RACEZ: A Lightweight and Non-Invasive Race Detection Tool for Production Applications

Tianwei Sheng[1] Neil Vachharajani [2] Stephane Eranian[2] Robert Hundt[2] Wenguang Chen[1]
Weimin Zheng[1]
[1]Tsinghua University,Beijing, China, 100084
[2]Google, Inc. 1600 Amphitheatre Parkway, Mountain View, CA 94043
tianwei.sheng@gmail.com, neil@purestorage.com, {eranian, rhundt}@google.com, {cwg,zwm-dcs}@tsinghua.edu.cn

ABSTRACT

Concurrency bugs, particularly data races, are notoriously difficult to debug and are a significant source of unreliability in multithreaded applications. Many tools to catch data races rely on program instrumentation to obtain memory instruction traces. Unfortunately, this instrumentation introduces significant runtime overhead, is extremely invasive, or has a limited domain of applicability making these tools unsuitable for many production systems. Consequently, these tools are typically used during application testing where many data races go undetected.

This paper proposes RACEZ, a novel race detection mechanism which uses a sampled memory trace collected by the hardware performance monitoring unit rather than invasive instrumentation. The approach introduces only a modest overhead making it usable in production environments. We validate RACEZ using two open source server applications and the PARSEC benchmarks. Our experiments show that RACEZ catches a set of known bugs with reasonable probability while introducing only 2.8% runtime slow down on average.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Design, Performance

Keywords

Data races, performance monitoring unit, sampling, probability analysis

1. INTRODUCTION

1.1 Motivation

The dominance of multi-core processors has made concurrent programming essential to achieve peak performance from modern systems. Unfortunately, parallel programming is considerably more difficult than its sequential counterpart. In addition to all the bugs common to sequential code, one must also contend with concurrency bugs [12] such as data races, atomicity violations, deadlock, and livelock. Among concurrency bugs, data races are often serious and extremely difficult to debug. For example, a race condition was partially responsible for the black out in the northeastern part of the United States, described as the "worst outage in North American history" [23].

A data race occurs when multiple threads concurrently access the same location without proper synchronization, and at least one of the accesses is a write. Data races are difficult to diagnose for two primary reasons. First they often manifest only under certain, potentially very rare, thread interleavings. This makes data race bugs difficult to reproduce. Second, the actual data race typically only corrupts data. User visible effects, such as program crashes or corrupted output, may occur much later which makes it difficult to isolate where in the code the race actually occurred.

Because data races are serious bugs that are notoriously difficult to find, the literature is replete with approaches to automatically detect data races. These approaches can broadly be classified into two categories: static race detection and dynamic race detection.

While static analyses have made great strides, currently, mainstream practical race detection tools use the dynamic approach [22, 17, 19, 10]. These tools monitor a program at runtime and check to see if any data race has (or could have) occurred. To enable this checking, synchronization operations and memory accesses in the program are instrumented, and the race detection tool analyzes the streams of synchronizations and accesses. Although these dynamic tools can detect many data races, they typically suffer from significant instrumentation overhead. For example, the Intel Thread Checker reported an average 100–200x slow down [21], and a new valgrind based tool ThreadSanitizer, which is widely used at Google [25], has a 30–40x slow down. Even recent proposals such as FastTrack [10] still incur an average 8.5x slow down.

The significant overhead of these tools makes them unusable in production environments. Instead, developers test for concurrency bugs during their pre-deployment testing, or to triage bugs after they have been observed in the field. In addition to using tools explicitly designed to find the root cause of data races, developers also use program testing methods designed to increase the probability that bugs are triggered [13, 18, 15, 6]. However, some concurrency bugs still escape testing and remain in the production code. Pre-deployment testing does not catch all data races for three primary reasons:

1. Lack of representative testing input. In order to expose all bugs during testing, developers must construct representative testing input to cover all the code in the program. This has been proven to be very challenging.
2. Incomplete modeling of the runtime environment. The testing environment is often very different from the real pro-

ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

duction environment (e.g., operating system kernel version, thread library version). This poses a serious challenge for concurrency bugs since differences in environment can affect the likelihood of various thread interleavings.

3. Insufficient testing time. Data races often manifest infrequently. Even with approaches to boost the probability of occurrence, data races will often not manifest during testing and will occur only infrequently in production leading to mysterious program crashes or data corruption.

Sampling has emerged as a promising technique to overcome these challenges. Sampling can dramatically lower overhead, and therefore allows the race detection tool to be run in production. Recent work [14, 5] has shown that monitoring only a small fraction of the memory accesses from a program is sufficient for capturing many race conditions. This approach can lower overhead as low as 28%. Unfortunately, even this overhead is still too high for many production systems.

Additionally, the approaches are invasive or have limited applicability. For example, to sample memory accesses LiteRace clones every function, one of which is instrumented, leading to a 2x increase in code size. At the entrance to each function, checks are inserted to decide whether to execute the instrumented or uninstrumented versions. This code growth can have significant performance ramifications due to instruction cache, instruction TLB, and branch prediction affects. Another recent sampling approach, Pacer, avoids the code growth by relying on a JIT compiler to insert instrumentation. Unfortunately, while JIT compilation works well for managed languages such as Java, it is inapplicable to unmanaged (e.g., C/C++) code.

Further, users may be hesitant to deploy LiteRace or Pacer protected code due to the invasive instrumentation that could create pathological performance issues observed only in production, lead to unforeseen program crashes, or obscure the root cause of other bugs observed in the field.

1.2 Our Contribution

In this paper, we propose a dynamic data race detection method that samples memory accesses using the hardware performance monitoring unit (PMU) rather than relying on instrumentation of memory instructions. Our approach is based on the observation that memory accesses occur much more frequently than synchronization operations. Consequently, instrumenting memory accesses is far more invasive and results in much higher overhead than instrumenting only synchronization primitives. Fortunately, modern hardware PMUs can sample memory accesses with very low overhead.

Unfortunately, using the PMU for race detection is not a panacea. The PMU was originally designed for performance monitoring. Consequently, the following artifacts in its behavior must be overcome to enable race detection:

1. **Precise PMU signal delivery** Traditional race detection tools instrument both the synchronization and memory operations in user-level code. As shown in Figure 1, when using the PMU to sample memory accesses, the synchronization data comes from user-level instrumentation, while the memory access data comes from hardware, through the kernel, to user-level code via a signal. In the figure, if the memory operation W1 is sampled by the PMU, we would like the signal to be delivered to thread 1 before the unlock (edge 1 in the figure). This guarantees the tool knows the correct lock context in which the memory was accessed. However, the signal may be delivered at the points indicated by edges 2 and 3 due to various OS and PMU issues.

