A taxonomy of race conditions.

D. P. Helmbold, C. E. McDowell*

UCSC-CRL-94-34
September 28, 1994

Board of Studies in Computer and Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064

ABSTRACT

Parallel programs are frequently non-deterministic, meaning they can give different results when provided the same input. These different results arise because variations in the timing of the multiple threads cause the threads to access shared resources in different orders. The phenomena that cause the non-deterministic behavior have been (and continue to be) variously referred to as access anomalies, race conditions or just races. In a recent paper, Netzer and Miller made an important contribution to formalizing and standardizing adjectives that can be applied to "races."

In this paper we continue this effort by presenting a refined taxonomy for races in parallel programs. The terminology we suggest is not always consistent with that used previously and we describe why we believe our terminology is more descriptive.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming, D.2.5 [Software Engineering]: Testing and Debugging - Debugging Aids, Tracing
Additional Keywords and Phrases: trace analysis, race detection, event ordering

*This work was partially supported by the National Science Foundation grant # CCR-9102635.

1 Introduction

Parallel computers are an increasingly important part of high performance computing. A significant number of these machines are programmed using a conventional language with extensions for creating threads[1] and some form of explicit synchronization (e.g. doall or fork with message passing). Many of these programs are intended to be deterministic, but due to synchronization errors are nondeterministic. Although some programs may be intentionally nondeterministic, errors can result in additional unwanted nondeterminism. In both cases it may be desirable to identify the sources of nondeterminism when debugging a program. This is particularly useful for those programs that are intended to be deterministic and might also be useful for intentionally nondeterministic programs provided the information about sources of nondeterminism is presented in a suitable manner.

Informally, a "race" exists between two program events if they conflict (e.g. one reads and the other writes the same memory location) and their execution order depends on how the threads are scheduled (this intuitive notion is formalized in Section 2).

Before you can design and build a tool to detect races in parallel programs you must first determine:

> What ordering relationships *should* hold between statement instances in a program?

Having done that, you then build a tool to determine when those ordering relationships do not hold.

In this paper we examine all possible ordering relationships that can hold between two program events and classify each possibility as either a non-race or belonging to one of four classes of races. We further refine our terminology for races by considering four additional orthogonal attributes of races: the affect on control flow, the severity, and the feasibility.

2 Events and Races

Informally, an execution of a program contains a race if the result of some computational step depends upon the scheduling of the individual threads of execution[2]. Netzer and Miller [NM92] developed a formal model of races that served as a starting point for our development. Their model includes two orthogonal attributes of races: with attributes *general* and *data* on one axis and *feasible*, *apparent*, and *actual* on the other axis. They define a data race as a pair of conflicting accesses that can overlap (i.e. are not atomic or protected by some type of critical section) and a general race as conflicting accesses where the access order is not guaranteed. An actual race is one that actually occurred in a particular execution and only

[1] For the purposes of this paper, thread, task and process are equivalent and we will use thread throughout.

[2] Within this definition we assume that the values read from an external clock are part of the fixed input rather than the result of a computational step. Without this assumption even sequential programs can give different results on different executions. The focus of this paper is on nondeterminism introduced by insufficient synchronization.

applies to general races. A feasible race, as the name suggests, is a race that did not occur but does occur in another execution. An apparent race is one that appears possible based only the limited information in a trace, but cannot occur.

In the remainder of this section we formalize our notion of a race and extend Netzer and Miller's categorization. We assume that any thread's execution can be represented by a sequence of atomic operations.

Definition 1: *An* **event** *is a contiguous sequence of one or more atomic operations executed by a single thread.*

Although events are not necessarily atomic, we can represent each event by an atomic beginning and an atomic ending. For our purposes an execution of the program is any ordered list of event beginnings and endings that adheres to the program semantics. Note that different observers might see different execution orders when examining the same run of the program (see [Lam78]). This is not a problem as we are primarily interested in the set of all possible executions rather than identifying a single execution associated with a particular run of the program.

Our definition of "event" is rather broad and can result in events that are "too big." In particular, if a single event may include several synchronization operations, the events may overlap in time but the subparts of the event that might cause a race could be properly ordered by the synchronization operations that are part of the event. This could result in spurious or harmless races being reported. In practice the events for a programming system should be sufficiently small to distinguish accesses to shared resources that are separated by synchronization. One example of how spurious races can be reported is when two properly synchronized tasks that access shared memory are treated in their entireties as single events. The two events would be concurrent and any shared accesses would also appear concurrent at this level of granularity.

