What Are Race Conditions? Some Issues and Formalizations

ROBERT H. B. NETZER and BARTON P. MILLER
University of Wisconsin — Madison

In shared-memory parallel programs that use explicit synchronization, *race conditions* result when accesses to shared memory are not properly synchronized. Race conditions are often considered to be manifestations of bugs, since their presence can cause the program to behave unexpectedly. Unfortunately, there has been little agreement in the literature as to precisely what constitutes a race condition. Two different notions have been implicitly considered: one pertaining to programs intended to be deterministic (which we call *general races*) and the other to nondeterministic programs containing critical sections (which we call *data races*). However, the differences between general races and data races have not yet been recognized. This paper examines these differences by characterizing races using a formal model and exploring their properties. We show that two variations of each type of race exist: *feasible* general races and data races capture the intuitive notions desired for debugging and *apparent* races capture less accurate notions implicitly assumed by most dynamic race detection methods. We also show that locating feasible races is an NP-hard problem, implying that only the apparent races, which are approximations to feasible races, can be detected in practice. The complexity of dynamically locating apparent races depends on the type of synchronization used by the program. Apparent races can be exhaustively located efficiently only for weak types of synchronization that are incapable of implementing mutual exclusion. This result has important implications since we argue that debugging general races requires exhaustive race detection and is inherently harder than debugging data races (which requires only partial race detection) Programs containing data races can therefore be efficiently debugged by locating certain easily identifiable races. In contrast, programs containing general races require more complex debugging techniques.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures*; D.4.1 [**Operating Systems**]: Process Management—*concurrency*, *mutual exclusion*, *synchronization*

General Terms: Design, Languages

Additional Key Words and Phrases: Critical sections, data races, debugging, nondeterminacy, parallel programs, race conditions

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

Authors' address: Computer Sciences Department, University of Wisconsin-Madison, 1210 W. Dayton Street, Madison, WI 53706; e-mail: R. H. B. Netzer, Netzer@cs.wisc.edu; B. P. Miller, bart@cs.wisc.edu.

1. INTRODUCTION

In shared-memory parallel programs, if accesses to shared memory are not properly synchronized, time-dependent failures[1] called *race conditions* can result. Race conditions occur when different processes access shared data without explicit synchronization. Because races can cause the program to behave in ways unexpected by the programmer, detecting them is an important aspect of debugging. However, in the literature, there seems to be little agreement as to precisely what constitutes a race condition. Indeed, two different notions have been used, but the distinction between them has not been previously recognized. Because no consistent terminology has appeared, several terms have been used with different intended meanings, such as *access anomaly* [6–8, 12, 18], *data race* [1, 4, 5, 11 , 16, 20, 22], *critical race* [13], *harmful shared-memory access* [24], *race condition* [10, 26], or just *race* [2, 9, 17]. This paper explores the nature of race conditions and uncovers some previously hidden issues regarding the accuracy and complexity of dynamic race detection. We present the following results.

(1) Two fundamentally different types of races, that capture different kinds of bugs in different classes of parallel programs, can occur. *General races* cause nondeterministic execution and are failures in programs intended to be deterministic. *Data races* cause nonatomic execution of critical sections and are failures in (nondeterministic) programs that access and update shared data in critical sections.[2]

(2) To represent the sources of race conditions precisely, we formally characterize the intuitive notion of a race we wish to detect for debugging (which we call a *feasible* race). In contrast, we show that there is a simpler to detect but less accurate notion of a race that most previously proposed race detection methods locate (which we call an *apparent* race). Feasible races are based on the possible behavior of the program; apparent races, which are approximations to feasible races, are based on only the behavior of the program's explicit synchronization (and not the semantics of the program's computation).

(3) Exactly locating the feasible general races or data races is an NP-hard problem. This result implies that the apparent races, which are simpler to locate, must be detected for debugging in practice.

(4) Apparent races can be exhaustively located efficiently only for programs that use synchronization incapable of implementing mutual exclusion (such as **fork/join** or **Post/Wait** synchronization without **Clear** operations); detection is NP-hard for more powerful types of synchronization (such as semaphores).

