Automated Classification of Data Races Under Both Strong and Weak Memory Models

BARIS KASIKCI, CRISTIAN ZAMFIR, and GEORGE CANDEA, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Data races are one of the main causes of concurrency problems in multithreaded programs. Whether all data races are bad, or some are harmful and others are harmless, is still the subject of vigorous scientific debate [Narayanasamy et al. 2007; Boehm 2012]. What is clear, however, is that today's code has many data races [Kasikci et al. 2012; Jin et al. 2012; Erickson et al. 2010], and fixing data races without introducing bugs is time consuming [Godefroid and Nagappan 2008]. Therefore, it is important to efficiently identify data races in code and understand their consequences to prioritize their resolution.

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? Do they appear to be harmless? How do their effects change under weak memory models? 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 seven real-world applications: it detected 93 true data races and correctly classified 92 of them, with no human effort. Six of them were 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; D.4 [**Process Management**]: Concurrency

General Terms: Reliability, Verification, Security

Additional Key Words and Phrases: Data races, concurrency, triage, symbolic execution

ACM Reference Format:
Baris Kasikci, Cristian Zamfir, and George Candea. 2015. Automated classification of data races under both strong and weak memory models. ACM Trans. Program. Lang. Syst. 37, 3, Article 8 (May 2015), 44 pages.
DOI: http://dx.doi.org/10.1145/2734118

1. INTRODUCTION

Two memory accesses conflict if they access a shared memory location and at least one of the two accesses is a write. A data race occurs when two threads perform a conflicting access and the two accesses are not ordered using non–ad hoc synchronization—that is, custom synchronization devised by a developer that relies on loops to synchronize with shared variables. By ad hoc synchronization, we do not refer to a

This work is supported by the European Research Council, VMware, Microsoft through an ICES grant, and a Google Faculty Award.
Authors' addresses: B. Kasikci, EPFL IC DSLAB, INN 321, Station 14 1015 Lausanne, Switzerland; email: baris.kasikci@epfl.ch; C. Zamfir, EPFL IC DSLAB, INN 319, Station 14 1015 Lausanne, Switzerland; email: cristian.zamfir@epfl.ch; G. Candea, EPFL IC DSLAB, INN 330, Station 14 1015 Lausanne, Switzerland; email: george.candea@epfl.ch.

developer-provided correct implementation of a synchronization construct, but rather to incorrect synchronization operation implementations that frequently appear in real-world code [Xiong et al. 2010]. Even though ad hoc synchronization may be able to order the two accesses, recent research has demonstrated that it can break unexpectedly [Xiong et al. 2010]; therefore, we consider ad hoc synchronization as not eliminating data races in the general case.

Data races are some of the worst concurrency bugs, even leading to the loss of human lives [Leveson and Turner 1993] and causing massive material losses [Associated Press 2004]. As programs become increasingly parallel, we expect the number of data races that they contain to increase; as hardware becomes increasingly parallel, we expect the probability that both orderings of any given data race get exercised during normal executions to increase as well.

Moreover, recent C [ISO9899 2011] and C++ [ISO14882 2011] standards do not provide meaningful semantics for programs involving data races. As a consequence, the compiler is allowed to perform optimizations on code with data races that may transform seemingly benign data races into harmful ones [Boehm 2011].

Finally, if a program with data races was written with the assumption of a particular memory model in place, the program may no longer operate correctly under the actual memory model [Flanagan and Freund 2010].

Ideally, programs would have no data races at all. In this way, programs would avoid possible catastrophic effects due to data races. This either requires programs to be data race free by design, or it requires finding and fixing all data races in a program. However, modern software still has data races either because it was written carelessly, the complexity of the software made it very difficult to properly synchronize threads, or the benefits of fixing all data races using expensive synchronization did not justify its costs.

To construct programs that are free of data races by design, novel languages and language extensions that provide a deterministic programming model have been proposed [Thies et al. 2002; Bocchino et al. 2009]. Deterministic programs are race free, and therefore their behavior is not timing dependent. Even though these models may be an appropriate solution for the long term, the majority of modern concurrent software is written in mainstream languages such as C, C++, and Java, which do not provide any data race freedom guarantees.

To eliminate all data races in current mainstream software, developers first need to find them. This can be achieved by using data race detectors, which come in two main flavors: static detectors and dynamic detectors.

Static data race detectors [Engler and Ashcraft 2003; Voung et al. 2007] analyze the program source without actually running it. Therefore, they scale to large code bases, and they deliver results in a short amount of time. For example, RELAY [Voung et al. 2007] can analyze the Linux kernel (4.5 MLOC) and report data races in less than 5 hours. Static detectors tend to have fewer false negatives (i.e., fewer missed real data races) than dynamic data race detectors [O'Callahan and Choi 2003]. However, static data race detectors tend to have more false positives (i.e., reports that do not correspond to real data races). For example, 84% of the data races reported by RELAY are false positives.

Dynamic data race detectors have fewer false positives.[1] However, most dynamic race detectors can only detect races in program executions that they can monitor[2]; therefore,

[1]Dynamic detectors that use happens-before relationships do not have false positives as long as they are aware of the synchronization mechanisms employed by the developer. Dynamic detectors that use the lockset algorithm can have false positives [Savage et al. 1997].

[2]An exception to this is data race detection using causal precedence [Smaragdakis et al. 2012], which can accurately predict data races that do not occur during actual program executions.

they have many false negatives. They also tend to incur high runtime overhead (e.g., up to 200× in the case of Intel ThreadChecker [Intel Corp. 2012] and up to 8.5× in the case of FastTrack [Flanagan and Freund 2009]), because they need to monitor all memory accesses and synchronization primitives.

Even if all real races in a program are found using race detectors, eliminating all of them still appears impractical. First, synchronizing all racing memory accesses would introduce performance overheads that may be considered unacceptable. For example, developers have not fixed a race that can lead to lost updates in memcached for a year—ultimately finding an alternate solution—because it leads to a 7% drop in throughput [Memcached 2009].Performance implications led to 23 data races in Internet Explorer and Windows Vista being purposely left unfixed [Narayanasamy et al. 2007]. Similarly, several races have been left unfixed in the Windows 7 kernel, because fixing those races does not justify the associated costs [Erickson and Olynyk 2010].

Another reason data races go unfixed is that 76% to 90% of data races are actually considered to be harmless [Erickson and Olynyk 2010; Narayanasamy et al. 2007; Engler and Ashcraft 2003; Voung et al. 2007]—*harmless races* are assumed to not affect program correctness, either fortuitously or by design, whereas *harmful races* lead to crashes, hangs, resource leaks, and 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 [Godefroid and Nagappan 2008]), so time-pressed developers may not even attempt this high-investment/low-return activity.

Given the large number of data race reports (e.g., Google's Thread Sanitizer [Serebryany and Iskhodzhanov 2009] reports more than 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 race that has no observable harmful effects and left in the code perhaps for performance reasons?

Alas, automated classifiers [Jannesari and Tichy 2010; Erickson and Olynyk 2010; Narayanasamy et al. 2007; Tian et al. 2008] are often inaccurate (e.g., Narayanasamy et al. [2007] reports a 74% false-positive rate in classifying harmful races). To our knowledge, no data race detector/classifier can do this without false positives.

We propose Portend$^+$, a technique and tool that detects data races and, based on an analysis of the code, infers each race's potential consequences and automatically classifies the races into four categories: "specification violated," "output differs," "k-witness harmless," and "single ordering." In Portend$^+$, harmlessness is circumstantial rather than absolute; it implies that Portend$^+$ did not witness a harmful effect of the data race in question for k different executions.

For the first two categories, Portend$^+$ produces a replayable trace that demonstrates the effect, making it easy on the developer to fix the race.

Portend$^+$ operates on binaries, not on source code (more specifically on LLVM [Lattner and Adve 2004] bitcode obtained from a compiler or from a machine-code-to-LLVM translator like RevGen [Chipounov and Candea 2011]). 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 [ISO9899 2011] and C++ [ISO14882 2011]).

We applied Portend$^+$ to 93 data race reports from seven 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 (see Section 6.6). This improvement comes from Portend$^+$'s ability to perform multipath and multithread schedule analysis, as well as Portend$^+$'s fine-grained classification scheme. We found not only that multipath multischedule 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 (see Section 6.2).

Portend$^+$ extends our earlier data race classification tool—Portend—with the support for classifying data races under different memory models using a technique called *symbolic memory consistency modeling* (SMCM). To evaluate this technique, we added specific support for a variant of the weak memory model [Dubois et al. 1986]. We then performed data race classification for microbenchmarks for which the underlying memory model makes a difference with regard to data race classification. We show how SMCM enables more accurate classification under different memory models and evaluate its runtime and memory overheads. In a nutshell, SMCM allows Portend$^+$ to achieve 100% classification accuracy for classifying races under different memory models while incurring low overhead.

This article makes four contributions:

—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
—A *technique for predicting the consequences of data races* that combines multipath and multischedule analysis with symbolic program-output comparison to achieve high accuracy in consequence prediction, and thus classification of data races according to their severity
—A detailed analysis of what *synergies* arise from the combination of multipath and multischedule analysis with symbolic output comparison
—*Symbolic memory consistency modeling SMCM*, a technique that can be used to model various architectural memory models in a principled way to perform data race classification under those memory models.

Portend$^+$, a *practical dynamic data race detector and classifier* that implements this technique and taxonomy, 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. For example, 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 encouraging developers to ignore seemingly harmless races, but rather it improves programmers' productivity so they can correctly fix as many important data race bugs as possible. We name our tool Portend$^+$ because it portends the consequences of data races and improves upon its predecessor, Portend

The rest of the article is structured as follows. We describe our proposed classification scheme in Section 3, then present Portend$^+$'s design and implementation in Sections 4 and 5, respectively. We conduct an evaluation on real-world applications and benchmarks in Section 6. Then, we discuss limitations in Section 7 and related work in Section 8. We conclude the article in Section 9.

2. DATA RACE DEFINITION

In the previous section, we defined a data race to occur when a program performs conflicting memory accesses to the same memory location while these accesses are not ordered by non–ad hoc synchronization. This definition is central to Portend$^+$'s classification scheme that we introduce in Section 3, and therefore we elaborate on our definition in this section.

Thread T_1

```
void * work0 (void *arg) {
  ...
  globalx = ...;
  globalDone = true;
  ...
}
```

Thread $T_{2...i}$

```
void * work1 (void *arg) {
  ...
  while (!globalDone)
  ... = x;
  ...
}
```

Fig. 1. A simple example of ad hoc synchronization.

Thread T_1

```
void * work0 (void *arg) {
  ...
  globalx = ...;
  globalDone = true;
  ...
}
```

Thread $T_{2...i}$

```
void * work0 (void *arg) {
  bool temp = globalDone;
  while (!temp)
  ... = x;
  ...
}
```

Fig. 2. Compiler-optimized version of the code in Figure 1.

Ad hoc synchronization is defined as custom synchronization devised by a developer that relies on using ad hoc loops that synchronize with shared variables in an effort to synchronize threads. This synchronization scheme is commonly employed in widely used programs such as Apache, MySQL, and Mozilla [Xiong et al. 2010].

Although common, ad hoc synchronization is in fact not a correct means to synchronize variables, because to synchronize threads it relies on using racing accesses to shared variables. However, programs with racing accesses do not have well-defined semantics in C/C++ [ISO9899 2011; ISO14882 2011]. Consequently, the compiler can generate code that violates the developers' assumptions and not fulfill their initial intention to synchronize threads.