2. **Distributed and biased samples from the PMU** All sampling approaches will only collect data for part of a program's execution. As shown in Figure 2(a), previous sampling approaches to race detection such as LiteRace[14] and Pacer[5] collect data from a contiguous sequence of instructions on each sample. However, as shown in Figure 2(b), the PMU collects data for only one instruction in each sample. This means that races must be detected by synthesizing information from temporally distant instructions in the program. Additionally, due to bias in PMU sampling, certain instructions (even memory instructions) may *never* be sampled [7].

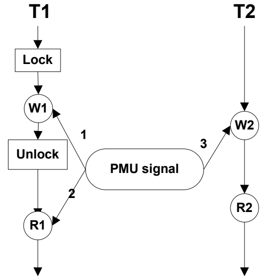


Figure 1: Challenges in precise signal delivery from the PMU. Edge *1* is the desired signal delivery, however, in practice edges *2* and *3* are possible.

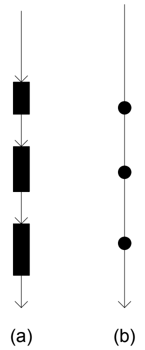


Figure 2: Different sampling mechanisms.

To address the first challenge, we propose both a combination of kernel enhancements and hardware and software solutions to guarantee that the sample of memory operations can be correlated back to the trace of synchronization operations. We address the second challenge using several strategies. First, by computing a program slice originating at each sampled instruction, we extend each instruction sample into effectively many. Not only does this allow us to reduce our sampling rate, it also provides coverage to instructions that may not be sampled due to PMU artifacts. Second, we rely on the lockset race detection algorithm which does not depend on bursts of memory operation data for race detection. Finally, we observe that if the tool's overhead is sufficiently low, it can be run in production. Consequently, in the context of data center applications, we need not catch race conditions with very high probability. Since each application is running on many machines, over very long periods of time, even a small probability of catching the race on one particular invocation of the application is acceptable since it is sufficient to catch the race on any one of the many machines running the application.

We refer to our implementation of these techniques as RACEZ. We evaluate RACEZ with two open source server applications and one standalone parallel benchmarks suite. For the two server applications, given a set of known data races bugs, using a modest

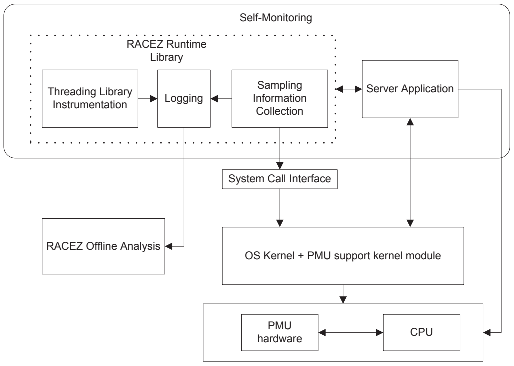


Figure 3: Overview of RACEZ architecture

sampling rate, RACEZ can successfully catch all of them with reasonable probability. At the same sampling rate, RACEZ only incurs a 2.8% slow down on average.

The main contributions of this paper include:

1. RACEZ, a lockset based race detection tool which uses the PMU, rather than instrumentation, to sample program memory accesses. Besides very low overhead with sampling, the most important merit of RACEZ is that it is non-invasive and requires minimum modification to the original applications. To the best of our knowledge, this is the first tool that uses existing PMU hardware to do race detection.
2. We develop several techniques to overcome the fundamental problems when using the PMU race detection.
3. We provide a theoretical analysis and detailed experimental evaluation to prove the effectiveness and efficiency of RACEZ

The rest of part of this paper is organized as follows. Section 2 gives the system overview of RACEZ. Section 3 describes the design and implementation of RACEZ. We give a simple mathematical analysis for the probability of catching data races and introduce our main optimization techniques in Section 4. Our experimental results are reported in Section 5. We discuss our limitations in Section 6 and briefly describe the related work in Section 7. Finally, we conclude in Section 8.

2. SYSTEM OVERVIEW

All data race detection tools require traces containing synchronization operations and memory references. Traditional race detection methods instrument both and apply race detection algorithms for online or postmortem analysis. As illustrated in Figure 4b, traditional methods instrument the lock and unlock operations, as well as the two memory operations(read,write). RACEZ also instruments the lock/unlock operations. However, it uses a different approach to obtain the trace of memory addresses. Instead of instrumenting each memory access, addresses are obtained from PMU samples. As shown in Figure 4c, RACEZ only instruments the two lock related functions and reads the memory addresses from the PMU.

Before providing an in-depth discussion in the following sections, we briefly introduce how RACEZ combines these two pieces of information to detect data races. Our architecture is shown in Figure 3. RACEZ uses self-monitoring to collect PMU samples from the server application; i.e., RACEZ runs as a component of the target application and resides in the same address space. With monitoring enabled, the thread library redirects synchronization calls to wrapper functions which update lockset information for each thread. A separate signal handler retrieves memory address information whenever an instruction of that thread is sampled by the PMU. The hardware PMU is accessed through a kernel system call interface. For each sample, the PMU generates processor interrupts which the kernel eventually transforms into an asynchronous signal delivered to the user application.

Depending on the processor family, there are different ways of sampling memory accesses with the PMU. In this paper, for simplicity, we only discuss the Intel implementation. We briefly discuss other PMU implementations in Section 3.5. Using precise event based sampling (PEBS), the Intel PMU can record the register state of the processor with each sample. However, it is not possible to sample only memory accesses. Therefore we use an offline phase to identify relevant samples and detect races using a standard lockset algorithm[22].

For the code in Figure 4c, when the *Write* operation is sampled by the PMU, the signal handler in T1 records the current register values together with the currently held lockset *{L}* to the log file. If a *Read* operation of the same shared variable with a disjoint lockset is sampled by the PMU in another thread, RACEZ's offline analysis tool will report a warning for these two memory references.

3. MONITORING AND ANALYSIS

RACEZ has two major parts: runtime monitoring and offline analysis. This section first discusses the runtime component and concludes by describing the offline analysis..

3.1 Definitions

We first provide some definitions for the information we want to collect.