[1] To be consistent with the fault tolerant research community [25], a *failure* occurs when a program's external behavior differs from its specification, and a fault is its algorithmic cause (although we use the term bug).

[2] There is some controversy over terminology that is the most descriptive. In place of data race and general race, atomicity race and determinacy race have also been suggested.

(5) Debugging race conditions in programs intended to be deterministic is inherently more difficult than in nondeterministic programs. Races that cause nonatomic execution of critical sections (data races) are "local" properties of the execution and can be detected directly from an execution trace. In contrast, races that cause nondeterministic execution (general races) are "global" properties of the program whose detection requires analyzing the entire execution to compute alternative event orderings possibly exhibited by the program.

These results provide an understanding of race conditions important for dynamic race detection. Previous work has not provided unambiguous characterizations of the different types of race conditions or related races to program bugs. For example, race conditions have only been defined as occurring when two shared-memory references "can potentially execute concurrently" [6] or have no "guaranteed run-time ordering" [10]. Our work is novel since we explicitly characterize two different types of race conditions using a formal model and explore their properties. The distinction between general races and data races is necessary because they are manifestations of different types of program bugs and require different detection techniques. Using a model has the advantage that issues regarding the accuracy and complexity of dynamic race detection then become clear. The accuracy issues have implications for debugging: accurate race detection means that races that are direct manifestations of program bugs are pinpointed, while less accurate detection can report spurious races that mislead a programmer. The complexity issues show which types of races allow exact and efficient detection and which can be only approximately located.

Our results show that locating exactly the desired races (the feasible races) is computationally intractable. Indeed, previously proposed race detection methods take an easier approach and locate only a subset of the apparent races. However, we argue that apparent races can often be spurious, and that effective debugging requires more sophisticated techniques. Race conditions can be debugged by attempting to determine which of these located (apparent) races are of interest for debugging (i.e., are feasible). Our results show that there is also a fundamental disparity between debugging race conditions in deterministic and nondeterministic programs. Nondeterministic programs that use critical sections can be safely debugged (to find data races) because we can easily determine if an execution is data-race free; when data races occur, the feasible races can be approximately located. However, debugging programs intended to be deterministic (to find general races) is inherently harder. We can be confident that execution was deterministic only if exhaustive race detection shows an absence of general races (and this is efficient only for programs using synchronization incapable of implementing mutual exclusion).

2. EXAMPLES

Explicit synchronization is often added to shared-memory parallel programs to coordinate accesses to shared data. Without proper coordination, different types of race conditions can result. To motivate these different types of races, we present an example of each. In subsequent sections we will characterize them in terms of a formal model and investigate their properties.

One purpose of adding explicit synchronization to shared-memory parallel programs is to implement *critical sections*, which are blocks of code intended to execute as if they were atomic. Atomic execution means that the final state of variables read and written in the section depends only upon their initial state at the start of the section and upon the operations performed by the code (and not operations performed by another process). *Bernstein's conditions* state that atomic execution is guaranteed if shared variables that are read and modified by the critical section are not modified by any other concurrently executing section of code [3]. A violation of these conditions has typically been called a *data race* [1, 4, 5, 11, 16, 17, 20, 22] or *access anomaly* [6–8, 18]. We prefer the term *data race.*

Figure 1 shows an example program for which a data race is considered a failure. This program processes commands from bank tellers that make deposits and withdrawals for a given bank account. Figure 1a shows a correct version of the program. Since the variables balance and interest are shared, operations that manipulate them are enclosed in critical sections. Because critical sections can never execute concurrently, this version will exhibit no data races. Figure 1b shows an erroneous version that can exhibit data races; the P and V operations that enforced mutual exclusion are missing. The deposit and withdraw code can therefore execute concurrently, causing their individual statements to effectively interleave, possibly violating the atomicity of one of the intended critical sections. The data races in this program are considered failures because the intent was that the deposit and withdrawal code execute atomically, without interference from other processes.

Even though programs like the one in Figure 1 may contain critical sections, they are often intended to be nondeterministic (e.g., 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. For a given input, all executions of such programs always produce the same result, regardless of any 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 been called a *race condition* [9, 10, 26], *access anomaly* [12], *critical race* [13], or *harmful shared-memory access* [24]. For a more consistent terminology, we propose the term *general race*, since such a race is more general than a data race.

