Data Races vs. Data Race Bugs: Telling the Difference with Portend

Baris Kasikci, Cristian Zamfir, and George Candea
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
{baris.kasikci,cristian.zamfir,george.candea}@epfl.ch

Abstract

Even though most data races are harmless, the harmful ones are at the heart of some of the worst concurrency bugs. Alas, spotting just the harmful data races in programs is like finding a needle in a haystack: 76%-90% of the true data races reported by state-of-the-art race detectors turn out to be harmless [45].

We present Portend, a tool that not only detects races but also automatically classifies them based on their potential consequences: Could they lead to crashes or hangs? Could their effects be visible outside the program? Are they harmless? Our proposed technique achieves high accuracy by efficiently analyzing multiple paths and multiple thread schedules in combination, and by performing symbolic comparison between program outputs.

We ran Portend on 7 real-world applications: it detected 93 true data races and correctly classified 92 of them, with no human effort. 6 of them are harmful races. Portend's classification accuracy is up to 89% higher than that of existing tools, and it produces easy-to-understand evidence of the consequences of harmful races, thus both proving their harmfulness and making debugging easier. We envision Portend being used for testing and debugging, as well as for automatically triaging bug reports.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Diagnostics

General Terms Reliability, Verification, Performance, Security

Keywords Data races, Concurrency, Triage, Testing

1. Introduction

A data race occurs when two threads access a shared memory location, at least one of the two accesses is a write, and the relative ordering of the two accesses is not enforced using synchronization primitives, like mutexes. Thus, the racing memory accesses may occur in any order or even simultaneously on a multi-processor. Data races are some of the worst concurrency bugs, even leading to the loss of human lives [35] and causing massive material losses [54]. As programs become increasingly parallel, we expect the number of data races they contain to increase; as hardware becomes increasingly parallel, we expect an increased probability that both orderings of any given race get exercised during normal executions.

Yet, eliminating all data races still appears impractical. First, synchronizing all racing memory accesses would introduce performance overheads that may be considered unacceptable. For example, for the last year, developers have not fixed a race in memcached that can lead to lost updates—ultimately finding an alternate solution—because it leads to a 7% drop in throughput [41]. Performance implications led to 23 data races in Internet Explorer and Windows Vista being purposely left unfixed [45]. Similarly, several races have been left unfixed in the Windows kernel, because fixing those races did not justify the associated costs [29].

Another reason why data races go unfixed is that 76%–90% of data races are actually harmless [15, 29, 45, 58]—*harmless races* do not affect program correctness, either fortuitously or by design, while *harmful races* lead to crashes, hangs, resource leaks, even memory corruption or silent data loss. Deciding whether a race is harmful or not involves a lot of human labor (with industrial practitioners reporting that it can take days, even weeks [23]), so time-pressed developers may not even attempt this high-investment/low-return activity. On top of all this, static race detectors can have high false positive rates (e.g., 84% of races reported by [58] were not true races), further disincentivizing developers. Alas, automated classifiers [27, 29, 45, 55] are often inaccurate (e.g., [45] reports a 74% false positive rate in classifying harmful races).

Given the large number of data race reports (e.g., Google's Thread Sanitizer [30] reports over 1,000 unique data races in Firefox when the browser starts up and loads http://bbc.co.uk), we argue that data race detectors should also triage reported data races based on the consequences they could have in future executions. This way, developers are better informed and can fix the critical bugs first. A race detector should be capable of inferring the possible consequences of a reported race: is it a false positive, a harmful race, or a harmless race left in the code perhaps for performance reasons? To our knowledge, no data race detector can do this soundly.

We propose Portend, a technique and tool that detects data races and, based on an analysis of the code, infers the races' potential consequences and automatically classifies them into four categories: "specification violated", "single ordering", "output differs", and "k-witness harmless". For harmful races, it produces a replayable trace that demonstrates the harmful effect, making it easy on the developer to fix the bug.

Portend operates on binaries, not on source code (more specifically on LLVM [33] bitcode obtained from a compiler or from an x86-to-LLVM translator like RevGen [11]). Therefore it can effectively classify both source-code-level races and assembly-level races that are not forbidden by any language-specific memory model (e.g., C [26] and C++ [25]).

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.

ASPLOS'12, March 3–7, 2012, London, England, UK.
Copyright © 2012 ACM 978-1-4503-0759-8/12/03…$10.00

We applied Portend to 93 data race reports from 7 real-world applications—it classified 99% of the detected data races accurately in less than 5 minutes per race on average. Compared to state-of-the-art race classifiers, Portend is up to 89% more accurate in predicting the consequences of data races. This improvement comes from Portend's ability to perform multi-path and multi-thread schedule analysis, as well as Portend's fine grained classification scheme. We found not only that multi-path multi-schedule analysis is critical for high accuracy, but also that the "post-race state comparison" approach used in state-of-the-art classifiers does not work well on our real-world programs, despite being perfect on simple microbenchmarks (§5.2).

This paper makes three contributions:

- A ***technique for predicting the consequences of data races*** that combines multi-path and multi-schedule analysis with symbolic program-output comparison to achieve high accuracy in consequence prediction, and thus classification of data races according to their severity;
- A four-category ***taxonomy of data races*** that is finer grain, more precise and, we believe, more useful than what has been employed by the state of the art;
- Portend, a ***practical dynamic data race detector and classifier*** that implements this technique and taxonomy, and a demonstration that it works well for C and C++ software.

Portend presents two main benefits: First, it can triage data race reports automatically and prioritize them based on their likely severity, allowing developers to focus on the most important races first. Second, Portend's automated consequence prediction can be used to double check developers' manual assessment of data races; e.g., Portend may prove that a race that was deemed harmless and left in the code for performance reasons is actually harmful. Portend does not encourage sloppy or fragile code by entitling developers to ignore seemingly benign races, but rather it improves programmers' productivity so they can correctly fix as many important data race bugs as possible.

In the rest of the paper, we describe our proposed classification scheme (§2), Portend's design (§3) and implementation (§4), an evaluation on real-world applications and benchmarks (§5), discuss limitations (§6), related work (§7), and then we conclude (§8).

2. Data Race Classification

We now describe the key existing approaches to race classification (§2.1) and the challenges they face, the idea behind Portend (§2.2), and our proposed classification scheme (§2.3).

2.1 Background and Challenges

We are aware of three main approaches for classifying data races into harmful vs. harmless: heuristic classification, replay and compare, and ad-hoc synchronization identification.

Heuristic classification relies on recognizing specific patterns that correspond to harmless races. For instance, DataCollider [29] prunes data race reports that appear to correspond to updates of statistics counters, read-write conflicts involving different bits of the same memory word, or that involve variables known to developers to have intentional races (e.g., a "current time" variable is read by many threads while being updated by the timer interrupt). Like any heuristic approach, such pruning can lead to both false positives and false negatives, depending on how well suited the heuristics are for the target system's code base. For instance, updates on a statistics counter might be harmless for the cases investigated by DataCollider, but if a counter gathers critical statistics related to resource consumption in a language runtime, classifying a race on such a counter as benign may not be correct.

Replay-based classification [45] starts from an execution that experienced one ordering of the racing accesses ("primary") and re-runs it while enforcing the other ordering ("alternate"). This approach compares the state of registers and memory immediately after the race in the primary and alternate interleavings. If differences are found, the race is deemed likely to be harmful, otherwise likely to be harmless.

This approach faces three challenges: First, differences in the memory and/or register state of the primary vs. alternate executions may not necessarily lead to a violation of the program specification (e.g., otherwise-identical objects may be merely residing at different addresses in the heap). Second, the absence of a state difference may be merely an artifact of the inputs provided to the program, and for some other inputs the states might actually differ. Third, it may be impossible to pursue the alternate interleaving, e.g., because ad-hoc synchronization enforces a specific ordering: instead of employing synchronization primitives like `mutex_lock` or `cond_wait`, ad-hoc synchronization uses programmer-defined synchronization constructs that often rely on loops to synchronize threads via a shared variable [60]. In this case, replay-based classification [45] conservatively classifies the race as likely to be harmful. These three challenges cause replay based classification to have a 74% false positive rate in classifying harmful races [45].

Identification of ad-hoc synchronization helps improve accuracy of race classification [27, 55]. If a shared memory access is found to be protected via ad-hoc synchronization, then it is classified as harmless, because its accesses can occur in only one order. Existing approaches use heuristics or dynamic instrumentation to detect such cases with misclassification rates as high as 50% [27]. Pruning race reports due to ad-hoc synchronization is effective in reducing the reported harmless races, but it is still insufficient to correctly classify all races, as there are many other kinds of harmless races [29, 45].

2.2 Portend's Approach

Portend addresses two challenges left unsolved by prior work: (1) how to accurately distinguish harmful state differences from harmless ones, and (2) how to identify harmful races even when the state of the primary and alternate executions do not immediately differ. We observe that it is not enough to reason about one primary and one alternate interleaving, but we must also reason about the *different paths* that could be followed before/after the race in the program in each interleaving, as well as about the *different schedules* the threads may experience before/after the race in the different interleavings. This is because a seemingly harmless race along one path might end up being harmful along another. To extrapolate program state comparisons to other possible inputs, it is not enough to look at the explicit state—what we must compare are the *constraints placed on that state* by the primary and alternate executions. These observations motivate Portend's combination of multi-path and multi-schedule analysis with symbolic output comparison (§3). Thus, instead of employing heuristic-based classification as in previous work [27, 29, 55], Portend symbolically executes the program to reason precisely about the possible consequences of races.

2.3 A New Classification Scheme

A simple harmless vs. harmful classification scheme is undecidable in general (as will be explained below), so prior work typically resorts to "likely harmless" and/or "likely harmful." Alas, we find that, in practice, this is less helpful than it seems (§5). We therefore propose a new scheme that is more precise.