DEFINITION 1 (EVENT AND EVENT TYPE). *An event is an operation that should be recorded for race detection. Currently we support three types of events: Lock/Unlock operations, memory references, and memory allocation/deallocation events.*

DEFINITION 2 (LOCKSET). *A lockset denotes the set of locks that a thread holds at a particular point during the program's execution.*

With these two definitions, the simplest lockset based algorithm only needs to track memory reference and lock/unlock events. However, this algorithm could report false positives due to two logical variables sharing the same virtual address. In RACEZ, we improve this simple algorithm by additionally tracking memory allocation events to help filter out these false positives. Figure 5 shows the format of the information logged by RACEZ.

3.2 Thread Library Instrumentation

To track lock/unlock events, RACEZ relies on instrumented synchronization functions in the threading library. Each thread maintains its lockset in thread local storage and updates these sets for each lock/unlock event.

To enable PMU self-monitoring, each thread must request the operating system to enable the PMU when the thread is running. RACEZ relies on instrumentation in the threading library's thread creation routines to perform the PMU setup. If the threading library supports a mechanism to traverse all live threads while a program is running, RACEZ can also be enabled on an already running application by traversing all threads and having each do the necessary PMU setup.

3.3 PMU Context Manipulation

The PMU is managed by the kernel and is accessible to user application via a system call interface. A PMU session, or context, is identified by a file descriptor, and it is used throughout the session to configure and read data from the PMU. A typical sampling PMU session consists of five phases:

```
1  T1:
2    Lock(L)
3    ..........
4    Write X
5    ..........
6    Unlock(L)
7    .........
8  T2:
9    ..........
10   Read X
11   ..........
```

(a) A program with a race

```
T1:
 //Instrument lock operation
 Lock(L)
 ..........
 //Instrument Memory operation
 Write X
 ..........
 Unlock(L)
 //Instrument unlock operation
T2:
 ..........
 //Instrument Memory operation
 Read X
 ..........
```

(b) Traditional method

```
T1:
 //Instrument lock operation
 Lock(L)
 ..........
 //Read the address from PMU
 Write X
 ..........
 Unlock(L)
 //Instrument unlock operation
T2:
 ..........
 //Read the address from PMU
 Read X
  ..........
```

(c) RACEZ method

Figure 4: Different data race detection methods

```
Format of an entry:
  no:  record number
  tid: thread id
  type: type of the event
  event data: varies based on the type
  stack trace : optional stack trace for the event
```

Figure 5: The format of a logging entry.

1. The user selects an event and a sampling period. RACEZ uses the *Instruction_Retired* event.
2. Set the sampling buffer size. The kernel manages a sampling buffer where samples are saved on PMU interrupts. The application is notified by a signal only when that buffer becomes full. The cost of user level notification is thus amortized over a large number of samples. Since RACEZ needs to correlate each memory access with the thread's lockset at the time of the memory access, the sampling buffer size must be set to one entry to trigger a signal on each sample.
3. Create the PMU context for monitoring. Using the file descriptor, the application can then program the event and sampling period. The file descriptor is also used to bind the signal delivery to a particular thread[31].
4. Start the monitoring. After the PMU context is configured, a system call is invoked to start PMU monitoring for the specified thread.
5. Read PMU samples and restart monitoring. On each signal notification, samples are read from the sampling buffer. The PMU is stopped during this processing and must be restarted after processing is complete. Before restarting the PMU, it can be reconfigured. RACEZ uses this opportunity to adjust the sampling period dynamically. This optimization is discussed further in Section 4.2.

Steps 1–4 are implemented in a wrapper function for thread creation. For Step 5, the signal handler is provided as part of the RACEZ library.

3.4 Signal Delivery

Whenever a memory access is sampled by the PMU, it triggers an interrupt on the processor core where the access was sampled. The kernel stores the PMU sample in the user provided sampling buffer, and, on buffer overflow, delivers an asynchronous signal (RACEZ uses *SIGIO*) to the monitored application. For RACEZ, there are two basic requirements for this signal delivery mechanism:

- The signal should be dispatched immediately after the hardware interrupt happens.
- The signal must be dispatched to the correct thread, i.e., the thread in which the sample was captured.

The first requirement is necessary because RACEZ combines lockset information that is updated by the lock/unlock instrumentation with the memory access information. Consider the code in Figure 4a. If the write operation at line 4 is sampled by the PMU, and if the user level code only receives the signal at line 7, then the log recorded by RACEZ will contain the wrong lockset for the access because *Unlock* already updated that lockset.

The second requirement stems from the fact that both the memory access and the lockset should come from the same thread. When an instruction in a thread is sampled, if we cannot guarantee that the corresponding thread will receive the signal, the output record entry will be wrong. For example, in Figure 4a, if Line 4 in T1 is sampled, but T2 receives the signal, RACEZ would incorrectly report that T2 performed the access. While this seems trivial and self-evident, the POSIX (and in particular Linux) signal delivery mechanism does not guarantee this behavior. POSIX signals may be delivered to any thread in the process. For Linux, it is even trickier. Older Linux versions provided an extension to support precise asynchronous signal dispatching. However, this was changed at version 2.6.12. As a result, this problem existed in the Linux kernel until it was fixed [31] as part of the RACEZ implementation.

3.5 Implementation

We implemented RACEZ on Linux and support Intel PMUs using PEBS and AMD PMUs using instruction-based sampling (IBS). The whole PMU framework is implemented based on the API provided by Perfmon2[3] which provides system call APIs to interact with the PMU. RACEZ can be downloaded from `http://code.google.com/p/racez/`.

To load our instrumentation code, we currently use *LD_PRELOAD* to redirect standard *pthread* functions to RACEZ wrapper functions which include custom instrumentation. For memory allocation routines, we also redirect standard malloc/free functions to wrapper functions that record memory allocation information. RACEZ also provides interfaces which can be inserted into custom memory allocators to record memory allocation information. These interfaces are very useful because large applications often have their own custom memory allocators.

