RACE CONDITION DETECTION FOR DEBUGGING SHARED-MEMORY PARALLEL PROGRAMS

by

ROBERT HARRY BENSON NETZER

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

1991

ACKNOWLEDGEMENTS

Bart Miller is an adviser in the truest sense of the word. Over the past four years, the sage advice which he has been more than able (and all too willing) to offer has had an indelible effect on my perception of computer science and on my research skills in general. This thesis would not exist in its present form without his insight.

Serving on a thesis committee is often a thankless task. To assuage this, I wish to thank Profs. Jim Larus, Marvin Solomon, Eric Bach, and Peter Monkmeyer for serving on my committee. Special thanks go to Jim and Marv for actually reading this thesis (and to Marv for the simple proof of Theorem 6.1), and to Peter Monkmeyer for staying awake during my thesis defense.

Finally, Eric Bach, Phil Pfeiffer and Jon Sorenson deserve recognition for always having time to discuss problems related to this work.

ABSTRACT

This thesis addresses theoretical and practical aspects of the dynamic detecting and debugging of race conditions in shared-memory parallel programs. To reason about race conditions, we present a formal model that characterizes actual, observed, and potential behaviors of the program. The actual behavior precisely represents the program execution, the observed behavior represents partial information that can be reasonably recorded, and the potential behavior represents alternate executions possibly allowed by nondeterministic timing variations. These behaviors are used to characterize different types of race conditions, *general races* and *data races*, which pertain to different classes of parallel programs and require different detection techniques. General races apply to programs intended to be deterministic; data races apply to nondeterministic programs containing critical sections.

We prove that, for executions using synchronization powerful enough to implement two-process mutual exclusion, locating every general race or data race is an NP-hard problem. However, for data races, we show that detecting only a subset of all races is sufficient for debugging. We also prove that, for weaker types of synchronization, races can be efficiently located.

We present post-mortem algorithms for detecting race conditions as accurately as possible, given the constraint of limited run-time information. We characterize those races that are direct manifestations of bugs and not artifacts caused by other races, imprecise run-time traces (causing false races to appear real), or unintentional synchronization (caused by shared-memory references). Our techniques analyze the observed behavior to conservatively locate races that either did occur or had the potential of occurring, and that were unaffected by any other race in the execution.

Finally, we describe a prototype data race detector that we used to analyze a sample collection of parallel programs. Experiments indicate that our techniques effectively pinpoint non-artifact races, directing the programmer to parts of the execution containing direct manifestations of bugs. In all programs analyzed, our techniques reduced hundreds to thousands of races down to four or fewer that required investigation.

Chapter 1

INTRODUCTION

1.1. Motivation

Parallel programs can be nondeterministic. The behavior of these programs often depends on the interactions between processes, which can depend on the relative speed of process execution. These interactions become more complex when programs communicate through a shared memory. If updates to common shared-memory locations are not carefully synchronized, time-dependent bugs called *race conditions* can result. Race conditions occur when shared data is accessed by different processes without explicit synchronization. Programs containing races can behave in ways unexpected by their programmers. Exploiting the full power of shared-memory parallel processors therefore requires mechanisms for determining when a program execution is free from race conditions and for assisting a programmer in locating race conditions when they occur. This thesis addresses both theoretical and practical aspects of race condition detection.

Program executions exhibiting race conditions present problems for both programmers and debuggers. First, a programmer may expect their program to behave deterministically, producing the expected result each time the program is run using the same input. A nondeterministic program can exhibit different behavior from run to run, even though the same input is used. Since nondeterministic behavior depends on the relative timing between processes, erroneous behavior may be manifested only on rare occasions. It may not be clear from a single execution of the program that nondeterministic behavior is possible. The program might appear deterministic, producing the expected results for many runs, but may actually be nondeterministic, having the potential of producing the wrong results. A mechanism is therefore desirable for automatically determining whether an execution is potentially nondeterministic. Nondeterminism is introduced when the order in which any two accesses to a shared variable (where at least one access is a modification), or the order in which any two synchronization operations are issued, is not guaranteed by the program's synchronization. We call such a possibility a *general race* (previous work uses the term *race condition*).

Second, even if the programmer expects the program to be nondeterministic, the program may lack the proper synchronization required for it to behave as expected. In such programs, explicit synchronization is often added to implement critical sections. Critical sections are portions of code intended to execute atomically[18]. Atomic execution is guaranteed if the shared variables read or modified by the critical section are not modified by any other concurrently executing section of code. Without proper synchronization, these shared variables may be modified by other processes as the code executes, violating the expected atomicity. When this type of interference occurs, a *data race* is said to exist. As with general races, a mechanism is also desirable for automatically detecting data races.

Finally, debugging program executions containing race conditions requires not only a tool for determining whether an execution exhibited races, but also for locating their causes. The reported race conditions must be related back to the program bugs that caused their existence. Making this connection requires precisely defining race conditions to clearly understand the semantics of the reported races. Moreover, the race detector should locate these races accurately. Accurate location requires that only those races that are direct manifestations of program bugs be reported, and not those that may be artifacts of other races or imprecise run-time traces. We therefore believe that debugging race conditions requires both a precise model of the semantics of race conditions and tools for accurate race detection. This thesis addresses both of these concerns.

1.2. Summary of Results

The main contributions of this work are (1) a theory in which to reason about race conditions and prove properties about race condition detection, (2) formal characterizations of different types of race conditions and results regarding the complexity and accuracy of dynamically detecting them, and (3) practical techniques that aid the programmer by locating those race conditions that are of interest for program debugging.

First, the foundation of this work is a formal model for reasoning about the properties of race conditions. This model provides a sound basis on which to formulate the race-detection problem and to investigate properties of race detection techniques. Our model is novel in that the actual, observed, and potential behaviors of the program are characterized. The actual behavior contains complete information about the execution, the observed behavior contains only information that can be reasonably recorded, and the potential behavior contains information about alternate orderings in which the execution's events could have been performed. Characterizing these behaviors is necessary for formal reasoning about race conditions. Using this approach, race conditions are precisely defined, and the reported races are related back to the behavior of the program. Moreover, this approach has the advantage that the complexity and accuracy of race detection is made explicit. Since previous work has only been founded on intuition, there has been little agreement as to precisely what constitutes a race condition, and it is unclear how the races detected by previously proposed methods relate to program bugs. This thesis presents new results regarding the complexity and accuracy of race detection, and shows how to accurately locate race conditions.

Second, we characterize different types of race conditions in terms of our model and explore their properties. We characterize both data races and general races, and argue that each pertains to a different class of parallel programs. Data races pertain to nondeterministic programs that have critical sections (and therefore use synchronization powerful enough to implement mutual exclusion). General races pertain to programs intended to be deterministic (which may use any type of synchronization). In the past, the distinction between data races and general races has not been examined. We also prove bounds on the complexity of locating both types of races. For programs using synchronization powerful enough to implement two-process mutual exclusion, exhaustively locating all data races, or locating any general races, is an NP-hard problem. For programs using weaker synchronization, we present the first efficient algorithm for computing the event orderings required for race detection. Moreover, we argue that general race detection (which always requires computing all potential behaviors) is inherently more difficult than data race detection (which can be performed given only the actual behavior). These results establish complexity bounds for race detection, showing for which types of synchronization the debugging problem is efficient or intractable.

We also uncover previously hidden issues regarding the accuracy of race detection. We use our model to analyze previously proposed methods and show that two variations of each type of race exist. One variation captures an intuitive notion (which we call a *feasible race*), and the other captures a less accurate notion (called an *apparent race*). Previous race detection methods locate the apparent races. The apparent races are less accurate in the sense that they can be artifacts of other races and not direct manifestations of program bugs. Reporting race artifacts complicates the debugging process by overwhelming the programmer with large amounts of irrelevant information. A race can be an artifact because of several factors. First, the outcome of another race may affect its presence so that the race would not have occurred had the other race not been present. Second, the need to keep run-time tracing overhead low results in limited information about the execution, causing more of the artifacts to be detected as apparent races. Finally, when the values of shared variables are used in conditional expressions or in shared-array subscripts, the data dependences (among shared variables) between some apparent races can prevent others from ever actually occurring. This latter factor is reminiscent of using shared memory to perform synchronization, instead of using explicit synchronization primitives. However, we show that such situations can occur even when the programmer does not intentionally use shared variables in this way.

Finally, identifying the distinction between feasible and apparent races motivates the practical techniques presented in this thesis. Given our foundations and results, we focus on data races and develop techniques for accurate data race detection. We present two techniques, called *validation* and *ordering*, for determining which apparent data races are direct manifestations of program bugs and not artifacts of other races. The first technique, data race validation, attempts to determine which data races are feasible. Feasible races are those that either occurred during the observed execution or had the potential of occurring. Each data race can either be guaranteed feasible, or sets of data races can be identified within which at least one is guaranteed feasible. The second technique, data race ordering, attempts to locate those data races that are not artifacts. Non-artifact races are those that were directly caused by a program bug and not only a result of a previous race. Partitions of data races can be identified, each of which is guaranteed to contain at least one feasible, non-artifact race. Validation is intended to refine the results of ordering by providing information regarding the feasibility of races in these partitions. By applying a combination of validation and ordering, the programmer can be directed to those races that are manifestations of bugs. Examining these races first simplifies the debugging task by localizing the portion of the execution that must be considered.

We present simple algorithms for performing post-mortem validation and ordering. These algorithms analyze the trace data by speculating on how events affected one another during execution. The discrimination power depends on how accurately this information can be determined. These techniques scale to any level of detail that is available, allowing the data race reports generated by previous methods to be refined, and allowing any sources of additional information to be used (such as a static analysis of the program that is already being performed for another purpose). Furthermore, we also believe that our techniques might be easily modified to accurately locate general races.

To test the effectiveness of our techniques, we have built a prototype race detector. The prototype automatically instruments parallel C programs that run on the Sequent Symmetry multiprocessor, and performs data race validation and ordering by analyzing the traces produced by this instrumentation. Experiments show that, in all of the test programs analyzed, our techniques narrowed the large number of races that the programmer must analyze down to four or less.

The final result of this work is both a theory in which to reason about current and future race detection methods, and simple but effective techniques for locating those races that are direct manifestations of program bugs. The practical applications of this work are new tools that aid the programmer in debugging programs containing race conditions.

1.3. Thesis Organization

This thesis is organized as follows. In Chapter 2 we outline previous work related to detecting race conditions. We also present an example that illustrates how race artifacts occur and why they present problems.

Chapter 3 presents our formal model for reasoning about race conditions. The model consists of three parts: the first part is a notation for describing the execution of a shared-memory parallel program (Section 3.1), the second part represents information that can be reasonably recorded about the execution (Section 3.2), and the third part characterizes behavior that the execution had the potential of exhibiting (Section 3.3).

The model is used in Chapter 4 to characterize different types of race conditions: general races and data races. Sections 4.1 and 4.2 present examples of these races and argue that each pertains to a different class of parallel programs. Section 4.3 discusses some fundamental differences between general races and data races.

Chapter 5 then proves results regarding the complexity of race detection. We formulate the race-detection problem in terms of binary relations, and prove the complexity of deciding these relations. The implications of these results as they impact general race and data race detection are also discussed.

Chapters 6 through 8 present techniques for accurate data race detection. A two-phase approach is presented. Chapter 6 discusses the first phase, which is essentially identical to previous methods, and detects the apparent data races. Chapters 7 and 8 then discuss how these races can be validated and ordered by a second phase. These chapters each begin by proving theoretical results and end by outlining algorithms for implementing the technique.

Chapter 9 discusses our prototype race detector and the results of experiments involving test programs that contained data races. Chapter 10 presents concluding remarks and topics for future work.

Brief explanations of notation defined in this thesis appear in the glossary.

Chapter 2

RELATED WORK

This chapter reviews previous work related to race condition detection. Race detection schemes perform either *static analysis* or *dynamic analysis*. Static analysis examines the program text, performing a conservative analysis and detecting a superset of all races that could be possibly exhibited by the program. In contrast, dynamic analysis examines a particular execution of the program and detects races exhibited only by that execution. Although this thesis addresses dynamic detection, we include a short review of static methods for completeness, and then concentrate on dynamic methods.

We also present an example program execution that exhibits data races and show how previous data race detection methods can report race artifacts. This example illustrates the problems caused by reporting artifacts and justifies the need for accurate race detection techniques.

2.1. Static Analysis

Static analysis methods for race detection examine only the program text, and assume that all execution paths through the program are possible. Under this assumption, determining whether sections of code may execute concurrently (or in some other order) requires examining only the explicit synchronization in the program (this assumption would always be correct if the program contained no conditionally executed code). Static analysis therefore examines how the program's synchronization might allow such potential orderings. An exception to this approach is the work by Young and Taylor[77], which employs symbolic execution to discover more information about possible execution paths. Using this ordering information, and a conservative analysis of which shared variables may be read and written in each section of code, data races and general races can be detected.

Taylor showed that, for Ada programs containing no branches or conditionally executed code, the problem of computing the ordering information is NP-complete[72, 73]. Given this NP-completeness result, two different approaches to static analysis have been developed. First, some methods traverse the space of all possible states that the program may enter. This state space can either be constructed explicitly, by building a graph[3, 4, 46, 47, 52, 72, 74, 75], or implicitly, by constructing a representation of the state space (such as a formal language or a petri-net)[5, 36, 67]. In the general case, these methods have exponential time complexity, and in some cases, exponential space complexity as well. Second, other static analysis methods perform a data-flow analysis of the program to discover potential event orderings[6, 10, 11, 13, 57, 58, 62, 70, 71]. These methods have polynomial time and space complexity, but are less accurate than the state-space methods, sometimes reporting races that the program could never exhibit (and that the state-space methods would never report).

Static analysis has also been used to complement dynamic methods. Static analysis can sometimes rule out the possibility of races between some sections of the program, precluding the need for tracing these program sections for dynamic analysis[2, 23-25, 75]. Static analysis can also compute input data that might manifest a race[63], allowing dynamic analysis to attempt to verify the existence of that race.

2.2. Dynamic Analysis

Unlike static analysis, dynamic analysis detects the race conditions exhibited by a particular execution of the program. Dynamic analysis is especially useful for debugging since precise information about manifestations of particular bugs is available. Below we outline previous work on dynamic data race and general race detection. We first discuss the similarities and differences of the various methods and then present the unique details of each. Finally, we present an example that focuses on the simple type of analysis that is common to all methods. We show why this analysis can report race artifacts, and discuss other aspects of previous work.

2.2.1. Common Characteristics

All previous race detection methods collect essentially the same information about the execution and conceptually analyze it in the same way. Below we outline this analysis. As discussed later, their main differences lie in when this information is collected and analyzed.

Race condition detection methods instrument the program so that information about program *events* is collected during execution. An event represents an execution instance of either a single synchronization operation (a *synchronization event*) or code executed between two synchronization operations (a *computation event*). Two kinds of information are recorded: (1) for each event in the execution, the sets of shared memory locations that are read and written by the event (called the *READ* and *WRITE* sets), and (2) the relative order in which some events execute.

The main part of race detection involves analysis of the recorded relative execution order. All methods conceptually represent this information with a DAG, which we call the *ordering graph*. Each node of the ordering graph represents an event[1]. Edges are added to show the order in which events belonging to the same process execute; an edge is added from each node to the next node belonging to the same process. Edges are also added between some pairs of synchronization nodes (involving the same synchronization variable) to indicate their relative execution order. The purpose of these edges is to show event orderings (in the particular execution being traced) that were caused by explicit synchronization (details appear later). No edge between two nodes implies that no explicit synchronization constrained their execution order.

Data race detection attempts to determine whether intended critical sections executed non-atomically (or had the potential of doing so). Non-atomicity can occur only when pairs of computation events have a *data conflict* (i.e., access common shared variables that at least one modifies) and can potentially execute concurrently. Data-

[1] 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[50, 65].

conflicting events are easily located by analyzing the *READ* and *WRITE* sets. Events that had the potential of executing concurrently can be located by analyzing the ordering graph. All previous methods search for pairs of events that are *unordered* by the graph. Two events are unordered by the graph if there is no path connecting the nodes representing the events.

General race detection attempts to determine whether the given execution was potentially non-deterministic. An execution is potentially non-deterministic if two data-conflicting events could have executed in any order. The key to general race detection is determining the event orderings that are guaranteed to be exhibited by any execution of the program on the given input. The general race detection method by Emrath, Ghosh, and Padua[24, 25] computes these guaranteed orderings by analyzing the execution's explicit synchronization and transforming the ordering graph. An edge in the transformed graph indicates that the synchronization semantics would force the node at the tail of the edge to precede the node at the head in any execution. Events unordered by the transformed graph are assumed to have potentially executed in any order.

All data race and general race detection methods analyze the ordering graph in the same way: races are reported between data-conflicting events that are unordered by the graph. These events are assumed to have either potentially executed concurrently (in the case of data race detection) or in any order (in the case of general race detection). This assumption follows from the observation that the ordering graph shows event orderings guaranteed by the execution's *explicit* synchronization. However, as we illustrate later in this chapter, this assumption is not always correct, and can lead to the detection of race artifacts.

2.2.2. Differences

The main differences among the race detection methods lie in the way information is collected and analyzed. Two approaches exist: *post-mortem* and *on-the-fly* analysis. Both post-mortem and on-the-fly methods instrument the program to collect information required for race detection. Post-mortem methods detect races after the execution has ended by analyzing trace files produced during execution. On-the-fly methods detect races by an on-going analysis as the program executes, without producing trace files.

Post-mortem methods use three phases. The first phase instruments the program to record during execution the *READ* and *WRITE* sets for each event, as well as the relative execution order among synchronization events involving the same synchronization variable. In the second phase, the instrumented version of the program is executed, writing this information to trace files. After execution is complete, the final phase constructs and analyzes the ordering graph to detect races. Once the instrumentation phase is complete, the last two phases can be repeated for any number of executions. Most post-mortem methods detect data races[1, 2, 16, 50]; only one method detects general races[23-25].

In contrast, on-the-fly methods use only two phases. As with post-mortem methods, the first phase instruments the program to collect information during execution. Unlike post-mortem methods, this information is not collected for later use, but is analyzed as execution progresses. The ordering graph and *READ* and *WRITE* sets are encoded in memory[2] so that they can be updated and accessed quickly during execution and discarded as they become obsolete. The only purpose of encoding this information is to reduce space overhead; race detection is still accomplished by locating events unordered by the ordering graph. The thrust of previous work has been the development of efficient encoding schemes. Several schemes have been proposed, but most buffer only enough information to guarantee detection of one race involving each shared variable[17, 19-22, 37, 49, 64, 65], leaving some races undetected (however, Choi and Min[17] propose a method, discussed further in Chapter 8, for re-executing the program to reproduce the undetected races). On-the-fly methods have primarily been used to detect data races[17, 19, 20, 22, 37, 51, 56, 64, 65, 69], although they can be used to detect general races for programs that use no synchronization other than **fork**/**join**.

On-the-fly methods have the advantage over post-mortem methods that large trace files are not generated, but have the disadvantages of incurring more space overhead during execution, not detecting some races, and discarding information that shows how the races occurred. The execution overhead can render on-the-fly methods unusable in some instances. Each shared variable being monitored for races incurs both space and time overhead. The space overhead is 2–20 times the size of the variable being monitored. The time overhead is incurred by performing race checks at each shared-memory access. All accesses to the same variable are serialized, introducing central bottlenecks into the execution (as discussed later, this serialization also provides extra ordering information that can be used for race detection). In systems where information about the execution is being collected for other purposes (such as debugging), post-mortem methods can usually be applied with little additional overhead. However, when long-running executions produce very large traces, post-mortem methods are generally unusable.

2.2.3. Details of Each Method

Conceptually, the various data race and general race detection methods differ mainly in the types of synchronization primitives supported and in how edges between synchronization nodes in the ordering graph are added. Although the on-the-fly methods also contain techniques for encoding the ordering graph in memory, we concentrate on the structure of the graph and how it is analyzed to detect races (further discussion of on-the-fly methods appears in the next sub-section and in Chapter 8). Below, we briefly outline these aspects of previous methods.

Allen and Padua[2] present a post-mortem data race detection method for Fortran programs that introduce parallelism with the **doall** construct, and synchronize with **test** and **testset** primitives. The **doall** construct is identical to a **do** loop except that each iteration of the loop is spawned as a separate process that executes concurrently with all other iterations of the loop. The **test(***x***)** primitive waits until the value of the synchronization variable x is greater than the iteration number of the enclosing **doall** loop. The **testset(***x***)** primitive increments the value of the synchronization variable x. An edge is drawn in the ordering graph from a **doall** node to the first node representing

---

[2] Some methods store the inverse of the *READ* and *WRITE* sets[17, 19, 37, 56].

each iteration of the loop. Similarly, an edge is drawn from the last node of each loop iteration to the **end doall** node corresponding to the end of the loop. An edge is also drawn from a node representing a **testset** operation to the node representing the **test** operation that it allowed to proceed. They do not specify how to draw these edges when there is more than one **test** or **testset** operation in a given loop iteration. To reduce the amount of instrumentation required, they employ a static analysis of the program. Events involving data races that can be detected statically are not traced. In addition, an event that could be affected by data-racing events is not traced, since tracing cannot prove the absence of a potential data race involving that event. Our work on data race ordering also considers how events potentially affect one another, and we contrast it with Allen and Padua's work in Chapter 8.

Miller and Choi[50] describe a parallel program debugger that incorporates post-mortem data race detection. They consider C programs that use **fork**/**join** and counting semaphores. An edge is drawn from a **fork** node to the first node representing each child of the **fork**, and from the last node in each child to the corresponding **join** node. To draw edges between nodes representing semaphore operations, they pair the i^{th} **V** operation on each semaphore with the i^{th} **P** operation on the same semaphore. An edge is then drawn from each **V** node to the **P** node with which it is paired.

Dinning and Schonberg[19, 20, 64, 65] describe on-the-fly techniques for detecting data races in executions of programs using **fork**/**join** and arbitrary synchronization (such as barriers, events, Ada rendezvous, and semaphores). As above, they (conceptually) construct an edge from each **fork** node to the first node in each child of the **fork**, and from the last node in each child to the corresponding **join** node. For other synchronization operations, they distinguish between *synchronous* and *asynchronous* coordination. Synchronous coordination occurs when neither of the two processes synchronizing may proceed beyond the synchronization point until both have reached it. In this case, a doubly directed edge between the two nodes involved in the coordination is drawn (such edges may introduce degenerate cycles, which are ignored). Asynchronous coordination occurs when one event (the sender) may proceed immediately but the other (the receiver) may not execute beyond the synchronization point until the sender has arrived. For asynchronous coordination, an edge is constructed from the node representing the sender to the node representing the receiver. They do not specify how to pair senders with receivers.

Dinning and Schonberg[22] also extend their techniques to uncover some races that are hidden by critical sections. Because an edge is added between the synchronization operations implementing critical sections (such as from a **V** to a **P** operation), the graph only allows detection of races that occurred under the particular order in which critical sections executed. Races that would have occurred had two critical sections executed in a different order remain hidden. To uncover these races, they propose omitting edges between synchronization operations that implement critical sections. However, when it is not possible for two critical sections to have executed in any order, race artifacts can be reported. They also present a static analysis, based on program slicing, for determining which parts of the program may cause such artifacts.

Hood, Kennedy, and Mellor-Crummey[37] describe an on-the-fly technique for detecting data races in PCF Fortran programs that use parallel **do** loops with **send** and **wait** primitives. Edges are added to the graph in the same way as the method by Dinning and Schonberg. However, the semantics of PCF Fortran dictate that exactly one **send** and one **wait** operation must be issued for each synchronization variable. Under this restriction, pairing **send** operations with **wait** operations is straightforward. In addition, Mellor-Crummey[49] presents a technique called *Offset-Span Labeling* for programs that use only **fork**/**join**. Offset-Span labeling is a technique for encoding the ordering graph that incurs lower space overhead than other methods.

Steele[69] and Nudler and Rudolph[56] also present on-the-fly schemes for programs that use only **fork**/**join**. They conceptually construct the ordering graph in the same way as all other methods, and differ in the techniques used to encode the graph in memory.

Choi and Min[17, 51] present techniques that address both the execution-time overhead of on-the-fly race detection and the problem of undetected races. They outline a hardware-based scheme for reducing the amount of execution-time overhead by using state information maintained by (a modified version of) the underlying cache coherence protocol[51]. The resulting race detection method is essentially the same as that of Dinning and Schonberg[19, 20]. To address the problem of undetected races, they also show how to guarantee deterministic re-execution of the program up to the point of the first detected race, allowing additional instrumentation to be added that locates the originally undetected races[17]. Their work has some similarities to ours; a more detailed discussion appears in Chapter 8.

The general race detection method described by Emrath, Ghosh, and Padua[23-25] considers programs that use **fork**/**join** and event style synchronization (using the **Post**, **Wait**, and **Clear** primitives). This method attempts to locate exactly those events whose execution order was not guaranteed by the program's synchronization. As with the data race detection methods described above, an ordering graph is constructed. Edges are added to this graph in the same way as described above for **fork**, **join**, and events belonging to the same process. The method differs from data race detection in its handling of other synchronization events (**Post**, **Wait**, and **Clear**). Edges are added between nodes representing these events to show orderings that the program is *guaranteed* to have exhibited (on the given input). They present both an algorithm for approximating and exhaustively computing these orderings. In their approximation algorithm, for each **Wait** node, all **Post** nodes that might have triggered that **Wait** are identified. A **Post** might trigger a **Wait** if there is no path in the graph from the **Wait** to the **Post** (which would indicate that the **Wait** must have preceded the **Post**), and no path from the **Post** to the **Wait** that includes a **Clear** node. Edges are then added from the closest common ancestors of these **Post**s to the **Wait**. In their exhaustive algorithm, every possible event ordering is considered, allowing the guaranteed orderings to be computed. The graphs resulting from these algorithms are interpreted as showing a guaranteed ordering between two events iff they are ordered by the graph. Two events are therefore said to not have a guaranteed execution order iff they are unordered by the graph. However, the graph constructed by their approximation algorithm sometimes omits some guaranteed orderings, possibly causing race artifacts to be reported. In Chapter 5 we prove that the problem of exactly computing the guaranteed orderings is NP-hard.

Helmbold, McDowell, and Wang[35] also present algorithms for computing guaranteed orderings for programs that use general semaphores. As above, these algorithms transform the ordering graph. They achieve polynomial running time by computing *safe orderings*, which are conservative approximations of the guaranteed orderings. The orderings given by the resulting graph can be used to (conservatively) detect both general races and data races. As above, the conservative nature of these orderings can cause race artifacts to be reported.

2.3. Problem Motivation

All of the previous methods described above share similar drawbacks. Although they provide valuable tools for determining whether or not a program execution is race-free, most provide little assistance, in general, for actually locating the program bugs responsible for the races. This problem stems from two sources. First, as illustrated below, previous methods can generate race reports containing artifacts, with no indication of which reported races are directly caused by program bugs. Second, this problem is further complicated by the informal treatment of races: since the meaning of the reported races is not characterized, it is unclear how to relate the race reports back to the behavior of the program. It is therefore difficult for the programmer to use the race reports for debugging the cause of the races. Below, we illustrate and discuss these problems for data race detection; analogous problems also exist for general race detection.

2.3.1. Example

To illustrate why data race artifacts can be reported, and why they present a problem, consider the program fragment in Figure 2.1. 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. A correct execution of this program should therefore exhibit no data races.

However, assume the ‘‘remove’’ operations do not properly synchronize their accesses to the shared queue. An ordering graph for one possible execution is shown (the internal lines only illustrate the data races and are not part of the graph). In this execution the ‘‘remove’’ operations execute concurrently (during the first loop iteration), causing the right child to correctly remove the second record, but the left child to incorrectly remove the lower bound from the first record and the upper bound from the second. The left child thus proceeds to operate (erroneously) on region [1,39].

In this graph, no paths connect any nodes of the left child with any nodes of the right child, meaning that no explicit synchronization constrained their execution order. The type of analysis that previous methods perform (i.e., scanning the ordering graph), would report three data races: two between the *work* events (shown by the dotted and dashed lines) and one between the ‘‘remove’’ events (shown by the solid line).

fork

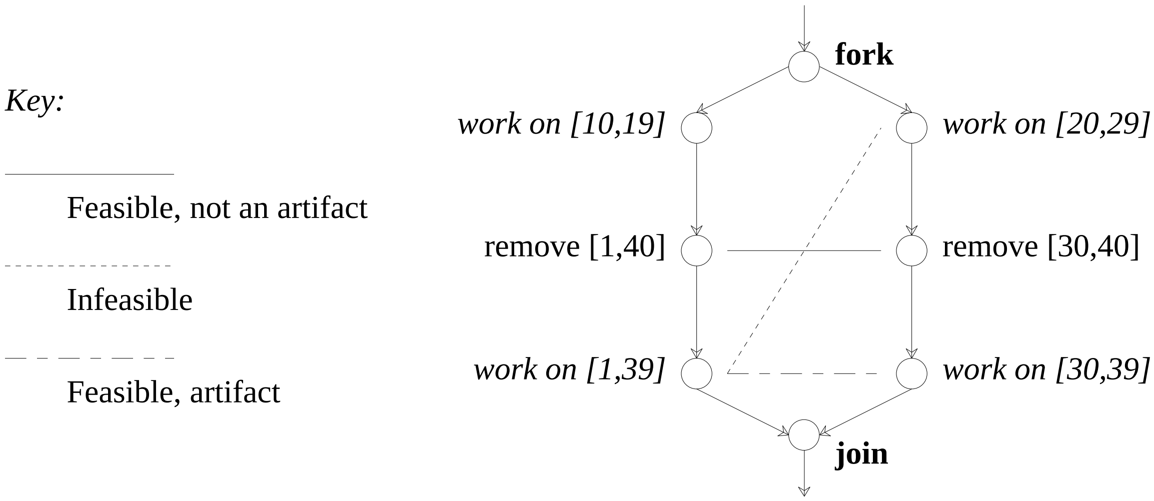
> *work on region [10,19]*
> **loop**
> remove (L,U) from Queue
> *work on region [L,U–1] of shared array*
> **while** QueueNotEmpty

> *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]

(a)



(b)

Figure 2.1. (a) example program that can exhibit data races, and (b) ordering graph for one execution (annotated with data-race information)

The race report between the ''remove'' events is not an artifact; it was directly caused by the bug (the omitted synchronization). The non-artifact status of this race can be determined by noticing that (1) the race is feasible, involving events that either did execute concurrently or had the potential of doing so, and (2) the race involves events that were not performed only as a result of the outcome of another race.

