Experience with Techniques for Refining Data Race Detection

Robert H. B. Netzer
Dept. of Computer Science
Brown University
Box 1910
Providence, RI 02912
rn@cs.brown.edu

Barton P. Miller
Computer Sciences Dept.
University of Wisconsin−Madison
1210 W. Dayton St.
Madison, WI 53706
bart@cs.wisc.edu

Appears in **Languages and Compilers for Parallel Computing**, Research Monographs in Parallel and Distributed Computing, MIT Press and Pitman Publishing, D. Gelernter, A. Nicolau, T. Gross, and D. Padua, eds., 1993.

Abstract

Dynamic data race detection is a critical part of debugging shared-memory parallel programs. The races that can be detected must be *refined* to filter out false alarms and pinpoint only those that are direct manifestations of bugs. Most race detection methods can report false alarms because of imprecise run-time information and because some races are caused by others. To overcome this problem, race refinement uses whatever run-time information is available to speculate on which of the detected races should be reported. In this paper we report on experimental tests of two refinement techniques previously developed by us. Our goal was to determine whether good refinement is possible, and how much run-time information is required. We analyzed two sets of programs, one set written by others (which they had tested and believed to be race-free but which in fact had subtle races) and another set written by us (in which we introduced more complex races). We performed race detection and refinement on executions of these programs, and recorded both the global event ordering and an approximate ordering recorded without a global clock. We found that in all the programs written by others, accurate refinement was possible even without the global ordering. In the other programs, accurate refinement was also possible but required the global ordering. These results suggest that our techniques refine races accurately, and lead a programmer directly to race-causing bugs. They also suggest that race detection methods should record only enough information necessary for good refinement (either global or approximate event orderings), and this information depends on the severity of the races being debugged.

1. Introduction

Tools for dynamically detecting data races are an important part of shared-memory parallel program debugging. To be useful, these tools should locate races accurately with few false alarms. False alarms can occur because recording every detail about the execution is impractical and because some races can be artifacts of others. In this paper we present results of experiments with two techniques, called *ordering* and *validation*[8, 9], and study their behavior as we vary the amount of run-time event ordering information provided to them. These techniques refine the set of all detected races down to a smaller set that should be investigated, mostly containing direct manifestations of bugs. We analyzed a collection of programs to see how accurately races could be refined when given the global order of events and an approximate order recorded without a global clock. Our techniques refined up to hundreds of races down to a small number that correlated well with the number of race-causing bugs. These refinements were usually made even without the global event order, and usually lead directly to the race-causing bugs in the program. These results suggest that validation and ordering are useful techniques for program debugging, and do not always require the global ordering recorded by some race detection methods.

Our interest is in dynamically locating and debugging *data races* in shared-memory parallel programs. Data races occur when concurrently executing sections of code access common shared variables, at least one of which is modified. Because data races are often considered bugs that cause unexpected nondeterminacy, debugging them is an important part of program development. However, when debugging data races, a programmer should examine only the *non-artifact* races. Although an execution may exhibit thousands of races, only a few may be directly caused by program bugs. Ideally, we wish to refine the set of all detected races down to only these bug-related races.

Our goal is to determine how accurately we can pinpoint non-artifact races. Non-artifact races are those that were unaffected by any other race. One race cannot affect another if data computed by events in the first race cannot affect the outcome of events in the second race. Locating these races is complicated because in practice only partial information about how races affect one another is available. Exact information may be costly to record and may depend on the semantics of the program. Locating non-artifacts with partial information involves estimating which races may have affected later races. This estimation becomes hard when a pair of races is "tangled", where each race has an event that precedes an event in the other, making it unclear which race affected the other. In previous work we presented two refinement techniques, *ordering* and *validation*, that conservatively locate non-artifact races[9]. Although tangled races and partial information mean that we cannot always exactly pinpoint these races, we can *order* partitions of races to identify *first partitions*, each of which is guaranteed to contain at least one non-artifact race. As a further refinement, we can then try to untangle the races in each first partition with *validation* to determine which may be of interest for debugging. In addition, the more accurately that the ordering of events is recorded, the better

---

This work was supported in part by NSF grants CCR-8815928 and CCR-9100968, ONR grant N00014-89-J-1222, and a grant from Sequent Computer Systems Inc.

these race refinements become.