For server applications, we need to insert start and stop function calls into the application code in order to enable and disable monitoring when the programs are running. In our experiments, the start function is inserted at program startup and the stop function in the signal handler for shutting down the server (server applications are typically shutdown by sending a signal such as *SIGUSR1*). In a production environment, in order to monitor a server for short window of time, one would, for example, insert an HTTP request handler in the code, that allows a user to enable and disable RACEZ by sending an HTTP request to a special port. For standalone applications, users do not need to modify any of their source code.

```
function foo
{
  ......
  *p = var
  .....
}
```

(a) Code with a write

```
.....
IP1: mov  -0x8(%rbp),%edx
IP2: mov  %edx,(%rax)
IP3: addl 0x1,-0x4(%rbp)
.....
```

(b) Assembly for the write

Figure 6: Memory Address Computation

3.6 Offline Analysis

Race detection can be performed online, while monitoring an application. However, in order to minimize overhead, we adopted an offline approach. For Intel platforms, our offline tool first must convert the PMU samples into memory accesses. Each PMU sample contains the contents of the integer register file for the sampled instruction. We use MAO [2], an assembly-level analysis tool, to determine if the sampled instruction accesses memory, and if so, to compute the effective address accessed based on the register contents. For example, for the assembly code in Figure 6b, if IP2 has been sampled, a final memory address can be computed by reading the contents of register *rax*. For AMD platforms, the PMU samples already contain effective addresses, so this preprocessing is unnecessary. These memory accesses are fed into an Eraser[22] style lockset algorithm to read the trace and detect races. Section 4 will describe several extensions to this basic offline analysis to improve the probability of finding races.

4. PROBABILITY ANALYSIS AND OPTIMIZATION

As discussed in Section 1, any sampling based analysis method will potentially miss information. For data race detection this means that memory accesses are only caught with a certain probability, there is no guarantee that a specific memory reference will be sampled. In this section, we first present a formal probability analysis. We then propose three key optimization techniques to greatly improve the results. We evaluate the effects of these optimizations in Section 5.

4.1 Probability Analysis

For the sampling based techniques in RACEZ, the fundamental question to ask is: given a fixed sampling period *T* and a dynamic instruction stream *{*I_1*,* I_2*, ..*I_i*.., *I_s*}*, if two memory accesses involved in a race occur *m* and *n* times in the stream, respectively, what is the probability to catch both memory references at least once?[1]

To answer this question, we first compute the total number of samples as $t = \frac{s}{T}$(s is the total number of instructions and T is the sampling period). The probability *P* for catching both memory references at least once is:

$$P = 1 - \frac{\binom{t}{s-m} + \binom{t}{s-n} - \binom{t}{s-m-n}}{\binom{t}{s}}$$
$$\approx 1 - \left(1 - \frac{m}{s}\right)^t - \left(1 - \frac{n}{s}\right)^t + \left(1 - \frac{m}{s} - \frac{n}{s}\right)^t \qquad (3)$$

where the notation $\binom{n}{k}$ represents the number of ways to choose k items from a population of n. The approximation assumes that after an instruction is sampled, it is returned to the pool and can be sampled again. Additionally, it assumes that instructions are sampled independently. Obviously, the PMU cannot sample the same dynamic instruction twice. However, if $s >> m$ and $s >> n$, this assumption does not significantly affect the calculation. In practice, because racy memory accesses only occupy a small part of the dynamic instruction stream, this assumption is satisfied. Similarly, the PMU samples are collected periodically (not randomly), so they are not completely independent. To mitigate this, rather than relying on pure periodic sampling, each sample period includes a random delta. As an application of the above equation, for the pair *{s= 1,000,000,000, T=200,000,* $\frac{m}{s}$*=0.01%,* $\frac{n}{s}$*=0.01%}*, we compute that *t=5000* and final probability will be 15.5%.

The equation 3 shows that to improve the probability of detecting a race, we must increase t since $\frac{m}{s}$ and $\frac{n}{s}$ are properties of the program and are constant. We can increase t in two ways. First, we can increase s, then the length of time we sample the application[2] Second, we can collect more samples by effectively decreasing T.

Below we present 2 techniques to effectively reduce the sampling period T and discuss how we can increase s in the context of server applications.

4.2 Sampling Period Adjustment

While the sampling period can be reduced to increase the probability of catching a race, the overhead of sampling is proportional to the number of hardware interrupts. The shorter the sampling period, the more hardware interrupts will be generated. We provide a way to dynamically adjust the sampling period according to an *overhead budget* for different users. In Section 5 we evaluate the overhead of different sampling periods for different applications. Additionally, we randomize the sampling period by adding a randomized factor to the sampling period to avoid the problem when every sample fall into a synchronized pattern in some loops.

4.3 Sampling Skid: Opportunity and Challenge

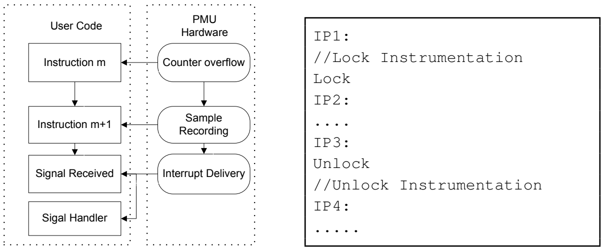
For each hardware interrupt, we can get two samples due to the skid introduced by the PMU. Figure 7 demonstrates the approach. If the retirement of instruction *m* causes *counter overflow*, the Intel PEBS hardware will record the state of instruction *m+1*. However, the signal indicating that the sample was collected will not be delivered until much later. Fortunately, the OS always delivers the current process state with any signal. Since this state is collected at the time of signal delivery, which is distinct from where the hardware recorded the process state, the OS process state can be used as a second sample, effectively halving the sampling period.

Unfortunately, the optimization is not this simple. For example, if *IP1* in Figure 7b(b) is *m+1*, and the signal is received at *IP2*, RACEZ will incorrectly assume that the memory access occurs with a lock held. This can be handled in two ways. First, a stall loop can be inserted in the synchronization primitives to reduce the probability that a sample collected before a synchronization is delivered after the synchronization. While this does increase the instrumentation overhead, it allows us to effectively halve the sampling period, thus decreasing other overhead and improving race detection probability.

Alternatively, a simple hardware extension can be used to overcome this. An additional register could be added to the PMU which gets recorded on each sample. RACEZ would increment this register each time it encounters a synchronization instruction. When the signal handler is invoked, instead of using the thread's current lock-

[1] Note that not all occurrences of the same instruction will be involved in a race. However, same memory operation with inconsistent lockset among its occurrences always indicates a potential problem.

[2] Increasing s will also increase m and n. The ratios $\frac{m}{s}$ and $\frac{n}{s}$ will remain constant.



```
IP1:
//Lock Instrumentation
Lock
IP2:
....
IP3:
Unlock
//Unlock Instrumentation
IP4:
.....
```

(a) signal skid problem (b) Example

Figure 7: Sample skid problem of PMU

set, it would use the lockset corresponding to the register value in the sample. This avoids any timing mismatch between the lockset data and the memory access data.

In our current implementation, we use the former approach and we treat the hardware solution as future work.