However, the two race reports involving *work* events are both artifacts. The data race shown by the dotted line is infeasible, since it involves events that never could have executed concurrently. For the accesses to [1,39] 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 [1,40] would not have been removed (since the two ''remove'' operations would not overlap), and a different array region would have been accessed.

Although the data race shown by the dashed line is feasible (it involves events that actually did execute concurrently), it is nonetheless an artifact. The access to [1,39] was a result of the preceding ''remove'' executing non-atomically, leaving the program's data in an inconsistent state.

Even though the simple approach of analyzing the ordering graph would report three races (two of which are artifacts), some methods have the potential of limiting the number of artifacts reported. Since on-the-fly methods serialize all accesses to a given memory location, they have information in addition to the ordering graph about the order in which individual shared-memory accesses occurred. As suggested by Choi and Min[17], these methods could stop the program after the first race is detected (even though earlier races may remain undetected). However, obtaining this extra information incurs overhead at each shared-memory access and introduces central bottlenecks into the execution. Although this technique may prevent artifacts from being reported, this issue has not yet been investigated. The results presented later in this thesis can be used to reason about this (or any other) approach to prove that artifacts are never reported or to characterize those that are.

2.3.2. Discussion

Reporting the above race artifacts to the programmer can complicate the debugging process; only a single bug exists, but three data races are reported. Previous work has not considered this problem. Race reports are further complicated by the lack of a formal semantics for race conditions; the meaning of the race reports is unclear, making it difficult to reason about artifacts.

If the example had been more complex, perhaps creating other children, there may have been many nodes in the graph representing the array accesses, and many data race artifacts would have been reported. Since the artifacts are not direct manifestations of program bugs but are caused only by other races, reporting them among the non-artifact races obscures the location of the bug. Artifacts can result whenever the values of shared variables are used (directly or indirectly) in conditional expressions or in expressions that determine which shared locations are accessed (e.g., shared array subscripts or pointer expressions). We believe that race artifacts are a problem because a large class of programs use shared data in this way. Some artifacts (such as the infeasible data race shown above) occur because of situations reminiscent of the programmer using shared variables to implement synchronization, instead of using explicit synchronization constructs. However, we should emphasize that race artifacts can occur even when the programmer *does not* intentionally use shared variables for synchronization, as the above example illustrates. As the example also shows, non-artifact data races can appear anywhere in the execution, making it difficult to locate them by inspection alone. We therefore conclude that automatic techniques for accurate race detection, that analyze how shared data flowed through the execution (causing some events to affect the outcome of others), are necessary.

The fact that previous work has not precisely defined the notion of a race (or a race artifact) exacerbates the race-artifact problem. Without a formal statement of the race-detection problem, developing correct techniques for dealing with race artifacts is complicated. Previous work has only characterized races informally[19, 24, 25] or not at all. For example, data races have only been defined as occurring when two blocks of code access common shared variables and ‘‘can potentially execute concurrently’’[19]. General races have been defined as occurring when there is no ‘‘guaranteed run-time ordering’’[24] between two shared-memory accesses to the same location. Such definitions refer to sets of alternate orderings that had the potential of occurring (those in which the accesses execute concurrently or in an order different than originally occurred), but these sets have not been explicitly defined. By not characterizing precisely what the race reports mean, it is not always clear what races are (and are not) being detected, or how to use the resulting reports to debug the program.

The purpose of this thesis is to address these concerns. Our first main goal is to formally define and explore the race-detection problem in terms of a model for reasoning about race conditions. Then, techniques for accurate data race detection are developed and proven correct, entirely in terms of the model.

Chapter 3

MODEL FOR REASONING ABOUT RACE CONDITIONS

In this chapter we present a formal model for reasoning about race conditions. In subsequent chapters we will formulate and reason about the race-detection problem entirely in terms of this model. Doing so not only allows us to unambiguously characterize race conditions, but also allows our techniques to be proven correct more easily than when reasoning purely on an intuitive level. Our model consists of three parts, representing the actual, observed, and potential program behaviors. The actual behavior precisely represents properties of the execution, the observed behavior represents only information that can be reasonably recorded about the execution, and the potential behavior shows alternate orderings in which the execution's events could have been performed. Representing all of these behaviors is necessary for reasoning about race conditions and techniques for race detection.

3.1. Actual Behavior

The first part of our model is simply a notation for representing the actual behavior of a shared-memory parallel program executing on a sequentially consistent multi-processor[39]. This part of the model contains objects that represent a program execution (such as which statements are executed and in what order) and axioms that characterize properties those objects must possess. Below, we first outline the basic model, and then describe the axioms for programs that use **fork**/**join** and general semaphores. Other types of synchronization are easily accommodated by including the appropriate axioms.

3.1.1. Basic Model

The first part of our model is based on Lamport's theory of concurrent systems[41, 42], 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*, which represent instances of operations performed by the system. Two relations on operation executions, *precedes* ($\longrightarrow$) and *can causally affect* ($\dashrightarrow$), describe a system execution. For two operation executions a and b, $a \longrightarrow b$ 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 multi-processors with sequentially consistent memory systems[39]. Sequential consistency ensures that shared-memory accesses behave as if they were all performed atomically and in some linear order (that is consistent with the order specified by the program). Analogous to a system execution, we define a *program execution*.

Definition 3.1

A *program execution*, P, is a triple $\langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$, where

(1) E is a finite set of *events*,

(2) $\xrightarrow{T}$ is the *temporal ordering* relation (defined over E), and

(3) $\xrightarrow{D}$ is the *shared-data dependence* relation (also defined over E).

P is an *actual program execution* if it represents an execution that the program at hand actually performed. ■

Definition 3.2

An *event* $e \in E$ represents

(1) a set of execution instances of one or more consecutively-executed program statements, and

(2) the sets of shared variables[1] read and written by those statements, denoted by $READ(e)$ and $WRITE(e)$. ■

Because we define an event to represent the statement instance *and* the shared variables accessed, two events are equivalent iff they represent the same execution instance of the same statements in which the same shared variables are accessed. Because the *READ* and *WRITE* sets do not contain the values read or written (but only the addresses), two such events can be equivalent even if they access different values.

When the underlying hardware guarantees sequential consistency, any two events that execute concurrently can affect one another (i.e., $a \not\longleftrightarrow b \Leftrightarrow a \dashrightarrow b \wedge b \dashrightarrow a)$[2]. A single relation is then sufficient to describe the temporal aspects of a system execution. The temporal ordering relation serves this purpose; $a \xrightarrow{T} b$ means that a completes before b begins, and $a \overset{T}{\not\longleftrightarrow} b$ means that a and b execute concurrently (i.e, neither completes before the other begins). Another useful way of viewing events and their temporal ordering is by modeling each event, e, as possessing a unique *start time* (e_s) and *finish time* (e_f)[40]. The temporal ordering can then be described by a total ordering on the start and finish times of all events, called a *global-time model*: $a \xrightarrow{T} b$ iff $a_f < b_s$, and $a \overset{T}{\not\longleftrightarrow} b$ iff $a_s < b_f \wedge b_s < a_f$. We should emphasize that the $\xrightarrow{T}$ relation describes the order in which events *actually* execute during a particular execution: $a \overset{T}{\not\longleftrightarrow} b$ means that a and b actually executed concurrently, it does not mean that a and b could have executed in any order.

---

[1]We use the term *shared variable* to refer to one or more memory locations that reside in shared memory.

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

We also replace the $\dashrightarrow$ relation with the shared-data dependence relation, $\xrightarrow{D}$. The $\xrightarrow{D}$ relation shows when one event can causally affect another through accesses to shared memory. Events in different processes can affect each other 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 and 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 then c references another variable that b later references. This definition of data dependence is somewhat nonstandard since we consider transitive dependences involving flow-, anti-, and output-dependences[26, 38], and do not explicitly state the variables involved.

3.1.2. Axioms

A program execution is simply a notation for describing the execution of a shared-memory parallel program. Following Lamport, our model also contains several axioms that describe properties every program execution must possess. Furthermore, the model so far does not describe any of the synchronization aspects of a program execution. We extend the general model by imposing some structure on the set of events, E, and by adding axioms that describe the semantics of synchronization operations. We consider programs that use **fork/join** and general semaphores. However, other synchronization constructs can be easily accommodated by adding the appropriate axioms.

The temporal ordering and shared-data dependence relations must satisfy the following axioms:

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

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

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

Axioms (A1) and (A2) force $\xrightarrow{T}$ to be consistent, and are equivalent to stating that $\xrightarrow{T}$ is defined by a total ordering of the start and finish times of all events (i.e., a global-time model). Axiom (A3) represents the law of causality: a cannot affect b if b precedes a in time[3].

To describe the presence of synchronization operations, we distinguish between two different types of events. A *synchronization* event is an execution instance of some synchronization operation. A *computation* event is an execution instance of a group of statements that executed consecutively, none of which are synchronization opera-

[3] Axioms (A1), (A2), and (A3) are identical to Lamport's. However, Lamport also includes another axiom stating that if $a \longrightarrow b \dashrightarrow c \ \vee \ a \dashrightarrow b \longrightarrow c$ then $a \dashrightarrow c$. We omit this axiom since it follows directly from axioms (A1) and (A2) for the class of system executions being considered (those having global-time models).

tions. Any arbitrary grouping of consecutively executed statement instances that does not include a synchronization operation defines a computation event.

We assume that throughout a program execution, a (perhaps variable) number of processes exist, and that each event belongs to some process. E_p denotes the set of events in the execution of process p, and $e_{p,i}$ denotes the i^{th} event in process p. The following axiom describes the linear ordering imposed on events belonging to the same process:

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

We also assume that each process 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. A **fork** event, $Fork_{p,i}$, is assumed to precede all events in the child processes that 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 $Join_{p,i+k}$,

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

To describe program executions that use general semaphores, we distinguish between **P** events and **V** events. The set of all **P** and **V** events on semaphore S is denoted by $E_{P(S)}$ and $E_{V(S)}$, respectively. We assume that in any program execution the semaphore invariant[34] is always maintained. For general 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, the $\xrightarrow{T}$ relation must satisfy the following axiom:

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

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

This version of axiom (A6) assumes that the initial value of each semaphore is zero. An arbitrary initial value, m, for some semaphore can be described by appropriately modifying the axiom, or by creating an artificial process that contains m V-events that precede all other events.

3.1.3. Higher-Level and Single-Access 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 it is sometimes 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*.

Definition 3.3

A *higher-level view* of a program execution $P = \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$ is another program execution $\mathcal{P} = \langle \mathcal{E}, \xrightarrow{\mathcal{T}}, \xrightarrow{\mathcal{D}} \rangle$, where

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

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

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

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

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 lower-level 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 valid program execution. When we refer to a higher-level view, this assumption is always implied.

It is also useful to sometimes view the execution at a very low level of detail. For this purpose we introduce a *single-access view*.

Definition 3.4

A *single-access* view of a program execution $P = \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$ is another program execution $P_S = \langle E_S, \xrightarrow{T_S}, \xrightarrow{D_S} \rangle$, where each computation event in E_S comprises at most one shared-memory access and P is a higher-level view of P_S. ■

Single-access views are useful when reasoning about the order in which individual shared-memory accesses were performed. We will make use of both higher-level and single-access views in the proofs of our results.

3.2. Observed Behavior

A program execution is a convenient notation for representing an execution of a shared-memory parallel program. However, a program execution captures complete information about the execution in the sense that $\xrightarrow{T}$ always shows the relative execution order between *any* two events, and $\xrightarrow{D}$ shows the *actual* shared-data dependences exhibited by the execution. In practice, recording such complete information is impractical, and existing dynamic methods record only a subset of this information. We now incorporate such partial information into our model by defining an *approximate program execution*, which consists of approximate counterparts to the temporal ordering and shared-data dependence relations. The accurate data race detection techniques presented in Chapters 7 and 8 are based on approximate program executions. Our intent here is to show how information recorded for race detection can be represented in our model, rather than to discuss details of program instrumentation.

3.2.1. Approximate Temporal Ordering

Existing methods record the temporal ordering among only a subset of the synchronization events. For example, 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 avoided. However, the lack of complete ordering information sometimes renders it impossible to determine the actual order in which two events were performed. Arbitrarily not recording the relative execution order between any pair of events would result in meaningless recorded information. We therefore require that the lack of ordering information between a pair of events occurs only in certain situations. To represent this incomplete ordering, we define an *approximate temporal ordering* relation, $\xrightarrow{\hat{T}}$.

Definition 3.5

An *approximate temporal ordering* relation, $\xrightarrow{\hat{T}}$, is a conservative approximation to $\xrightarrow{T}$ (i.e., $\xrightarrow{\hat{T}} \subseteq \xrightarrow{T}$) that possesses the following two properties:

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

(2) if $a \overset{\hat{T}}{\not\longleftrightarrow} b$ then explicit synchronization did not prevent a and b from executing concurrently. ■

The ordering information recorded by all existing methods possesses these properties. The first property states that the recorded ordering must be consistent with the actual ordering. The second property stipulates when the ordering between a and b may not be recorded. In this case, if the order between two events is not observed, it must be because they either executed concurrently or were not prevented by explicit synchronization from doing so. These two properties simply impose consistency constraints on the recorded ordering information, allowing meaningful information regarding the actual ordering to be extracted from it.

3.2.2. Approximate Shared-Data Dependences

Although existing methods do not attempt to record shared-data dependences, dependences can be approximated given the temporal ordering and the *READ* and *WRITE* sets which are recorded for each computation event.

Definition 3.6

An *approximate shared-data dependence* relation, $\xrightarrow{\hat{D}}$, is a conservative approximation to $\xrightarrow{D}$ (i.e., $\xrightarrow{\hat{D}} \supseteq \xrightarrow{D}$). ■

The $\xrightarrow{\hat{D}}$ relation conservatively shows what the actual shared-data dependences were; it must be conservative to avoid overlooking any of the actual dependences. For example, consider two events, a and b, that access common shared variables (that at least one modifies). If $a \xrightarrow{\hat{T}} b$, then we can determine with certainty that a direct shared-data dependence exists 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 unknown), and we must 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. Transitive shared-data dependences can be similarly estimated. Algorithms for computing $\xrightarrow{\hat{D}}$ from the *READ* and *WRITE* sets are presented in Chapter 7.

3.2.3. Approximate Program Executions

The approximate relations described above form an *approximate program execution*, representing information that can be reasonably recorded about the execution.

Definition 3.7

An *approximate program execution*, $\hat{P}$, is a triple $\langle E, \xrightarrow{\hat{T}}, \xrightarrow{\hat{D}} \rangle$, where

(1) E is a finite set of events,

(2) $\xrightarrow{\hat{T}}$ is an approximate temporal ordering relation described above, and

(3) $\xrightarrow{\hat{D}}$ is an approximate shared-data dependence relation described above. ■

Approximate executions form the link between the real execution and the model. An approximate execution, $\hat{P}$, can actually be constructed from a trace of the execution. Characteristics of the actual program execution, P, can then be inferred by using the properties described above.

3.3. Potential Behavior

Actual and approximate program executions describe behavior that a particular execution *actually* exhibited. However, to characterize race conditions, it is necessary to also describe behavior that the execution *could have* exhibited. Previous work has not carefully considered this issue. Race conditions have typically been defined only as data-conflicting accesses ‘‘that can execute in parallel’’[19] or whose execution order is not ‘‘guaranteed’’[24, 25]. To address this concern, the third part of our model characterizes sets of *feasible* program executions, which represent other executions that had the potential of occurring. We first discuss several possible ways in which these sets can be defined, and then give conditions sufficient for guaranteeing the feasibility of a program execution. In Chapters 4 and 5 we use these sets to formulate the race-detection problem and prove complexity results. In Chapters 7 and 8 we use the sufficient conditions to develop our data race validation and ordering techniques.

3.3.1. Program Execution Prefixes

To characterize when a race condition exists between two events, a and b, in an actual program execution, $P = \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$, we must consider other program executions that also perform a and b. Characterizing such executions allows us to determine if a and b can potentially execute in an order different than in P. We will focus on program executions that are *prefixes* of P (defined below), as opposed to considering *all* possible executions of the program that also perform a and b.

Characterizing races by focusing only on prefixes of P has two advantages. First, the existence of a race between a and b then means that some prefix of the *observed* execution had the potential of exhibiting the race (i.e., would allow a different execution order between a and b). Trace data obtained from this execution thus provides valuable debugging information: the trace can be browsed to understand the execution history of the race. This approach is consistent with the goals of dynamic analysis (discussed in Chapter 2) in which trace information reflecting a particular execution is used for debugging. In contrast, if we were to also consider executions other than prefixes, we would have no information about how the observed execution differs from one in which a and b could have a different execution order. Second, determining whether an arbitrary execution exists that also performs a and b appears to be an undecidable problem. Defining a race to exist if *any* execution could perform a and b in a different order leads to an impractical detection problem.

Definition 3.8

A program execution $P' = \langle E', \xrightarrow{T'}, \xrightarrow{D'} \rangle$ is a *prefix* of a program execution $P = \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$ iff for each process p, the sequence of events in $E_p{}'$ (imposed by axiom (A4)) is a prefix of the sequence of events in E_p; i.e.,

(1) $E_p{}' \subseteq E_p$, and $\forall\, e_{p,i} \in E_p,\ e_{p,i} \xrightarrow{T} e_{p,i+1} \Rightarrow$

(a) $e_{p,i} \xrightarrow{T'} e_{p,i+1}$, or

(b) $\forall x \in E_p{}',\ e_{p,j} \not\xrightarrow{T'} x$ (i.e., $e_{p,j}$ is the last event in process p); or

(2) $E_p{}' = \emptyset$. ■

3.3.2. Sets of Alternate Program Executions

Given the actual program execution, P, we define three sets of program execution prefixes by considering successively fewer restrictions on each set. The first two sets are restricted to contain only program executions that are feasible. The first set contains all feasible program executions that exhibit the same shared-data dependences as P. The second set contains feasible executions for which no restrictions on their shared-data dependences are placed[4]. We denote these two sets by F_{SAME} and F_{DIFF}.

[4] The structure of these sets depends on the details of our definition of shared-data dependence. Although different definitions are possible (e.g., that characterize only flow dependences), they would not alter this structure in a significant way.

Definition 3.9

F_{SAME} is the set of program executions, $P' = \langle E', \xrightarrow{T'}, \xrightarrow{D'} \rangle$, such that

(1) P' represents an execution that the program could actually perform,

(2) $E' = E$, and

(3) $\xrightarrow{D'} = \xrightarrow{D}$. ■

Definition 3.10

F_{DIFF} is the set of program executions, $P' = \langle E', \xrightarrow{T'}, \xrightarrow{D'} \rangle$, such that

(1) P' represents an execution the program could actually perform,

(2) P' is a prefix of P, and

(3) $\xrightarrow{D'}$ is arbitrary. ■

These sets contain executions that are derived by considering all temporal orderings in which P's events (or a prefix of P's events) could have been performed. Since F_{DIFF} contains executions with different shared-data dependences, it captures more of these orderings than F_{SAME}, and is therefore a larger set. For example, Figure 3.1(a) shows an actual program execution, P, in which $a \xrightarrow{T} b$ and $a \xrightarrow{D} b$. The program execution P' shown in Figure 3.1(b) belongs to F_{SAME} because $a \xrightarrow{D'} b$ even though $a \not\xleftrightarrow{T'} b$ (the dependence can still occur if a updates S before b reads S). Figure 3.1(c) shows a program execution P' that belongs to F_{DIFF} (but not F_{SAME}) because b reads S before a updates S, resulting in $b \xrightarrow{D'} a$. Both of these program executions are feasible because the program is capable of producing them.

Finally, like F_{DIFF}, the third set also contains program executions with arbitrary shared-data dependences, but they are no longer required to be feasible; they are only required to obey the synchronization axioms. We denote this set by F_{SYNC}. Characterizing this set is useful since it provides an approximation to F_{SAME} and F_{DIFF} that involves only the semantics of the execution's explicit synchronization. Moreover, since previous dynamic race condition detection methods analyze only explicit synchronization, this is the set of alternate executions that they implicitly assume.

Definition 3.11

F_{SYNC} is the set of program executions, $P' = \langle E', \xrightarrow{T'}, \xrightarrow{D'} \rangle$, such that

(1) $\xrightarrow{T'}$ obeys the semantics of P's explicit synchronization (axioms (A4) – (A6)),

(2) P' is a prefix of P, and

(3) $\xrightarrow{D'}$ is arbitrary. ■

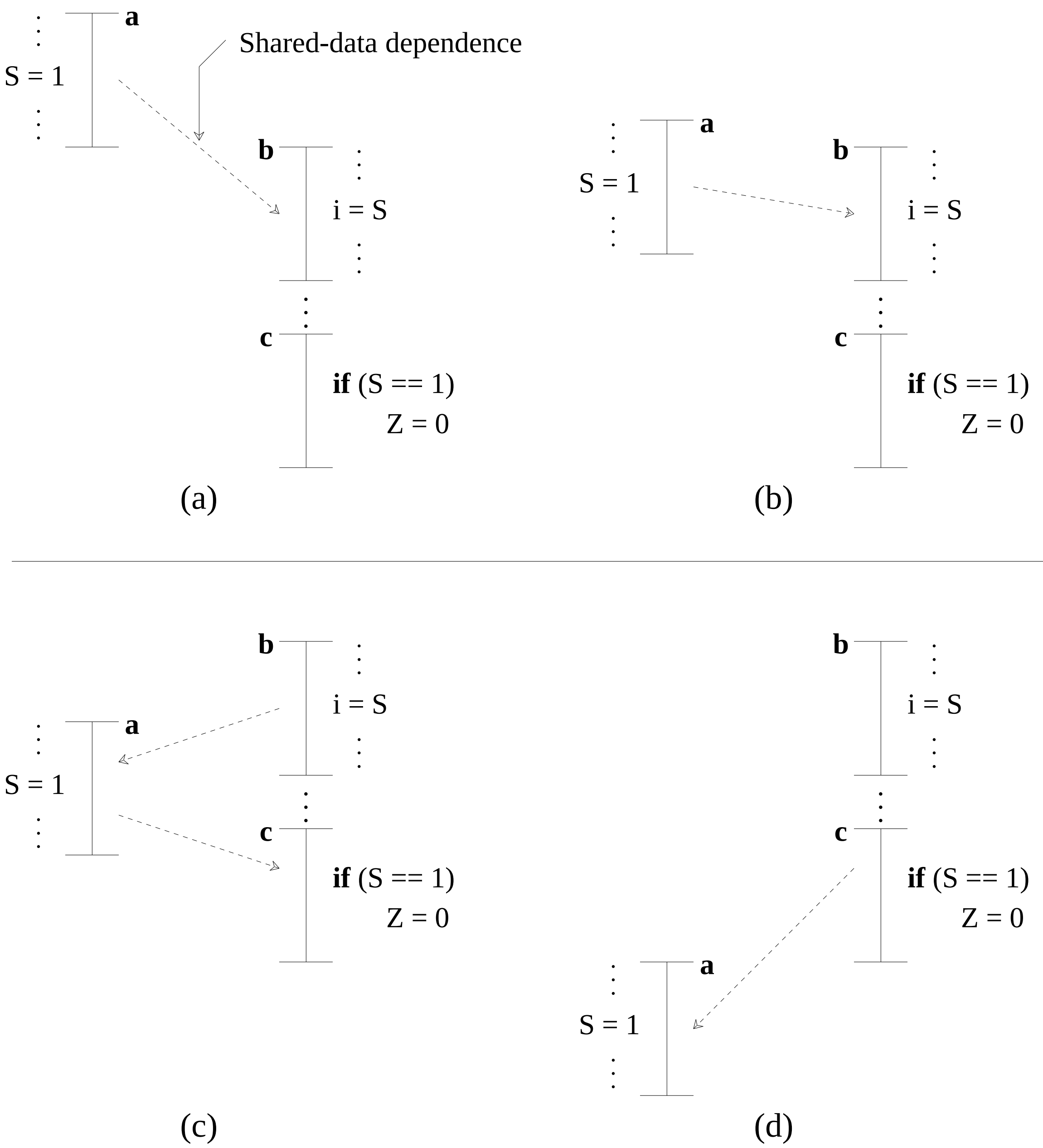


Figure 3.1. (a) an actual program execution, P,
(b) a feasible program execution, $P' \in F_{SAME}$,
(c) a feasible program execution, $P' \in F_{DIFF}$, $P' \notin F_{SAME}$, and
(d) an infeasible program execution, $P' \in F_{SYNC}$, $P' \notin F_{SAME}, F_{DIFF}$
(assuming an initial value of 0 for S)

F_{SYNC} is the set of executions that would have been possible had P not exhibited any shared-data dependences. However, since programs in general do access shared-memory, F_{SYNC} may contain executions the program could never exhibit. When general races occur (whether or not they are considered bugs), F_{SYNC} may contain infeasible executions. Figure 3.1(d) illustrates this situation. The program execution shown, P', is infeasible because the program cannot produce event c. Because $c \xrightarrow{T'} a$, event c will not read a value of 1 for S (assuming that S is initially 0), making it impossible for the program to execute “Z=0”. In this case, even though no explicit synchronization prevented c from preceding a, such an ordering is not possible because event c would then no longer occur. The example in Figure 2.1 illustrates a similar situation. The two events comprising the infeasible data race were not prevented by explicit synchronization from executing concurrently; the program execution in which these events execute concurrently therefore belongs to F_{SYNC}, even though such an execution could never occur.

3.3.3. Sufficient Conditions for Feasibility

The above sets of alternate program executions characterize various behaviors that the execution might have exhibited. However, to actually locate race conditions, we require some means of determining whether a given program execution belongs to one of these sets. Below we present conditions sufficient for a program execution to be feasible, belonging to either F_{SAME} or F_{DIFF}. We first develop conditions for F_{SAME}: any program execution containing the same events as P is feasible if it also contains the same shared-data dependences as P. We then show that even when only a subset of these dependences occur, some events can remain unchanged. Only certain types of shared-data dependences, called *event-control dependences*, can actually affect the outcome of events. We finally develop conditions for F_{DIFF}: any program execution containing a prefix of the events in P is feasible if no event in the prefix is event-control dependent on any event omitted from the prefix. In Chapters 7 and 8 we use both sets of conditions in developing our techniques for data race validation and ordering. The first set is useful as it only requires computing the shared-data dependences; the second is more general but requires knowledge of the event-control dependences.

3.3.3.1. Feasible Program Executions

We first develop conditions sufficient for showing that a program execution belongs to F_{SAME}. We must consider how to guarantee the feasibility of a potential program execution, $P' = \langle E, \xrightarrow{T'}, \xrightarrow{D'} \rangle$, which performs the same events as the actual program execution, $P = \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$. Guaranteeing feasibility requires determining how much the temporal ordering of P can be disturbed without affecting the events performed. Below we prove that any execution exhibiting the same shared-data dependences as P is also capable of performing the same events.

Consider a single-access view, $P_S = \langle E_S, \xrightarrow{T_S}, \xrightarrow{D_S} \rangle$, of the actual program execution. 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. Because the execution outcome of each statement instance depends only upon the values of the variables it reads, the single-access dependences uniquely determine the program state at each step in each process[5]. This result can be proven by showing that the program's input and the single-access dependences uniquely determine these values[48]. Any temporal ordering that could still allow these dependences to occur (and that would not violate the semantics of the synchronization operations) is an ordering that the execution could have exhibited. This result holds even if the program executes nondeterministic statements (such as guarded commands), since the execution is still capable of performing the same events. 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.

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 prove that any higher-level program execution possessing the same events and (higher-level) shared-data dependences as P describes an execution the program could have exhibited. Such a program execution is therefore feasible, and belongs to F_{SAME}. This theorem shows that it is necessary to analyze only higher-level information about a program execution to guarantee feasibility. This result is significant because information might be efficiently recorded about the execution only at a higher level.

Theorem 3.1 (Feasible Execution Theorem)

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

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

(2) $E' = E$, and

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

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[48]. 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 exe-

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

cution 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. We 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 show that 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 $b_S \overset{T_S'}{\nrightarrow} a_S$ whenever $a_S \xrightarrow{D_S} b_S$, and $a_S \overset{T_S'}{\nrightarrow} b_S$ 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 $a \overset{T'}{\nleftrightarrow} b$. 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 $b_S \overset{T_S'}{\nrightarrow} a_S$ 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 $a \overset{T'}{\nleftrightarrow} b$.

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 axiom (A3)).

Case (4): $a \overset{T}{\nleftrightarrow} b$ 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 axiom (A3)), this case is analogous to case (1).

Case (5): $a \overset{T}{\nleftrightarrow} b$ and $a \overset{T'}{\nleftrightarrow} b$. In this case, $\xrightarrow{D_S}$ can contain shared-data dependences in both directions between the a_S and b_S. 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 $a \overset{T'}{\nleftrightarrow} b$, 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 execute concurrently, 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. ■

3.3.3.2. Feasible Program Execution Prefixes

We now develop conditions sufficient for showing that a program execution belongs to F_{DIFF}. We must consider how to guarantee the feasibility of a program execution, $P' = \langle E', \xrightarrow{T'}, \xrightarrow{D'} \rangle$, that performs only a prefix of the events in P and that may exhibit only a subset of its shared-data dependences. Guaranteeing feasibility in this case requires determining how different shared-data dependences might have affected the outcome of P. We characterize those dependences, called *event-control dependences*, that might affect event outcome. We then present a generalization of the feasible execution theorem that uses event-control dependences to determine which of P's shared-data dependences can be omitted from P' while still maintaining feasibility.

The feasible execution theorem shows that any program execution, P', exhibiting the same shared-data dependences as P is capable of performing the same events as P and is therefore feasible. However, even if some of these dependences are not allowed to occur, *some* of the events performed by P may still be performed. Consider a shared-data dependence in P from event 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, in Figure 3.2(a) the dependence exists because b writes a shared variable, S, that c later reads. From axiom (A3) we know that 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 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 those performed by P. However, this different value of S may not immediately alter the events performed but only alter the values computed. For example, in Figure 3.2(a) events c and d do not use S to determine what statements to execute or what shared locations to access; 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.

