute non-atomically, the program may subsequently behave unexpectedly in ways that otherwise would not have been possible. If this behavior includes another data race, this second race may be an artifact of the first, and would never have occurred had the first race not existed. By applying a combination of data race validation and ordering, the data race reports generated by existing methods can be refined, and the programmer can be directed to those races of interest, providing information crucial for debugging.

The remainder of this paper starts by first presenting an example that illustrates why existing methods can report data race artifacts (Section 2). To reason about both artifact and non-artifact data races, Section 3 presents a formal model for representing the behavior exhibited by a program execution. Using this model, we characterize alternate orderings the execution had the potential of exhibiting, by analyzing how events in the execution affected each other (Section 4). In Section 5.1 we review our earlier results on data race validation and extend them using this characterization of alternate orderings. Section 5.2 contains new results regarding data race ordering. Finally, Section 6 concludes the paper.

2. Existing Data Race Detection Methods

To illustrate the need for data race validation and ordering, we show how most existing methods for dynamic data race detection operate on an example program execution. One proposed method addresses the need to locate first data races[2]; we postpone discussing this method until the end of Section 5, after our terminology and results have been presented.

Existing data race detection methods[1,3,4,6,8] operate by first instrumenting the program so that information about its execution is recorded, and then executing the program and analyzing the collected information. Although these methods differ in how and when this information is collected and analyzed (there are *on-the-fly* and *post-mortem* approaches), all analyze essentially the same information about the execution: which sections of code executed, the sets 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 (explicitly or in an encoded form), which we call the *ordering graph*, in which nodes represent execution instances of either synchronization operations (*synchronization events*) or code executed between two synchronization operations (*computation events*)†. Edges are added from each event to the next event in the same process, and between some pairs of synchronization events (belonging to different processes) to indicate their relative execution order. Various types of synchronization are handled; all methods handle some form of **fork/join**. Edges are added from a **fork** event to the first event in each created child, and from the last event in each child to the **join** event.

The crux of existing methods 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 accessed by each event are recorded. To determine if two events could have executed concurrently, all existing methods analyze the ordering graph, and assume that two events could have executed concurrently if no path

† Some methods represent computation events by the intra-process edges, instead of constructing a separate node[3,6].

connects the two events. Data races are therefore reported between any events that accessed common shared variables and that have no connecting path. However, this approach can report potentially many data race artifacts.

To illustrate why artifacts can be reported, consider the program fragment in Figure 1. This program creates two children that execute in parallel. Each starts by performing some initial work on disjoint regions of a shared array, and then enters a loop to perform more work on the array. Inside the loop, the lower and upper bounds of an array region to operate upon are removed from a shared queue, then computation on that array region is performed. The queue initially contains records representing disjoint regions along the lower and upper boundaries of the array, which do not overlap with the internal regions initially operated upon by the children. A correct execution of this program should therefore exhibit no data races.

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 shown (the internal lines only illustrate the data races and are not part of the graph). In this execution (during the first loop iteration) the "remove" operations execute concurrently, causing the right child to correctly remove the fourth record, but the left child to incorrectly remove the upper bounds from the last two records. The left child thus proceeds to operate (erroneously) on region [10,39].

In this graph, no paths connect any nodes of the left child with any nodes of the right child. Existing methods would therefore report two data races between the *work* events (shown by the dotted and dot-dashed lines), and one data race between the "remove" events (shown by the solid line). The race report between the "remove" events was a direct cause of the bug, and is not an artifact. This race involves events that either did execute concurrently or could have (which we call *feasible data races*), and also were not performed only as a result of the outcome of another data race. However, the race reports between the *work* events are artifacts. The data race shown by the dotted line is infeasible, since it involves events that could never have executed concurrently. For the accesses to [10,39] and [20,29] to have executed concurrently, the left child's "remove" operation would had to have executed before the right child's "remove" operation (with which it originally overlapped). If this had happened, the erroneous record [10,40] would not have been removed (since the two "remove" operations would not overlap), and a different array region would be accessed. Although the data race shown by the dot-dashed line is feasible (it involves events that actually did execute concurrently), it is nonetheless an artifact since the access to [10,39] was a result of the preceding "remove" executing non-atomically, leaving data in an inconsistent state.

If the array accesses had been more complex, perhaps creating other children, there may have been many nodes in the graph representing these accesses, and many data race artifacts would have been reported. Since the artifacts are not direct manifestations of program bugs but rather caused only by previous races, reporting them to the programmer can complicate debugging since they obscure the location of the bug. Artifacts 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). As this example shows, the only non-artifact data races can be located anywhere in the execution. Accurate

```
fork
  | work on region [10,19]
  | loop
  |   remove (L,U) from Queue
  |   work on region [L,U-1]
  |       of shared array
  | while QueueNotEmpty

  | work on region [20,29]
  | loop
  |   remove (L,U) from Queue
  |   work on region [L,U-1]
  |       of shared array
  | while QueueNotEmpty
join
```

Initial state of Queue:
[1,10]
[30,40]

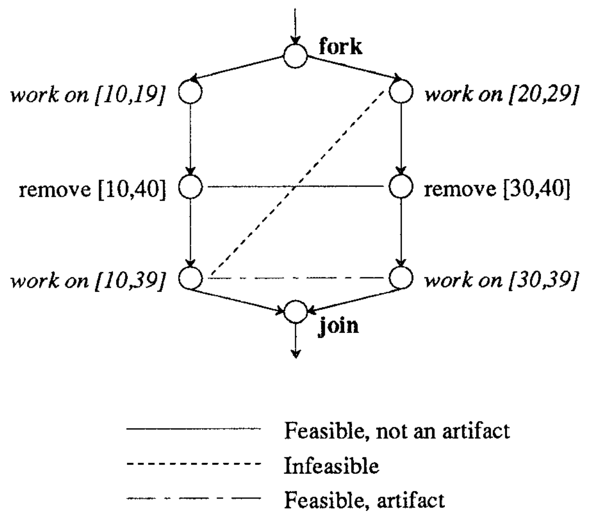


Figure 1. Example program and ordering graph (annotated with data-race report information)

race detection involves analyzing how shared data flowed through the execution and either constrained alternate orderings or caused the existence of one race to depend on the outcome of another. This paper presents results showing how such analyses can be performed without recording additional information about the execution.

3. Representing Program Executions

In this section we present a formal model as a mechanism for reasoning about shared-memory parallel program executions. The model contains 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. We use the model as a notation for describing behavior *actually* exhibited by the execution, and to represent what can be reasonably recorded about the execution. We later use the model to speculate on behavior that the execution *could have* exhibited (such as alternate event orderings) due to nondeterministic timing variations.

