Dynamic Recognition of Synchronization Operations for Improved Data Race Detection

Chen Tian[1] Vijay Nagarajan[1] Rajiv Gupta[1] Sriraman Tallam[2]

[1]University of California at Riverside, CSE Department, Riverside, CA 92521
[2]Google Inc., 1600 Amphitheatre Parkway, Mountain View, CA 94043
{tianc,vijay,gupta}@cs.ucr.edu, tmsriram@google.com

ABSTRACT

Debugging multithreaded programs, which involves detection and identification of the cause of data races, has proved to be a hard problem. Although there has been significant amount of research [20, 28, 25, 12, 10] on this topic, prior works rely on one important assumption – the debuggers must be aware of all the synchronization operations that take place during a program run. This assumption is a significant limitation as multithreaded programs, including the popular SPLASH-2 benchmark [30], have barriers and flag synchronizations implemented in the user code. We show that the lack of knowledge of these synchronization operations leads to unnecessary reporting of numerous races. Our experiments with SPLASH-2 benchmark suite show that 12-131 distinct segments in source code, on an average, give rise to well over 4 million dynamic instances of falsely reported races for these programs. We propose a dynamic software technique that identifies the user defined synchronizations exercised during a program run. This information not only helps avoids reporting of unnecessary races, but also helps a record/replay system to speedup the replay.

Our evaluation confirms that our synchronization detector is highly accurate with no false negatives and very few false positives. Thus, reporting of nearly all unnecessary races is avoided. Finally, we show that the knowledge of synchronization operations resulted in about 23% reduction in replay time.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Testing tools*

General Terms

Algorithms, Measurement, Reliability

Keywords

data races, synchronization and infeasible races, record and replay

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA'08, July 20–24, 2008, Seattle, Washington, USA.
Copyright 2008 ACM 978-1-60558-050-0/08/07 ...$5.00.

1. INTRODUCTION

With the advent of multicores, multithreaded programming has acquired more importance. Unfortunately, debugging multithreaded programs, involving detecting and identifying the causes of data races, has proved to be a hard problem. Although, there has been significant research [20, 28, 25, 12, 10] on this topic, prior works suffer from one serious limitation: *an inherent assumption is made that the data race detectors are aware of all the synchronization operations happening in the program.* This is not a good assumption to make in general as multithreaded programs, including the popular SPLASH-2 benchmarks [30], have barriers and flag synchronizations implemented in the user code. It is unreasonable to assume that the data race detector is aware of all such user defined synchronization operations. However, the knowledge of synchronization operations is crucial to data race detection. This is due to the following two reasons.

First, synchronization operations themselves cause races in the program. These races, known as synchronization races [25], arise due to the implementations of the synchronization operations. Any *synchronization-unaware* data race detector is bound to report these as data races. Unfortunately, these are not the races that the user is interested in, as these synchronization races are benign. In our experiments with SPLASH-2 [30] programs, we found that user defined synchronization operations including barriers and flag synchronization were used across all programs in the suite and resulted in 1 to 19 distinct segments of code being present in the programs, which gave rise to numerous synchronization races being reported.

Second, a data race detector that is not aware of synchronization operations is liable to report races that are *infeasible* [25] and consequently cause more false positives. This is because shared memory accesses that are protected by synchronization operations are not actually data races; if the race detector is unaware of the synchronization operations, it will report these protected accesses as races. In our experiments, we found that this caused an additional 11 to 107 distinct segments of code being present in programs that resulted in infeasible races, i.e false positives, to be reported.

Unfortunately, identifying synchronization operations in the program is not trivial. Synchronization may happen via simple flag synchronizations or through complex barrier synchronization or spin locks. Often these synchronization operations are implemented in the program source code itself and not in libraries and there are several different algorithms to accomplish each kind of synchronization peration. Hence

identifying synchronization operations, at best, is a tedious process requiring manual source code inspection. In this paper, we propose a dynamic technique to identify user defined synchronization operations. Our technique is based on the observation that the *spinning read* is the essential part of each synchronization construct and is the major cause of synchronization races. Our software implementation of this technique is built on top of the Pin [14] dynamic instrumentation engine. Our experiments confirm that our dynamic technique is able to identify the user defined synchronization operations with no false negatives and very few false positives.

There has also been significant research on record/replay systems [4, 19, 31, 18], whose purpose is to enable *deterministic replay debugging*. We propose a scheme where our synchronization detection technique can be used to optimize replay. This is based on the observation that it is not necessary to implement the synchronization operations exactly during replay; it suffices if we just enforced those dependencies during replay, that the synchronization operations were themselves trying to enforce. Our experiments on the SPLASH-2 benchmark suite, confirm that synchronization-aware replay, on a uniprocessor, is 23% faster.

The remainder of the paper is organized as follows. Section 2 gives a discussion of benign races and closely related work. In section 3 we present our approach for dynamic detection of synchronization operations. In section 4 we describe how this information is used in data race detection and efficient replay. Section 5 presents results of our experiments. Section 6 contains a discussion of related work. We conclude in Section 7.

2. DATA RACES AND SYNCHRONIZATION

A data race occurs when two or more different threads access a shared memory location without any synchronization, and at least one of these accesses is a write access. Data races are considered to be harmful, and are known as *concurrency bugs*, if they lead to unpredictable results or cause the behavior of programs to be different from users' expectation. There has been significant recent research [20, 28, 25, 12, 10] done to help users find such data races, and thus fix concurrency bugs. But not all races reported by such tools are actually concurrency bugs. In this section, we examine how synchronization operations cause data race detectors to report false positives. Specifically, we classify the false positives reported by the race detection tools into two categories: *Intentional races due to synchronization* and *infeasible races due to missed synchronizations*. The first category refers to harmless races that are *intentionally* programmed to implement synchronization. The second category refers to shared memory accesses which are actually protected by synchronization. However, they are erroneously considered to be races by the race detector, since the race detector is *unaware* of the synchronization.