Note that there is a distinction between false positives and harmless races: when a purported race is not a true race, we say it is a *false positive*. Static [15], lockset [49], and hybrid [47] data race detectors typically report false positives. Detectors based on happens-before relationship [50] do not report false positives unless applications employ unrecognized happens-before relationships, such as ad-hoc synchronization. A false positive is clearly harmless (since it is not a race to begin with), but not the other way around.

Our proposed scheme classifies the true races into four categories: "spec violated", "output differs", "k-witness harmless", and "single ordering". We illustrate this taxonomy in Fig. 1.

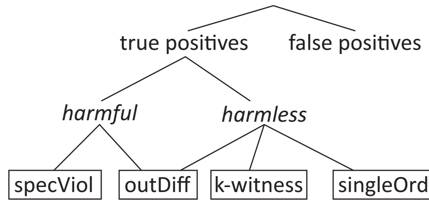

Figure 1. Portend taxonomy of data races.

"Spec violated" corresponds to races for which at least one ordering of the racing accesses leads to a violation of the program's specification. These are, by definition, harmful. For example, races that lead to crashes or deadlocks are generally accepted to violate the specification of any program; we refer to these as "basic" specification violations. Higher level program semantics could also be violated, such as the number of objects in a heap exceeding some bound, or a checksum being inconsistent with the checksummed data. Such semantic properties must be provided as explicit predicates to Portend, or be embedded as assert statements in the code.

"Output differs" is the set of races for which the two orderings of the racing accesses can lead to the program generating different outputs, thus making the output depend on scheduling. Such races are often considered harmful: one of those outputs is likely "the incorrect" one. However, "output differs" races can also be harmless, whether intentional or not. For example, a debug statement that prints the ordering of the racing memory accesses is intentionally order-dependent, thus an intentional harmless race. An example of an unintentional harmless race is one in which one ordering of the accesses may result in a duplicated syslog entry—while technically a violation of any reasonable logging specification, a developer may decide that such a benign consequence makes the race not worth fixing, especially if she faces the risk of introducing new bugs or degrading performance when fixing the bug.

As with all high level program semantics, automated tools *cannot* decide on their own whether an output difference violates some non-explicit specification or not. Moreover, it might even be subjective, depending on which developer is asked. It is for this reason that we created the "output differs" class of races. The key is to provide developers a clear characterization of the output difference, so they can easily decide whether that difference matters.

"K-witness harmless" are races for which the harmless classification is performed with some quantitative level of confidence: the higher the k, the higher the confidence. Such races are guaranteed to be harmless for at least k combinations of paths and schedules; this guarantee can be as strong as covering a virtually infinite input space (e.g., a developer may be interested in whether the race is harmless for all positive inputs, not caring about what happens for zero or negative inputs). Portend achieves this using a symbolic execution engine [6, 9] to analyze entire equivalence classes of inputs (e.g., all positive integer inputs for which the program executes along the same path). Depending on the time and resources available, developers can choose k according to their needs—in our experiments we found $k = 5$ to be sufficient to achieve 99% accuracy for all the tested programs. The value of this category will become obvious after reading §3.

"Single ordering" are races for which only a single ordering of the accesses is possible, typically enforced via ad-hoc synchronization [60]. In such cases, although no explicit synchronization primitives are used, the shared memory could be protected using busy-wait loops that synchronize on a flag. We consider this a race because the ordering of the accesses is not enforced using synchronization primitives (§1), even though it is not actually possible to exercise both interleavings of the memory accesses (hence the name of the category). Such ad-hoc synchronization, even if bad practice, is frequent in real-world software [60]. Previous data race detectors generally cannot tell that only a single order is possible for the memory accesses, and thus report this as a race; such cases turn out to be a major source of harmless data races [27, 55].

3. Design

Portend feeds the target program through its own race detector (or even a third party one, if preferred), analyzes the program and the report automatically, and determines the potential consequences of the reported data race. The report is then classified, based on these predicted consequences, into one of the four categories in Fig. 1. To achieve the classification, Portend performs targeted analysis of multiple schedules of interest, while at the same time using symbolic execution [9] to simultaneously explore multiple paths through the program; we call this technique *multi-path multi-schedule data race analysis*. Portend can thus reason about the consequences of the two orderings of racing memory accesses in a richer execution context than prior work. When comparing program states or program outputs, Portend employs *symbolic output comparison*, meaning it compares constraints on program output, instead of the concrete values of the output, in order to generalize the comparison to more possible inputs that would bring the program to the specific race condition. Unlike prior work, Portend can accurately classify even races that, given a fixed ordering of the original racing accesses, are harmless along some execution paths, yet harmful along others. In §3.1 we go over one such race (Fig. 4) and explain how Portend handles it. Fig. 2 illustrates Portend's architecture. Portend is based on Cloud9 [6], a parallel symbolic execution engine that supports running multi-threaded C/C++ programs.

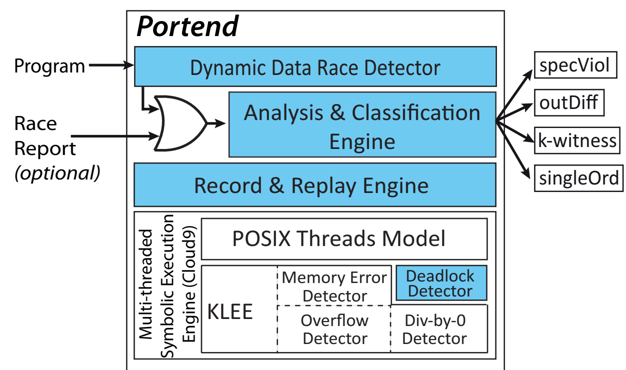

Figure 2. High-level architecture of Portend. The four shaded boxes indicate new code written for Portend, whereas clear boxes represent reused code from KLEE [9] and Cloud9 [6].

When Portend determines that a race is of "spec violated" kind, it provides the corresponding evidence in the form of program inputs (including system call return values) and thread schedule that reproduce the harmful consequences deterministically. Developers can replay this "evidence" in a debugger, to fix the race.

In the rest of this section, we give an overview of our approach and illustrate it with an example (§3.1), describe the first step, single-path/single-schedule analysis (§3.2), followed by the second step, multi-path analysis and symbolic output comparison (§3.3) augmented with multi-schedule analysis (§3.4). We describe Portend's race classification (§3.5) and the generated report that helps developers debug the race (§3.6).

3.1 Overview and Example

Portend's race analysis starts by executing the target program and dynamically detecting data races (e.g., developers could run their existing test suites under Portend). Portend detects races using a dynamic happens-before algorithm [31]. Alternatively, if a third party detector is used, Portend can start from an existing execution trace; this trace must contain the thread schedule and an indication of where in the trace the suspected race occurred. We developed a plugin for Thread Sanitizer [30] to create a Portend-compatible trace; we believe such plugins can be easily developed for other dynamic race detectors [24].

Portend has a record/replay infrastructure for orchestrating the execution of a multi-threaded program; it can preempt and schedule threads before/after synchronization operations and/or racing accesses. Portend uses Cloud9 to enumerate program paths and to collect symbolic constraints.

A trace consists of a schedule trace and a log of system call inputs. The schedule trace contains the thread id and the program counter at each preemption point. Portend treats all POSIX threads synchronization primitives as possible preemption points and uses a single-processor cooperative thread scheduler (see §6 for a discussion of the resulting advantages and limitations). Portend can also preempt threads before and after any racing memory access. We use the following notation for the trace: $(T_0 : pc_0) \rightarrow (T_1 \rightarrow RaceyAccess_{T_1} : pc_1) \rightarrow (T_2 \rightarrow RaceyAccess_{T_2} : pc_2)$ means that thread T_0 is preempted after it performs a synchronization call at program counter pc_0; then thread T_1 is scheduled and performs a memory access at program counter pc_1, after which thread T_2 is scheduled and performs a memory access at pc_2 that is racing with the previous memory access of T_1. The schedule trace also contains the absolute count of instructions executed by the program up to each preemption point. This is needed in order to perform precise replays when an instruction executes multiple times (e.g., a loop) before being involved in a race; this is not shown as part of the schedule trace, for brevity. The log of system call inputs contains the non-deterministic program inputs (e.g., *gettimeofday*).

In a first analysis step (illustrated in Fig. 3a), Portend replays the schedule in the trace up to the point where the race occurs. Then it explores two different executions: one in which the original schedule is followed (the *primary*) and one in which the alternate ordering of the racing accesses is enforced (the *alternate*). As described in §2.1, some classifiers compare the primary and alternate program state immediately after the race, and, if different, flag the race as potentially harmful. Even if program outputs are compared rather than states, "single-pre/single-post" analysis (Fig. 3a) may not be accurate, as we will show below. Portend uses "single-pre/single-post" analysis mainly to determine whether the alternate schedule is possible at all. In other words, this stage identifies any ad-hoc synchronization that might prevent the alternate schedule from occurring.

If there is a difference between the primary and alternate post-race states, we do not consider the race as necessarily harmful. Instead, we allow the primary and alternate executions to run, independently of each other, and we observe the consequences. If, for instance, the alternate execution crashes, the race is harmful. Of course, even if the primary and alternate executions behave identically, it is still not certain that the race is harmless: there may be some unexplored pair of primary and alternate paths with the same pre-race prefix as the analyzed pair, but which does not behave the same. This is why single-pre/single-post analysis is insufficient, and we need to explore *multiple* post-race paths. This motivates "single-pre/multi-post" analysis (Fig. 3b), in which multiple post-race execution possibilities are explored—if any primary/alternate mismatch is found, the developer must be notified.