3.1. Program Execution Model

We provide only a brief overview of our model, which was first presented in an earlier paper[7]. Our model is based on Lamport's theory of concurrent systems[5], which provides a formalism for reasoning about concurrent systems that does not assume the existence of atomic operations. We consider executions of programs, on sequentially consistent processors, that use fork/join and counting semaphores. A program execution is modeled as a collection of *events*, E, where each event $e \in E$ represents the execution of a set of program statements and possesses two attributes, *READ* (e) and *WRITE* (e), the sets of shared variables read and written by those statements. A *synchronization event* is an instance of some synchronization operation, and 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. A *data conflict* exists between two events if one writes a shared variable that the other reads or writes. We denote the i^{th} event in process p by $e_{p,i}$, and the set of all **P** and **V** operations on semaphore S by $E_{P(S)}$ and $E_{V(S)}$. No generality is lost by modeling each event, e, as having a unique *start time* (e_s) and *finish time* (e_f).

The *temporal ordering* relation[†] among events, $\xrightarrow{T}$, describes the temporal aspects of a program execution; $a \xrightarrow{T} b$ means that a completes before b begins (in the sense that the last action of a can affect the first action of b), and $a \overset{T}{\not\longleftrightarrow} b$ means that a and b execute concurrently (i.e., neither completes before the other begins). We should emphasize that the temporal ordering relation is defined to describe the order in which events *actually* executed during a particular execution; e.g., $a \overset{T}{\not\longleftrightarrow} b$ means that a and b actually executed concurrently; it does not mean that a and b could have executed in any order.

The *shared-data dependence* relation, $\xrightarrow{D}$, indicates when one event can causally affect another, either because of a direct or transitive data dependence involving shared 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 general flow through non-shared variables 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; e.g., if a accesses a shared variable that another event, c, later accesses, and c then references a variable that b later references.

A *program execution*, P, is a triple, $\langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$. 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. The temporal ordering and shared-data dependence relations of any program execution must satisfy the following axioms.

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

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

(A3) If $a \xrightarrow{D} b$ then $b \overset{T}{\not\rightarrow} a$.

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

(A5) For all child processes, c, created by each $Fork_{p,i}$ event and terminated at event $Join_{p,i+k}$,
$Fork_{p,i} \xrightarrow{T} e_{c,j} \xrightarrow{T} Join_{p,i+k} \quad 1 \le j \le |E_c|$.

(A6) For every subset of **P** events, $P \subseteq E_{P(S)}$,
$|\{ v \mid v \in E_{V(S)} \wedge \exists p \in P \; (v \xrightarrow{T} p \;\vee\; v \overset{T}{\not\longleftrightarrow} p) \}| \ge |P|$.

The first two axioms simply state that $\xrightarrow{T}$ is consistent, axiom (A3) enforces the law of causality, axiom (A4) imposes a process structure on the set of events, and axioms (A5) and (A6) represent the semantics of **fork/join** and semaphores. Axiom (A6) states that the semaphore invariant is always maintained; namely, that the number of **V** operations that have either completed or have begun executing is always greater than or equal to the number of **P** operations that have completed (this version of axiom (A6) assumes that the initial value of each semaphore is zero).

3.2. Representing Recorded Information

So far, our model captures complete information about a program execution in the sense that $\xrightarrow{T}$ shows the relative execution order between *any* two events, and $\xrightarrow{D}$ shows the *actual* shared-data dependences. Recording such

† Superscripted arrows denote relations; $a \not\rightarrow b$ is a shorthand for $\neg(a \rightarrow b)$, and $a \not\longleftrightarrow b$ is a shorthand for $\neg(a \rightarrow b) \wedge \neg(b \rightarrow a)$.

complete information is impractical. Indeed, existing methods record only a subset of this information. We now discuss how to represent partial information in our model, by defining an *approximate program execution*, $\hat{P} = \langle E, \xrightarrow{\hat{T}}, \xrightarrow{\hat{D}} \rangle$, where $\xrightarrow{\hat{T}}$ and $\xrightarrow{\hat{D}}$ are the approximate counterparts to $\xrightarrow{T}$ and $\xrightarrow{D}$. Our intent is not to discuss details of instrumentation, but rather to represent the type of information existing methods record.

Existing methods record the temporal ordering among only some synchronization events; e.g., the order among **fork** and **join** events and their children is recorded, but the relative order of events performed by the children is not. Recording such an incomplete ordering has the advantage that the required instrumentation can be embedded into the implementation of the synchronization operations without introducing additional synchronization. A central bottleneck that could reduce the amount of parallelism achievable by the program is thus avoided, allowing the methods to scale well to large numbers of processors. We use the relation $\xrightarrow{\hat{T}}$ to represent this incomplete temporal ordering, and assume that it possesses two properties:

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

(2) If $a \not\xleftrightarrow{\hat{T}} b$ then explicit synchronization did not prevent a and b from executing concurrently.

The ordering information recorded by all existing methods has these properties. The first property states that the recorded ordering must be consistent with the actual ordering (i.e., $\xrightarrow{\hat{T}} \subseteq \xrightarrow{T}$). The second property applies when insufficient information is recorded to determine the actual execution order between a and b. In this case, if two events are not observed as being ordered, it must be because they either executed concurrently or were not prevented by explicit synchronization from doing so. As illustrated in Figure 1, however, such events may nonetheless be ordered. The goal of data race validation is to determine if the events could have indeed executed concurrently.

Although existing methods do not attempt to record shared-data dependences, they can be approximated from the *READ* and *WRITE* sets which are recorded for each computation event. The *approximate shared-data dependence* relation, $\xrightarrow{\hat{D}}$, is defined by conservatively speculating on what the actual shared-data dependences were. Consider two events, a and b, that access a common shared variable (that at least one modifies). If $a \xrightarrow{\hat{T}} b$, then there is a direct shared-data dependence from a to b. When $a \not\xleftrightarrow{\hat{T}} b$, the direction of any direct dependence cannot be determined (since the actual temporal ordering between a and b is unknown), and we make the conservative assumption that a dependence exists from a to b and from b to a. This assumption will always include the actual dependences, although it may indicate a dependence from b to a when in fact the only dependence is from a to b. As with $\xrightarrow{D}$, a transitive shared-data dependence from a to b is indicated by $\xrightarrow{\hat{D}}$ if a chain of direct dependences exists from a to b. However, as we will see, data race validation only requires $\xrightarrow{\hat{D}}$ to be determined when $a \not\xleftrightarrow{\hat{T}} b$.

