A taxonomy of race detection algorithms.

D. P. Helmbold, C. E. McDowell

UCSC-CRL-94-35
September 28, 1994

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

ABSTRACT

This paper presents a taxonomy that categorizes methods for determining event orders in executions of parallel programs. These event orderings can then be used to detect race conditions in parallel programs. The paper also shows how recent race results fit into the event ordering taxonomy, and presents some new results for previously unexamined points in the taxonomy.

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

1 Introduction

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 with extensions for some form of explicit parallelism and 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. Other programs are intended to be nondeterministic, at least at some level. 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[1] are scheduled. The formal definition of a race is given in the appendix. The appendix also contains a structural breakdown of races into four groups.

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 *do* hold between statement instances?
- What are all of the races in this program?
- Are there any races in this program?
- What shared memory locations 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 [HM93b] 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[2]. The remaining questions above are addressed by this paper. In Section 2 we present a taxonomy of event ordering approaches. Determining the possible order of events recorded or observed during the traced execution of a parallel program is important to most race detection algorithms. Section 3 places known results on event ordering into this taxonomy. This section also presents three new negative results. In Section 4 we summarize the current known algorithms that can correctly answer the question, "Are there *any* races in this program?" In Section 5 we briefly touch on the the issue of determining the conflicting accesses to shared data.

[1]For the purposes of this paper, the notions of *thread*, *task* and *process* are equivalent. We use the term "thread" throughout.

[2]Most of the definitions found in [HM93b] are included in the appendix of this paper.

2 A taxonomy of event ordering approaches

Previously, results in race detection have been classified as static analysis, post-mortem trace based, or on-the-fly. Static analysis techniques are generally applied at compile time, and do not require that the program be executed. 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 exactly which shared memory locations were accessed, as space limitations generally prevent this information from being saved for post mortem analysis).

We were unable to come up with a formal characterization of race detection algorithms that corresponded directly to static, post-mortem and on-the-fly. For example many on-the-fly algorithm can be done post mortem with at most a constant amount of memory per traced item. Of course the number of events may make this prohibitive in practice for any constant. Likewise, any post-mortem approach could be done on-the-fly with a sufficiently large buffer. It might not be able to detect races as they occur, but the point is that there is no clear dividing line between the on-the-fly techniques and the post-mortem trace analysis methods. Finally, both on-the-fly and post-mortem algorithms might incorporate some information obtained by preprocessing the program (i.e. via static analysis).

Despite their apparent differences, we will unify the static analysis, 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 constraints on the control flow constructs used by the program. The second axis identifies the kinds of synchronization used by the program. The current known results on computing ordering relationships are described in Section 3 and summarized in Table 2.1 at the end of this section. The following subsections detail the taxonomy.

2.1 Constraints on control flow

We consider three possible constraints on the control flow: no branching (i.e. all loops can be unrolled at compile time), no loops containing synchronization constructs, and unconstrained control flow. A loop that is always executed the same number of times does not present the same difficulties as a `while` loop iterating until a dynamic condition is satisfied. For the purposes of the above definition and the remainder of this section, the term “loop” applies only to those statements which cannot be unrolled at compile time.

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.

Analyzing a trace is thus analogous to analyzing a branch free program. This leads to the questions "how hard is the ordering problem for branch free programs?" and "what can we infer about the original program that contained branches?"

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 unsynchronized events in the execution. Thus only concurrent races and some general races (see appendix) can be detected in this way.

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 some 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 2.1 for an example of how races may be missed or incorrectly included.

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 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

```
 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 2.1: 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 hard-wired to false. The 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.

```
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 2.2: 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 13, this program does not contain a race.

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 2.2, 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 2.2 are executed depends on the input, but is the same on each particular input. Although some might consider this a race, we feel that this behavior is neither nondeterministic nor particularly indicative of an error. By our definitions (see appendix), the code fragment in Figure 2.2 is race-free.

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 (as in Figure 2.2). 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 (and/or recursion) 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[3] 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.

2.2 Type of synchronization

The second axis identifies the type of synchronization used by the program. At the top level we only distinguish two types of synchronization: monotonic and non-monotonic. These terms were first applied to synchronization in [HM93a]. 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 1: *A set of synchronization constructs is* **monotonic** *if every branch-free parallel program composed entirely of synchronization constructs from the set either always terminates normally (all threads complete) or always deadlocks in the same state.*

[3]Determining which pairs of events are ordered or semi-ordered is undecidable in general, see Theorem 3. However, the assumption that all combinations of branches are possible alleviates this problem.

		Exact Solution	Approximations
Branch free	Mono-tonic	• all monotonic are in $\mathcal{P}$ [HM93a], • fork/join is in $\mathcal{P}$ [MC91, DS90, NR88], • ordered critical sections are in $\mathcal{P}$ (section 3.1), • post/wait no clear is in $\mathcal{P}$ [NG92],	
	Non-mono-tonic	• single semaphore is in $\mathcal{P}$ [LKN93], • post/wait/clear is $\mathcal{NP}$-hard (Thm: 1), • semaphores are co-$\mathcal{NP}$-hard[NM90]	semaphores [HMW93]
No loops	Mono-tonic	• 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-mono-tonic	• post/wait/clear is $\mathcal{NP}$-hard (Thm: 1 or [CS88]), • semaphores are $\mathcal{NP}$-hard	fork/join [MC91, DS90, NR88], post/wait no clear [CKS90]
Unre-stricted	any	Undecidable (Thm: 3)	fork/join [MC91, DS90, NR88], ordered critical sections [Ste93], semaphores [McD89], message passing [DKF93], rendezvous [Tay83, LC89]

Table 2.1: What ordering relationships hold between statement (instances)?