As an example of programs for which general races are manifestations of bugs, consider parallel programs that are constructed from sequential programs by parallelizing loops. The sequential version of a program behaves

```
            Process 1                                 Process 2

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

(a): no-data-race version

```
            Process 1                                 Process 2

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

(b): data-race version

Fig. 1. (a) C program fragment manipulating bank account. (b) Erroneous version exhibiting data races.

deterministically, producing a particular result for any given input. Typically, the parallelized version is intended to have the same semantics. 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. 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).

Both general races and data races are notions that are necessary for debugging. Although they both occur when shared-memory accesses occur in an incorrect or unexpected order, they are manifestations of different types of bugs that occur in different classes of parallel programs.

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 exhibit no data races but still be nondeterministic. General races alone will not suffice, since general races that are not data races are not always failures. The preceding examples illustrate these cases. For example, the no-data-race version of Figure 1 executes nondeterministically, depending on when commands are entered by the tellers. This version correctly exhibits no data races (its critical sections execute atomically), even though it is nondeterministic. However, even though general races occur, they are not considered to be manifestations of bugs.

General races and data races also 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.

Emrath and Padua have also characterized different types of race conditions but have only addressed programs intended to be deterministic [9]. They considered four levels of nondeterminism 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 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 using a formal framework.

3. FORMAL MODEL FOR REASONING ABOUT RACE CONDITIONS

Now that we have given examples illustrating different types of race conditions, we next discuss how they can be characterized using a formal model. Doing so not only provides unambiguous characterizations of each, but also provides a mechanism with which to reason about their properties. In this section, we briefly overview our model for reasoning about race conditions that was first presented in an earlier paper [22] and that is based on Lamport's theory of concurrent systems [15]. Our model consists of two parts: one to represent the actual behavior exhibited by the program and the other to represent potential behaviors possibly exhibited by the program.

3.1 Actual Program Executions

The first part of our model is simply a notation for representing an execution of a shared-memory parallel program on a sequentially consistent [14] processor.[3] A *program execution*, P, is a triple, $\langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$, where E is a finite set of *events*, and $\xrightarrow{T}$ (the *temporal ordering relation*) and $\xrightarrow{D}$ (the *shared-data dependence relation*) are relations[4] over those events. Intuitively, E represents the actions performed by the execution, $\xrightarrow{T}$ represents the order in which they are performed, and $\xrightarrow{D}$ shows how events affect one another through accesses to shared memory. We refer to P as an *actual program execution* when P represents an execution that the program at hand actually performed.

Each event $e \in E$ represents both the execution instance of a set of program statements and the sets of shared memory locations they read and write. A *synchronization event* represents an instance of some synchronization operation; a *computation event* represents the execution instance of any group of statements (belonging to the same process) that executed consecutively, none of which are synchronization operations. A *data conflict* exists between two events if one writes a shared memory location that the other reads or writes.

For two events, a and b, $a \xrightarrow{T} b$ means that a completes before b begins (in the sense that the last action of a can affect the first action of b), and $a \overset{T}{\nrightarrow} b$ means that a and b execute concurrently (i.e., neither completes before the other begins). We should emphasize that $\xrightarrow{T}$ is defined to describe the *actual* execution order between events in a particular execution; e.g., $a \overset{T}{\nrightarrow} b$ means that a and b actually execute concurrently; it does not mean that they could have executed in any order. A shared-data dependence $a \xrightarrow{D} b$ exists if a accesses a shared variable that b later accesses (where at least one access modifies the variable), or if a precedes b in the same process (since data can in general flow through nonshared variables local to the process). A dependence also exists if there is a chain of dependences from a to b; e.g., if a accesses a shared variable that another event, c, later accesses, and c then references a variable that b later references.