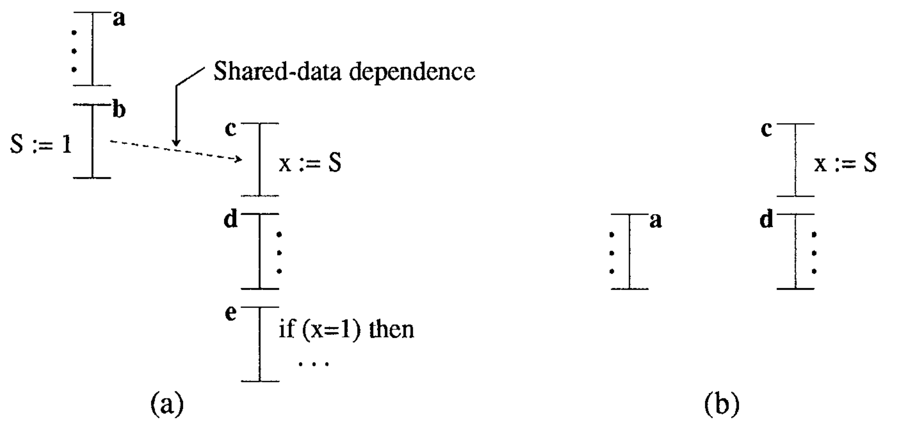
4. Characterizing Alternate Temporal Orderings

Our goal is to validate and order data races to locate those that are not artifacts. Validation requires determining which events had the potential of executing concurrently, and ordering requires determining which events might not have been performed had earlier data races not occurred. These analyses involve speculating on behaviors that the program had the *potential* of exhibiting. In this section, we speculate on alternate temporal orderings that could have potentially occurred. We extend our previous work[7] by first considering a type of shared-data dependence, called an *event-control dependence*, which describes how events affect each other in the actual program execution P. Using these dependences, the effects that alternate orderings might have had on the performed events can be determined. We then characterize *feasible program execution prefixes*, which describe executions that could have potentially occurred, and which exhibit different temporal orderings than P. In a feasible prefix, each process may perform only an initial subset of the events performed by P. The notion of a feasible prefix is central to the results in Section 5.

4.1. Event-Control Dependences

Given an actual program execution P, we consider how its events affected one another during execution to determine how a different temporal ordering might have altered the outcome of the execution. For example, if the program is nondeterministic, it may have had the potential of performing different events than P. In addition, the execution may have also had the potential of performing the same events as P but under a different ordering. We have shown[7] that any ordering that could have allowed the same shared-data dependences as those exhibited by P to occur could have also caused the same events to be performed (if interactions with the external environment are modeled as shared-data dependences). However, still other orderings may have occurred (those that would not have allowed the same shared-data dependences to occur). We observe that under such orderings, *some* of the events performed by P may still have been performed.

For example, consider a shared-data dependence in P from b to c. If the execution instead exhibits a temporal ordering in which c precedes b, this shared-data dependence can no longer occur, but part of the execution beyond b and c may still perform the same events as P. For example, assume that the dependence exists because b writes a shared variable, S, that c later reads, as shown in Figure 2(a). From axiom (A3) we know b either completes before or executes concurrently with c. Consider how the execution would differ from P if b and all subsequent events in the same process are not performed. Assuming



Dependences in (a):

$$b \xrightarrow{D} c \qquad b \xrightarrow{D} e$$
$$b \not\xrightarrow{E} c \qquad b \xrightarrow{E} e$$

Figure 2. (a) an actual program execution, and (b) a feasible program execution prefix in which *b* is excluded

that *c* could still be performed, the dependence from *b* to *c* cannot occur, and *c* may read a different value from *S*, possibly causing events performed from this point forward to differ from *P*. However, this different value of *S* may not immediately alter the events performed but only alter the values computed. In Figure 2(a), events *c* and *d* do not use *S* to determine what statements to execute or what shared locations to access, so a different value for *S* will not change these events. Different events may be performed only when *S* is finally used to determine control flow or the shared locations referenced, such as in event *e*.

To formally capture this notion of one event affecting the outcome of another, we define the *event-control dependence* relation, $\xrightarrow{E}$, on events: $a \xrightarrow{E} b$ (read as "*a* can event-control *b*") if

(1) $a \xrightarrow{D} b$ and *a* writes a shared variable whose value *b* uses (directly or through other variables) in a conditional or to determine which shared locations to access (e.g., in a shared-array subscript), or

(2) *a* is a **fork** event and *b* is the first event in a child process created at *a*, or

(3) *b* is a **join** event and *a* is the last event in a child process terminated at *b*, or

(4) *a* is a **V** event, *b* is a **P** event on the same semaphore, and *a* allowed *b* to proceed (i.e., the semaphore invariant would be violated without *a*), or

(5) *a* precedes *b* in the same process, or

(6) $a \xrightarrow{E} c \wedge c \xrightarrow{E} b$.

The $\xrightarrow{E}$ relation shows the possible effects had some event *a* (and all subsequent events in the same process) not been performed. Condition (1) includes those events receiving shared-data dependences that were used to determine control flow or the shared locations referenced. Conditions (2) through (5) include those events that would no longer be performed either because they followed *a* in the same process or because their presence depended on synchronization that followed *a*. These events are also those that are ordered after *a* by $\xrightarrow{\hat{T}}$. Condition (6) forces $\xrightarrow{E}$ to be transitive. In Figure 2(a), for example, $b \xrightarrow{D} c$, $b \xrightarrow{D} e$, and $b \xrightarrow{E} e$, but $b \not\xrightarrow{E} c$.

The above characterization of $\xrightarrow{E}$ is based on the actual shared-data dependences, $\xrightarrow{D}$. As previously discussed, recording the actual dependences is impractical, so we also define an approximate version of $\xrightarrow{E}$. These approximate event-control dependences, $\xrightarrow{\hat{E}}$, are characterized as above except that Condition (1) is based on $\xrightarrow{\hat{D}}$ instead of $\xrightarrow{D}$.

The $\xrightarrow{\hat{E}}$ relation can be conservatively computed to varying degrees of accuracy, depending on the amount of overhead that is incurred. For example, condition (1) is captured by a subset of the shared-data dependences (those involving conditionals and shared-array subscripts), and conditions (2) through (5) are captured by $\xrightarrow{\hat{T}}$; the $\xrightarrow{\hat{E}}$ relation is therefore approximated by $\xrightarrow{\hat{D}} \cup \xrightarrow{\hat{T}}$. Without examining the individual actions performed by *a* and *b*, a better approximation to $a \xrightarrow{\hat{E}} b$ can be computed by excluding from $\xrightarrow{\hat{D}}$ those direct dependences that cannot be flow dependences (i.e., a write followed by a read without an intervening write), since condition (1) requires a flow dependence. This exclusion can be done by simply examining the *READ* and *WRITE* sets; a direct dependence from *a* to *b* cannot be a flow dependence if $WRITE(a) \cap READ(b) = \varnothing$. More accurate approximations can be obtained from a static analysis of the program to further refine $\xrightarrow{\hat{D}}$. Even more accurate information can be extracted from a complete address trace, but this approach may only be practical if such a trace is already being collected for other purposes.

