What is a race in a program and when can we detect it?

D. P. Helmbold, C. E. McDowell

UCSC-CRL-93-30
August 2, 1993

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

ABSTRACT

This paper presents a taxonomy of methods for detecting race conditions in parallel programs, shows how recent results fit into the taxonomy, and presents some new results for previously unexamined points in the taxonomy. It also presents a taxonomy of "races" and suggested terminology.

Keywords: trace analysis, race detection, debugging, parallel programming, event ordering

1 Introduction

In this paper we present a taxonomy of methods for detecting race conditions, show how recent results fit into the taxonomy, and present some new results for previously unexamined points in the taxonomy.

Shared memory parallel computers are an important part of high performance computing today and will continue to be so for many years. A significant number of these machines are programmed using a conventional language, modified to contain some form of explicit parallelism (e.g. fork, doall) and some form of explicit synchronization (e.g. join, semaphores). Sometimes these programs are intended to be deterministic, but due to synchronization errors are nondeterministic (i.e. contain race conditions). Other programs are intended to be nondeterministic (at least at some level – this is discussed further later). In both cases it may be desirable to identify the sources of nondeterminism. This is particularly useful for programs that were intended to be deterministic but 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 (or tasks) are scheduled (a formal definition is given in Section 2).

There are many questions that can be asked about the possible "races" in a parallel program.

- What ordering relationships should hold between statement instances (i.e. what statement instances conflict)?
- What ordering relationships hold between statement instances?
- What are all of the races in this program?
- Are there any races in this program?
- What shared memory addresses are accessed by a statement (instance)?

Current algorithms for detecting races in programs answer (or attempt to answer) one or more of the above questions.

In Section 2 we examine all possible ordering relationships than can hold between two program events and then classify them into non-races and four classes of races. In Section 4 we present our taxonomy of race detection methods. This section surveys results regarding the complexity of determining precise event ordering relationships, and includes three new negative results. In Section 5 we summarize the current known algorithms that can correctly answer the question, "Are there *any* races in this program?" In Section 6 we briefly touch on the the issue of determining the conflicting accesses to shared data.

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[1]. Netzer and Miller [NM92] developed a formal model of races that distinguished races that served as a starting point for our development. Their model includes two orthoganol attributes of races, with attributes *general* and *data* on one axis and *feasible*, *apparent* and *actual* on the other axis. They define a general 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 data race as confliciting acceses where the order is not guaranteed but which cannot overlap. An actual race is one that actually occured in a particular execution and only 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 catagorization. We assume that any thread's (or task'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.*

This definition 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 (i.e. be concurrent) 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. The practical impact would be that a race might be reported that didn't exist. Unless otherwise specified we assume that the particular events for a programming system are sufficiently small to distinguish accesses to shared resources that are separated by synchronization. An example of how spurious races being reported would be when two properly synchronized tasks that access shared memory are each treated in their entirety as an event. The two events would be concurrent and any shared accesses would also appear concurrent at this level of granularity.

We would like an intuitive equivalence relation for events from different executions. This will allow us to draw conclusions about the program statements from which the events are derived. In order to do so we must assume that any conditional reading or writing of shared memory is treated as if it was done using explicit conditional control flow (e.g. a case statement). For example, 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 access the same memory locations.

Definition 2: *Two events from 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,*

[1] Within this definition we assume that the values read from an external clock are part of the fixed input.

- *their constituent atomic operations are derived from the same source program statements, and*
- *both events are the n^{th} execution of their constituent atomic action sequence 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.

Definition 3: *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 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 concurrent 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 4: *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 5: *Fix an input to the program. Event e_1 is* **semi-ordered before** *event e_2 if*
- *in every execution on the input where both e_1 and e_2 occur, $e_1 \rightarrow e_2$,*
- *there exists an execution on the input containing e_1 but not e_2 and*
- *no execution on the input contains e_2 but not 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.*

Definition 6: *Two events are* **unordered** *if they are neither ordered nor semi-ordered.*

Definition 7: *Two elementary statements* **conflict** *if they both access the same shared memory location[2] and one (or both) of the accesses changes the value (or state) of the location. The accesses can be explicit as in access to a shared variable or implicit as in a communication port used for message passing.*

Definition 8: *Two different events* **conflict** *if they represent the execution of conflicting simple statements.*

Definition 9: *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.*

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 the 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.