3.2 Feasible Program Executions

An actual program execution is a convenient notation for describing the behavior of a particular execution. However, to characterize race conditions, it is necessary to also describe behavior that the program *could have* exhibited. Most previous work has not explicitly considered this issue; race conditions have typically been defined only as data-conflicting accesses "that can execute in parallel" [6] or whose execution order is not "guaranteed" [10]. Such definitions implicitly refer to a set of alternative orderings that had the potential of occurring. Our work is novel in that we explicitly define these sets of orderings. The second part of our model characterizes sets of *feasible* program executions, which represent other executions that had the potential of occurring. We next discuss several possible ways in which these sets can be

[3] Sequential consistency ensures that shared-memory accesses behave as if they were all performed atomically and in some linear order. The model also contains axioms describing properties that any program execution must possess [22]. We omit these axioms here as they are unnecessary for simply characterizing race conditions.

[4] Superscripted arrows denote relations, and $a \nrightarrow b$ is a shorthand for $\neg(a \rightarrow b) \wedge \neg(b \rightarrow a)$.

defined. Instead of simply intuitively reasoning about alternative orderings, formally defining these sets uncovers issues important for debugging.

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 focus on program executions, $P' = \langle E', \xrightarrow{T'}, \xrightarrow{D'} \rangle$, that are *prefixes* of P and implicitly consider executions on the same input as P. P' is a prefix of P if each process in P' performs the same events as some initial part of the corresponding process in P. Focusing on prefixes allows us to pinpoint where nondeterminacy is introduced. If P' were not required to be a prefix of P, we could not in general determine where or if a and b occur in P'. When P' contains a different number of execution instances of some statement, we can not draw a correspondence between events in P and events in P' (because events are defined to represent the *execution instance* of one or more statements).

We define three sets of program execution prefixes by considering successively fewer restrictions on the different ways in which P's events could have been performed. We will see that these different sets characterize races with varying accuracy and complexity. The first two sets are restricted to contain only program executions that are feasible (i.e., that could have actually occurred); these sets characterize races most accurately. The first set, denoted F_{SAME}, contains all feasible executions that exhibit the same shared-data dependences as P; the second set, denoted F_{DIFF}, contains feasible executions with no restrictions on their shared-data dependences. F_{DIFF} includes all executions that perform a prefix of the events performed by P regardless of which shared-data dependences may result. F_{SAME} includes the executions that perform exactly the same events and exhibit the same shared-data dependences[5] as P.

Definition 3.1. 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 3.2. 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) $\xrightarrow{D'}$ represents any shared-data dependences that satisfy (1) and (2).

[5] The structure of this set 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.

The last set, denoted F_{SYNC}, also contains executions with arbitrary shared-data dependences, but they are no longer required to be feasible; they are only required to obey the semantics of the program's explicit synchronization. This simpler set characterizes races less accurately, but is a useful approximation to F_{DIFF} that involves only the semantics of explicit synchronization (and not the program). Moreover, since previous race detection methods analyze only explicit synchronization, F_{SYNC} is the set of alternative executions they implicitly assume.

Definition 3.3. F_{SYNC} is the set of program executions, $P' = \langle E', \overset{T'}{\rightarrow}, \overset{D'}{\rightarrow} \rangle$, such that

(1) $\overset{T'}{\rightarrow}$ obeys the semantics of the program's explicit synchronization,
(2) P' is a prefix of P, and
(3) $\overset{D'}{\rightarrow}$ represents any shared-data dependences that satisfy (1) and (2).

F_{SYNC} includes all executions that would have been possible had P not exhibited any shared-data dependences (or any dependences that affect control flow). Since programs in general access shared-memory, F_{SYNC} may include executions the program could never exhibit. For example, assume that one process in P assigns to a shared variable S the value 1, and then another process conditionally executes a procedure only if S equals 1. If no explicit synchronization forces the assignment to occur before the procedure call, then F_{SYNC} will contain an execution in which the procedure is called before S is assigned 1. However, since the procedure can be called only if S equals 1, such an execution is not feasible (assuming that S is not initialized to 1). In general, for another execution to perform the same events as P, it must also exhibit the same shared-data dependences as P; when general races occur (whether or not they are considered failures), F_{SYNC} may contain infeasible executions [19, 22]. As discussed later, the existence of such infeasible executions impacts the accuracy of races reported by methods that analyze only explicit synchronization. Nevertheless, we will see that this notion is useful because it allows a simple characterization of race conditions that are easy to detect.

