ProRace: Practical Data Race Detection for Production Use

Tong Zhang Changhee Jung Dongyoon Lee
Virginia Tech
{ztong, chjung, dongyoon}@vt.edu

Abstract

This paper presents PRORACE, a dynamic data race detector practical for production runs. It is lightweight, but still offers high race detection capability. To track memory accesses, PRORACE leverages instruction sampling using the performance monitoring unit (PMU) in commodity processors. Our PMU driver enables PRORACE to sample more memory accesses at a lower cost compared to the state-of-the-art Linux driver. Moreover, PRORACE uses PMU-provided execution contexts including register states and program path, and reconstructs unsampled memory accesses offline. This technique allows PRORACE to overcome inherent limitations of sampling and improve the detection coverage by performing data race detection on the trace with not only sampled but also reconstructed memory accesses. Experiments using racy production software including *apache* and *mysql* shows that, with a reasonable offline cost, PRORACE incurs only 2.6% overhead at runtime with 27.5% detection probability with a sampling period of 10,000.

1. Introduction

In the manycore era, concurrency errors are more common than ever in multithreaded software [15, 37]. They are a frequent source of persistent errors in many production applications, and they have caused many serious real-world problems including the Northeast blackout [51], mismatched Nasdaq Facebook share prices [44], and security vulnerabilities [59].

Towards addressing this problem, the development of efficient dynamic data race detectors has been a focus of researchers in industry [19, 52, 61] and academia [12, 14, 45]. However, precise data race detection requires monitoring every memory operation at runtime, leading to high performance overhead. For instance, FastTrack incurs a 8.5x slowdown for Java programs [14] and a 57x slowdown for C/C++ programs [11]. In industry, Intel's Inspector XE with dynamic binary instrumentation incurs a 200x slowdown [50], and Google's ThreadSanitizer with static instrumentation incurs a 12x slowdown [63]. The high runtime overhead prohibits the use of these dynamic race detectors in production runs, relegating them to pre-deployment testing only.

Unfortunately, despite undergoing extensive in-house testing, races often exist in deployed software and manifest in customer usage [29, 52, 53]. These test escapes occur because data races are highly sensitive to thread interleavings, program inputs, and other execution environments that testing cannot completely cover [2, 63]. For the same reasons, data races are notoriously difficult to reproduce and fix after being observed in a production run. Consequently, there is an urgent need for a lightweight data race detector that can monitor production runs.

In production settings, it makes sense to trade off soundness (may miss data races) for performance. Sampling [3, 13, 40, 53] has been proposed as a promising technique to address the problem. However, LiteRace [40] and Pacer [3] still incur unaffordable slowdown for some applications (e.g., Pacer [3] adds 86% overhead at the 3% sampling ratio) due to code instrumentation based runtime checks. Though DataCollider [13] uses hardware breakpoint support instead, their detection coverages are limited to sampled accesses only. RaceZ [53] pioneered the use of hardware performance monitoring unit (PMU) to sample memory accesses, but it has to keep the low sampling frequency for performance thereby compromising the detection coverage.

This paper presents PRORACE, a new practical sampling-based data race detector for production runs. PRORACE is *lightweight*, minimally affecting the application execution; *transparent*, requiring neither recompilation nor static analysis; and *effective*, ensuring high race detection coverage.

PRORACE consists of online program tracing and offline trace-based data race analysis. Though offline analysis is required, the principal advantage of PRORACE is that very low runtime overhead of the online part enables PRORACE to monitor real-time, interactive, or internetworked applications at nearly full speed.

PRORACE makes use of the hardware PMU in commodity processors to monitor an unmodified program at a very low overhead. To be specific, PRORACE samples

ASPLOS '17 April 8–12, 2017, Xi'an, China.
Copyright © 2017 ACM 978-1-4503-4465-4/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3037697.3037708

memory accesses using Intel's Precise Event Based Sampling (PEBS) [20]. PRORACE's newly designed PEBS driver avoids unnecessary kernel-to-user copying and sampled data processing, reducing overhead by more than half compared to the latest Linux PEBS driver. This allows PRORACE to take much more samples for a given performance budget, enhancing its detection coverage.