4.4 Extending Samples

We can effectively decrease the sampling period in another way. As described earlier, we apply an offline analysis to compute the effective address based on the register state for each event. Besides this address computation, RACEZ employs static instruction simulation based on the recorded register values. Static instruction simulation is based on static data flow analysis.

Given a set of register value for an instruction *i*, we can use these register value to compute memory address for another instruction *j* if the registers are not invalidated between *i* and *j*. For instance, if *IP1* in Figure 6b was sampled, since the value of register *rax* is not killed by the instruction at *IP1*, we can use the register value of *rax* at the IP1 to compute memory address of the instruction at *IP2* through forward simulation.Similarly, if we *IP3* were sampled, we could use reverse simulation to compute the memory address of the instruction at *IP2*.

RACEZ implements both forward and reverse simulation in MAO within one basic block. Extending beyond the basic block boundaries requires branch history information which is currently not collected. As a future work, we are considering to use branch tracing features in the PMU [26] to collect the branch history thus allow us to further enlarge our analysis scope. The sampling extension method is very effective and the final results are reported in Section 5.

4.5 Increasing the Sampling Window

Finally, the probability of catching a race can be increased simply by increasing the number of samples collected, i.e., make the program run longer or sample for a longer time. A server application running in a data center often runs on many machines concurrently. Consequently, the time can be effectively increased by sampling all the machines. Since all machines are running the same code, it does not matter which machine catches a race, as long as it is caught on some machine.

In Section 5, for the bugs we studied, we simply run the program longer to make memory accesses involved in a race occur more frequently. It is equivalent to monitoring the program constantly and reading multiple traces together to detect race.

5. EVALUATION

In this section, we first present our experimental platform and target benchmarks in Section 5.1. Section 5.2 reports our overall race detection results. Finally, we evaluate the overhead and false positives in Section 5.4 and Section 5.5.

Table 1: Benchmarks used in the evaluation.

Benchmarks	**Description**	**LOC**
Test	A custom testing case	
Apache Httpd	Web Server	220K
MySQL	Database Server	1.1M
PARSEC	A parallel benchmark suite	1.99M

Bug	BugId	Type	Description/Symptoms
Test		Benign	update the same variable in two threads
httpd-1	44402	Harmful	race in fd_queue.c
httpd-2	25520	Harmful	race in buffered_log_writer, corrupted logs
httpd-3	21287	Harmful	race in decrease_refcount, crash server
httpd-4		Benign	update *requests_this_child* in worker.c
httpd-5		False	false positives in apr_pool.c
mysqld-1	28249	Harmful	Wrong sync for query cache, wrong result
mysqld-2	791	Harmful	flushing log is not atomic, corrupted logs
mysqld-3	3596	Harmful	reference thr->proc is not atomic,cold,crash server
mysqld-4		Benign	Benign race in statistic_increment
mysqld-5		False	false positives in thr_lock.c

Table 2: Known Bugs we studied.

5.1 Experimental Platform and Benchmarks

The experiments were conducted on a machine with a 2.40GHz Intel Core 2 Q6600 processor, 4GB of memory. The linux kernel version is 2.6.30 with perfmon2[3] kernel patch.

We evaluated 2 real-world server applications as shown in Table 1, including a web server(Apache httpd) and a database server (MySQL). Besides, in order to learn the overhead for different kinds of applications, we also used a parallel benchmark suite (PARSEC) which includes several standalone multithreaded programs. The *Test* benchmark is written by ourselves for detection and overhead study. We extracted 11 races from 3 applications as shown in Table 2. For each race, we classify it into 3 categories: *Harmful*, *Benign*, *False*.

For httpd server, we use its own performance testing script *ab* as the client to generate queries. For the httpd-21287, we use the *httperf* as the testing client. For mysqld server, we adopt its testing script under *mysql-test* to produce detection testing inputs and the script under *sql_bench* as the overhead testing input. The PARSEC benchmark suite has its own testing script to report performance numbers.

5.2 Race Detection

5.2.1 Methodology

In general, we run applications with input that will manifest data races and monitor the execution with RACEZ to see if we can catch these data races. In principle, the execution and monitoring can be infinite. But in real scenarios, even with very low overhead, the monitoring framework such as RACEZ may only be allowed to execute for a period of time(e.g. 5-10 minutes) per day instead of monitoring all the time. The reason for this limitation is due to some requirements on real production systems which is out of the scope of this paper. To reflect this limitation, instead of executing applications infinitely, we execute them period by period. In each period, the application and RACEZ are cold started without using any data from previous execution. It is clear that with this experiment setup, RACEZ can only detect data races inside an execution period.

At a certain sampling rate, the length of period is almost proportional to the number of samples we get during the period. Thus, We define the *Experiment Unit* to represent the execution period of applications which is measured by number of samples we get during the period.

For each test case, we repeat *Experiment Unit* many times and RACEZ will report if it catches data races in the the *Experiment Unit*

Bugs	Experiment Unit(samples)	Running result		
		Base	OS Sample	Offline Ext.
Test	10,000	1%	3%	7%
httpd-1	80,000	1%	3%	4%
httpd-2	80,000	4%	4%	6%
httpd-3	80,000	x	x	x
httpd-4	8,000	1%	3%	5%
httpd-5	6,000	2%	4%	4%
mysqld-1	20,000	2%	4%	9%
mysqld-2	20,000	0%	1%	1%
mysqld-3	20,000	x	x	x
mysqld-4	10,000	1%	2%	3%
mysqld-5	10,000	3%	6%	8%

Table 3: Overall Detection Result for T=200,000. *Experiment Unit* is defined based on the number of samples that is equal to *s/T*. *OS Sample* is the optimization that can get another sample from OS in addition to the PMU sample. The *Offline Ext.* result is reported by adding the *OS sample* optimization. RACEZ cannot catch httpd-3 and mysqld-3 since they either are located in cold regions or crash the server immediately.

or not. We use the ratio of number of Experiment Unit in which RACEZ catches data races to the total number of *Experiment Unit* tested for the bug as the metric. In our experiment, the sampling rate is 1/200,000 which is an acceptable sampling rate for production systems. For a one minute Experiment Unit, this sampling rate normally results in 100,000 samples in our testing platform.