---

[2] Reads and writes of a communication port or channel than can be shared are considered to be accesses to shared memory. Thus or race model can be applied to both "shared memory" systems and message passing systems.

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.

By combining the possibilities there are 64 ($= 2^6$) ways the events can be ordered considering all of the executions on the given input. 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.

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 groups. Recall that races between events are with respect to a particular input.

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 on the fixed input 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 2.1 summarizes these definitions in the 29 cases where both e_1 and e_2 occur.

It may appear strange that two events must occur concurrently, especially if they do not contain synchronization primitives. However, consider the following code fragements from two different threads.

```
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; y:=0;`" must occur concurrently with the event "`x:=0; y:=1;`" in the other thread.

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 sequential
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 2.1: Summary of possible ordering relationships.

We have borrowed the term "artifact race" from Netzer [NM91], these races result from other races in the program. 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 suggests an orthogonal attribute of races.

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[3].

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

```
lock;
  x := x + 2;
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 (depending on the granularity of the atomic memory actions). Unordered races

[3]Every race by definition 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.

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.

Finally we note that previous work in race detection has distinguished between *feasible* and *infeasible* races. 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*.

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

Definition 10: *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 a conflicting instance of s_2, and*
3. *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 A taxonomy of event ordering approaches

Previously, results in race detection have been classified as static (compile time), post-mortem trace based, or on-the-fly. The primary distinction between on-the-fly analysis and post mortem analysis is that in on-the-fly analysis the trace is analyzed as it is generated, thus the entire trace does not need to be stored. This permits more detailed tracing, often including all of the accesses to shared memory. On-the-fly race detection naturally focuses on those races involving the shared memory accesses reported during the execution. This is somewhat different from the problem generally addressed in post mortem trace analysis where an attempt is made to determine orderings between all blocks (without regard to shared memory accesses which cannot in general be stored in a post mortem trace due to space limitations).

We will unify static, post-mortem and on-the-fly approaches by viewing each as a type of static analysis on an appropriately constrained programming model. We will constrain the programming model along two major axes. The first axis identifies the type of synchronization. The second axis identifies the constraints on the control flow of the program. The current known results on computing ordering relationships are summarized in Table 3.1 at the end of this section and described in Section 4. The following subsections detail the taxonomy.

[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.

3.1 Type of synchronization

The first axis identifies the type of synchronization. At the top level we only distinguish two types of synchronization: monotonic and non-monotonic. These terms were first applied to synchronization in [HM93]. Intuitively, a synchronization construct is monotonic if once a blocking operation becomes unblocked, it remains unblocked for the duration of the program (e.g. Post and Wait with no Clear - once an event is posted, any Wait operations on that event become unblocked and the effect of the Post cannot be undone). This intuitive description is only intended to give a general idea of the classification and to motivate the choice of monotonic to describe the class. The intuitive notion also accurately describes all "real" monotonic synchronization constructs that we have examined but is not sufficient to precisely characterize the class. The formal definition is given below.

Definition 11: *A set of synchronization constructs is* **monotonic** *if every branch-free parallel program using only synchronization constructs in the set either always terminates normally (all threads complete) or always deadlocks in the same state.*

Thus a set of synchronization constructs is monotonic if whenever an event becomes enabled, the event is executed before the program terminates or deadlocks.

Monotonic synchronization operations include: nested fork-join (e.g. nested parallel loops), ordered critical sections (i.e. properly paired and nested lock-unlock operations where whenever multiple locks are simultaneously held, they are always obtained in the same order), buffered send-receive where the sender names the receiver, and post and wait with no clear. Non-monotonic synchronization operations for which results have been published include: post and wait with clear, and semaphores.

3.2 Constraints on control flow

The possible constraints on the control flow are: no branching, no loops containing synchronization constructs (or equivalently, loops containing synchronization are unrolled), and unconstrained control flow.