4.2. Feasible Program Execution Prefixes

We use the event-control dependences as the key to characterizing alternate temporal orderings that P could have exhibited. Formally, a *feasible program execution prefix*, $PP' = \langle E', \xrightarrow{T'}, \xrightarrow{D'} \rangle$, contains a subset of the events and shared-data dependences exhibited by P that we can guarantee the program could have performed. Consider each shared-data dependence, $a \xrightarrow{D} b$, exhibited by P. If both a and b are to belong to E', then we require that the dependence also be exhibited by PP' (to ensure that b can occur in PP'). If we wish to exclude a from E', then any event that is event-controlled by a must also be excluded, since it may never occur when a is not present.

Theorem 4.1

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

(P1) PP' is a valid program execution (axioms (A1)-(A6) are satisfied),

(P2) each process in PP' is a *prefix* of the corresponding process in P; i.e., for each process p, $E'_p \subseteq E_p$, and if $e_{p,i} \in E_p$ is excluded from E'_p, then so is every event $e_{p,j}$ $(j > i)$ in process p,

(P3) $\forall a,b \in E$ such that $a \xrightarrow{D} b$, either

(1) $a,b \in E'$ and $a \xrightarrow{D'} b$, or

(2) $a \notin E'$ and $\forall x \in E$ $(a \xrightarrow{E} x \Rightarrow x \notin E')$, or

(3) $b \notin E'$.

Proof Sketch. Proving this theorem requires reasoning about the individual shared-memory references made by the execution. Viewing the execution at this level (where each event represents at most one shared-memory access), it is easily argued that the theorem holds. The crux of the proof involves showing that the result extends to a higher-level view in which events represent arbitrarily many accesses. We have proven that any other program execution, obeying axioms (A1)-(A6), and possessing the same $\xrightarrow{D}$ relation as P, describes an execution performing exactly the same events as P, and is thus feasible[7]. Showing that Theorem 4.1 holds at a higher-level is analogous to this proof, and we omit the details here. ■

Figure 2(b) shows an example feasible prefix. Since b is excluded from the prefix, e must also be excluded because $b \xrightarrow{E} e$, but c can remain (even though $b \xrightarrow{D} c$) because $b \not\xrightarrow{E} c$. This example shows that a and d could have executed concurrently even though the shared-data dependence from a to b would then no longer occur.

We finally mention that our event-control dependences are similar to the *hides* relation used by Allen and Padua[1], and the *semantic dependences* defined by Podgurski and Clarke[9]. The hides relation is defined to show when the data computed by an event in one data race may have been used by an event in another race to determine either control flow or the shared locations accessed. Allen and Padua propose computing the hides relation by a static analysis of the program, and use it primarily to locate data races that might have been prevented from occurring because of a previous race. In contrast, we use the event-control dependences to locate data races that were caused by a previous race. Podgurski and Clarke statically define a semantic dependence to exist from one statement in a sequential program to another if the function computed by the first statement can affect the execution behavior of the second in any way. Our event-control dependence can be viewed as a type of dynamic semantic dependence but generalized to parallel programs (where dependences involving synchronization as well as data must be considered).

5. Validating and Ordering Data Races

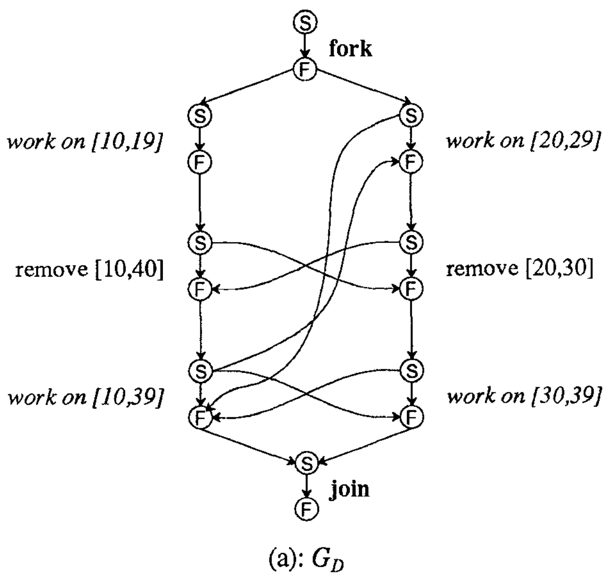
Existing data race detection methods can report data races that are artifacts of other races, either because the reported data races involve events that could never have executed concurrently, or because they were performed only as a result of other data races. To determine which are not artifacts, we now show how the shared-data and event-control dependences can be used to perform analyses that validate and order these races.

5.1. Data Race Validation

As discussed in Section 2, existing methods construct an ordering graph and report a data race between any two data-conflicting events, a and b, whose nodes have no connecting path. We call such a race an *apparent data race*, denoted $\langle a,b \rangle$. As shown in Figure 1, not all apparent data races involve events that could ever have executed concurrently. We call a data race between events that either did or could have executed concurrently a *feasible data race*. The goal of validation is to conservatively determine which apparent data races are feasible. Below we re-state our previous results in terms of feasible prefixes, and then sharpen them by employing the event-control dependences.

To validate the apparent data races, we use a variation of the ordering graph we call the *temporal ordering graph*, G. Unlike existing methods, which use only one node per event, G contains two nodes, $\mathbf{e_s}$ and $\mathbf{e_f}$, for every event e (corresponding to the start and finish of e)[†]. This graph defines the approximate temporal ordering, $\xrightarrow{\hat{T}}$, 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 $a \xleftrightarrow{\hat{T}} b$ otherwise. Given a temporal ordering graph, we say that a is a *predecessor* of b in the graph if a path exists from $\mathbf{a_f}$ to $\mathbf{b_s}$, a is a *successor* of b if a path exists from $\mathbf{b_f}$ to $\mathbf{a_s}$, and a and b are *unordered* by the graph if no such paths exist. Our validation results are based on augmenting G with edges representing shared-data and event-control dependences

† In practice, constructing two nodes per event is unnecessary, but we use such a representation here as it conceptually follows our model.