Intentional races in synchronization algorithms. In some situations, a data race is intentionally programmed to introduce non-determinism into the program. For instance, implementation of synchronization operations often introduces data races to enable competition of processors to enter a critical section, to lock a semaphore, etc. Let us consider the flag synchronization shown in Figure 1(a) taken from one of the SPLASH-2 benchmarks named *barnes*. When thread 2 starts executing line 407, it spins on variable Done(r), which can only be modified by thread 1 on line 396. Therefore thread 2 cannot proceed, until the shared variable is marked as true by thread 1. Consequently, the executions of write operation on line 396 of thread 1 and the read operation on line 407 of thread 2 form many dynamic data races. However, the purpose of these races is only to ensure execution order and thus, these races do not constitute a concurrency bug. These races that are intentionally programmed to implement synchronization constructs are also known as *synchronization races* [25]. Figure 1 (b) shows another example of synchronization race, due to a barrier implementation. Here, the while loop (line 227, statement 4) keeps spinning until all processors have reached the barrier and statements 2 and 4 of the barrier implementation race with each other.

There are several other situations in which the programmer intentionally introduces data races that are benign. In [20], Narayanasamy et al. provide a detailed categorization of these benign synchronization races.

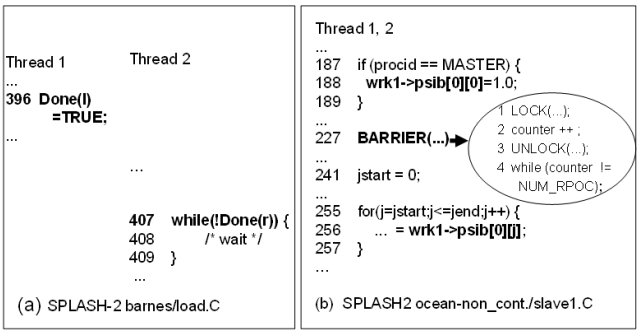

Figure 1: Flag and Barrier Synchronizations in Splash-2 Programs.

Infeasible races due to missed synchronizations. Data race detectors, due to inherent limitations of the race detection algorithms [13, 28], sometimes report races which are not actually real races. For example, let us consider Figure 1(b) which shows the barrier synchronization. Let us consider the lines 188 and 256 that respectively write to and read from the same location, when the value of j is 0. Although, each of these operations access the same shared memory location, they are not actually data races as they are protected by the barrier synchronization. However, a race detector that is unaware of the barrier synchronization, will consider these as data races. Thus, shared memory accesses that are actually protected by synchronization, but erroneously considered to be races by the race detector as it is *unaware* of the synchronization, constitute a major reason for false races [20]. These false races are also known as *infeasible* races [25].

Replay Analysis to classify reported Races. In recent work, Narayanasamy et al. [20] describe a technique to automatically classify those races that are reported by a data race detector into harmful and harmless races. Their technique works by replaying the execution twice for a reported data race, once for each possible execution order between the conflicting memory operations. It is possible for this technique to identify the synchronization races as harmless, but it in-

volves a significant offline processing. To see why, let us again consider the flag synchronization shown in Figure 1(a). Let us assume that in the actual execution of the program, the while loop (line 407) was executed n times, before line 396 was executed in another processor. This results in n dynamic races between lines 396 and 407. To confirm that this race is benign, the execution order of each of these racing instances has to be inverted and the program has to be replayed each time. If these synchronization races are identified on-line, as we do in this current work, then these races need not even be reported to the user. Thus our work is complementary to offline replay analysis of Narayanasamy et al., as far as synchronization races are concerned.

On the other hand, the replay analysis *cannot* classify the infeasible races due to the missed synchronization as harmless, in general. To see why, let us consider the barrier synchronization example shown in Figure 1(b). Recall that the read and write operations on lines 256 and 188 respectively constitute the infeasible race, when the value of the loop iterator j is 0. The replay analysis works by inverting the order of the memory accesses – line 256 will not read the value it is supposed to read, i.e. the value coming from line 188. This will cause the program to misbehave, in general. Thus, if the replayer is not aware of synchronization operations, replay analysis [20] cannot be used to correctly identify these infeasible races. In contrast the approach we present next effectively handles infeasible races.

3. DYNAMIC SYNCHRONIZATION DETECTION

As discussed above, the detection of synchronization operations is the key to filtering out the benign synchronization races and infeasible races from being reported to the user. In this section, we study the synchronization races that occur in various algorithms for implementing widely used synchronization operations like flag synchronizations, locks and barriers and formulate a generalized online algorithm for identifying these operations. Then we present a software dynamic implementation of this online algorithm.

3.1 Common Patterns in Synchronizations

We first examine the various implementations of barriers, locks, and flag synchronizations to see if there is a common pattern among them that can be used to formulate an algorithm for identifying the synchronizations.

3.1.1 Data Races in Flag Synchronizations

Flag synchronization is the simplest mechanism to synchronize two threads as it does not need any special instructions such as Test-and-Set, Compare-and-Swap, etc. Instead, its implementation only needs one shared variable, called the *flag*. When a flag synchronization is encountered in a multithreaded program, one thread executes a while loop waiting for the value of *flag* to be changed by another thread. Once the value has changed, the waiting thread is able to leave the while loop and proceed. From Figure 1(a), we can clearly see a pattern of flag synchronization. Here thread 2 performs *a spinning read* (line 407) and thread 1 performs *a remote write* (line 396) on a same shared location and these two instructions are those that cause the synchronization races.

3.1.2 Data Races in Lock Implementations

We consider different lock implementations including the test-and-test-and-set lock, which is frequently used in several thread libraries to implement a spin lock, and the state-of-the-art CLH queuing based lock. We intend to find a common pattern that spans across these lock implementations.