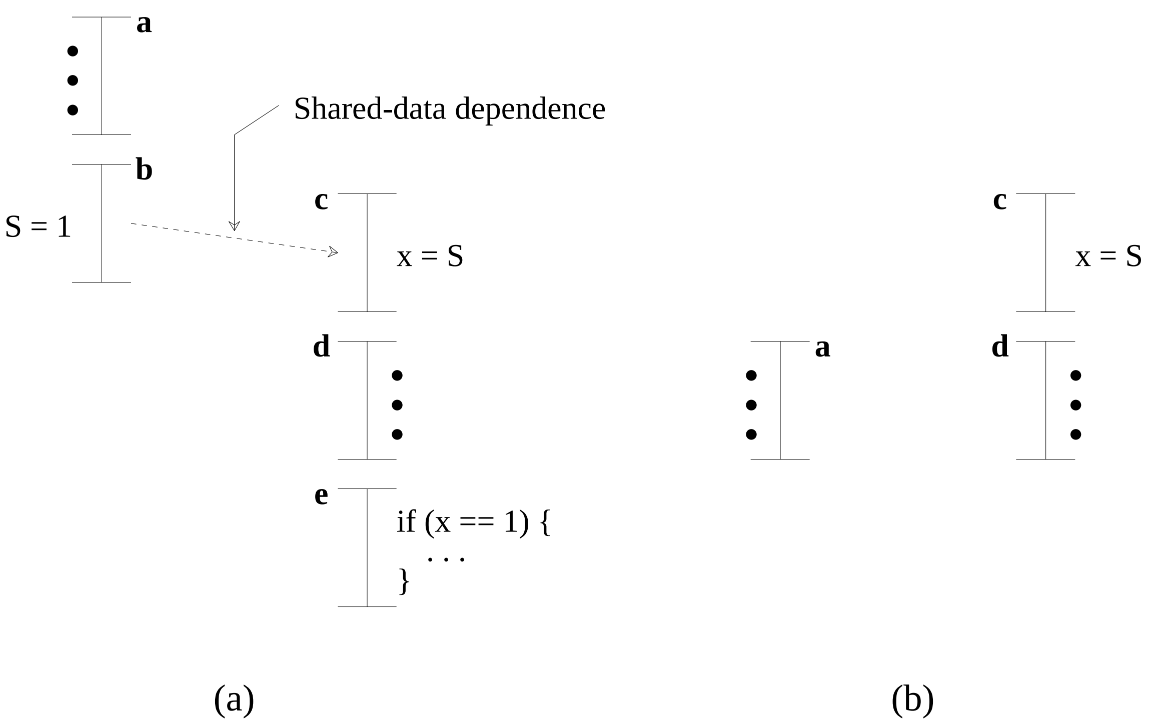


Figure 3.2. (a) an actual program execution, and
(b) a feasible program execution prefix in which *b* is excluded

To formally capture this notion of one event affecting the outcome of another, we define the following relation.

Definition 3.12

The *event-control dependence* relation, $\xrightarrow{\mathrm{E}}$, shows when events can affect each other, and is defined as follows:

$a \xrightarrow{\mathrm{E}} b$ (read as ''a can event-control b'') iff

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

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

(3) b is a **join** 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.

An *approximate event-control dependence* relation, $\xrightarrow{\hat{\mathrm{E}}}$, is a conservative approximation to $\xrightarrow{\mathrm{E}}$ (i.e., $\xrightarrow{\hat{\mathrm{E}}} \supseteq \xrightarrow{\mathrm{E}}$), and is defined as above except that Condition (1) is based on $\xrightarrow{\hat{\mathrm{D}}}$ instead of $\xrightarrow{\mathrm{D}}$. ■

The $\xrightarrow{\mathrm{E}}$ relation shows the possible effects on the execution 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 either control flow or the shared-memory 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 those ordered after a by $\xrightarrow{\hat{\mathrm{T}}}$. The $\xrightarrow{\mathrm{E}}$ relation is therefore a subset of $\xrightarrow{\mathrm{D}} \cup \xrightarrow{\hat{\mathrm{T}}}$. In Figure 3.2(a), for example, $b \xrightarrow{\mathrm{D}} c$, $b \xrightarrow{\mathrm{D}} e$, and $b \xrightarrow{\mathrm{E}} e$, but $b \stackrel{\mathrm{E}}{\not\rightarrow} c$.

We can now extend the feasible execution theorem to accomodate program executions, $P' = \langle E', \xrightarrow{\mathrm{T}'}, \xrightarrow{\mathrm{D}'} \rangle$, that may not contain all of the events or shared-data dependences as P. Consider each shared-data dependence, $a \xrightarrow{\mathrm{D}} b$, exhibited by P. If both a and b are to belong to E', then we require that the dependence also be exhibited by P' (to ensure that b can occur in P'). 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 3.2 (Feasible Prefix Theorem)

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

(P1) P' is a valid program execution (axioms (A1) – (A6) are satisfied),

(P2) P' is a prefix of P, and

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

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

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

(3) $b \notin E'$.

Proof. Viewing the execution at the single-access level, it is easily argued that this theorem holds. Like Theorem 3.1 (the feasible execution theorem), the crux of proving this theorem involves showing that the result extends to a higher-level view. Doing so is analogous to the proof of Theorem 3.1 and we omit the details. ■

Figure 3.2(b) shows an example feasible program execution prefix. The prefix is constructed by excluding *b* from the actual program execution shown in part (a). Since *b* is excluded from the prefix, *e* must also be excluded because $b \xrightarrow{\mathrm{E}} e$, but *c* can remain (even though $b \xrightarrow{\mathrm{D}} c$) because $b \overset{\mathrm{E}}{\not\rightarrow} c$. This example shows how we can guarantee that the temporal ordering in which *a* and *d* execute concurrently could have occurred.

We finally mention that our event-control dependences are similar to the *hides* relation used by Allen and Padua[2, 24, 25] and the *semantic dependences* defined by Podgurski and Clarke[60, 61]. 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).

Chapter 4

CHARACTERIZING RACE CONDITIONS

As discussed in Chapter 1, two different types of race conditions, general races and data races, are of interest for two different classes of parallel programs. In this chapter we present examples of each type of race and characterize them in terms of our model. Given these formal characterizations, we argue that both general races and data races are necessary for debugging, and discuss some fundamental differences between them. Moreover, the difference between the feasible and infeasible races that previous methods can report becomes apparent. We see that infeasible races are a consequence of the set of program executions that previous methods implicitly consider. In later chapters, we use these characterizations to formally define and develop correct techniques for accurate race detection.

4.1. Data Races

Explicit synchronization is often added to shared-memory parallel programs to coordinate accesses to shared data. Different classes of parallel programs typically employ different strategies for this coordination. Below we present an example of one type of coordination (that implements critical sections) to motivate one type of race condition, a *data race*. We then formally characterize data races in terms of our model, and uncover different variations that explain the infeasible races previous methods can report.

4.1.1. A Data Race Example

One purpose of adding explicit synchronization to shared-memory parallel programs is to implement critical sections. Critical sections are any blocks of code intended to execute as if they were atomic[18], regardless of the synchronization constructs used to enforce this atomicity. Atomic execution means that the final state of variables read and written in the section depends only upon their state at the start of the section and the operations performed by the code (and not operations performed by another process). *Bernstein's conditions* state that atomic execution of a critical section is guaranteed if the shared variables it reads and modifies are not modified by any other concurrently executing section of code[9]. A violation of these conditions has typically been called a *data race*[2, 16, 17, 50, 54, 55], *access anomaly*[20, 37, 51], or *harmful shared-memory access*[56]. We prefer the term *data race*.

Figure 4.1 shows an example program for which data races are considered bugs. This program contains two processes that process commands from bank teller terminals. Bank tellers make either deposits or withdrawals from the given bank account (we are assuming a small bank that has only a single customer). Figure 4.1(a) shows a correct version of the program. Since the variables `balance` and `interest` are shared among the two

Process 1

```
/* DEPOSIT */
amount = read_amount();
P(mutex);
balance += amount;
interest += rate*balance;
V(mutex);
```

Process 2

```
/* WITHDRAW */
amount = read_amount();
P(mutex);
if (balance < amount)
    printf("NSF");
else {
    balance -= amount;
    interest += rate*balance;
}
V(mutex);
```

(a): no-data-race version

Process 1

```
/* DEPOSIT */
amount = read_amount();
P(mutex);
balance += amount;
interest += rate*balance;
V(mutex);
```

Process 2

```
/* WITHDRAW */
amount = read_amount();
if (balance < amount)
    printf("NSF");
else {
    balance -= amount;
    interest += rate*balance;
}
```

(b): data-race version

Figure 4.1. (a) C program fragment manipulating a bank account, and (b) erroneous version that can exhibit data races

processes, operations that manipulate them are enclosed in critical sections, implemented using `P` and `V` operations on the semaphore `mutex`. Because critical sections can never execute concurrently, this version exhibits no data races. Figure 4.1(b) shows an erroneous version that can exhibit data races. Due to a programmer oversight, the `P` and `V` operations that should surround the withdraw code are missing. The deposit and withdraw code can therefore execute concurrently, causing their individual statements to effectively interleave, possibly resulting in an incorrect balance and interest (if the atomicity of one of the intended critical sections fails). The data races in this program are considered bugs because the programmer's intent was that the deposit and withdrawal code execute atomically, without interference from other processes.

4.1.2. Characterizing Data Races

We now formally characterize the existence of a data race between two events, a and b. The above example shows that a data race exists when the atomicity of an intended critical section fails. We also consider the case where the atomicity potentially fails, representing a potential data race. We formally characterize these situations as follows: a data race exists between two events, a and b, if they (1) have a data conflict and (2) execute concurrently either in P or in some feasible program execution. We uncover different variations of data races by considering different sets of program executions.

To allow different sets of alternate program executions to be easily considered, we define the notion of a data race between a and b (denoted $\langle a,b \rangle$) over some given set of program executions, F, and then consider different choices for the set F.

Definition 4.1

A *data race* $\langle a,b \rangle$ *over F* exists if

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

(2) there exists a program execution, $P' = \langle E', \xrightarrow{T'}, \xrightarrow{D'} \rangle \in F$, such that $a \overset{T'}{\not\longleftrightarrow} b$. ■

We first consider the case when a data race *actually* occurs, possibly violating the atomicity of a critical section. This type of data race can be characterized in terms of only the actual program execution, P.

Definition 4.2

An *actual data race* $\langle a,b \rangle$ exists if a data race $\langle a,b \rangle$ in P exists[1]. ■

An actual data race exists only between events that *actually* executed concurrently. The absence of actual data races indicates that all intended critical sections in P behaved atomically; the presence of an actual data race shows points in the execution where inconsistent data can be expected. However, even if P exhibits no actual data races, there may be feasible program executions that do. Because of nondeterministic timing variations, some events may have had the potential of executing concurrently even if they did not in P. It is also desirable to characterize these situations.

Definition 4.3

A *feasible data race* $\langle a,b \rangle$ exists if a data race $\langle a,b \rangle$ over F_{DIFF} (or F_{SAME}) exists. ■

[1] A data race in P exists if a data race exists over $\{P\}$, the set containing only P.

A feasible data race indicates that the execution had the potential of exhibiting an actual data race. Unlike actual data races, the existence of a feasible data race only indicates the potential for a failed critical section. Feasible data races can be defined using either set of feasible program executions, F_{DIFF} or F_{SAME}.

In practice, locating feasible data races would require analyzing the program's semantics to determine if the program could have allowed *a* and *b* to execute concurrently. As described in Chapter 2, existing methods take a simpler approach and analyze only the explicit synchronization performed by the execution. We can characterize this simpler notion of a data race by using F_{SYNC}, which is based only on orderings that the program's explicit synchronization might allow, whether or not the program could ever exhibit such orderings.

Definition 4.4

An *apparent data race* $\langle a,b \rangle$ exists if a data race $\langle a,b \rangle$ over F_{SYNC} exists. ■

Previous work in data race detection locates apparent data races. Apparent data races are a less accurate notion of a data race than feasible races because not all program executions in F_{SYNC} are feasible. We call an apparent data race that is not feasible an *infeasible data race*. However, the notion of apparent data races is nonetheless useful since it provides a simple, safe way to detect data races. Detecting apparent races is simple because analyzing the program semantics is not required; we prove in Chapter 6 that it is safe because apparent races are detected only when feasible races exist somewhere in the execution. The drawback of apparent races is their diagnostic precision; some of the apparent data races may be infeasible, masking the location of the program bugs causing the races. In Chapters 7 and 8 we present techniques for approximating which apparent data races are feasible.

We should mention that, because we are concerned with dynamic analysis, we are characterizing data races that occur either in the actual execution or another execution performing the same events (or a prefix of the same events). For example, a feasible data race $\langle a,b \rangle$ indicates the possibility of an actual data race in some execution that is a prefix of P (and that performs *a* and *b*). Because we do not consider all possible executions of the program, but only prefixes of P, the absence of a feasible data race does not imply that all executions of the program are data-race free. A data race between events not performed by P could still have the potential of occurring.

4.2. General Races

We now consider another type of race condition that pertains to a different class of parallel programs. We present an example of a *general race*, which applies to programs intended to be deterministic. Then, as with data races, we formally characterize different variations of a general race (feasible and apparent) in terms of our model.

4.2.1. A General Race Example

Even though programs (such as the one in Figure 4.1) may contain critical sections, they are often intended to be nondeterministic. For example, the order of deposits and withdrawals may occur unpredictably, depending on how fast the tellers type. However, other classes of programs are intended to be completely deterministic, and a different type of race condition pertains to such programs. In these programs, synchronization provides determinism by forcing all accesses to the same shared resource to always execute (on a given input) in a specific order. On a particular input, all executions of such programs always produce the same result, regardless of random timing variations among the processes in the program (e.g., due to unpredictable interrupts, or other programs that may be executing on the same processors). Nondeterminism is generally introduced when the order of two accesses to the same resource is not enforced by the program's synchronization. The existence of two such unordered accesses has typically been called a *race condition* or *race*[23, 24]. For a more consistent terminology, we propose the term *general race*.

As an example of programs for which general races are considered bugs, consider parallel programs that are constructed from sequential programs by parallelizing loops. The sequential version of a program behaves deterministically, producing a particular result for any given input. Typically, the parallelized version is intended to have the same semantics as the sequential version. Preserving these semantics can be accomplished by adding synchronization to the program that ensures all of the data dependences ever exhibited by the sequential version are also exhibited by the parallel version[76]. Such programs exhibit no general races, since preserving these dependences requires that all operations on any specific location are performed in some specific order (independent of external timing variations).

4.2.2. Characterizing General Races

Intuitively, a general race exists in P when a and b have a data conflict and their access order is not ‘‘guaranteed’’ by the execution's synchronization[23, 24]. We formally characterize this situation as follows: a general race exists if a and b are issued in a different order in some feasible program execution than they are in P. As with data races, two variations (feasible and apparent) of a general race exist.

As we did for general races, we define the notion of a general race between a and b (denoted $\langle a,b \rangle$) over some given set of program executions, F, and then consider different choices for the set F.

Definition 4.5

A *general race* $\langle a,b \rangle$ *over F* exists if

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

(2) there exists a program execution, $P' = \langle E', \xrightarrow{T'}, \xrightarrow{D'} \rangle \in F$, such that

(a) $b \xrightarrow{T'} a$ if $a \xrightarrow{T} b$, or

(b) $a \xrightarrow{T'} b$ if $b \xrightarrow{T} a$, or

(c) $a \not\xleftrightarrow{T'} b$. ■

Condition (2) is true if a program execution exists in F in which either a and b occur in an order opposite as in P, or a and b execute concurrently. These cases capture the notion that the execution order among a and b is not ‘‘guaranteed’’.

Characterizing a general race requires that we consider general races over F_{DIFF} (described in Chapter 3). Intuitively, a general race $\langle a,b \rangle$ exists when a and b could have executed in any order. Using the smaller set F_{SAME} is inadequate since, by definition, a general race $\langle a,b \rangle$ exists only if the shared-data dependence between a and b is different in some feasible execution than in P; F_{SAME} only contains executions exhibiting the same shared-data dependences.

Definition 4.6

A *feasible general race* $\langle a,b \rangle$ exists if a general race $\langle a,b \rangle$ over F_{DIFF} exists. ■

As with data races, previous methods for detect general races by analyzing only the execution's explicit synchronization to determine potential event orderings. We can characterize this simpler but less accurate notion of a general race by using F_{SYNC}.

Definition 4.7

An *apparent general race* $\langle a,b \rangle$ exists if a general race $\langle a,b \rangle$ over F_{SYNC} exists. ■

We call an apparent general race that is not feasible an *infeasible general race*. Apparent general races have the same properties as apparent data races; namely, they are simpler but have less diagnostic precision than feasible races.

4.3. Differences Between General Races and Data Races

The main difference between general races and data races is the *granularity* at which interest is focused. This granularity difference has two major implications. First, general races are not always data races, implying that both notions are needed to uncover program bugs. Second, general races are inherently more difficult to detect than data races.

A general race shows when any part of the entire program execution may be nondeterministic. A data race shows when a section of code (intended to be a critical section) may execute non-atomically (or, equivalently, nondeterministically). An implication of this difference is that a general race is not always a data race: two critical sections may be able to execute in either order but never concurrently. A program containing data-conflicting critical sections may exhibit general races, if the critical sections could occur in either order. However, no data races would ever be exhibited, since the critical sections could never execute concurrently (their atomicity would never fail). The no-data-race version of Figure 4.1 illustrates this case — the critical sections execute atomically, but in an unpredictable order determined by when commands are entered by the tellers. This version of the program (correctly) exhibits no data races, and even though general races occur, they are not considered bugs.

This difference implies that both general races and data races are of interest for debugging. The notion of a data race is needed to discover critical sections that were not implemented properly (i.e., those whose atomicity may have failed). The notion of a general race is needed to discover potential nondeterminism anywhere in the program execution. Data races alone will not suffice for these purposes, since a program can be nondeterministic even though it exhibits no data races. General races alone will not suffice, since general races that are not data races are not always bugs. The examples in this chapter illustrate these points.

Moreover, these two different types of race conditions pertain to different classes of parallel programs. General races are typically of interest for programs in which determinism is implemented by forcing all shared-memory accesses (to the same location) to occur in a specific order. Many scientific programs fall into this category (e.g., those constructed by many automatic parallelization techniques). In contrast, data races are typically of interest for asynchronous programs. Programs using shared work-pools fall into this category. They are not intended to be deterministic, but critical sections (that access shared data) are still expected to behave atomically.

The second difference between general races and data races is one of complexity. General races are inherently more difficult to detect than data races. For example, an actual data race captures the possibility that the atomicity of a critical section may have actually failed. Unlike general races, actual data races can be easily located if the complete temporal ordering, $\xrightarrow{T}$, is known; computing alternate temporal orderings is unnecessary[2]. Locating all actual and some feasible data races is sufficient for debugging executions of programs for which data races apply — the absence of actual races implies that the execution (or at least its critical sections) behaved as expected. In contrast, for programs expected to be deterministic, it is necessary to exhaustively locate *all* general races — the execution is deterministic only if exhaustive analysis shows a complete absence of general races. In this sense, locating data races is easier than locating general races.

The granularity difference between general races and data races explains this disparity. A data race can be viewed as a general race at a different granularity, by viewing it as occurring when some computation event (the intended critical section) is possibly nondeterministic, in contrast to a general race, which occurs when any part of the entire execution is possibly nondeterministic. Since computation events contain no explicit synchronization, detecting data races is simpler: the violation of Bernstein's conditions implies that two shared-memory accesses exist whose execution order is not ‘‘guaranteed’’. In contrast, detecting general races requires analyzing the entire execution's synchronization to determine which orderings are not guaranteed. In the next chapter, we prove that, in general, computing the guaranteed orderings is an NP-hard problem.

We finally mention that Emrath and Padua[23] have also attempted to characterize different types of races, but their work has a different scope. They only address programs intended to be deterministic, and consider four levels of nondeterminacy of a program (on a given input). *Internally deterministic* programs are those whose executions on the given input exhibit no general races. *Externally deterministic* programs exhibit general races, but they do not cause the final result of the program to change from run to run. *Associatively nondeterministic* programs exhibit general races only between associative arithmetic operations, and are externally nondeterministic only because

---

[2] In practice, actually recording this complete ordering may incur unacceptable run-time overhead. As discussed in Chapter 2, existing methods only record a subset of $\xrightarrow{T}$, and we later show that this subset is sufficient to detect all actual and some feasible data races.

of roundoff errors (different runs can produce different roundoff errors). Finally, *completely nondeterministic* programs are those exhibiting general races that do not fall into one of the above categories.

Our work complements these characterizations by also considering nondeterministic programs (and data races), and by uncovering the distinction between feasible and apparent races. Moreover, our formal framework provides a convenient mechanism for proving properties of race conditions and developing techniques for race condition detection.

Chapter 5

THE COMPLEXITY OF RACE DETECTION

In the previous two chapters, we presented a formal model for reasoning about race conditions, and characterized general races and data races. In this chapter, we formulate the race-detection problem in terms of our model and prove results regarding its complexity. We define the problem of detecting general races and data races in terms of several *ordering relations*, which summarize the temporal orderings in the sets F_{DIFF}, F_{SAME}, and F_{SYNC}. For programs using synchronization powerful enough to implement two-process mutual exclusion, we prove that deciding each ordering relation is either an NP-hard or co-NP-hard problem. For weaker synchronization, we prove that any ordering relation can be decided in time linear in the number of events in the program execution. We finally discuss the implications of these results; they show for which types of synchronization race detection is either efficient or intractable.

5.1. Formulating the Race Detection Problem

To investigate the complexity of race detection, we must formulate the problem of detecting general races and data races, both feasible and apparent. In the last chapter, these various types of races were defined in terms of temporal orderings exhibited by program executions in the sets F_{DIFF}, F_{SAME}, and F_{SYNC}. We now succinctly formulate the race-detection problem in terms of several *ordering relations* which summarize these temporal orderings. For each set, two types of relations are defined. The *must-have* relations describe orderings that are present in *all* program executions in the set. The *could-have* relations describe orderings that occur in at least one program execution in the set.

To summarize the temporal orderings in a given set of program executions, F, we define the six relations shown below. We will choose either F_{DIFF}, F_{SAME}, or F_{SYNC} for the set F.

Must-Have Relations	
Happened-Before	$a \xrightarrow[\mathrm{F}]{\mathrm{MHB}} b \Leftrightarrow \forall \langle E', \xrightarrow{\mathrm{T}'}, \xrightarrow{\mathrm{D}'} \rangle \in F,\ a \xrightarrow{\mathrm{T}} b$
Concurrent-With	$a \xleftrightarrow[\mathrm{F}]{\mathrm{MCW}} b \Leftrightarrow \forall \langle E', \xrightarrow{\mathrm{T}'}, \xrightarrow{\mathrm{D}'} \rangle \in F,\ a \overset{\mathrm{T}}{\not\longleftrightarrow} b$
Ordered-With	$a \xleftrightarrow[\mathrm{F}]{\mathrm{MOW}} b \Leftrightarrow \forall \langle E', \xrightarrow{\mathrm{T}'}, \xrightarrow{\mathrm{D}'} \rangle \in F,\ \neg(a \overset{\mathrm{T}}{\not\longleftrightarrow} b)$
Could-Have Relations	
Happened-Before	$a \xrightarrow[\mathrm{F}]{\mathrm{CHB}} b \Leftrightarrow \exists \langle E', \xrightarrow{\mathrm{T}'}, \xrightarrow{\mathrm{D}'} \rangle \in F,\ a \xrightarrow{\mathrm{T}} b$
Concurrent-With	$a \xleftrightarrow[\mathrm{F}]{\mathrm{CCW}} b \Leftrightarrow \exists \langle E', \xrightarrow{\mathrm{T}'}, \xrightarrow{\mathrm{D}'} \rangle \in F,\ a \overset{\mathrm{T}}{\not\longleftrightarrow} b$
Ordered-With	$a \xleftrightarrow[\mathrm{F}]{\mathrm{COW}} b \Leftrightarrow \exists \langle E', \xrightarrow{\mathrm{T}'}, \xrightarrow{\mathrm{D}'} \rangle \in F,\ \neg(a \overset{\mathrm{T}}{\not\longleftrightarrow} b)$

Three relations of each type are defined. The *happened-before* relations show events that execute in a specific order, the *concurrent-with* relations[1] show events that execute concurrently, and the *ordered-with* relations show events that execute in either order but not concurrently. We indicate to which set of program executions these relations apply by including the set name (*DIFF*, *SAME*, or *SYNC*) below the arrow (e.g., $\xrightarrow[\mathrm{SYNC}]{\mathrm{MHB}}$).

Although there are 18 ordering relations, only the $\xrightarrow{\mathrm{CHB}}$ and $\xleftrightarrow{\mathrm{CCW}}$ relations are of interest for race detection; we include the others only for completeness. Given these relations, the problem of determining whether a general race or data race exists between two events, *a* and *b*, reduces to deciding the following logical statements.

	General Race	Data Race
Actual	not applicable	$a \overset{\mathrm{T}}{\not\longleftrightarrow} b$
Feasible	$(a \xrightarrow[\mathrm{DIFF}]{\mathrm{CHB}} b \wedge b \xrightarrow[\mathrm{DIFF}]{\mathrm{CHB}} a) \vee a \xleftrightarrow[\mathrm{DIFF}]{\mathrm{CCW}} b$	$a \xleftrightarrow[\mathrm{DIFF}]{\mathrm{CCW}} b$
Apparent	$(a \xrightarrow[\mathrm{SYNC}]{\mathrm{CHB}} b \wedge b \xrightarrow[\mathrm{SYNC}]{\mathrm{CHB}} a) \vee a \xleftrightarrow[\mathrm{SYNC}]{\mathrm{CCW}} b$	$a \xleftrightarrow[\mathrm{SYNC}]{\mathrm{CCW}} b$

5.2. Complexity of Deciding the Ordering Relations

We now prove results on the complexity of computing the ordering relations. The results depend over which set (F_{DIFF}, F_{SAME}, or F_{SYNC}) the ordering relations are defined, and what type of synchronization the execution uses. We first consider the cases for which computing these relations is intractable. For programs using synchronization powerful enough to implement two-process mutual exclusion (such as general semaphores), computing any of the

[1] We use double arrows ($\longleftrightarrow$) to denote the concurrent-with and ordered-with relations to emphasize that they are symmetric.

must-have relations is a co-NP-hard problem and computing any of the could-have relations is an NP-hard problem. For the ordering relations defined over F_{DIFF}, these results hold no matter what type of synchronization the program uses. We then consider a weaker type of synchronization, **Post**/**Wait** style synchronization[2], incapable of implementing mutual exclusion. For this case, we present an efficient algorithm for deciding any of the ordering relations. In the next sub-section, we discuss the implications of these results on the complexity of race detection.

Theorem 5.1

For a program execution that uses any type of synchronization powerful enough to implement two-process mutual exclusion, the problem of deciding whether $a \xrightarrow[\text{SYNC}]{\text{MHB}} b$, $a \xleftrightarrow[\text{SYNC}]{\text{MCW}} b$, or $a \xleftrightarrow[\text{SYNC}]{\text{MOW}} b$ is co-NP-hard, and the problem of deciding whether $a \xrightarrow[\text{SYNC}]{\text{CHB}} b$, $a \xleftrightarrow[\text{SYNC}]{\text{CCW}} b$, or $a \xleftrightarrow[\text{SYNC}]{\text{COW}} b$ is NP-hard.

Proof.

We first present a proof for programs that use general semaphores, and then argue that it easily extends to any type of synchronization that can implement two-process mutual exclusion. Furthermore, we present a proof only for the happened-before relations ($\xrightarrow[\text{SYNC}]{\text{MHB}}$ and $\xrightarrow[\text{SYNC}]{\text{CHB}}$); proofs for the other relations are analogous. We give a reduction[3] from 3CNFSAT[29] such that any Boolean formula is satisfiable (or not satisfiable) iff $b \xrightarrow[\text{SYNC}]{\text{CHB}} a$ (or $a \xrightarrow[\text{SYNC}]{\text{MHB}} b$) for two events, a and b, defined in the reduction. Let an arbitrary instance of 3CNFSAT be given by a set of n variables, $V = \{X_1, X_2, \cdots, X_n\}$, and a Boolean formula B consisting of m clauses, $C_1 \wedge C_2 \wedge \cdots \wedge C_m$, where each clause is a disjunction of three literals (a literal is any variable in V or its negation). From such an instance of 3CNFSAT, we construct a program consisting of $3n+3m+2$ processes that uses $3n+m+1$ semaphores (all semaphores are assumed to be initialized to zero). The execution of this program simulates a nondeterministic evaluation of the Boolean formula B. Semaphores are used to represent the truth values of each variable and clause. As we will show, the execution can exhibit certain orderings iff B is satisfiable (or not satisfiable).

For each variable, X_i, construct the following three processes:

---

[2]This type of synchronization uses only **Post** and **Wait** primitives[13, 14, 24]. All **Wait**s on a synchronization variable block until a **Post** on the same variable is issued. Unlike semaphores, a **Wait** does not change the value of the variable; once a **Post** is issued, all past and subsequent **Wait**s are allowed to proceed. Although some implementations also provide a **Clear** primitive, which resets the synchronization variable, we are considering the use of only **Post** and **Wait**.

[3] This reduction was motivated by the ones Taylor[72, 73] constructed to prove that certain static analysis problems are NP-complete.

P($mutex_i$)	P($mutex_i$)	V($mutex_i$)
V(X_i)	V($\overline{X}_i$)	P($Pass\,2$)
.	.	V($mutex_i$)
.	.	
.	.	
V(X_i)	V($\overline{X}_i$)	

where “ $\cdots$ ” indicates as many V(X_i) (or V($\overline{X}_i$)) operations as occurrences of the literal X_i (or $\overline{X}_i$) in the formula B. The semaphores X_i and $\overline{X}_i$ are used to represent the truth value of variable X_i; a signaling of semaphore X_i (or $\overline{X}_i$) represents the assignment of *True* (or *False*) to variable X_i. The above processes operate in two passes. The first pass is a nondeterministic guessing phase in which each variable used in the Boolean formula is assigned a unique truth value. This assignment is accomplished by allowing either the V(X_i) operations or the V($\overline{X}_i$) operations to proceed, but not both (semaphore $mutex_i$ ensures only one such set of operations is performed). The second pass, which begins after semaphore $Pass\,2$ is signaled, is used only to ensure that all processes terminate; the semaphore operations that were not allowed to execute during the first pass are allowed to proceed.

For each clause, C_j, construct the following three processes:

P(L_1)	P(L_2)	P(L_3)
V(C_j)	V(C_j)	V(C_j)

where L_1, L_2, and L_3 are the semaphores corresponding to the literals in clause C_j. The semaphore C_j represents the truth value of clause C_j. This semaphore will be signaled if the truth assignments guessed during the first pass cause clause C_j to evaluate to *True*.