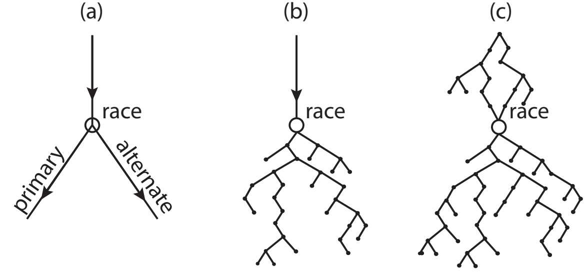

Figure 3. Increasing levels of completeness in terms of paths and schedules: [a. single-pre/single-post] $\ll$ [b. single-pre/multi-post] $\ll$ [c. multi-pre/multi-post].

Even if all feasible post-race paths are explored exhaustively and no mismatch is found, one still cannot conclude that the race is harmless: it is possible that the absence of a mismatch is an artifact of the specific pre-race execution prefix, and that some different prefix would lead to a mismatch. Therefore, to achieve higher confidence in the classification, Portend explores multiple feasible paths even in the pre-race stage, not just the one path witnessed by the race detector. This is illustrated as "multi-pre/multi-post" analysis in Fig. 3c. The advantage of doing this vs. considering these as different races is the ability to systematically explore these paths.

Finally, we combine multi-path analysis with *multi-schedule* analysis, since the same path through a program may generate different outputs depending on how its execution segments from different threads are interleaved. The branches of the execution tree in the post-race execution in Fig. 3c correspond to different paths that stem from both multiple inputs and schedules, as we detail in §3.4.

Of course, exploring all possible paths and schedules that experience the race is impractical, because their number typically grows exponentially with the number of threads, branches, and preemption points in the program. Instead, we provide developers a "dial" to control the number k of path/schedule alternatives explored during analysis, allowing them to control the "volume" of paths and schedules in Fig. 3. If Portend classifies a race as "k-witness harmless", then a higher value of k offers higher confidence that the race is harmless for *all* executions (i.e., including the unexplored ones), but it entails longer analysis time. We found $k = 5$ to be sufficient for achieving 99% accuracy in our experiments in less than 5 minutes per race on average.

To illustrate the benefit of multi-path multi-schedule analysis over "single-pre/single-post" analysis, consider the code snippet in Fig. 4, adapted from a real data race bug. This code has racing accesses to the global variable *id*. Thread T_0 spawns threads T_1 and T_2; thread T_1 updates *id* (line 15) in a loop and acquires a lock each time. However, thread T_2, which maintains statistics, reads *id* without acquiring the lock—this is because acquiring a lock at this location would hurt performance, and statistics need not be precise. Depending on program input, T_2 can update the statistics using either the *update1* or *update2* functions (lines 20-23).

Say the program runs in Portend with input *–use-hash-table*, which makes *useHashTable=true*. Portend records the primary trace $(T_0 : pc_9) \rightarrow ... (T_1 \rightarrow RaceyAccess_{T_1} : pc_{15}) \rightarrow (T_2 \rightarrow$

```
                    Thread T0
 1: int id = 0, MAX_SIZE = 32;
 2: int stats_array[MAX_SIZE];
 2: bool useHashTable;
 4:
 5: int main(int argc, char *argv[]){
 6:   pthread_t t1, t2;
 7:   useHashTable = getOption(argc, argv);
 8:   pthread_create (&t1, 0, reqHandler, 0);
 9:   pthread_create (&t2, 0, updateStats, 0);
10:   ...
                    Thread T1
11: void * reqHandler(void *arg){
12:   while(1){
13:     ...
14:     lock(l);
15:     id++;
16:     unlock(l);
17:     ...
                    Thread T2
18: void * updateStats(void* arg){
19:   if(useHashTable){
20:     update1();
21:     printf(..., hash_table[id]);
22:   } else {
23:     update2();
24:     ...

25:   void update1(){
26:     int tmp = id;
27:     if (hash_table.contains(tmp))
28:       hash_table[tmp] = getStats();

29:   void update2(){
30:     if (id < MAX_SIZE)
31:       stats_array[id] = getStats();
```

Figure 4. Simplified example of a harmful race from Ctrace [39] that would be classified as harmless by classic race classifiers.

$RaceyAccess_{T_2} : pc_{26}) \rightarrow ...\ T_0$. This trace is fed to the first analysis step, which replays the trace with the same program input, except it enforces the alternate schedule $(T_0 : pc_9) \rightarrow ...(T_2 \rightarrow RaceyAccess_{T_2} : pc_{26}) \rightarrow (T_1 \rightarrow RaceyAccess_{T_1} : pc_{15}) \rightarrow ...\ T_0$. Since the printed value of *hash_table[id]* at line 21 would be the same for the primary and alternate schedules, a "single-pre/single-post" classifier would deem the race harmless.

However, in the multi-path multi-schedule step, Portend explores additional paths through the code by marking program input as symbolic, i.e., allowing it to take on any permitted value. When the trace is replayed and Portend reaches line 19 in T_2 in the alternate schedule, *useHashTable* could be both true and false, so Portend splits into two executions, one in which *useHashTable* is set to *true* and one in which it is *false*. Assume, for example, that *id* = 31 when checking the if condition at line 30. Due to the data race, *id* is incremented by T_1 to 32, which overflows the statically allocated buffer (line 31). Note that in this alternate path, there are two racing accesses on *id*, and we are referring to the access at line 31.

Portend detects the overflow (via Cloud9), which leads to a crashed execution, flags the race as "spec violated", and provides the developer the execution trace in which the input is *–no-hash-table*, and the schedule is $(T_0 : pc_9) \rightarrow ...(T_2 \rightarrow RaceyAccess_{T_2} : pc_{30}) \rightarrow (T_1 \rightarrow RaceyAccess_{T_1} : pc_{15}) \rightarrow (T_2 : pc_{31})$. The developer can replay this trace in a debugger and fix the race.

Note that this data race is harmful only if the program input is *–no-hash-table*, the given thread schedule occurs, and the value of *id* is 31; therefore the crash is likely to be missed by a traditional single-path/single-schedule data race detector.

We now describe Portend's race analysis in detail: §3.2–§3.4 focus on the *exploration* part of the analysis, in which Portend looks for paths and schedules that reveal the nature of the race, and §3.5 focuses on the *classification* part.

Algorithm 1: Single-Pre/Single-Post Analysis (singleClassify)

Input: Primary execution trace *primary*
Output: Classification result $\in \{specViol, outDiff, outSame, singleOrd\}$

```
1  current ← execUntilFirstThreadRacyAccess(primary)
2  preRaceCkpt ← checkpoint(current)
3  execUntilSecondThreadRacyAccess(current)
4  postRaceCkpt ← checkpoint(current)
5  current ← preRaceCkpt
6  preemptCurrentThread(current)
7  alternate ← execWithTimeout(current)
8  if alternate.timedOut then
9      if detectInfiniteLoop(alternate) then
10         return specViol
11     else
12         return singleOrd
13 else
14     if detectDeadlock(alternate) then
15         return specViol
16 primary ← exec(postRaceCkpt)
17 if detectSpecViol(primary) ∨ detectSpecViol(alternate)
   then
18     return specViol
19 if primary.output ≠ alternate.output then
20     return outDiff
21 else
22     return outSame
```

3.2 Single-Pre/Single-Post Analysis

The goal of this first analysis step is to identify cases in which the alternate schedule of a race cannot be pursued, and to make a first classification attempt based on a single alternate execution. Algorithm 1 describes the approach.

Portend starts from a trace of an execution of the target program, containing one or more races, along with the program inputs that generated the trace. For example, in the case of the Pbzip2 file compressor used in our evaluation, Portend needs a file to compress and a trace of the thread schedule. As mentioned earlier, such traces are obtained from running, for instance, the developers' test suites (as done in CHESS [44]) with a dynamic race detector enabled.

Portend takes the primary trace and plays it back (line 1). Note that *current* represents the system state of the current execution. Just before the first racing access, Portend takes a checkpoint of system state; we call this the *pre-race checkpoint* (line 2). The replay is then allowed to continue until immediately after the second racing access of the race we are interested in (line 3), and the primary execution is suspended in this post-race state (line 4).

Portend then primes a new execution with the pre-race checkpoint (line 5) and attempts to enforce the alternate ordering of the racing accesses. To enforce this alternate order, Portend preempts the thread that did the first racing access (T_i) in the primary execution and allows the other thread (T_j) involved in the race to be scheduled (line 6). In other words, an execution with the trace $...(T_i \rightarrow RaceyAccess_{T_i} : pc_1) \rightarrow (T_j \rightarrow RaceyAccess_{T_j} : pc_2)...$ is steered toward the execution $...(T_j \rightarrow RaceyAccess_{T_j} : pc_2) \rightarrow (T_i \rightarrow RaceyAccess_{T_i} : pc_1)...$

This attempt could fail for one of two reasons: (a) T_j gets scheduled, but T_i cannot be scheduled again; or (b) T_j cannot be scheduled, because it is blocked by T_i. Case (a) is detected by Portend

via a timeout (line 8) and is classified either as “spec violated”, corresponding to an infinite loop (i.e., a loop with a loop-invariant exit condition) in line 10 or as ad-hoc synchronization in line 12. Case (b) can correspond to a deadlock (line 15) and is detected by Portend by keeping track of the lock graph. Both the infinite loop and the deadlock case cause the race to be classified as “spec violated”, while the ad-hoc synchronization case classifies the race as “single ordering” (more details in §3.5). While it may make sense to not stop if the alternate execution cannot be enforced, under the expectation that other paths with other inputs might permit the alternate ordering, our evaluation suggests that continuing adds little value.