Branch-free programs

During a program's execution, each instance of a conditional statement takes a particular branch. When the program's execution is traced, a record is made of the events (or perhaps only the important events) executed by each thread and when they are executed. This record defines a branchless program since all of the branching has been "hard wired" when the trace was generated. The way the branches get "hard wired" depends on both the input supplied to the program and the outcome of control races in the traced execution.

One possible goal is to determine the races exhibited by the traced execution. Since only one execution is considered, each detected race will involve two events which execute concurrently in the execution. Thus only concurrent races and some general races can be detected in this way.

Thread A		Thread B	
A1:	j := 0;	B1:	i := 1;
A2:	i := 0;	B2:	if (i=0) then
A3:	if (i=1) then	B3:	j := 1;
A4:	*k := 1;*	B4:	k := 2;

Figure 3.1: This program fragment has conflicting updates to shared variables i, j, and k (as well as conflicting reads to i in the if conditions). Assume each labeled statement is an event. Consider the branch-free program that results when event A2 is executed after event B1 and before event B2. Event A4 does not appear in this branch free program as the condition "i=1" in event A3 is hardwired to false. The branch-free program gives only a poor approximation to the races in the original program. In both the original program and the branch free program the event pairs (A2, B1), (A3, B1), and (A2, B2) are general races. However, another general race (A4, B4) exists in the original program but not the branch-free program. Furthermore, the pair (A1, B3) is a race in the branch-free program but not in the original program. In the original program A1 is semi-ordered before B3 and in the branch-free program the condition in B2 always evaluates to true whether or not statement A1 has been executed.

A more powerful approach is to consider all possible executions of the branch-free program on a particular input. The key sub-goal of this approach is a partial order indicating which pairs of events are ordered or semi-ordered. From this partial order and the knowledge of which events conflict one can determine which pairs of events are races. Since the branch-free program has the same set of possible executions on every input, one can use the pairs of events that are races for any particular input to determine which statement pairs in the program form a race.

Note that control races can affect the evaluation of branch conditions. Thus, even an exact analysis of the branch-free program can lead to incorrect results for the original program generating the trace. Some races may be missed because the branches leading to them were not taken in the traced execution. Other races may be incorrectly included because some branch conditions would be evaluated differently in the executions responsible for them. See Figure 3.1 for an example.

Programs with branches but no loops

The problem becomes even more difficult when we consider analyzing programs with branching (but without loops). For each input, the program with branching can be viewed as a set of branch-free programs. Each legal combination of branch choices for that input leads to one branch-free program. A simplifying assumption [CS88] is that all branch combinations

Thread A	Thread B	ThreadC
if (input=1)	wait(x);	wait(y);
then post(x);	S1;	S2;
else post(y);	post(y);	post(x);

Figure 3.2: This program fragment contains two conflicting statements, S1 and S2. Although either S1 or S2 can happen first, for any given input either S1 happens before S2 or S2 happens before S1 but not both. By Definition 10, this program does not contain a race.

are possible, so that any set of branch choices is legal. Without this assumption it is $\mathcal{NP}$-hard to determine which branch choices are legal (see Theorem 2).

Each branch-free program associated with a branching program/input pair has its own set of races between events. What one would like to determine is a partial order over the events where there is an arc from event e_1 to event e_2 if and only if e_1 and e_2 are ordered (or semi-ordered) by every branch-free program represented by the program/input pair. As above, this partial order can be combined with conflict information to obtain those pairs of events forming races.

Now consider the possible inputs for the branching program. For each input there is a set of event pairs which are unordered (with respect to that input). Taking the union of these sets of event pairs gives us all pairs of events that are unordered on any possible input. Using information on which event pairs conflict, we can then list the pairs of events forming races in the program.

The set of pairs of unordered events must be computed separately for each possible input. As shown in Figure 3.2, two conflicting statements that are not ordered the same across all inputs do not necessarily constitute a race. The order in which S1 and S2 from Figure 3.2 are executed depends on the input, but is the same on each particular input.