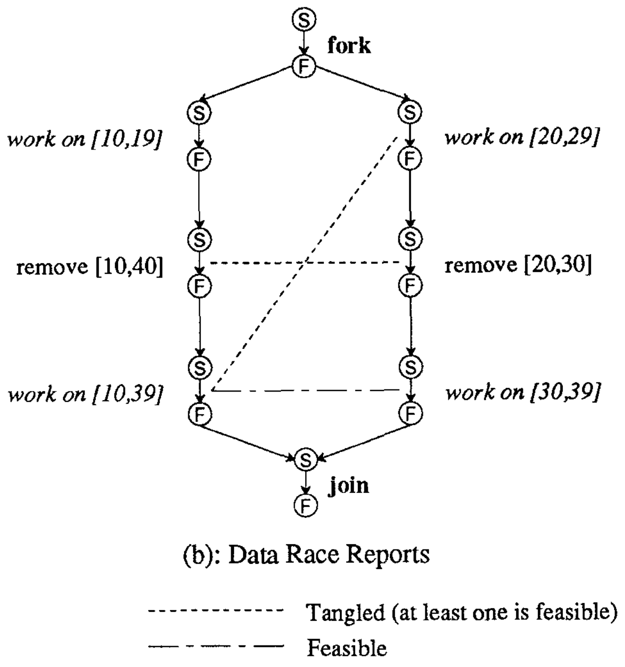


Figure 3. (a) G_D for example in Figure 1, and (b) validation results from analyzing G_D

and analyzing the orderings given by the resulting graph.

To show an apparent data race $\langle a,b \rangle$ is feasible, it suffices to guarantee that a feasible program execution prefix, $PP' = \langle E', \xrightarrow{T'}, \xrightarrow{D'} \rangle$, exists such that $a \overset{T'}{\not\leftrightarrow} b$. The existence of such a prefix means that the program *could have* executed in such a way that a and b executed concurrently. To determine if such a prefix exists requires knowledge of the shared-data and event-control dependences exhibited by the observed execution. Without exhaustive execution tracing, however, the exact dependences are unknown. As previously discussed, $\xrightarrow{\hat{D}}$ reflects a conservative estimate of $\xrightarrow{D}$, and $\xrightarrow{\hat{E}}$ can be approximated from $\xrightarrow{\hat{T}}$ and (perhaps refinements of) $\xrightarrow{\hat{D}}$.

We augment the temporal ordering graph G with edges reflecting these approximations. We first construct the graph G_D by augmenting G with edges that ensure there is a path from $\mathbf{a_s}$ to $\mathbf{b_f}$ whenever $a \xrightarrow{\hat{D}} b$. Such a path already exists when $a \xrightarrow{\hat{T}} b$, so edges are only added between events unordered by G. Figure 3(a) shows an example G_D (''S'' and ''F'' label the start and finish nodes).

In a previous paper[7], we showed how some apparent data races can be validated by analyzing G_D. Intuitively, the edges added to construct G_D reflect the possible orderings caused by shared-data dependences, similar to the way in which the edges in G reflect orderings caused by the execution's explicit synchronization. For example, if an apparent data race $\langle a,b \rangle$ exists, then a and b are unordered by G because the program's explicit synchronization did not prevent them from executing concurrently. Similarly, if a and b are also unordered by G_D, then no shared-data dependence could have prevented them from executing concurrently either, so the data race must be feasible.

Determining if a and b are unordered by G_D is complicated when the approximation of $\xrightarrow{D}$ is so conservative that G_D contains cycles. We therefore classified the apparent data races into those that participate in cycles and those that do not. An apparent data race $\langle a,b \rangle$ is *tangled* if $\mathbf{a_f}$ and $\mathbf{b_s}$ (or $\mathbf{b_f}$ and $\mathbf{a_s}$) belong to the same strongly connected component of G_D. Each strongly connected component defines a set of tangled data races, called a *tangle*. The following theorems summarize our previous results.

Theorem 5.1.

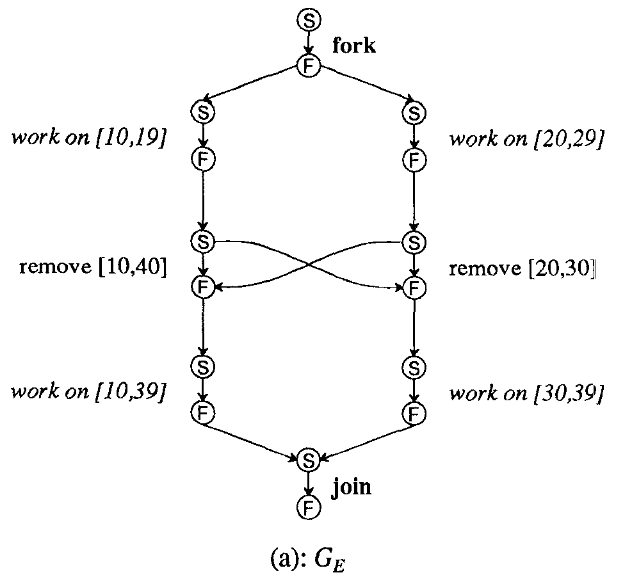
An apparent data race $\langle a,b \rangle$ is feasible if a and b are unordered by G_D (i.e., if no path from $\mathbf{a_f}$ to $\mathbf{b_s}$, or from $\mathbf{b_f}$ to $\mathbf{a_s}$, exists in G_D).

Theorem 5.2.

In each tangle defined by G_D, at least one of the tangled data races is feasible.

For example, in Figure 3(b), the apparent data race between the last two *work* events is not tangled, and is therefore feasible; the other two data races form a tangle, at least of which must be feasible.

We extend these results here by observing that not all shared-data dependences necessarily cause events to actually be ordered. We construct the graph G_E by removing those edges from G_D representing shared-data dependences that are not also event-control dependences. Figure 4(a)



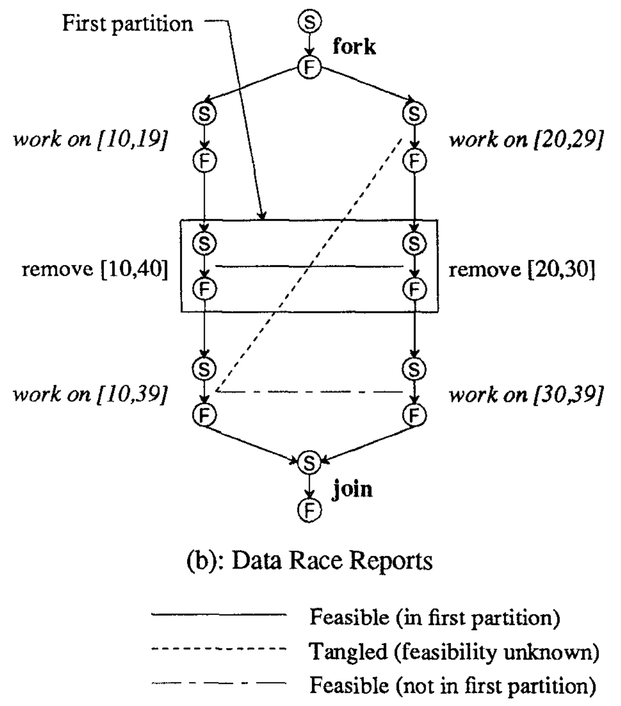


Figure 4. (a) G_E for example in Figure 1, and (b) final validation and ordering results

shows G_E for the example in Figure 1[†].