If the alternate schedule succeeds, Portend executes it until it completes, and then records its outputs. Then, Portend allows the primary to continue (while replaying the input trace) and also records its outputs. During this process, Portend watches for “basic” specification violations (crashes, deadlocks, memory errors, etc.) as well as “high level” properties given to Portend as predicates—if any of these properties are violated, Portend immediately classifies (line 18) the race as “spec violated”. If the alternate execution completes with no specification violation, Portend compares the outputs of the primary and the alternate; if they differ, the race is classified as “output differs” (line 20), otherwise the analysis moves to the next step. This is in contrast to replay-based classification [45], which compares the program state immediately after the race in the primary and alternate interleavings.

3.3 Multi-Path Data Race Analysis

The goal of this step is to explore variations of the single paths found in the previous step (i.e., the primary and the alternate) in order to expose Portend to a wider range of execution alternatives.

First, Portend finds multiple primary paths that satisfy the input trace, i.e., they (a) all experience the same thread schedule (up to the data race) as the input trace, and (b) all experience the target race condition. These paths correspond to *different* inputs from the ones in the initial race report. Second, Portend uses Cloud9 to record the “symbolic” outputs of these paths—that is, the constraints on the output, rather than the concrete output values themselves—and compares them to the outputs of the corresponding alternate paths; we explain this below. Algorithm 2 describes the functions invoked by Portend during this analysis in the following order: 1) on initialization, 2) when encountering a thread preemption, 3) on a branch that depends on symbolic data, and 4) on finishing an execution.

Unlike in the single-pre/single-post step, Portend now executes the primary *symbolically*. This means that the target program is given symbolic inputs instead of regular concrete inputs. Cloud9 relies in large part on KLEE [9] to interpret the program and propagate these symbolic values to other variables, corresponding to how they are read and operated upon. When an expression with symbolic content is involved in the condition of a branch, *both* options of the branch are explored, if they are feasible. The resulting path(s) are annotated with a constraint indicating that the branch condition holds true (respectively false). Thus, instead of a regular single-path execution, we get a tree of execution paths, similar to the one in Fig. 5. Conceptually, at each such branch, program state is duplicated and constraints on the symbolic parameters are updated to reflect the decision taken at that branch (line 11). Describing the various techniques for performing symbolic execution efficiently [6, 9] is beyond the scope of this paper.

An important concern in symbolic execution is “path explosion,” i.e., that the number of possible paths is large. Portend offers two parameters to control this growth: (a) an upper bound M_p on the number of primary paths explored; and (b) the number and size of symbolic inputs. These two parameters allow developers to trade performance vs. classification confidence. For parameter (b),

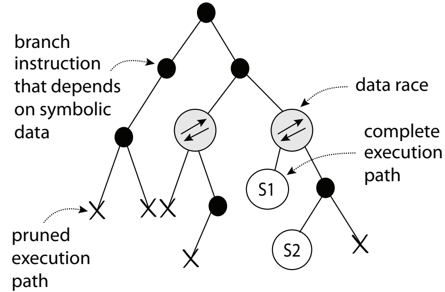

Figure 5. Portend prunes paths during symbolic execution.

the fewer inputs are symbolic, the fewer branches will depend on symbolic input, so less branching will occur in the execution tree.

Determining the optimal values for these parameters may require knowledge of the target system as well as a good sense of how much confidence is required by the system’s users. Reasonable (i.e., good but not necessarily optimal) values can be found through trial and error relatively easily—we expect development teams using Portend to converge onto values that are a good fit for their code and user community, and then make these values the defaults for their testing and triage processes. We empirically study in §5 the impact of these parameters on classification accuracy on a diverse set of programs and find that relatively small values achieve high accuracy for a broad range of programs.

During symbolic execution, Portend prunes (Fig. 5) the paths that do not obey the thread schedule in the trace (line 8), thus excluding the (many) paths that do not enable the target race. Moreover, Portend attempts to follow the original trace only until the second racing access is encountered; afterward, it allows execution to diverge from the original schedule trace. This enables Portend to find more executions that partially match the original schedule trace (e.g., cases in which the second racing access occurs at a different program counter, as in Fig. 4). Tolerating these divergences significantly increases Portend’s accuracy over the state of the art [45], as will be explained in §5.4.

Once the desired paths are obtained (at most M_p, line 14), the conjunction of branch constraints accumulated along each path is solved by KLEE using an SMT solver [19] in order to find concrete inputs that drive the program down the corresponding path. For example, in the case of Fig. 5, two successful leaf states S_1 and S_2 are reached, and the solver provides the inputs corresponding to the path from the root of the tree to S_1, respectively S_2. Thus, we now have $M_p = 2$ different primary executions that experience the data race.

3.3.1 Symbolic Output Comparison

Portend now records the output of each of the M_p executions, like in the single-pre/single-post case. However, this time it propagates the constraints on symbolic state all the way to the outputs, i.e., the outputs of each primary execution contain a mix of concrete values and symbolic constraints (i.e., symbolic formulae). Note that by output we mean all arguments passed to output system calls.

Next, for each of the M_p executions, Portend produces a corresponding alternate (analogously to the single-pre/single-post case) and records its outputs (lines 19-21). The function $singleClassify$ in Algorithm 2 performs the analysis described in Algorithm 1. Portend then checks whether the outputs of each alternate satisfy the constraints of the corresponding primary’s outputs, i.e., verifies equivalence between the (partly symbolic) outputs of the primary and those of the alternate. This is what we refer to as *symbolic output comparison* (line 22).

Algorithm 2: Multi-path Data Race Analysis (Simplified)

```
Input: Schedule trace trace, initial program state S_0, set of states
       S = ∅, upper bound M_p on the number of primary paths
Output: Classification result ∈ {specViol, outDiff, singleOrd
        k-witness}
1  function init ()
2    S ← S ∪ S_0
3    current ← S.head()
4    pathsExplored ← 0
5  function onPreemption ()
6    t_i ← scheduleNextThread(current)
7    if t_i ≠ nextThreadInTrace(trace, current) then
8      S ← S.remove(current)
9      current ← S.head()
10 function onSymbolicBranch ()
11   S ← S ∪ current.fork()
12 function onFinish ()
13   classification ← classification ∪ classify(current)
14   if pathsExplored < M_p then
15     pathsExplored ← pathsExplored + 1
16   else
17     return classification
18 function classify (primary)
19   result ← singleClassify(primary)
20   if result = outSame then
21     alternate ← getAlternate(primary)
22     if symbolicMatch(primary.symOutput,
       alternate.output) then
23       return k-witness
24     else
25       return outDiff
26   else
27     return result
```

When executing the primaries and recording their outputs, Portend relies on Cloud9 to track all symbolic constraints on variables, and Portend records these constraints as symbolic output. For example, when Portend runs the primary execution of "*i=getInput(); if (i ≥ 0) output(i);* " on input 10, it records the output as $i \geq 0$, not merely value $i = 10$. Thus, the outputs of primary executions are recorded as sequences of symbolic formulae.

The alternate executions are fully concrete, and Portend records their concrete outputs. When comparing outputs, Portend first checks that the number of output operations match in the two executions. If yes, then, for each output operation, it checks that the concrete output (from the alternate) is in the set of values allowed by the constraints of the symbolic output (from the primary). For the example above, any positive value output by the alternate will satisfy the primary's $i \geq 0$ output. This symbolic comparison enables Portend's analysis to extend over more possible primary executions for which i is a positive integer. This comes at the price of potential false negatives; despite this theoretical shortcoming, we have not encountered such a case in practice, but we plan to investigate further in future work. Of course, determining semantic equivalence of output is undecidable, and our comparison may still wrongly classify as "output differs" a sequence of outputs that are equivalent at some level (e.g., *<print ab; print c> vs. <print abc>*).

To determine if the concrete outputs satisfy the symbolic ones, Portend directly employs an SMT solver [19]. As will be seen in §5.2, using symbolic comparison leads to substantial improvements in classification accuracy.

We do not detail here the case when the program reads input *after* the race—it is a natural extension of the algorithm above.

3.4 Multi-Schedule Data Race Analysis

The goal of multi-schedule analysis is to further augment the set of analyzed executions by diversifying the thread schedule.

We mentioned earlier that, for each of the M_p primary executions, Portend obtains an alternate execution. Once the alternate ordering of the racing accesses is enforced, Portend randomizes the schedule of the *post-race* alternate execution: at every preemption point in the alternate, Portend randomly decides which of the runnable threads to schedule next. This means that every alternate execution will most likely have a different schedule from the original input trace (and thus from the primary).

Consequently, for every primary execution P_i, we obtain *multiple* alternate executions $A_i^1, A_i^2, ...$ by running up to M_a multiple instances of the alternate execution. Since the scheduler is random, we expect practically every alternate execution to have a schedule that differs from all others. Recently proposed techniques [43] can be used to quantify the probability of these alternate schedules discovering the harmful effects of a data race.

Portend then uses the same symbolic comparison technique as in §3.3.1 to establish equivalence between the concrete outputs of $A_i^1, A_i^2, ...A_i^{M_a}$ and the symbolic outputs of P_i.

Schedule randomization can be employed also in the pre-race stage of the alternate-execution generation as well as in the generation of the primary executions. We did not implement these options, because the level of multiplicity we obtain with the current design appears to be sufficient in practice to achieve high accuracy. Note however that, as we show in §5.2, multi-path multi-schedule analysis is indeed crucial to attaining high classification accuracy.

In summary, multi-path multi-schedule analysis explores M_p primary executions and, for each such execution, M_a alternate executions with different schedules, for a total of $M_p \times M_a$ path-schedule combinations. For races that end up being classified as "k-witness harmless", we say that $k = M_p \times M_a$ is the lower bound on the number of concrete path-schedule combinations under which this race is harmless.

Note that the k executions can be simultaneously explored in parallel: if a developer has p machines with q cores each, she could explore $p \times q$ parallel executions in the same amount of time as a single execution. Given that Portend is "embarrassingly parallel," it is appealing for cluster-based automated bug triage systems.