Our experiments measured how accurately non-artifact races could be identified from the set of all detected races, given both the global event ordering and an approximate ordering recorded without a global clock. We implemented a race detector that traces programs on the Sequent Symmetry multiprocessor, and analyzed a collection of parallel C programs, most developed by others, and some written by us. The programs written by others had been tested by their authors and were thought to be race-free, but in fact exhibited subtle races. The programs written by us were modified to exhibit more complex races. For executions of each program, we measured the total number of data races that could be detected, and the number and size of the first partitions after ordering and validation. We found that refinement produced first partitions that contained a small number of races. For all of the programs written by others, this refinement could be performed well even without the global ordering; exact information was not necessary to refine the simple patterns of races exhibited by these programs. The other programs exhibited more complex patterns and required the global ordering. We conjecture that the subtle races that are discovered in later stages of program testing and debugging can be located even without the global ordering, while more complex races require the global ordering. Furthermore, the number of refined races matched the number of race-causing bugs well, and these bugs were easy to locate after investigating the reported races. These experiments suggest that data races can be effectively debugged by incorporating refinement into race detection. They also suggest that race detection methods should be able to record and utilize either global or approximate ordering information. For example, on-the-fly data race detection methods, which currently record a global order of individual shared-memory references, might be effective (and more efficient) even with less information.

2. Data Race Example

To illustrate the need for data race refinement, we now show how most existing dynamic data race detection methods work on an example program execution. This example shows that the set of detected races can contain artifacts, and that race refinement is necessary to filter out irrelevant information and pinpoint race-causing bugs.

Existing data race detection methods[3, 6, 7, 1, 11, 12, 5, 2, 4] work by first instrumenting the program so that information about its execution is recorded, then executing the program, and finally analyzing the collected information. Although these methods differ in how this information is collected and analyzed (*on-the-fly* and *post-mortem* approaches exist), all analyze essentially the same information about the execution: which sections of code executed, the sets of shared variables read and written by each section of code, and the relative execution order between some synchronization operations. To represent the relative ordering, a DAG is constructed (explicitly or in an encoded form), which we call the *ordering graph*, in which each node represents either a *synchronization event* (the execution instance of a single synchronization operation) or a *computation event* (the execution instance of code between two synchronization operations). Edges are added from each event to the next event in the same process, and between some pairs of synchronization events (belonging to different processes) to show their relative execution order. Although various types of synchronization have been addressed, all methods handle some form of task spawning such as **fork/join**. Edges are added from a **fork** event to the first event in each created child, and from the last event in each child to the

join event.

The crux of existing methods is the location of pairs of events that accessed common shared variables (that at least one wrote) and that either did or could have executed concurrently. All such event pairs are reported as data races. Finding events that accessed common shared variables is straightforward, since the sets of shared variables referenced by each event are recorded. To determine if two events could have executed concurrently, all existing methods conceptually analyze the ordering graph (although some methods avoid constructing the entire graph). They assume that two events could have executed concurrently if no path connects the two events. A data race is therefore reported between any two events that have no connecting path and that accessed common shared variables. We call such races the *apparent data races*.

The set of all apparent data races can include both non-artifacts and artifacts. The non-artifact races are directly caused by bugs and should be reported to the programmer, but the artifact races should be filtered out. To illustrate these points, consider the program fragment in Figure 1. This program spawns two children that execute in parallel. Each starts by performing some initial work on disjoint regions of a shared array, and then enters a loop to perform more work on the array. Inside the loop, the lower and upper bounds of an array region to operate upon are removed from a shared queue, then computation with that array region occurs. The queue starts with disjoint regions along the lower and upper boundaries of the array, which do not overlap the internal regions initially operated upon by the children. Because none of the regions worked upon overlap, a correct execution of this program should therefore exhibit no

```
fork
    work on region [10,19]
    loop
            remove (L,U) from Queue
            work on region [L,U−1]
                    of shared array
    while QueueNotEmpty

    work on region [20,29]
    loop
            remove (L,U) from Queue
            work on region [L,U−1]
                    of shared array
    while QueueNotEmpty
join
```