To clarify why ad hoc synchronization can fail to properly synchronize threads, consider the example in Figure 1, which is adapted from an in-depth treatment of the problem [Boehm and Adve 2012]. In the example, thread T_1 initializes a variable `globalx` to some value and sets a flag called `globalDone` to true. Any other thread in the program that reads `x` first checks whether `globalDone` is true.

The problem with this kind of "synchronization" is that a compiler can easily break this code and violate the developer's assumptions. Typically, the compiler will observe that the flag `globalDone` is not modified within the loop. Based on this observation, the compiler can generate code such as shown in Figure 2. In this case, whenever any thread $T_2 \cdots T_i$ starts, if `globalDone` is not set to true, the thread will enter an infinite loop and the developer's intention to synchronize threads will not be fulfilled.

Because of its potentially erroneous semantics, our definition of data races does not consider ad hoc synchronization as a proper means of synchronization. It is worth reiterating that by ad hoc synchronization, we solely refer to the construct that we define in this section. If a programmer properly implements a reader-writer lock by leveraging a mutex lock, we do not consider it as ad hoc.

Finally, for Portend$^+$ to recognize a developer-provided correct synchronization, it needs to be provided with the semantics of such synchronization (as is the case with many other race detection/classification tools). Portend$^+$ currently recognizes POSIX thread primitives as legitimate synchronization.

3. DATA RACE CLASSIFICATION

We now describe the key existing approaches to race classification (Section 3.1), the challenges they face and the idea behind Portend^{+} (Section 3.2), and our proposed classification scheme (Section 3.3).

3.1. Background and Challenges

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

Heuristic classification relies on recognizing specific patterns that correspond to races that are considered harmless. For instance, DataCollider [Erickson and Olynyk 2010] prunes data race reports that appear to correspond to updates of statistics counters and to 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 considered 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 harmless may be incorrect. More importantly, even data races that developers consider harmless may become harmful (e.g., cause a crash) for a different compiler and hardware combination [Boehm 2011].

Replay-based classification [Narayanasamy et al. 2007] starts from an execution that experienced one ordering of the racing accesses ("primary") and reruns 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 or otherwise likely to be harmless. Prior to Portend [Kasikci et al. 2012], this approach was the only automated classification technique for all kinds of data races.

This approach faces several challenges: differences in the memory and/or register state of the primary versus 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). Conversely, 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. Finally, it may be impossible to pursue the alternate interleaving, for example, because developer-provided legitimate synchronization enforces a specific ordering. In this case, replay-based classification [Narayanasamy et al. 2007] conservatively classifies the race as likely to be harmful. Such races may indeed be harmful, but this is not why replay-based classification classifies them as harmful, but because its classification technique cannot handle such races. These three challenges cause replay-based classification to have a 74% false positive rate in classifying harmful races [Narayanasamy et al. 2007].

Identification of ad hoc synchronization performs data race classification by finding custom synchronization operations in a program [Jannesari and Tichy 2010; Tian et al. 2008]. In such approaches, if a shared memory access is found to be protected via ad hoc synchronization, then it is considered as harmless because its accesses can occur in only one order.

The main limitation of these approaches is that they perform automated classification based on the presence or absence of only a single criterion: ad hoc synchronization. This makes the classification too coarse grained and less helpful than it seems (see Section 6).

Furthermore, existing approaches use heuristics or dynamic instrumentation to detect such cases with misclassification rates as high as 50% [Jannesari and Tichy 2010]. But when these approaches accurately identify ad hoc synchronization, it is unclear whether this synchronization is correctly implemented and the data race that the ad hoc synchronization "eliminates" is harmless in an absolute sense or not. Recent work has shown that ad hoc synchronization is tricky to get right [Xiong et al. 2010] and may often fail to provide proper synchronization.

Another shortcoming stems from the assumptions made by prior classification frameworks regarding the underlying memory model. Prior data race classification approaches [Kasikci et al. 2012] and multithreaded bug-finding tools [Babic and Hu 2008; Bucur et al. 2011; Godefroid et al. 2005] assume that the analyzed programs are run on a uniprocessor under sequential consistency. Thus, they make the following assumptions: (1) the threads are scheduled using a uniprocessor scheduler, (2) the updates to the shared memory are instantaneous,[3] and (3) the instructions in the program text are not reordered.

Based on the first assumption, prior analysis and classification tools typically control how threads are scheduled during the analysis, and they allow one thread to be scheduled at a time as if they were using a uniprocessor scheduler [Bucur et al. 2011]. In a modern multiprocessor, multiple threads may be scheduled to run concurrently on different CPU cores. This concurrent execution can cause the updates to the shared memory locations by a thread on a given core to not be immediately visible to a different thread on a different core.

The view of the memory structure of prior analysis and classification tools is flat (i.e., they do not model store buffers or the cache), and the updates to the global memory are propagated to all threads instantaneously. This is in stark contrast to how most modern computer architectures behave: an update to a memory location initially goes into a per-core store buffer. The contents of the store buffer may not immediately propagate to the main memory or even the cache; therefore, threads may not observe the updates in the same order. This delayed propagation effect is further amplified by multiprocessor schedulers that schedule multiple threads to run concurrently on different CPU cores.

Finally, prior analysis and classification tools do not consider instruction reordering. Instruction reordering can occur as a side effect of the buffering in hardware store buffers [Adve and Hill 1990] or as a result of compiler optimizations [Boehm 2007]. Due to the store buffer, unbuffered instructions may be executed earlier than buffered instructions despite occurring after the latter in the program text. Compilers can also reorder instructions by typically allowing loads to be executed earlier. This gives the programs more time to wait for the results of the loads that are crucial for the forward progress of programs.

The assumptions made by prior analysis and classification tools are compounded by the assumptions made by developers when writing programs. Typically, developers write their code with the assumption that it gets executed under sequential consistency, although they should not do so, as this is the more natural model for them. Programmers tend to expect that programs get executed with statements being run in the order specified in the program text by them, without considering potential reorderings [Ceze et al. 2007].

There seems to be a vicious circle. Programmers assume sequential consistency when writing their programs, and this could cause bugs. On the other hand, many program

[3]We do not refer to this property as atomicity. Atomic write instructions cannot be interrupted. Our current prototype does not handle nonatomic write instructions that can be interrupted (i.e., those that consist of multiple machine instructions). We discuss how such interruptible write instructions can be handled in Section 9.

testing and analysis tools on which the developers should rely to fix the bugs in their programs also assume sequential consistency. End results are undesirable: subtle bugs show up under rare circumstances, software ends up being less reliable, and so forth. This subtleness, when combined with even subtler bugs such as data races, presents real challenges to the developers [Flanagan and Freund 2010].

3.2. Our Approach to Detection and Classification

Note that in the rest of this article, we use the name Portend$^+$ to refer to both the techniques that we introduce and the prototype we built, depending on the context.

Portend$^+$ addresses three challenges left unsolved by prior work: (1) how to accurately distinguish harmful state differences from ones that do not have any observable harmful effect, (2) how to identify harmful races even when the state of the primary and alternate executions do not immediately differ, and (3) how to factor in the effect of the underlying memory model when classifying races.

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 and reason symbolically about the resulting state. These observations motivate Portend$^+$'s combination of multipath and multischedule analysis with symbolic output comparison (see Section 4). Thus, instead of employing heuristic-based classification as in previous work [Erickson and Olynyk 2010; Jannesari and Tichy 2010; Tian et al. 2008], Portend$^+$ symbolically executes the program to reason precisely about the possible consequences of races.

We also introduce SMCM, a technique that allows augmenting existing program analysis frameworks with the capability to reason about relaxed consistency models. SMCM is not specific to Portend$^+$ and can be employed in any program analysis tool. To implement SMCM, a tool needs to perform its analysis along all of the paths that can be followed by the program as a result of reading a shared memory location. The values that can be read from a shared memory location are in turn constrained by the memory model in place that the tool supports.

3.3. A New Classification Scheme

A simple harmless versus harmful classification scheme is undecidable in general (as will be explained later), so prior work typically resorts to "likely harmless" and/or "likely harmful." Alas, in practice, this is less helpful than it seems (see Section 6). 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 that it is a false positive. When a true race's consequences are deemed to be harmless for all the witnessed executions, we refer to it as a harmless race.[4]

A false positive is harmless in an absolute sense (since it is not a race to begin with), but not the other way around—harmless races are still true races. Static [Engler and Ashcraft 2003], lockset [Savage et al. 1997], and hybrid [O'Callahan and Choi 2003] data race detectors typically report false positives.

[4]In the rest of the article, whenever we mention harmless data races, we refer to this definition. If we refer to another definition adopted by prior work, we make the distinction clear.

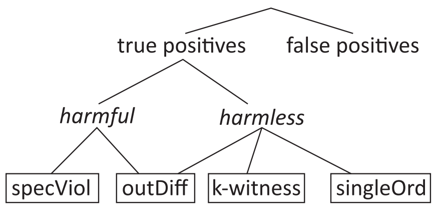


Fig. 3. Portend$^+$ taxonomy of data races.

If a data race does not have any observable effect (crash, hang, data corruption) for all the witnessed executions, we say that the data race is harmless with respect to those executions. Harmless races are still true races. Note that our definition of harmlessness is circumstantial; it is not absolute. It is entirely possible that a harmless race for some given executions can become harmful in another execution.

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 Figure 3.

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 that is beyond the application's control. Such races are often considered harmful: one of those outputs is likely the "incorrect" one. However, "output differs" races can also be considered as 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 and 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 nonexplicit specification or not. Moreover, whether the specification has been violated or not might even be subjective, depending on which developer is asked. It is for this reason that we created the "output differs" class of races: we provide developers a clear characterization of the output difference and let them decide using the provided evidence 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 [Cadar et al. 2008; Bucur et al. 2011] to analyze entire equivalence classes of inputs. 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 (manually verified) for all tested programs. The value of this category will become obvious after reading Section 4. We also evaluate the individual contributions of exploring paths versus schedules later in Section 6.2.

Single ordering are races for which only a single ordering of the accesses is possible, typically enforced via ad hoc synchronization [Xiong et al. 2010]. 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. Considering these to be nonraces is inconsistent with our definition (see Section 1) because the ordering of the accesses is not enforced using non–ad hoc synchronization primitives, even though it may not actually be 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 [Xiong et al. 2010]. Previous data race detectors generally cannot tell that only a single order is possible for the memory accesses and thus report this as an ordinary data race. Such data races can turn out to be both harmful [Xiong et al. 2010], or they can be a major source of harmless data races [Jannesari and Tichy 2010; Tian et al. 2008]. That is why we have a dedicated class for such data races.

4. 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 Figure 3. To achieve the classification, Portend$^+$ performs targeted analysis of multiple schedules of interest while at the same time using symbolic execution [Cadar et al. 2008] to simultaneously explore multiple paths through the program; we call this technique *multipath multischedule 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 that it compares constraints on program output as well as path constraints when these outputs are made, in addition to comparing the concrete values of the output, to generalize the comparison to more possible inputs that would bring the program to the specific data race and to determine if the data race affects the constraints on program output. 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 Section 4.1, we go over one such race (see Figure 6) and explain how Portend$^+$ handles it.

Figure 4 illustrates Portend$^+$'s architecture. Portend$^+$ is based on Cloud9 [Bucur et al. 2011], a parallel symbolic execution engine that supports running multithreaded C/C++ programs. Cloud9 is in turn based on Klee [Cadar et al. 2008], which is a single-threaded symbolic execution engine. Cloud9 has a number of built-in checkers for memory errors, overflows, and division-by-0 errors, on top of which Portend$^+$ adds an additional deadlock detector. Portend$^+$ has a built-in race detector that implements a dynamic happens-before algorithm [Lamport 1978]. This detector relies on a component that tracks Lamport clocks [Lamport 1978] at runtime (details are in Section 4.5). Portend$^+$'s analysis and classification engine performs multipath multischedule data race analysis and symbolic output comparison. This engine also works together with the Lamport clock tracker and the SMCM plugin to perform classification. The SMCM plugin defines the rules according to which a memory read operation from a shared memory location can return previously written values to that location. The SMCM plugin is crucial for classifying races under different memory consistency models.