3.5 Data Race Classification

We showed how Portend explores paths and schedules to give the classifier an opportunity to observe the effects of a data race. We now provide details on how the classifier makes its decisions.

"Spec violated" races cause a program's explicit specification to be violated; they are guaranteed to be harmful and thus should have highest priority for developers. To detect violations, Portend watches for them during exploration.

First, Portend watches for "basic" properties that can be safely assumed to violate any program's specification: crashes, deadlocks, infinite loops, and memory errors. Since Portend already controls the program's schedule, it also keeps track of all uses of synchronization primitives (i.e., POSIX threads calls); based on this, it determines when threads are deadlocked. Infinite loops are diagnosed as in [60], by detecting loops for which the exit condition cannot be modified. For memory errors, Portend relies on the mechanism already provided by KLEE inside Cloud9. Even when Portend runs the program concretely, it still interprets it in Cloud9.

Second, Portend watches for "semantic" properties, which are provided to Portend by developers in the form of assert-like predicates. Developers can also place these assertions inside the code.

Whenever an alternate execution violates a basic or a semantic property (even though the primary may not), Portend classifies the corresponding race as "spec violated".

"Output differs" races cause a program's output to depend on the ordering of the racing accesses. As explained in §2.1, a difference between the post-race memory or register states of the primary and the alternate is not necessarily indicative of a harmful race (e.g., the difference may just be due to dynamic memory allocation). Instead, Portend compares the outputs of the primary and the alternate, and it does so symbolically, as described earlier. In case of a mismatch, Portend classifies the race as "output differs" and gives the developer detailed information to decide whether the difference is harmful or not.

"K-witness harmless" races: If, for every primary execution P_i, the outputs of alternate executions $A_i^1, A_i^2, ...A_i^{M_a}$ satisfy P_i's output constraints, then Portend classifies the race as "k-witness harmless", where $k = M_p \times M_a$, because there exist k executions witnessing the conjectured harmlessness. The value of k is often an underestimate of the number of different executions for which the race is guaranteed to be harmless; as suggested earlier in §2.3, symbolic execution can even reason about a virtually infinite number of executions.

Theoretical insights into how k relates to the confidence a developer can have that a "k-witness harmless" race will not cause harm in practice are beyond the scope of this paper. One can think of k in ways similar to code coverage in testing: 80% coverage is better than 60%, but does not exactly predict the likelihood of bugs not being present. For all our experiments, $k = 5$ was shown to be sufficient for achieving 99% accuracy. We consider "k-witness harmless" analyses to be an intriguing topic for future work, in a line of research akin to [43]. Note that Portend explores many more executions before finding the required k path-schedule combinations that match the trace, but the paths that do not match the trace are pruned early during the analysis.

"Single ordering" races are harmless races, because only one ordering of the racing accesses is possible. One might even argue they are not races at all. Yet, dynamic data race detectors are not aware of the implicit happens-before relationship and do report a race.

When Portend cannot enforce an alternate interleaving in the single-pre/single-post phase, this can either be due to ad-hoc synchronization that prevents the alternate ordering, or other thread in question cannot make progress due to a deadlock or an infinite loop. If none of the previously described infinite-loop and deadlock detection mechanisms trigger, Portend simply waits for a configurable amount of time and, upon timeout, classifies the race as "single ordering." Note that it is possible to improve this design with a heuristic-based static analysis that directly identifies ad-hoc synchronization [55, 60].

3.6 Portend's Debugging Aid Output

To help developers decide what to do about an "output differs" race, Portend dumps the output values and the program locations where the output differs. Portend also aims to help in fixing harmful races by providing for each race two items: a textual report and a pair of execution traces that evidence the effects of the race and can be played back in a debugger, using Portend's runtime replay environment. A simplified report is shown in Fig. 6.

In the case of an "output differs" race, Portend reports the stack traces of system calls where the program produced different output, as well as the differing outputs. This simplifies the debugging effort (e.g., if the difference occurs while printing a debug message, the race could be classified as benign with no further analysis).

```
Data Race during access to: 0x2860b30
current thread id: 3: READ
racing thread id: 0: WRITE
Current thread at:
  /home/eval/pbzip/pbzip2.cpp:702
Previous at:
  /home/eval/pbzip/pbzip2.cpp:389
size of the accessed field: 4 offset: 0
```

Figure 6. Example debugging aid report for Portend.

4. Implementation

The current Portend prototype consists of approximately 8 KLOC of C++ code, incorporating the various analyses described earlier and modifications to the underlying symbolic execution engine. The four shaded components in Fig. 2 were developed from scratch as part of Portend: the dynamic data race detector, the analysis and classification engine, the record-replay engine, and the deadlock detector.

Portend works on programs compiled to LLVM [33] bitcode and can run C/C++ programs for which there exists a sufficiently complete symbolic POSIX environment [6]. We have tested it on C programs as well as C++ programs that do not link to libstdc++; this latter limitation results from the fact that an implementation of a standard C++ library for LLVM is in progress, but not yet available [12]. Portend uses Cloud9 [6] to interpret and symbolically execute LLVM bitcode; we suspect any path exploration tool will do (e.g., CUTE [52], SAGE [22]), as long as it supports multi-threaded programs.

Portend intercepts various system calls, such as *write*, under the assumption that they are the primary means by which a program communicates changes in its state to the environment. A separate Portend module is responsible for keeping track of symbolic outputs in the form of constraints, as well as of concrete outputs. Portend hashes program outputs (when they are concrete) and can either maintain hashes of all concrete outputs or compute a hash chain of all outputs to derive a single hash code per execution. This way, Portend can deal with programs that have a large amount of output.

Portend clusters the data races it detects, in order to filter out similar races; the clustering criterion is whether the racing accesses are made to the same shared memory location by the same threads, and the stack traces of the accesses are the same. Portend provides developers with a single representative data race from each cluster.

The timeout used in discovering ad-hoc synchronization is conservatively defined as 5 times what it took Portend to replay the primary execution, assuming that reversing the access sequence of the racing accesses should not cause the program to run for longer than that.

In order to run multi-threaded programs in Portend, we extended the POSIX threads support found in Cloud9 to cover almost the entire POSIX threads API, including barriers, mutexes and condition variables, as well as thread-local storage. Portend intercepts calls into the POSIX threads library to maintain the necessary internal data structures (e.g., to detect data races and deadlocks) and to control thread scheduling.

5. Evaluation

In this section, we answer the following questions: Is Portend effective in telling developers which races are true bugs and in helping them fix buggy races (§5.1)? How accurately does it classify race reports into the four categories of races (§5.2)? How long does classification take, and how does it scale (§5.3)? How does Portend compare to the state of the art in race classification (§5.4)?

To answer these questions, we apply Portend to 7 applications: SQLite, an embedded database engine (used, for example, by Firefox, iOS, Chrome, and Android), that is considered highly reliable, with 100% branch coverage [53]; Pbzip2, a parallel implementation of the widely used bzip2 file compressor [20]; Memcached [16], a distributed memory object cache system (used, for example, by services such as Flickr, Twitter and Craigslist); Ctrace [39], a multi-threaded debug library; Bbuf [61], a shared buffer implementation with a configurable number of producers and consumers; Fmm, an n-body simulator from the popular SPLASH2 benchmark suite [59]; and Ocean, a simulator of eddy currents in oceans, from SPLASH2.

Portend classifies with 99% accuracy the 93 known data races we found in these programs, with no human intervention, in under 5 minutes per race on average. It took us one person-month to manually confirm that the races deemed harmless by Portend were indeed harmless—this is typical of how long it takes to classify races in the absence of an automated tool [23] and illustrates the benefits of Portend-style automation. For the deemed-harmful races, we confirmed classification accuracy in a few minutes by using the replayable debug information provided by Portend.

We additionally evaluate Portend on homegrown micro-benchmarks that capture most classes of harmless races [30, 45]: "redundant writes" (RW), where racing threads write the same value to a shared variable, "disjoint bit manipulation" (DBM), where disjoint bits of a bit-field are modified by racing threads, "all values valid" (AVV), where the racing threads write different values that are nevertheless all valid, and "double checked locking" (DCL), a method used to reduce the locking overhead by first testing the locking criterion without actually acquiring a lock. Table 1 summarizes the properties of our 11 experimental targets.

Program	Size (LOC)	Language	# Forked threads
SQLite 3.3.0	113,326	C	2
ocean 2.0	11,665	C	2
fmm 2.0	11,545	C	3
memcached 1.4.5	8,300	C	8
pbzip2 2.1.1	6,686	C++	4
ctrace 1.2	886	C	3
bbuf 1.0	261	C	8
AVV	49	C++	3
DCL	45	C++	5
DBM	45	C++	3
RW	42	C++	3

Table 1. Programs analyzed with Portend. Source lines of code are measured with the `cloc` utility.

We ran Portend on several other systems (e.g., HawkNL, pfscan, swarm, fft), but no races were found in those programs with the test cases we ran, so we do not include them here. For all experiments, the Portend parameters were set to $M_p = 5$, $M_a = 2$, and the number of symbolic inputs to 2. We found these numbers to be sufficient to achieve high accuracy in a reasonable amount of time. To validate Portend's results, we used manual investigation, analyzed developer change logs, and consulted with the applications' developers when possible. All experiments were run on a 2.4 GHz Intel Core 2 Duo E6600 CPU with 4 GB of RAM running Ubuntu Linux 10.04 with kernel version 2.6.33. The reported numbers are averages over 10 experiments.

5.1 Effectiveness

Of the 93 distinct races detected in 7 real-world applications, Portend classified 5 as definitely harmful by watching for "basic" properties (Table 2): one hangs the program and four crash it.

Program	Total # of races	# of "Spec violated" races		
		Deadlock	Crash	Semantic
SQLite	1	1	0	0
pbzip2	31	0	3	0
ctrace	15	0	1	0
fmm	13	0	0	1
memcached	18	0	1	0

Table 2. "Spec violated" races and their consequences.