By analyzing G_E, a tangled data race $\langle a,b \rangle$ (which cannot be validated by analyzing G_D) can sometimes be validated. In this case, even though shared-data dependences caused a and b to be ordered by G_D, these dependences may not have affected the outcome of a and b. The tangled data race is feasible if we can guarantee that both a and b would have remain unchanged had these dependences not occurred. If a and b are unordered by G_E, we are guaranteed that all *predecessors* of a or b would have remain unchanged, because in this case neither a nor b (nor any of their successors) event-control any of these predecessors. In addition, if none of the successors of a or b can event-control a or b, then we are also guaranteed that a and b themselves would have remained unchanged. The following theorem states this result (proofs of theorems appear in the appendix).

Theorem 5.3.

An apparent data race $\langle a,b \rangle$ is feasible if a and b are unordered by G_E, and no successor of a or b in G_E can event-control a or b (i.e., if no path from $\mathbf{a_f}$ to $\mathbf{b_f}$, or from $\mathbf{b_f}$ to $\mathbf{a_f}$, exists in G_E).

For example, Figure 4(b) shows we can determine that the apparent race between the "remove" events is feasible.

† In this example, we assume that the "remove" events can event-control each other, but the *work* events cannot (e.g., because the computed array values are not used to determine control flow).

The above results show how the apparent data races can be validated. Validation allows the programmer to be directed to those data races that are feasible. When insufficient information exists to determine the feasibility of a data race, tangles can nonetheless be identified, localizing a portion of the execution within which at least one feasible data race is guaranteed to exist.

5.2. Data Race Ordering

A data race between two events may result in either of the events executing non-atomically. In programs expected to be data race free, this non-atomicity may leave the program's data in an inconsistent state unanticipated by the programmer, possibly causing subsequent unexpected program behavior. Subsequent data races may result only because of this unexpected behavior. Reporting these data races can complicate debugging, since they are artifacts of the previous races and not directly caused by program bugs. We now present new results showing how to use the event-control dependences to first order the data races and then identify groups of races that could not have been artifacts of others. These groups of "first" data races should be reported to the programmer.

To determine which data races may have been artifacts of others, we define an ordering, $\xrightarrow{R}$, on the apparent data races. The purpose of this ordering is to show when events in a data race may have been affected by the possible atomicity failures of events in other races.

$<a,b> \xrightarrow{R} <c,d> \Leftrightarrow$
$(a \xrightarrow{\hat{E}} c \;\wedge\; b \xrightarrow{\hat{E}} c) \;\vee\; (a \xrightarrow{\hat{E}} d \;\wedge\; b \xrightarrow{\hat{E}} d)$.

If $<a,b> \xrightarrow{R} <c,d>$, then $<c,d>$ could possibly have been an artifact of $<a,b>$ if either a or b executed non-atomically. Note that $\langle c,d \rangle$ cannot have been an artifact unless both a and b event-control c (or d).

Theorem 5.4.

If $\langle a,b \rangle \overset{R}{\not\rightarrow} \langle c,d \rangle$ then $\langle c,d \rangle$ could not have been an artifact of $\langle a,b \rangle$.

Because $\xrightarrow{\hat{E}}$ is based on approximate information, $\xrightarrow{R}$ can be symmetric, causing both $<a,b> \xrightarrow{R} <c,d>$ and $<c,d> \xrightarrow{R} <a,b>$. In this case, insufficient information is available to determine whether $<c,d>$ may have been an artifact of $<a,b>$ or *vice-versa*. We therefore partition the apparent data races into groups such that two data races belong to the same partition if it is unknown which may have been an artifact of the other. More specifically, $<a,b>$ and $<c,d>$ belong to the same partition iff $<a,b> \xrightarrow{R} <c,d>$ and $<c,d> \xrightarrow{R} <a,b>$. Sufficient information is available to order by $\xrightarrow{R}$ data races belonging to different partitions. We can thus identify the *first* partitions, which are those containing no data races that may have been artifacts of races in other partitions. These first partitions should be reported to the programmer. Figure 4(b) shows the first partition (containing only one race) for the example. Since partitions in general can contain both tangled and non-tangled data races, the programmer should also be provided with a report of which races in each partition are tangled and which are feasible. The following theorem shows that each first partition contains at least one data race that was not an artifact of any kind.

Theorem 5.5.

Each first partition contains at least one feasible data race that was not an artifact of any other data race.

Partitions are similar to tangles in the sense that both contain races for which insufficient information is known to determine whether they are feasible (in the case of tangles) or may have been artifacts of other races (in the case of partitions). However, it appears that partitions cannot be identified by simply augmenting the graph, as was done to identify tangles. Nonetheless, simple algorithms exist to identify and order the partitions; they are the topic of a future paper.

5.3. Related Work

One proposed method for detecting data races does address the need for locating first races. Choi and Min[2] present an on-the-fly approach for executions of programs intended to be deterministic. Their goal is to locate a set of data races, called the *Race Frontier* (containing at most one race in each process), up to which re-execution of the program is guaranteed to be deterministic. The Race Frontier consists of the first event (in each process) that participates in a data race, unless such a race is tangled, in which case the Frontier is chosen so it contains both events of the tangled race that occurred before any other race in the tangle. To determine which tangled race occurred first, they rely upon additional ordering information which is by-product of the on-the-fly approach; a race check is performed at each shared-memory access, allowing the order of all accesses to the same location to be determined.

Their work has similarities to ours, since they locate first races. However, they assume that an event is potentially affected by any predecessor in the ordering graph. We use a more refined notion of how events affect each other (the event-control dependences), allowing us to determine that some data races are not artifacts even if they did not temporally occur first (and even if only approximate information is available). Our results also apply to nondeterministic programs. Finally, we use only the ordering information captured by the ordering graph, avoiding the central bottleneck caused by the on-the-fly approach of serializing all accesses to the same shared-memory location.

6. Conclusion

Existing methods for data race detection can often report data races that are not directly caused by program bugs, but are artifacts of other data races. The purpose of this paper was to present results showing how to *validate* and *order* these data races to conservatively locate those that are not artifacts. Validation locates *feasible* data races, which had the potential of actually occurring, and ordering locates races that were not caused only as a result other races. Locating non-artifact data races aids debugging by directing the programmer to the direct cause of the races.