A classic **Test and Test-and-Set** algorithm, which is used in *pthread* library (pthread_spinlock) is shown in Figure 2(a). To acquire the lock each thread executes an atomic Test-and-Set instruction (line 3). This instruction reads and saves the value of lock and sets the lock to true. If the lock is available, then the Test-and-Set instruction returns a false, which makes a winner enter the critical section. Other threads have to spin on the lock (line 4) until there is a possibility that Test-and-Set instruction can succeed. The reason for the spinning on line 4 is to avoid executing Test-and-Set instruction repeatedly which causes cache invalidations that generate significant overhead due to cache coherence messages that are generated. For this implementation we can see that there is a spinning read at line 4 that races with a remote write at line 9. Also, we observe that the atomic instruction in line 3 simultaneously reads and writes the lock variable, and consequently races with lines 4, 7, and itself.

```
1  bool lock = false;

2  aquire_lock:
3        while (TS(lock)) {
4              while(lock) ;
5        }

6  release_lock:
7        lock = false;
```

(a) Test and Test-and-Set lock

```
1   type qnodede = record
2      prev : ^qnode
3      succ_must_wait : Boolean
4   type lock = ^qnode
5    acquire_lock (L : ^lock, I : ^qnode)
6      I->succ_must_wait := true
7      pred : ^qnode := I->prev := fetch_and_store(L, I)
8      repeat while pred->succ_must_wait
9   procedure release_lock (ref I : ^qnode)
10     pred : ^qnode := I->prev
11     I->succ_must_wait := false
12     I := pred
```

(b) CLH lock

Figure 2: Test & Test-and-Set lock and CLH lock.

CLH lock [15] is another well-studied spin lock, which is a variant of the popular MCS [17] lock. The main idea of this lock is that each processor that wants to acquire the lock is put into the queue of waiting processors. A waiting processor is made to poll the flag of the predecessor which is set by the predecessor only when it releases the lock. As we can see from Figure 2(b), this implementation also has a similar pattern, a spinning read at line 8 and a remote write at line 11 to the same shared variable *succ_must_wait*, and there is also an atomic instruction at line 7 that handles the case when multiple processors want to enter the queue at the same time.

Thus the synchronization races due to the lock synchronization follows the following pattern: *a spinning read with its corresponding write and an atomic instruction in the vicinity of the spinning read.*

3.1.3 Data Races in Barrier implementations

In this section we consider different barrier implementations including the simple centralized barrier (which is used in the source code of several SPLASH-2 benchmarks), the sense reversing barrier, and the arrival tree barrier. Here again, we find that *the spinning read along with its corresponding spin ending write* is the cause for synchronization races.

In Figure 1(b), we have shown the **centralized barrier**, where all threads except the last one, are delayed by a spinning read on variable *counter* (line 227, statement 4). In this implementation, every thread also increments variable *counter* (line 227, statement 2), which is a remote write to all earlier-arrived threads.

```
                                4  barrier:
                                5        l_sense = not l_sense;
1  shared int count = P;        6        Lock();
2  shared bool sense = TRUE;    7           count --;
3  private bool l_sense = TRUE; 8           if (count == 0) {
                                9                 count = P;
                                10                sense = l_sense;
                                11          }
                                12       Unlock();
                                13       while (sense != l_sense) ;
```

Figure 3: Sense-reversing Counter Barrier.

To make the centralized counter barrier reusable, a **sense-reversing centralized barrier**, described in [17], is shown in Figure 3. Each arriving processor decrements *count* by exclusively executing line 7 and then waits for the value of variable *sense* to be changed by the last arriving processor (line 10). Similar to the simple counter barrier, line 13 is a spinning read and line 10 is a write on variable *sense*, which is the cause of synchronization races produced due by this barrier.

```
1  typedef struct {                  For each node[pid], execute
2     bool parentsense;              11  tree_barrier:
3     bool *parentpointer;           12  while (childnotready !={false,
4     bool *childpointers[2];                    false, false , false}) ;
5     bool havechild[4];             13    childnotready = havechild;
6     bool childnotready[4];         14    *parentponter = false;
7   }TREENODE;                       19    if (pid != 0) {
                                     20       while (parentsense != sense) ;
                                     21    }
8   shared TREENODE nodes[P];        22    *childpointers [0] = sense;
9   private int  pid;                23    *childpointers [1] = sense;
10  private bool sense = TRUE;       24    sense = not sense;
```

Figure 4: Arrival Tree Barrier.

Another efficient barrier algorithm, is the **arrival tree barrier** which is described in prior work [11, 17]. Every processor is assigned a unique tree node to form two trees, an arrival tree and a wakeup tree. In the arrival tree, the arrival information is propagated from the leaves up to the root. In the wakeup tree, the wakeup information is propagated in an opposite direction, from root to the leaves. To obtain the best performance, the degree of arrival tree is set to 4 and that of wakeup tree is 2. Figure 4 shows the source code for this barrier. In the arrival tree, each processor waits for the arrival of its four children by spinning on variables *childnotready*[]. When all children have arrived, it informs its parent by updating a variable in its parent's node pointed by *parentpointer*. Thus line 12 and 14 forms a spinning-read and a remote-write pattern in the arrival tree. Similarly, line 20 is a spinning read and line 22, 23 are remote writes in the wakeup tree.

Having studied the different implementations of various synchronization operations, we find that the *spinning read and its corresponding remote write* is a common pattern among the synchronization operations.

3.2 Algorithm to Detect the Pattern

In the previous section, we found that the spinning read and its corresponding remote write is a common pattern across different implementations of various synchronization primitives. It is worth noting that it is difficult to find this pair by statically examining the code. Even if we can reduce a candidate set of spinning reads, it is not clear how the remote write can be statically identified. Hence, we explore a dynamic technique to identify this pattern. By examining the dynamic values and the addresses accessed by a load instruction, we decide on whether the load is a part of a spinning read. We identify the corresponding remote write, by identifying the store instruction from which the *last iteration* of the spinning read obtained its value.