5.2.2 Detection Result

The race detection result is shown in Table 3. RACEZ can catch 9 bugs out of 11 bugs. The table also shows how long it takes for RACEZ to catch these bugs and the effects of two optimizations proposed in this paper. The column *base* shows result for basic RACEZ approach while the *OS Sample* column denotes the result after applying the optimization(Get an extra sample from OS in addition to the hardware PMU sample), and the *Offline Ext* column reports the result after applying offline sample extension optimization. For bugs that can be caught by RACEZ, the average ratio after the optimizations is around 5% which means we only need to monitor about 20 Execution Units to catch these data races. Considering each Exeuction Unit is usually 1 minutes, it only takes around 20 minutes for RACEZ to detect these data races. Since RACEZ can be deployed in multiple servers, the time required for RACEZ to catch bugs is likely much shorter.

5.2.3 Uncaught Data Races

As shown in Table 3, there are 2 bugs that RACEZ fails to catch. Taking the mysqld-3 bug as an example, the racy memory accesses are all located in a very infrequently executed code region. One of racy memory accesses will happen only once every 10^8 instructions given the trigger testing input. According to Equation 3, the probability to catch both memory accesses in one Experiment Unit(20,000 Samples) is only 0.0008%. On the contrary, for bugs that are caught by RACEZ, the probability to sample each of the racy accesses is much higher. For example, the two racy memory accesses in *httpd-2* bug will be executed once every 10^6 instructions, and the theoretical probability to catch the data race with 80000 samples is 2.2%(Note that the result in Table 3 is running experimental result, not theoretical result).

5.2.4 Effect of Offline Sample Extension

Table 4 shows the effects of offline sample extension that is described in Section 4.4. For the two memory accesses in each race , the extending phase can consistently extend average 4.1x and 6.3x samples.

As an example, in Figure 9, we list the final assembly code for the write statement in thread T2 of Figure 8 that corresponds to the *mysqld-1* bug. In the assembly code, Line 15 is the instruction we want to sample; however, because *rax* is not killed from Line 1 to Line 18, any sample from these 18 instructions can be finally attributed into the sample of Line 15 by applying our sample extension optimization. Such structure copy operations are pretty common in program, and it further justifies our sample extension optimization.

Bugs	Memory Access 1		Memory Access 2	
	Base	Extend	Base	Extend
Test	1	3	1	3
httpd-1	1	3	1	5
httpd-2	1	3	1	3
httpd-4	1	2	1	2
httpd-5	1	10	1	10
mysqld-1	1	3	1	18
mysqld-2	1	3	1	6
mysqld-4	1	4	1	4
mysqld-5	1	6	1	6
average	1	4.1	1	6.3

Table 4: The sampling extension optimization result.

5.3 Real Races Study

In this section, we present our races detection experiences in real applications. The bugs demonstrate the weakness of traditional in-house testing and detection methods. Although RACEZ can only detect them with a probability, it at least shows a low overhead and non-invasive solution to catch them.

5.3.1 Race Exposed by Special Input

Some races can only be exposed by special external input. In this part, we present one of such bugs found in *MySQL*. The code is shown in Figure 8. This bug corresponds to mysqld-1 in Table 3.

```
T1:
Query_cache::store_query() {
  LOCK L1
  if (!handler->register_query_cache_table(...)
  UNLOCK L1}

ha_myisam::register_query_cache_table(...) {
⇒ actual_data_file_length=
    file->s->state.state.data_file_length;
}
T2:
void thr_unlock(...)
{
  LOCK L2
  (*lock->update_status)(...);
  UNLOCK L2
}
void mi_update_status(...)
{
⇒  info->s->state.state= *info->state;
}
```

Figure 8: Sample code extracted from a race of MySQL

```
1  mov     (%rax),%rax
2  mov     -0x10(%rbp),%rdx
3  mov     0x8(%rdx),%rdx
4  mov     (%rdx),%rcx
5  mov     %rcx,0x18(%rax)
6  mov     0x8(%rdx),%rcx
7  mov     %rcx,0x20(%rax)
8  mov     0x10(%rdx),%rcx
9  mov     %rcx,0x28(%rax)
10 mov     0x18(%rdx),%rcx
11 mov     %rcx,0x30(%rax)
12 mov     0x20(%rdx),%rcx
13 mov     %rcx,0x38(%rax)
14 mov     0x28(%rdx),%rcx
15 ⇒ mov     %rcx,0x40(%rax)
16 mov     0x30(%rdx),%rdx
17 mov     %rdx,0x48(%rax)
18 mov     -0x10(%rbp),%rax
```

Figure 9: The final assembly code for the write statement in thread T2 of code in Figure8

This bug can only be triggered when the following conditions are satisfied:

- Enable the *concurrent_insert=1* to allow concurrent insertion when other query operations to the same table are still pending.
- Set special *query cache* flags.The *query cache* is a common optimization for database server to cache previous query results.
- A thread added to lock one of the two involved tables

If the data race happens, the second query will use old value in query cache and return wrong value while not aware of the concurrent insert from another client. In our experiment, we send the buggy input to the server frequently to make the race occur more often, and as shown in Table 3, we successfully caught the race with desirable probability at a low sampling rate.

5.3.2 Corrupted Data or Logs

As shown in Table 2, some races only will produce wrong result or corrupted logs. The httpd-2 is a race condition bug that will finally make the server produce corrupted logs. As the sample code shown in Figure 10, the shared variable references *buf->outcnt* at Line 2 and 7 should be guarded inside a critical section. However, the buggy code did not maintain such atomicity for Line 2 and Line 7. While there is context switch between Line 2 and 7, the result will be wrong. This bug is located in hot region and RACEZ can catch it easily based on the lockset algorithm.

```
1 ap_buffered_log_writer(....) {
2   for (i = 0, s = &buf->outbuf[buf->outcnt];
3        i < nelts; ++i) {
4        memcpy(s, strs[i], strl[i]);
5        s += strl[i];
6   }
7   buf->outcnt += len;
8 }
```

Figure 10: The sample code for race httpd-2. The race is between the read and write references at Line 2 and 7.

The mysql-2 is another race condition that will produce wrong log files and RACEZ can catch it easily.

5.4 Overhead Study

5.4.1 Overall Runtime Overhead

Figure 11 shows the overall overhead result for 3 benchmarks. The average slow down at *T=200,000* is 2.8% which is practical for production usage. Note that, at *T=20,000*, the slow down quickly increase into 30%. In the following sections, we will further analyze the overhead. For each benchmark, we run it 5 times and get the average number for the final result.