Some kinds of races require the identification of events across executions. We would like an equivalence relation allowing us to determine when two events from different executions are the same. Because events are associated with source statements, it is possible to draw conclusions about the program statements from which the events are derived. For our equivalence between events we treat any conditional access to a shared resource as if it were explicitly represented by the program's flow of control. For example, if array `A[]` is a set of shared variables then the assignment `A[i] := expr` is treated as a case statement that branches on the value of `i`. The main effect of this assumption is that all instances of a simple statement (Definition 2) access the same shared memory locations. Note, however, that different instances of a compound statement can access different memory locations. Furthermore, we treat the evaluation of the branch condition in a compound statment like a simple assignment statement.

Definition 2: *A* **simple statement** *is a syntactic portion of a program such that if any instruction in the machine level translation of the statement is executed every instruction from the machine level translation of the statement will be executed. A* **compound statement** *is any group of syntactically contiguous simple statements.*

Definition 3: *Two events in different executions of the same program are* **equal** *(i.e. can be considered to be the same event) if*

- *they occur in the same thread,*
- *their constituent atomic operations are derived from the same simple source program statements, and*
- *both events are the n^{th} occurrence of their constituent atomic operation sequences by the thread.*

Thus an event is uniquely identified by its source program statements, the thread executing it, and a count indicating the number of times its source program statements have been previously executed by its thread. Two events are not the same if they represent the same compound statement but different simple statements. For example, the statement `if (x=0) then x:=1; else x:=2;` can be executed two different ways. The test can succeed, causing the execution of `x:=1;`, or the test can fail, causing the execution of `x:=2`. As these have different sequences of atomic operations they cannot be called the same event.

Note also that two events are the same if they are the n^{th} execution of their entire action sequences. Partial executions of their action sequences attributed to other events don't count. Consider the statement `if (x=0) then x:=1; else x:=2;` occurring in a loop which gets executed several times. Assume that the level of event granularity is such that each execution of this compound statement is represented by its own event (either a `true; x:=1` event or a `false; x:=2` event). Now the third `true; x:=1` event in one execution is equal to the third `true; x:=1` event in another execution, even if they occur in different iterations and/or after different numbers of `false; x:=2` events.

Definition 4: *Let events e_1 and e_2 be two events occurring in an execution of a program. If e_1 completes before e_2 begins then we say e_1* **happened before**[3] *e_2, written $e_1 \rightarrow e_2$. If e_1 begins before e_2 ends and e_2 begins before e_1 ends then the two events* **overlap**. *If either e_1 and e_2 overlap or $e_1 \rightarrow e_2$, then we write $e_2 \not\rightarrow e_1$.*

Note that the happened before and overlap relationships are for a particular execution and that if $e_1 \rightarrow e_2$ (or $e_1 \not\rightarrow e_2$) then *both* e_1 and e_2 occur in the execution.

Definition 5: *Fix an input to the program. Event e_1 is* **ordered before** *event e_2 if in every execution of the program on the input in which either event occurs, $e_1 \rightarrow e_2$.*
Two events, e_1 and e_2, are **ordered** *if e_1 is ordered before e_2 or e_2 is ordered before e_1.*

Definition 6: *Fix an input to the program. Event e_1 is* **semi-ordered before** *event e_2 if for that input*

- *every execution where both e_1 and e_2 occur, $e_1 \rightarrow e_2$,*
- *there exists an execution containing e_1 but not e_2 and*
- *every execution that contains e_2 also contains e_1.*

Two events, e_1 and e_2, are **semi-ordered** *if e_1 is semi-ordered before e_2 or e_2 is semi-ordered before e_1.*

[3] This is a strictly *temporal* relation and should not be confused with Lamport's *causal* "happened before" relation [Lam78].

Definition 7: *Two events are* **unordered** *if they are neither ordered nor semi-ordered.*

Definition 8: *Two simple statements* **conflict** *if they both access the same shared resource and one (or both) of the accesses modifies the resource. The accesses can be explicit as in access to a shared variable or implicit as in a communication port used for message passing.*

Definition 9: *Two different events* **conflict** *if they represent the execution of conflicting simple statements.*

Definition 10: *Fix an input to the program. If two conflicting events are unordered (with respect to the input) then there is a* **race** *between the two events on the input.*

It is sometimes desirable to discuss races and the ordering relationships of statements in programs (in contrast to events in executions of programs).