Finally, create the following two processes:

		a :	**skip**
	P(C_1)		V($Pass\,2$)
	$\cdots$		$\cdots$
	P(C_m)		V($Pass\,2$)
b :	**skip**		

where there are n V($Pass\,2$) operations (one for each variable). Event b is reached only if all clauses evaluate to *True* (since each clause is a disjunct of literals, a signaling of C_j means that clause j evaluates to *True*).

Since the program contains no conditional statements or shared variables, every execution of the program performs the same events (and exhibits no shared-data dependences). Consider all program executions $P' = \langle E', \xrightarrow{T'}, \xrightarrow{D'} \rangle \in F_{SYNC}$ that perform events a and b. We claim that (1) B is not satisfiable iff, for all P', $a \xrightarrow{T'} b$ (or, equivalently, $a \xrightarrow[\text{SYNC}]{\text{MHB}} b$), and (2) B is satisfiable iff, for some P', $b \xrightarrow{T'} a$ (or, equivalently, $b \xrightarrow[\text{SYNC}]{\text{CHB}} a$).

Claim 1: To show the ''if'' part, assume that $a \xrightarrow[\text{SYNC}]{\text{MHB}} b$. That is, there is no execution in which b either precedes a or executes concurrently with a. For a contradiction, assume that B is satisfiable. Then some truth assignment can be guessed during the first pass that satisfies all of the clauses. Event b can then execute before event a, contradicting the assumption. Therefore, B cannot be satisfiable. To show the ''only if'' part, assume that B is not satisfiable. Then there is always some clause, C_j, that is not satisfied by the truth values guessed during the first pass. Therefore, no V(C_j) operation is issued during the first pass. Event b cannot execute until this V operation is issued, which can then only be done during the second pass. The second pass does not occur until after event a executes, so event a must precede event b.

Claim 2: To show the ''if'' part, assume that $b \xrightarrow[\text{SYNC}]{\text{CHB}} a$. That is, some execution exists in which b precedes a. Clearly, b can precede a only if B is satisfiable. To show the ''only if'' part, assume that B is satisfiable. Some execution then exists in which the satisfying truth assignment is guessed, allowing b to precede a.

Since $a \xrightarrow[\text{SYNC}]{\text{MHB}} b$ iff B is not satisfiable, the problem of deciding $\xrightarrow[\text{SYNC}]{\text{MHB}}$ is co-NP-hard. By similar reductions, programs can be constructed such that the non-satisfiability of B can be determined from the $\xleftrightarrow[\text{SYNC}]{\text{MCW}}$ or $\xleftrightarrow[\text{SYNC}]{\text{MOW}}$ relations. The problem of deciding these relations is therefore also co-NP-hard.

Since $b \xrightarrow[\text{SYNC}]{\text{CHB}} a$ iff B is satisfiable, the problem of deciding $\xrightarrow[\text{SYNC}]{\text{CHB}}$ is NP-hard. By similar reductions, programs can be constructed such that the satisfiability of B can be determined from the $\xleftrightarrow[\text{SYNC}]{\text{CCW}}$ or $\xleftrightarrow[\text{SYNC}]{\text{COW}}$ relations. The problem of deciding these relations is therefore also NP-hard.

The above reduction constructs processes containing semaphore operations that implement two-process mutual exclusion (using, for each variable X_i, the semaphore $mutex_i$). In addition, when truth values for the variables are nondeterministically guessed, semaphore operations are used to signal a *True* guess, and are also used to signal when a clause evaluates to *True*. This reduction can be carried out using any type of synchronization capable of implementing these constructs. For example, synchronization that can implement **mutex begin/mutex end** primitives suffices. Therefore, the results also hold for program executions using such synchronization. ■

Theorem 5.2

For a program execution that uses any type of synchronization powerful enough to implement two-process mutual exclusion, the problem of deciding whether $a \xrightarrow[\text{SAME}]{\text{MHB}} b$, $a \xleftrightarrow[\text{SAME}]{\text{MCW}} b$, or $a \xleftrightarrow[\text{SAME}]{\text{MOW}} b$ is co-NP-hard, and the problem of deciding whether $a \xrightarrow[\text{SAME}]{\text{CHB}} b$, $a \xleftrightarrow[\text{SAME}]{\text{CCW}} b$, or $a \xleftrightarrow[\text{SAME}]{\text{COW}} b$ is NP-hard.

Proof.

The proof of Theorem 5.1 suffices. Since the program constructed in the reduction contains no shared variables, all executions exhibit the same shared-data dependences (none), so $F_{SAME} = F_{SYNC}$. Therefore, B (in the proof of Theorem 5.1) is not satisfiable iff $a \xrightarrow[\text{SYNC}]{\text{MHB}} b$, showing that deciding $\xrightarrow[\text{SYNC}]{\text{MHB}}$ is co-NP-hard. The other cases are analogous. ■

Theorem 5.3

For a program execution that uses any type of synchronization, the problem of deciding whether $a \xrightarrow[\text{DIFF}]{\text{MHB}} b$, $a \xleftrightarrow[\text{DIFF}]{\text{MCW}} b$, or $a \xleftrightarrow[\text{DIFF}]{\text{MOW}} b$ is co-NP-hard, and the problem of deciding whether $a \xrightarrow[\text{DIFF}]{\text{CHB}} b$, $a \xleftrightarrow[\text{DIFF}]{\text{CCW}} b$, or $a \xleftrightarrow[\text{DIFF}]{\text{COW}} b$ is NP-hard.

Proof.

As argued in the proof of Theorem 5.1, the reduction from 3CNFSAT requires constructing a program containing synchronization that implements two-process mutual exclusion. For this proof, we perform the reduction by constructing this program without using explicit synchronization at all, but instead implement two-process mutual exclusion using only shared variables (using Peterson's trick[68], for example). For a given execution, P, of this program, the set F_{DIFF} contains all other executions that perform a prefix of the same events, regardless of how their shared-data dependences differ from those of P. As in the proof of Theorem 5.1, we claim that (1) B is not satisfiable iff, for any $P' \in F_{DIFF}$ that also performs a and b, $a \xrightarrow{T'} b$ (or, equivalently, $a \xrightarrow[\text{DIFF}]{\text{MHB}} b$), and (2) B is satisfiable iff, for some $P' \in F_{DIFF}$, $b \xrightarrow{T'} a$ (or, equivalently, $b \xrightarrow[\text{DIFF}]{\text{CHB}} a$). This claim is proven true by arguments analogous to those given in the proof of Theorem 5.1. Therefore, the results hold for executions that use no explicit synchronization at all, implying that they also hold for executions that use any type of explicit synchronization. ■

Theorem 5.4

For a program execution that uses **Post**/**Wait** style synchronization[4], the problem of deciding any of the ordering relations between two events, a and b, can be solved in time in time $O(n)$, where n is the number of events in the program execution.

---

[4] Recall that this style of synchronization uses only **Post** and **Wait** primitives (and not **Clear**).

Given: An actual program execution, $P = \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$, that uses only **Post**/**Wait** style synchronization, and two events $a, b \in E$ that belong to different processes.

Compute: Whether $a \xrightarrow[\text{SYNC}]{\text{MHB}} b$, $b \xrightarrow[\text{SYNC}]{\text{MHB}} a$, or $a \not\xleftrightarrow[\text{SYNC}]{\text{MHB}} b$.

Algorithm:

Decide_MHB_SYNC (a, b):
    First = the set containing the first event in each process;
    Visitable = the set of **Post**s or computation events in *First*;
    for each event variable, x
        $Untriggered_{\text{X}}$ = the set of **Wait**(x) events in *First*;
    while ($Visitable - \{a, b\}$) is nonempty {
        remove any event, $e_{p,i}$, from *Visitable*, except a or b;
        if $e_{p,i}$ is a **Post**(x) **then**
            move all events in $Untriggered_{\text{X}}$ to *Visitable*;
        if $e_{p,i+1}$ exists **then**
            if $e_{p,i+1}$ is an untriggered **Wait**(x) **then**
                add $e_{p,i+1}$ to $Untriggered_{\text{X}}$;
            else
                add $e_{p,i+1}$ to *Visitable*;
    }
    if $a \in$ *Visitable* **and** $b \notin$ *Visitable* **then**
        output $a \xrightarrow[\text{SYNC}]{\text{MHB}} b$;
    else if $b \in$ *Visitable* **and** $a \notin$ *Visitable* **then**
        output $b \xrightarrow[\text{SYNC}]{\text{MHB}} a$;
    else
        output $a \not\xleftrightarrow[\text{SYNC}]{\text{MHB}} b$;

Algorithm 5.1. Decide the $\xrightarrow[\text{SYNC}]{\text{MHB}}$ relation for Post/Wait style synchronization

---

Proof.

We first argue that Algorithm 5.1 correctly decides the $\xrightarrow[\text{SYNC}]{\text{MHB}}$ relation[5], and that the other ordering relations can be similarly computed. We then show that its time complexity is $O(n)$.

Algorithm 5.1 determines how a and b are ordered by $\xrightarrow[\text{SYNC}]{\text{MHB}}$ by simulating the execution to determine if a can be reached before b or *vice-versa*. Execution is simulated by traversing, or *visiting*, events subject to the semantics of **Post**/**Wait** style synchronization. Events belonging to the same process must be visited in the intra-process order, and a **Wait**(x) event cannot be visited until it is *triggered* by a previously visited **Post**(x) event. The set *Visitable* contains events currently eligible for visitation. The set $\mathit{Untriggered}_{\mathrm{X}}$ contains those **Wait**(x) events that are ready to be visited but are not yet triggered. When a **Post**(x) is visited, all untriggered **Wait**(x) events are moved from $\mathit{Untriggered}_{\mathrm{X}}$ into *Visitable*. The algorithm visits as many events as possible while avoiding a and b. When no more events can be visited, the ordering among a and b is determined by checking whether a and/or b are then eligible for visitation. For example, if a is eligible but b is not, then b cannot ever proceed until after a, and the algorithm outputs $a \xrightarrow[\text{SYNC}]{\text{MHB}} b$. To argue correctness, we claim that the algorithm outputs $a \xrightarrow[\text{SYNC}]{\text{MHB}} b$ iff a precedes b in every possible visit of all the events.

If part. If a precedes b in all possible visits, then the algorithm outputs $a \xrightarrow[\text{SYNC}]{\text{MHB}} b$. Clearly, if a must precede b, then the visit performed by the algorithm will eventually add a to *Visitable*; whether or not b is reached, it will not be added to *Visitable* since it cannot be performed until after a. The algorithm will thus report $a \xrightarrow[\text{SYNC}]{\text{MHB}} b$. The other cases are analogous.

Only if part. If the algorithm outputs $a \xrightarrow[\text{SYNC}]{\text{MHB}} b$, then a precedes b in all possible visits of the events. Such a report is made iff a is in *Visitable* and b is not (line 15) at the end of the traversal. To establish a contradiction, assume a visit exists in which b precedes a. In this were the case, b would become visitable at some point before a is visited. Because the traversal ends when no more events can be visited, b would then belong to *Visitable*, contradicting the assumption. The other two cases ($b \xrightarrow[\text{SYNC}]{\text{MHB}} a$ and $a \overset{\text{MHB}}{\underset{\text{SYNC}}{\not\longleftrightarrow}} b$) are analogous.

Even though the algorithm decides only the $\xrightarrow[\text{SYNC}]{\text{MHB}}$ relation, the other ordering relations can be easily be computed from $\xrightarrow[\text{SYNC}]{\text{MHB}}$. For example, the $\xrightarrow[\text{SYNC}]{\text{MHB}}$ and $\xrightarrow[\text{SYNC}]{\text{CHB}}$ relations are complements of one another, so $a \overset{\text{MHB}}{\underset{\text{SYNC}}{\not\longleftrightarrow}} b$ implies $a \xrightarrow[\text{SYNC}]{\text{CHB}} b \wedge b \xrightarrow[\text{SYNC}]{\text{CHB}} a$. Moreover, the concurrent-with and ordered-with relations between two events can be decided by consider how the immediately preceding and following events are ordered by $\xrightarrow[\text{SYNC}]{\text{MHB}}$ and $\xrightarrow[\text{SYNC}]{\text{CHB}}$.

[5] For simplicity, Algorithm 5.1 does not handle program executions that create and destroy processes with **fork**/**join**, but it can easily be extended to do so.

Finally, we consider the algorithm's running time. Since each event is visited at most once, the **while** loop iterates at most n times. If the *Visitable* and *Untriggered* sets are implemented as linked lists, the set operations inside the loop can be performed in constant time. The overall running time is thus $O(n)$. ■

5.3. Complexity of Race Detection

The above results have implications for both general race and data race detection. The NP-hardness results indicate which types of race detection are intractable, and Algorithm 5.1 shows which are efficient. For the intractable cases, the question arises of whether conservative or pessimistic approximations are possible. Below we discuss these issues, for both general races and data races.

Determining whether general races exist between two events, a and b, requires deciding the $\xrightarrow{\text{CHB}}$ (or $\xrightarrow{\text{CCW}}$) relation between these events, as shown earlier in this chapter. Because deciding $\xrightarrow[\text{DIFF}]{\text{CHB}}$ is NP-hard, regardless of the type of synchronization used by the execution, determining whether a feasible general race exists between a and b is always NP-hard[6]. However, the complexity of deciding $\xrightarrow[\text{SYNC}]{\text{CHB}}$, and therefore determining the existence of an apparent general race, depends on the type of synchronization used. For executions using synchronization powerful enough to implement two-process mutual exclusion, the problem is NP-hard. For weaker types of synchronization, the problem can be efficiently solved (by Algorithm 5.1). Efficient general race detection is therefore possible only for executions that use weaker synchronization.

Similarly, exactly determining the existence of feasible data races is NP-hard, regardless of the type of synchronization used. However, in contrast to general races, determining the existence of apparent data races is still NP-hard for any execution for which data races are of interest. Data races are of interest only for programs that contain critical sections; they are not applicable to weaker types of synchronization. Since critical sections implement mutual exclusion, exact and exhaustive apparent data race detection is always NP-hard.

Even though exactly determining the existence of general races and data races is NP-hard for executions using powerful synchronization, approximations can be efficiently computed. Emrath, Ghosh, and Padua[24, 25] present an algorithm for computing a superset of the $\xrightarrow[\text{SYNC}]{\text{CHB}}$ relation for executions that use an extended form of **Post/Wait** style synchronization that includes **Clear** operations[7]. Helmbold, McDowell, and Wang[35] present algorithms for computing a subset of $\xrightarrow[\text{SYNC}]{\text{MHB}}$ and a superset of $\xrightarrow[\text{SYNC}]{\text{CCW}}$ for programs that use semaphores. Both of these approximations are conservative, allowing detection of a superset of the apparent general races (by using the approximation

---

[6]A rigorous proof would reduce the problem of deciding $\xrightarrow[\text{DIFF}]{\text{CHB}}$ to the problem of deciding whether $\langle a,b \rangle$ is a feasible general race. The details of such a proof are trivial, and are omitted here.

[7]**Clear** operations reset the value of x, causing subsequent **Wait**(x)'s to block until another **Post**(x) is issued. Even though **Post** and **Wait** alone are insufficient to implement two-process mutual exclusion, the addition of **Clear** operations allows mutual exclusion to be implemented[53], making exact race detection NP-hard.

to $\xrightarrow[\text{SYNC}]{\text{CHB}}$) or the apparent data races (by using the approximation to $\xrightarrow[\text{SYNC}]{\text{CCW}}$). Conservative approximations are useful for debugging since the absence of race reports indicates that the execution was race-free. However, the diagnostic precision of these approximations can be poor because races can be reported where none exist.

Pessimistic approximations are also possible. Pessimistic approximations are insufficient for guaranteeing that the execution was race-free, but exactly pinpoint races when they are reported. A fundamental difference between general races and data races lies in the usefulness of pessimistic approximations. The utility of data-race detection for debugging lies in locating actual races, as well as a subset of the feasible and apparent races. Even if not all feasible or apparent data races can be located in practice, locating all actual races and some feasible races is nonetheless valuable. The absence of actual data races implies that the execution's critical sections behaved as expected (atomically). The location of actual races pinpoints portions of the execution where inconsistent data should be expected. In contrast, there is no corresponding notion of an actual general race. Pessimistic general race detection can pinpoint some races, but cannot guarantee that the execution was race-free. This contrast further supports our earlier claim that general races are inherently more difficult to detect than data races.

To summarize the results of this chapter, Figure 5.1 shows the complexity of determining whether different types of races exist between two given events. The first column pertains to executions that use synchronization powerful enough to implement two-process mutual exclusion; the second column pertains to weaker synchronization.

	Type of Synchronization	
	Two-process Mutex	Weaker
Data Races		
Actual	$O(1)$	not applicable
Feasible	NP-hard	not applicable
Apparent	NP-hard	not applicable
General Races		
Actual	not applicable	not applicable
Feasible	NP-hard	NP-hard
Apparent	NP-hard	$O(n)$

Figure 5.1. Complexity of exactly deciding whether two given events form a race

Chapter 6

DETECTING APPARENT DATA RACES

In the previous chapters, we formally defined the problem of race detection and proved results about its complexity. We now focus on techniques for the accurate detection of data races. In this chapter, we discuss apparent data races, which are the types of data races that previously proposed methods detect. We define apparent data race detection in terms of the *temporal ordering graph*, which is a variation of the ordering graph constructed by these methods. We first conceptually define the temporal ordering graph, and then discuss apparent data race detection. We prove that the simple approach of analyzing the graph only to detect apparent data races is safe for debugging in the sense that no actual data races are ever missed. We also present algorithms for apparent data race detection. Discussion of experience with an implementation of these algorithms appears in Chapter 9. Although in this and subsequent chapters we present algorithms for post-mortem detection, applying our results does not require that the temporal ordering graph explicitly be constructed. Our results can be used to reason about any race detection method that can be described in terms of the graph (such as on-the-fly approaches).

6.1. Temporal Ordering Graph

As discussed in section 3.2, an approximate program execution, $\hat{P} = \langle E, \xrightarrow{\hat{T}}, \xrightarrow{\hat{D}} \rangle$, represents information that can be reasonably recorded about the execution. Previous methods represent $\xrightarrow{\hat{T}}$ with an ordering graph (whether or not the graph is explicitly constructed). We will take a similar approach, and use a variation of this graph to reason about data race detection.

Definition 6.1

The *temporal ordering graph*, *G*, contains two nodes per event: for every event *e*, the nodes $\mathbf{e_s}$ and $\mathbf{e_f}$ represent the start and finish of *e*. *G* contains edges representing the same information as any ordering graph described in Chapter 2: an edge from *a* to *b* in an ordering graph is equivalent to an edge from $\mathbf{a_f}$ to $\mathbf{b_s}$ in *G*.

Given a graph *G*, an event *a* is a *predecessor* of another event *b* 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* if neither is a predecessor or successor of the other. ■

We define G to contain two nodes per event as it simplifies the proofs in Chapters 7 and 8[1]. Figure 6.1 shows an example temporal ordering graph for the program execution shown in Figure 2.1. The events are labeled with the letters ''(a)''–''(f)'', and ''S'' and ''F'' indicate start and finish nodes. We will use this program execution as a running example in the next two chapters.

The temporal ordering graph can be represented by the approximate temporal ordering, $\xrightarrow{\hat{T}}$: $a \xrightarrow{\hat{T}} b$ iff a is a predecessor of b, $b \xrightarrow{\hat{T}} a$ iff b is a predecessor of a, and $a \overset{\hat{T}}{\not\longleftrightarrow} b$ otherwise. The only assumption we make

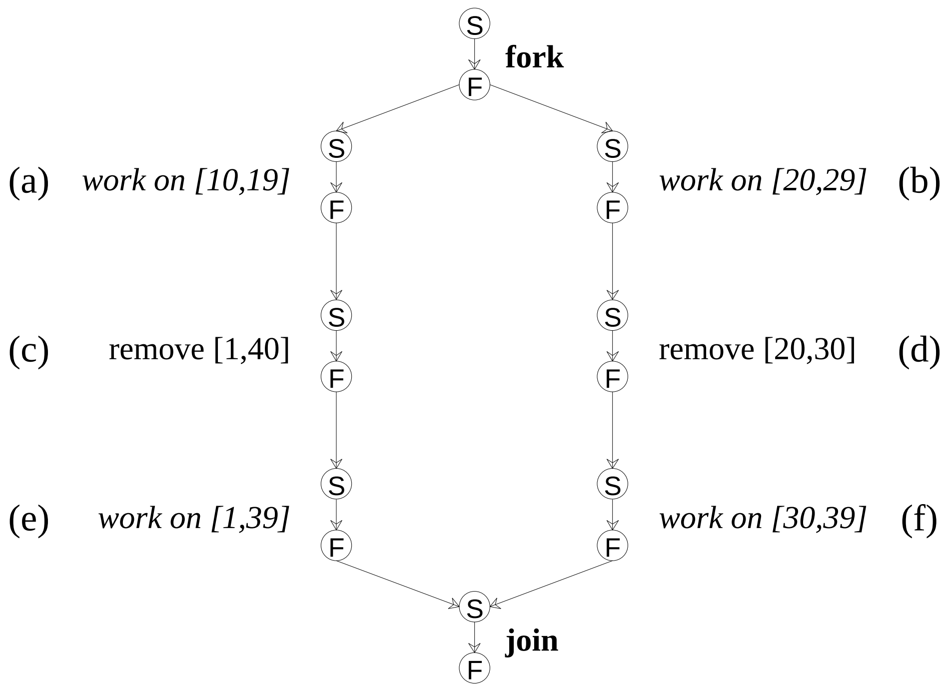


Figure 6.1. Temporal ordering graph for example shown in Figure 2.1

---

[1]Storing two nodes per event is unnecessary in an actual implementation; only one node per synchronization event suffices. To accommodate such a representation, only small modifications are required to the algorithms presented in this thesis.

about G is that any linear order of its nodes must be a global-time model that defines a temporal ordering obeying the synchronization axioms (axioms (A4)–(A6)). Any of the ordering graphs constructed by previous methods (extended to contain two nodes per event) suffice.

6.2. Apparent Data Races

Apparent data races are those detected by analyzing only the execution's explicit synchronization. In Chapter 4 we defined an apparent data race $\langle a,b \rangle$ to exist when a and b have a data conflict and $a \overset{\hat{T}}{\not\longleftrightarrow} b$. The apparent data races can be located by searching the temporal ordering graph for pairs of data-conflicting events that are unordered by the graph (implying that $a \overset{\hat{T}}{\not\longleftrightarrow} b$). We next prove that this approach is safe for debugging since it never leaves actual data races undetected, and then present algorithms for apparent data race detection.

6.2.1. Safeness of Detecting Only Apparent Data Races

In general, apparent data races include all actual data races, plus additional races, both feasible and infeasible (Chapter 2 presented an example of an infeasible apparent data race). The simple approach of detecting apparent data races cannot distinguish among these types of races. However, below we prove that each actual data race is also an apparent data race. The naive method of simply reporting all apparent data races to the programmer (which is the approach of previous methods) is therefore safe in the sense that no actual data races are left undetected.

Theorem 6.1 (Actual Data Race Theorem)

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 $a \overset{\hat{T}}{\not\longleftrightarrow} b$. The assumption $a \overset{T}{\not\longleftrightarrow} b$ is equivalent to $a \overset{T}{\not\rightarrow} b \wedge b \overset{T}{\not\rightarrow} a$. By definition, $a \overset{\hat{T}}{\longrightarrow} b \Rightarrow a \overset{T}{\longrightarrow} b$ (see Section 3.2), or $a \overset{T}{\not\rightarrow} b \Rightarrow a \overset{\hat{T}}{\not\rightarrow} b$, giving us $a \overset{\hat{T}}{\not\rightarrow} b \wedge b \overset{\hat{T}}{\not\rightarrow} a$ which is equivalent to $a \overset{\hat{T}}{\not\longleftrightarrow} b$. ■

The above proof 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. Detecting all actual data races is important for debugging. When no actual races occur, the execution's critical sections are guaranteed to have behaved atomically; when actual races are located, they pinpoint places in the execution where inconsistent data can be expected. As we will show in the next chapter, apparent data races also have the property that their presence implies the existence of a feasible data race somewhere in the program execution. This property also implies that when no actual data races occur, no apparent data races will be reported.

6.2.2. Algorithms for Detecting Apparent Data Races

We now present algorithms for post-mortem detection of apparent data races. These algorithms first preprocess the temporal ordering graph to compute the event orderings given by the graph (Algorithm 6.1). After preprocessing, the ordering between any two events can be determined in constant time (Algorithm 6.2). Finally, apparent data races are detected by checking all pairs of unordered events for data conflicts (Algorithm 6.3). Although the general techniques discussed here are not original, we present them as they form the basis of the algorithms in Chapters 7 and 8 for data race validation and ordering. Throughout this chapter, we assume that the entire temporal ordering graph, and the *READ* and *WRITE* sets for each computation event, are available.

Given: The temporal ordering graph, *G*.

Compute: The timestamp for each node.

Algorithm:

```
Compute_T̂_Timestamps (G):
        compute a topological sort of G;
        for each node, x, in G {
                x.timestamp = ⟨−1,...,−1⟩;
                x.timestamp[x.proc_num] = x.ser_num;
        }
        for each node, x, in topological order {
                for each out-edge from x to y {
                        y.timestamp = TimestampMax(y.timestamp, x.timestamp);
                }
        }

TimestampMax (t1, t2):
        for each process, i {
                tmp[i] = max(t1[i], t2[i]);
        }
        return tmp;
```

Algorithm 6.1. Compute timestamps for *G*

Detecting apparent data races involves locating pairs of events, *a* and *b*, that (1) have a data conflict and (2) are unordered by the temporal ordering graph. Locating unordered events requires analyzing *G* to decide whether a path connects two given nodes. Since this determination must be made many times, it should be as efficient as possible. We therefore preprocess the graph to compute a *timestamp* for each node[16, 27, 28, 33, 35, 45, 50]. Timestamps provide a mechanism for quickly determining the ordering between two given events. Each timestamp is a vector (of length *p*) of event serial numbers[2]. To manage timestamps, we associate the following information with each node, *n*, in *G*.

n.timestamp — Timestamp for representing event ordering.
n.proc_num — Process number to which *n* belongs.
n.ser_num — Serial number of node *n* within its process.

The i^{th} slot of a timestamp, *n.timestamp*[*i*], equals the serial number of the last node in process *i* from which a path to *n* exists (or equals −1 if no such path exists). By definition, if *i* equals *n.proc_num*, then *n.timestamp*[*i*] equals the serial number of *n*.

Given: Two nodes, *m* and *n*, for which timestamps have been computed.

Compute: Whether a path exists from *m* to *n*, from *n* to *m*, or whether no such paths exist.

Algorithm:

```
OrderOf (m, n):
    if (n.timestamp[m.proc_num] ≥ m.ser_num)
        return PRED;       /* path from m to n */
    else if (m.timestamp[n.proc_num] ≥ n.ser_num)
        return SUCC;       /* path from n to m */
    else
        return UNORDERED;         /* no path connecting m and n */
```

Algorithm 6.2. Determine the connectivity between two nodes

---

[2]This space bound can be optimized for executions that create and destroy processes on-the-fly, by re-using process numbers[19, 33]. For simplicity, we assume that a fixed number of processes exist during execution.

Algorithm 6.1 computes timestamps. The timestamp of a given node, n, is the maximum over the timestamps of all nodes that have edges into n. After initializing the timestamps, the algorithm traverses the graph in a topological order. For each node, its outgoing edges are followed, and the component-wise maximum is taken of its timestamp with the timestamp at the head of the edge. The running time of this algorithm is $O(ep)$, where e and p are the number of edges and processes in the graph; TimestampMax, requiring $O(p)$ time, is called once for each edge. The topological sort can be performed in time $O(n+e)$, where n is the number of nodes[66], and is dominated by the rest of the computation. In the worst case, the number of edges in the graph is of order n; all existing race detection methods add edges only between pairs of synchronization operations on the same variable, or between **fork**/**join** nodes and their children. The worst-case running time is thus $O(np)$. The space required to store the timestamps is also $O(np)$.

Once the timestamps are computed, determining whether a path exists between any two nodes in G involves comparing their timestamps and requires only constant time, as shown in Algorithm 6.2.

Finally, Algorithm 6.3 detects apparent data races by checking all pairs of computation events that are unordered by G for data conflicts. For each computation event, $e_{p,i}$, this algorithm checks all events, $e_{q,j}$, that are unordered with $e_{p,i}$. To avoid checking any pair of events twice, only events belonging to a process $q > p$ are considered. The events in q unordered with $e_{p,i}$ are traversed by starting after the last event in q that is a predecessor of $e_{p,i}$. This event can be retrieved in constant time if each process in G is stored as an array of nodes. The event's serial number, determined from the timestamps for $e_{p,i}$, gives the array index. Process q is then scanned forward from this event until reaching the first event that is a successor of $e_{p,i}$. An array representation is well suited to post-mortem analysis since the graph is static; the arrays can be dynamically allocated before processing the traces and remain unchanged during analysis.

The outer **for** loop in Algorithm 7.3 iterates $O(n)$ times, and the inner loop $O(p)$ times. The inner loop first performs $O(1)$ work locating $e_{q,j}$ (lines 4–5) and then scans through all events unordered with $e_{p,i}$ (lines 6–10). The total work performed by the algorithm is thus $O(np(1+\bar{u}))$, where $\bar{u}$ is the average number of events per process unordered with a given computation event. If we define $U = np\bar{u}$, the total number of unordered event pairs, the total work becomes $O(np+U)$. In the worst case, U is of order n^2, and $O(n^2)$ apparent data races can exist.

Given: The temporal ordering graph, G.

Compute: Apparent data races.

Algorithm:

DetectApparentRaces (G):
    for each computation event, $e_{p,i}$, in G {
        for each process $q > p$ {
            let $e_{q,j}$ be the last event in q that is a predecessor in G of $e_{p,i}$;
            $j = j + 1$;
            while ($e_{p,i}$ and $e_{q,j}$ are unordered by G) {
                if ($e_{q,j}$ is a computation event **and** $e_{p,i}$, $e_{q,j}$ have a data conflict)
                    output apparent data race $\langle e_{p,i}, e_{q,j} \rangle$;
                $j = j + 1$;
            }
        }
    }

Algorithm 6.3. Detect apparent data races

Chapter 7

VALIDATING APPARENT DATA RACES

We now consider the two causes of data race artifacts, infeasible races and races caused by other races. In this chapter we address the first cause, infeasible races. We present a technique called *validation* for determining which apparent data races are feasible. We prove results showing how validation can be performed, and present simple algorithms for post-mortem validation. Validation involves augmenting the temporal ordering graph with additional edges representing the shared-data and event-control dependences. Each apparent data race can be characterized as either feasible or possibly infeasible depending on whether certain paths or cycles result. In Chapter 9 we discuss our experience with an implementation of race validation.