The assumption that all branch combinations are possible has the fundamental drawback that extra (spurious) races may be reported. Certain combinations of branches are often infeasible, and races in the branch-free program(s) using infeasible combinations of branches may result in infeasible races being reported. A combination of branches may be infeasible because two branch conditions may always compute the same value or because statements in (or the absence of statements from) one branch may determine the value of a later branch condition.

Unrestricted programs

Programs containing loops present an additional difficulty. If the number of loop iterations cannot be bounded at compile time, then the number of events executed by the program (and the number of branch conditions evaluated) is also unbounded. Thus a single program with loops can represent an infinite number of branch-free programs.

For each choice of input, we obtain a version of the looping program. Each version of the looping program represents a (possibly infinite) number of branch-free programs. For each input, we can (at least conceptually) identify[5] which pairs of events are ordered or semi-ordered, and (given conflict information) which pairs of events form races for that input.

We can then proceed in the same way as the loop-free case. The union over all possible inputs of these pairs of events forming races can then be used to determine which pairs of statements in the program are races.

4 Details of known results in our taxonomy of ordering event results

4.1 Unrestricted programs

If programs are allowed to have branches and unbounded loops, determining the ordering relationships between statement instances is undecidable, regardless of the type of synchronization used.

Theorem 1: *Deciding if there exists a race between two conflicting statements in an arbitrary shared memory parallel program is as hard as the halting problem.*

Proof: Given an arbitrary (sequential) program P and input $\mathcal{I}$, we create a new parallel program containing a new shared variable x initialized to 0. The parallel program forks, executing "print(x);" in one branch. The other branch first checks that the parallel program's input equals $\mathcal{I}$. If the input matches $\mathcal{I}$ then program P is simulated and when (if) the original program halts, the statement "x := 1;" is executed. If the input does not match $\mathcal{I}$ then the second branch terminates without accessing variable x. There is a race between the "print(x);" statement and the "x := 1;" assignment if and only if program P halts on input $\mathcal{I}$.

□

Nevertheless, programmers must still uncover data races in their parallel programs. Therefore approaches that compute approximate answers to the problem have been studied and continue to be investigated [Tay83, LC89, HM91].

4.2 No Loops and Monotonically Synchronized

Excluding arbitrary loops is necessary to avoid termination problems and undecidability. Loops executing a fixed number of times can be unrolled. This clearly affects the complexity of any analysis algorithm, but is essentially what happens in any trace based approach to race detection. Loops that do not contain synchronization operations and which are guaranteed to terminate are allowed because they do not affect the order analysis between events.

[5] Determining which pairs of events are ordered or semi-ordered is undecidable in general, see Theorem 1. However, the assumption that all combinations of branches are possible alleviates this problem.

		Exact Solution	Approximations
Branch free	Monotonic	• all monotonic are in $\mathcal{P}$ [HM93], • fork/join is in $\mathcal{P}$ [MC91, DS90, NR88], • ordered critical sections are in $\mathcal{P}$ (section 4.4), • post/wait no clear is in $\mathcal{P}$ [NG92],	
	Non-monotonic	• single semaphore is in $\mathcal{P}$ [LKN93], • post/wait/clear is $\mathcal{NP}$-hard (Thm: 3), • semaphores are co-$\mathcal{NP}$-hard[NM90]	semaphores [HMW93]
No loops	Monotonic	• fork/join is $\mathcal{NP}$-hard (Thm: 2), • post/wait no clear is Co-$\mathcal{NP}$-hard[CS88] even if all paths are executable,	fork/join [MC91, DS90, NR88], post/wait no clear [CKS90]
	Non-monotonic	• post/wait/clear is $\mathcal{NP}$-hard (Thm: 3 or [CS88]), • semaphores are $\mathcal{NP}$-hard	fork/join [MC91, DS90, NR88], post/wait no clear [CKS90]
Unrestricted	any	Undecidable (Thm: 1)	fork/join [MC91, DS90, NR88], ordered critical sections [Ste93], semaphores [McD89], message passing [DKF93], rendezvous [Tay83, LC89]

Table 3.1: What ordering relationships hold between statement (instances)?