Initial state of Queue:
[1,10]
[30,40]

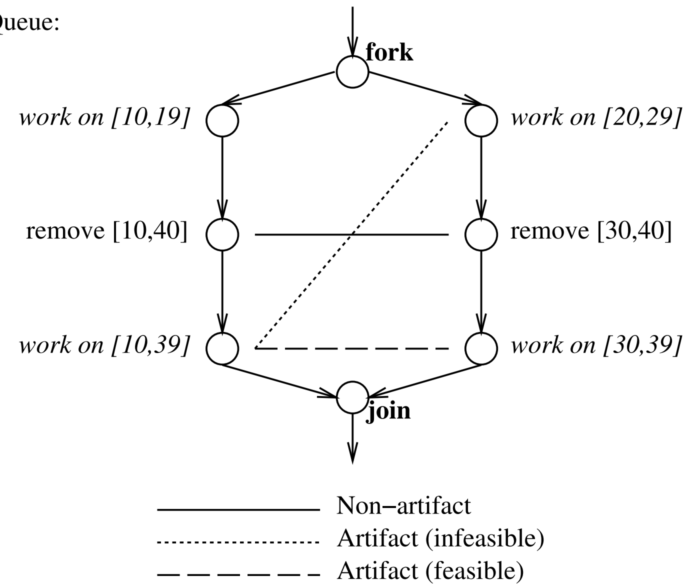

Figure 1. Example program and ordering graph (annotated with types of data races)

data races. However, assume that the "remove" operations do not properly synchronize their accesses to the shared queue. An ordering graph for one possible execution is shown (the internal lines only illustrate the data races and are not part of the graph). In this execution (during the first loop iteration) the "remove" operations execute concurrently, causing the right child to correctly remove the fourth record, but the left child to incorrectly remove the upper bounds from the last two records. The left child thus proceeds to operate (erroneously) on region [10,39].

Existing methods would report three data races. Because no paths in the graph connect any node of the left child with any node of the right child, there are two races between the *work* events (shown by the dotted and dot-dashed lines), and one race between the "remove" events (shown by the solid line). However, this set of races must be refined, since only the race between the "remove" events is not an artifact. This race was a direct cause of the bug because it (1) is *feasible*, involving events that either did execute concurrently or could have, and (2) was not performed only as a result of the outcome of another race. In contrast, the two races between the *work* events are both artifacts. The data race shown by the dotted line is infeasible, since it involves events that could never have executed concurrently. For the accesses to [10,39] and [20,29] to have executed concurrently, the left child's "remove" operation would had to have executed before the right child's "remove" operation (with which it originally overlapped). If this had happened, the erroneous record [10,40] would not have been removed (since the two "remove" operations would not overlap), and a different array region would be accessed. Although the data race shown by the dot-dashed line is feasible (it involves events that actually did execute concurrently), it is nonetheless an artifact since the access to [10,39] was a result of the preceding "remove" executing non-atomically, leaving data in an inconsistent state.

Refining the reported races to exclude the artifacts is crucial for debugging. If the array accesses had been more complex, perhaps creating other children, there may have been many nodes in the graph representing these accesses, and many data race artifacts would have been reported. Since the artifacts are not direct manifestations of program bugs but rather caused only by previous races, reporting them to the programmer obscures the location of the bug. Pinpointing race-causing bugs requires determining where the non-artifact races occurred. In practice, race artifacts are a problem because they can occur whenever shared variables are used (either directly or transitively) in conditional expressions or in expressions determining which shared locations are accessed (e.g., shared array subscripts). As this example shows, the only non-artifact data races can be located anywhere in the execution. In the next section we outline two techniques that refine races by analyzing how shared data flowed through the execution and caused events to affect one another.

3. Race Refinement with Ordering and Validation

We now briefly review our two race refinement techniques, *ordering* and *validation*. Readers familiar with this work can skip to Section 4 where we present experiments that show these techniques effectively refine races in a collection of test programs.