4. ISSUES IN CHARACTERIZING RACE CONDITIONS

We now characterize general races and data races in terms of our model and explore some issue that arise. We show that two different types of each race exist. One type, the *feasible* race, captures the intuitive notion that we wish to express, but is NP-hard to locate exactly. The other type, the *apparent* race, captures a less accurate notion (assumed by most race detection methods) but can be more easily detected. Moreover, we argue that debugging programs intended to be deterministic (to find general races) requires exhaustive race detection and is inherently harder than debugging nondeterministic programs that use critical sections (to find data races), which requires only partial race detection.

4.1 General Races and Data Races

Intuitively, a general race potentially introduces nondeterminism and exists in a program execution P if two events a and b have data conflicts and their access order is not "guaranteed" by the execution's synchronization. A data race potentially causes the atomicity of critical sections to fail and exists if a and b either execute concurrently or have the potential of doing so. To explore the nature of races, we first formalize what it means to potentially execute concurrently (or in a different order) by using the different sets of program executions discussed above. For example, a general race exists if a and b occur in some feasible program execution in an order different than in P. Similarly, a data race exists if some feasible program execution exists in which a and b execute concurrently. We first define the notion of a general race or data race between a and b (denoted $\langle a, b\rangle$) over some given set of program executions, F, and then consider the implications of different choices for the set F.

Definition 4.1. A *general race* $\langle a, b\rangle$ *over* F exists iff

(1) a data conflict exists in P between a and b, and
(2) there exists a program execution, $P' = \langle E', \xrightarrow{T'}, \xrightarrow{D'} \rangle \in F$, such that $a, b \in E'$ and
  (a) $b \xrightarrow{T'} a$ if $a \xrightarrow{T} b$, or
  (b) $a \xrightarrow{T'} b$ if $b \xrightarrow{T} a$, or
  (c) $a \overset{T'}{\nrightarrow} b$.

Condition (2) is true if a and b can occur in an order opposite as in P or concurrently; these cases capture the notion that the execution order among a and b is not "guaranteed."

Definition 4.2. A *data race* $\langle a, b\rangle$ *over* F exists iff

(1) a data conflict exists in P between a and b, and
(2) there exists a program execution, $P' = \langle E', \xrightarrow{T'}, \xrightarrow{D'} \rangle \in F$, such that $a, b \in E'$ and $a \overset{T'}{\nrightarrow} b$.

4.2 Feasible Races

The most natural way to characterize a race $\langle a, b\rangle$ is to consider races over F_{DIFF}, since F_{DIFF} precisely captures the set of possible executions that also perform a and b. For general races, we have no choice: the smaller set F_{SAME} is inadequate since, by definition, a general race exists between two accesses only if they execute in an order different than in P (and therefore the shared-data dependence between them is also different). In contrast, data races could be reasonably viewed as occurring over either F_{DIFF} or F_{SAME}.

Definition 4.3. A *feasible general race* $\langle a, b\rangle$ exists iff a general race $\langle a, b\rangle$ over F_{DIFF} exists.

A *feasible data race* $\langle a, b\rangle$ exists iff a data race $\langle a, b\rangle$ over F_{DIFF} exists.

A feasible race locates precisely those portions of the execution that allowed a race, and thus represents the intuitive notion of a race illustrated in Section 2. However, checking for the presence of a feasible general race or data race $\langle a, b\rangle$ requires determining whether a feasible program execution in which $b \xrightarrow{T'} a$ (or $a \xrightarrow{T'} b$, or $a \overset{T'}{\nrightarrow} b$) is a member of F_{DIFF}. We have proven that deciding these membership problems is NP-hard (no matter what type of synchronization the program uses) and that locating feasible races is also NP-hard [19, 21]. It is therefore an intractable problem to locate precisely the race conditions exhibited by an execution of the program. This result suggests that, in practice, we must settle for an approximation, discussed next. Indeed, previously proposed race detection methods compute such an approximation.