Theorem 2: *Deciding if there exists a race between two conflicting statements in an arbitrary shared memory parallel program (containing explicit thread creation but no loops) is $\mathcal{NP}$-hard.*

Proof: By reduction from 3SAT. Create a parallel program that forks executing the statements `x:=1; print(x);` in one branch and `if(3SAT formula over input) then x:=0;` in the other branch. There is a race between the `print(x)` and the assignment `x:=0`, if and only if the formula is satisfiable for some input. □

The key difference between this result and the Post/Wait no Clear result of Callahan and Subhlok (see Section 4.2) is they assume all paths are executable and this trivial proof hinges

on whether or not one path is executable. The set of programs where all paths are executable is clearly a subset of all programs and hence they have shown that with the addition of Post and Wait the problem is still $\mathcal{NP}$-hard even for the smaller set of programs.

Post/Wait no Clear

Callahan et. al. [CKS90] have studied simple programs containing only if-then-else conditionals and Post/Wait synchronization without Clear (i.e. no loops). The Post/Wait operations are permitted to specify events within an array. They claim that as generally used, the index expressions for these events are amenable to standard dependence analysis for computing a dependence distance (i.e. the difference between the parallel loop index and the array index used by the Post or Wait). In an earlier paper [CS88] they prove that the problem of determining if a program is race free is Co-$\mathcal{NP}$-hard for even these relatively simple programs under the further assumption that all program paths are feasible.

In [CKS90], they have gone on to develop a dataflow formulation of the problem for which they can compute an approximate solution in polynomial time (the paper does not give the actual complexity). This approximation only applies to programs that are "serializable." By that they mean that if all parallel loops and parallel case statements (the only types of forking they support) are executed in sequential order (the cases from the parallel case are executed in the order they appear textually) then the program will complete without blocking. i.e. no Wait will be encountered until after a Post for the same event has been executed.

They give an algebraic formulation of the problem when the program is further restricted to contain only one Post for each event variable. The algebraic formulation provides an exact solution that appears faster in practice than the previous method. However, it involves a transformation to a system of linear equations and determining if there exists a non-negative integral solution to the system of equations. Although such interger linear programming problems are $\mathcal{NP}$-hard, the systems generated in practice are claimed to be generally small enough so that this is not a problem.

4.3 No Loops and Non-Monotonically Synchronized

All results in this area indicate that exact solutions are not tractable. As in the case of unrestricted programs, programmers must still uncover data races in their parallel programs. Therefore approaches that compute approximate answers to the problem have been studied and continue to be investigated [MC91, DS90, NR88, CKS90].

Post/Wait/Clear

The Co-$\mathcal{NP}$-Hard result from [CS88] also applies here. In fact, with the addition of Clear, even detecting races in branch-free (i.e. no conditionals or loops) programs is $\mathcal{NP}$-Hard (see Theorem 3).

Semaphores

Determining precisely the ordering relationships for branch-free programs containing semaphore synchronization is co-$\mathcal{NP}$-hard[NM90]. Therefore the problem is also co-$\mathcal{NP}$-hard when branches are permitted.

4.4 No Branches and Monotonically Synchronized

We proved in a previous paper [HM93] that computing the precise ordering relationships between events in branch-free monotonically synchronized programs can be done in polynomial time. For completeness we include here several previous polynomial time results for determining the precise ordering relationships between events for programs using specific sets of monotonic synchronization constructs.

Fork/Join

A number of methods have been developed in the context of on-the-fly race detection that could be used as polynomial time algorithms for determining event orders in branch free fork/join programs[MC91, DS90, NR88]. Some recent efforts have focused on reducing the number of events that must be traced [Net93, MC93]. More importantly, these are on-line algorithms that require only small amounts of storage and can thus be done "on-the-fly."

Critical Sections with Lock/Unlock

In programs that contain only fork/join synchronization, if there is a race between two events, then it must be a general race. With the addition of ordered critical sections, the races may be either general races (i.e. not protected by the same lock) or unordered races (i.e. protected by the same lock). These two kinds of races can be distinguished by comparing the locks held when the events were executed. For branch-free programs, this comparision can easily be done using $O(L^2)$ time and $O(L)$ space per event, where L is the maximum lock nesting depth. In practice the lock nesting depth is very small (i.e. 0 or 1) [DS91].