As illustrated in Section 2, race artifacts can stem from two sources. First, the cause of some races may be depend on the outcome of others (such as the feasible race artifact in Figure 1). To filter out these races, *ordering* computes a partial order on partitions of races, and locates partitions that are *first* in this order. Each first partition is guaranteed to contain at least race that could not have been caused by any other. In general, only partitions of races can be ordered (instead of individual races) because exactly determining how races affect one another is not always possible. Second, some races are infeasible because their events are implicitly synchronized (such as the infeasible race artifact in Figure 1). These races involve events that were not prevented from executing concurrently by explicit synchronization but were nonetheless implicitly synchronized. *Validation* attempts to determine the feasibility of races by analyzing the patterns in which they occur. Each race can be certified either as feasible or as belonging to a set of races at least one of which is feasible. Validation can refine the results of ordering by providing information about the feasibility of races inside the first partitions.

3.1. Data Race Ordering

Data race ordering uses available run-time information about how events affect one another to define an ordering among the races. In practice, only partial information is available since exact information may be costly to collect and may depend on the semantics of the program. The race ordering conservatively identified partitions of races that could not have been affected by others, and thus contain non-artifacts.

To represent how events affect one another, we use a type of data dependence called an *event-control dependence*. The purpose of this dependence is to characterize when the data computed by one event is used by another event in a way that affects its presence or outcome. The event-control dependence relation,† $\xrightarrow{\hat{\mathrm{E}}}$ is defined over the events in the execution: $a \xrightarrow{\hat{\mathrm{E}}} b$ iff a writes a shared variable whose value b uses (directly or transitively) in a way that affects the outcome of b, such as in a conditional expression, in an expression that determines which shared-memory locations are accessed (e.g., a shared-array subscript), or in a synchronization operation. This relation can be conservatively estimated from the information that data race detection methods record (discussed in Section 2). For example, if an event a writes a shared variable S, and another event b is later performed that reads S, a conservative estimate would say that $a \xrightarrow{\hat{\mathrm{E}}} b$ if it could not be determined that b did not use S in one of the above ways. In addition, since accesses to non-shared variables are not normally traced, we would (conservatively) assume that an event always event-controls later events in the same process.

Given the event-control dependences, we define a relation on the races (i.e., pairs of events) to show how they affect one another. The purpose of this relation is to show when one race may have been caused by the (possibly erroneous) data computed by another race. If we denote a data race between a and b as $\langle a, b\rangle$ the *data-race ordering* relation, $\xrightarrow{\hat{\mathrm{R}}}$, is defined as follows:

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

If $\langle a, b\rangle \xrightarrow{\hat{\mathrm{R}}} \langle c, d\rangle$ because a and b event-control c (or d), then $\langle c, d\rangle$ could have possibly been an artifact of $\langle a, b\rangle$ meaning that $\langle c, d\rangle$ may have occurred only because of erroneous data com-

† We denote relations with superscripted arrows. To be consistent with earlier work, the hats over the superscripts indicate that the relation is based on partial run-time information that can be reasonably recorded.

puted by $\langle a, b \rangle$

Because $\xrightarrow{\hat{E}}$ is based on partial and imprecise run-time information, $\xrightarrow{\hat{R}}$ may be symmetric (causing both $\langle a, b \rangle \xrightarrow{\hat{R}} \langle c, d \rangle$ and $\langle c, d \rangle \xrightarrow{\hat{R}} \langle a, b \rangle$ and cannot partially order the races to precisely determine which are non-artifacts. Sufficient information is sometimes unavailable to determine whether $\langle a, b \rangle$ is an artifact of $\langle c, d \rangle$ or *vice-versa*. To overcome this problem, we group the races into partitions so that two races belong to the same partition iff it is unknown which may have affected the other. Sufficient information then exists to partially order races in different partitions. We can thus identify the *first* partitions, which are those containing no races that may have been affected by races in any other partition. We have proven that each first partition always contains at least one non-artifact race[9].