During the offline phase, PRORACE reconstructs unsampled memory accesses to overcome the inherent limitation of sampling and to increase data race detection coverage further. The key idea is to replay the program from each sample and reconstruct the addresses of other memory instructions. Over the sampling, PEBS provides not only the sampled instruction but also its architectural execution context (e.g., register states) at sample time. PRORACE re-executes the program binary starting from each sampled instruction with the register states, and re-calculates the addresses of unsampled memory operations while emulating register and memory states.

Furthermore, to recover more memory accesses around each sample, PRORACE collects the complete control-flow trace using Intel's Processor Trace (PT) [21], a new feature in the Intel processor's PMU, at runtime. The control-flow information guides which path to take during the offline replay, enabling PRORACE to reproduce many other unsampled memory operations preceding and following each sample along the observed program path.

Finally, PRORACE analyzes the recovered memory trace and the synchronization trace, to detect data races using the happens-before based race detection algorithm [14].

This paper makes the following contributions:

- PRORACE presents a lightweight, transparent, and effective data race detector that can be easily deployed to monitor production runs.
- PRORACE proposes a new methodology to reconstruct unsampled memory addresses using the control-flow trace collected at runtime. To the best of our knowledge, PRORACE is the first software scheme that demonstrates how commodity hardware support for control-flow tracing can be used to enable the forward and backward reconstruction of unsampled memory trace. The proposed solution can benefit future research on runtime monitoring beyond race detection.
- This paper describes a PEBS driver that is many more efficient than the state-of-the-art Linux PEBS driver.
- The experiments using production software including *apache* and *mysql* show that PRORACE can detect significantly more races than RaceZ, a PEBS based race detector, at a much lower overhead.

2. Motivation

This section discusses the limitations of recent sampling-based and static-analysis-combined dynamic data race detection techniques when used in production environments, and motivates the need for a new approach.

LiteRace [40] and Pacer [3] pioneered the use of sampling for reducing the overhead of dynamic data race detection. LiteRace focuses on sampling more accesses in infrequently-exercised code regions, based on the heuristic that for a well-tested application, data races are likely to occur in such a cold region. On the other hand, Pacer uses random sampling and thus its coverage is approximately proportional to the sampling rate used. However, these code instrumentation-based race detectors cause an unaffordable slowdown for some applications, and their detection coverage is limited to the sampled accesses only. For example, though LiteRace shows low 2-4% overhead for Apache, it makes CPU-intensive applications 2.1-2.4x slower, and incurs 1.47x slowdown on average for their tested applications. Similarly, Pacer also reports the average of 1.86x overhead at the 3% sampling frequency.

DataCollider [13] and RaceZ [53] avoid code instrumentation and thus incur a very low overhead, but suffer from low detection coverage. DataCollider [13] makes use of hardware debug breakpoints. After sampling a code/memory location, it sets a data breakpoint and inserts a time delay. A trap during this delay indicates a conflicting access from another thread. Though longer timing delays increase the likelihood of overlapping data races, they also increase the overhead. In addition, hardware restrictions limit the number of concurrently monitored memory locations to four in the latest x86 hardware [22].

RaceZ leverages Intel's PEBS to sample memory accesses. However, due to its reliance on the inefficient Linux PEBS driver, RaceZ has to use a low sampling frequency for performance, thereby compromising the detection coverage. RaceZ also attempts to reconstruct unsampled memory accesses, but its scope is limited to a single basic block. This work shows that PRORACE has much less overhead, but detects significantly more data races compared to RaceZ.

Another line of work takes a hybrid static-dynamic approach. RaceMob [29], a recent low-overhead solution, employs static analysis [56] to compute potential data races, and crowdsources runtime race checks across thousands of users. To limit the overhead each user may experience, RaceMob requires a large number of runs to distribute checks, and the number of runs required depends on the precision of the static analysis. Elmas et al. [12] and Choi et al. [6] are other examples that make use of static data race analysis to reduce runtime cost. In spite of its benefits, static analysis often suffers from precision and scalability issues for large-scale applications, and the recompilation requirement is often not a viable option in production settings.