The detailed algorithm following our approach is shown in Figure 5. We first introduce a *load table* which stores the information of 3 most recent load instructions for each thread. The information includes the *pc*, the previous address *addr* accessed by the load instruction, the previous value *val* in *addr* and a variable *counter*, which essentially maintains the current count of spin loop. The reason that we set the size of *load table* to 3 is based on the fact that it is sufficient to have as many entries as the maximum number of static loads in a spinning loop. In our experiments, we found this number to be less than 3 and so we limited the number of entries to 3. This is a reasonable limit, as in a spin loop there are typically not more than two loads, a load instruction that loads the shared memory value and possibly another load that loads the address of the shared memory location.

For every memory location accessed by a store instruction, we maintain the PC of the last store in *writepc* and the thread id of a thread that performs the last store in *writetid*. We also have a synchronization table, *syn_table*, which stores a pair of instructions: pc of spinning read as *readpc*; and the pc of the corresponding remote write as *writepc*.

With the above data structures, we now can use our algorithm to dynamically identify the synchronization pattern. The general idea of our algorithm is as follows. For each load instruction, we examine if it has been loading the same value from the address for a threshold number of times by one thread, until the value of this location is changed by another thread. It is worth noting that the threshold is a heuristic to give importance to the process of spinning and thus distinguish it from other potential situations that are not spin loops.

On every load instruction that has not been determined as a spinning read in a thread, we first examine if the information of the load has been stored in *load_table* by searching the matching PC. If not, we need to find a location in *load_table* for this load instruction. The location can be either empty (line 21) or the one that has the oldest entry (line 22-24). Then we store the information of the load into this location (line 25-29). If we can find an entry in *load_table* that matches current load (line 1), we first check if the current load accesses the same address as before (line 2-4). If so, we then compare the current value with the previous value of this address to determine if the variable *counter* reflecting the number of executions of a spinning read, should be in-

```
For each entry in load_table:
pc:   program counter of a load inst.
addr: address accessed by a load inst.
val:  value loaded by a load inst.
counter: spinning count of a load inst.
possible_spin: indicate if the counter reaches the threshold

For each load or store instruction ins:
addr: address accessed by a ins
val:  value loaded by a load inst.
cur_tid: current id of thread that performs ins

For each shadow memory location:
writetid: Thread id of a store inst.
writepc: PC of a store inst.

For each entry in syn_table:
readpc:  program counter of a load inst.
writepc: program counter of a store inst.

On executing each load instruction ins that is not in syn_table:
1  IF find the location loc for matched ins's PC in load_table
2      IF load_table[loc].addr != addr
3          GOTO reset;
4      ENDIF
5      IF load_table[loc].val = val
6          IF load_table[loc].possible_spin = 1
7              RETURN;
8          ENDIF
9          load_table[loc]. counter++;
10         IF load_table[loc]. counter = THRESHOLD
11             load_table[loc].possible_spin := 1;
12         ENDIF
13     ELSE
14         IF shadow_mem[addr].writetid = cur_tid OR
15             load_table[addr].possible_spin != 1
16             GOTO reset;
17         ENDIF
18         add (ins's PC, shadow_mem[loc].writepc)
               into syn_table and RETURN;
19     ENDIF
20 ELSE
21     loc =find an empty entry in load_table
22     IF no empty entry
23         loc = find the oldest entry in load_table;
24     ENDIF
Reset:
25     load_table[loc].pc := PC of ins ;
26     load_table[loc].addr := addr;
27     load_table[loc].val := val;
28     load_table[loc].counter:= 1;
29     load_table[loc].possible_spin:=0;
30 ENDIF

On executing each store instruction ins that is not in syn_table:
31     shadow_mem[addr].writepc := PC of ins;
32     shadow_mem[addr].writetid := cur_tid;
```

Figure 5: Dynamic Detection of Synchronization Pattern.

cremented by 1. The flag *possible_spin* is set to 1 indicating that the current load is a possible spinning read, if *counter* has reached the threshold number (line 5-12). Recall that to determine a synchronization pattern, we also need to ensure that the value of this address has to be changed by another thread. Therefore we check this condition by comparing the id of current thread with the id of the thread that performs the most recent write to this address. If they are same or the flag *possible_spin* has not been updated to 1, we reset the information about this load, expecting that the pattern can be determined subsequently (line 14-17). Otherwise, a synchronization pattern has been recognized. Thus, we store the load PC and store PC into *syn_table* and then return (line 18).

On every store instruction that has not been stored into the synchronization table, we simply record its PC and thread-id in the shadow memory corresponding to the location accessed by this store (line 31-32).

Finally, recall that *atomic* instructions were responsible for creating synchronization races in some lock implementations. We consider those atomic instructions that appear in the vicinity of a spinning read to be a potential synchronization race. Specifically, we capture those atomic instructions whose PCs are near to the PCs of spinning read instructions.

4. EXPLOITING DYNAMIC INFORMATION

Next we discuss how synchronization information is exploited to filter out the harmless data races, namely *synchronization races* and *infeasible races*. We will also present a scheme where the knowledge of synchronizations can be used to speed up the replay process.

Race detection. Significant amount of recent research [28, 25, 12, 10] has focused on data race detection. However, if these tools cannot recognize all synchronization operations in a program execution, they will report many *synchronization races* and *infeasible races*. Since now we can dynamically recognize synchronization operations with the detection technique described in the previous section, we can easily make the existing race detector stop reporting harmless races.

To filter out *infeasible races*, the existing race detectors do not have to monitor any calls to the library synchronization functions. Instead, they only need to look up our synchronization report captured by the *syn_table* to get the information about synchronization operations. Next, the race detector can apply the same technique as before (for example, computing *happens-before* partial order) to discover the harmful data races. Since our synchronization table contains more accurate synchronization knowledge, the *infeasible races* will go away automatically. To filter out *synchronization races*, the existing race detectors only need to compare the races discovered with the synchronization operations stored in *syn_table*. If there is a match, then we do not report this data race. This is because the pairs of instructions stored in the *syn_table* are actually involved in synchronization operations which give rise to synchronization races.

Thus, our synchronization detection technique can be easily used to filter out benign races. Moreover, it does not require any big changes to the existing race detection work.