3.2. Data Race Validation

Data race validation can sometimes further refine the first partitions identified by data race ordering. Validation attempts to determine which races in each first partition are feasible. Recall that a feasible race is one involving events that either executed concurrently or had the potential of doing so. If the first partitions contain many races, this information can be useful in determining which of these races are not artifacts. However, as we will discuss in Section 4, because ordering was often very successful in identifying small partitions, validation was not usually necessary for race debugging. We therefore just give a high-level overview of the technique.

Validation conceptually works by adding additional edges to the ordering graph. The ordering graph normally contains edges that show how events were ordered by explicit synchronization. Validation adds additional edges to show possible implicit orderings caused by shared-memory references. A doubly directed edge is added between each pair of racing events to indicate that they could have affected each other, possibly in a way that implicitly synchronizes them. Then, some races can be certified feasible by searching for strongly connected components in the resulting graph. Each connected component encompasses a set of *tangled* races, at least one of which must be feasible[8]. Races that are non-tangled (involving events belonging to no cycle) are guaranteed to be feasible. An optimization to validation involves adding edges only between racing events that event-control one another and then searching for slightly smaller cycles[9].

Given the races in each first partition, validation can sometimes certify some of them as feasible, providing useful debugging information. For example, if ordering identifies a first partition with 10 races, it is unknown which of the 10 are non-artifacts. Validation might then be able to certify that one of them is feasible, meaning that it may have computed erroneous data (as did the ‘‘remove’’ operations in Figure 1) that affected the other races in the partition.

4. Experiences with Race Refinement

We now present the results of experiments with race refinement using ordering and validation. We first outline the implementation of our data race analyzer for parallel C programs on the Sequent Symmetry multiprocessor, and then discuss our experiments. The goal of race refinement is to identify non-artifact races and pinpoint direct manifestations of bugs. We tested the effectiveness of ordering and validation when given varying amounts of run-time event ordering information. Our race detector recorded both the global event order and an approximate order recorded without a global clock. We analyzed the sensitivity to ordering information because some proposed race detection (such as the on-the-fly) methods record the global order of all shared-memory references, which might substantially slow program execution. The results of this study show how well we can perform race refinement and how much information is required.

We analyzed several programs containing data races, most previously developed by others, and some developed specifically to test our techniques in more pathological cases. The programs written by others had been extensively tested and debugged by their authors, and although they were believed to be race-free, they each exhibited subtle races. The other programs were modified by us to exhibit more complex races that might be discovered during initial stages of program testing. We measured the number and size of the first partitions that ordering identified, and the number of races in these partitions that validation further refined. We found that ordering alone produced excellent results on all of the programs written by others, even with the approximate event order. Because these programs exhibited simple, untangled race patterns, the global order was not necessary. The other programs (written by us) had tangled races and required the global event order before good refinement was possible. In either case validation provided little information. These results suggest that ordering alone is a capable debugging technique, and that race detection tools should be designed to utilize both approximate and global event order information.

4.1. Implementation

Our implementation traces shared-memory parallel C programs on the Sequent Symmetry multiprocessor and analyzes the traces post-mortem for race refinement. Tracing the program allows us to obtain a wealth of information for this study (although a production data race detector might be designed differently). Our detector instruments C programs to record run-time trace information about the shared-memory locations accessed and the order in which events execute. To obtain worst-case race refinement results, the memory locations accessed are recorded at a coarse level of granularity. To determine how much event ordering information is required, both the global order and an approximate order are simultaneously recorded.

We fix the level of granularity at which shared-memory locations are traced by viewing events as maximal. A maximal event in some process represents all computation performed between two synchronization operations in that process. For each such event, we record the sets of shared-memory addresses it reads from and writes to (but not the values read or written). This granularity represents the worst choice in terms of precision, since no information is traced about the order in which the accesses are performed inside the event. The results we obtain will be conservative; a finer granularity would only provide more information. Moreover, this choice is the same that previous post-mortem schemes have made.