Definition 11: *A program contains a race between statements s_1 and s_2 if there is an input $\mathcal{I}$ and events e_1 and e_2 such that:*

1. *e_1 represents the execution of an instance of s_1,*
2. *e_2 represents the execution of an instance of s_2,*
3. *s_1 and s_2 contain (or are) conflicting simple statements, and*
4. *there is a race between e_1 and e_2 on input $\mathcal{I}$.*

Note that a race between statements is a property of the program whereas a race between events is a property of the program/input pair.

3 Ordering properties of Races

Each execution provides certain ordering or concurrency relationships between the events in the execution. A race exists for a particular input if there are two conflicting events, e_1 and e_2, and either an execution (on that input) where e_1 and e_2 overlap or a pair of executions (on that input) where $e_1 \rightarrow e_2$ in one execution and e_2 happens before (or without) e_1 in the other.

When only a single input is considered, we can classify races based on the relationships between the two events in the various executions. Given two events e_1 and e_2, there may be executions where:

1. $e_1 \rightarrow e_2$,
2. $e_2 \rightarrow e_1$,
3. e_1 and e_2 overlap,
4. e_1 occurs but e_2 does not,
5. e_2 occurs but e_1 does not, or
6. neither e_1 nor e_2 occur.

For any two events some set of the six possible orderings above will occur when all executions are considered. Given six distinct elements there are 64 (= 2^6) distinct sets possible. Each set potentially represents a different kind of event pair (e.g. always ordered, sometimes ordered, never ordered, never occur together...). Some of the 64 sets are not very interesting. The presence of Case 6 executions, where neither event occurs, does not affect the existence of races. This reduces the number of potentially interesting possibilities to 32 (see Table 3.1 below).

Of these 32 combinations, two describe ordered events (1 only and 2 only), and two combinations describe semi-ordered events (1 with 4 and 2 with 5). In three other combinations at least one of the two events is never executed (4 only, 5 only, and none of 1–5). The remaining 25 combinations describe races. We divide these into four different kinds of races. Recall that races between events are with respect to a particular input. This provides the first of four orthogonal attributes we will assign to a race. We call this the *ordering* attribute.

concurrent race: In every execution of the program on the fixed input where both e_1 and e_2 occur, they overlap.

general race: There exist executions of the program on the fixed input in which e_1 and e_2 overlap and executions where either $e_1 \rightarrow e_2$ or $e_2 \rightarrow e_1$.

unordered race: There exist executions of the program on the fixed input in which $e_1 \rightarrow e_2$ and executions in which $e_2 \rightarrow e_1$ but no execution in which e_1 and e_2 overlap.

artifact race: There exist executions of the program on the fixed input where e_2 occurs but e_1 does not and there exist executions where either $e_1 \rightarrow e_2$ or e_1 occurs but e_2 does not, but there are no executions on the fixed input where either e_1 and e_2 are concurrent or $e_2 \rightarrow e_1$.

Table 3.1 summarizes these definitions in the 29 cases where both e_1 and e_2 occur. Note that the definition of artifact races should be read as symmetric (if the conditions are met with e_1 and e_2 swapped, then it is still an artifact race).

Our definition of a concurrent race may appear strange at first (it did to us). If two events *can* happen at the same time, it would appear that either could happen *before* the other, especially if they do not contain synchronization primitives. The following two-thread code fragment gives one example of how events without synchronization must be executed concurrently.

```
begin                                   begin
  x:=1;                                   x:=0
  if (x=0) then y:=0;                     y:=1
end                                     end
```

The event “`x:=1;` test `x;` `y:=0;`” can only happen if variable `x` is set to zero concurrently. Thus if these are the only assignments to `x`, then the event “`x:=1;` test `x;` `y:=0;`” must occur concurrently with the event “`x:=0;` `y:=1;`” in the other thread.

Netzer and Miller's [NM92] “data race” is equivalent to the union of both our concurrent races and general races. Specifically they use data race to describe any race where the events do overlap in some execution. Our classification is more specific (i.e. it separates their data

There exists executions where					
$e_1 \rightarrow e_2$	$e_2 \rightarrow e_1$	overlap	e_1 only	e_2 only	
yes	yes	yes	y/n	y/n	general race
yes	yes	no	y/n	y/n	unordered race
yes	no	yes	y/n	y/n	general race
yes	no	no	y/n	yes	artifact race
yes	no	no	y/n	no	not a race
no	yes	yes	y/n	y/n	general race
no	yes	no	yes	y/n	artifact race
no	yes	no	no	y/n	not a race
no	no	yes	y/n	y/n	concurrent
no	no	no	yes	yes	artifact race

Table 3.1: Summary of possible ordering relationships.

races into two subgroups) and we believe the qualifier "data" is more descriptive when used as we do below.