Synchronization-aware record and replay. Recently there has been research on providing software support [4, 26] and hardware support [3, 31, 19, 18] for recording a program's execution. The key idea of record/replay systems is to record outcomes of all non-deterministic events, including memory races, so that the program can be replayed accurately. In other words, recording systems record all the memory dependencies exercised in the program, so that they can be enforced while replay.

A benefit of having the knowledge of synchronization races is that it can lead to optimized replay, especially if the replay happens on a uniprocessor. Here, we make a key observation that if the recorder is aware of all the synchronization races, it is possible for the replayer to replay the original program without the execution of synchronization operations. This is because, the main purpose of synchronization operations themselves in a multithreaded program is to enforce a set of memory dependencies in the program. Suppose we know a priori, the dependencies that the synchronization operations are trying to enforce, then we can modify the replayer to enforce these dependencies and consequently, there is no need to replay the synchronization operations.

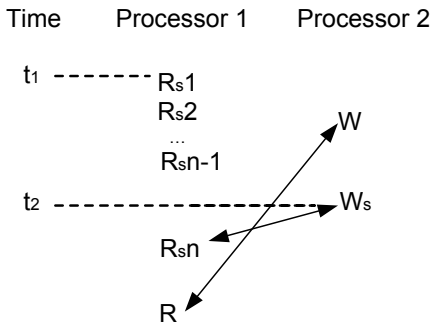

Figure 6: Synchronization Aware Record/Replay.

Consider the example in Figure 6 which shows barrier synchronization between two processors. Processor 1 reaches the barrier first (time $t1$) and spins until processor 2 also reaches the barrier (time $t2$). The spinning reads (denoted by $R_s1...R_sn$) races with the write (W_s) from the processor 2 when it eventually reaches the barrier. Now let us consider the dependence W, R, which is one of the dependencies that the barrier is actually trying to enforce. Clearly, if we are able to enforce this dependency, then we can safely remove the execution of the spinning reads from the replay. To enable this replay without synchronization, the recorder system have to be slightly modified. As far as the barrier is concerned, many recorders will record the last dependency (W_s, R_sn) along with the respective processors' instruction counts, since this is the last read is the one that is obtained from coherence reply (all other reads are local reads). When the W, R dependency is encountered, it is optimized using Netzer's transitive optimization [22] as it is implied by the W_s, R_sn dependency.

In our synchronization aware recording scheme, by the time the last read (R_sn) is executed, we would have inferred that this is a spinning read and hence we do not record this dependency. At the same time, we decrement the instruction count of spinning processor by the number of time the processor spins to enable replay without execution of spinning reads. When the W, R is now encountered, it is recorded and not optimized away as we do not record the W_s, R_sn dependence. Synchronization-aware replay, happens as usual except that we do not execute the reads that are identified as spinning reads due to synchronization.

Thus with only small changes to record and replay mechanism, the knowledge of synchronization races enables us to avoid the execution of synchronization operations during the replay.

5. EXPERIMENTAL EVALUATION

In this section, we first introduce the experimental environment and examine the number of synchronization races and infeasible races present in SPLASH-2 programs. Next we evaluate the effectiveness of our online synchronization detection algorithm in filtering out synchronization and infeasible races. We also evaluate the runtime overhead of dynamic synchronization detection. Finally, we evaluate benefits of synchronization detection in performing execution replay.

5.1 Experimental Setup

Implementation. Our synchronization detection algorithm is implemented by using the Pin [14] dynamic instrumentation framework to instrument the program as it executes. A *load_table* is implemented by a struct variable for each thread to store the information about each potential spinning read, to help us determine if it actually is one. The information includes the value, address, the execution count of each load and a flag. We also use shadow memory support [21] for maintaining information about each store instruction. Specifically, we store the PC of the store instruction and the thread-id in the shadow memory corresponding to the memory location for the store. To determine if a read is a spinning read, we examine if the value loaded (and the address) remains unchanged for a threshold number of times; and when the value changes, we ensure that it is caused by a store coming from a different thread. This store, incidentally, is also the store that races with the spinning read. If the above conditions are satisfied, we can conclude that the load PC is actually a spinning read. Note that the accesses to the shadow memory have to be done atomically, so we use *Pin Lock* to prevent any possible violations. Since the *Pin Lock* is provided by the instrumentation tool, it will not affect the detection of synchronizations in the original program. All our experiments were conducted under Fedora 4 OS running on a dual quad-core Xeon machine with 16GB memory. Each core is running at 3.0 GHz.

Programs	LOC	Input	Description
BARNES	2.0K	8192	Barnes-Hut alg.
FMM	3.2K	256	fast multipole alg.
OCEAN-1	2.6K	258×258	non-contiguous
OCEAN-2	4.0K	258×258	contiguous
RADIOSITY	8.2K	batch	diffuse radiosity alg.
RAYTRACE	6.1K	tea	ray tracing alg.
VOLREND	2.5K	head -a	ray casting alg.
WATER-1	1.2K	512	nsquared
WATER-2	1.6K	512	spatial

Table 1: SPLASH-2 Benchmarks Description.

Programs	Realistic				Optimistic				Pessimistic			
	Sync.		Infeasible		Sync.		Infeasible		Sync.		Infeasible	
	Dist.	Dyn.	Dist.	Dyn.	Dist.	Dyn.	Dist.	Dyn.	Dist.	Dyn.	Dist.	Dyn.
BARNES	7	6.5M	21	102.5K	1	1.6K	6	7.5K	82	17.6M	2.5M	1121.5M
FMM	14	4.2M	107	7.7K	5	7.3K	10	1.8K	147	6.1M	0.2M	176.7M
OCEAN-1	19	5.9M	40	75.1K	0	0	0	0	275	8.5M	3.9M	5074.1M
OCEAN-2	18	6.2M	57	82.4K	0	0	0	0	270	7.1M	17.8M	5871.1M
RADI.	3	51.6K	32	30.3K	1	1.2K	13	2.7K	55	0.6M	11.3M	8442.3M
RAYTRC	1	15.1K	11	7.8K	0	0	0	0	53	0.3M	1.1M	5162.7M
VOLREND	7	578.9K	68	40.8K	1	1.1K	7	1.3K	93	1.5M	0.5M	371.4M
WATER-1	7	17.3M	62	18.21K	0	0	0	0	47	19.7M	4.1M	352.16M
WATER-2	9	1.2M	53	156.75K	0	0	0	0	98	2.4M	1.2M	288.7M

Table 2: Synchronization Races and Infeasible Races.