We simultaneously record two different levels of event order information by maintaining clocks and timestamping synchronization operations. We record the global order of all synchronization operations by maintaining a global clock and tracing the clock at each operation. We also record an approximate order by associating a clock with each synchronization variable (such as a spin lock or semaphore) and tracing the clock at each operation on that variable. In practice, recording an approximate order has the advantage that the required instrumentation can be embedded into the implementation of the synchronization operations without introducing additional synchronization. A central bottleneck that could reduce the amount of parallelism achievable by the program is avoided. In contrast, the global order allows the relative execution order between any two events to be determined, but maintaining the global clock introduces a central bottleneck. We will later see that the global order is often unnecessary for race refinement.

4.2. Test Programs

We analyzed two sets of parallel C programs, listed in Figure 2. The first set of programs (the *unmodified* programs) were developed by colleagues for other studies and were analyzed as is, without modification. With the exception of *join* (which uses shared memory for some of its synchronization), all of these programs were previously debugged by their authors and thought to be data-race-free. However, analysis uncovered executions of each program that exhibited one or more data races. These programs represent a sample of realistic programs that might be found in practice. In contrast, the second set of programs (the *modified* programs) were originally data-race-free but were modified by us to exhibit data races. In each program, a single synchronization operation was removed, and an off-by-one error was introduced into a shared-array subscript. These modifications were made to allow more complex patterns of data races to be investigated (because the unmodified programs had been extensively debugged, their races were more subtle). These programs represent more pathological cases, which allow our techniques to be evaluated more thoroughly. Each of the test programs was run using four processors.

The test programs span a variety of applications from symbolic to numeric computation. *chol* performs Choleski factorization on sparse matrices, and was run with a 20×20 matrix as

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

Figure 2. Test programs containing data races

input. *join* implements a hash-join algorithm for a relational database, and was run with randomly generated relations containing 5000 records each with 16 attributes. *pnet* solves minimum-cost network flow problems, and was run with a 10-node network consisting of 50 arcs. *shpath* implements a parallel version of Dijkstra's shortest path algorithm, and was run on graphs with 50 nodes that were assigned random inter-node costs. *tycho* is a cache simulator, and was run with an address trace consisting of 1024 memory references. *qs* implements a quick sort, and was run on randomly generated arrays of length 3000. *workq* implements the shared work queue example of Figure 1. Each child process dequeues a record indicating a region of a shared array to work upon, forks children to perform the work, and then continues by dequeueing another record. The children simply read and write all elements of the array region. A total of 300 records were initially added to the queue.

4.3. Experimental Results

To test the effectiveness of ordering and validation, we performed experiments to determine how well these techniques refine races, and how much ordering information is required. We first measured how well races were refined when using the global event order. This measurement gives a bound on how accurately races can be refined with precise ordering information. In some cases, recording this information may be possible, allowing the bound to be achieved in practice. In other cases, this bound allows us to understand how well refinement could perform when less complete information must be recorded. We then measured how well races were refined when using only the approximate ordering. By comparing these results, we assessed the effectiveness of race refinement, and estimated how much information is required for accurate results in practice.

We found that in all of the programs written by others (which we did not modify), races were refined accurately, even without the global event order. However, in the programs for which we introduced bugs, good refinement required the global ordering. Since all of the unmodified programs had been extensively debugged and thought by their authors to be race-free, we conjecture that such programs exhibit subtle races whose detection does not require global ordering information. In contrast, we introduced multiple bugs in the other programs that caused tangled, harder to refine races. Race detection methods should thus be designed to record and utilize either global or approximate ordering information, depending on the severity of the races.