In summary, each of the current dynamic data race detectors lacks one or more of the critical criteria for production run monitoring: performance (low overhead), transparency

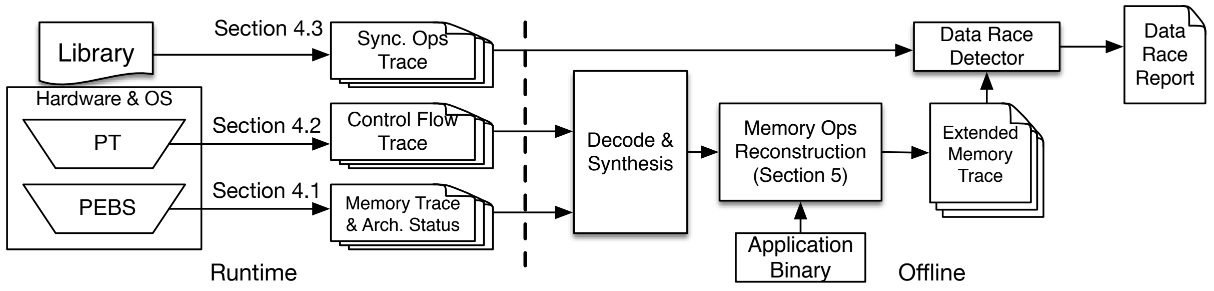

Figure 1: Overview of the ProRace Architecture.

(no recompilation), and effectiveness (high detection coverage).

3. Overview

The goal of ProRace is to provide lightweight yet effective race detection for practical use in a production environment. We envision a production environment similar to Google/Facebook's real-world datacenter in which various traces of production applications are already collected for monitoring purposes, and dedicated analysis machines exist in the datacenter to process the collected trace [26, 48]. In such environment, runtime monitoring overhead is much more critical concerns than the size of trace and offline analysis overhead. Production and analysis machines share a separate network, and thus writing a trace has a minimal impact on the QoS of production applications that use another network. Analysis machines can periodically process the trace to delete the ones analyzed in prior periods.

Figure 1 shows an overview of ProRace's two-phase architecture: online program tracing and offline data race detection. The online stage leverages the hardware PMU to trace a program execution at low overhead. Specifically, PEBS is used to collect the sampled memory access trace. PEBS provides both the sampled instruction and the architectural execution context (e.g., register states) at the sample time. PT is used to obtain the complete control-flow trace. The online stage also tracks the synchronization operations for later use in data race detection.

The offline stage first combines the memory access and control flow traces into a time-synchronized trace. Next it reconstructs *unsampled* memory operations. This is the critical step that allows ProRace to achieve higher detection coverage than other sampling-based approaches. Using the sampled instruction, register states, and control-flow information, ProRace replays the program and recomputes the addresses of unsampled memory accesses around each sample. The unsampled memory instructions whose target addresses can be reconstructed during this step are included in an *extended* memory access trace. Combining this with the synchronization trace, ProRace performs happens-before based data race detection using the FastTrack [14] algorithm to detect data races.

ProRace improves existing PMU-sampling-based data race detection in three ways. First, ProRace presents a PEBS driver much more efficient than the latest Linux PEBS driver. The improved design allows ProRace to take more samples for a given performance budget, enhancing its race detection coverage. Second, ProRace recovers unsampled memory accesses. ProRace re-executes the program binary starting from each sampled instruction with the PEBS-provided register states reconstructing the unsampled memory accesses while emulating register and memory states. Third, ProRace uses the PT-collected control-flow trace to choose which path to take during the offline binary re-execution. This permits ProRace to recover many other unsampled memory operations around each sample along the observed program path.

4. Lightweight Program Tracing

This section presents how ProRace traces a program execution at low overhead. At runtime, ProRace collects three type of traces: memory access samples, control-flows, and synchronization operations.

4.1 PEBS-based Memory Access Sampling

ProRace samples memory accesses using PEBS [20]. PEBS users can specify types of events to monitor such as retired memory instructions and taken branches, as well as whether to sample user-level or both user- and kernel-level events. ProRace tracks only the user-level retired load and store instruction events because of its interests in application memory accesses for data race detection.