To illustrate the checking for "high level" semantic properties, we instructed Portend to verify that all timestamps used in fmm are positive. This caused it to identify the 6th "harmful" race in Table 2; without this semantic check, this race turns out to be harmless, as the negative timestamp is eventually overwritten.

To illustrate a "what-if analysis" scenario, we turned an arbitrary synchronization operation in the memcached binary into a no-op, and then used Portend to explore the question of whether it is safe to remove that particular synchronization point (e.g., we may be interested in reducing lock contention). Removing this synchronization induces a race in memcached; Portend determined that the race could lead to a crash of the server for a particular interleaving, so it classified it as "spec violated".

Portend's main contribution is the classification of races. If one wanted to eliminate all harmful races from their code, they could use a static race detector (one that is complete, and, by necessity, prone to false positives) and then use Portend to classify these reports.

For every harmful race, Portend's comprehensive report and replayable traces (i.e., inputs and thread schedule) allowed us to confirm the harmfulness of the races within minutes. Portend's report includes the stack traces of the racing threads along with the address and size of the accessed memory field; in the case of a segmentation fault, the stack trace of the faulting instruction is provided as well—this information can help in automated bug clustering. According to developers' change logs and our own manual analysis, the races in Table 2 are the only known harmful races in these applications.

5.2 Accuracy and Precision

To evaluate Portend's accuracy and precision, we had it classify all 93 races in our target applications and micro-benchmarks. Table 3 summarizes the results. The first two columns show the number of distinct races and the number of respective instances, i.e., the number of times those races manifested during race detection. The "spec violated" column includes all races from Table 2 minus the semantic race in fmm and the race we introduced in memcached. In the "k-witness harmless" column we show for which races the post-race states differed vs. not.

By accuracy, we refer to the correctness of classification: the higher the accuracy, the higher the ratio of correct classification. Precision on the other hand, refers to the reproducibility of experimental results: the higher the precision, the higher the ratio with which experiments are repeated with the same results.

To determine accuracy, we manually classified each race and found that Portend had correctly classified 92 of the 93 races (99%) in our target applications: all except one of the races classified "k-witness harmless" by Portend are indeed harmless in an absolute sense, and all "single ordering" races indeed involve ad-hoc synchronization.

To measure precision, we ran 10 times the classification for each race. Portend consistently reported the same data set shown in Table 3, which indicates that, for these races and applications, it achieves full precision.

Program	Number of data races						
	Distinct races	Race instances	Spec violated	Output differs	K-witness harmless		Single ordering
					states same	*states differ*	
SQLite	1	1	1	0	0	0	0
ocean	5	14	0	0	0	1	4
fmm	13	517	0	0	0	1	12
memcached	18	104	0	2	0	0	16
pbzip2	31	97	3	3	0	0	25
ctrace	15	19	1	10	0	4	0
bbuf	6	6	0	6	0	0	0
AVV	1	1	0	0	1	0	0
DCL	1	1	0	0	1	0	0
DBM	1	1	0	0	1	0	0
RW	1	1	0	0	1	0	0

Table 3. Summary of Portend's classification results. We consider two races to be distinct if they involve different accesses to shared variables; the same race may be encountered multiple times during an execution—these two different aspects are captured by the *Distinct races* and *Race instances* columns, respectively. The last 5 columns classify the distinct races. The *states same/differ* columns show for how many races the primary and alternate states were different after the race, as computed by the Record/Replay Analyzer [45].

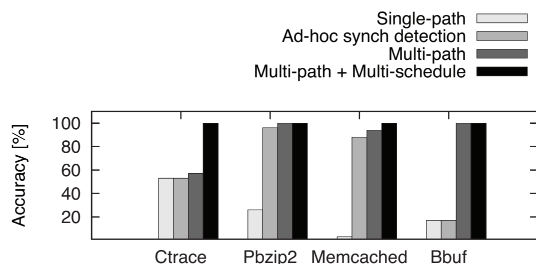

Figure 7. Breakdown of the contribution of each technique toward Portend's accuracy. We start from single-path analysis and enable one by one the other techniques: ad-hoc synchronization detection, multi-path analysis, and finally multi-schedule analysis.

As can be seen in the "k-witness harmless" column, for each and every one of the 7 real-world applications, a state difference (as used in [45]) does not correctly predict harmfulness, while our "k-witness harmless" analysis correctly predicts that the races are harmless with one exception.

This suggests that differencing of concrete state is a poor classification criterion for races in real-world applications with large memory states, but may be acceptable for simple benchmarks. This also supports our choice of using symbolic output comparison.

Multi-path multi-schedule exploration proved to be crucial for Portend's accuracy. Fig. 7 shows the breakdown of the contribution of each technique used in Portend: ad-hoc synchronization detection, multi-path analysis, and multi-schedule analysis. In particular, for 16 out of 21 "output differs" races (6 in bbuf, 9 in ctrace, 1 in pbzip2) and for 1 "spec violated" race (in ctrace), single-path analysis revealed no difference in output; it was only multi-path multi-schedule exploration that revealed an output difference (9 races required multi-path analysis for classification, and 8 races required also multi-schedule analysis). Without multi-path multi-schedule analysis, it would have been impossible for Portend to accurately classify those races by just using the available test cases. Moreover, there is a high variance in the contribution of each technique for different programs, which means that none of these techniques alone would have achieved high accuracy for a broad range of programs.

We also wanted to evaluate Portend's ability to deal with false positives, i.e., false race reports. Race detectors, especially static

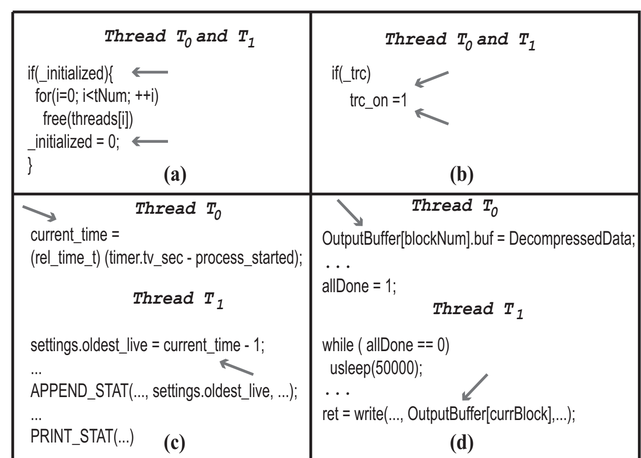

Figure 8. Simplified examples for each race class from real systems. (a) and (b) are from ctrace, (c) is from memcached and (d) is from pbzip2. The arrows indicate the pair of racing accesses.

ones, may report false positives for a variety of reasons, depending on which technique they employ. To simulate an imperfect detector for our applications, we deliberately removed from Portend's race detector its awareness of mutex synchronizations. We then eliminated the races in our micro-benchmarks by introducing mutex synchronizations. When we re-ran Portend with the erroneous data race detector on the micro-benchmarks, all four were falsely reported as races by the detector, but Portend ultimately classified all of them as "single ordering". This suggests Portend is capable of properly handling false positives.

Fig. 8 shows examples of real races for each category: (a) a "spec violated" race in which resources are freed twice, (b) a "k-witness harmless" race due to redundant writes, (c) an "output differs" race in which the schedule-sensitive value of the shared variable influences the output, and (d) a "single ordering" race showing ad-hoc synchronization implemented via busy wait.

5.3 Performance (Time to Classify)

We evaluate the performance of Portend in terms of efficiency and scalability. Portend's performance is mostly relevant if it is to be used interactively, as a developer tool, and also if used for a

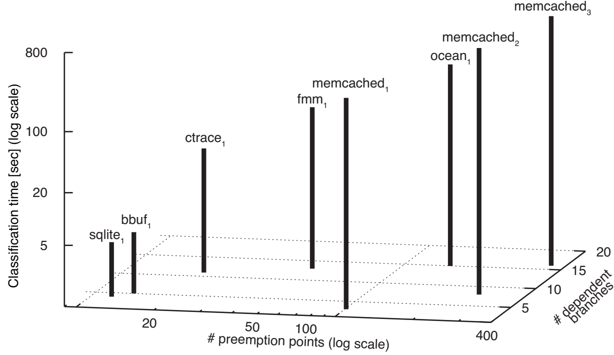

Figure 9. Change in classification time with respect to number of preemptions and number of dependent branches for some of the races in Table 3. Each sample point is labeled with race id.

large scale bug triage tool, such as in Microsoft's Windows Error Reporting system [21].

We measure the time it takes Portend to classify the 93 races; Table 4 summarizes the results. We find that Portend classifies all detected data races in a reasonable amount of time, the longest taking less than 11 minutes. For bbuf, ctrace, ocean and fmm, the slowest classification time is due to a race from the "k-witness harmless" category, since classification into this category requires multi-path multi-schedule analysis.

The second column reports the time it took Cloud9 to interpret the programs with concrete inputs. This provides a sense of the overhead incurred by Portend compared to regular LLVM interpretation in Cloud9. Both data race detection and classification are disabled when measuring baseline interpretation time. In summary, the overhead introduced by classification ranges from 1.1× to 49.9× over Cloud9.

In order to get a sense of how classification time scales with program characteristics, we measured it as a function of program size, number of preemption points, number of branches that depend (directly or indirectly) on symbolic inputs, and number of threads. We found that program size plays almost no role in classification time. Instead, the other three characteristics play an important role. We show in Fig. 9 how classification time varies with the number of dependent branches and the number of preemptions in the schedule (which is roughly proportional to the number of preemption points and the number of threads). Each vertical bar corresponds to the classification time for the indicated data race. We see that, as the number of preemptions and branches increase, so does classification time.