4.3.1. Refinement with the Global Ordering

The effectiveness of ordering and validation is measured by how *precisely* and *completely* they refine the detected races. Precise refinement identifies first partitions that contain few races, allowing the programmer to be lead directly to non-artifact races. Complete refinement orders most non-artifact races into the first partitions, causing few program bugs to be hidden. We next consider how well ordering and validation perform when given the global event ordering. Doing so allows us to determine whether our techniques are fundamentally useful at all, and establishes an upper bound on the diagnostic precision that can be achieved. Figures 3 and 4 summarize the results for both sets of test programs. These figures contain results of using both the global ordering (labeled with "∘") and the approximate ordering (labeled with "○").

		chol	*join*	*pnet*	*shpath*	*tycho*
Before Refinement						
Detected races		23	19	64	631	4
After Ordering						
First partitions	(◦)	2	1	1	3	2
	(○)	1	1	1	3	2
Average partition size	(◦)	2	1	1	1	1
	(○)	2	1	1	1	1
Total number of first races	(◦)	3 (13%)	1 (5%)	1 (2%)	3 (<1%)	2 (50%)
	(○)	2 (9%)	1 (5%)	1 (2%)	3 (<1%)	2 (50%)
After Validation						
Feasible first races	(◦)	3	1	1	3	2
	(○)	2	1	1	3	2

Figure 3. Results of race refinement for unmodified programs
Numbers are averages over all executions of each program
and have been rounded off to the nearest integer.
Results of analysis with global ordering (marked with "◦") and
approximate ordering (marked with "○") are shown.
Percentages indicate the fraction of detected data races.

Below we consider the precision and completeness of the results obtained with the global ordering; the next sub-section discusses the approximate ordering.

To assess the precision of ordering and validation, we measured the average size of each first partition, and the number of races in these partitions that validation determined were feasible. Recall that feasible races are those involving events that could have executed concurrently, and although they may be artifacts, feasibility provides some information useful for understanding the races. Figure 3 shows the results for the unmodified test programs. In executions of these programs, ordering located first partitions that contained only one or two races, all of which were certified as feasible by validation. Because ordering pinpointed races so accurately, validation provided no additional information. Figure 4 shows the results for the modified test programs. Ordering and validation also refined races accurately for these programs; each first partition contained only 1–4 races, all of which were validated.

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

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

		qs		*workq*	
Before Refinement					
Detected races		7522		2890	
After Ordering					
First partitions	(◦)	1		1	
	(○)	1		1	
Average partition size	(◦)	4		1	
	(○)	2604		2860	
Total number of first races	(◦)	4	(<1%)	1	(<1%)
	(○)	2604	(32%)	2860	(99%)
After Validation					
Feasible first races	(◦)	4		1	
	(○)	98		15	

Figure 4. Results of race refinement for modified programs
Numbers are averages over all executions of each program and have been rounded off to the nearest integer.
Results of analysis with global ordering (marked with "◦") and approximate ordering (marked with "○") are shown.
Percentages indicate the fraction of apparent data races.

4.3.2. Refinement with the Approximate Ordering

To determine the sensitivity of ordering and validation to the amount of run-time event ordering information, we now consider how well they refine races using the approximate ordering. As discussed earlier, the approximate ordering can be easily recorded without introducing central bottlenecks into the execution. Determining how well refinement performs can suggest how much information race detection methods should record. For executions of the unmodified programs (which had been extensively debugged), we found that the results varied little with the level of ordering information available. However, for executions of the modified programs (in which we introduced bugs), refinement performed well with the global ordering but was less accurate with the approximate ordering. The simple race patterns exhibited by the unmodified programs were easy to refine, but the modified programs exhibited more complex, harder to refine race patterns, These results suggest that in the early stages of debugging the global order may be required, but in later stages races can be refined with the lesser overhead of an approximate order.

For executions of the unmodified programs, Figure 3 shows that refinement was excellent even without the global ordering. First races were precisely located, with each first partition containing no more races than when the global ordering was used (except for *chol*, where the partitions were only slightly larger). For *chol*, one fewer first partition was identified, but its single bug was still reflected in the first races reported. The simple nature of the race patterns exhibited by these programs explains the results. In each of the programs, races were caused either by the omission of a single synchronization operation, or by an incorrect shared-array subscript. In addition, the programs used synchronization in a disciplined way that allowed these races to be easily refined. Most programs either performed no synchronization between child processes, or children synchronized on the same synchronization variable. Such synchronization patterns prevented races from becoming ‘‘tangled’’, where two races each have one event that precedes an event in the other race. In this case, it was often easy to determine how events affected each other, and the data race ordering, $\xrightarrow{\hat{R}}$, was precise.