The results in this chapter not only form a basis for validation algorithms, they also provide a means for reasoning about any data race detection method that can be described in terms of the temporal ordering graph. Any such method can be analyzed to determine when infeasible races might be reported.

7.1. Validation Results

Validation is based on augmenting the temporal ordering graph, G, with additional edges. We present two sets of results: validation using shared-data dependences and validation using event-control dependences. First, G is augmented with edges representing the shared-data dependences, to obtain the graph G_D. Each apparent data race $\langle a,b \rangle$ is characterized as either *nontangled* or *tangled*, depending on whether a and b are unordered or ordered by G_D. Nontangled races are guaranteed to be feasible, and each tangled race belongs to a set containing at least one feasible race. Second, some tangled data races can be proven feasible by pruning edges from G_D. The graph G_E is constructed by removing those shared-data dependence edges that are not also event-control dependences. A tangled data race $\langle a,b \rangle$ is feasible if a and b are then unordered by G_E.

Our results are based on proving the feasibility of an apparent data race $\langle a,b \rangle$ by showing that a feasible program execution (or feasible program execution prefix), $P' = \langle E', \xrightarrow{T'}, \xrightarrow{D'} \rangle$, exists such that $a \overset{T'}{\not\longleftrightarrow} b$. The existence of such a feasible execution means that the program *could have* executed in such a way that a and b formed a data race. Exactly determining the existence of these program executions requires precise knowledge of the shared-data or event-control dependences exhibited by the execution. Without exhaustive execution tracing, however, the exact dependences are unknown. We therefore take a conservative approach and augment G using the estimates $\xrightarrow{\hat{D}}$ and $\xrightarrow{\hat{E}}$.

7.1.1. Validation using Shared-Data Dependences

We construct the graph G_D by augmenting G with edges that ensure a path exists 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 need be added only between events unordered by G. Intuitively, these edges reflect possible orderings caused by shared-data dependences (as illustrated in the example of Figure 2.1), similar to the way in which edges in G reflect orderings caused by 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 $\langle a,b \rangle$ must be feasible.

Figure 7.1 shows G_D for our example (from Figure 6.1). Edges are added between nodes representing events that have data conflicts and that are unordered by G. Direct shared-data dependences exist between these events (which are exactly those involved in apparent data races). In general, edges must also be added to represent transitive dependences. Algorithms for doing so are presented later in this chapter.

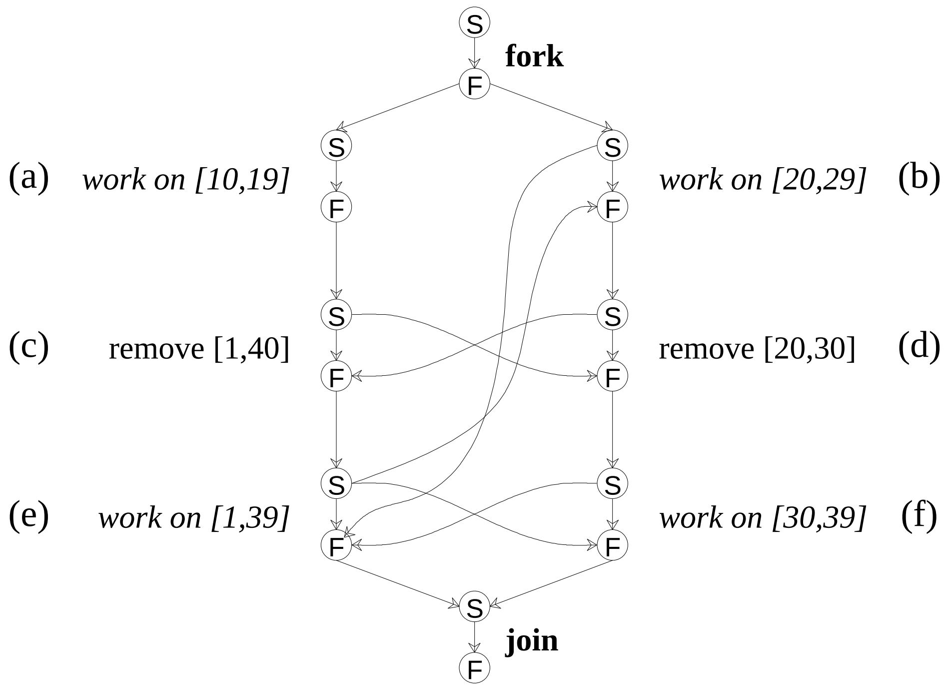


Figure 7.1. Example G_D (G augmented with shared-data dependences)

Determining if a and b are unordered by G_D is complicated when $\xrightarrow{\hat{D}}$ is so conservative that G_D contains cycles (if the exact dependences were known, cycles would never be introduced). We therefore classify the apparent data races into those that belong to strongly connected components[1] and those that do not.

Definition 7.1

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, and is *nontangled* otherwise.

Each strongly connected component of G_D defines a set of tangled data races, called a *tangle*. ■

We next prove (in Theorem 7.1) that any nontangled apparent data race is feasible. We then show (in Theorem 7.2) that each tangle contains at least one feasible data race.

Theorem 7.1 (Feasible Race Theorem)

An apparent data race $\langle a,b \rangle$ is feasible if it is nontangled (or, equivalently, if a and b are unordered by G_D).

Proof.

We introduce the graph $G_{D-ACTUAL}$ as a device for showing certain feasible program executions must exist. $G_{D-ACTUAL}$ is identical to G_D, except that edges representing shared-data dependences that did not actually occur do not appear (they were conservatively added to G_D so that no actual dependences were missed). Even though we do not have enough information to construct $G_{D-ACTUAL}$, it nonetheless exists. We first prove that a and b are unordered by $G_{D-ACTUAL}$, and then show that this implies $\langle a,b \rangle$ is feasible.

First, we prove that a and b are unordered by G_D (i.e., that it contains no path from $\mathbf{a_f}$ to $\mathbf{b_s}$ or from $\mathbf{b_f}$ to $\mathbf{a_s}$). Since $G_{D-ACTUAL}$ contains no more edges than G_D, a and b must also be unordered by $G_{D-ACTUAL}$. For a contradiction, assume there is a path from $\mathbf{a_f}$ to $\mathbf{b_s}$, or a path from $\mathbf{b_f}$ to $\mathbf{a_s}$, in G_D. 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. Because $a \xrightarrow{\hat{D}} b$ and $b \xrightarrow{\hat{D}} a$ ($\langle a,b \rangle$ is an apparent data race), G_D 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_D, 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, a and b are unordered by both G_D and $G_{D-ACTUAL}$.

Next, we prove that a and b unordered implies the existence of a feasible program execution, $P' = \langle E, \xrightarrow{T'}, \xrightarrow{D} \rangle$, such that $a \overset{T'}{\not\longleftrightarrow} b$. The existence of P' proves that $\langle a,b \rangle$ is a feasible data race. Because a and b are unordered by $G_{D-ACTUAL}$, there exists a linear order L of its nodes that is a global-time model defining a temporal

---

[1] A strongly connected component is a maximal cycle; 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[66].

ordering relation $\xrightarrow{T'}$ such that $a \overset{T'}{\not\longleftrightarrow} b$. By the definition of G, $\xrightarrow{T'}$ obeys axioms (A4)–(A6) (since G contains no more edges than $G_{D-ACTUAL}$). Furthermore, $\xrightarrow{T'}$ obeys axiom (A3): because $G_{D-ACTUAL}$ contains a path 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 $b \overset{T'}{\not\longrightarrow} a$. Finally, $\xrightarrow{T'}$ trivially obeys axioms (A1) and (A2) since it is defined by a linear order of an acyclic graph. Since $\xrightarrow{T'}$ obeys axioms (A1)–(A6), by the feasibility theorem, $P' = \langle E, \xrightarrow{T'}, \xrightarrow{D} \rangle$ is a feasible program execution. ■

Theorem 7.2 (Tangle Theorem)

Each tangle in G_D contains at least one feasible data race.

Proof.

We first prove that the tangle must contain an apparent data race between two events unordered by $G_{D-ACTUAL}$, and then argue that this implies the race is feasible.

Let T be the set of events in the tangle, and $\mathbf{T}$ be the set of nodes in $G_{D-ACTUAL}$ representing these events. To establish a contradiction, assume that for all apparent data races $\langle a,b \rangle$ in the tangle, a and b are ordered by $G_{D-ACTUAL}$. 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 ($G_{D-ACTUAL}$ is acyclic), assume that the path from $\mathbf{a_f}$ to $\mathbf{b_s}$ exists. No such path exists in G (a and b are unordered by G). The path in $G_{D-ACTUAL}$ 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 a path exists 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 $\mathbf{b_s}$. This 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 $\mathbf{d_f}$ represents either a direct or transitive dependence. Either there is a data conflict between c and d (and therefore also an apparent data race), or c data conflicts with some other event that data conflicts with d. 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 from $\mathbf{c_s}$ to $\mathbf{d_f}$ represents a transitive dependence, 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, an apparent data race $\langle a,b \rangle$ must exist such that a and b are unordered by $G_{D-ACTUAL}$.

By the argument in the proof of Theorem 7.1, 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 apparent data races in the tangle is feasible. ■

Figure 7.2 shows which apparent data races can be validated by G_D for our example. The apparent data race $\langle e,f \rangle$ is nontangled; $\mathbf{e_f}$ and $\mathbf{f_s}$ are not in the same strongly connected component, nor are $\mathbf{f_f}$ and $\mathbf{e_s}$. By Theorem 7.1, $\langle e,f \rangle$ is feasible. In contrast, the races $\langle c,d \rangle$ and $\langle e,b \rangle$ belong to the same tangle. By Theorem 7.2, at least one of them must be feasible (recall that $\langle c,d \rangle$ is feasible and $\langle e,b \rangle$ is not), but G_D contains insufficient information to determine which one.

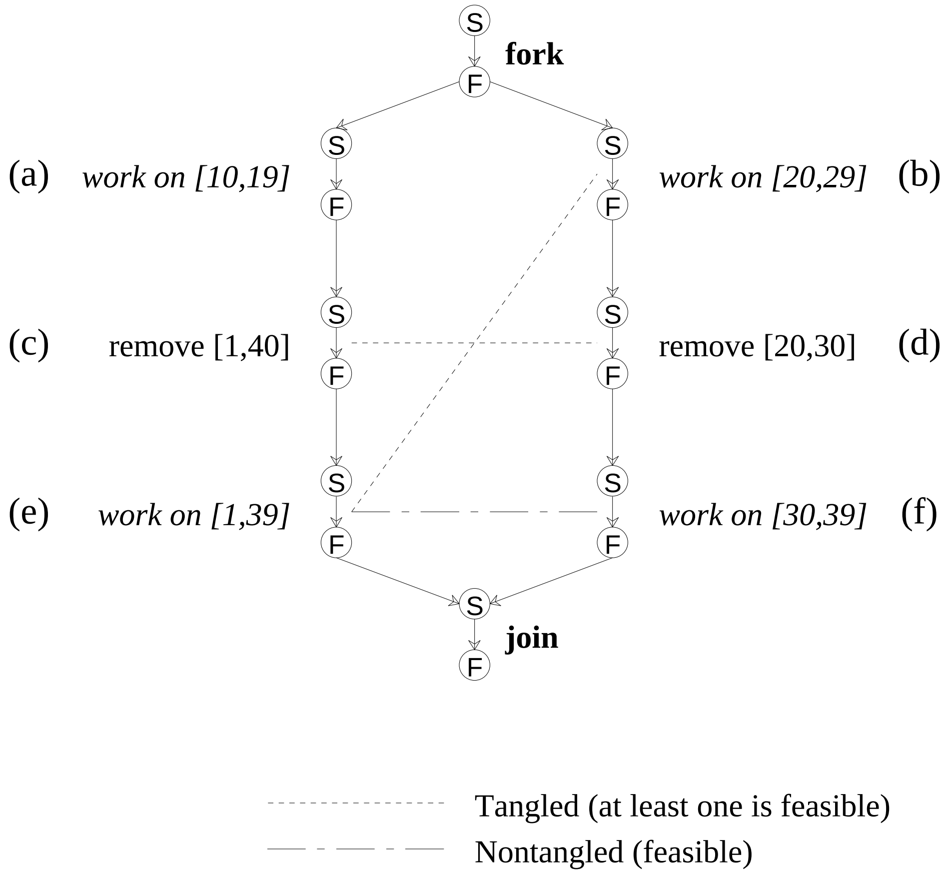


Figure 7.2. Results of validation using G_D

7.1.2. Validation using Event-Control Dependences

We extend the above results by observing that not all shared-data dependences necessarily cause events to 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. G_E therefore contains edges that represent only event-control dependences. By analyzing G_E, tangled data races (which cannot be validated using G_D) can sometimes be proven feasible.

In some cases, shared-data dependences that cause an apparent data race $\langle a,b \rangle$ to be tangled (because a and b are ordered by G_D) may not have affected the outcome of a and b. Recall that an event represents both the execution instance of a group of statements and the shared-memory locations they reference. The outcome of a and b can thus be affected only if these dependences affect either of these two components; in Section 3.3 we defined the event-control dependences to represent such dependences. The tangled race can be proven 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, then neither a nor b (nor any of their successors) can event-control any of their predecessors, so we are guaranteed that all predecessors of a or b would have remain unchanged. 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. Theorem 7.3 below proves this result.

Figure 7.3 shows G_E for our example. In this example, we conservatively assume that the ‘‘remove’’ events can event-control each other, because we cannot determine otherwise. However, we assume that the *work* events cannot event-control each other (see Definition 3.11), e.g., because they only wrote to the array (which could be discovered by a simple examination of the *READ* and *WRITE* sets), or because the computed array values were not used to determine control flow (which might be discovered from a static analysis of the program). G_E therefore contains no edges between the *work* events, but the edges between the ‘‘remove’’ events remain.

Theorem 7.3 (Feasible Tangled Race Theorem)

> 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 and *vice-versa* (i.e., if G_E contains no path from $\mathbf{a_f}$ to $\mathbf{b_f}$, or from $\mathbf{b_f}$ to $\mathbf{a_f}$).

Proof.

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 $a \overset{T'}{\not\longleftrightarrow} b$.

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, by constructing a feasible program execution prefix, $PP' = \langle E', \xrightarrow{T'}, \xrightarrow{D'} \rangle$, as follows.

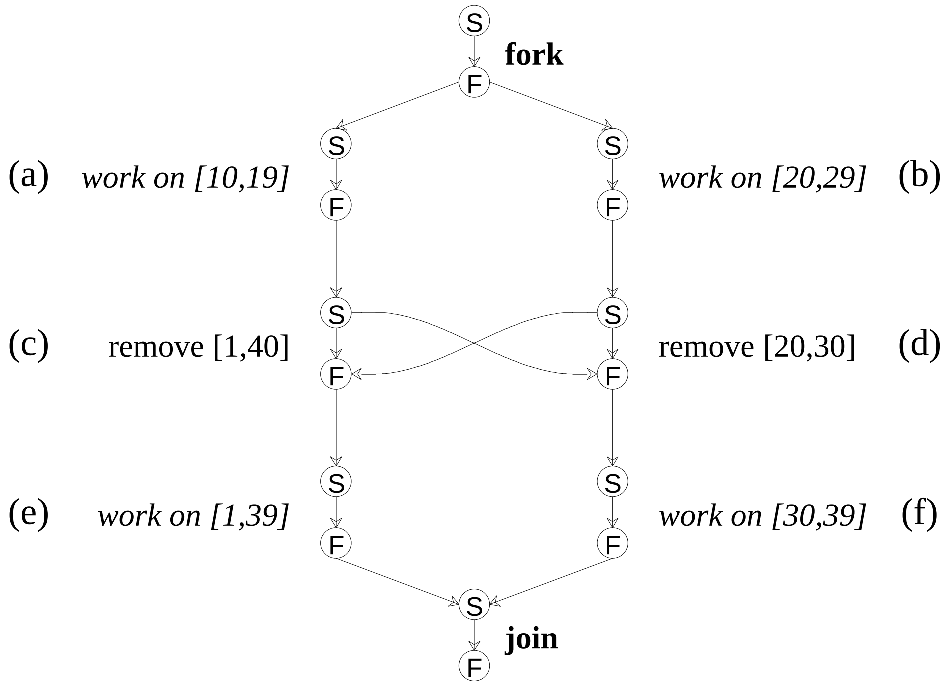


Figure 7.3. Example G_E (G augmented with event-control dependences)

(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 $G'_{D-ACTUAL}$, constructed by removing all nodes (and any incident edges) from $G_{D-ACTUAL}$ 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 Chapter 3). 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 $G_{D-ACTUAL}$. Since $\xrightarrow{T'}$ is constructed from $G_{D-ACTUAL}$, such a path also implies that $y \not\xrightarrow{T'} x$. Axioms (A4)–(A6) are satisfied since all linear orderings of G obey these axioms (see Chapter 6), 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 $a \overset{T'}{\not\longleftrightarrow} b$. 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 $G'_{D-ACTUAL}$ 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 $a \overset{T'}{\not\longleftrightarrow} b$. Therefore, since a feasible program execution prefix, $PP' = \langle E', \xrightarrow{T'}, \xrightarrow{D'} \rangle$,

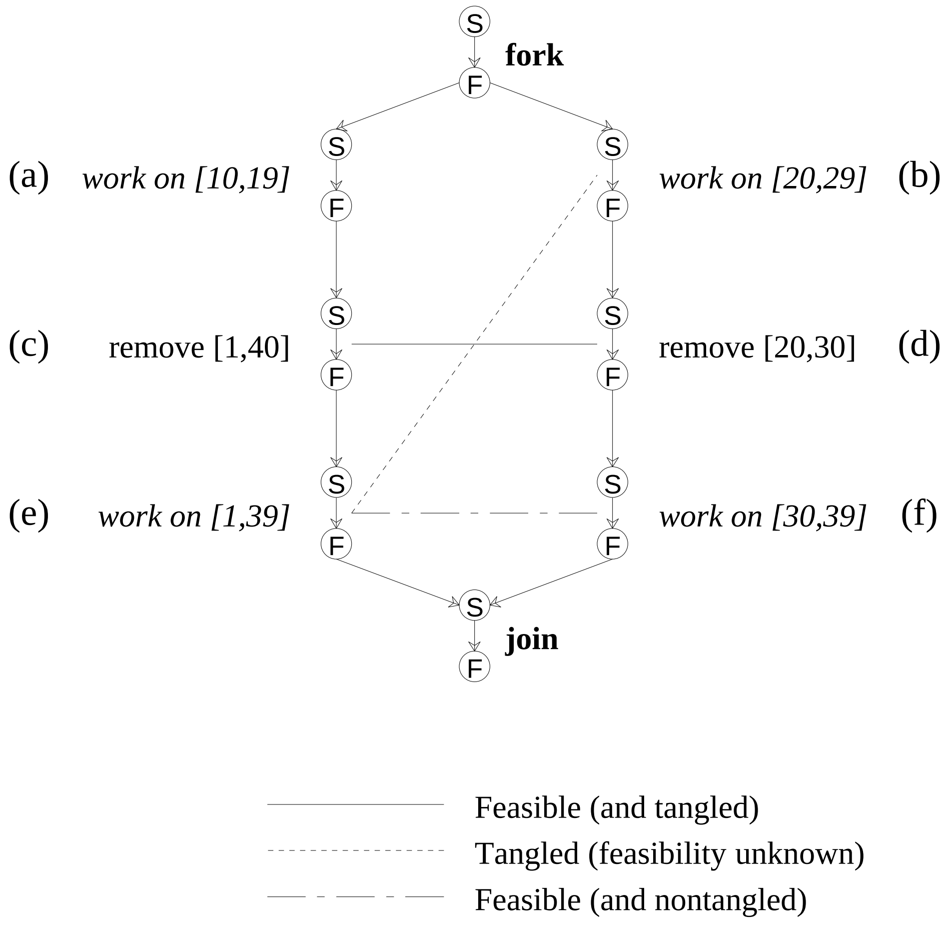


Figure 7.4. Results of validation using G_E

exists in which $a \overset{T'}{\not\longleftrightarrow} b$, the apparent data race $\langle a,b \rangle$ is feasible. ■

Figure 7.4 shows which tangled data races can be validated by G_E for our example. Of the two tangled data races ($\langle c,d \rangle$ and $\langle e,b \rangle$), two events, c and d, are no longer ordered by G_E (nor are they event-controlled by any of their successors in G_E). By Theorem 7.3, the tangled data race $\langle c,d \rangle$ is feasible. The other events, e and b, are still ordered by G_E, so the feasibility of the race $\langle e,b \rangle$ is unknown (recall that this race is indeed infeasible). In general, it is difficult to prove that a race is infeasible.

7.2. Algorithms for Post-Mortem Validation

We now present algorithms for performing post-mortem data race validation. First, we present algorithms for augmenting the graph with shared-data and event-control dependence edges. Since the augmented graphs can contain cycles, we then show how to generalize timestamps to apply to cyclic graphs so the orderings given by these graphs can be quickly determined. Finally, we bring all the pieces together to perform validation. Our goal in developing these algorithms was to require no more space than previous post-mortem methods use for simply locating apparent data races, while incurring only a modest time increase.

7.2.1. Augmenting the Temporal Ordering Graph

Augmenting G involves computing conservative approximations to the shared-data and event-control dependences. These approximations can be computed by simply analyzing the *READ* and *WRITE* sets for data conflicts, and by optionally using additional information obtained by static analysis. Below we discuss these approximations and present algorithms for computing both the direct and transitive dependences.

Estimating shared-data dependences involves simply searching for pairs of computation events that have data conflicts. However, event-control dependences can be approximated to varying degrees of accuracy, depending on the amount of available information. Because $\overset{\hat{E}}{\longrightarrow} \subseteq (\overset{\hat{D}}{\longrightarrow} \cup \overset{\hat{T}}{\longrightarrow})$, only those event-control dependences that are also shared-data dependences must be computed (G already contains edges representing $\overset{\hat{T}}{\longrightarrow}$). Such a dependence involves an event, a, that modifies a variable whose value is used by another event, b, to determine the outcome of a conditional or the shared locations accessed (e.g., with a shared-array subscript). Because a must modify a variable that b later reads, a first approximation to $a \overset{\hat{E}}{\longrightarrow} b$ can be computed by excluding dependences from $\overset{\hat{D}}{\longrightarrow}$ where $WRITE(a) \cap READ(b) = \emptyset$. By examining the individual actions performed by a and b, better approximations can be computed. A static analysis of the program can further refine $\overset{\hat{D}}{\longrightarrow}$, and even more accurate information can be extracted from a complete address trace, but this approach may be practical only if such a trace is already being collected for other purposes. The algorithms described below conservatively approximate $\overset{\hat{D}}{\longrightarrow}$ and $\overset{\hat{E}}{\longrightarrow}$ by analyzing only the *READ* and *WRITE* sets (which contain the shared variables accessed by each event but not the intra-event access order).

Computing the direct dependences is the first step in this approximation. A direct dependence from a to b exists if a accesses a shared variable that b later accesses (and at least one access is an update). Edges must be added

Given: The temporal ordering graph, G.

Compute: Augment G with direct shared-data and event-control dependence edges.

Algorithm:

```
AugmentDirect_D̂ (G):
    for each computation event, e_{p,i}, represented by G {
        for each process q > p {
            let e_{q,j} be the last event in q that is a predecessor of e_{p,i};
            j = j + 1;
            while (e_{p,i} and e_{q,j} are unordered by G) {
                if (data conflict between e_{p,i} and e_{q,j}) {
                    AddEdge (D̂, e_{p,i}, e_{q,j}, IF_NONE_EARLIER);
                    AddEdge (D̂, e_{q,j}, e_{p,i}, ALWAYS);
                }
                j = j + 1;
            }
        }
    }

AugmentDirect_Ê (G):
    for each computation event, e_{p,i}, represented by G {
        for each process q > p {
            let e_{q,j} be the last event in q that is a predecessor of e_{p,i};
            j = j + 1;
            while (e_{p,i} and e_{q,j} are unordered by G) {
                if (WRITE (e_{p,i}) ∩ READ (e_{q,j}) ≠ ∅)
                    AddEdge (Ê, e_{p,i}, e_{q,j}, IF_NONE_EARLIER);
                if (READ (e_{p,i}) ∩ WRITE (e_{q,j}) ≠ ∅)
                    AddEdge (Ê, e_{q,j}, e_{p,i}, ALWAYS);
                j = j + 1;
            }
        }
    }
```

Algorithm 7.1. Augment graph with direct dependences

Given: An edge type ($\hat{\mathrm{D}}$ or $\hat{\mathrm{E}}$) and two events.

Compute: Add an edge to the temporal ordering graph.

Algorithm:

AddEdge (*depType*, $e_{p,i}$, $e_{q,j}$, *when*):
    case (*when*) {
        IF_NONE_EARLIER:
            add *depType* edge from start node of $e_{p,i}$ to finish node of $e_{q,j}$,
                if no edge from $e_{p,i}$ to process q already exists;
        ALWAYS:
            add *depType* edge from start node of $e_{p,i}$ to finish node of $e_{q,j}$,
                replacing any existing edge from process p to $e_{q,j}$;
    }

Algorithm 7.2. Add edges for augmenting the temporal ordering graph

to ensure a path exists from $\mathbf{a_s}$ to $\mathbf{b_f}$. When $a \xrightarrow{\hat{\mathrm{T}}} b$, such a path already exists, so direct dependence edges are only required when $a \overset{\hat{\mathrm{T}}}{\not\longleftrightarrow} b$. Because these events are exactly those involved in apparent data races, direct dependence edges can be added while apparent races are located. Algorithm 7.1, which is a modification of the apparent data race detection algorithm of Chapter 6, adds these edges. This algorithm scans, for every event a, those events in each process p that are unordered with a, searching for data conflicts. Instead of adding an edge for every direct dependence found, redundant edges are omitted: an edge from a is added (by Algorithm 7.2) only to the earliest event in process p to which a dependence exists. An advantage is that at most p out-edges are added from any node, maintaining the $O(np)$ space bound. The running time is the same as for Algorithm 6.3: $O(np+U)$), where n and p are the number of events and processes, and U is the total number of unordered event pairs. In the worst case, U is of order n^2, leading to a worst-case running time of $O(n^2)$.

Computing the transitive dependences is the second step in augmenting the graph. A transitive dependence from a to b exists if a accesses a shared variable that another event x later accesses, and then x references a different shared variable that b finally references. Adding transitive dependences is non-trivial because simply computing the transitive closure of the direct dependences is overly conservative. For example, any dependence from a to b (direct or transitive) can exist only if $b \overset{\mathrm{T}}{\not\longrightarrow} a$ (axiom (A3)). A transitive dependence $a \xrightarrow{\mathrm{D}} b$ can therefore exist only if $b \overset{\mathrm{T}}{\not\longrightarrow} x \wedge x \overset{\mathrm{T}}{\not\longrightarrow} a \wedge b \overset{\mathrm{T}}{\not\longrightarrow} a$. Because events can in general represent several shared-memory accesses, it is possible that b can precede a even though direct dependences exist from a to x to b (if x overlaps both a and b). In this case, $b \xrightarrow{\mathrm{T}} a$ even though $b \overset{\mathrm{T}}{\not\longrightarrow} x \wedge x \overset{\mathrm{T}}{\not\longrightarrow} a$. Adding a transitive dependence from a to b would be overly conservative because no such dependence could exist. To determine when a transitive dependence might exist, the relative ordering must be analyzed of all events in a chain of direct dependences. As with direct dependences, where an edge is necessary only when $a \overset{\hat{T}}{\not\longleftrightarrow} b$, a transitive edge representing $a \xrightarrow{\hat{D}} b$ is required only when $a \overset{\hat{T}}{\not\longleftrightarrow} x \wedge x \overset{\hat{T}}{\not\longleftrightarrow} b \wedge a \overset{\hat{T}}{\not\longleftrightarrow} b$. In general, for a chain of direct dependences involving n events to form a transitive dependence, all n events must be mutually unordered.

Algorithm 7.3 adds transitive shared-data dependence edges, and Algorithm 7.4 adds transitive event-control dependence edges. Both algorithms are identical except for the type of direct dependences for which they search (in

---

Given: The temporal ordering graph, G, augmented with direct dependences.

Compute: Augment G with transitive shared-data dependence edges.

Algorithm:

```
AugmentTransitive_D̂ (G,  depType):
    for each event, orig, in G from which a direct D̂ dependence exists {
        mark unvisited all events unordered with orig by G;
        Visit(orig, NULL);
    }

Visit (e_p,i,  path)
    mark e_p,i as visited;
    add e_p,i to path;
    for each direct D̂ dependence from e_p,i to process j {
        let e_q,j be the last event in q that is a predecessor of any event in path;
        j = j + 1
        while (e_q,j is unvisited and e_q,j is unordered by G with all events in path) {
            if (data conflict between e_p,i and e_q,j) {
                AddEdge(D̂, orig, e_q,j, IF_NONE_EARLIER);
                Visit(e_q,j, path);
                break out of while loop;
            }
            j = j + 1;
        }
    }
```

Algorithm 7.3. Augment graph with transitive shared-data dependences

---

Given: The temporal ordering graph, G, augmented with direct dependences.

Compute: Augment G with transitive event-control dependence edges.