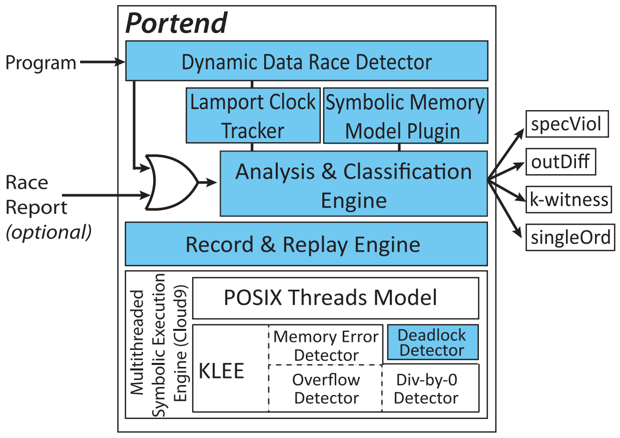


Fig. 4. High-level architecture of Portend$^+$. The six shaded boxes indicate new code written for Portend$^+$, whereas clear boxes represent reused code from KLEE [Cadar et al. 2008] and Cloud9 [Bucur et al. 2011].

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 (Section 4.1); describe the first step, single-path/single-schedule analysis (Section 4.2); and follow with the second step, multipath analysis and symbolic output comparison (Section 4.3) augmented with multischedule analysis (Section 4.4). We introduce SMCM and describe how it can be used to model various memory models while performing data race classification (Section 4.5). We describe Portend$^+$'s race classification (Section 4.6) and the generated report that helps developers debug the race (Section 4.7).

4.1. 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 [Lamport 1978]. 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 [Serebryany and Iskhodzhanov 2009] to create a Portend$^+$-compatible trace; we believe that such plugins can easily be developed for other dynamic race detectors [Helgrind 2012].

Portend$^+$ has a record/replay infrastructure for orchestrating the execution of a multithreaded 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 Section 7 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:

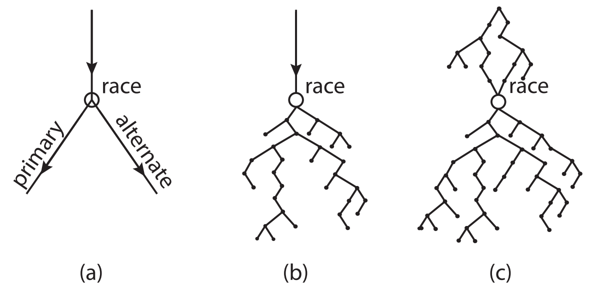


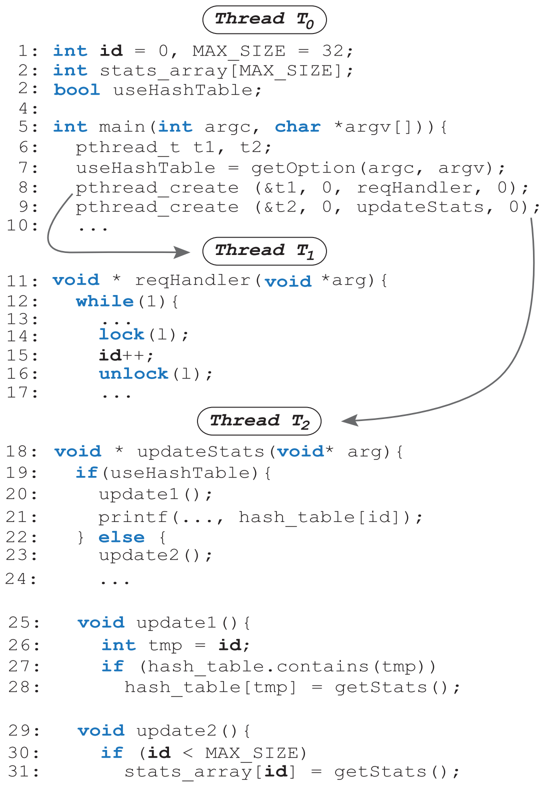
Fig. 5. Increasing levels of completeness in terms of paths and schedules: single-pre/single-post (a) $\ll$ single-pre/multipost (b) $\ll$ multipre/multipost (c).

$(T_0 : pc_0) \rightarrow (T_1 \rightarrow \mathit{RaceyAccess}_{T_1} : pc_1) \rightarrow (T_2 \rightarrow \mathit{RaceyAccess}_{T_2} : pc_2)$ 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 to perform precise replays when an instruction executes multiple times (e.g., a loop) before being involved in a race; for brevity, this is not shown as part of the schedule trace. The log of system call inputs contains the nondeterministic program inputs (e.g., *gettimeofday*).

In a first analysis step (illustrated in Figure 5(a)), 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 Section 3.1, some classifiers compare the primary and alternate program state immediately after the race and, if different, flag the race as potentially harmful, and, if same, flag the race as potentially harmless. Even if program outputs are compared rather than states, "single-pre/single-post" analysis (Figure 5(a)) may not be accurate, as we will show later. Portend$^+$ uses "single-pre/single-post" analysis mainly to determine whether the alternate schedule is possible. 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 postrace 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 prerace 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* postrace paths. This motivates "single-pre/multipost" analysis (Figure 5(b)), in which multiple postrace execution possibilities are explored—if any primary/alternate mismatch is found, the developer must be notified.

Even if all feasible postrace 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 prerace 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 prerace stage, not just the one path witnessed by the race detector. This is illustrated as "multipre/



```
 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:   ...

11: void * reqHandler(void *arg){
12:   while(1){
13:     ...
14:     lock(l);
15:     id++;
16:     unlock(l);
17:     ...

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();
```

Fig. 6. Simplified example of a harmful race from Ctrace [McPherson 2012] that would be classified as harmless by classic race classifiers.

multipost" analysis in Figure 5(c). The advantage of doing this versus considering these as different races is the ability to systematically explore these paths.

Finally, we combine multipath analysis with multischedule analysis, as 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 postrace execution in Figure 5(c) correspond to different paths that stem from both multiple inputs and schedules, as we detail in Section 4.4.

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 Figure 5. 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 multipath multischedule analysis over "single-pre/single-post" analysis, consider the code snippet in Figure 6, adapted from a real data race bug. This code has racing accesses to the global variableid. 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 pdate1 or update2 functions (lines 20 through 23).

Say the program runs in Portend$^+$ with input–use-hash-table, which makes useHashTable=true. Portend$^+$ records the primary trace $(T_0 : pc_9) \rightarrow \cdots (T_1 \rightarrow \mathit{RaceyAccess}_{T_1} : pc_{15}) \rightarrow (T_2 \rightarrow \mathit{RaceyAccess}_{T_2} : pc_{26}) \rightarrow \cdots\ T_0$. This trace is fed to the first analysis step, which replays the trace with the same program input, except it enforces the alternate schedule $(T_0 : pc_9) \rightarrow \cdots (T_2 \rightarrow \mathit{RaceyAccess}_{T_2} : pc_{26}) \rightarrow (T_1 \rightarrow \mathit{RaceyAccess}_{T_1} : pc_{15}) \rightarrow \cdots\ T_0$. 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 multipath multischedule step, Portend$^+$ explores additional paths through the code by marking program input as symbolic—that is, 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 the execution 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 $(T_0 : pc_9) \rightarrow \cdots (T_2 \rightarrow \mathit{RaceyAccess}_{T_2} : pc_{30}) \rightarrow (T_1 \rightarrow \mathit{RaceyAccess}_{T_1} : pc_{15}) \rightarrow (T_2 : pc_{31})$. 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: Sections 4.2 through 4.4 focus on the exploration part of the analysis, in which Portend$^+$ looks for paths and schedules that reveal the nature of the race, and Section 4.6 focuses on the *classification* part.

4.2. 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 [Musuvathi et al. 2008]) with a dynamic data 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 prerace checkpoint (line 2). The replay is then allowed to continue until immediately after the second racing access of the race in which we are interested (line 3), and the primary execution is suspended in this postrace state (line 4).

Portend$^+$ then primes a new execution with the prerace checkpoint (line 5) and attempts to enforce the alternate ordering of the racing accesses. To enforce this

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

```
Input: Primary execution trace primary
Output: Classification result ∈ {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
```

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 $\cdots$ ($T_i \rightarrow RaceyAccess_{T_i} : pc_1) \rightarrow (T_j \rightarrow RaceyAccess_{T_j} : pc_2) \cdots$ is steered toward the execution $\cdots (T_j \rightarrow RaceyAccess_{T_j} : pc_2) \rightarrow (T_i \rightarrow RaceyAccess_{T_i} : pc_1) \cdots$

This attempt could fail for one of three reasons: (a) T_j gets scheduled but T_i cannot be scheduled again; or (b) T_j gets scheduled but $RaceyAccess_{T_j}$ cannot be reached because of a complex locking scheme that requires a more sophisticated algorithm [Kahlon et al. 2005] than Algorithm 1 to perform careful scheduling of threads; or (c) 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. Portend$^+$ does not implement the more complex algorithm mentioned in (b), and this may cause it to have false positives in data race classification. However, we have not seen this limitation impact the accuracy of data race classification for the programs in our evaluation. Case (c) 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," whereas the ad hoc synchronization case classifies the race as "single ordering" (more details in Section 4.6). Although 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.

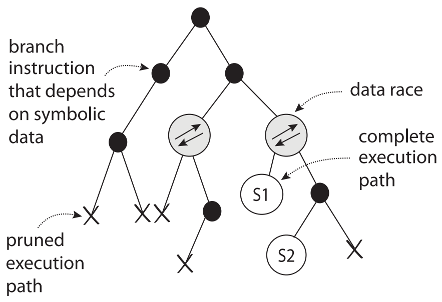


Fig. 7. Portend$^+$ prunes paths during symbolic execution.

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 [Narayanasamy et al. 2007], which compares the program state immediately after the race in the primary and alternate interleavings.

4.3. Multipath 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) to expose Portend$^+$ to a wider range of execution alternatives.

First, Portend$^+$ finds multiple primary paths that satisfy the input trace—in other words, they (1) all experience the same thread schedule (up to the data race) as the input trace and (2) 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—as well as path constraints when these outputs are made, and compares them to the outputs and path constraints of the corresponding alternate paths; we explain this later. 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 [Cadar et al. 2008] 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 Figure 7. 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 [Cadar et al. 2008; Bucur et al. 2011] is beyond the scope of this article.

An important concern in symbolic execution is "path explosion"—in other words, 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 versus classification confidence. For parameter (b), 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. In Section 6, we empirically study 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 (Figure 7) 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 Figure 6). Tolerating these divergences significantly increases Portend$^+$'s accuracy over the state of the art [Narayanasamy et al. 2007], as will be explained in Section 6.4.

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 [Ganesh and Dill 2007] to find concrete inputs that drive the program down the corresponding path. For example, in the case of Figure 7, 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.