Post/Wait no Clear

Netzer and Ghosh [NG92] have an algorithm that precisely determines the event orderings for a trace of a program that uses Post/Wait synchronization with no Clears. The algorithm constructs a DAG where the nodes are the events in the trace and the edges represent the guaranteed orderings between events. That is, two events e_1 and e_2 are ordered (definition 4), if and only if there is a path from the node for e_1 to the node for e_2. The graph construction requires $O(np)$ time and $O(np)$ space where n is the number of events in the trace and p is the number of threads.

4.5 No Branches and Non-monotonically Synchronized

Similar to Sections 4.1 and 4.3 most results in this area indicate that exact solutions are not tractable. The one exception to date is a recent result showing that finding the exact solution for programs that use only a single semaphore can be done in polynomial time.

Single Semaphore

Computing the exact ordering relationship between events for a loop-free program that synchronizes using only a single semaphore can be done in $O(n^{1.5}p)$ time [LKN93] where n is the number of events and p is the number of threads. The algorithm presented by Lu, Klein, and Netzer determines if two events are ordered by solving a kind of scheduling problem. When P-operations are assigned a cost of $+1$ and V-operations are assigned a cost of -1, a branch-free program using a single semaphore can execute to completion if and only if it has a schedule whose cumulative cost is always ≤ 0. Thus one can tell if a program can complete by finding a schedule where the maximum cumulative cost is minimized. Although this kind of scheduling problem is $\mathcal{NP}$-complete in general, Lu, Klein, and Netzer show how a solution for series-parallel graphs can be modified to determine if two events in a branch-free program are ordered.

As presented in their paper, Lu et.al. will determine some events to be ordered that should not be. This derives from their claim that if you artificially order two events and then fail to find a schedule the events cannot occur in that order (and hence are always ordered in the reverse direction). Only a small change is needed to get the preferred result. It could be that two events could occur in the artificially added order but the program would deadlock later in its execution. Instead of insisting on a schedule for the entire program it is only necessary to fine a prefix of a schedule that includes the two artificially ordered events. Their algorithm provides the necessary information to determine if such a prefix exists.

Post/Wait/Clear

With the addition of the Clear operation, determining precisely the ordering relationships for branch-free programs becomes $\mathcal{NP}$-hard.

Theorem 3: *Deciding if there exists a race between two conflicting statements in an arbitrary shared memory parallel program, containing explicit thread creation and Post/Wait/Clear synchronization but no loops or branches, is $\mathcal{NP}$-hard.*

Proof: The proof is by reduction from 3-SAT. We construct the following program which encodes an instance of the 3 CNF satisfiability problem. This program will contain a race if and only if there is a satisfying assignment to the 3 CNF formula.

- Define signal START.
- For each variable X define 4 signals, Xt (X is true), Xf (X is false), and XisT XisF (X has a value).
- For each clause C define a signal Ct (C is true).

- For each variable X create two threads, TXt and TXf as follows:

TXt:	**TXf:**
wait START	wait START
clear Xf	clear Xt
post XisT	post XisF
wait Xt	wait Xf
for each C containing X	for each C containing not X
post Ct	post Ct
end for	end for

- Create two other threads - main and racer as follows:

main:	**racer:**
for each variable X	for each variable X
post Xt	wait XisT
post Xf	wait XisF
end for	end for
post START	*race statement*
for each clause C	for each variable X
wait Ct	post Xt
end for	post Xf
race statement	end for

- **Claim:** The race statements can execute concurrently if there is a truth assignment satisfying all of the clauses.
  1. Run main until just after "post START"
  2. Run to completion each TXt if X true in truth assignment or TXf if X false in truth
  3. Run to completion remaining TXt's and TXf's.
  4. Observe that all Ct, XisT and XisF are now posted.
- **Claim:** The race statements cannot execute concurrently if there is no truth assignment satisfying all of the clauses.
  1. All Ct must be posted before main executes race statement.
  2. For some X, both TXt and TXf must have posted signals for all Ct to be posted (otherwise the clauses are all satisfiable)
  3. Either Xt or Xf must have been posted twice, and thus the race statement in racer must have already been posted.