Our results are proven within a model for reasoning about data races in which the execution's actual and potential behavior is characterized. We showed how to validate data races by augmenting the ordering graph with additional edges representing the shared-data and event-control dependences. By analyzing the augmented graph, each race can either be guaranteed feasible, or when insufficient information is known to make this determination, tangles can be identified within which at least one feasible race is guaranteed to exist. The more precisely that the event-control dependences can be computed, the more data races can be validated. To order the data races, we showed how to group them into partitions and locate the *first* partitions, each of which is guaranteed to contain at least one data race that is both feasible and did not occur only as a result of another race. Future work includes developing efficient algorithms for actually performing validation and ordering.

Appendix. Proofs of Theorems

Theorem 5.3.

An apparent data race $\langle a,b \rangle$ is feasible if a and b are unordered by G_E, and a (or any successor of a in G_E) cannot event-control b or *vice-versa* (i.e., if no path from $\mathbf{a_f}$ to $\mathbf{b_f}$, or from $\mathbf{b_f}$ to $\mathbf{a_f}$, exists in G_E).

Proof. Let $P = \langle E, \xrightarrow{T}, \xrightarrow{D}\rangle$ and $\hat{P} = \langle E, \xrightarrow{\hat{T}}, \xrightarrow{\hat{D}}\rangle$ be the actual and approximate program executions, let *Pred* be the set that includes a, b, and all their predecessors in G_E, and let *Succ* be the set of all successors of a and b. We first show that no event in *Succ* can event-control any event in *Pred*, and then show that the data race $\langle a,b\rangle$ is feasible by constructing a feasible program execution prefix, $PP' = \langle E', \xrightarrow{T'}, \xrightarrow{D'}\rangle$, containing a and b in which $a \overset{T'}{\nleftrightarrow} b$.

First, the absence of a path from $\mathbf{a_f}$ to $\mathbf{b_f}$ means that no successor of a can event-control b or any predecessor of b. Since no path exists from $\mathbf{b_f}$ to $\mathbf{a_f}$ either, no event in *Succ* can event-control any event in *Pred*.

Next, we show that the apparent data race $\langle a,b\rangle$ is feasible. We first introduce $G_{D\text{-}ACTUAL}$, the temporal ordering graph augmented with edges representing the actual shared-data dependences, $\xrightarrow{D}$. $G_{D\text{-}ACTUAL}$ is identical to G_D, except shared-data dependence edges representing only the actual dependences appear (extraneous edges appear in G_D since it reflects $\xrightarrow{\hat{D}}$, the conservative estimate). Even though we do not have enough information to construct $G_{D\text{-}ACTUAL}$, it nonetheless exists, and we use it to define a feasible prefix, $PP' = \langle E', \xrightarrow{T'}, \xrightarrow{D'}\rangle$, as follows.

(1) E' contains all the events in E, except for events either in *Succ* or event-controlled by events in *Succ*.

(2) $\xrightarrow{T'}$ is defined by a linear ordering of the nodes of the graph $G'_{D\text{-}ACTUAL}$, constructed by removing all nodes (and any incident edges) from $G_{D\text{-}ACTUAL}$ representing events not in E'.

(3) $\forall x,y \in E',\ x \xrightarrow{D'} y \;\Leftrightarrow\; x \xrightarrow{D} y$.

We claim that PP' is a feasible program execution prefix. PP' obeys axioms (A1)-(A6) and therefore satisfies condition (P1) (see Section 4). Axioms (A1) and (A2) are satisfied since $\xrightarrow{T'}$ is defined by a linear ordering of an acyclic graph. Axiom (A3) is satisfied because if $x \xrightarrow{D'} y$ then $x \xrightarrow{D} y$, implying that a path from $\mathbf{x_s}$ to $\mathbf{y_f}$ exists in $G_{D\text{-}ACTUAL}$. Since $\xrightarrow{T'}$ is constructed from $G_{D\text{-}ACTUAL}$, such a path also implies that $y \overset{T'}{\nrightarrow} x$. Axioms (A4)-(A6) are satisfied since some linear ordering of G obeys these axioms (see properties (1) and (2) in Section 3.2), and $\xrightarrow{\hat{T}} \Rightarrow \xrightarrow{\hat{E}}$ (so the successors in such a linear ordering of any event excluded from G are also excluded). Condition (P2) is clearly satisfied by the definition of E'. Finally, (P3) is satisfied since all events that can be event-controlled by an event excluded from E' are also excluded.

To show that the apparent data race $\langle a,b\rangle$ is feasible, we must show that E' contains a and b and that $a \overset{T'}{\nleftrightarrow} b$. Because only the events event-controlled by those in *Succ* are excluded from E' (which, as already shown, includes no event in *Pred*), a and b remain in E'. Since a and b are the last events in their processes appearing in E', there is a linear ordering of the nodes of $G'_{D\text{-}ACTUAL}$ in which $\mathbf{a_s}$ appears before $\mathbf{b_f}$ and $\mathbf{b_s}$ appears before $\mathbf{a_f}$ (since a and b are not synchronization events, no synchronization prevents this ordering). Such a linear ordering defines a $\xrightarrow{T'}$ relation such that $a \overset{T'}{\nleftrightarrow} b$. Therefore, since a feasible program execution prefix, $PP' = \langle E', \xrightarrow{T'}, \xrightarrow{D'}\rangle$, exists in which $a \overset{T'}{\nleftrightarrow} b$, the data race between a and b is feasible. ∎

Theorem 5.4.

If $\langle a,b\rangle \overset{R}{\nrightarrow} \langle c,d\rangle$ then $\langle c,d\rangle$ could not have been an artifact of $\langle a,b\rangle$.

Proof. The apparent data race $\langle c,d\rangle$ is an artifact only if a or b executed non-atomically and this non-atomicity affected c or d. To prove this theorem, we must reason about the portions of a and b that executed non-atomically. Since in our model computation events can be defined to comprise any amount of computation performed in between synchronization operations, we can view a and b as comprising lower-level events[7]. Let a_{atom} and b_{atom} be the initial portions of a and b that executed atomically, and let a_{natom} and b_{natom} be the remainder. Executing atomically means that each variable read in a_{atom} returned the value of the last write in a_{atom} to the variable (or the initial value at the start of a_{atom} if no such write occurred). The remainder of a exhibited non-atomicity because b wrote a shared variable read by a_{natom}. We must show that a feasible program execution prefix exists in which a_{natom} and b_{natom} are excluded but the apparent data race $\langle c,d\rangle$ remains. The existence of such a feasible prefix shows that, no matter how a_{natom} or b_{natom} might have changed had a or b executed atomically, the race $\langle c,d\rangle$ would have remained unaffected. To prove the prefix exists, we show that neither a_{natom} nor b_{natom} can event-control either c or d. Since $\xrightarrow{\hat{E}}$ is transitive, no successor in G_E of a or b can event-control c or d either, which suffices to show that the prefix exists (see the construction of PP' in the proof of Theorem 5.3).