Algorithm:

1: AugmentTransitive_$\hat{E}$ (G, *depType*):
2:   **for** each event, *orig*, in G from which a direct $\hat{E}$ dependence exists {
3:   mark *unvisited* all events unordered with *orig* by G;
4:   Visit(*orig*, NULL);
5:   }

6: Visit ($e_{p,i}$, *path*)
7:   mark $e_{p,i}$ as *visited*;
8:   add $e_{p,i}$ to *path*;
9:   **for** each direct $\hat{E}$ dependence from $e_{p,i}$ to process j {
10:   let $e_{q,j}$ be the last event in q that is a predecessor of any of the events in *path*;
11: $j = j + 1$
12:   **while** ($e_{q,j}$ is *unvisited* **and** $e_{q,j}$ is unordered by G with all events in *path*) {
13:   **if** ($WRITE\,(e_{p,i}) \cap READ\,(e_{q,j}) \neq \emptyset$) {
14:   AddEdge($\hat{E}$, *orig*, $e_{q,j}$, IF_NONE_EARLIER);
15:   Visit($e_{q,j}$, *path*);
16:   **break** out of **while** loop;
17:   }
18: $j = j + 1$;
19:   }
20:   }

Algorithm 7.4. Augment graph with transitive event-control dependences

lines 2, 9, 13, 14). For each event, *orig*, these algorithms use a depth-first search to follow a path of direct dependences involving only events that are unordered with all previous events along the path. The procedure Visit attempts to extend the current path by finding a direct dependence from $e_{p,i}$ to an event $e_{q,j}$ that qualifies as a transitive dependence from *orig*. To qualify, there must be a direct dependence from $e_{p,i}$ to $e_{q,j}$ (or to a preceding event in process q) and $e_{q,j}$ must be unordered with all events in *path* (line 12). For each event considered (by lines 10–12), $O(p)$ work is required to determine if it qualifies, since *path* contains up to p events (at most p events can be mutually unordered since two events belonging to the same process are always ordered by $\xrightarrow{\hat{T}}$). Since only events unordered with *orig* are ever considered, and no event is visited twice, at most $O(up)$ work is performed for computing transitive dependences from *orig*, where u is the number of events unordered with *orig*. For an arbitrary event, $O(\bar{u}p)$ work is performed on the average, where $\bar{u}$ is the average number of events unordered with any given event. The total running time of the algorithm is thus $O(n\bar{u}p) = O(Up)$, where U is the total number of unordered event pairs. In the worst case, U is of order n^2, leading to a worst-case running time of $O(n^2p)$. As with direct dependences, at most p transitive out-edges are ever added for any event, maintaining the $O(np)$ space bound.

7.2.2. Computing Timestamps for Cyclic Graphs

Previously, timestamps have only been used to represent connectivity in acyclic graphs[16, 33, 35, 50], as was done in Chapter 6. However, because the augmented graphs may contain cycles, we now generalize the use of timestamps to cyclic graphs and show how to compute them for G_E. Timestamps for G_D are unnecessary (even

---

Given: The augmented temporal ordering graph, G_E.

Compute: The timestamp for each node.

Algorithm:

```
Compute_Ê_Timestamps (G):
    topOrder = TopologicalSort(G);
    for each node, x, in G {
        x.timestamp = ⟨−1,...,−1⟩;
        x.timestamp[x.proc_num] = x.ser_num;
    }
    for each node, x, in the list topOrder {
        tmp = ⟨−1,...,−1⟩;
        for each node, y, in the same SCC as x
            tmp = TimestampMax(tmp, y.timestamp);
        for each node, y, in the same SCC as x {
            y.timestamp = tmp;
            for each out-edge from y to z
                z.timestamp = TimestampMax(z.timestamp, tmp);
        }
    }
```

Algorithm 7.5. Compute timestamps for G_E

---

though it may be cyclic); identifying tangled races only requires locating strongly connected components.

Algorithm 7.5 computes timestamps for G_E. To handle cycles, the graph is conceptually *reduced* by treating each strongly connected component as a single meta-node; an edge emanating from any node in the component is treated as an edge emanating from the meta-node. Because the reduced graph is acyclic, timestamps can be easily computed (all nodes in the same component have the same timestamp). Algorithm 7.5 computes timestamps in the same way as Algorithm 6.1 (for acyclic graphs), except that the topological order it traverses contains only one representative node from each component, which is treated as the meta-node for that component. Algorithm 7.6

Given: The augmented temporal ordering graph, G_E.

Compute: A topological sort of the graph.

Algorithm:

```
TopologicalSort (G_E):
    locate the strongly connected components (SCCs) of G_E;
    mark each node in G_E as unvisited;
    topOrder = NULL;
    for each process p {
        if (the first node, n, in p has no in-edges)
            Visit(n);
    }
    return topOrder;

Visit (x):
    mark as visited the node x and all nodes in the same SCC;
    for each node, y, in the same SCC as x {
        for each out-edge from y to z {
            if (z is unvisited)
                Visit(z);
        }
    }
    add x to the beginning of the list topOrder;
```

Algorithm 7.6. Compute a topological order for G_E

---

computes the topological order by a generalization of the standard depth-first-search approach to topological sorting[66]. When the depth-first search this algorithm performs visits a node in a component, it avoids visiting any other node in the component by marking them all visited (line 12), and then follows out-edges from all nodes in the component (lines 13–14). The resulting topological order contains one representative node from each component.

The running time of Algorithm 7.6 is $O(n+e)$, the same as for standard topological sorting. Each node is visited (and each edge is followed) exactly once, and locating all the strongly connected components[66] only requires $O(n+e)$ time. Since Algorithm 7.5 operates in the same way as for Algorithm 6.1, its running time is also the same, $O(ep)$. However, in the worst case, e can be of order np; each node in the augmented graph can have a dependence edge to every other process. The worst-case running time is thus $O(np^2)$. However, $O(np)$ edges exist only when every event participates in at least one race; for most cases we expect e to be of order n. As with Algorithm 6.1, $O(np)$ space is required to store the timestamps.

7.2.3. Validation

We finally present algorithms for actually performing data race validation. Algorithm 7.7 contains procedures for validation using G_D (Validate_$\hat{\mathrm{D}}$) and using G_E (Validate_$\hat{\mathrm{E}}$).

The time required for Validate_$\hat{\mathrm{D}}$ is simply the sum of the times to augment the graph with direct and transitive dependences, locate strongly connected components, and then check each apparent race: $O(np+U)+O(pU)+O(n+e)+O(r)=O(np+pU+r)$, where r is the number of apparent data races. In the worst case, r is of order n^2 and U is of order n^2, leading to a worst-case running time of $O(n^2p)$. Similarly, the time required for Validate_$\hat{\mathrm{E}}$ is $O(np+U)+O(pU)+O(ep)+O(t)=O(p(n+U+e)+t)$, where t is the number of tangled data races. In the worst case, e is of order np, leading to a worst-case running time of $O(n^2p+np^2)=O(n^2p)$.

The space required by these algorithms for the augmented graph is $O(np)$, for storing timestamps and edges. This space bound is no worse than previous post-mortem methods which only detect apparent data races. However, the worst-case time required ($O(n^2p)$) is greater than the worst-case time for simply locating apparent races ($O(n^2)$).

Given: The temporal ordering graph, G, and the apparent data races.

Compute: Validate the apparent data races.

Algorithm:

Validate_$\hat{\mathrm{D}}$ (G):
    AugmentDirect_$\hat{\mathrm{D}}$(G);
    AugmentTransitive_$\hat{\mathrm{D}}$(G);
    locate the strongly connect components (SCCs) of G;
    for each apparent data race $\langle a,b \rangle$ {
        if ($\mathbf{a_f}$ and $\mathbf{b_s}$, or $\mathbf{b_f}$ and $\mathbf{a_s}$, belong to the same SCC)
            output $\langle a,b \rangle$ is tangled;
        else
            output $\langle a,b \rangle$ is feasible;
    }

Validate_$\hat{\mathrm{E}}$ (G):
    AugmentDirect_$\hat{\mathrm{E}}$(G);
    AugmentTransitive_$\hat{\mathrm{E}}$(G);
    Compute_$\hat{\mathrm{E}}$_Timestamps(G);
    for each tangled data race $\langle a,b \rangle$ {
        if (OrderOf($\mathbf{a_f}$,$\mathbf{b_f}$) = UNORDERED)
            output $\langle a,b \rangle$ is feasible;
    }

Algorithm 7.7. Validate apparent data races

Chapter 8

ORDERING APPARENT DATA RACES

In this chapter we address the second cause of data race artifacts: races that occurred only because of other races. We present a technique called data race *ordering* for determining which apparent data races may have affected others. Ordering involves defining a relation over the apparent data races that shows how data flowed through the execution. Those races that are ordered first by this relation are guaranteed to be non-artifacts, since they could not have been affected by other races. We define data race ordering using the event-control dependences and present results that show how to identify the first races. We then present an algorithm for post-mortem data race ordering and contrast race ordering with related work. In the next chapter, we present the results of experiments involving an implementation, which suggest that ordering is an effective tool in practice for debugging.

8.1. Ordering Results

We first use the event-control dependences to define an ordering over the apparent data races. This ordering shows when atomicity failures caused by one race can affect the outcome of events in other races. The apparent races that are first in this ordering cannot be affected by any atomicity failures. These first races are guaranteed to be non-artifacts — they would have remain unchanged whether or not any other races had occurred. However, identifying first races is complicated when only approximate information is available. We therefore show how to *partition* the races, regardless of the amount of information available, so that each partition contains at least one non-artifact race. More detailed information about the event-control dependences leads to smaller partitions and hence better diagnostic precision.

To determine which data races may have been artifacts of others, we must consider how the presence of one apparent data race, $\langle c,d \rangle$, is affected by another race, $\langle a,b \rangle$. By definition, $\langle c,d \rangle$ is an artifact of $\langle a,b \rangle$ iff $\langle c,d \rangle$ would never have occurred had $\langle a,b \rangle$ not occurred either. This situation arises when $\langle a,b \rangle$ is an actual data race that results in the atomicity failure of a or b. These atomicity failures can result in inconsistent data which is subsequently used by the program, resulting in the $\langle c,d \rangle$ race, which would not have occurred without inconsistent data. If we locate data races that could not have possibly been affected by atomicity failures of other races, we are guaranteed that they are not artifacts. To locate these races, we define an ordering over the apparent data races.

Definition 8.1

The *data-race ordering* relation, $\xrightarrow{R}$, is defined over the apparent data races, and shows when one race may have been affected by the possible atomicity failures caused by another race:

$$\langle a,b\rangle \xrightarrow{R} \langle c,d\rangle \;\Leftrightarrow\; (a \xrightarrow{E} c \;\wedge\; b \xrightarrow{E} c) \;\vee\; (a \xrightarrow{E} d \;\wedge\; b \xrightarrow{E} d).$$

The *approximate data-race ordering* relation, $\xrightarrow{\hat{R}}$, is a conservative approximation to $\xrightarrow{R}$ (i.e., $\xrightarrow{\hat{R}} \supseteq \xrightarrow{R}$), and is defined as above by using $\xrightarrow{\hat{E}}$ instead of $\xrightarrow{E}$. ■

If $\langle a,b\rangle \xrightarrow{R} \langle c,d\rangle$, then $\langle c,d\rangle$ could possibly have been an artifact of $\langle a,b\rangle$. The somewhat surprising aspect of this relation is that $\langle c,d\rangle$ cannot have been an artifact unless *both* a and b event-control c (or d). Intuitively, if only one of a and b event-control c, then even if the atomicity of a or b failed, the shared-memory accesses causing the failure cannot affect the outcome of c or d. The following theorem proves this observation.

Theorem 8.1 (Data-Race Ordering Theorem)

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 of $\langle a,b\rangle$ 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. We will view the execution at a lower level by letting a_{atom} and b_{atom} be the initial portions of a and b that executed atomically and 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, it suffices to show that neither a_{natom} nor b_{natom} can event-control either c or d (see the construction of PP' in the proof of Theorem 7.3).

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

Ideally, to locate the non-artifact races, we would like to identify the races that are ordered *first* by the $\xrightarrow{\hat{R}}$ relation (i.e., those not affected by any others). However, because $\xrightarrow{\hat{R}}$ is not a partial order, it is not always guaranteed to order any of the races first; every race can be ordered after another. This situation occurs because $\xrightarrow{\hat{R}}$ is based on $\xrightarrow{\hat{E}}$, which can sometimes be symmetric and nontransitive. The $\xrightarrow{\hat{E}}$ relation can be symmetric, causing $\langle a,b \rangle \xrightarrow{\hat{R}} \langle c,d \rangle$ and $\langle c,d \rangle \xrightarrow{\hat{R}} \langle a,b \rangle$, because it is based on approximate information[1]. In this case, insufficient information exists to determine whether $\langle c,d \rangle$ may have been an artifact of $\langle a,b \rangle$ or *vice-versa*. The $\xrightarrow{\hat{E}}$ relation can be nontransitive, causing $\langle a,b \rangle \xrightarrow{\hat{R}} \langle c,d \rangle \xrightarrow{\hat{R}} \langle e,f \rangle$, but $\langle a,b \rangle \not\xrightarrow{\hat{R}} \langle e,f \rangle$, because events can represent several shared-memory references. If c and d represent large portions of the execution, they can both overlap the other events, allowing $\langle a,b \rangle \xrightarrow{\hat{R}} \langle c,d \rangle \xrightarrow{\hat{R}} \langle e,f \rangle$. But e and f can still both precede a and b, preventing $\langle a,b \rangle$ from affecting $\langle e,f \rangle$. To address these difficulties, we group the apparent data races into partitions such that data races belonging to different partitions can be partially ordered, allowing first *partitions* to be identified. To overcome the problem of symmetry, we add two races to the same partition if it is unknown which may have affected the other. To overcome the problem of nontransitivity, we simply force $\xrightarrow{\hat{R}}$ to be transitive by using its transitive closure to order races.

Definition 8.2

Two apparent data races, $\langle a,b \rangle$ and $\langle c,d \rangle$, belong to the same partition iff

$$\langle a,b \rangle \xrightarrow{\hat{R}*} \langle c,d \rangle \wedge \langle c,d \rangle \xrightarrow{\hat{R}*} \langle a,b \rangle,$$

where $\xrightarrow{\hat{R}*} = (\xrightarrow{\hat{R}})^*$, the transitive closure of $\xrightarrow{\hat{R}}$.

A *first partition* is one containing data races that are not ordered by $\xrightarrow{\hat{R}*}$ after a race in any other partition.
■

Because $\xrightarrow{\hat{R}*}$ is transitive, and races that cause $\xrightarrow{\hat{R}}$ to be symmetric belong to the same partition, the above definition does indeed partition the races, and races belonging to different partitions are partially ordered by $\xrightarrow{\hat{R}*}$. At least one first partition therefore always exists[2]. Theorem 8.2 below proves that each first partition is guaranteed to contain at least one race that is both feasible and not an artifact of any other race. We first prove the following lemma, and then prove the theorem.

---

[1]Even $\xrightarrow{R}$ can be symmetric, if $\langle a,b \rangle$ and $\langle c,d \rangle$ race on multiple shared variables. This case is also caused by approximate information, since the relative order of shared-memory accesses within events is unknown.

[2]Although information is lost by using the transitive closure of $\xrightarrow{\hat{R}}$, it is unclear how the (non-transitive) $\xrightarrow{\hat{R}}$ can instead be used to define a partitioning and identify the non-artifact races. We leave investigating how information about non-transitivity may be used to improve accuracy to future work.

Lemma 8.1

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

Proof.

As in Chapter 7, we employ $G_{E-ACTUAL}$, the temporal ordering graph augmented with event-control dependence edges representing actual event-control dependences. If $\langle a,b \rangle$ 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 $G_{E-ACTUAL}$ event-control b or predecessors of b, or *vice-versa*. Assume the former. $G_{E-ACTUAL}$ 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 $a \xrightarrow{\hat{\text{T}}} x_0 \xrightarrow{\text{E}} b$ for some event x_0. Since $x_0 \xrightarrow{\text{E}} b$, there exists a chain of events, $x_0, x_1, \cdots, x_n$ (x_n=b), such that $x_0 \; (\xrightarrow{\text{DD}} \cup \xrightarrow{\hat{\text{T}}}) \; x_1 \; (\xrightarrow{\text{DD}} \cup \xrightarrow{\hat{\text{T}}}) \; \cdots \; (\xrightarrow{\text{DD}} \cup \xrightarrow{\hat{\text{T}}}) \; x_n$. Since $a \overset{\text{T}'}{\not\longleftrightarrow} b$, not all events in this chain can be ordered by $\xrightarrow{\hat{\text{T}}}$, so $x_i \xrightarrow{\text{DD}} x_{i+1}$ for some i. An apparent data race therefore exists between x_i and x_{i+1}. By condition (1) of the definition of $\xrightarrow{\text{E}}$, $x_i \xrightarrow{\text{E}} b$ for $1 \le i < n$. Since $x_i \xrightarrow{\text{E}} b$ and $x_{i+1} \xrightarrow{\text{E}} b$, we have $\langle x_i, x_{i+1} \rangle \xrightarrow{\text{R}} \langle a,b \rangle$. ■

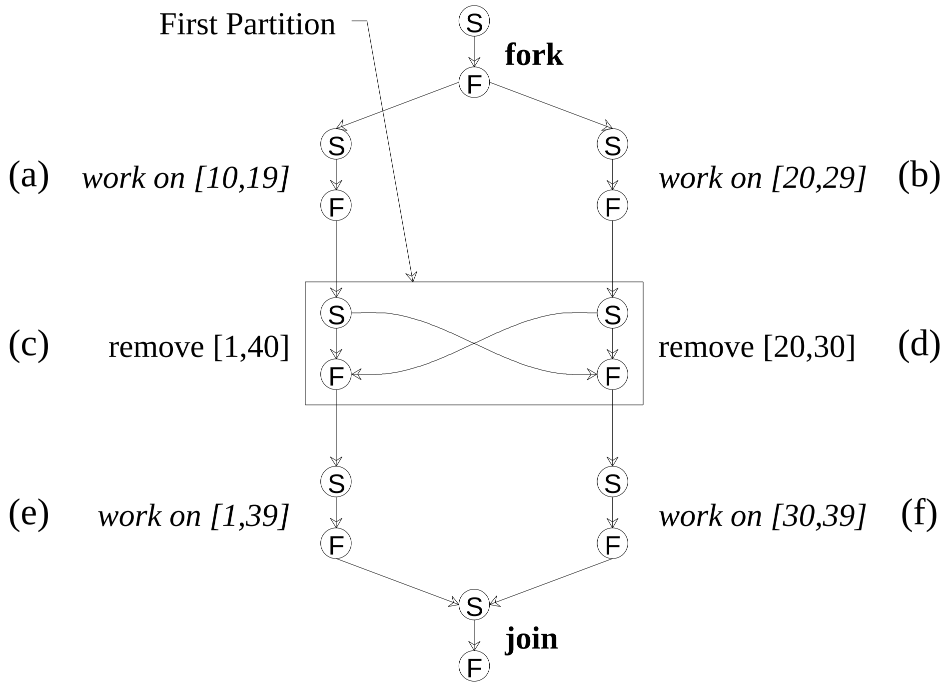
Theorem 8.2 (Partition Theorem)

Each first partition contains at least one apparent data race that is both feasible and not an artifact of any other data race.

Proof.

Consider the single-access program execution, P_S (see the proof of Theorem 3.1). Each computation event in P_S represents at most one shared-memory access, and the $\xrightarrow{\text{R}}$ relation defined over the single-access apparent data races, $\xrightarrow{\text{R}_\text{S}}$, is a partial order. Now consider any first partition of the apparent data races in P, and the corresponding single-access races in P_S. Since these single-access races are partially ordered by $\xrightarrow{\text{R}_\text{S}}$, at least one of them, $\langle a_s, b_s \rangle$, appears first in the ordering. By Theorem 8.1 and Lemma 8.1, 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 $\xrightarrow{\text{R}_\text{S}}$ ordering). The higher-level events, a and b, containing a_s and b_s thus contain a feasible data race that is not an artifact of any other race. ■

Figure 8.1 shows the data race orderings and first partition for our example; the graph G_E, which contains event-control dependence edges, is shown. In this example, each partition contains only one race, and only one partition is first. In this particular case, data race ordering alone was able to locate the only non-artifact race in the execution. In general, however, the first partitions contain more than one race. Data race validation can be applied inside each first partition to gain more information about the nature of its races. Chapter 9 discusses how validation and ordering can be used by the programmer for debugging.



$$\langle c,d\rangle \xrightarrow{\hat{R}} \langle e,b\rangle$$

$$\langle c,d\rangle \xrightarrow{\hat{R}} \langle e,f\rangle$$

$$\langle e,b\rangle \xrightarrow{\hat{R}} \langle e,f\rangle$$

Figure 8.1. Results of data race ordering

8.2. Algorithm for Post-Mortem Ordering

We now show how the first partitions can be identified. Algorithm 8.1 considers each apparent data race, $\langle a,b\rangle$, to determine whether it belongs to a first or non-first partition. This determination is made by checking the set firstPartitions, which contains partitions that are ordered first among the races considered so far. If $\langle a,b\rangle$ is found to belong to one of these partitions, then it is added (line 7). If $\langle a,b\rangle$ is affected by a race that is currently first, then it cannot belong to a first partition and is discarded (line 12). If $\langle a,b\rangle$ affects a race in a partition that is currently first, then that partition cannot actually be first and is discarded (line 17). Finally, if $\langle a,b\rangle$ is not found to be affected by any race currently ordered first, then a first partition containing only $\langle a,b\rangle$ is added to firstPartitions (line 21).

Given: The apparent data races and the data-race ordering relation $\xrightarrow{\hat{R}*}$.

Compute: The first partitions.

Algorithm:

1: LocateFirstPartitions:
2:     firstPartitions = $\varnothing$;
3:     **for** each apparent data race $\langle a,b \rangle$ {
4:         newPartition = True;
5:         **for** each partition P in firstPartitions {
6:             let $\langle x,y \rangle$ be any data race in P;
7:             **if** ($\langle x,y \rangle \xrightarrow{\hat{R}*} \langle a,b \rangle \wedge \langle a,b \rangle \xrightarrow{\hat{R}*} \langle x,y \rangle$) {
8:                 add $\langle a,b \rangle$ to the partition P;
9:                 newPartition = False;
10:                 **break**;
11:             }
12:             **else if** ($\langle x,y \rangle \xrightarrow{\hat{R}*} \langle a,b \rangle$) {
13:                 newPartition = False;
14:                 **break**;
15:             }
16:             **else if** ($\langle a,b \rangle \xrightarrow{\hat{R}*} \langle x,y \rangle$) {
17:                 remove P from firstPartitions;
18:             }
19:             /* **else** $\langle x,y \rangle \overset{\hat{R}*}{\not\longleftrightarrow} \langle a,b \rangle$ */
20:         }
21:         **if** (newPartition = True)
22:             add the new partition $\{\langle a,b \rangle\}$ to firstPartitions;
23:     }
24:     **output** firstPartitions;

Algorithm 8.1. Locate first partitions

In Algorithm 8.1, the outer **for** loop (line 3) iterates over all apparent races, and the inner **for** loop (line 5) iterates over all partitions currently in firstPartitions. The inner loop simply determines how $\langle a,b \rangle$ and $\langle x,y \rangle$ are ordered by $\xrightarrow{\hat{R}*}$, which can be performed in constant time (discussed below). In the worst case, every apparent race considered can be added to its own first partition, causing the size of firstPartitions to steadily grow. In this case, firstPartitions would contain i partitions after the i^{th} iteration of the outer loop. The total running time would then be $O(\sum_{i=1}^{r} i) = O(r^2)$, where r is the number of apparent data races (recall that r can be of order n^2). However, this worst-case bound results only when the outer **for** loop (line 3) happens to iterate over the non-first races early and not reach the first races until later. In this case, many non-first races are added to firstPartitions only to be subsequently removed when the first races are finally considered. In cases where all the first partitions are discovered early by the algorithm, the running time is $O(rf)$, where f is the number of first partitions.

Achieving the $O(r^2)$ time bound requires computing a representation of the $\xrightarrow{\hat{R}*}$ relation that can be evaluated in constant time. Computing $\xrightarrow{\hat{R}*}$ can be reduced to computing $\xrightarrow{\hat{E}*}$, the transitive closure of $\xrightarrow{\hat{E}}$, by noticing that $\xrightarrow{\hat{R}*}$ can be equivalently defined using $\xrightarrow{\hat{E}*}$:

$$\langle a,b\rangle \xrightarrow{\hat{R}*} \langle c,d\rangle \iff (a \xrightarrow{\hat{E}*} c \;\wedge\; b \xrightarrow{\hat{E}*} c) \;\vee\; (a \xrightarrow{\hat{E}*} d \;\wedge\; b \xrightarrow{\hat{E}*} d).$$

The $\xrightarrow{\hat{R}*}$ relation can be evaluated in constant time if timestamps representing $\xrightarrow{\hat{E}*}$ are available. These timestamps can be computed by Algorithm 7.5 if the event-control dependence edges in G_E are first adjusted. If the direct dependence edges are moved so that they enter start nodes instead of finish nodes, transitivity then follows from paths formed by chains of direct dependence edges. For example, direct dependences from a to b to c would be represented by direct dependence edges from $\mathbf{a_s}$ to $\mathbf{b_s}$ and from $\mathbf{b_s}$ to $\mathbf{c_s}$, resulting in a path from $\mathbf{a_s}$ to $\mathbf{c_s}$ representing the transitive closure. The cost of adjusting the graph is $O(e)$, since each direct dependence edge must be moved; the timestamps can be computed in time $O(ep)$ by Algorithm 7.5. Computing $\xrightarrow{\hat{R}*}$ by transforming the graph in this way allows the flexibility of ordering the races either before or after validation. In addition, the convention of constructing dependence edges from start node to start node is attractive when only data race ordering is being performed. In such a case, augmenting the graph with transitive edges can be avoided entirely.

8.3. Related Work

We now discuss the only other work that addresses the issue of locating first data races. Choi and Min[17] consider an on-the-fly approach for detecting data races in which they locate a subset of the races called the *Race Frontier*. Below we describe their work and contrast it with ours.

The Race Frontier approach is a modified on-the-fly scheme to identify one event in each process that participates in a race. If the first racing event in a process is nontangled, then that event belongs to the Frontier. If the first such event is tangled, then the Frontier contains the tangled race that occurred before any other race in the tangle. To determine which events occur first, additional information provided by the on-the-fly approach is used; a race check is performed at each shared-memory access, allowing the order of all accesses to the same location to be determined.

The Race Frontier provides a valuable tool for debugging data races on-the-fly. Events up to (but not including) the Race Frontier are not affected in any way by the racing events, allowing them to be easily re-executed for debugging. By breakpointing each process, during re-execution, immediately before reaching the Race Frontier, program replay can be guaranteed deterministic. Deterministic replay allows the portion of the execution that caused the races to be easily repeated for debugging. The execute/replay cycle can be repeated until all races in the execution have been debugged.

Choi and Min's work contrasts with ours in two important ways. First, a single-access view is hard-wired into their work. Identifying the Race Frontier requires detailed information about the order of individual memory accesses. This information is obtained as a by-product of the on-the-fly approach, which serializes all accesses to the same location (introducing central bottlenecks that potentially reduce parallelism). In contrast, our results apply to single-access or higher-level views in which any level of information is available, allowing them to be applied to any data race detection method.

Second, they do not directly address the issue of race artifacts. Instead, they prove that events up to the Race Frontier are guaranteed to be reproduced during replay. In contrast, one of our goals was formalizing the notion of a race artifact and locating non-artifact races. Because the Race Frontier is not affected in any way by racing events, it never contains artifacts, but it is overly pessimistic. Theorem 8.1 shows that races affected by racing events are not always artifacts. We can sometimes determining that races are not artifacts even when they did not temporally occur first (and are therefore beyond the Race Frontier).

Chapter 9

EXPERIENCE WITH VALIDATION AND ORDERING

In the previous chapters we presented results and algorithms for data race validation and ordering. We now discuss our experience with an actual implementation. We first outline both the run-time and post-mortem phases of our data race analyzer, which is targeted to parallel C programs on the Sequent Symmetry multiprocessor. We then discuss our experiments. The goal of accurate race detection is to report only those races that are direct manifestations of bugs, in the hope that these races will lead to discovering the bugs. The thrust of our experiments was to ascertain how effectively validation and ordering achieve this goal when given varying amounts of run-time information. The purpose of this study was to understand our techniques; run-time and post-mortem efficiencies were not the main concern. We recorded information at two different levels of detail. The finer level of detail records the complete temporal ordering and closely approximates the actual shared-data and event-control dependences. The coarser level detail records no more information than previous post-mortem methods and computes coarser approximations to the dependences.

We analyzed several programs containing data races, some previously developed by others (which represent a cross-section of practical programs) and some developed specifically to test our techniques (which represent more pathological cases). We collected statistics on the number of apparent data races that were successfully validated, and the number and size of the first partitions that were reported. We found that validation and ordering produced excellent results when given the finer level of trace information, completely pinpointing the non-artifact races. The coarser level of trace information produced comparable results except when tangled data races occurred. Validation and ordering therefore seem to be effective tools for reducing the number of races that require consideration for debugging.

9.1. Implementation

Our system for post-mortem detection locates data races for shared-memory parallel C programs on the Sequent Symmetry. As mentioned in Chapter 2, post-mortem race detection employs three phases: program instrumentation, program execution, and post-mortem analysis. The program instrumentation phase instruments C programs to record run-time trace information at two different levels of detail. The post-mortem analysis phase performs validation and ordering using both levels of this trace information. As discussed in a later section, analyzing races with both levels of information allows us to assess the overall effectiveness of validation and ordering, and to estimate how much information is required for them to perform well. Below, we discuss the levels of detail at which we collect run-time information, including how the program is instrumented to record this information and how it is used by the post-mortem analyzer.

9.1.1. Program Instrumentation

The program instrumentation consists of both a source-code instrumentation phase (to record the *READ* and *WRITE* sets) and instrumented libraries (to record the temporal ordering of synchronization events). There are two aspects of this instrumentation that determine at what level of detail information is recorded: granularity and completeness. The granularity at which information is collected is governed by the size of computation events. Recording the *READ* and *WRITE* sets for small computation events provides detailed information about the intra-process order of shared-memory accesses; larger events provide less detailed information. The completeness of the collected information depends on how much temporal ordering information is recorded. The complete temporal ordering ($\xrightarrow{T}$) contains exhaustive information about the event order; an approximate temporal ordering ($\xrightarrow{\hat{T}}$) contains only partial information. Our instrumentation scheme records two levels of detail. We record both complete and approximate temporal orderings, but fix the granularity by tracing *READ* and *WRITE* sets at the coarsest possible level. Each execution of the program records both forms of temporal ordering, allowing both levels of trace to be simultaneously obtained.

We fix the level of granularity by making the size of computation events maximal. Such a maximal event represents all computation performed between two synchronization operations (in the same process). This granularity represents the worst choice in terms of precision, since the *READ* and *WRITE* sets contain no information about the order in which shared-memory accesses are performed inside an event. By chosing a coarse granularity, the results we obtain about the effectiveness of validation and ordering will be conservative; a finer granularity would only provide more information. Moreover, previous post-mortem schemes have made the same choice. To record the *READ* and *WRITE* sets, the GNU C compiler (version 1.38) was modified to instrument each variable occurrence in the program. Each occurrence of a variable ‘‘`x`’’ is translated into the following expression:

```
*(tmp=&x, TRACE(tmp,accessType,sizeof(x)), (typeOfx *)tmp)
```

where ‘‘`TRACE(tmp,accessType,sizeof(x))`’’ is a call to a procedure that determines if `x` resides in shared memory, and if so, writes to a trace file the address of `x`, the access type (read, write, or read/write), its size, and the line number in which it appears; ‘‘`(typeOfx *)`’’ casts the type of the expression to the same type as `x`. In addition to scalar variables, array accesses and pointer dereferences are similarly translated[1].