Benchmarks. In our experiments, we choose the SPLASH-2 benchmark [30] suite as it is widely used to facilitate the study of shared memory multiprocessors. Table 1 shows the name, number of lines, input, and brief description for each program used in our experiments. These programs have different types of synchronizations (flag synchronizations, locks, and barriers) most of which are defined by PARMACS Macro constructs [2]. These constructs are different from library routines. Unlike the implementations of library routines, during compilation, the implementations of the constructs will cause their code to be inlined into the user code.

Using PARMACS constructs in source code gives programmers the flexibility to choose different implementations of synchronizations according to their needs like performance, portability etc. For example, the programmer may use an available implementation of a synchronization operation from a library routine (e.g., *pthread* library) or the programmer may develop his or her own implementation. For instance, in some SPLASH-2 benchmarks, we observe that programmers have implemented their own flag synchronizations in the user code (Figure 1(a)) rather than using PARMACS Macro construct. Even for Macro constructs, library routines may not provide the implementations that programmers want. For example, counter based barrier using spin lock is not available in *pthread* library. Therefore, programmers have to implement their own algorithms for such constructs (Figure 1(b)).

We studied the benchmarks by identifying occurrences of synchronization and infeasible races in these benchmarks. To carry out this study we applied a race detection tool, which computes the happens-before partial order based on the knowledge of synchronization operations in the libraries. All synchronizations implemented in the user code are ignored. The results of this study depend upon what synchronization operations are used from libraries. While the *flag* synchronization must always be expressed in user code, in general, the LOCK/UNLOCK and BARRIER operations could be either expressed in user code or used from an available library. For SPLASH-2 benchmarks we consider three different scenarios:

(Realistic) Given the availability of the *pthread* library on Fedora 4, spin lock implementation is available in form of two library routines, *pthread_spin_lock* and *pthread_spin_unlock*. However, BARRIER operation must be implemented in the user code;

(Optimistic) We assume that LOCK/UNLOCK and BARRIER operations are available as library routines. To study this scenario we compiled our own implementation of BARRIER into a library file; and

(Pessimistic) We assume that LOCK/UNLOCK and BARRIER operations are implemented in user code.

Table 2 shows the number of synchronization races and infeasible races found in the scenarios. Note that we create 4 threads for each benchmark in our experiments. For each scenario, 4 columns giving the number of distinct synchronization races, dynamic synchronization race instances, distinct infeasible races and dynamic infeasible race instances.

For the *realistic* scenario (columns 2-5), barriers and flag synchronizations contribute to the synchronization races. The number of observed distinct races in synchronization and their corresponding dynamic instances are shown in column 2 and 3, which varies from 1-19 and 15.1K - 17.3M respectively. Since these synchronizations cannot be captured by race detector, 11-107 distinct infeasible races will be reported. These correspond to thousands of dynamic infeasible races that are reported (column 4-5). Thus we can see that user defined synchronization operations cause a significant number of distinct false positives (12-131) to be reported.

For the *pessimistic* scenario (columns 10-13) these numbers are even higher. As we can see in Table 2, 55-275 distinct synchronization race instances contribute to millions of dynamic synchronization races. In addition, millions of distinct and dynamic infeasible races are also reported. To perform this experiment we implemented a Test and Test-and-Set lock via an atomic decrement x86 instruction for LOCK/UNLOCK Macro constructs, and a counter-based sense-reversing barrier for BARRIER Macro constructs.

Finally, as expected, for the optimistic scenario (columns 6-9), the number of distinct synchronization and infeasible races is small and these gave rise to few thousand dynamic races. It should be noted that to conduct this experiment we had to disassemble our own library code and hard-code the instruction addresses into the race detector. Thus, while this approach gives good results, it places a great deal of burden on the programmer.

5.2 Filtering Data Races

To evaluate the effectiveness of our synchronization detection algorithm, we added our software implementation into

the happens-before race detector and conducted the experiments on all three scenarios described above. This experiment yielded the following key results.

Robustness. First, the results were identical for all three scenarios. In other words, our synchronization detection based approach is highly robust as it is equally effective in filtering out synchronization and infeasible races in varying scenarios.

Programs	Sync. Races		Infeasible Races	
	Distinct	Dynamic	Distinct	Dynamic
BARNES	0	0	0	0
FMM	4	0.4K	4	0.6K
OCEAN-1	0	0	0	0
OCEAN-2	0	0	0	0
RADI.	0	0	0	0
RAYTRC	0	0	0	0
VOLRND	0	0	0	0
WATER-1	0	0	0	0
WATER-2	0	0	0	0

Table 3: Synchronization Races and Infeasible Races with Synchronization Detection.

Filtering effectiveness. Second, nearly all the number of synchronization and infeasible races were successfully filtered out using the dynamically detected synchronization operations. Let us examine the data in Table 3 in more detail. With the exception of *FMM*, no synchronization races or infeasible races are reported in any of the benchmarks. In *FMM*, we found 4 synchronization races and 4 infeasible races. The dynamic numbers are 0.4K and 0.6K respectively. The reason we report these races is that our algorithm missed 4 flag synchronizations; 2 in the function *VListInteration* and 2 in the function *WListInteraction* in the file *interaction.C*. We investigated why we missed these synchronizations and found the reason to be following. In each of the flag synchronizations we missed, the *spinning read was executed exactly once*. In other words, the spinning read did not actually experience any spin. We also measured the effect of missing the synchronizations and found that this caused 4 additional distinct infeasible races to be reported.