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 [CS88], and semaphores [LKN93, NM90].

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

The taxonomy introduced in the previous section has six major categories {monotonically synchronized, non-monotonically synchronized} $\times$ {no branches, no loops, unrestricted}. In this section we briefly describe the known results in the various categories and provide some new results. The categories are presented in order of increasing computational complexity. Since unrestricted programs create undecidability problems regardless of the synchronization primitives used, we have combined the two unrestricted program categories.

3.1 No Branches and Monotonically Synchronized

We proved in a previous paper [HM93a] that computing the precise ordering relationships between events in branch-free monotonically synchronized programs can be done in polynomial time. (This generalizes a result of Netzer and Gosh [NG92], see Section 3.1.) 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[MC93] or recorded[Net93]. As these fork/join analysis algorithms read the trace only once and have limited storage requirements they can often be executed "on-the-fly," concurrently with the parallel program they are analyzing.

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 comparison 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 6), 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.

3.2 No Branches and Non-monotonically Synchronized

As can be seen in table 2.1, most results in this section (and sections 3.3 and 3.4) indicate that exact solutions are not tractable. The only exception that we are aware of is a recent result by Lu et.al. [LKN93] showing that the exact solution for programs using only a single semaphore can be found 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, the algorithm of Lu et.al. determines some events to be ordered that should not be (according to our definitions). This derives from their claim that if you artificially order two events and then fail to find a complete schedule, the events cannot occur in that order (and hence are always ordered in the reverse direction). It could be that two events can occur in the artificially added order, but then the program deadlocks later in its execution. Only a small change to their algorithm is needed to get the preferred result. Instead of insisting on a schedule for the entire program it is only necessary to find 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 1: *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:** If there is no truth assignment satisfying all of the clauses then the race statement in racer must execute before the race statement in main.
    1. All Ct must be posted before main executes the 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 executed.

NOTE: The operations on the XisT and XisF events are not necessary 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. The first group has been attempting to find as many races (unordered blocks) as possible, while the other has been trying to reduce the number of reported races that cannot actually occur.

3.3 No Loops and Monotonically Synchronized

Excluding arbitrary loops is necessary to avoid termination problems and the undecidability shown in section 3.5. 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.

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 3.3) 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 integer 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.

3.4 No Loops and Non-Monotonically Synchronized

As already indicated, all results in this area show that exact solutions are not tractable. Some approximation algorithms can be found in [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 1).

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.

3.5 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 3: *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 $\mathcal{I}$.

$\square$

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 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[4]. 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.

4.1 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

[4]A related problem has been observed by Netzer and Miller [NM91]. Even reporting only races that can actually occur (feasible races) can be too much. There may be a small number of important "first" races towards which the programer should be directed and then a possibly large number of other "artifact" races.

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.

4.2 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. Nondeterminism is propagated by a critical section if the occurrence of some event depends on the ordering of some critical section. This property can be conservatively checked statically in polynomial time.

In addition to needing the lock covers, this approach requires a larger history for each shared variable than the approaches described in Section 3.1. 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.[5] Dinning and Schonberg claim that the use of nested critical sections is rare resulting in very few lock covers in practice.

4.3 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.

[5]Simply checking 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.

5 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).

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.

6 Conclusion

We have presented a taxonomy of approaches for determining event orders in executions of parallel programs (which can then be used for race detection). 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 (Table 2.1). Finally we have presented some new results as a first step in filling in the missing pieces of the event ordering 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.

[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.

[HM93a] D. P. Helmbold and C. E. McDowell. A class of synchronization operations that permit efficient race detection. Technical report, U. of Calif. Santa Cruz, UCSC-CRL-93-29, 1993.

[HM93b] D. P. Helmbold and C. E. McDowell. What is a race in a program and when can we detect it? Technical report, U. of Calif. Santa Cruz, UCSC-CRL-93-30, 1993.

[HM94] D. P. Helmbold and C. E. McDowell. A taxonomy of race conditions. Technical report, in preparation, 1994.

[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.

[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.

7 Appendix

In this appendix we present our terminology for races and the ordering relationships between events. A complete discussion of this terminology can be found in [HM94]. In [HM94] we also define four additional orthogonal attributes of races: the affect on control flow (control/data race), the severity (benign/critical race), the affect on other races (depends on), and the feasibility (feasible/infeasible).

Definition 2: *An* **event** *is a contiguous sequence of one or more atomic operations executed by a single thread.*

Definition 3: *A* **simple statement** *is a syntactic structure from 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 syntactically contiguous sequence of simple statements.*

Definition 4: *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,*

- *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 action sequences by the thread.*

Definition 5: *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**[6] *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 $e_2 \not\rightarrow e_1$.*

Definition 6: *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 7: *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 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 8: *Two events are* **unordered** *if they are neither ordered nor semi-ordered.*

Definition 9: *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 10: *Two different events* **conflict** *if they represent the execution of conflicting simple statements.*

Definition 11: *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.*

Given two events that occur in some execution[7] of a program for a fixed input, in any particular execution on that same input:

- the two events will overlap or
- one will happen before the other or
- only one of the two events will occur or
- neither of the two events will occur.

The possible combinations (except neither event occurring) are shown in Table 7.1.

Definition 12 (Kinds of Races.): *The following four kinds of races are disjoint.*

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

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

[7]This need not be the same execution for both events.

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	omission 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	omission race
no	yes	no	no	y/n	not a race
no	no	yes	y/n	y/n	concurrent
no	no	no	yes	yes	omission race

Table 7.1: Summary of possible ordering relationships.

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.*

omission 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 $e_2 \rightarrow e_1$.*

Definition 13: *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 $\mathcal{I}$.*