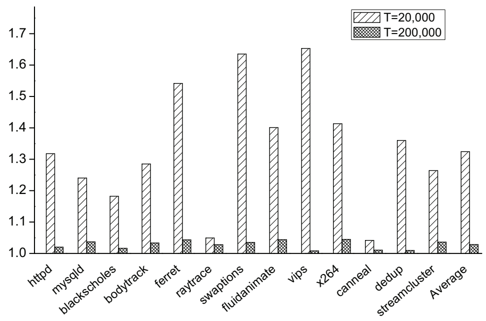


Figure 11: The overhead for different benchmarks. *T=20,000* means the sampling period is 20,000. The average slow down is 2.8% for *T=200,000*. The result has been normalized according to the base result.

5.4.2 Factors of Runtime Overhead

The overhead of runtime monitoring mainly comes from two parts: lockset instrumentation and signal delivery. Furthermore, since we instrument the standard and custom memory allocators to filter out false positives , we also measure their overhead. Finally, we include the overhead of the two online optimizations as mentioned before. The breakdown result is reported in Figure 12. We can see that the *signal handler* processing is the major source of overhead for RACEZ. This processing requires several OS system calls to interrupt user level code and read the information from OS kernel and hardware. The overhead of *sampling adjust* is moderately high because it also needs to invoke system calls to set the hardware control bits. The *OS sample* incurs a very small overhead. The overhead of *lockset* instrumentation is determined by the lock usage of different benchmarks. Since *mysqld* uses much more locks to protect concurrent accesses of same table, its overhead is higher than the other two benchmarks. We insert a number of lock/unlock and memory allocation functions into the custom *Test* benchmark to study the breakdown of the overhead. We do not include the overhead of stacktrace that is relatively high in our experiments. The stacktrace function is turned off by default in RACEZ. We are also developing a much cheaper stacktrace collection solution.

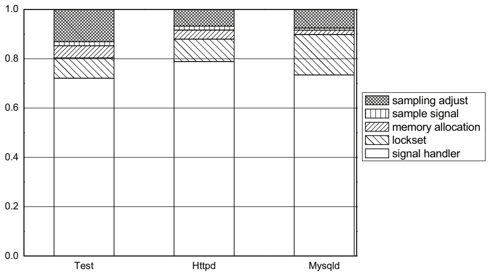


Figure 12: The breakdown of overhead.

For different sampling period, we give the overhead result in Figure 13. We can see that the overhead is nearly linear with sample period. This is because the overhead is mainly determined by the number of hardware interrupts which is controlled by sampled period as the Figure 12 shows.

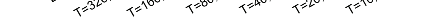


Figure 13: The overhead for httpd with different sampling periods.The input is fixed at 100,000 requests in total. The x-axis is the sampling period and y-axis is the throughput result, i.e., requests per second.

5.4.3 Scalability Analysis

Normally, a large server application will initialize large number of threads during startup and put them into a thread pool for later use. We also measure the overhead by varying the number of threads for monitored application in Figure 14. The result shows that the number of threads does not matter for RACEZ. This is because hardware PMU only has effects for scheduled threads. If one thread is switched out by kernel and does not run on any of cores in CPU, the PMU will not sample it. The final number of hardware interrupts is only determined by the number of concurrent running threads on CPU which is again controlled by the number of physical CPU cores. One subtle limitation is that RACEZ relies on kernel modules to maintain a context for each thread, thus we need to enlarge some critical OS kernel resources, such as locked memory, in order to monitor applications which have large number of threads.

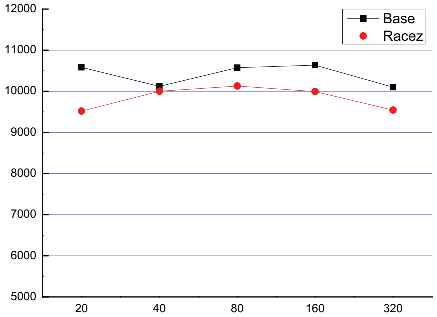


Figure 14: The overhead for httpd with different number of threads. The sampling period is fixed with *T=200,000*. The x-axis is the number of threads and y-axis is the throughput result, i.e., requests per second.

5.5 False Positives Classification and Solution

RACEZ will produce a lot of false positives because it used the lockset algorithm. The major source of false positives comes from memory allocator and happen-before edges. For example, during a monitoring of httpd with 600M log file size, it will produce 131 warnings. After applying the standard and custom memory allocation instrumentation, the number will dramatically drop down to 11 warnings. Among these 11 warnings, only two are real races. Other 9 warnings are false positives caused by happen-before edges. Although 80% false positive rate is very high, however, we can employ the method proposed in [24, 28] to filter out these warnings.

6. LIMITATIONS AND DISCUSSIONS

6.1 A Preliminary Race Characteristic Study

As we have shown in the early part of the paper, it is very hard for RACEZ to catch races that occurred very infrequently. In addition, if most data races causes immediate system crash, users don't need RACEZ to know there are data races. So we face a question to answer: Are there many data races that happens frequently and don't crash system immediately?

It is difficult to answer the question and there're contradictory evidence in the literature. While [29] reveals that concurrency bugs are highly correlated with program crash, [5] find that potentially harmful data races, which occur quite frequently without causing any crashing errors, are also common in reality.

To understand this problem better, we collect 14 race condition bugs from previous research works[29, 28, 1] and perform a preliminary study on data race characteristics in real applications. We categorize each data race with the following two criteria:

- Does the race condition bug finally crash the server or only produce wrong result?
- Is the race condition bug located in hot executed code region? We define one racy memory access per million instructions as hot code.

Table 5 shows the result. Among the 14 data race bugs studied, 7 will crash systems immediately and 7 will not. Among 7 bugs that will not crash systems immediate, 4 of them are in hot region. Although the study is preliminary, we believe it hints that bugs that can be caught with RACEZ are not rare.

Total	crash server		wrong result/logs	
	hot	cold	hot	cold
14	5	2	4	3

Table 5: Race condition bug characteristics

6.2 False Positive Reduction

Another problem is that we are using lockset method to reduce the overhead, which is known as an imprecise technique. Fortunately, recent research has made progress on classifying real races from original races warning produced by imprecise races detection tools. One of such techniques is RACEFUZZER[24] which is already used by one widely used valgrind based tool[25]. The method proposed in [28] also can be used to filter out false positives.

6.3 Overhead in Sampling Window

The overhead over the sampling window of RACEZ can be much larger than lightweight software instrumentation method. In our experiment, it is about 5000 cycles(including signal handler processing, PMU information collecting, etc). That means it may affect the latency of individual client request if it is just interrupted by hardware PMU. However, as the throughout result shows, the overall overhead for all requests is very slow.