False positives and negatives. Finally, it is worth noting that if our synchronization detector does not *miss* any synchronization operation, the false positives caused due to synchronization will be eliminated. At the same time, if our synchronization detector falsely considers some other program operations as synchronization operations, this can potentially lead to a real race being considered as a synchronization race and hence cause *false negatives*. However, the latter situation does not arise. This is because the pattern we captured, namely a spinning read and a corresponding write, is the essence of synchronization semantic. Accesses to a shared memory that is not used for synchronization will not experience any spin.

Selection of Threshold Value. In the above experiment the threshold value used by the algorithm was set to 10. We also varied the threshold value to study its impact on the effectiveness of our approach. In Table 4, we show the number of distinct synchronizations reported by our algorithm under the realistic scenario with *threshold* values of 10, 100, and 500. Since the threshold number is our heuristic used in the algorithm to quantify the number of spins, the higher this threshold, the higher the chance that we may miss synchronization races and vice versa. The actual number of synchronizations are also presented for the purpose of comparison in column "Actual". From this table, we can see that setting the threshold to 500 causes some synchronization operations to escape detection. Then we reduced the threshold to 10 and found that we were able to find most of the synchronizations, missing only 4 in *FMM* which was discussed earlier. We could not observe an increase in the detected synchronizations, if we lowered the threshold any further. Thus, the synchronization detector works well when the threshold is set to 10. To evaluate the sensitivity, we also considered a threshold of 100 and found that the results were same with 100 as with the threshold of 10.

Programs	Actual	T = 500	T = 10	T = 100
BARNES	7	4	7	7
FMM	14	7	10	10
OCEAN-1	19	19	19	19
OCEAN-2	18	16	18	18
RADIOSITY	3	2	3	3
RAYTRACE	1	1	1	1
VOLREND	7	5	7	7
WATER-1	7	7	7	7
WATER-2	9	7	9	9

Table 4: Number of User Defined Synchronizations in SPLASH-2 Benchmarks.

In conclusion, from the above experiment, we are able to observe the following. First, there was no situation where we falsely considered some other program operation to be a synchronization operation. Hence our synchronization detector did not cause any false negatives in race detection. Second, the number of missed synchronizations depended on the threshold with more synchronizations missed when the threshold was set higher. Third, even if the threshold value was set to be low, we still can miss synchronizations if there is no spin experienced.

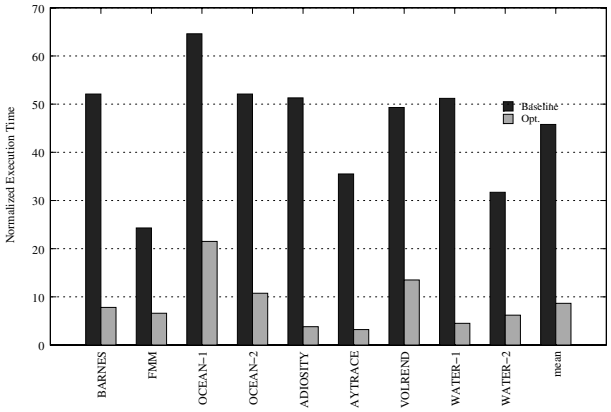

Figure 7: Overhead of Synchronization Detection.

Overhead of Synchronization Detection. We also studied the overhead of our technique. Figure 7 shows the perfor-

mance overhead involved in our implementations. In this experiment the realistic scenario is used. Note that in our *Baseline* implementation we instrument every load and store instruction, while in our optimized version (*Opt.*), we only instrument the specific loads and stores that are *likely* to be spinning reads and writes. Specifically, we instrument only those loads that are within a spin loop and those stores that do not operate on the stack. We identify potential spin loops by first identifying branch instructions that branch backwards in code which contain just loads and compare instructions. As we can see from Figure 7 our average baseline overhead is a slowdown by a factor of 45, while our optimization is able to significantly reduce the overhead to a slowdown factor of 9.

5.3 Synchronization-Aware Replay

In this experiment, we wanted to measure the savings of *synchronization-aware* replay when the replay is carried out on a uniprocessor. Recall that if we are aware of synchronization operations, we do not need to faithfully re-execute the synchronization events during replay; it suffices if we just enforced the appropriate dependencies during replay. We have not actually implemented a replayer system, but we measured the time spent on synchronization operations for each of the programs. As this is a measure of time we can save during replay, this percentage is a good indicator of the speed up that can be achieved during replay. As we can see from Figure 8, the savings varies from 7% to 48% and the average savings is 23%.

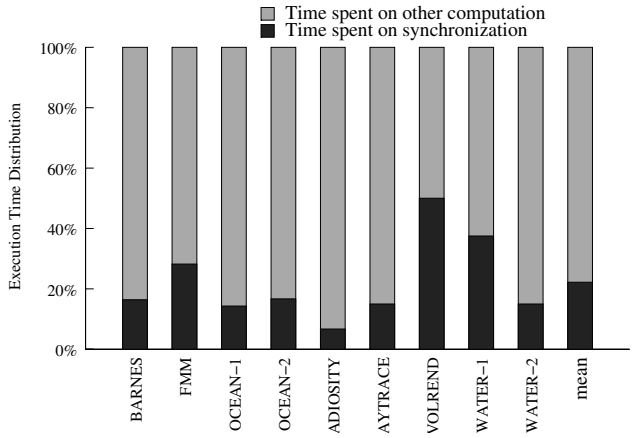

Figure 8: Savings during Replay

6. RELATED WORK

Software based race detection techniques can be broadly classified into static and dynamic approaches. Static analysis techniques [27, 5, 9] utilize type analysis and knowledge of synchronization operations to identify data races. The important limitation of static analysis approaches is their accuracy in terms of false positives and false negatives reported [20].