4.3.1. Symbolic Output Comparison. Portend$^+$ now records the output of each of the M_p executions, as in the single-pre/single-post case, and it also records the path constraints when these outputs are made. However, this time, in addition to simply recording concrete outputs, Portend$^+$ propagates the constraints on symbolic state all the way to the outputs—that is, 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). The alternate executions are fully concrete, but Portend$^+$ records constraints on the alternate's outputs (lines 19 through 21) as well as the path constraints when these outputs are made. The function *singleClassify* in Algorithm 2 performs the analysis described in Algorithm 1. Portend$^+$ then checks whether the constraints on outputs of each alternate and the path constraints when these outputs are made match the constraints of the corresponding primary's outputs and the path constraints when primary's outputs are made. This is what we refer to as symbolic output comparison (line 22). The purpose behind comparing symbolic outputs is that Portend$^+$ tries to figure out if the data race caused the

ALGORITHM 2: Multipath Data Race Analysis (Simplified)

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

constraints on the output to be modified or not, and the purpose behind comparing path constraints is to be able generalize the output comparison to more possible executions with different inputs.

This symbolic comparison enables Portend$^+$'s analysis to extend over more possible primary executions. To see why this is the case, consider the example in Figure 8. In this example, the *Main* thread reads input to the shared variable i and then spawns two threads T_1 and T_2, which perform racing writes to *globalx*. T_1 prints 10 if the input is positive. Let us assume that during the symbolic execution of the primary, the write to *globalx* in T_1 is performed before the write to *globalx* in T_2. Portend$^+$ records that the output at line 6 is 10 if the path constraint is $i \geq 0$. Let us further assume that Portend$^+$ runs the program while enforcing the alternate schedule with input 1. The output of the program will still be 10 (since $i \geq 0$), and the path constraint when the output will be made will still be $i \geq 0$. The output and the path constraint when the output is made is therefore the same regardless of the order in which the accesses to *globalx* are performed (i.e., the primary or the alternate order). Therefore, Portend$^+$ can assert that the program output for $i \geq 0$ will be 10 regardless of the way in which the data race goes even though it only explored a single alternate ordering with input 1.

```
int globalx = 0;
int i = 0;
                  Thread T1
void * work0 (void *arg) {
  globalx = 1;
  if(i >= 0)
     printf("10\n");
  return 0;
}
                  Thread T2
void * work1 (void *arg) {
  globalx = 2;
  return 0;
}
                  Main Thread
int main (int argc, char *argv[]){
  pthread_t t0, t1;
  int rc;
  i = getInput(argc, argv)
  rc = pthread_create (&t0, 0, work0, 0);
  rc = pthread_create (&t1, 0, work1, 0);
  pthread_join(t0, 0);
  pthread_join(t1, 0);
  return 0;
}
```

Fig. 8. A program to illustrate the benefits of symbolic output comparison.

This comes at the price of potential false negatives because path constraints can be modified due to a different thread schedule; despite this theoretical shortcoming, we have not encountered such a case in practice, but we plan to investigate further in future work.

False negatives can also arise because 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>*).

When executing the primaries and recording their outputs and the path constraints, Portend$^+$ relies on Cloud9 to track all symbolic constraints on variables. To determine if the path constraints and constraints on outputs match for the primary and the alternates, Portend$^+$ directly employs an SMT solver [Ganesh and Dill 2007].

As will be seen in Section 6.2, using symbolic comparison in conjunction with multipath multischedule analysis 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 preceding algorithm.

4.4. Multischedule Data Race Analysis

The goal of multischedule 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 postrace 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 $A_i^1, A_i^2, \ldots$ 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 [Musuvathi et al. 2010] 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 Section 4.3.1 to establish equivalence between the constraint on outputs and path constraints of $A_i^1, A_i^2, \ldots A_i^{M_a}$ and the symbolic outputs and path constraints of P_i.

In addition, schedule randomization can be employed in the prerace 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 that we obtain with the current design appears to be sufficient in practice to achieve high accuracy. However, note that, as we show in Section 6.2, multipath multischedule analysis is indeed crucial to attaining high classification accuracy.

In summary, multipath multischedule 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.

4.5. Symbolic Memory Consistency Modeling

Modern processor architectures rarely assume sequential consistency, as this would hurt program performance. Instead, they adopt relaxed memory consistency models such as weak ordering [Dubois et al. 1986] and rely on the programmers to explicitly specify orderings among program statements using synchronization primitives.

Previous work has shown that subtle bugs may arise in code with data races because programmers make assumptions based on sequential consistency. Despite the fact that none of the modern processors provide sequential consistency [Flanagan and Freund 2010]. Such assumptions may be violated under relaxed consistency models, and bugs that are deemed unlikely may appear when the program is running on various CPUs, causing programs to crash, hang, or violate some given specification of a program.

Therefore, a program analysis tool should ideally have the capability to reason about different memory models and their effects on the performed analysis. The effect of the memory model on the consequences of a data race are serious: code written with the assumption of a particular memory model may end up computing wrong results—or worse, it can crash or cause data loss [Flanagan and Freund 2010].

4.5.1. Why Does the Memory Model Matter?

To better show why reasoning about relaxed memory consistency models matters while performing program analysis and testing, let us consider the example in Figure 9. There are two shared variables `globalx` and `globaly` that both have an initial value of 0. There is a *Main* thread that spawns two threads T_1 and T_2. T_1 writes 2 to a global variable `globalx` and 1 to another global variable `globaly`. T_2 writes 2 to `globalx`. Then, the *Main* thread reads the value of the global variables. If the read value of `globalx` and `globaly` are 0 and 1, respectively, the program crashes on line 18.

```
1:  int volatile globalx = 0;
2:  int volatile globaly = 0;

                Thread T1

3:  void * work0 (void *arg) {
4:    globalx = 2;
5:    globaly = 1;
6:    return 0;
7:  }
                Thread T2

8:  void * work1 (void *arg) {
9:    globalx = 2;
10:   return 0;
11: }
               Main Thread

12: int main (int argc, char *argv[]){
13:   pthread_t t0, t1;
14:   int rc;
15:   rc = pthread_create (&t0, 0, work0, 0);
16:   rc = pthread_create (&t1, 0, work1, 0);
17:   if(globalx == 0 && globaly == 1)
18:     ; //crash!
19:   pthread_join(t0, 0);
20:   pthread_join(t1, 0);
21:   return 0;
22: }
```

Fig. 9. Simple multithreaded program.

Programmers naturally expect the program statements to be executed in the order as they appear in the program text. A programmer making that assumption expects that the value of `globaly` being 1 implies the value of `globalx` being 2. This assumption is equivalent to assuming sequential consistency as the underlying memory model: if sequential consistency is assumed as the underlying memory model for the execution of this program, the value of `globalx` cannot be 0 when the value of `globaly` is 1. This is simply because the order of the program text would require `globalx` to be 2.

Under a different memory model, such as weak ordering [Dubois et al. 1986], nothing prevents the write to `globalx` on line 4 and the write to `globaly` on line 5 to swap places. This stems from the fact that under weak consistency, if instructions are not conflicting (see Section 1) and they are not ordered by synchronization operations, then any reordering is allowed. In such a scenario, it is possible for T_1 to write 1 to `globaly` while the value of `globalx` is still 0. Furthermore, there is a race between the write to `globalx` in T_1 and the read from it in *Main*. This means that T_1 can be preempted right after setting `globaly` to 1 and `globaly` and `globalx` can be equal to 1 and 0, respectively. This can cause the program to crash on line 18.

4.5.2. Limitations of Cloud9 as a Multithreaded Testing Platform. Portend$^+$ is built on Cloud9, which is a multithreaded parallel symbolic execution engine [Bucur et al. 2011]. Cloud9 is essentially an LLVM interpreter that can concretely interpret programs compiled to LLVM, and during this interpretation, it can also keep track of the symbolic values and constraints.

Cloud9 makes the following sequential consistency assumptions: (1) uniprocessor scheduler: the Cloud9 scheduler picks threads in a round robin fashion and runs them by interpreting their text until an opportunity for scheduling arises (e.g., a sleep or a synchronization operation); (2) immediate updates to shared memory: shared memory is modeled as a flat structure with no cache model; therefore, any update to a shared

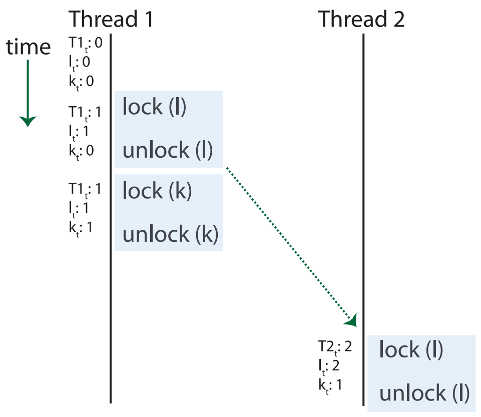


Fig. 10. Lamport clocks and a happens-before graph.

memory location is immediately visible to all other thread; and (3) no instruction reordering: the Cloud9 interpretation engine works by fetching instructions from the LLVM binary and executing them sequentially without any instruction reordering.

Since shared memory updates are not reordered, and they are directly visible to all threads, and threads are scheduled one after the other, one at a time, any analysis that builds on Cloud9 is bound to perform the analysis within the confines of sequential consistency. However, as was demonstrated previously, such an analysis may unable to expose insidious bugs.

4.5.3. Symbolic Memory Consistency Modeling in Portend$^+$. Previously, we showed how Portend$^+$ explores multiple paths and schedules to observe the consequences of a data race. The goal of SMCM is to further augment multipath multischedule analysis to factor in the effects of the underlying memory model. SMCM is in essence similar to multipath analysis: multipath analysis explores multiple execution paths that the execution could take due to different program input values; SMCM explores multiple paths that the execution could take due to different values that could be read from the shared memory.

SMCM has two main components: the Lamport clock tracker and the SMCM plugin.

The first component is the Lamport clock tracker. Lamport clocks are logical counters that maintain a partial order among synchronization operations to determine the relative occurrence sequence of events in a concurrent program [Lamport 1978]. This order is partial because an order is only present among related synchronization operations (e.g., a lock and an unlock on the same lock).

Lamport clocks are maintained per synchronization operation. A thread's Lamport clock is equal to the greatest of the clocks of all events that occur in the thread. Lamport clocks are incremented under the following conditions:

—Each thread increments the clock of an event before the occurrence of that event in that thread.
—When threads communicate, they also communicate their clocks (upon fork/join or wait/signal).
—The thread that receives a clock sets its own clock to be greater than the maximum of its own clock or that of the received message.

The graph that captures the relative order of events in a concurrent program using Lamport clocks is called a happens-before graph. Figure 10 shows an example happens-before graph and the Lamport clocks associated with a given execution. Note that `locks`

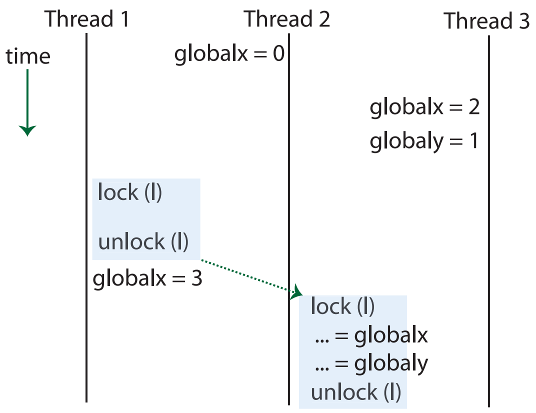


Fig. 11. Write buffering.

and `unlocks` on lock l induce a happens-before edge denoted by the arrow between Thread 1 and Thread 2. On the other hand, the lock/unlock block on lock k does not have any partial order with any of the events in Thread 2; therefore, although the current timeline shows it as occurring before the lock/unlock block in Thread 2, in some other execution it can occur after that lock/unlock block.