Program	Cloud9 running time (sec)	Portend classification time (sec)		
		Avg	*Min*	*Max*
SQLite	3.10	4.20	4.09	4.25
ocean	19.64	60.02	19.90	207.14
fmm	24.87	64.45	65.29	72.83
memcached	73.87	645.99	619.32	730.37
pbzip2	15.30	360.72	61.36	763.43
ctrace	3.67	24.29	5.54	41.08
bbuf	1.81	4.47	4.77	5.82
AVV	0.72	0.83	0.78	1.02
DCL	0.74	0.85	0.83	0.89
DBM	0.72	0.81	0.79	0.83
RW	0.74	0.81	0.81	0.82

Table 4. Portend's classification time for the 93 races in Table 3.

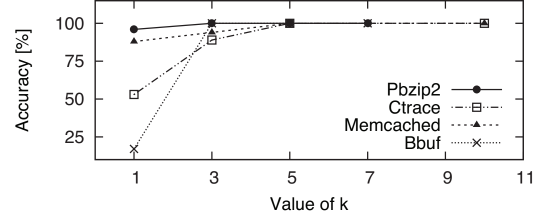

Figure 10. Portend's accuracy with increasing values of k.

We analyzed Portend's accuracy with increasing values of k and found that $k = 5$ is sufficient to achieve overall 99% accuracy for all the programs in our evaluation. Fig. 10 shows the results for Ctrace, Pbzip2, Memcached, and Bbuf. We therefore conclude that it is possible to achieve high classification accuracy with relatively small values of k.

5.4 Comparison to State of the Art

We compare Portend to the Record/Replay-Analyzer technique [45], Helgrind$^+$'s technique [27], and Ad-Hoc-Detector [55] in terms of the accuracy with which races are classified. We implemented the Record/Replay-Analyzer technique in Portend and compared accuracy empirically. For the ad-hoc synchronization detection techniques, since we do not have access to the implementations, we analytically derive the expected classification based on the published algorithms. We do not compare to RACEFUZZER [51], because it is primarily a bug finding tool looking for harmful races that occur due to exceptions and memory errors; it therefore does not provide a fine-grain classification of races. Similarly, no comparison is provided to DataCollider [29], since race classification in this tool is based on heuristics that pertain to races that we rarely encountered in our evaluation.

In Table 5 we show the accuracy, relying on manual inspection as "ground truth". Record/Replay-Analyzer does not tolerate replay failures and classifies races that exhibit a post-race state mismatch as harmful (shown as specViol), causing it to have low accuracy (10%) for that class. When comparing to Helgrind$^+$ and Ad-Hoc-Detector, we conservatively assume that these tools incur no false positives when ad-hoc synchronization is present, even though this is unlikely, given that both tools rely on heuristics. This notwithstanding, both tools are focused on weeding out races due to ad-hoc synchronization, so they cannot properly classify the other races (36 out of 93). In contrast, Portend classifies a wider range of races with high accuracy.

	specViol	**k-witness**	**outDiff**	**singleOrd**
Ground Truth	100%	100%	100%	100%
Record/Replay Analyzer	10%	95%	0% (not-classified)	
Ad-Hoc-Detector, Helgrind$^+$	0% (not-classified)			100%
Portend	100%	99%	99%	100%

Table 5. Accuracy for each approach and each classification category, applied to the 93 races in Table 3. "Not-classified" means that an approach cannot perform classification for a particular class.

The main advantage of Portend over Record/Replay-Analyzer is that it is immune to replay failures. In particular, for all the races classified by Portend as "single ordering", there was a replay divergence (that caused replay failures in Record/Replay-Analyzer), which would cause Record/Replay-Analyzer to classify the corresponding races as harmful despite them being harmless; this accounts for 57 of the 84 misclassifications. Note that even if

Record/Replay-Analyzer were augmented with a phase that pruned "single ordering" races (57/93), it would still diverge on 32 of the remaining 36 races and classify them as "spec violated", whereas only 5 are actually "spec violated". Portend, on the other hand, correctly classifies 35/36 of those remaining races. Another advantage is that Portend classifies based on symbolic output comparison, not concrete state comparison.

We manually verified and, when possible, checked with developers that the races in the "k-witness harmless" category are indeed harmless. Except for one race, we concluded that developers intentionally left these races in their programs because they considered them harmless. These races match known patterns [29, 45], such as redundant writes to shared variables (e.g., we found such patterns in Ctrace). However, for one race in Ocean, we confirmed that Portend did not figure out that the race belongs in the "output differs" category (the race can produce different output if a certain path in the code is followed, which depends indirectly on program input). Portend was not able to find this path even with $k = 10$ after one hour. Manual investigation revealed that this path is hard to find because it requires a very specific and complex combination of inputs.

In summary, Portend is able to classify with 99% accuracy and full precision all the 93 races into four data race classes defined in §2.3 in under 5 minutes per race on average. Furthermore, Portend correctly identifies 6 serious harmful races. Compared to previously published race classification methods, Portend performs more accurate classification and is able to correctly classify up to 89% (83 out of 93) more data races than existing replay-based tools. Portend also correctly handles false positive race reports.

6. Discussion

In this section, we discuss Portend's usage model and limitations.

Data Race Detection. Portend currently only handles data races with respect to the POSIX threads synchronization primitives thus not detecting data races that may occur inside synchronization primitives (as in [29]). Detection is done on the LLVM bitcode, not x86 assembly; analyzing the program at this level shields Portend from the peculiarities of a specific CPU's ISA and memory consistency model, while at the same time being ignorant of features that may make code be correctly synchronized on a specific platform despite not using POSIX primitives.

Multi-processors and Memory Consistency. Portend works for multi-processors, however it can simulate only serializable schedules that assume sequential memory consistency [32]. In order to augment Portend with the capability to reason about weaker memory consistency models, two key aspects of such models need to be handled, namely the relaxation of write atomicity and program order modification [1]. In order to model the relaxation of the write atomicity requirement, it is possible to break non-atomic instructions into multiple LLVM instructions. Then, Portend would be able to generate currently missing schedules that stem from non-atomic operations. Modeling the modification of program order can be achieved using a technique similar to adversarial memory [17]. Adversarial memory maintains a history buffer of writes per shared memory location and, upon a read from that memory location, computes a subset of values that can be validly read under the constraints of a particular relaxed consistency model. Portend can use those different read values to explore more paths during multi-path multi-schedule analysis.

Language Memory Models. Several language specifications, such as Ada 83 [34] and the new C [26] and C++ [25] specifications, do not guarantee any semantics for programs that contain data races at the source level, essentially considering all such races to be harmful. In other words, the corresponding compilers are allowed to perform optimizations that break racy programs in arbitrary ways [4]. Nonetheless, races at assembly level are not disallowed by any specification (typically synchronization primitives are implemented with racy assembly code), and, more importantly, there is a plethora of software written with races at the source level. Since Portend operates on LLVM bitcode, it can classify both source-code-level races and LLVM bitcode-level races. Moreover, Portend can also correctly classify not-buggy-in-source races that the compiler (legitimately) turned into LLVM level buggy ones [4].

Single-Processor Scheduler. Most data race detectors use a single processor thread scheduler and Portend does too. Such a scheduler has the advantage that it simplifies the record/replay component and improves its performance: for example, Portend does not have to record the ordering of memory reads and writes in different threads, since they are naturally ordered by scheduling (a single thread at a time). The scheduler does not impact Portend's ability to detect races, but it may decrease the probability of exploring some atomicity violations [36] or thread interleavings that would occur on a multi-processor (and are specific to its memory model). This loss of coverage is compensated by the fact that, if, during the analysis, Portend detects a racing access, it will consider it as a possible preemption point. Moreover, Portend's multi-schedule analysis is more likely to uncover more interleavings than "single-pre/single-post" analysis, thus increasing the thread schedule coverage.

Scalability of Symbolic Execution. Advances in improving the scalability of symbolic execution would help Portend explore more executions in the same amount of time, improving triaging accuracy. Since Portend is "embarrassingly parallel", performance could also be improved by running Portend in a cluster.

K-witness Harmless Races. It is theoretically undecidable to say if a race classified as "k-witness harmless" by Portend is indeed harmless unless *all* thread interleavings and *all* possible inputs were analyzed. However, as we found in our evaluation, in most cases, Portend produces highly accurate (but not perfect) verdicts in a reasonable amount of time.

We envision integrating Portend with an IDE to perform race classification in the background while the developer is modifying the code, in the spirit of automated software reliability services [10]. Portend would perform race classification and warn the developer that a race on the variable in question can have harmful effects, and should be protected. Test cases that are generated by Portend in the background can be integrated into a test suite for developers to validate their applications as part of regression tests.

Portend could also use reports from static race detectors [15] by feeding them to an execution synthesis tool like ESD [63], which can try to obtain the desired trace and then classify it with Portend. If ESD finds such an execution, it effectively confirms that the race is not a false positive, and Portend can automatically predict its consequences.

While analyzing a specific data race, Portend may also detect other unrelated races. This is a side effect of exploring various different thread schedules during classification. Portend automatically detects these new races, records their execution trace, and analyzes them subsequently, one after the other.

7. Related Work

Data race detection techniques can be broadly classified into two major categories: static data race detection [15, 58] and dynamic data race detection [49, 50, 62]. Static data race detection attempts to detect races by reasoning about source code, and dynamic race detection discovers races by monitoring particular execution of a program. Dynamic data race detection can further be classified into three categories: (1) detection using the happens-before relationship [5, 31, 40, 42, 46, 48, 50], (2) detection based on the lockset algorithm [49, 62], and (3) hybrid algorithms that combine these

two techniques [30, 47, 62]. Portend uses a happens-before algorithm.

Portend's race detector can benefit from various optimizations, such as hardware support [42, 48] that speeds up race detection, or sampling methods [5, 38] that reduce detection overhead.