We refine Netzer and Miller's term "general race" into our categories "concurrent", "general", and "unordered". We feel that this refinement is useful, as the different kinds of races typically result from different kinds of errors. Unordered races are a particularly important category as they typically result from the use of mutual exclusion when a stronger kind of synchronization is required, such as using critical sections to protect non-associative or non-commutative modifications to a shared variable. Concurrent races occur when two events interact in an unforeseen way, and mutual exclusion is an appropriate fix. Our general races are similar to unordered races except that they indicate a total lack of synchronization rather than the presence of mutual exclusion.

We have borrowed the term "artifact race" from Netzer and Miller [NM91], these races result from other races in the program. In their later paper [NM92] they do not mention artifact races and their definition of general and data races does not include what we now define as artifact races. Although not identical to their earlier definition of "artifact," our definition is intuitively similar and hence our decision to use the same term.

An artifact race can never be in the group of "first" races (as defined in [NM91]). In particular, an artifact race has the property that the result of some "earlier" race affects the flow of control, preventing an event from being executed. This observation suggested an orthogonal attribute of races which we describe next.

4 Other attributes of races

Races can have other important attributes in addition to their ordering properties. Here we briefly discuss three other attributes: whether the race affects the program's control flow, the severity of the race, and the feasibility of the race.

4.1 Control vs Data

A **control** race causes a thread to take different paths depending upon how the race is resolved. If the control flow is not affected by a race then it is a **data** race.

By definition, every race involves conflicting data accesses and could be intuitively thought of as a data race but we reserve data race for those races that *only* affect data and not control flow.

4.2 Severity

A third potentially useful attribute of a race is its severity. We currently identify two severity levels, **critical** and **benign**. A benign race has no external effect on the results of the program (Padua and Emrath [EP88] call this internal non-determinism), while the outcome of a critical race can affect the program's result. Protecting a critical section with locks (mutual exclusion) does not prevent a race, but can make races benign. Consider the following code fragments with `x` initialized to zero.

```
lock;                                 lock;
  x := x + 1;                           x := x + 2;
unlock;                               unlock;
```

The two updates to `x` can happen in either order and thus create a race. However, the value of `x` is always three after both critical sections have been executed. Whenever a set of commutative updates to a shared variable must be completed before a variable is used and there are only unordered races between the updates, the races between updates are benign. This might not be the case if the two assignment statements were able to execute concurrently (so that both read the original value of `x`). Unordered races are often benign when they are caused by commutative[4] updates to a shared variable. The goal of at least one tool [Ste93] is to ignore the unordered races and report only concurrent and/or general races.

4.3 Feasibility

Finally we note that previous work in race detection has distinguished between **feasible** and *infeasible* races [NM92]. This is really a characteristic of the race detection system which results from the need for approximate solutions. Any race that is reported but could never actually occur is **infeasible**.

[4] Even when protected by locks, non-commutative updates (such as when the "`x := x + 2;`" statement is replaced by "`x := x * 2;`") are still likely to be sources of nondeterminism.

5 Conclusion

We have presented a classification of races that includes four orthogonal attributes, event ordering, control versus data, severity and feasibility. Whenever possible we have adopted terminology that has been previously proposed. Our race taxonomy is complete in that it encompasses **all** possible races. It also separates races into different categories based on the type of error that typically causes that kind of race (e.g. unordered races indicate mutual exclusion was used when a stronger form of synchronization is necessary). Further refinement is possible, but any type of race can be precisely categorized by our taxonomy (Table 3.1).

Acknowledgement

The taxonomy of races was significantly influenced by an extended email dialogue with Rob Netzer.

References

[EP88] P. A. Emrath and D. A. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proc. Workshop on Parallel and Distributed Debugging*, pages 89–99, May 1988.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, July 1978.

[NM91] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. *SIGPLAN Notices (Proc. PPOPP)*, 26(7):133–144, 1991.

[NM92] Robert H.B. Netzer and Barton P. Miller. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, pages 74–88, March 1992.

[Ste93] N. Sterling. WARLOCK - a static data race analysis tool. In *Proc. Winter Usenix*, pages 97–106, 1993.

End of paper. Total pages = 13.