The Lamport clock tracker needs to monitor the synchronization operations performed by each thread to construct the happens-before graph. All synchronization events are intercepted, and the happens-before graph is constructed behind the scenes according to the rules that were stated previously while the program is being executed.

The Lamport clock tracker is a critical component of Portend$^+$, as it actually forms a well-defined ordering among events during program execution that stem from synchronization operations. This is important because different memory models and reordering constraints are defined using synchronization operations in programs.

The second component, namely the SMCM plugin, defines the memory model protocol according to which a read returns previously written values.

The memory model protocol encodes the rules of the particular memory model that Portend$^+$ uses. In our case, we define two such protocols: one default protocol for sequential consistency and another one for weak consistency. We described the semantics of sequential consistency in Section 3.1. Under Portend's weak memory consistency, a read R may see a previous write A, provided that there is no other write B such that B happens before R and A happens before B, with the exception that within the same thread, a sequence of reads from the same variable with no intervening writes to that variable will read the same value as the first read. We call this exception in Portend's weak memory model write buffering. Write buffering is responsible for keeping a write history for each shared memory location. Write buffering enables Portend$^+$ to compute a subset of the values written to a memory location when that location is read. That subset is computed considering the happens-before graph that is generated during program execution by the Lamport clock tracker.

Similar forms of weak memory consistency has been implemented in architectures such as SPARC [Weaver and Germond 1994] and Alpha [Sites 1992].

To see how write buffering and the memory model protocol work, consider the example given in Figure 11. Again, vertical order of events imply the order of events in time. In this particular execution, Thread 2 writes 0 to `globalx`, then execution switches to Thread 3, which writes 2 to `globalx` and 1 to `globaly` before the execution switches to Thread 1. Then, Thread 1 writes 3 to `globalx` after a lock/unlock region on l and finally execution switches back to Thread 2, which reads both `globalx` and `globaly` while holding the same lock l.

So what values do Thread 2 read? Note that since both `globalx` and `globaly` are shared variables, the CPU can buffer all of the values that were written to `globalx` $(0, 2, 3)$ and `globaly` (1). For `globaly`, the only value that can be read is 1. Now, when the value `globalx` is read, Portend$^+$ knows that, under its weak consistency model, the values that can be read are $0, 2$, and 3. This is because there is no ordering constraint (a happens-before edge) that prevents from making those three write values readable at the point of the read.

Then, Portend$^+$ will use these multiple possible reads to augment multipath analysis: Portend$^+$ will split the execution to as many possible "read" values there are by checkpointing the execution state prior to the read and binding each one of those possible reads to one such checkpointed state's thread. By binding a read value, we mean copying the value in question into the state's memory. Therefore, in this case, there will be three such forked states: one with values $(0, 1)$, one with $(2, 1)$, and the other with values $(3, 1)$ corresponding to (`globalx`, `globaly`). Portend$^+$ will continue exploring the forked states, forking further if their threads read global variables that can potentially return multiple values.

If an already-bound global value is read by the same thread in a state without being altered after the last time it had been read, Portend$^+$ makes sure to return the already-bound value. This is a necessary mechanism to avoid false positives (a thread reading two different values in a row with no intervening writes) due to write buffering. This is achieved by maintaining a last reader thread ID field per write buffer.

Write buffering and the optimization that we use to bind a read value to a thread are performed for a given thread schedule that Portend$^+$ explores at a time. For example, in Figure 11, if Thread 2 were to read `globalx` twice, it would have been possible for the first read to return 2 and the second read to return 3 (or vice versa) if there had been an intervening write between the two reads. Portend$^+$ relies on multischedule data race analysis to handle this case rather than relying on SMCM to reason about potential prior thread schedules that would lead to such a behavior.

This example demonstrates the power of SMCM in reasoning about weak ordering. Note that if sequential consistency were assumed for the given sequence of events, there would not have been a scenario where the value of `globaly` is 1 whereas the value of `globalx` is 0. This is because the given sequence of events would imply that writing 2 to `globalx` in Thread 3 occurs before writing 1 to `globaly` in the same thread. However, this is not the case under weak consistency. Since there is no synchronization enforcing the ordering of the write to `globalx` and `globaly` in Thread 3, these accesses can be reordered. Therefore, it is perfectly possible for Thread 2 to see the value of `globaly` as 1 and `globalx` as 0.

4.6. 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 be of highest priority to 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 Xiong et al. [2010] 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 Section 3.1, a difference between the postrace 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.

In *k-witness harmless* races, if for every primary execution P_i the constraints on the outputs of alternate executions $A_i^1, A_i^2 \cdots A_i^{M_a}$ and the path constraints when these outputs are made match P_i's output and path 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 Section 3.3, 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 article. 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 of 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 Musuvathi et al. [2010]. 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 may be harmless races if the ad hoc synchronization is properly implemented. In that case, one might even argue that 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, and our definition of a data race (see Section 1) considers these reports as races.

When Portend$^+$ cannot enforce an alternate interleaving in the single-pre/single-post phase, this can be due to one of two things: (1) ad hoc synchronization that prevents the alternate ordering or (2) the 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 can in some cases identify ad hoc synchronization [Xiong et al. 2010; Tian et al. 2008].

4.7. 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

```
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
```

Fig. 12. Example debugging aid report for Portend$^+$.

aims to help in fixing harmful races by providing two items for each race: 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 Figure 12.

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).

5. IMPLEMENTATION

The current Portend$^+$ prototype consists of approximately eight KLOC of C++ code, incorporating the various analyses described earlier and modifications to the underlying symbolic execution engine. The six shaded components in Figure 4 were developed from scratch as part of Portend$^+$: the dynamic data race detector, Lamport clock tracker, SMCM plugin, analysis and classification engine, the record-replay engine, and the deadlock detector.

Portend$^+$ works on programs compiled to LLVM [Lattner and Adve 2004] bitcode and can run C/C++ programs for which there exists a sufficiently complete symbolic POSIX environment [Bucur et al. 2011]. 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 [Lattner 2012]. Portend$^+$ uses Cloud9 [Bucur et al. 2011] to interpret and symbolically execute LLVM bitcode; we suspect any path exploration tool will do (e.g., CUTE [Sen et al. 2005], SAGE [Godefroid et al. 2008]), as long as it supports multithreaded 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$^+$ uses a dynamic race detector that is based on the happens-before relationship [Lamport 1978]. The happens-before relationships are tracked using the Lamport clock tracker component. This component, together with the SMCM plugin, is used by the analysis and classification engine to perform classification under a particular memory model.

Portend$^+$ keeps track of Lamport clocks per execution state on the fly. Note that it is essential to maintain the happens-before graph per execution state because threads may get scheduled differently depending on the flow of execution in each state, and therefore synchronization operations may end up being performed in a different order.

The state space exploration in Portend$^+$ is exponential in the number of values that reads can return in a program. Therefore, the implementation needs to handle bookkeeping as efficiently as possible. There are several optimizations that are in place to enable a more scalable exploration. The most important ones are (1) copy-on-write for keeping the happens-before graph and (2) write buffer compression.

Portend$^+$ employs copy-on-write for tracking the happens-before graphs in various states. Initially, there is a single happens-before graph that gets constructed during program execution before any state is forked due to a read with multiple possible return values. Then, when a state is forked, the happens-before graph is not duplicated. The forking state rather maintains a reference to the old graph. Then, when a new synchronization operation is recorded in either one of the forked states, this event is recorded as an incremental update to a previously saved happens-before graph. In this way, maximum sharing of the happens-before graph is achieved among forked states.

The copy-on-write scheme for states can further be improved if one checks whether two different states perform the same updates to the happens-before graph. If that is the case, these updates can be merged and saved as part of the common happens-before graph. This feature is not implemented in the current prototype, but it is a potential future optimization.

The second optimization is write buffer compression. This is performed whenever the same value is written to the same shared variable's buffer and the constraints imposed by the happens-before relationship allow these same values to be returned upon a read. Then, in such a case, these two writes are compressed into one, as returning two of them would be redundant from the point of view of state exploration. For example, if a thread writes 1 to a shared variable `globalx` twice before this value is read by another thread, the write buffer will be compressed to behave as if the initial thread has written 1 once.

Portend$^+$ clusters the data races that it detects 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 five 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.

To run multithreaded 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.

6. 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 (Section 6.1)? How accurately does it classify race reports into the four categories of races (Section 6.2)? How long does classification take, and how does it scale (Section 6.3)? How does Portend$^+$ compare to the state of the art in race classification (Section 6.4)? How effectively and efficiently does Portend$^+$ implement SMCM, and what is its memory overhead (Sections 6.5 and 6.6)? Throughout this section, we highlight the synergy of the techniques used in Portend$^+$: in particular, Section 6.1 shows how symbolic output comparison allows more accurate race classification compared to postrace state

Table I. Programs Analyzed with Portend$^+$

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
no-sync	45	C++	3
no-sync-bug	46	C++	3
sync	47	C++	3
sync-bug	48	C++	3

Note: Source lines of code are measured with the `cloc` utility.

comparison, and Section 6.2 shows how the combination of multipath multischedule analysis improves upon traditional single-path analysis.

To answer these questions, we apply Portend$^+$ to seven applications: SQLite, an embedded database engine (e.g., used by Firefox, iOS, Chrome, and Android) that is considered highly reliable, with 100% branch coverage [SQLite 2013]; Pbzip2, a parallel implementation of the widely used bzip2 file compressor [Gilchrist 2013]; Memcached [Fitzpatrick 2013], a distributed memory object cache system (e.g., used by services such as Flickr, Twitter, and Craigslist); Ctrace [McPherson 2012], a multithreaded debug library; Bbuf [Yang et al. 2007], a shared buffer implementation with a configurable number of producers and consumers; Fmm, an n-body simulator from the popular SPLASH2 benchmark suite [Woo et al. 1995]; and Ocean, a simulator of eddy currents in oceans, from SPLASH2.

Portend$^+$ classifies with 99% accuracy the 93 known data races that we found in these programs, with no human intervention, in less than 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 by examining the program as well as the generated assembly. This is typical of how long it takes to classify races in the absence of an automated tool [Godefroid and Nagappan 2008] 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 microbenchmarks that capture most classes of races considered as harmless in the literature [Serebryany and Iskhodzhanov 2009; Narayanasamy et al. 2007]: “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. Additionally, we have four other microbenchmarks that we used to evaluate the SMCM.We detail those microbenchmarks in Section 6.5. Table I summarizes the properties of our 15 experimental targets.

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

Table II. "Spec Violated" Races and Their Consequences

Program	Total Races (#)	"Spec Violated" Races (#)		
		Deadlock	Crash	Semantic
SQLite	1	1	0	0
pbzip2	31	0	3	0
ctrace	15	0	1	0
Manually Inserted Errors				
fmm	13	0	0	1
memcached	18	0	1	0

them here. For all experiments, the Portend$^+$ parameters were set to $M_p = 5$, and $M_a = 2$, and the number of symbolic inputs was set 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 2.4GHz Intel Core 2 Duo E6600 CPU with 4GB of RAM running Ubuntu Linux 10.04 with kernel version 2.6.33. The reported numbers are averages over 10 experiments.

6.1. Effectiveness

Of the 93 distinct races detected in seven real-world applications, Portend$^+$ classified five of them as definitely harmful by watching for "basic" properties (Table II): one hangs the program and four crash it.

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 sixth "harmful" race in Table II; 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 II are the only known harmful races in these applications.