Prior work on data race classification employs record/replay analysis [45], heuristics [29], detection of ad-hoc synchronization patterns [27, 55] or simulation of the memory model [17].

Record/replay analysis [45] records a program execution and tries to enforce a thread schedule in which the racing threads access a memory location in the reverse order of the original race. Then, it compares the contents of memory and registers, and uses a difference as and indication of potential harmfulness. Portend does not attempt an exact comparison, rather it symbolically compares outputs and explores multiple paths and schedules, all of which increase classification accuracy over replay-based analysis.

DataCollider [29] uses heuristics to prune predefined classes of likely-to-be harmless data races, thus reporting fewer harmless races overall. Portend does not employ heuristics to classify races, but instead employs precise analysis of the possible consequences of the data race.

Helgrind$^+$ [27] and Ad-Hoc Detector [55] eliminate race reports due to ad-hoc synchronization; Portend classifies such races as "single ordering". Detecting ad-hoc synchronizations or happens-before relationships that are generally not recognized by race detectors can help further prune harmless race reports, as demonstrated recently by ATDetector [28].

Adversarial memory [17] finds races that occur in systems with memory consistency models that are more relaxed than sequential consistency, such as the Java memory model [37]. This approach uses a memory that returns stale yet valid values for memory reads, in an attempt to crash target programs. This approach is useful for a race like DCL, which is harmless on x86 but can be harmful under weaker memory models. On the other hand, systems like Sober [7] and Relaxer [8] detect executions that do not obey sequential consistency under relaxed memory models while only exploring sequentially consistent executions. Either of these approaches can be used to enable Portend to reason about weaker memory consistency models.

RACEFUZZER [51] generates random schedules from a pair of racing accesses to determine whether the race is harmful. Therefore, RACEFUZZER performs multi-schedule analysis but not multi-path, and does so only with the goal of finding bugs, not classifying races. Portend uses multi-path analysis in addition to multi-schedule analysis to improve triage accuracy.

Output comparison was used by Pike to find concurrency bugs while fuzzing thread schedules [18]. Pike users can also write state summaries to expose latent semantic bugs that may not always manifest in the program output. If available, such state summaries could also be used in Portend.

Frost [57] follows a similar approach to Record/Replay Analyzer and Pike in that it explores complementary schedules (similar to primary and alternate schedules in Portend) and detects and avoids potentially harmful races comparing the program states after following these schedules. This detection is based on state comparison and therefore is prone to false positives as shown in §5.4. If used in conjunction with Portend, Frost could avoid provably harmful races.

Deterministic execution systems have recently gained popularity in academia [2, 3, 13, 14]. Deterministic execution requires making the program merely a function of its inputs [56]. In order to achieve this in the general case, races must be eliminated from programs, which leads to high overhead. Combined with Portend, it may be possible to relax determinism guarantees and eliminate races that really matter from the point of view of a developer or user, and make deterministic execution more practical.

8. Conclusion

This paper presents the first technique for triaging data races based on their potential consequences through an analysis that is both multi-path and multi-schedule. Triaging is done based on a new four-category data race classification scheme. Portend, an embodiment of our proposed technique, detected and classified 93 different data races in 7 real-world applications with 99% accuracy and full precision, with no human effort.

Acknowledgments

We thank the anonymous reviewers for providing insightful feedback and suggesting paths for future research. We thank Katerina Argyraki, Ryan Johnson, Olivier Crameri, Christopher Ming-Yee Iu, Sotiria Fytraki, and all our DSLAB colleagues for helping us improve this paper. We thank Microsoft for supporting Cristian Zamfir through an ICES grant.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 1996.

[2] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Symp. on Operating Sys. Design and Implem.*, 2010.

[3] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Core-Det: a compiler and runtime system for deterministic multithreaded execution. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.

[4] H.-J. Boehm. How to miscompile programs with "benign" data races. In *USENIX Workshop on Hot Topics in Parallelism*, 2011.

[5] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: proportional detection of data races. In *Conf. on Programming Language Design and Implem.*, 2010.

[6] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *ACM EuroSys European Conf. on Computer Systems*, 2011.

[7] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *Intl. Conf. on Computer Aided Verification*, 2008.

[8] J. Burnim, K. Sen, and C. Stergiou. Testing concurrent programs on relaxed memory models. In *Intl. Symp. on Software Testing and Analysis*, 2011.

[9] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Sys. Design and Implem.*, 2008.

[10] G. Candea, S. Bucur, V. Chipounov, V. Kuznetsov, and C. Zamfir. Automated software reliability services: Using reliability tools should be as easy as webmail. Symp. on Operating Sys. Design and Implem., 2010. Research Vision Session.

[11] V. Chipounov and G. Candea. Enabling sophisticated analyses of x86 binaries with RevGen. In *Intl. Conf. on Dependable Systems and Networks*, 2011.

[12] Chris Lattner. libc++. http://libcxx.llvm.org/ .

[13] H. Cui, J. Wu, C. che Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *Symp. on Operating Sys. Design and Implem.*, 2010.

[14] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2009.

[15] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Symp. on Operating Systems Principles*, 2003.

[16] B. Fitzpatrick. memcached. http://memcached.org/ .

[17] C. Flanagan and S. N. Freund. Adversarial memory for detecting destructive races. In *Conf. on Programming Language Design and Implem.*, 2010.

[18] P. Fonseca, C. Li, and R. Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *ACM EuroSys European Conf. on Computer Systems*, 2011.

[19] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Intl. Conf. on Computer Aided Verification*, 2007.

[20] J. Gilchrist. Parallel BZIP2. http://compression.ca/pbzip2 .

[21] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *Symp. on Operating Systems Principles*, 2009.

[22] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symp.*, 2008.

[23] P. Godefroid and N. Nagappan. Concurrency at Microsoft – An exploratory survey. In *CAV Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.

[24] Helgrind. http://valgrind.org/docs/manual/hg-manual.html .

[25] ISO/IEC 14882:2011: Information technology – programming languages – C++. International Organization for Standardization, 2011.

[26] ISO/IEC 9899:2011: Information technology – programming languages – C. International Organization for Standardization, 2011.

[27] A. Jannesari and W. F. Tichy. Identifying ad-hoc synchronization for enhanced race detection. In *Intl. Parallel and Distributed Processing Symp.*, 2010.

[28] Y. L. Jiaqi Zhang, Weiwei Xiong, S. Park, Y. Zhou, and Z. Ma. ATDetector: Improving the accuracy of a commercial data race detector by identifying address transfer. In *IEEE/ACM International Symposium on Microarchitecture*, 2011.

[29] S. B. John Erickson, Madanlal Musuvathi and K. Olynyk. Effective data-race detection for the kernel. In *Symp. on Operating Sys. Design and Implem.*, 2010.

[30] T. I. Konstantin Serebryany. ThreadSanitizer - data race detection in practice. In *Workshop on Binary Instrumentation and Applications*, 2009.

[31] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[32] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, Sep 1979.

[33] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Intl. Symp. on Code Generation and Optimization*, 2004.

[34] H. Ledgard. *Reference Manual for the ADA Programming Language*. Springer-Verlag New York, Inc., 1983.

[35] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, July 1993.

[36] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2006.

[37] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Symp. on Principles of Programming Languages*, 2005.

[38] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. In *Conf. on Programming Language Design and Implem.*, 2009.

[39] C. McPherson. Ctrace. http://ctrace.sourceforge.net .

[40] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing*, 1991.

[41] Memcached issue 127. http://code.google.com/p/memcached/issues/detail?id=127 .

[42] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1991.

[43] M. Musuvathi, S. Burckhardt, P. Kothari, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.

[44] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Symp. on Operating Sys. Design and Implem.*, 2008.

[45] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. *Conf. on Programming Language Design and Implem.*, 2007.

[46] A. Nistor, D. Marinov, and J. Torrellas. Light64: Lightweight hardware support for data race detection during systematic testing of parallel programs. In *IEEE/ACM International Symposium on Microarchitecture*, 2009.

[47] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Symp. on Principles and Practice of Paralle Computing*, 2003.

[48] M. Prvulovic and J. Torrellas. ReEnact: using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Intl. Symp. on Computer Architecture*, 2003.

[49] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4), 1997.

[50] E. Schonberg. On-the-fly detection of access anomalies (with retrospective). *SIGPLAN Notices*, 39(4), 2004.

[51] K. Sen. Race directed random testing of concurrent programs. *Conf. on Programming Language Design and Implem.*, 2008.

[52] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Symp. on the Foundations of Software Eng.*, 2005.

[53] SQLite. http://www.sqlite.org/ , 2010.

[54] The Associated Press. General Electric acknowledges Northeastern blackout bug. http://www.securityfocus.com/news/8032 .

[55] C. Tian, V. Nagarajan, R. Gupta, and S. Tallam. Dynamic recognition of synchronization operations for improved data race detection. In *Intl. Symp. on Software Testing and Analysis*, 2008.

[56] N. H. Tom Bergan, Joseph Devietti and L. Ceze. The deterministic execution hammer: How well does it actually pound nails? In *Workshop on Determinism and Correctness in Parallel Programming*, 2011.

[57] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. In *Symp. on Operating Systems Principles*, 2011.

[58] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static race detection on millions of lines of code. In *Symp. on the Foundations of Software Eng.*, 2007.

[59] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. *Intl. Symp. on Computer Architecture*, 1995.

[60] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad-hoc synchronization considered harmful. In *Symp. on Operating Sys. Design and Implem.*, 2010.

[61] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Distributed dynamic partial order reduction based verification of threaded software. In *Intl. SPIN Workshop*, 2007.

[62] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *Symp. on Operating Systems Principles*, 2005.

[63] C. Zamfir and G. Candea. Execution synthesis: A technique for automated debugging. In *ACM EuroSys European Conf. on Computer Systems*, 2010.