7. RELATED WORK

The most closely related work of RACEZ are LiteRace[14] and Pacer[5] that both proposed a sampling-based techniques to detect races with software instrumentation. However, they either need to clone the program that causes at least 2X code size expansion[14] or only can be applicable to managed Java language program[5]. As discussed in Section 1, even though they can achieve low overhead, their invasive instrumentation makes it hard to be applied in some production environments.

Static data races[9, 27] detection techniques have the advantage that they can exploit all execution paths through data flow analysis thus do not have testing input problem as dynamic analysis. However, static methods always produce large number of false positives since it is hard to model pointer aliasing and synchronization mechanism precisely.

In literature for data races detection, most of them are dynamic methods. It can be further classified as lockset based methods and happen-before based methods. Lockset methods [22] usually report false positives due to the widely used other synchronization operations, such as signal-wait, custom synchronization mechanisms,etc. In contrast, pure happen-before based methods consider all synchronization operations, and apply a vector clock algorithm to order all memory accesses in different threads. Happens-before methods were known as expensive methods because both the instrumentation and detection algorithm are expensive. Recently, some new lightweight vector clock algorithms[10] were proposed to improve the detection overhead. However, the overhead of these methods is still very high and only can be used during pre-deployment testing.

Software instrumentation based solutions always incur significant slow down to the original programs, thus several hardware solutions [16, 30] are proposed. They always evaluate their methods on simulators thus the effectiveness is hard to prove. Our solution can also be treated as a hardware solution, but we use the PMU hardware which is popular and fairly mature on commodity processors.

Besides data races detection, several low overhead dynamic monitoring techniques are proposed to detect software defects when the applications are deployed. QVM[4] incorporates an overhead manager component in JVM to control the overhead and detects software defects in deployed systems. Dimmunix[11] uses a quite low overhead monitoring methods and can successfully defend against deadlocks in real systems. Rx[20] makes an interesting observation that software bugs are sensitive to the underlying environment thus are likely to disappear after modifying the environments. However, for race detection, because of its inherent high overhead, few works have been done towards deployed systems.

Finally, PMU based sampling[26, 8, 7, 3] is widely used to identify performance problem for production applications when they are deployed. Instead of focusing on performance problem, we use PMU sampling to improve correctness of programs.

8. CONCLUSION

In this paper, we presented RACEZ, an approach to detect data races in production systems with PMU sampling techniques. We show that RACEZ can catch reasonable portion of data races without big overhead and aggressive instrumentation. Although PMUs are originally designed to catch performance characteristics of applications, we demonstrate that they are versatile and can be employed for bug detection purposes effectively. We expect this paper will motivate more new usage model of PMUs.

9. ACKNOWLEDGEMENTS

We would thank anonymous reviewers for their useful suggestions. We thank Zhizhong Tang, Dehao Chen, Jidong Zhai, Yan Zhai, Jiangzhou He, Tian Xiao, Sharad Singhai for their valuable feedbacks. The research is supported by Google Research Award 2009 and partially supported by IBM CAS Project No. 674. The student author of this paper is also supported by the National Science and Technology Major Project of China(2009ZX01036-001-002).

10. REFERENCES

[1] Concurrency bugs collection. `http://www.eecs.umich.edu/~jieyu/bugs.html`.

[2] Mao - an extensible micro-architectural optimizer. `http://code.google.com/p/mao/`.

[3] perfmon2: the hardware-based performance monitoring interface for linux. `http://perfmon2.sourceforge.net/`.

[4] M. Arnold, M. T. Vechev, and E. Yahav. QVM: an efficient runtime for detecting defects in deployed systems. In *OOPSLA*, pages 143–162. ACM, 2008.

[5] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional detection of data races. In *PLDI*, 2010.

[6] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, pages 167–178, 2010.

[7] D. Chen, N. Vachharajani, R. Hundt, S. wei Liao, V. Ramasamy, P. Yuan, W. Chen, and W. Zheng. Taming hardware event samples for fdo compilation. In *Code Generation and Optimization(CGO)*, April 24-28, 2010.

[8] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Z. Chrysos. *ProfileMe*: Hardware support for instruction-level profiling on out-of-order processors. In *MICRO*, pages 292–302, 1997.

[9] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252, New York, 2003.

[10] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. In *PLDI*, pages 121–133, 2009.

[11] H. Jula, D. M. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, pages 295–308, 2008.

[12] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339, 2008.

[13] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS*, pages 37–48, 2006.

[14] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *PLDI*, pages 134–143, 2009.

[15] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, pages 267–280, 2008.

[16] A. Muzahid, D. S. Gracia, S. Qi, and J. Torrellas. Sigrace: signature-based data race detection. In S. W. Keckler and L. A. Barroso, editors, *ISCA*, pages 337–348. ACM, 2009.

[17] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPOPP*, pages 167–178, 2003.

[18] S. Park, S. Lu, and Y. Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS*, pages 25–36, 2009.

[19] E. Pozniansky and A. Schuster. Multirace: efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience*, 19(3):327–340, 2007.

[20] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies - a safe method to survive software failures. In *SOSP*, pages 235–248, 2005.

[21] P. Sack, B. E. Bliss, Z. Ma, P. Petersen, and J. Torrellas. Accurate and efficient filtering for the intel thread checker race detector. In *ASID*, pages 34–41, 2006.

[22] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst*, 15(4):391–411, 1997.

[23] SecurityFocus. Software bug constributed to blackout. `http://www.securityfocus.com/news/8016`.

[24] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, pages 11–21, 2008.

[25] K. Serebryany and T. Iskhodzhanov. Threadsanitizer – data race detection in practice. In *WBIA09*, December 12, 2009.

[26] A. Shye, M. Iyer, V. J. Reddi, and D. A. Connors. Code coverage testing using hardware performance monitoring support. In *AADEBUG*, pages 159–163, 2005.

[27] J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In *ESEC/SIGSOFT FSE*, pages 205–214, 2007.

[28] J. Wu, H. Cui, and J. Yang. Bypassing races in live applications with execution filters. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, 2010.

[29] W. Zhang, C. Sun, and S. Lu. Conmem: detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS*, pages 179–192, 2010.

[30] P. Zhou, R. Teodorescu, and Y. Zhou. Hard: Hardware-assisted lockset-based race detection. In *HPCA*, pages 121–132, 2007.

[31] P. Zijlstra. Per-thread self-monitoring official linux support patch. `http://lkml.org/lkml/2009/8/4/128`.