6.2. Accuracy and Precision

To evaluate Portend$^+$'s accuracy and precision, we had it classify all 93 races in our target applications and microbenchmarks. Table III 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 II minus the semantic race in fmm and

Table III. Summary of Portend$^+$'s Classification Results

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

Note: 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. Portend$^+$ uses the stack traces and the program counters of the threads making the racing accesses to identify distinct races. The last five columns classify the distinct races. The States Same/States Differ columns show for how many races the primary and alternate states were different after the race, as computed by the Record/Replay Analyzer [Narayanasamy et al. 2007].

the race we introduced in memcached. In the "k-witness harmless" column, we show for which races the postrace states differed versus 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 the classification for each race 10 times. Portend$^+$ consistently reported the same dataset shown in Table III, which indicates that for these races and applications, it achieves full precision.

As can be seen in the *K-Witness Harmless* column, for each and every one of the seven real-world applications, a state difference (as used in Narayanasamy et al. [2007]) does not correctly predict harmfulness, whereas our "k-witness harmless" analysis correctly predicts that the races are harmless with one exception.

This suggests that the 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.

Multipath multischedule exploration proved to be crucial for Portend$^+$'s accuracy. Figure 13 shows the breakdown of the contribution of each technique used in Portend$^+$: ad hoc synchronization detection, multipath analysis, and multischedule 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 multipath multischedule exploration that revealed an output difference (9 races required multipath analysis for classification, and 8 races required multischedule analysis as well). Without multipath multischedule analysis, it would have been impossible for Portend$^+$ to accurately classify those races by just using the

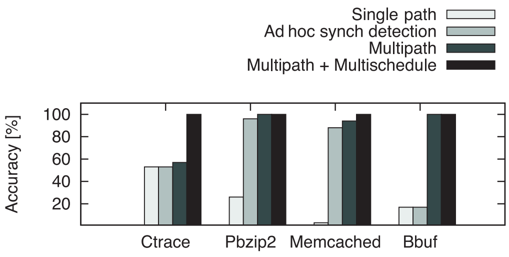
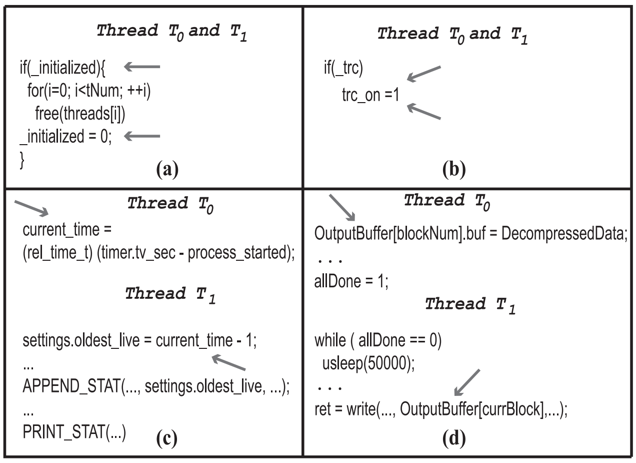


Fig. 13. Breakdown of the contribution of each technique toward Portend$^+$'s accuracy. We start from single path analysis and enable the other techniques one by one: ad hoc synchronization detection, multipath analysis, and finally multischedule analysis.



(a) Thread T_0 and T_1

```
if(_initialized){
  for(i=0; i<tNum; ++i)
    free(threads[i])
_initialized = 0;
}
```

(b) Thread T_0 and T_1

```
if(_trc)
    trc_on =1
```

(c) Thread T_0

```
current_time =
(rel_time_t) (timer.tv_sec - process_started);
```

Thread T_1

```
settings.oldest_live = current_time - 1;
...
APPEND_STAT(..., settings.oldest_live, ...);
...
PRINT_STAT(...)
```

(d) Thread T_0

```
OutputBuffer[blockNum].buf = DecompressedData;
. . .
allDone = 1;
```

Thread T_1

```
while ( allDone == 0)
  usleep(50000);
. . .
ret = write(..., OutputBuffer[currBlock],...);
```

Fig. 14. 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.

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 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 microbenchmarks by introducing mutex synchronizations. When we reran Portend$^+$ with the erroneous data race detector on the microbenchmarks, all four were falsely reported as races by the detector, but Portend$^+$ ultimately classified all of them as "single ordering." This suggests that Portend$^+$ is capable of properly handling false positives.

Figure 14 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.

Table IV. Portend$^+$'s Classification Time for the 93 Races in Table III

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

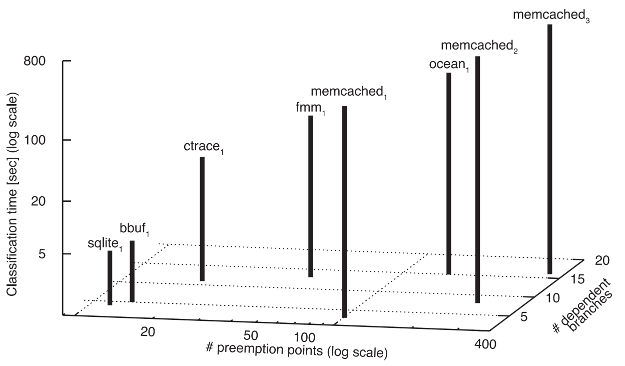


Fig. 15. Change in classification time with respect to number of preemptions and number of dependent branches for some of the races in Table III. Each sample point is labeled with a race ID.

6.3. 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 large-scale bug triage tool, such as in Microsoft's Windows Error Reporting system [Glerum et al. 2009].

We measure the time it takes Portend$^+$ to classify the 93 races; Table IV summarizes the results. We find that Portend$^+$ classifies all detected data races in a reasonable amount of time, with 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, as classification into this category requires multipath multischedule 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.

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. In Figure 15, we show how classification

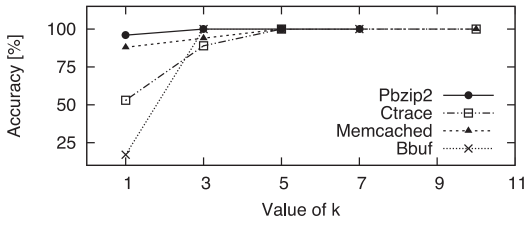


Fig. 16. Portend$^+$'s accuracy with increasing values of k.

Table V. Accuracy for Each Approach and Each Classification Category, Applied to the 93 Races in Table III

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

Note: "Not classified" means that an approach cannot perform classification for a particular class.

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.

We analyzed Portend$^+$'s accuracy with increasing values of k and found that $k = 5$ is sufficient to achieve overall 99% accuracy for all programs in our evaluation. Figure 16 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.

6.4. Comparison to State of the Art

We compare Portend$^+$ to the Record/Replay Analyzer technique [Narayanasamy et al. 2007], Helgrind$^+$'s technique [Jannesari and Tichy 2010], and Ad-Hoc-Detector [Tian et al. 2008] 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 [Sen 2008], because it primarily is 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 [Erickson and Olynyk 2010], since race classification in this tool is based on heuristics that pertain to races that we rarely encountered in our evaluation.

In Table V, 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 postrace 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.

The main advantage of Portend$^+$ over Record/Replay Analyzer is that it is immune to replay failures. In particular, for all 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 exhibiting no apparent harmfulness; 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, and therefore its classification results can apply to a range of inputs rather than a single input.

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 [Narayanasamy et al. 2007; Erickson and Olynyk 2010], 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 1 hour. Manual investigation revealed that this path is hard to find because it requires a very specific and complex combination of inputs.

6.5. Efficiency and Effectiveness of Symbolic Memory Consistency Modeling

The previous evaluation results were using the SMCM plugin in the sequential consistency mode. The sequential memory consistency mode is the default in Cloud9 as well as in Portend$^+$. In this section, we answer the following questions while operating the SMCM plugin in Portend$^+$'s weak consistency mode: Is Portend$^+$ effective in discovering bugs that may surface under its weak consistency model? What is Portend$^+$'s efficiency and memory usage while operating the SMCM plugin in Portend$^+$'s weak consistency mode?

We use simple microbenchmarks that we have constructed to test the basic functionality of SMCM. The simplified source code for these microbenchmarks can be seen in Figure 17. These benchmarks are as follows:

—*no-sync*: The source code for this microbenchmark can be seen in Figure 17(a): a program with opportunities for write reordering. Reorderings cause the `printf` statement on line 17 to produce different program outputs.

—*no-sync-bug*: The source code for this benchmark can be seen in Figure 17(b): a program with opportunities for write reordering. A particular write reordering causes the program to crash; however, the program does not crash under sequential consistency.

—*sync*: The source code for this microbenchmark can be seen in Figure 17(c): a program with no opportunities for write reordering. There is a race on both `globalx` and `globaly`. Since both threads 1 and 2 write the same value 2 to `globalx`, the output of the program is the same for any execution, assuming that writes are atomic.[5]

[5]If writes are nonatomic, even a seemingly benign race, where two threads write the same value to a shared variable, may end up producing unexpected results. Details can be found in Boehm [2011]

```
int volatile globalx = 0;
int volatile globaly = 0;

              Thread T1

void * work0 (void *arg) {
  globalx = 2;
  globaly = 1;
  return 0;
}
              Thread T2

void * work1 (void *arg) {
  globalx = 2;
  return 0;
}
             Thread Main

int main (int argc, char *argv[]){
  pthread_t t0, t1;
  int rc;
  rc = pthread_create (&t0, 0, work0, 0);
  rc = pthread_create (&t1, 0, work1, 0);
  printf("%d,%d", globalx, globaly);
  pthread_join(t0, 0);
  pthread_join(t1, 0);
  return 0;
}
```

(a)

```
int volatile globalx = 0;
int volatile globaly = 0;

              Thread T1

void * work0 (void *arg) {
  globalx = 2;
  globaly = 1;
  return 0;
}
              Thread T2

void * work1 (void *arg) {
  globalx = 2;
  return 0;
}
             Thread Main

int main (int argc, char *argv[]){
  pthread_t t0, t1;
  int rc;
  rc = pthread_create (&t0, 0, work0, 0);
  rc = pthread_create (&t1, 0, work1, 0);
  if(globalx == 0 && globaly == 2)
    ; //crash!
  pthread_join(t0, 0);
  pthread_join(t1, 0);
  return 0;
}
```

(b)

```
int volatile globalx = 0;
int volatile globaly = 0;

              Thread T1

void * work0 (void *arg) {
  globalx = 2;
  pthread_barrier_wait(&barr);
  globaly = 1;
  return 0;
}
              Thread T2

void * work1 (void *arg) {
   globalx = 2;
   pthread_barrier_wait(&barr);
   return 0;
}
             Thread Main

int main (int argc, char *argv[]){
  pthread_t t0, t1;
  int rc;
  pthread_barrier_init(&barr, NULL, 2);
  rc = pthread_create (&t0, 0, work0, 0);
  rc = pthread_create (&t1, 0, work1, 0);
  printf("%d,%d", globalx, globaly);
  pthread_join(t0, 0);
  pthread_join(t1, 0);
  return 0;
}
```

(c)

```
int volatile globalx = 0;
int volatile globaly = 0;

              Thread T1

void * work0 (void *arg) {
  globalx = 2;
  globaly = 1;
  pthread_barrier_wait(&barr);
  return 0;
}
              Thread T2

void * work1 (void *arg) {
   globalx = 2;
   pthread_barrier_wait(&barr);
   return 0;
}
             Thread Main

int main (int argc, char *argv[]){
  pthread_t t0, t1;
  int rc;
  pthread_barrier_init(&barr, NULL, 2);
  rc = pthread_create (&t0, 0, work0, 0);
  rc = pthread_create (&t1, 0, work1, 0);
  if(globalx == 0 && globaly == 2)
    ; //crash!
  pthread_join(t0, 0);
  pthread_join(t1, 0);
  return 0;
}
```

(d)

Fig. 17. Microbenchmarks used for evaluating SMCM.