4.3 Apparent Races

In practice, locating the intuitive notion of a race (a feasible race) would require analyzing the program's semantics to determine if the execution could have allowed b to precede a (or b to execute concurrently with a). Previously proposed methods take a simpler approach and analyze only the explicit synchronization performed by the execution. For example, a and b are said to have potentially executed concurrently (or in some other order) if no explicit synchronization *prevented* them from doing so. We can characterize the races detected by this approach by using F_{SYNC}, which is based only on orderings that the program's explicit synchronization might allow. Because the program may not be able to exhibit such orderings, these races capture a less accurate notion than feasible races.

Definition 4.4. An *apparent general race* $\langle a, b\rangle$ exists iff a general race $\langle a, b\rangle$ over F_{SYNC} exists.

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

Because not all program executions in F_{SYNC} are feasible, some apparent races may be spurious. Spurious races can occur whenever the values of shared variables are used (directly or indirectly) in conditional expressions or shared-array subscripts [20, 22]. In such cases, the existence of one event may depend on another event occurring first. For example, consider one process in P that adds data to a shared buffer and then sets a flag *BufEmpty* to *false,* and another process that first tests *BufEmpty* and then removes data from the buffer only if *BufEmpty* equals *false.* Such an execution has two apparent general races, between the accesses to *BufEmpty* and between the accesses to the buffer. However, the race involving the buffer is spurious—no feasible execution exists in which data is removed from the buffer *before* the buffer is filled (since *BufEmpty* is first tested before data is removed). If the operations on the shared buffer were instead complex and involved many shared-memory references, a large number of spurious races could be reported. Spurious races pose a problem since they are not direct manifestations of any program bug [20, 22]. The programmer can be overwhelmed with large amounts of misleading information, irrelevant for debugging, that masks the location of actual failures. Nonetheless, apparent race detection provides valuable information, since apparent races exist if and only if at least one feasible race exists somewhere in the execution [22]. Moreover, we have proven results showing how to reason about the potential feasibility of apparent races, and how a postmortem race detector can be extended to conservatively determine which apparent races are feasible and of interest for debugging [20–22].

We have also proven that, for program executions using synchronization powerful enough to implement two-process mutual exclusion, determining membership in F_{SYNC} and locating apparent races is NP-hard [19]. Membership in F_{SYNC} is efficiently computable only for weaker synchronization incapable of implementing mutual exclusion (such as **Post/Wait** style synchronization without **Clear** operations); all apparent races can thus be efficiently detected in executions of such programs [23]. It is important to note that data races are not of interest for such programs since weaker synchronization cannot implement critical sections. Exhaustively locating all apparent data races is therefore always NP-hard. In contrast, the complexity of apparent general race detection depends on the type of synchronization used. However, as we discuss next, it is sufficient for debugging data races to detect only a certain subset of apparent races, while debugging general races requires exhaustive detection.

4.4 Debugging with Race Condition Detection

An important aspect of debugging involves determining whether portions of the execution are race-free in the sense that they are unaffected by incorrect or inconsistent data produced by a race [5, 20]. For example, a programmer browsing an execution trace might focus only on portions of the trace recorded before any races occurred. These portions of the trace contain events that are guaranteed to be unaffected by the outcome of any race. Locating such events is important because the program may exhibit meaningless behavior after a race. In Figure 1b, for example, a data race that results in a negative balance might cause subsequent withdrawals to fail because of insufficient funds when in fact more money has been deposited than withdrawn. To locate events unaffected by a race, it is necessary to determine if portions of the execution are race free. However, we now argue that making this determination is inherently harder for general races (which introduce nondeterminism) than for data races (which only cause critical sections to fail). For executions containing data races, it suffices to detect the presence (or absence) of only a certain subset of the apparent races. In contrast, to debug executions containing general races, it is necessary to perform exhaustive apparent general race detection.

Data races can be viewed as a *local* property of the execution which can be determined directly from the actual program execution, P. Because an intended critical section can execute nonatomically only when other data-conflicting events are concurrently executing, an *actual* data race can be detected directly from P.