We collect two different levels of trace information by simultaneously recording both approximate and complete temporal orderings (at the selected granularity). These orderings are recorded by instrumenting the `pps` and `c` libraries, which contain primitives for process creation and synchronization (with spin locks and barriers). The approximate ordering ($\xrightarrow{\hat{T}}$) is recorded by associating a counter with each lock (or barrier). The counter is incre-

---

[1]Although this tracing strategy does provide information about the intra-event order of shared-memory accesses, the post-mortem analyzer makes no use of this information.

mented and traced at each synchronization operation performed on the lock, allowing the order of all operations on that lock to be determined. Following previous approaches[35, 50], we pair each unlock operation with the subsequent lock on the same variable, and mutually pair all related barriers; we then have $a \xrightarrow{\hat{T}} b$ if a is paired with b. Recording such an approximate ordering has the advantage that the required instrumentation can be embedded into the implementation of the lock without introducing additional synchronization. A central bottleneck that could reduce the amount of parallelism achievable by the program is avoided. The complete temporal ordering ($\xrightarrow{T}$) is recorded by maintaining a global counter that is incremented and traced at each synchronization operation performed by the program. Although this complete ordering allows the relative order between any two events to be determined (at the selected granularity), maintaining the global counter introduces a central bottleneck.

9.1.2. Post-Mortem Analysis

The post-mortem analysis phase reads the trace files that are produced by the program instrumentation during execution (each process creates one trace file that contains a record for every shared-memory access and synchronization operation it issues). The analyzer constructs and augments the temporal ordering graph, locates apparent data races, and finally performs validation and ordering. Using the two levels of trace information, we construct two versions of the augmented graphs, one containing close estimates of the shared-data and event-control dependences and the other containing less precise estimates. Because the less precise estimates are computed only from the *READ* and *WRITE* sets and the approximate temporal ordering, they are based on the same level of information that previous methods record. The analyzer performs validation and ordering on both versions, and collects statistics (discussed later) about the number of races that are successfully validated and ordered. Below we discuss how these estimates are computed.

Given the complete temporal ordering, the post-mortem analyzer computes close approximations of the actual dependences. As discussed in Chapters 3 and 7, the *READ* and *WRITE* sets and the temporal ordering (complete or approximate) can be used to compute conservative estimates of the dependences. For example, if a and b have a data conflict, and $a \xrightarrow{T} b$, then we know with certainty that a shared-data dependence exists from a to b. When $a \stackrel{T}{\nleftrightarrow} b$, insufficient information is known to determine the direction of the dependence, because the order in which a and b perform their individual shared-memory accesses is not recorded at our coarse level of granularity. We therefore make the conservative assumption that dependences exist from a to b and from b to a. However, this approximation can contain (conservative) errors only between events involved in actual data races. This level of trace information thus provides close estimates of the dependences. We later argue that, in some cases, recording this level of information may also be practical.

Given the approximate temporal ordering, the post-mortem analyzer computes coarser approximations of the dependences. These coarser approximations are computed in the same way as above, except that only the approximate ordering is used. These approximations represent a more practical case based only on information that can be reasonably recorded (and that is recorded by previous post-mortem methods).

9.2. Test Programs

We analyzed two sets of parallel C programs, listed in Figure 9.1. The first set of programs (the *unmodified* programs) were developed by colleagues for other studies and were analyzed as is, without modification. With the exception of *join* (which uses shared memory for some of its synchronization), all of these programs were previously debugged by their authors and thought to be data-race-free. However, analysis uncovered executions of each program that exhibited one or more data races. These programs represent a sample of realistic programs that might be found in practice. In contrast, the second set of programs (the *modified* programs) were originally data-race-free but were modified by us to exhibit data races. In each program, a single synchronization operation was removed, and an off-by-one error was introduced into a shared-array subscript. These modifications were made to allow more complex patterns of data races to be investigated. These programs represent more pathological cases, which allow our techniques to be evaluated more thoroughly. Each of the test programs was run using four processors.

Name	Description	# source lines	Synchronization
Unmodified:			
chol	Choleski factorization	2000	**fork/join**, spin locks
join	Join two relations	6000	**fork/join**, spin locks shared memory, barriers
pnet	Network flow solver	4800	**fork/join**, spin locks
shpath	Shortest path finder	370	**fork/join**, spin locks barriers
tycho	Cache simulator	6400	**fork/join**
Modified:			
parq	Parallel queue	550	**fork/join**, spin locks
workq	Shared work queue	300	**fork/join**, spin locks

Figure 9.1. Test programs containing data races

The test programs span a variety of applications from symbolic to numeric computation. *chol* performs Choleski factorization on sparse matrices, and was run with a 20×20 matrix as input. *join* implements a hash-join algorithm for a relational database, and was run with randomly generated relations containing 5000 records each with 16 attributes. *pnet* solves minimum-cost network flow problems, and was run with a 10-node network consisting of 50 arcs. *shpath* implements a parallel version of Dijkstra's shortest path algorithm, and was run on graphs with 50 nodes that were assigned random inter-node costs. *tycho* is a cache simulator, and was run with an address trace consisting of 1024 memory references. *parq* implements a parallel queue[30]; child processes loop, queueing and dequeueing records from the queue. Each child queued and dequeued 300 records. *workq* implements the shared work queue example of Figure 2.1. Each child process dequeues a record indicating a region of a shared array to work upon, forks children to perform the work, and then continues by dequeueing another record. The children simply read and write all elements of the array region. A total of 300 records were initially added to the queue.

9.3. Empirical Results

To test the effectiveness of validation and ordering, we analyzed several executions of each program. Our experiments were aimed at determining how well these analyses perform, and how useful the results are in understanding the bugs causing the races. To assess the performance of validation and ordering, we first investigate their ability to locate non-artifact races using the close approximations of the dependences. We obtain a bound on how accurately non-artifact races can be pinpointed when given precise information. In some cases, recording this information may be possible, allowing the bound to be achieved in practice. In other cases, this bound allows us to understand how well validation and ordering perform when less complete information must be recorded. We then investigate how much diagnostic precision is lost when given only the coarser approximations of the dependences. As discussed earlier, these approximations are obtained solely from the recorded approximate temporal ordering and the *READ* and *WRITE* sets. By comparing the results given these two different levels of detail, we can assess the effectiveness of our analyses, and estimate how much information is required for accurate results in practice. Finally, we explore how the results of validation and ordering might be used for debugging the data races.

9.3.1. Using Close Approximations

The effectiveness of validation and ordering is measured by how *precisely* and *completely* they locate non-artifact races. Non-artifacts are precisely located if each first partition typically contains few races, allowing the programmer to be lead directly to non-artifact races. Non-artifacts are completely located if most non-artifact races are ordered first, causing few program bugs to be hidden. Below we consider how well validation and ordering perform when given a close approximation of the actual dependences. Doing so allows us to determine whether our techniques are fundamentally useful at all, and establishes an upper bound on the diagnostic precision that can achieved. Figures 9.2 and 9.3 summarize the results for both sets of our test programs. These figures contain results of using both the close approximations (labeled with ‘‘○’’) and the coarser approximations (labeled with ‘‘◯’’).

		chol	*join*	*pnet*	*shpath*	*tycho*
Race Detection						
Apparent races		23	19	64	631	4
Actual races		1 (4%)	6 (32%)	64 (100%)	631 (100%)	4 (100%)
Validation						
Nontangled races		23 (100%)	19 (100%)	64 (100%)	631 (100%)	4 (100%)
Ordering						
First partitions	(○)	2	1	1	3	2
	(◯)	1	1	1	3	2
Average partition size	(○)	2 (9%)	1 (5%)	1 (2%)	1 (<1%)	1 (25%)
	(◯)	2 (9%)	1 (5%)	1 (2%)	1 (<1%)	1 (25%)
Total number of first races	(○)	3 (13%)	1 (5%)	1 (2%)	3 (<1%)	2 (50%)
	(◯)	2 (9%)	1 (5%)	1 (2%)	3 (<1%)	2 (50%)

Figure 9.2. Results of post-mortem analysis for unmodified programs
Numbers are averages over all executions of each program
and have been rounded off to the nearest integer.
Results of analysis with close approximations (marked with ‘‘○’’) and
coarser approximations (marked with ‘‘◯’’) are shown.
Percentages indicate the fraction of apparent data races.

Below we only consider the precision and completeness of the results obtained with the close approximations. We will discuss the results obtained with the coarser approximations in the next sub-section.

To assess the precision of validation and ordering, we measured the number of races validated and the average size of each first partition. Figure 9.2 shows the results for the unmodified test programs. In executions of these programs, no tangled races were discovered, allowing every apparent race to be validated. In addition, all first partitions contained only one or two races. Figure 9.3 shows the results for the modified test programs. Executions of these programs exhibited many tangled races. Even in the presence of these tangles, validation was able to prove that 46–91% of the races were feasible. In addition, the first partitions contained only 1–4 races, all of which were validated.

To assess the completeness of validation and ordering, we investigated the first races to see if they reflected most of the race-causing bugs. Examination of our test programs showed that the number of first races correlated well with the number of such bugs. For example, in *chol*, a single first race was reported, and we found that the program contained a single bug (a missing synchronization operation). In *shpath*, two first races were reported, reflecting two different program bugs (incorrect shared-array indexing). In addition, we closely examined selected executions of *parq* and *workq* and found that each first partition always contained only non-artifact races, even though many race artifacts typically occurred in the executions. *parq* exhibited four first races, representing two instances of each bug introduced. In contrast, *workq* exhibited only one first race, and is the only test case in which one of the introduced bugs was hidden because a non-artifact race was never ordered first.

These results suggest that validation and ordering are fundamentally capable techniques, locating races with high precision and completeness. The tens to hundreds of apparent races were precisely narrowed down to a few feasible races guaranteed to not be artifacts of other races. Moreover, the first partitions typically contained races that completely covered the race-causing bugs in the execution. Even though not all non-artifact races were ordered first, the first partitions usually contained at least one race caused by each program bug.

9.3.2. Using Coarser Approximations

To assess the effectiveness of validation and ordering in practice, we now consider how well they locate non-artifacts when less complete run-time information is available. The coarser approximations are derived from no more information than that recorded by previously proposed post-mortem methods (the ordering graph and the *READ* and *WRITE* sets). For executions of the unmodified programs (which exhibited no tangled races), the results varied little with the level of information available. However, for executions of the modified programs (which exhibited many tangled races), validation and ordering were less accurate with the coarser approximations. Below we discuss these results and notice that tangled data races explain the disparity. In the next sub-section we explore why the unmodified programs exhibited no tangled races while almost all races in the modified programs were tangled.

For executions of the unmodified test programs, Figure 9.2 shows that the results of validation and ordering were excellent even when more limited run-time information was used. First races were precisely located, with each first partition containing no more races than when more precise information was used (except for *chol*, where the partitions were only slightly larger). In one case, one fewer first partition was identified; the coarser approximation to $\xrightarrow{\mathrm{E}}$ ordered a race non-first that the closer approximation determined was not an artifact. However, the single bug in *chol* was still reflected in the first races reported. The absence of tangled races explains these results. Investigation of the executions showed that the large number of races is caused not by a large number of bugs in the program but by looping; different instances of the same points in the program race many times. When such races are nontangled, it is always possible to determine how they affect each other. For example, consider two races, $\langle a,b \rangle$ and $\langle c,d \rangle$, where a and c (and b and d) represent different instances of the same portions of the program. If the races are nontangled, because for example $a \xrightarrow{\hat{\mathrm{T}}} c$ and $b \xrightarrow{\hat{\mathrm{T}}} d$, then sufficient information always exists to order

		parq		*workq*	
Race Detection					
Apparent races		7522		2890	
Actual races		777	(10%)	263	(9%)
Validation					
Tangled races		7424	(99%)	2890	(100%)
Tangles		1		1	
Average tangle size		6438	(87%)	2890	(100%)
# races validated using G_D	(○)	777	(10%)	386	(14%)
	(◯)	99	(1%)	0	(0%)
# races validated using G_E	(○)	6072	(81%)	945	(33%)
	(◯)	900	(12%)	15	(<1%)
Total number of validated races	(○)	6849	(91%)	1330	(46%)
	(◯)	999	(13%)	15	(<1%)
Ordering					
First partitions	(○)	1		1	
	(◯)	1		1	
Average partition size	(○)	4	(<1%)	1	(<1%)
	(◯)	2604	(32%)	2860	(99%)
Total number of first races validated	(○)	4	(<1%)	1	(<1%)
	(◯)	98	(<1%)	15	(<1%)
Total number of first races	(○)	4	(<1%)	1	(<1%)
	(◯)	2604	(32%)	2860	(99%)

Figure 9.3. Results of post-mortem analysis for modified programs
Numbers are averages over all executions of each program and have been rounded off to the nearest integer.
Results of analysis with close approximations (marked with ‘‘○’’) and coarser approximations (marked with ‘‘◯’’) are shown.
Percentages indicate the fraction of apparent data races.

them: either $\langle a,b\rangle \xrightarrow{\hat{\mathrm{R}}} \langle c,d\rangle$ or $\langle a,b\rangle \overset{\hat{\mathrm{R}}}{\not\longleftrightarrow} \langle c,d\rangle$, and $\langle c,d\rangle \overset{\hat{\mathrm{R}}}{\not\longrightarrow} \langle a,b\rangle$ (because $c \overset{\hat{\mathrm{E}}}{\not\longrightarrow} a$ and $d \overset{\hat{\mathrm{E}}}{\not\longrightarrow} b$). If all races involving the same program sections are nontangled, they will always belong to different partitions, allowing the first partitions to be accurately identified.

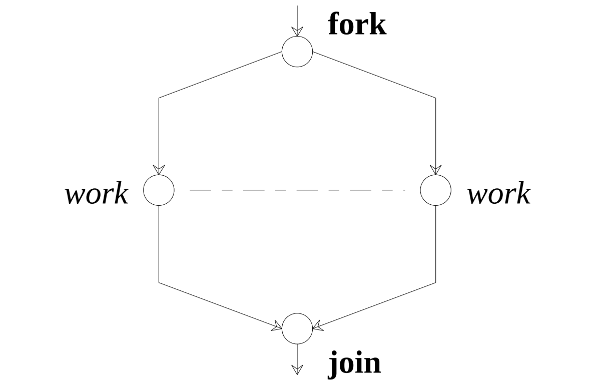
However, for executions of the modified test programs, Figure 9.3 shows that validation and ordering were less accurate. The first partitions contained 32–99% of the races, many of which were left unvalidated. Even though the first partitions were still complete (they covered all race-causing program bugs), the low accuracy diminishes the usefulness of completeness. Moreover, validation was only able to validate up to 13% of the races. The presence of tangled races explains these results. Investigation showed that many of the tangled races (which were actually feasible) were left unvalidated because the coarse approximation to $\xrightarrow{\mathrm{E}}$ provided almost no more information about the event-control dependences than the approximation to $\xrightarrow{\mathrm{D}}$. Because many events that accessed a shared variable both read and modified the variable, analysis of the *READ* and *WRITE* sets to estimate $\xrightarrow{\mathrm{E}}$ produced little useful information, resulting in $\xrightarrow{\hat{\mathrm{E}}} \approx (\xrightarrow{\hat{\mathrm{D}}} \cup \xrightarrow{\hat{\mathrm{T}}})$. In such a case, when two races $\langle a,b\rangle$ and $\langle c,d\rangle$ are tangled, insufficient information is available to determine which affected the other; they will always belong to the same partition. When large tangles exist that include the non-artifact races, the first partitions will encompass the tangles and themselves become large.

These results suggest that validation and ordering perform very well given our coarse approximations when no tangled races occur, but can perform poorly in the presence of tangles. However, even when tangled races occur, validation was able to sometimes to prove feasibility of some races in the first partitions. We later argue that this information is useful for debugging, even though the large partitions may render it impossible to discover all race-causing bugs.

9.3.3. The Nature of Tangled Data Races

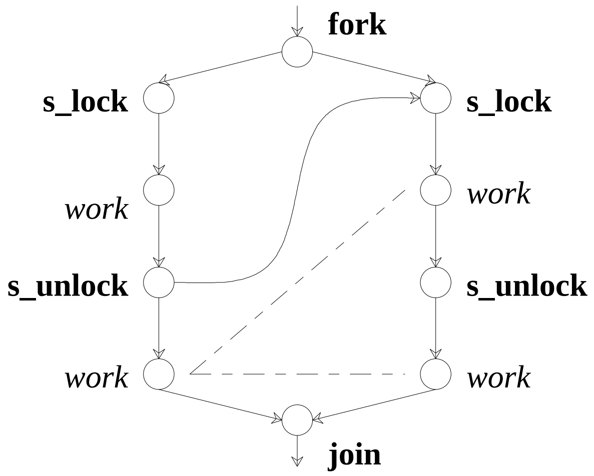
The disparity between the results under the two levels of trace information is explained by the presence or absence of tangled races. To understand why some test programs exhibited many tangled races while others exhibited none, we investigate the way each program uses synchronization. Understanding the nature of tangled races may provide some insight into when validation and ordering can be expected to perform well. The unmodified programs exhibited no tangled races because of their simple synchronization patterns. The modified test programs used more complex patterns that lead to tangles. Figures 9.4 and 9.5 show code fragments that illustrate the essence of these patterns. Figure 9.4 shows two example code fragments from the unmodified programs; code executed by each child process is shown with an ordering graph for one execution (only two processes, and one node per event, is shown for clarity). Figure 9.5 shows an example fragment from one of the modified programs; this program divides its children into groups of two which execute different code (an ordering graph for a four-process execution is shown).

```
child()
{
    for (...) {
        work on shared data;
    }
}
```



(a)

```
child()
{
    for (...) {
        s_lock(&globalLock);
        work on shared data;
        s_unlock(&globalLock);
        work on shared data;
    }
}
```



(b)

– – – – – – – Nontangled data races

Figure 9.4. Child processes that exhibit only nontangled data races

child1and2()
{
    for (...) {
        s_lock (&lock1);
        work on shared data;
        s_unlock (&lock1);
        work on shared data;
    }
}

child3and4()
{
    for (...) {
        s_lock (&lock2);
        work on shared data;
        s_unlock (&lock2);
        work on shared data;
    }
}

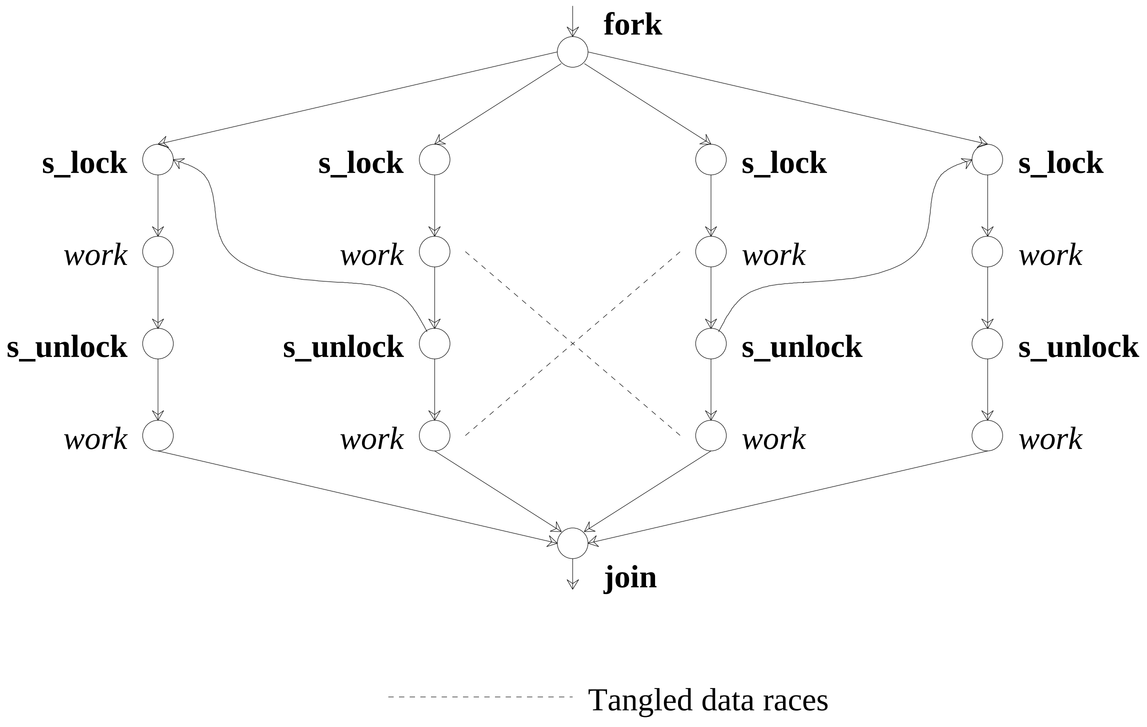


Figure 9.5. Child processes that exhibit tangled data races

Figure 9.4 illustrates two examples why the unmodified programs never exhibited tangled races. After forking children to perform work, one program (*tycho*) performed no synchronization within the children, resulting in only a single computation event in each child (Figure 9.4(a)). When races occurred between children, they involved only a single pair of computation events, which can never be tangled. Although the other programs do perform synchronization within the children, it is often on the same synchronization variable, resulting in an edge between their nodes (Figure 9.4(b)). The children iterate while executing lock/unlock pairs on common spin locks (or barriers) to implement mutual exclusion. When a synchronization edge is added from iteration i in one child to iteration j in the other, all events in iterations prior to i (in one child) become ordered with all events in iterations subsequent to j (in the other child). These orderings effectively prevent tangles from ever occurring.

Figure 9.5 illustrates the more complex synchronization patterns exhibited by the modified programs. In *workq*, groups of children cooperate in pairs to perform work on common shared variables. Because different groups of children (erroneously) perform synchronization on different locks, no synchronization edges are added between their synchronization nodes. Since the off-by-one errors introduced into shared-array subscripts caused many events to have data conflicts, tangles could then result. The program *parq* exhibited similar synchronization patterns, but for a different reason. In *parq*, children perform synchronization on the same lock, but because we randomly removed a lock operation, one child performed only unlock operations. Incoming synchronization edges which could order events were therefore never present in this child.

In summary, only nontangled races occur when child processes either perform no synchronization or cooperate and synchronize only with each other. Tangled races occur when children perform unrelated tasks and synchronize on different variables.

The absence of tangled races in any of the unmodified programs leads us to believe that validation and ordering will provide useful debugging tools in practice. Although diagnostic precision can become poor when tangled races exist, useful information can sometimes still be obtained from validation (discussed next). Additional investigation of tangled data races and how they may be untangled is left to future work.

9.3.4. Debugging with Validation and Ordering

The last part of our experiments reflects the ultimate goal of accurate race detection: debugging the cause of the data races. As discussed in Chapter 2, debugging requires first locating those races that are direct manifestations of bugs and then understanding how the program allowed them to occur. Below we discuss how the results of validation and ordering can be used to locate and understand these races.

Debugging data races first requires locating non-artifact races. When each first partition contains only one race, the race is guaranteed to be non-artifact. However, because a first partition can be large, analysis may not always precisely locate non-artifact races. When a first partition contains several races, ordering provides insufficient information to determine which are not artifacts (although at least one non-artifact is guaranteed to exist). In such a case, validation can provide additional information about the races contained within. If the partition contains a feasible race, we can still attempt to reason about the execution and understand how the race occurred. Feasibility guarantees that the race involves events that had the potential of executing concurrently, a necessary condition for a non-artifact. Even if the race is an artifact, it is only due to the racing events accessing common shared locations because of inconsistent data caused by a previous race. Because the race is feasible, synchronization errors in the program may be discovered, and some knowledge about other races in the partition may be gained. If all races in the partition are tangled, we still may be able to gain information by analyzing the extent of the tangle.

In each the unmodified test programs, the first partitions contained few races, and investigating these races lead directly to the bugs in the programs. However, in the modified programs, when the first partitions contained many races, not all of them could be investigated. We found that validation provided useful information in this case. Investigating the first races that were successfully validated lead to the synchronization bug in both programs, even though some of these races were artifacts. Although this approach is not guaranteed to always provide useful information, it did appear to be helpful in this case. Because these races were sometimes artifacts, we were not always able to locate the off-by-one subscript errors by examining them.

Once races have been identified, the program must then be analyzed to understand how they occurred. Investigation of the unmodified programs made it clear that more information than simply the location of non-artifact races is required. For example, for all of the unmodified programs, each first partition contained only one or two races. Even when presented with such precise information it was difficult to understand how the program could have allowed the race to occur. At a minimum, examining the execution history prior to the race was necessary. By modifying the post-mortem analyzer to output the temporal ordering graph, we were able to understand the history sufficiently to understand the bugs causing the race. In practice, however, we feel that race debugging requires both accurate race detection (to locate program points where debugging should begin) and the facilities of a parallel program debugger.

9.4. Summary

The experiments presented in this chapter suggest that validation and ordering can provide useful tools for debugging data races. The potentially large number of apparent data races can be refined to provide a starting point for debugging. When the complete temporal ordering was recorded, a close approximation of the actual dependences could be computed, allowing validation and ordering to pinpoint non-artifact races with high precision and completeness. In practice, however, the cost of recording the complete ordering may be unacceptable, requiring instead the recording of the approximate ordering. In this case, only a coarser approximation to the dependences can be attained. When no tangled races exist, we found that even this coarse approximation allowed non-artifacts to be precisely located. When tangled races occurred, the first partitions became large, but validation sometimes provided useful information for discovering synchronization errors. Clearly, more temporal information yields more accurate results. A practical implementation should record as much ordering information as possible. For programs in which a global clock can be efficiently maintained, the complete ordering can be recorded. In other programs, a subset of the complete ordering in which local clocks are occasionally synchronized might be possible. In the worst case, when only the approximate ordering can be recorded, non-artifacts can still be pinpointed when tangled races do not occur.

Even though validation and ordering were able to pinpoint non-artifact races, the bugs in our test programs were often too subtle to be understood from the location of the races alone. Given the location of a non-artifact race, the programmer must reason about the execution to understand how the race was allowed to occur; accurate race detection solves only part of the debugging problem. At the very least, understanding the nature of a race requires browsing the execution history prior to the race. Ideally, the facilities of a parallel program debugger would be desirable for this task. We conclude that effective race debugging can only be accomplished by integrating accurate race detection and parallel program debugging into a single tool.

Chapter 10

CONCLUSIONS AND FUTURE WORK

10.1. Conclusions

This thesis addresses the problem of dynamic race condition detection from the bottom up. We address both theoretical and practical issues, providing a model for reasoning about race conditions and techniques for detecting races that are direct manifestations of bugs. We also experimented with these techniques on a collection of parallel programs and found that they help to accurately pinpoint these races. Overall, the goal of this research was to provide formally based results and techniques to support race condition detection and debugging. To put the contributions of this research into perspective, Figure 10.1 shows one possible classification of the spectrum of knowledge about race condition detection, indicating the areas that have received attention by us and by previous work. We conclude by summarizing our work in these areas.

To reason about race conditions, we develop a model that characterizes actual, observed, and potential behaviors of the program. Characterizing these behaviors is necessary for understanding race conditions. This model is an advancement over previous work, which has only characterized race conditions intuitively or not at all. We then use the model to characterize two different types of race conditions, *general races* and *data races*, and prove results about the complexity of detecting them. In the past, the distinction between these races has not been explored. This distinction is important because these races appear in different classes of parallel programs and require different detection techniques. We also prove new results about the complexity of race detection. We prove that, for programs using synchronization powerful enough to implement mutual exclusion, detecting all general races and data races is an NP-hard problem. However, for debugging, we argue that it is useful to know whether an execution is free from actual data races, which can be efficiently determined. No such analogue exists for general races. We also prove that, for programs using weaker synchronization, all general races can be efficiently located. We prove this result by presenting the first efficient algorithm that computes ordering relations (necessary for general race detection) for such synchronization.

To address the issue of accurate race detection, we use the model to characterize which races are of interest for debugging (the feasible, non-artifact races) and contrast them with those reported by simple race detection schemes (the apparent races). Feasible, non-artifact races are direct manifestations of bugs. We then focus on techniques for accurate detection of data races. We present results showing how two techniques, *validation* and *ordering*, can prove that some apparent data races are feasible and not artifacts of other races. Validation involves first adding edges to the temporal ordering graph to represent shared-data or event-control dependences and then analyzing the resulting event orderings. Ordering involves partially ordering groups of data races to show how they affect each other. We also develop algorithms for performing validation and ordering in a post-mortem fashion. To test the effectiveness of these techniques and explore how they can be used for debugging, we analyzed a collection of

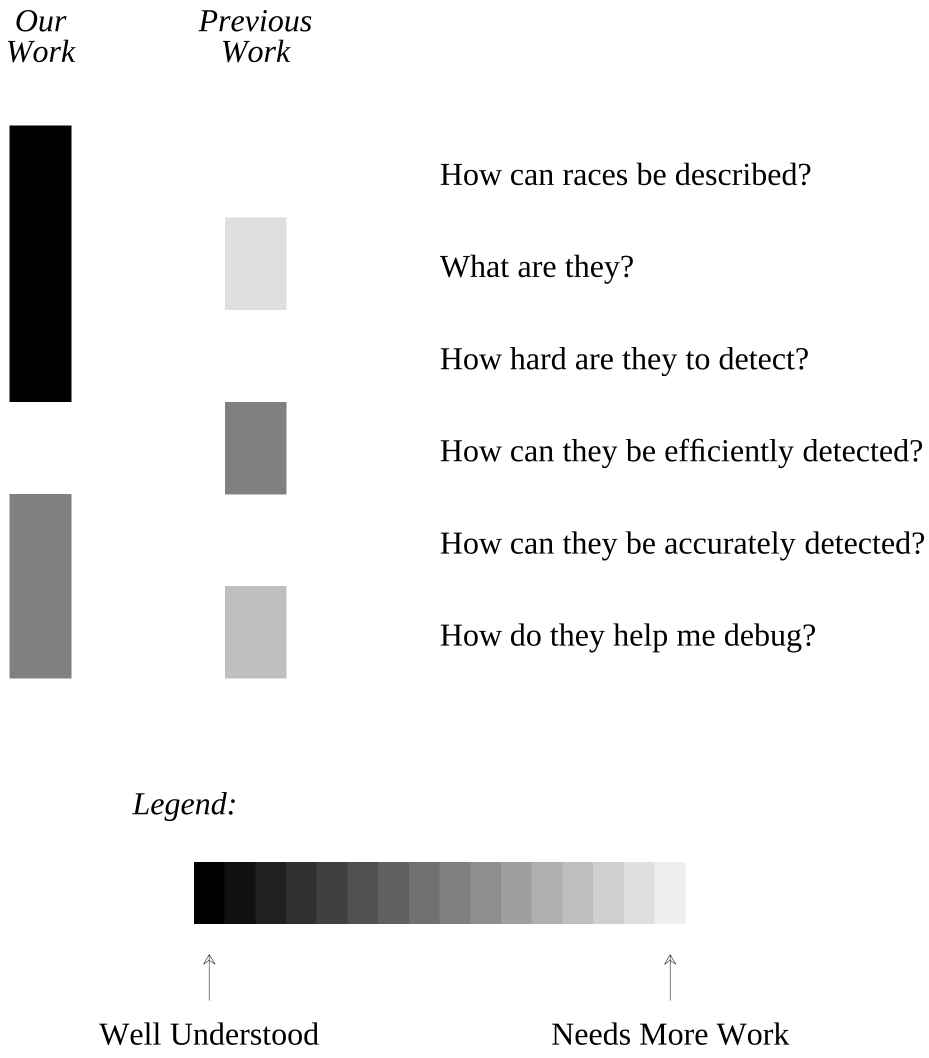


Figure 10.1. Spectrum of knowledge about race condition detection

test programs. Some of these programs were previously debugged and thought to be data-race free; others were modified by us to exhibit races. We found that the previously developed programs exhibited simple patterns of data races, which validation and ordering were able to precisely pinpoint. The programs that we modified exhibited more complex patterns, which were less accurately located.