NOTE: The operations on the XisT and XisF events are not neccessary for the theorem. However, if these operations are removed then the program will have many executions which end in deadlock.

□

Semaphores

Determining precisely the ordering relationships for even branch-free programs containing semaphore synchronization is co-$\mathcal{NP}$-hard[NM90].

The results in this area are therefore restricted to approximations. Helmbold et.al. [HMW91] and Netzer and Miller [NM91] have pursed two complimentary approaches. One group has been attempting to find as many races (unordered blocks) as possible, while the other has been trying to distinguish between feasible, infeasible and artifacts (see below) for those races they can find.

5 Are there *any* races in this program?

Because the problem of detecting races in parallel programs is in general intractable, approximations must suffice. There are two ways to err: report races that do not really exist (*infeasible races*) or fail to report some of the races. The problem with the former is that the user may be inundated with infeasible races and miss the real race. The problem with the latter is that a program may be reported to be race free when in fact it is not. A compromise that has been achieved in some situations is to guarantee to report a *non-empty* subset of the actual races. While some races may still be missed, if a program (or execution) is reported to be race free, then the report is accurate.

5.1 Fork/Join

Mellor-Crummey [MC91] describes a method for analyzing programs containing only properly nested fork/join parallelism. This approach requires $O(VN)$ space where V is the number of shared variables and N is the maximum nesting depth of the forks. Also each monitoring operation requires $O(N)$ time. The method is called Offset-Span labeling and is similar to *English-Hebrew* labeling [NR88]. In particular the label for each thread that is created during the execution of the program is computed based only on the labels of its immediate predecessors (the thread executing the fork or the threads resulting in a successful join). The length of each label is proportional to the current nesting depth and at most three labels must be stored on-the-fly for each shared variable. The most significant contribution of Offset-Span labeling is that a single execution is sufficient to identify a non-empty subset of the races that could occur for a given input.

5.2 Critical Sections

Dinning and Schonberg [DS91] describe an approach to detecting access anomalies in programs that contain critical sections (i.e. properly nested binary semaphores). This approach can use any existing method for determining when two blocks are ordered (e.g. Offset-Span labeling) ignoring the orderings imposed by the unlock-lock operations. As one

would expect, ignoring the unlock-lock orderings results in many false anomalies. This is solved by adding lock covers to the labels for blocks in critical sections. A lock cover indicates what locks are held when a block executes. If there is no nondeterminism "propagated" by the critical sections then the access anomalies reported will include at least one anomaly (if there are any) from the set of access anomalies that could occur given the input supplied during the analyzed execution.

In addition to needing the lock covers, this approach requires a larger history for each shared variable than the approaches described in Section 4.4. For each shared variable the history may contain as many as $T \times R$ labels and lock covers representing the latest writes and similarly for the reads. T is the maximum degree of concurrency and R is the number of lock covers which is bounded by 2^K where K is the number of locks.[6] Dinning and Schonberg claim that the use of nested critical sections is rare resulting in very few lock covers in practice.

5.3 Semaphores

We have developed an algorithm for analyzing traces of programs that contain semaphore synchronization. In [HMW93] we proved that our algorithm will find at least one race from the set of possible races that can occur for a given input if any exist.

6 What shared memory addresses are accessed by a statement (instance)?

Operationally, race detection systems can be divided into three groups, compile time systems, post-mortem trace based systems and on-the-fly systems. A distinguishing characteristic is the degree to which the aliasing problem is solved/avoided. Compile time approaches must attempt to solve the problem (e.g. conventional vectorizing compiler analysis). Space limitations generally prohibit post-mortem systems from storing all shared memory accesses during data collection. Instead some type of summary information is recorded and then the actual addresses are estimated or re-generated when needed. On-the-fly systems have no such space limitation and can use the actual memory addresses in the analysis, thereby eliminating any aliasing problems.

Any monitoring/trace based approach can therefore easily answer the question: "Given shared memory location X, what statements access X?" By "easily" we mean that the cost of answering this question is dominated by the cost of determining the ordering relationships. In general it will add a constant time cost to the processing of each statement (event).