Table VI. Portend$^+$'s Effectiveness in Bug Finding and State Coverage for Two Memory Model Configurations: Sequential Memory Consistency Mode and Portend$^+$'s Weak Memory Consistency Mode

	Number of Bugs		State Coverage (%)	
System	Portend$^+$-seq	Portend$^+$-weak	Portend$^+$-seq	Portend$^+$-weak
no-sync	0/0	0/0	50	100
no-sync-bug	0/1	1/0	50	100
sync	0/0	0/0	100	100
sync-bug	0/1	1/1	50	100

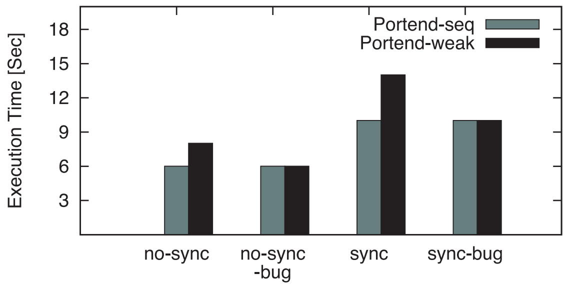


Fig. 18. Running time of Portend$^+$-weak and Portend$^+$-seq.

—*sync-bug*: The source code for this microbenchmark can be seen in Figure 17(d): a program with opportunities for write reordering. The barrier synchronization does not prevent the write to `globalx` and `globaly` from reordering. A particular write reordering causes the program to crash; however, the program does not crash under sequential consistency.

To evaluate Portend$^+$'s effectiveness in in finding bugs that may only arise under Portend$^+$'s weak ordering, we ran the microbenchmarks with Portend$^+$'s SMCM plugin configured in two modes: sequential consistency (Portend$^+$-seq) and Portend$^+$'s weak consistency (Portend$^+$-weak). We provide the number of bugs found by each configuration of Portend$^+$ and also the percentage of possible execution states that each configuration covers if Portend$^+$'s weak consistency were assumed. Note that ground truth (i.e., the total number of states that can be covered under Portend$^+$'s weak consistency) in this case is manually identified, as the number of possible states is small. Effectively identifying this percentage for arbitrarily large programs is undecidable.

We present the results in Table VI. As can be seen, Portend$^+$-weak discovers the bugs that can only be discovered under Portend$^+$'s weak consistency, whereas Portend$^+$-seq cannot find those bugs because of sequential consistency assumptions. Similar reasoning applies to state exploration. Portend$^+$ covers all possible states that may arise from returning multiple values at reads, whereas Cloud9 simply returns the last value that was written to a memory location and hence has lower coverage.

We also evaluate the performance of Portend$^+$-weak for our microbenchmarks and compare its running time to that of Portend$^+$-seq. The results of this comparison can be seen in Figure 18. The running times of the benchmarks under Portend$^+$-seq essentially represent the "native" LLVM interpretation time. For the *no-sync* benchmark, we can see that the running time of Portend$^+$ is about 2 seconds more than that of Portend$^+$-seq. This is expected, as Portend$^+$-weak covers more states compared to Portend$^+$-seq.

However, it should be noted that in the case of the *no-sync-bug* benchmark, the running times are almost the same for Portend$^+$-weak and Portend$^+$-seq (although not visible on the graph, the running time of Portend$^+$-weak is slightly larger than that

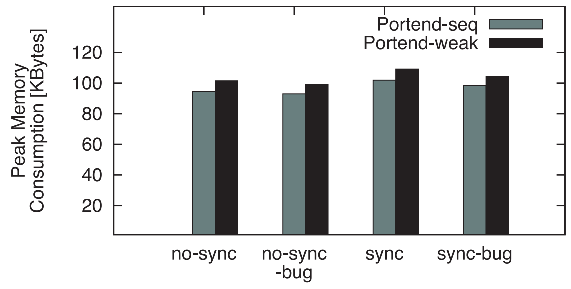


Fig. 19. Memory usage of Portend$^+$-weak and Portend$^+$-seq.

of Portend$^+$-seq, on the order of a few milliseconds). This is simply because the bug in *no-sync-bug* is immediately discovered after the exploration state has been forked in Portend$^+$. The bug is printed at the program output, and the exploration ends for that particular state. Similar reasoning applies to the other benchmark pair, namely *sync* and *sync-bug*.

6.6. Memory Consumption of Symbolic Memory Consistency Modeling

In this final section of the evaluation, we measure the peak memory consumption of Portend$^+$-weak and Portend$^+$-seq for the microbenchmarks we have tested. The results can be seen in Figure 19. The memory consumption increases for all benchmarks. This is because for all of the benchmarks, Portend$^+$-weak always forks off more states and/or performs more bookkeeping than Portend$^+$-seq, even though it does not always explore those states.

Although the memory consumption consistently increases for Portend$^+$-weak, it does not increase proportionally with the state forking. This is possible due to the copy-on-write mechanism employed for exploring states and keeping track of the happens-before graph. However, when we ran Portend$^+$-weak on real-world programs, the memory consumption quickly exceeded the memory capacity of the workstation that we used for our experiments. We plan on incorporating techniques such as partial order reduction from model checking to reduce the number of states that SMCM needs to explore and improve its scalability as part of future work.

In summary, Portend$^+$ is able to classify with 99% accuracy and full precision all 93 races into four data race classes defined in Section 3.3 in less than 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 (92 out of 93, 99%) and is able to correctly classify up to 89% more data races than existing replay-based tools (9 out of 93, 10%). Portend$^+$ also correctly handles false-positive race reports. Furthermore, SMCM allows Portend$^+$ to accurately perform race classification under relaxed memory consistency models with low overhead.

7. 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 Erickson and Olynyk [2010]). 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.

Multiprocessors and write atomicity. Portend$^+$ works for multiprocessors; however, it can simulate only atomic write operations. To model the relaxation of the write atomicity requirement, it is possible to break nonatomic instructions into multiple LLVM instructions. Then, Portend$^+$ would be able to generate currently missing schedules that stem from nonatomic operations.

Language memory models. Several language specifications, such as Ada 83 [Ledgard 1983] and the new C [ISO9899 2011] and C++ [ISO14882 2011] 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 [Boehm 2011]. 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 [Boehm 2011].

Single-processor scheduler. Most data race detectors use a single-processor thread scheduler, and Portend$^+$ does as well. 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, as 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 [Lu et al. 2006] or thread interleavings that would occur on a multiprocessor (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 multischedule analysis is more likely to uncover more interleavings than “single-pre/single-post” analysis, thus increasing the thread schedule coverage. A single-processor scheduler in Portend$^+$ could potentially impact data race classification under relaxed memory consistency models because simultaneous scheduling of threads in multiprocessors may not be taken into account. This could in turn cause simultaneous accesses to shared memory locations to be ignored. However, Portend$^+$ overcomes this limitation with the use of SMCM.

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 [Candea et al. 2010]. 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 [Engler and Ashcraft 2003] by feeding them to an execution synthesis tool such as ESD [Zamfir and Candea 2010], 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.

8. RELATED WORK

Data race detection techniques can be broadly classified into two major categories: static data race detection [Voung et al. 2007; Engler and Ashcraft 2003] and dynamic data race detection [Savage et al. 1997; Schonberg 2004; Yu et al. 2005]. 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 [Lamport 1978; Bond et al. 2010; Schonberg 2004; Mellor-Crummey 1991; Prvulovic and Torrellas 2003; Min and Choi 1991; Nistor et al. 2009], (2) detection based on the lockset algorithm [Savage et al. 1997; Yu et al. 2005], and (3) hybrid algorithms that combine these two techniques [Serebryany and Iskhodzhanov 2009; O'Callahan and Choi 2003; Yu et al. 2005]. Portend$^+$ uses a happens-before algorithm.

Portend$^+$'s race detector can benefit from various optimizations, such as hardware support [Prvulovic and Torrellas 2003; Min and Choi 1991] that speeds up race detection, or sampling methods [Marino et al. 2009; Bond et al. 2010] that reduce detection overhead.

Prior work on data race classification employs record/replay analysis [Narayanasamy et al. 2007], heuristics [Erickson and Olynyk 2010], detection of ad hoc synchronization patterns [Jannesari and Tichy 2010; Tian et al. 2008], or simulation of the memory model [Flanagan and Freund 2010].

Record/replay analysis [Narayanasamy et al. 2007] 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 an 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 [Erickson and Olynyk 2010] 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$^+$ [Jannesari and Tichy 2010] and Ad-Hoc-Detector [Tian et al. 2008] 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 [Zhang et al. 2011].

Adversarial memory [Flanagan and Freund 2010] finds races that occur in systems with memory consistency models that are more relaxed than sequential consistency, such as the Java memory model [Manson et al. 2005]. This approach uses a model of memory that returns stale yet valid values for memory reads (using various heuristics) in an attempt to crash target programs. If a data race causes the program to crash as a result of using the adversarial memory approach, that data race will be classified as harmful. Portend$^+$ follows a similar approach to adversarial memory in that it uses a special model of the memory—SMCM—which buffers all prior writes to memory. However, unlike prior work, SMCM allows Portend$^+$ to systematically explore all possible memory model behaviors. Furthermore, SMCM is a technique to augment any program analysis tool with the capability of systematically reasoning about various memory consistency models. Finally, Portend$^+$ provides finer-grained classification than the adversarial memory approach.

Prior work employed bounded model checking to formally verify concurrent algorithms for simple data types under weak memory models [Burckhardt et al. 2006]. Formal verification of programs under relaxed memory models (using bounded model checking or any other technique) is a difficult problem and is undecidable in the general case [Atig et al. 2010]. Portend$^+$ does not intend to formally verify a program under weak memory consistency. Instead, Portend$^+$ focuses on determining the combined effects of data races and weak memory models.

RaceFuzzer [Sen 2008] generates random schedules from a pair of racing accesses to determine whether the race is harmful. Therefore, RaceFuzzer performs multischedule analysis, but not multipath, and does so only with the goal of finding bugs, not classifying races. Portend$^+$ uses multipath analysis in addition to multischedule analysis to improve triage accuracy.

Output comparison was used by Pike to find concurrency bugs by fuzzing thread schedules [Fonseca et al. 2011]. 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 [Veeraraghavan et al. 2011] 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 Section 6.4. If used in conjunction with Portend$^+$, Frost could avoid provably harmful races.

We envision that Portend$^+$ can be used in conjunction with race avoidance tools such as LOOM [Wu et al. 2010]. LOOM requires developers to write execution filters that allow programs to filter out racy interleavings. Thread schedules that correspond to racy interleavings can be classified by Portend$^+$, and consequently, developers can prioritize writing execution filters.

Deterministic execution systems have recently gained popularity in academia [Liu et al. 2011; Cui et al. 2010; Bergan et al. 2010; Devietti et al. 2009; Aviram et al. 2010; Thies et al. 2002; Bocchino et al. 2009]. Deterministic execution requires making the program merely a function of its inputs [Bergan et al. 2011].These systems allow executing arbitrary concurrent software deterministically using hardware, runtime, or operating system support. Alternatively, they allow developers to build deterministic software from scratch using languages that provide determinism guarantees.

Deterministic execution systems do not inherently simplify parallel programming [Hand 2012], and they may incur high overheads. Therefore, they are not yet adopted in production software. DMP proposes hardware extensions that allows arbitrary software to be executed deterministically. CoreDet [Bergan et al. 2010] extends DMP's deterministic execution algorithms and is a software-only system that provides a compiler and a runtime to deterministically execute multithreaded software.