Dynamic race detection techniques can be broadly divided into three categories, those using *happens-before* algorithm, those using *lockset* algorithm, and those combining *happens-before* and *lockset* algorithms. The *happens-before* based techniques use the execution order and the knowledge of synchronization events to compute *happens-before* partial order, which was proposed by Lamport in [13]. If two accesses from different threads to a shared location are not ordered by the *happens-before* relation, a data race will be reported. Race detection techniques used in [22, 6, 1, 7, 25, 16] are in this category.

Race detectors using *lockset* algorithm basically focus on lock-based multithreaded programs. The idea first proposed by Savage et al. in [28] is to verify that every shared-memory access is associated with a correct locking behavior. To avoid false positives due to some common programming practices, such as using read-only shared variables after they are initialized, many improved *lockset* algorithms that basically track the states of each memory location are proposed and utilized in recent works [28, 12, 10, 29].

Both *happens-before* and *lockset* algorithms have their own drawbacks for race detection. *Happens-before* algorithm is hard to implement in software and more likely to miss some potential races (false negatives). On the other hand, *lockset* method is efficient to implement but it usually gives too many false alarms. Therefore, many works attempt to combine these two algorithms in some way in order to overcome the drawbacks. The hybrid techniques are discussed in [8, 23, 32, 24].

To report data races as accurately as possible, all of the above approaches need exact synchronization information regardless of whether synchronizations are implemented in libraries or user code. Unfortunately, when monitoring a program, none of those approaches attempt to recognize every synchronization event, especially those implemented in user code. They either make an assumption that all synchronizations are in the library or ignore synchronizations defined by user. In reality, however, programmers may use different synchronization implementations according to their demands as shown in SPLASH-2 benchmark suite rather than using library implementations. Thus, when those detectors are applied, many synchronization races and infeasible races will be falsely reported to debuggers, which may consume vast amounts of time. On the contrary, our technique is effective in avoiding the reporting of benign synchonization or infeasible races by automatically identifying the synchronizations no matter how the synchronizations are implemented. Hence, our current work is a complement to prior work.

Narayanasamy et. al. [20] focus on classifying the reported races into benign and harmful races using offline replay analysis. As discussed earlier, the above technique can handle synchronization races but it cannot handle infeasible races. Compared to the above work, our approach can find synchronization races on-the-fly. It can also avoid reporting infeasible races when working together with other race detection techniques.

7. CONCLUSION

In this paper we first discussed how lack of knowledge of user defined synchronizations can lead to a lot of false positives in race detection tools. We then proposed a technique to dynamically identify synchronization operations that are exercised in a program run. This information was demonstrated to be highly effective in filtering out synchronization and infeasible races. Furthermore, our technique can be

easily exploited by a record/replay system to significantly speedup the replay. A scheme of using the knowledge of synchronizations to optimize replay is proposed.

Our evaluation confirms that our synchronization detector is highly accurate with no false negatives and very few false positives. We also show that, on average, our optimized software implementation causes a 9 fold slowdown in program execution. Finally, we showed that the knowledge of synchronization operations resulted in about 23% reduction in replay time.

Acknowledgements. This work is supported by NSF grants CNS-0810906, CNS-0751961, CCF-0753470, and CNS-0751949 to the University of California, Riverside.

8. REFERENCES

[1] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, volume 19, pages 234–243, New York, NY, 1991.

[2] E. Artiaga, N. Navarro, X. Martorell, Y. Becerra, M. Gil, and A. Serra. Experiences on the implementation of parmacs macros using different multiprocessor operating system interfaces.

[3] D. F. Bacon and S. C. Goldstein. Hardware-assisted replay of multiprocessor programs. In *PADD '91: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 194–206, New York, NY, USA, 1991. ACM Press.

[4] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 154–163, New York, NY, USA, 2006. ACM.

[5] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–230, New York, NY, USA, 2002.

[6] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Trans. Program. Lang. Syst.*, 13(4):491–530, 1991.

[7] M. Christiaens and K. D. Bosschere. Trade, a topological approach to on-the-fly race detection in java programs. In *JVM'01: Proceedings of the JavaTM Virtual Machine Research and Technology Symposium on JavaTM Virtual Machine Research and Technology Symposium*, pages 15–15, Berkeley, CA, USA, 2001. USENIX Association.

[8] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *PADD '91: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 85–96, New York, NY, USA, 1991. ACM Press.

[9] C. Flanagan and S. N. Freund. Type-based race detection for Java. *ACM SIGPLAN Notices*, 35(5):219–232, 2000.

[10] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, 39(1):256–267, 2004.

[11] R. Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. In *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 54–63, New York, NY, USA, 1989. ACM Press.

[12] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing data races on-the-fly. In *PADTAD '07: Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging*, pages 54–64, New York, NY, USA, 2007. ACM Press.

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

[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, 2005.

[15] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. pages 165–171.

[16] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 24–33, New York, NY, USA, 1991. ACM.

[17] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[18] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 229–240, New York, NY, USA, 2006. ACM Press.

[19] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 284–295, Washington, DC, USA, 2005. IEEE Computer Society.

[20] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data racesallusing replay analysis. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 22–31, New York, NY, 2007.

[21] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *VEE '07: Proceedings of the 3rd international conference on*

Virtual execution environments, pages 65–74, New York, NY, USA, 2007. ACM Press.

[22] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *PADD '93: Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*, pages 1–11, New York, NY, USA, 1993. ACM Press.

[23] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–178, New York, NY, USA, 2003. ACM Press.

[24] V. Project. Helgrind, a data race detector. In *http://valgrind.org/docs/manual/hg-manual.html*, 2003.

[25] M. Ronsse and K. D. Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[26] Y. Saito. Jockey: a user-space library for record-replay debugging. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 69–76, New York, NY, USA, 2005. ACM.

[27] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 83–94, New York, NY, USA, 2005. ACM Press.

[28] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 27–37, New York, NY, USA, 1997. ACM Press.

[29] C. von Praun and T. R. Gross. Object race detection. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 70–82, New York, NY, USA, 2001. ACM.

[30] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, 1995.

[31] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 122–135, New York, NY, USA, 2003. ACM Press.

[32] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 221–234, New York, NY, USA, 2005. ACM Press.