Definition 4.5. An *actual data race* $\langle a, b\rangle$ exists iff a data race $\langle a, b\rangle$ in P exists.[6]

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

An actual data race indicates the possibility that the atomicity of a critical section may have actually failed. In contrast, general races are a *global* property of the program which can be determined only by computing alternative event orderings. In general, determining if two events could have occurred in a different order requires analyzing the synchronization over the entire execution; there is no notion of an actual general race. Thus, unlike general races, actual data races can be easily located if the temporal ordering, $\xrightarrow{T}$, is known; computing alternative orderings is unnecessary.

This difference has important implications for debugging. Apparent data races that are not actual data races cannot produce inconsistent data (because no critical sections fail), but *any* apparent general race indicates such a possibility (because two shared-memory accesses could have executed in the incorrect order). The absence of an actual data race therefore indicates that all intended critical sections in P executed atomically, and the location of an actual race pinpoints places in the execution where inconsistent data should be expected. Even though not all apparent data races can be efficiently located, we can efficiently determine whether any actual data races occurred. Thus, we can easily determine whether the observed execution contains inconsistent data which cannot be relied upon for debugging. However, because there is no notion of an actual general race, we must exhaustively locate all apparent general races to make this determination. Only if *no* apparent races exist can we be sure that all shared-memory references occurred in the expected order (because no other order was possible).

These results suggest that we can efficiently debug programs containing data races by locating all actual and some apparent data races. However, we can only apply such an approach to programs containing general races if all apparent races can be efficiently located. As discussed above, exhaustive apparent general race detection is efficient only for programs that use synchronization incapable of implementing mutual exclusion. However, for more powerful types of synchronization (such as semaphores), conservative approximations that locate a superset of the apparent general races have been proposed [10, 11]. Such methods provide a means of debugging general races in the programs using such synchronization, but have the potential of misleading the programmer with potentially large numbers of spurious race reports.

5. CONCLUSION

This paper explores the nature of race conditions that can arise in shared-memory parallel programs. By employing a formal model we uncover previously hidden issues regarding the accuracy and complexity of dynamic race detection. We show that two fundamentally different types of races can occur: general races, pertaining to deterministic programs, and data races, pertaining to nondeterministic programs that use critical sections. We formally characterize these types of races. Previously, races have only been defined intuitively as occurring between data-conflicting blocks of code that "can potentially execute concurrently" or whose execution order is not "guaranteed." Our work is novel in that we explicitly characterize sets of alternative orderings that had the potential of occurring. Doing so is important because properties of the defined races, such as the accuracy and complexity of detecting them, depend upon which sets are considered. We uncover two variations of each type of race: one describes the intuitive notion of a race (the feasible race) and the other describes a less accurate notion (the apparent race) assumed by most race detection methods. Since locating the feasible races is NP-hard, the less accurate apparent races must be relied upon for debugging in practice. Moreover, we uncover fundamental differences in the complexity of debugging general races and data races. Debugging general races requires exhaustively computing alternative orderings (to determine whether the execution is nondeterministic), which is NP-hard for programs using synchronization powerful enough to implement mutual exclusion. Debugging data races requires simpler analyses (to determine if critical sections fail), which can be efficiently performed.