However, for executions of the modified programs, Figure 4 shows that ordering and validation were less accurate. The first partitions contained 32–99% of the races, and although validation was able to determine that some races were feasible, this information was not very useful. The large size of the first partitions made it impossible to determine which of their races were non-artifacts. These results are explained by the tangled race patterns that occurred. We introduced bugs into each program with two perturbations, by removing a single synchronization operation and by introducing an off-by-one error into a shared-array subscript. The combination of these perturbations introduced multiple races that became tangled. Without the global event order, it was impossible to determine precisely how events affected one another. The data race ordering was thus very conservative, including almost every race in the first partitions.

Our experiments suggest that ordering and validation perform very well without the global order when simple, untangled race patterns occur, and that refining tangled races requires the global order. However, since *none* of the programs written by others exhibited tangled races, we conjecture that approximate ordering information is sufficient in many cases for race debugging. These programs had all been extensively debugged and were thought by their authors to be race-free; the more obvious races had already been eliminated. We therefore suspect that more subtle races, which survive initial debugging and testing efforts, can be easily refined.

The design of race detection methods can benefit from these results. On-the-fly methods, which currently record the global order of all shared-memory references, might be optimized to record an approximate ordering while still recording information sufficient for refinement with data race ordering.

5. Conclusions

In this paper we presented empirical evidence that accurate data race refinement for debugging is possible. Accurate refinement is necessary to direct programmers to the race sources, but refinement is complicated because only imprecise run-time information is usually available. Our two techniques, ordering and validation, deal with imprecise information by conservatively refining races. Ordering directs a programmer to partitions of races, each of which is guaranteed to contain at least one non-artifact race. Validation further refines races by attempting to determine which races in these partitions are feasible.

We tested these techniques on a collection of shared-memory parallel programs, most written by others (which were thought to be race-free but that had races) and some written by us (in which we introduced races). The programs written by others had been extensively debugged and contained subtle races, representing those that might be found in later stages of debugging and testing. The programs we wrote contained more obvious races, representing those that might be found during initial testing. We determined how well refinement could be performed for each type of program, and how much event ordering information was needed. For executions of each program, we refined races using both the global event ordering and an approximate ordering recorded without a global clock. Our results suggest how race detection methods should be designed. Currently, most methods do not refine the detected races, and some (on-the-fly) methods require collecting global ordering information, while others (post-mortem) record only approximate information.

We found that often data race ordering alone is a useful refinement technique; validation did not provide much additional information. In all of the programs written by others, ordering refined the detected races down to a handful, even without the global ordering. This accuracy was possible because the race patterns were simple and untangled. Because these programs had been tested and were thought to be race-free, we conjecture that such subtle races can usually be easily refined without global ordering information. However, in the programs written by us, the race patterns were more complex and tangled, and accurate refinement required the global order.

These results suggest that we can perform effective race refinement in practice and lead programmers directly to race-causing bugs. In the earlier stages of testing and debugging, the global event order is required, while in later stages an approximate order (which may be much cheaper to record) suffices. We therefore need race detection methods that incorporate race refinement and that can record either global or approximate event orderings, depending on the severity of the races being debugged.

6. Acknowledgements

We wish to thank Joann Ordille, Jim Larus, Mark Hill, Madhu Talluri, and Rob Welch for providing test programs.

References

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

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

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

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

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

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

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

8. Robert H. B. Netzer and Barton P. Miller, "Detecting Data Races in Parallel Program Executions," pp. 109-129 in *Advances in Languages and Compilers for Parallel Processing*, ed. A. Nicolau, D. Gelernter, T. Gross, and D. Padua, MIT Press (1991).

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

10. Robert H. B. Netzer and Barton P. Miller, "What are Race Conditions? Some Issues and Formalizations," *ACM Letters on Programming Languages and Systems* **1**(1)(March 1992).

11. Itzhak Nudler and Larry Rudolph, "Tools for the Efficient Development of Efficient Parallel Programs," *1st Israeli Conf. on Computer System Engineering*, (1988).

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