[6] Simply seeing if the intersection of the locks held when accessing a variable is not sufficient. One access may be protected by locks a and b, another by locks b and c, and a third by locks a and c. Although the intersection of the locks held is empty, there is no concurrent race between the three accesses.

For compile time systems there has been a large body of work performed on this problem restricted to statements within the same loop nest. This work answers a variation of the previous question, the new question being:

Given two statements, S1 and S2, can they access the same location?

For two statements outside of a common loop nest there has been no published work that we are aware of.

7 Conclusion

This paper makes three primary contributions in the area of race detection in parallel programs. First, we have proposed a taxonomy of "races" that describes all possible, shared memory races. Further refinement is possible, but any type of race can be precisely categorized by our taxonomy (Table 3.1). Second, we have presented a taxonomy of approaches to detecting races. The purpose of this taxonomy is to organize the previous results and determine just how much we actually know today about the "race detection" problem. We then summarized previous results and placed them into the taxonomy. Finally we have presented some new results as a first step in filling in the missing pieces of the race detection taxonomy (two more $\mathcal{NP}$-Hardness results and an undecidability result).

Acknowledgement

The taxonomy of races was significantly influenced by an extended email dialogue with Rob Netzer. This work was partially supported by a grant from the National Science Foundation (grant # CCR-9102635).

References

[CKS90] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *Proceedings of Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), SIGPLAN Notices*, pages 21–30, March 1990.

[CS88] D. Callahan and J. Subhlok. Static analysis of low-level synchronization. In *Proc. Workshop on Parallel and Distributed Debugging*, pages 100–111, May 1988.

[DKF93] S. K. Damodaran-Kamal and J. M Francioni. Nondeterminacy: Testing and debugging in message passing parallel programs. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 118–128, 1993.

[DS90] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 1990.

[DS91] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proc. Workshop on Parallel and Distributed Debugging*, pages 79–90, May 1991.

[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.

[HM91] D. P. Helmbold and C. E. McDowell. Computing reachable states of parallel programs (extended abstract). *SIGPLAN Notices: Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, 26(12):76–84, December 1991.

[HM93] D. P. Helmbold and C. E. McDowell. A class os synchronization operations that permit efficient race detection. In *Submitted to Supercomputing '93*, 1993.

[HMW91] D. P. Helmbold, C. E. McDowell, and J. Z. Wang. Detecting data races from sequential traces. In *Proc. of Hawaii International Conference on System Sciences*, pages 408–417, 1991.

[HMW93] D. P. Helmbold, C. E. McDowell, and J. Z. Wang. Determining possible event orders by analyzing sequential traces. *IEEE Transactions on Parallel and Distributed Systems*, 1993. Also UCSC Tech. Rep. UCSC-CRL-91-36.

[LC89] D. L. Long and L. A. Clarke. Task interaction graphs for concurrency analysis. In *Proc. 11th Int. Conf. on Software Engineering*, 1989.

[LKN93] H-I. Lu, P. N. Klein, and R. H. B. Netzer. Detecting race conditions in parallel programs that use one semaphore. Technical report, Brown Univ., 1993.

[MC91] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing '91*, pages 24–33, November 1991. Albuquerque, NM.

[MC93] John Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 129–139, 1993.

[McD89] C. E. McDowell. A practical algorithm for static analysis of parallel programs. *Journal of Parallel and Distributed Computing*, June 1989.

[Net93] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, 1993.

[NG92] R. H. B. Netzer and S. Ghosh. Efficient race condition detection for shared-memory programs with Post/Wait synchronization. In *Proc. International Conf. on Parallel Processing*, 1992.

[NM90] R. H. B. Netzer and B. P. Miller. On the complexity of event ordering for shared-memory parallel program executions. In *Proc. International Conf. on Parallel Processing*, volume II, pages 93–97, 1990.

[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.

[NR88] I. Nudler and L. Rudolph. Tools for efficient development of efficient parallel programs. In *First Israeli Conference on Computer Systems Engineering*, 1988.

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

[Tay83] R. N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *CACM*, 26(5):362–376, May 1983.