Determinator [Aviram et al. 2010] is a novel operating system kernel that aims to deterministically execute arbitrary programs. Determinator allocates each thread of execution a private working space that is a copy of the global state. Threads reconcile their view of the global state at well-defined points in a program's execution.

The use of private workspaces eliminates all read/write conflicts in Determinator, and write/write conflicts are transformed into runtime exceptions. Determinator allows running a number coarse-grain parallel benchmarks with comparable performance to a nondeterministic kernel. Current operating system kernels are not built with Determinator's determinism guarantees, and it is unclear if they will be in the future.

StreamIt [Thies et al. 2002] is a stream-based programming language that allow threads to communicate only via explicitly defined streams and therefore provides determinism for stream-based programs. DPJ [Bocchino et al. 2009] is a type and effect system for Java that is deterministic by default and only allows explicit nondeterminism. These systems allow the developer to build deterministic systems by construction; however, they are not yet widely adopted.

To achieve deterministic execution in the general case, races must be eliminated from programs in some way, which leads to high overhead. We believe that this overhead has been an important obstacle for the widespread adoption of deterministic execution systems in production software. 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.

9. CONCLUSION

This article presents the first technique for triaging data races based on their potential consequences through an analysis that is multipath, multischedule, and sensitive to memory consistency models. 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 seven real-world applications with 99% accuracy and full precision, with no human effort.

ACKNOWLEDGMENTS

We thank Emery Berger for providing feedback that helped us greatly improve the quality of this article. We thank the anonymous reviewers for providing insightful feedback. We thank TOPLAS editor-in-chief Jens Palsberg and TOPLAS traffic manager Aubrey Hooser for their help and guidance throughout the revision process. We thank Babak Falsafi for providing feedback on the initial design of SMCM. We thank Katerina Argyraki, Ryan Johnson, Olivier Crameri, Christopher Ming-Yee Iu, Sotiria Fytraki, and our DSLAB colleagues for helping us improve this article. We thank the European Research Council and VMware for supporting Baris Kasikci. We thank Microsoft for supporting Cristian Zamfir through an ICES grant.

REFERENCES

Sarita V. Adve and Mark D. Hill. 1990. Weak ordering-a new definition. *Computer Architecture News* 18, 2, 2–14.

Associated Press. 2004. GE Acknowledges Blackout Bug. Retrieved April 2, 2015, from http://www.securityfocus.com/news/8032.

Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2010. On the verification problem for weak memory models. In *Proceeedings of the Symposium on Principles of Programming Languages*.

Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. 2010. Efficient system-enforced deterministic parallelism. In *Proceedings of the Symposium on Operating Systems Design and Implementation*.

Domagoj Babic and Alan J. Hu. 2008. Calysto: Scalable and precise extended static checking. In *Proceedings of the 30th International Conference on Software Engineering*.

Tom Bergan, Joseph Devietti and Luis Ceze. 2011. The deterministic execution hammer: How well does it actually pound nails? In *Proceedings of the Workshop on Determinism and Correctness in Parallel Programming*.

Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. 2010. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

Robert L. Bocchino Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'09)*.

Hans-J. Boehm. 2007. Reordering constraints for pthread-style locks. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*.

Hans-J. Boehm. 2011. How to miscompile programs with "benign" data races. In *Proceedings of the USENIX Workshop on Hot Topics in Parallelism*.

Hans-J. Boehm. 2012. Position paper: Nondeterminism is unavoidable, but data races are pure evil. In *Proceedings of the ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES'12)*.

Hans-J. Boehm and Sarita V. Adve. 2012. You don't know jack about shared variables or memory models. *Communications of the ACM* 55, 2, 48–54.

Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. 2010. PACER: Proportional detection of data races. In *Proceedings of the International Conference on Programming Language Design and Implementation.*

Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the ACM EuroSys European Conference on Computer Systems*.

Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. 2006. Bounded model checking of concurrent data types on relaxed memory models: A case study. In *Proceedings of the International Conference on Computer Aided Verification*.

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Symposium on Operating Systems Design and Implementation.*

George Candea, Stefan Bucur, Vitaly Chipounov, Vova Kuznetsov, and Cristian Zamfir. 2010. Automated software reliability services: Using reliability tools should be as easy as Webmail. In *Proceedings of the Symposium on Operating Systems Design and Implementation*.

Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. 2007. BulkSC: Bulk enforcement of sequential consistency. In *Proceedings of the International Symposium on Computer Architecture*.

Vitaly Chipounov and George Candea. 2011. Enabling sophisticated analyses of x86 binaries with RevGen. In *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems and Networks*.

Heming Cui, Jingyue Wu, Chia Che Tsai, and Junfeng Yang. 2010. Stable deterministic multithreading through schedule memoization. In *Proceedings of the Symposium on Operating Systems Design and Implementation.*

Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. 2009. DMP: Deterministic shared memory multiprocessing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

Michel Dubois, Christoph Scheurich, and Faye Briggs. 1986. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture.*

Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Symposium on Operating Systems Principles*.

John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective data-race detection for the kernel. In *Proceedings of the Symposium on Operating System Design and Implementation (OSDI'10).*

Brad Fitzpatrick. 2013. Memcached Home Page. Retrieved April 2, 2015, from http://memcached.org.

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the International Conference on Programming Language Design and Implementation.*

Cormac Flanagan and Stephen N. Freund. 2010. Adversarial memory for detecting destructive races. In *Proceedings of the International Conference on Programming Language Design and Implementation.*

Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues. 2011. Finding complex concurrency bugs in large multithreaded applications. In *Proceedings of the ACM EuroSys European Conference on Computer Systems*.

Vijay Ganesh and David L. Dill. 2007. A decision procedure for bit-vectors and arrays. In *Proceedings of the International Conference on Computer Aided Verification*.

Jeff Gilchrist. 2013. Parallel BZIP2 (PBZIP2). Retrieved April 2, 2015, from http://compression.ca/pbzip2.

Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. 2009. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of the Symposium on Operating Systems Principles*.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the International Conference on Programming Language Design and Implementation.*

Patrice Godefroid, Michael Y. Levin, and David Molnar. 2008. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium.*

Patrice Godefroid and Nachiappan Nagappan. 2008. Concurrency at Microsoft—an exploratory survey. In *Proceedings of the International Conference on Computer Aided Verification*.

Steven Hand. 2012. An experiment in determinism. *Communications of the ACM* 55, 5, 110.

Helgrind. 2012. Helgrind Home Page. Retrieved April 2, 2015, from http://valgrind.org/docs/manual/hg-manual.html.

Intel Corp. 2012. Parallel Inspector. Retrieved April 2, 2015, from https://software.intel.com/en-us/intel-inspector-xe.

ISO14882. 2011. *ISO/IEC 14882:2011: Information Technology—Programming languages—C++*. International Organization for Standardization, London, UK.

ISO9899. 2011. *ISO/IEC 9899:2011: Information Technology—Programming Languages—C*. International Organization for Standardization, London, UK.

Ali Jannesari and Walter F. Tichy. 2010. Identifying ad-hoc synchronization for enhanced race detection. In *Proceedings of the International Parallel and Distributed Processing Symposium.*

Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. 2012. Automated concurrency-bug fixing. In *Proceedings of the Symposium on Operating Systems Design and Implementation.*

Vineet Kahlon, Franjo Ivančić, and Aarti Gupta. 2005. Reasoning about threads communicating via locks. In *Proceedings of the International Conference on Computer Aided Verification*.

Baris Kasikci, Cristian Zamfir, and George Candea. 2012. Data races vs. data race bugs: Telling the difference with Portend. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21, 7, 558–565.

Chris Lattner. 2012. “libc++” C++ Standard Library. Retrieved April 2, 2015, from http://libcxx.llvm.org/.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*.

Henry Ledgard. 1983. *Reference Manual for the ADA Programming Language*. Springer-Verlag, New York, NY.

Nancy G. Leveson and Clark S. Turner. 1993. An investigation of the Therac-25 accidents. *IEEE Computer* 26, 7, 18–41.

Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. Dthreads: Efficient deterministic multithreading. In *Proceedings of the Symposium on Operating Systems Principles*.

Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2006. AVIO: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In *Proceedings of the Symposium on Principles of Programming Languages*.

Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. 2009. LiteRace: Effective sampling for lightweight data-race detection. In *Proceedings of the International Conference on Programming Language Design and Implementation.*

Cal McPherson. 2012. Ctrace Home Page. Retrieved April 2, 2015, from http://ctrace.sourceforge.net.

John Mellor-Crummey. 1991. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the ACM/IEEE Conference on Supercomputing (Supercomputing’91)*.

Memcached. 2009. Issue 127: INCR/DECR Operations Are Not Thread Safe. Retrieved April 2, 2015, from http://code.google.com/p/memcached/issues/detail?id=127.

Sang L. Min and Jong-Deok Choi. 1991. An efficient cache-based access anomaly detection scheme. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

Madanlal Musuvathi, Sebastian Burckhardt, Pravesh Kothari, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the Symposium on Operating Systems Design and Implementation.*

Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. 2007. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the International Conference on Programming Language Design and Implementation.*

Adrian Nistor, Darko Marinov, and Josep Torrellas. 2009. Light64: Lightweight hardware support for data race detection during systematic testing of parallel programs. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*.

Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid dynamic data race detection. In *Proceedings of the Symposium on Principles and Practice of Parallel Computing*.

Milos Prvulovic and Josep Torrellas. 2003. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems* 15, 4, 391–411.

Edith Schonberg. 2004. On-the-fly detection of access anomalies (with retrospective). *ACM SIGPLAN Notices* 39, 4, 313–327.

Koushik Sen. 2008. Race directed random testing of concurrent programs. In *Proceedings of the International Conference on Programming Language Design and Implementation.*

Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. In *Proceedings of the Symposium on the Foundations of Software Engineering.*

Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer—data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*.

Richard L. Sites (Ed.). 1992. *Alpha Architecture Reference Manual*. Digital Press.

Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound predictive race detection in polynomial time. *ACM SIGPLAN Notices* 47, 1, 387–400.

SQLite. 2013. SQLite Home Page. Retrieved April 2, 2015, from http://www.sqlite.org/.

William Thies, Michal Karczmarek, and Saman P. Amarasinghe. 2002. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC'02)*.

Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. 2008. Dynamic recognition of synchronization operations for improved data race detection. In *Proceedings of the International Symposium on Software Testing and Analysis*.

Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. Detecting and surviving data races using complementary schedules. In *Proceedings of the Symposium on Operating Systems Principles*.

Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: Static race detection on millions of lines of code. In *Proceedings of the Symposium on the Foundations of Software Engineering.*

David L. Weaver and Tom Germond (Eds.). 1994. *The SPARC Architecture Manual, Version 9*. Prentice Hall.

Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture.*

Jingyue Wu, Heming Cui, and Junfeng Yang. 2010. Bypassing races in live applications with execution filters. In *Proceedings of the Symposium on Operating Systems Design and Implementation.*

Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. 2010. Ad-hoc synchronization considered harmful. In *Proceedings of the Symposium on Operating Systems Design and Implementation.*

Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. 2007. Distributed dynamic partial order reduction based verification of threaded software. In *Proceedings of the International SPIN Workshop*.

Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the Symposium on Operating Systems Principles*.

Cristian Zamfir and George Candea. 2010. Execution synthesis: A technique for automated debugging. In *Proceedings of the ACM EuroSys European Conference on Computer Systems*.

Jiaqi Zhang, Weiwei Xiong, Yang Liu, Soyeon Park, Yuanyuan Zhou, and Zhiqiang Ma. 2011. ATDetector: Improving the accuracy of a commercial data race detector by identifying address transfer. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*.

Received February 2013; revised December 2014; accepted February 2015