By definition, $\langle a,b\rangle \overset{R}{\nrightarrow} \langle c,d\rangle$ implies that $(a \overset{\hat{E}}{\nrightarrow} c \wedge a \overset{\hat{E}}{\nrightarrow} d) \vee (a \overset{\hat{E}}{\nrightarrow} c \wedge b \overset{\hat{E}}{\nrightarrow} d) \vee (b \overset{\hat{E}}{\nrightarrow} c \wedge a \overset{\hat{E}}{\nrightarrow} d) \vee (b \overset{\hat{E}}{\nrightarrow} c \wedge b \overset{\hat{E}}{\nrightarrow} d)$. We show that $a \overset{\hat{E}}{\nrightarrow} c \Rightarrow b_{natom} \overset{\hat{E}}{\nrightarrow} c$; analogous arguments also apply to d and a_{natom}. Assume that c was affected by the non-atomicity of b; i.e., $b_{natom} \xrightarrow{\hat{E}} c$. This non-atomicity was caused by a writing a shared variable that was read by b_{natom}. By condition (1) of the definition of event-control dependence, $a \xrightarrow{\hat{E}} c$, which is a contradiction. Therefore, neither a_{natom} nor b_{natom} can event-control c or d. ∎

Lemma 5.5.

If $<a,b>$ is an infeasible apparent data race, then there exists another apparent data race, $<c,d>$, such that $<c,d> \xrightarrow{R} <a,b>$.

Proof. First, let $G_{E\text{-}ACTUAL}$ be the temporal ordering graph augmented with event-control dependence edges representing actual event-control dependences; $G_{E\text{-}ACTUAL}$ is similar to $G_{D\text{-}ACTUAL}$ (see the proof of Theorem 5.3) in that it only contains edges representing the actual dependences. If $<a,b>$ is infeasible, then no feasible program execution prefix exists in which a and b can execute concurrently. The non-existence of such a prefix can occur only if successors of a in $G_{E\text{-}ACTUAL}$ event-control b or predecessors of b, or *vice-versa*. Assume the former. $G_{E\text{-}ACTUAL}$ must therefore contain a path from $\mathbf{a_f}$ to $\mathbf{b_f}$. We show that this path implies another apparent data race exists.

The path from $\mathbf{a_f}$ to $\mathbf{b_f}$ implies that $a \xrightarrow{\hat{T}} x_0 \xrightarrow{E} b$ for some event x_0. Since $x_0 \xrightarrow{E} b$, there exists a chain of events, $x_0, x_1, \cdots, x_n$ $(x_n = b)$, such that $x_0 \; (\xrightarrow{DD} \cup \xrightarrow{\hat{T}}) \; x_1 \; (\xrightarrow{DD} \cup \xrightarrow{\hat{T}}) \cdots (\xrightarrow{DD} \cup \xrightarrow{\hat{T}}) \; x_n$. Since $a \not\xleftrightarrow{T'} b$, not all events in this chain can be ordered by $\xrightarrow{\hat{T}}$, so $x_i \xrightarrow{DD} x_{i+1}$ for some i. An apparent data race therefore exists between x_i and x_{i+1}. By condition (1) of the definition of $\xrightarrow{E}$, $x_i \xrightarrow{E} b$ for $1 \le i < n$. Since $x_i \xrightarrow{E} b$ and $x_{i+1} \xrightarrow{E} b$ (and since $\xrightarrow{E} \Rightarrow \xrightarrow{\hat{E}}$), we have $<x_i, x_{i+1}> \xrightarrow{R} <a,b>$. ■

Theorem 5.5.

Each first partition contains at least one feasible data race that was not an artifact of any other data race.

Proof. To prove this theorem, we must reason about the individual shared-memory accesses performed by the program execution $P = \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$. Since in our model computation events can be defined to comprise any amount of computation performed in between synchronization operations, we can view $P = \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$ as a *higher-level view* of a program execution, $P_S = \langle E_S, \xrightarrow{T_S}, \xrightarrow{D_S} \rangle$, in which each computation event is defined to comprise at most one shared-memory access, in the sense that each event $e \in E$ can be thought of as containing a set of lower-level events from E_S [7]. The *single-access view* P_S contains information regarding the relative order in which individual shared-memory accesses were performed. Even though P_S is unknown (and cannot be uniquely determined from P), it is nonetheless a valid program execution (just describing more detail than P), and the various relations and theorems defined on the events in E are also defined on the events in E_S. Since each computation event in P_S represents at most one shared-memory access, the $\xrightarrow{R}$ relation among the (single-access) apparent data races in E_S is a partial order. Consider any first partition of the apparent data races in P, and the corresponding single-access races in P_S. Since the single-access races are partially ordered by $\xrightarrow{R}$, at least one of them, $<a_s, b_s>$, appears first in the ordering. By Theorem 5.4 and Lemma 5.5, these first races are both feasible and not artifacts of any other race. The events a_s and b_s are the first events contained in a and b that participate in a data race (or else they would not appear first in the $\xrightarrow{R}$ ordering). The higher-level events, a and b, thus contain a feasible data race that is not an artifact of any other race. ■

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] Choi, J.-D. and S. L. Min, "Race Frontier: Reproducing Data Races in Parallel Program Debugging," *Proc. of Symp. on Principles and Practice of Parallel Prog.*, Williamsburg, VA, (April 1991).

[3] Dinning, A. and E. Schonberg, "An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection," *Proc. of Symp. on Principles and Practice of Parallel Prog.*, pp. 1-10 Seattle, WA, (Mar. 1990).

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

[5] Lamport, L., "The Mutual Exclusion Problem: Part I — A Theory of Interprocess Communication," *JACM* 33(2)(Apr. 1986).

[6] Miller, B. P. and J.-D. Choi, "A Mechanism for Efficient Debugging of Parallel Programs," *Proc. of Conf. on Prog. Lang. Design and Impl.*, pp. 135-144 Atlanta, GA, (June 1988). Also *SIGPLAN Notices* 23(7) (July 1988).

[7] Netzer, R. H. B. and B. P. Miller, "Detecting Data Races in Parallel Program Executions," in *Languages and Compilers for Parallel Computing*, ed. D. Gelernter, T. Gross, A. Nicolau, and D. Padua, MIT Press (1991). Also *Proc. of the 3rd Workshop on Prog. Langs. and Compilers for Parallel Computing*, Irvine, CA, (Aug. 1990).

[8] Nudler, I. and L. Rudolph, "Tools for the Efficient Development of Efficient Parallel Programs," *Proc. of 1st Israeli Conf. on Computer System Eng.*, (1988).

[9] Podgurski, A. and L. A. Clarke, "A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance," *IEEE Trans. on Software Engineering* **16**(9) pp. 965-979 (Sep. 1990).