The final result of this work is both a theory in which to reason about current and future race detection methods, and simple, effective techniques for locating races that are direct manifestations of bugs. The locations of these races provide a starting point for program debugging.

10.2. Future Research

Many issues remain unexplored by this thesis. Further research includes both short-term work that refines and extends our results, and long-term work that explores the practical applications of this research to the problem of parallel program debugging. Below we list and discuss these issues.

10.2.1. Short-Term Work

Short-term work includes extending our results to general races, refining some of our results for data races, and reducing the overhead of post-mortem analysis.

One main area of future work involves general races. Issues include developing an efficient algorithm for detecting apparent general races in executions using weaker synchronization, and extending the results of accurate data race detection to general races. No efficient algorithm currently exists for general race detection in executions that use synchronization incapable of implementing mutual exclusion (but more powerful than only **fork**/**join**). In Chapter 5 we presented the first efficient algorithm (Algorithm 5.1) deciding ordering relations for such synchronization. Preliminary investigation shows that this algorithm can be extended to efficiently detect apparent general races for weaker synchronization. Once apparent general races can be detected, the results of Chapter 8 might be easily extended to order them. Extending these results is important because accuracy may be more critical for debugging general races than data races. Data races that are not actual races introduce no unexpected results in the execution (no critical sections fail), but *any* general race indicates a chance that the racing memory accesses were not issued in the intended order.

Other areas of future work involve data race validation and ordering. The most important validation issue concerns unifying the use of shared-data and event-control dependences. Currently, event-control dependences are used to validate only tangled data races; sometimes G_E is unable to validate races that are not tangled. For example, an apparent race $\langle a,b \rangle$ is nontangled if no path exists in G_D from $\mathbf{a_f}$ to $\mathbf{b_s}$ (or from $\mathbf{b_f}$ to $\mathbf{a_s}$). Such a race may not be validated using G_E if a path exists from $\mathbf{a_f}$ to $\mathbf{b_f}$ (because a successor of *a* event-controls *b*), even though no path exists from $\mathbf{a_f}$ to $\mathbf{b_s}$. A single validation scheme based only on event-control dependences would be desirable.

For data race ordering, utilizing information about the non-transitivity of $\xrightarrow{\hat{R}}$ may improve accuracy. Preliminary research shows that a race in a first partition that does not affect some other race in the same partition can sometimes be safely removed. However, it is unclear in what order races should be removed or even how to efficiently determine which races are eligible.

Another ordering issue involves speculating on whether non-artifact races might be hidden inside non-first partitions. Although ordering seems to report small first partitions, non-artifact races may still exist that are not reported, causing some program bugs to be hidden. In Chapter 9, we found that in the program *workq* only one first race was reported but two race-causing bugs existed (see Chapter 9). A possible approach is to report ‘‘second’’ (and subsequent) partitions, allowing the programmer to determine if they contain races of interest.

Other areas of work pertain to reducing the overhead of post-mortem analysis. A criticism of post-mortem detection is that the potentially large trace files must be examined in their entirety, resulting in lengthy analysis. Improvements in our algorithms for data race detection and ordering might address this criticism. First, we may be able to use information about race ordering to avoid scanning the entire trace. If ordering can be performed during apparent race detection, portions of the trace that cannot possibly contain first races need not be analyzed. Second, to reduce the space overhead of post-mortem analysis, alternatives to using timestamps should be investigated. Work on the incremental maintenance of ordering graphs[31, 32], which avoids timestamps, might provide some insights. Third, to reduce the time overhead, the analysis itself might be parallelized. One obvious opportunity is to parallelize the detection of apparent races, but validation and ordering would also benefit.

10.2.2. Long-Term Work

The short-term issues described above concern immediate refinements of accurate race condition detection. However, long-term work involves applying race detection to its ultimate purpose: locating and fixing race-causing bugs. As discussed in Chapter 9, we believe that debugging races requires more than just being directed to the source lines involved in the race; some mechanism is required for analyzing the execution context of the race. One possibility is to simply summarize the recorded temporal ordering, letting the programmer reason about how the program could have exhibited the ordering (we took this approach when trying to understand the bugs in our test programs). However, in general, we feel that the facilities of a debugger are necessary to analyze the execution history more closely, for example, allowing the programmer to examine the values of variables, and set breakpoints and re-execute.

Two main problems must be addressed to successfully solve this debugging problem. First, the information required for accurate race detection must be efficiently recorded. Although naive tracing strategies such as the one used in our prototype race analyzer suffice for experimentation, they are not acceptable for practical use. Run-time overhead must be kept low to allow long-running executions to be analyzed. Second, race detection must be integrated into current debugging technology to allow debugging of programs that contain race conditions. For example, existing schemes for replaying shared-memory parallel programs either assume the program is race-free (resulting in incorrect re-execution when races exist), or record the order of essentially every shared-memory access (which conflicts with the solution to our first problem). Below we discuss some ideas regarding these two problems.

Program instrumentation records the information necessary for race detection, such as the temporal ordering and the *READ* and *WRITE* sets. As discussed in Chapter 2, recording the complete temporal ordering can introduce a central bottleneck into the execution. However, the complete temporal ordering allows the actual dependences to be closely approximated; only a coarser approximation can be computed when the approximate temporal ordering is recorded. In Chapter 9 we found that validation and ordering produce excellent results given the closer approximation but produce less accurate results (when tangled races exist) given the coarser approximation. These results suggest the temporal ordering should be recorded as completely possible, perhaps by using a hardware clock, software clocks that are occasionally synchronized, or a combination of these ideas.

Furthermore, naively recording the *READ* and *WRITE* sets can incur substantial overhead. We often observed the naive strategy used in our prototype increase execution times by one to three orders of magnitude, and occasionally produce trace files of over 100 megabytes. Reducing overhead primarily requires reducing the amount of trace data generated, and is essentially a data compression problem: the shared-memory locations accessed by an event must be efficiently summarized. One approach is to write a compressed trace by using simple data compression schemes or methods for summarizing regular access patterns[7, 8, 12]. Another approach is to employ two passes. The first pass would record an incomplete, coarse trace containing only enough information to replay a portion of the execution[43, 44, 50]. The second pass would then re-execute parts of the execution to record the more detailed *READ* and *WRITE* sets. This approach has the advantage that overhead is incurred only for parts of the execution for which the programmer desires race detection (e.g., detection in parts after a first race may not be desired). A different approach might be to use a hybrid post-mortem/on-the-fly scheme. On-the-fly schemes have the advantage that data conflicts between events can be located during execution without producing traces. Such a scheme might be helpful to pinpoint the source lines involved in a race while only compressed *READ* and *WRITE* sets (which would lose this information) are being traced.

The second problem that must be addressed for successful debugging is integrating the race detector and the debugger. Previous work on the efficient debugging of shared-memory parallel programs has not considered race detection in depth. Program replay is one example of an area where race detection and debugging can cooperate. Naive replay schemes simply trace the value of every shared-memory access and restore that value during re-execution[59]. More efficient schemes record and reproduce only the temporal ordering[15, 44], but these schemes fail in general to produce correct replay for executions containing actual data races. Since supporting replay requires run-time information similar to that needed for race detection, integrating race detection and replay might allow re-execution of programs that contain race conditions, without incurring the overhead of tracing every shared-memory access.

Another area where race detection and debugging might be integrated is the use of data and control dependences to understand the program behavior. For example, debugging using *flowback analysis* attempts to follow data and control dependences backwards through the execution to understand the causal relationship between events[16, 50]. Such an analysis uses static information obtained by semantic analysis of the program to first locate potential dependences, and dynamic information obtained by coarse execution traces to locate actual dependences. Similarly, accurate race detection uses dynamic information (the *READ* and *WRITE* sets) to approximate actual dependences to understand how events affect each other. Because of the similarity in the type of information collected and the type of analyses performed, these two techniques might be profitably unified.

It is clear that debugging shared-memory parallel programs containing race conditions is a difficult task. The results for understanding race conditions, and the techniques for accurate race detection, that we have presented in this thesis provide part of the solution. However, more research and experience with the debugging of parallel programs that contain race conditions is required before a truly practical tool can be developed.

A GLOSSARY OF ARROWS

This glossary contains brief explanations of notation defined in this thesis. We use labeled arrows (e.g., $\xrightarrow{T}$ and $\xrightarrow{D}$) to indicate relations that describe the actual program execution, hatted arrows (e.g., $\xrightarrow{\hat{T}}$ and $\xrightarrow{\hat{D}}$) to indicate relations that describe the approximate program execution, primed arrows (e.g., $\xrightarrow{T'}$ and $\xrightarrow{D'}$) to indicate relations that describe a feasible program execution, and arrows subscripted with ‘‘S’’ (e.g., $\xrightarrow{T_S}$ and $\xrightarrow{D_S}$) to indicate relations that describe a single-access program execution. Section numbers in parentheses indicate where the term is defined.

$P = \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$ *Actual Program Execution* (§3.1)
Precise information about the execution.

$\hat{P} = \langle E, \xrightarrow{\hat{T}}, \xrightarrow{\hat{D}} \rangle$ *Approximate Program Execution* (§3.2)
Execution information based on recorded trace data.

$P' = \langle E, \xrightarrow{T'}, \xrightarrow{D'} \rangle$ *Feasible Program Execution* (§3.3)
An execution that had the potential of occurring.

$P_S = \langle E_S, \xrightarrow{T_S}, \xrightarrow{D_S} \rangle$ *Single-access Program Execution* (§3.1)
A view of the execution in which each event represents at most one shared-memory access.

$\xrightarrow{T}$ *Temporal ordering relation* (§3.1).
$a \xrightarrow{T} b$ means that a precedes b; $a \overset{T}{\not\longleftrightarrow} b$ means that a and b execute concurrently.

$\xrightarrow{\hat{T}}$ *Approximate temporal ordering relation* (§3.2).
$a \xrightarrow{\hat{T}} b$ means that a was recorded as preceding b; $a \overset{\hat{T}}{\not\longleftrightarrow} b$ means the relative execution order was not recorded, because explicit synchronization did not prevent a and b from executing concurrently.

$\xrightarrow{T'}$ *Feasible temporal ordering relation* (§3.3).
A temporal ordering that the execution could have exhibited.

$\xrightarrow{T_S}$ *Single-access temporal ordering relation* (§3.1).
The temporal ordering among the individual shared-memory accesses.

$\xrightarrow{D}$ *Shared-data dependence relation* (§3.1).

$a \xrightarrow{D} b$ means that a accessed a shared variable that b later accessed (which at least one modified).

$\xrightarrow{\hat{D}}$ *Approximate shared-data dependence relation* (§3.2).

Conservative approximation to $\xrightarrow{D}$ ($\xrightarrow{\hat{D}} \supseteq \xrightarrow{D}$), based on the recorded temporal ordering and *READ* and *WRITE* sets.

$\xrightarrow{D'}$ *Feasible shared-data dependence relation* (§3.3).

Shared-data dependences that the execution could have exhibited.

$\xrightarrow{D_S}$ *Single-access shared-data dependence relation* (§3.1).

The shared-data dependences between the individual shared-memory accesses.

$\xrightarrow{E}$ *Event-control dependence relation* (§3.3).

$a \xrightarrow{E} b$ means that a affects the outcome of b (b may have changed had a occurred differently).

$\xrightarrow{\hat{E}}$ *Approximate event-control dependence relation* (§3.3).

Conservative approximation to $\xrightarrow{E}$ ($\xrightarrow{\hat{E}} \supseteq \xrightarrow{E}$), based on a refinement of $\xrightarrow{\hat{D}}$ and perhaps information obtained by static analysis.

$\xrightarrow{R}$ *Data-race ordering relation* (§8.1).

$\langle a,b\rangle \xrightarrow{R} \langle c,d\rangle$ means that the race $\langle c,d\rangle$ may have been an artifact of the race $\langle a,b\rangle$.

$\xrightarrow{\hat{R}}$ *Approximate data-race ordering relation* (§8.1).

Conservative approximation to $\xrightarrow{R}$ ($\xrightarrow{\hat{R}} \supseteq \xrightarrow{R}$), based on $\xrightarrow{\hat{E}}$.

$\xrightarrow{\hat{R}*}$ *Transitive approximate data-race ordering relation* (§8.1).

$\xrightarrow{\hat{R}*} = (\xrightarrow{\hat{R}})^*$, the transitive closure of $\xrightarrow{\hat{R}}$.

$\xrightarrow{R_S}$ *Single-access data-race ordering relation* (§8.1).

Data-race ordering on single-access data races; $\xrightarrow{R_S}$ is a partial order.

REFERENCES

[1] Adve, Sarita V., Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer, ''Detecting Data Races on Weak Memory Systems,'' *18th Int'l Symp. on Computer Architecture*, pp. 234-243 Toronto, Ontario, (May 1991).

[2] Allen, Todd R. and David A. Padua, ''Debugging Fortran on a Shared Memory Machine,'' *1987 Intl. Conf. on Parallel Processing*, pp. 721-727 St. Charles, IL, (August 1987).

[3] Appelbe, William F. and Charles E. McDowell, ''Anomaly Reporting — A Tool for Debugging and Developing Parallel Numerical Algorithms (extended abstract),'' *1st Intl. Conf. on Supercomputing Systems*, pp. 386-391 (1985).

[4] Appelbe, William F. and Charles E. McDowell, ''Integrating Tools for Debugging and Developing Multitasking Programs,'' *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 78-88 Madison, WI, (May 1988). Also appears in *SIGPLAN Notices* **24**(1) (January 1989).

[5] Apt, Krzysztof R., ''A Static Analysis of CSP Programs,'' *Workshop on Logics of Programs*, pp. 1-17 (June 1983).

[6] Balasundaram, Vasanth and Ken Kennedy, ''Compile-time Detection of Race Conditions in a Parallel Program,'' *3rd Intl. Conf. on Supercomputing*, pp. 175-185 Crete, Greece, (June 1989).

[7] Balasundaram, Vasanth and Ken Kennedy, ''A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations,'' *SIGPLAN '89 Conf. on Programming Language Design and Implementation*, pp. 41-53 Portland, OR, (1989).

[8] Balasundaram, Vasanth, ''Interactive Parallelization of Numerical Scientific Programs,'' *Ph.D. Thesis; also Rice Univ. Computer Science Dept. Tech. Rep. TR89-95*, (July 1989).

[9] Bernstein, A. J., ''Analysis of Programs for Parallel Processing,'' *IEEE Trans. on Electronic Computers* **EC-15**(5) pp. 757-763 (October 1966).

[10] Bristow, Guy, ''The Static Detection of Synchronization Anomalies in HAL/S Programs,'' *Ph.D. Thesis, also available as Comp Sci Tech Rep CU-CS-165-79*, Univ. Colorado at Boulder, (1979).

[11] Bristow, G., C. Drey, B. Edwards, and W. Riddle, ''Anomaly Detection in Concurrent Programs,'' *4th Intl. Conf. on Software Engineering*, pp. 265-273 (1979).

[12] Callahan, David, ''A Global Approach to Parallelism Detection,'' *Ph.D. Thesis*, Rice University, (July 1986).

[13] Callahan, David and Jaspal Subhlok, ''Static Analysis of Low-Level Synchronization,'' *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 100-111 Madison, WI, (May 1988). Also appears in *SIGPLAN Notices* **24**(1) (January 1989).

[14] Callahan, David, Ken Kennedy, and Jaspal Subhlok, ''Analysis of Event Synchronization in A Parallel Programming Tool,'' *2nd Symp. on Principle and Practice of Parallel Programming*, pp. 21-30 Seattle, WA, (March 1990).

[15] Carver, Richard H. and Kuo-Chung Tai, ‘‘Reproducible Testing of Concurrent Programs Based on Shared Variables,’’ *6th Intl. Conf. on Distributed Computing Systems*, pp. 428-432 Boston, MA, (May 1986).

[16] Choi, Jong-Deok, Barton P. Miller, and Robert H. B. Netzer, ‘‘Techniques for Debugging Parallel Programs with Flowback Analysis,’’ *ACM Trans. on Programming Languages and Systems* **13**(4) pp. 491-530 (October 1991).

[17] Choi, Jong-Deok and Sang Lyul Min, ‘‘Race Frontier: Reproducing Data Races in Parallel Program Debugging,’’ *3rd ACM Symposium on Principles and Practice of Parallel Programming*, pp. 145-154 Williamsburg, VA, (April 1991).

[18] Dijkstra, E. W., ‘‘Solution of a Problem in Concurrent Programming Control,’’ *Communications of the ACM* **8**(9) p. 569 (September 1965).

[19] Dinning, Anne and Edith Schonberg, ‘‘The Task Recycling Technique for Detecting Access Anomalies On-The-Fly,’’ *IBM Tech. Rep. RC 15385*, (January 1990).

[20] Dinning, Anne and Edith Schonberg, ‘‘An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection,’’ *2nd ACM Symposium on Principles and Practice of Parallel Programming*, pp. 1-10 Seattle, WA, (March 1990).

[21] Dinning, Anne, ‘‘Detecting Nondeterminism in Shared Memory Parallel Programs,’’ *Ph.D. Thesis; also Dept. of Computer Science Tech. Rep. 526*, New York University, (November 1990).

[22] Dinning, Anne and Edith Schonberg, ‘‘Detecting Access Anomalies in Programs with Critical Sections,’’ *ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 85-96 Santa Cruz, CA, (May 1991).

[23] Emrath, Perry A. and David A. Padua, ‘‘Automatic Detection Of Nondeterminacy in Parallel Programs,’’ *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 89-99 Madison, WI, (May 1988). Also appears in *SIGPLAN Notices* **24**(1) (January 1989).

[24] Emrath, Perry A., Sanjoy Ghosh, and David A. Padua, ‘‘Event Synchronization Analysis for Debugging Parallel Programs,’’ *Supercomputing ’89*, pp. 580-588 Reno, NV, (November 1989).

[25] Emrath, Perry A., Sanjoy Ghosh, and David A. Padua, ‘‘Detecting Non-Determinacy in Parallel Programs,’’ *IEEE Software* **9**(1) pp. 69-77 (January 1992).

[26] Ferrante, J., Karl J. Ottenstein, and Joe D. Warren, ‘‘The Program Dependence Graph and its Use in Optimization,’’ *ACM Trans. on Programming Languages and Systems* **9**(3) pp. 319-349 (1987).

[27] Fidge, C. J., ‘‘Partial Orders for Parallel Debugging,’’ *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 183-194 Madison, WI, (May 1988). Also appears in *SIGPLAN Notices* **24**(1) (January 1989).

[28] Fowler, Jerry and Willy Zwaenepoel, ‘‘Causal Distributed Breakpoints,’’ *10th Intl. Conf. on Distributed Computing Systems*, pp. 134-141 Paris, France, (June 1990).

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

[30] Gottlieb, Allan, B. D. Lubachevsky, and Larry Rudolph, ‘‘Basic Techniques for the Efficient Coorindation of Very Large Numbers of Cooperating Sequential Processors,’’ *ACM Trans. on Programming Languages and Systems* **5**(2) pp. 164-189 (April 1983).

[31] Griswold, Victor Jon, ‘‘Determining Interior Vertices of Graph Intervals,’’ *Dept. of Computer Science Tech. Rep. WUCS-90-40,* Washington Univ, (December 1990).

[32] Griswold, Victor Jon, ‘‘Core Algorithms for Autonomous Monitoring of Distributed Systems,’’ *ACM/ONR Workshop on Parallel and Distributed Debugging,* pp. 36-45 Santa Cruz, CA, (May 1991).

[33] Haban, Dieter and Wolfgang Weigel, ‘‘Global Events and Global Breakpoints in Distributed Systems,’’ *21st Hawaii Int’l Conf. on System Sciences, Vol II,* pp. 166-175 (1989).

[34] Habermann, A. Nico, ‘‘Synchronization of Communicating Processes,’’ *Communications of the ACM* **12**(3) pp. 171-176 (March 1972).

[35] Helmbold, David P., Charles E. McDowell, and Jian-Zhong Wang, ‘‘Analyzing Traces with Anonymous Synchronization,’’ *1990 Intl. Conf. on Parallel Processing,* pp. 70-77 St. Charles, IL, (August 1990).

[36] Herzog, Otthein, ‘‘Static Analysis of Concurrent Processes for Dynamic Properties Using Petri Nets,’’ *Intl. Symp. on Semantics of Concurrent Computation,* pp. 66-90 (July 2-4, 1979).

[37] Hood, Robert, Ken Kennedy, and John Mellor-Crummey, ‘‘Parallel Program Debugging with On-the-fly Anomaly Detection,’’ *Supercomputing ’90,* pp. 74-81 New York, NY, (November 1990).

[38] Kuck, D. J., R. H. Kuhn, B. Leasure, D. A. Padua, and M. Wolfe, ‘‘Dependence Graphs and Compiler Optimizations,’’ *8th ACM Symp. on Principles of Programming Languages,* pp. 207-218 Williamsburg, VA, (January 1981).

[39] Lamport, Leslie, ‘‘How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs,’’ *IEEE Trans. on Computers* **C-28**(9) pp. 690-691 (September 1979).

[40] Lamport, Leslie, ‘‘Interprocess Communication,’’ *SRI Technical Report*, (March 1985).

[41] Lamport, Leslie, ‘‘The Mutual Exclusion Problem: Part I — A Theory of Interprocess Communication,’’ *Journal of the ACM* **33**(2) pp. 313-326 (April 1986).

[42] Lamport, Leslie, ‘‘On Interprocess Communication, Part I: Basic Formalism,’’ *Distributed Computing,* pp. 77-85 (1986).

[43] Larus, James R., ‘‘Abstract Execution: A Technique for Efficiently Tracing Programs,’’ *Software — Practice and Experience* **20**(12) pp. 1241-1258 (December 1990).

[44] LeBlanc, Thomas J. and John M. Mellor-Crummey, ‘‘Debugging Parallel Programs with Instant Replay,’’ *IEEE Trans. on Computers* **C-36**(4) pp. 471-482 (April 1987).

[45] Mattern, F., ‘‘Virtual Time and Global States of Distributed Systems,’’ pp. 215-226 in *Parallel and Distributed Algorithms*, ed. Michel Cosnard, North Holland (1989).

[46] McDowell, Charles E. and William F. Appelbe, ‘‘Minimizing the Complexity of Static Analysis of Parallel Programs,’’ *20th Annual Hawaii Intl. Conf. on System Sciences*, pp. 171-176 (1987).

[47] McDowell, Charles E. and David P. Helmbold, ‘‘Computing Reachable States of Parallel Programs,’’ *ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, CA, (May 1991).

[48] Mellor-Crummey, John M., ‘‘Debugging and Analysis of Large-Scale Parallel Programs,’’ *Ph.D. Thesis, also available as Computer Science Dept. Tech. Rep. 312*, Univ. of Rochester, (September 1989).

[49] Mellor-Crummey, John M., ‘‘On-the-Fly Detection of Data Races for Programs with Nested Fork-Join Parallelism,’’ *Supercomputing ’91*, pp. 24-33 Albuquerque, NM, (November 1991).

[50] Miller, Barton P. and Jong-Deok Choi, ‘‘A Mechanism for Efficient Debugging of Parallel Programs,’’ *SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 135-144 Atlanta, GA, (June 1988). Also appears in *SIGPLAN Notices* **23**(7) (July 1988).

[51] Min, Sang Lyul and Jong-Deok Choi, ‘‘An Efficient Cache-based Access Anomaly Detection Scheme,’’ *4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 235-244 Palo Alto, CA, (April 1991).

[52] Morgan, E. Timothy and Rami R. Razouk, ‘‘Interactive State-Space Analysis of Concurrent Systems,’’ *IEEE Trans on Software Engineering* **SE-13**(10) pp. 1080-1091 (October, 1987).

[53] Netzer, Robert H. B. and Barton P. Miller, ‘‘On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions,’’ *1990 Intl. Conf. on Parallel Processing*, pp. II-93–II-97 St. Charles, IL, (August 1990).

[54] Netzer, Robert H. B. and Barton P. Miller, ‘‘Improving the Accuracy of Data Race Detection,’’ *3rd ACM Symposium on Principles and Practice of Parallel Programming*, pp. 133-144 Williamsburg, VA, (April 1991).

[55] Netzer, Robert H. B. and Barton P. Miller, ‘‘Detecting Data Races in Parallel Program Executions,’’ pp. 109-129 in *Advances in Languages and Compilers for Parallel Processing*, ed. A. Nicolau, D. Gelernter, T. Gross, and D. Padua, MIT Press (1991).

[56] Nudler, Itzhak and Larry Rudolph, ‘‘Tools for the Efficient Development of Efficient Parallel Programs,’’ *1st Israeli Conf. on Computer System Engineering*, (1988).

[57] Olender, Kurt M. and Leon J. Osterweil, ‘‘Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation,’’ *IEEE Trans. on Software Engineering* **16**(3) pp. 268-280 (March 1990).

[58] Osterweil, Leon, ‘‘Integrating the Testing, Analysis and Debugging of Programs,’’ *Software Validation*, pp. 73-93 Elsevier Science Publishers, (1984).

[59] Pan, Douglas Z. and Mark A. Linton, ‘‘Supporting Reverse Execution of Parallel Programs,’’ *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 124-129 Madison, WI, (May 1988). Also appears in *SIGPLAN Notices* **24**(1) (January 1989).

[60] Podgurski, Andy and Lori 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 (September 1990).

[61] Podgurski, H. Andy, ‘‘The Significance of Program Dependences for Software Testing, Debugging, and Maintenance,’’ *Ph.D. Thesis*, Univ. of Massachusetts, (September 1989).

[62] Reif, John H., ‘‘Data Flow Analysis of Communicating Processes,’’ *6th ACM Symp. on Principles of Programming Languages*, pp. 257-268 (1979).

[63] Schatz, Emmi and Barbara G. Ryder, ‘‘Directed Tracing to Detect Race Conditions,’’ *LCSR Tech. Rep. 155*, Rutgers Univ., (October 1990).

[64] Schonberg, Edith, ‘‘On-The-Fly Detection of Access Anomalies,’’ *Ultracomputer Note #149*, (October 1988).

[65] Schonberg, Edith, ‘‘On-The-Fly Detection of Access Anomalies,’’ *SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 285-297 Portland, OR, (July 1989). Also appears in *SIGPLAN Notices* **24**(7) (July 1989).

[66] Sedgewick, Robert, *Algorithms*, Addison-Wesley, Reading, MA (1983).

[67] Shatz, S. M. and W. K. Cheng, ‘‘An Approach to Automated Static Analysis of Distributed Software,’’ *1st Intl. Conf. on Supercomputing Systems*, pp. 377-385 (1985).

[68] Silberschatz, Abraham, James L. Peterson, and Peter B. Galvin, *Operating System Concepts, 3rd ed.*, Addison-Wesley (1991).

[69] Steele, Guy L., ‘‘Making Asynchronous Parallelism Safe for the World,’’ *17th Annual ACM Symp. on Principles of Programming Languages*, pp. 218-231 San Francisco, CA, (January 1990).

[70] Taylor, Richard N. and Leon J. Osterweil, ‘‘A Facility for Verification, Testing, and Documentation of Concurrent Process Software,’’ *IEEE COMPSAC ’78*, pp. 36-41 (1978).

[71] Taylor, Richard N. and Leon J. Osterweil, ‘‘Anomaly Detection in Concurrent Software by Static Data Flow Analysis,’’ *IEEE Trans on Software Engineering* **SE-6**(3) pp. 265-277 (May 1980).

[72] Taylor, Richard N., ‘‘Static Analysis of the Synchronization Structure of Concurrent Programs,’’ *Ph.D. Thesis*, Univ. of Colorado at Boulder, (1980).

[73] Taylor, Richard N., ‘‘Complexity of Analyzing the Synchronization Structure of Concurrent Programs,’’ *Acta Informatica* **19** pp. 57-84 (1983).

[74] Taylor, Richard N., ‘‘A General-Purpose Algorithm for Analyzing Concurrent Programs,’’ *Communications of the ACM* **26**(5) pp. 362-376 (May 1983).

[75] Taylor, Richard N., ‘‘Analysis of Concurrent Software by Cooperative Application of Static and Dynamic Techniques,’’ *Software Validation*, pp. 127-137 Elsevier Science Publishers, (1984).

[76] Wolfe, Michael J., ‘‘Optimizing Supercompilers for Supercomputers,’’ *Ph.D. Thesis*, University of Illinois at Urbana-Champaign, (1982).

[77] Young, Michal and Richard M. Taylor, ‘‘Combining Static Concurrency Analysis with Symbolic Execution,’’ *IEEE Trans. on Software Engineering* **14**(10) pp. 1499-1511 (October, 1988).

TABLE OF CONTENTS

TABLE OF ALGORITHMS

TABLE OF FIGURES