REFERENCES

1. Allen, T. R., and Padua, D. A. Debugging Fortran in a shared memory machine. In *Proceedings of the 1987 International Conference on Parallel Processing* (St. Charles, IL, Aug. 1987), pp. 721-727.
2. Balasundaram, V., and Kennedy, K. Compile-time detection of race conditions in a parallel program. In *Proceedings of the 3rd International Conference on Supercomputing* (Crete, Greece, June 1989), pp. 175-185.
3. Bernstein, A. J. Analysis of programs for parallel processing. *IEEE Trans. Electronic Comput. EC-15* 5 (Oct. 1966), 757-763.
4. Choi, J.-D., Miller, B. P., and Netzer, R. H. B. Techniques for debugging parallel programs with flowback analysis. *ACM Trans. Program. Lang. Syst. 13*, 4 (Oct. 1991), 491-530.
5. Choi, J.-D., and Min, S. L. Race frontier: Reproducing data races in parallel program debugging. In *Proceedings of the 3rd ACM Symposium on Principles and Practice of Parallel Programming* (Williamsburg, VA, Apr. 1991), pp. 145-154.
6. Dinning, A., and Schonberg, E. The task recycling technique for detecting access anomalies on-the-fly. IBM Tech. Rep. RC 15385, Jan. 1990.
7. Dinning, A., and Schonberg, E. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the 2nd ACM Symposium on Principles and Practice of Parallel Programming* (Seattle, WA, Mar. 1990), pp. 1-10.
8. Dinning, A., and Schonberg, E. Detecting access anomalies in programs with critical sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging* (Santa Cruz, CA, May 1991), pp. 79-90.
9. Emrath, P. A., and Padua, D. A. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* (Madison, WI, May 1988), pp. 89-99. Also appears in *SIGPLAN Notices 24*, 1 (Jan. 1989).
10. Emrath, P. A., Ghosh, S., and Padua, D. A. Event synchronization analysis for debugging parallel programs. In *Proceedings of Supercomputing '89* (Reno, NV, Nov. 1989), pp. 580-588.
11. Hemlbold, D. P., McDowell, C. E., and Wang, J.-Z. Analyzing traces with anonymous synchronization. In *Proceedings of the 1990 International Conference on Parallel Processing* (St. Charles, IL, Aug. 1990), pp. 70-77.
12. Hood, R., Kennedy, K., and Mellor-Crummey, J Parallel program debugging with on-the-fly anomaly detection. In *Proceedings of Supercomputing '90* (New York, NY, Nov. 1990), pp. 74-81.

13. Karam, G. M., Stanczyk, C. M., and Bond, G. W. Critical races in Ada programs. *IEEE Trans. Softw. Eng. 15,* 11 (Nov. 1989), 1471-1480.

14. Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. C-28* 9 (Sept. 1979), 690-691.

15. Lamport, L. The mutual exclusion problem: Part I—A theory of interprocess communication. *J. ACM 33,* 2 (Apr. 1986), 313-326.

16. Mellor-Crummey, J. M. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing '91* (Albuquerque, NM, Nov. 1991), 24-33.

17. Miller, B. P., and Choi, J.-D. A mechanism for efficient debugging of parallel programs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (Atlanta, GA, June 1988), 135-144. Also appears in *SIGPLAN Notices 23,* 7 (July 1988).

18. Min, S. L., and Choi, J.-D. An efficient cache-based access anomaly detection scheme. In *Proceedings of the 4th International Conference on Architectural Support for Programmıng Languages and Operating Systems* (Palo Alto, CA, Apr. 1991), 235-244.

19. Netzer, R. H. B., and Miller, B. P. On the complexity of event ordering for shared-memory parallel program executions. In *Proceedings of the 1990 International Conference on Parallel Processing* (St. Charles, IL, Aug. 1990), pp. II-93-II-97.

20. Netzer, R. H. B., and Miller, B. P. Improving the accuracy of data race detection. In *Proceedings of the 3rd ACM Symposium on Principles and Practice of Parallel Programming* (Williamsburg, VA, Apr. 1991), pp. 133-144.

21. Netzer, R. H. B. Race condition detection for debugging shared-memory parallel programs. Computer Sciences Dept. Tech. Rep. #1039 (Ph.D. Thesis), Univ. of Wisconsin-Madison, Aug. 1991.

22. Netzer, R. H. B., and Miller, B. P. Detecting data races in parallel program executions. In *Advances in Languages and Compilers for Parallel Processing,* A. Nicolau, D. Gelernter, T. Gross, and D. Padua, Eds. MIT Press 1991, 109-129.

23. Netzer, R. H. B., and Ghosh, S. Efficient race condition detection for shared-memory programs with post/wait synchronization. In *Proceedıngs of the 1992 International Conference on Parallel Processing* (St. Charles, IL, Aug. 1992).

24. Nudler, I., and Rudolph, L. Tools for the efficient development of efficient parallel programs. In *Proceedings of the 1st Israeli Conference on Computer System Engineering* (1988).

25. Randell, B., Lee, P. A., and Treleaven, P. C. Reliability issues in computing system design. *Comput. Surv. 10,* 2 (June 1978), 123-165.

26. Steele, G. L. Making asynchronous parallelism safe for the world. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages* (San Francisco, CA, Jan. 1990), pp. 218-231.

Received April 1991; revised October 1991; accepted February 1992