PEBS enables users to set a *sampling period k* for each monitored event type. After every *k* events of a given type, PEBS delivers the sampled event along with its architectural execution context at the sample time such as register values, the time stamp counter (TSC)[1], but not memory states, to the corresponding listener.

Care must be taken when choosing the sampling period. Small values of *k* yield more samples but higher performance

[1] In old Intel processors, the PEBS samples did not include the time stamp, and the OS interrupt handler logged its wall-clock time during the processing. As a result, there was a small timing gap between the actual hardware sample time and the interrupt handler logging time. However, this is no longer an issue in recent processors such as Skylake and Broadwell.

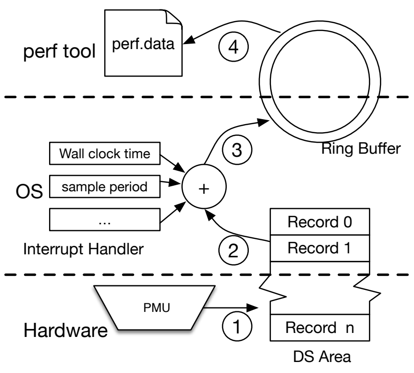

Figure 2: The vanilla Linux PEBS driver

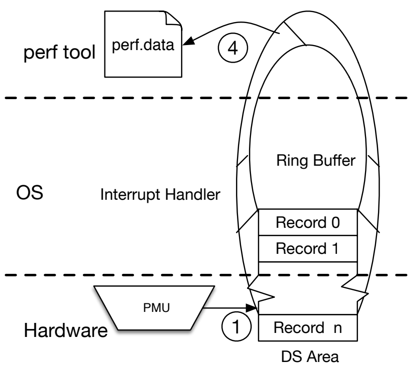

Figure 3: The ProRace PEBS driver

overhead. In addition, samples may be dropped if the kernel finds that too much time has been spent on the interrupt handling.

4.1.1 The Current Linux PEBS Driver

While the previous version of the Linux PEBS driver delivered every event using an overflow interrupt, a mechanism called *Debug Store (DS)* was added in the 4.2 Linux kernel to reduce the interrupt frequency. Figure 2 illustrates the interactions between the hardware PEBS, the OS interrupt handler, and the user-level *perf* tool.

DS permits PEBS to automatically store samples in a kernel-space buffer referred to as the *DS save area* whose default size is 64 KB (step ①). The interrupt is delivered only when the DS buffer is nearly full, reducing the frequency of interrupts.

On each interrupt, the OS interrupt handler processes the raw 'PEBS events', adding additional information such as wall-clock time, sample size, and sample period (step ②), and yielding 'perf events'. It then copies the perf events into another buffer, a ring-buffer shared with the user-land *perf* tool, resetting the DS save area for further PEBS events (step ③).

Finally, the *perf* tool polling on the ring-buffer commits the perf events to a file (step ④). Since the user-land perf tool may be configured to monitor incoming data from different cores, and store them into the same file, the events in the file may not be ordered sequentially. Thus, it reads the entire file later before its exit to sort all events and include other information.

Though DS support reduces the runtime overhead in using PEBS compared to the naive interrupt-per-sample mechanism, our experimental results show that a sampling period more frequent than 10K-100K will still incur slowdowns approaching 10%.

4.1.2 PRORACE's New PEBS Driver

PRORACE presents a new PEBS driver that significantly lowers the performance overhead in using PEBS. The new design makes it possible to collect more samples for the same performance cost. Figure 3 shows our new design incorporating the following changes:

First, PRORACE eliminates expensive kernel-to-user copying by maintaining a single ring buffer named *aux-buffer*. The ring buffer is partitioned into multiple 64 KB segments. Initially, PRORACE provides PEBS with one segment of the ring buffer; when PEBS finds it full and raises an interrupt, the OS interrupt handler simply proffers the aux-buffer's next available segment. The user-level *perf* tool eventually comes into play, dumping the segment filled with records into the file and making it available for further tracing. In this design, the interrupt handler need only swap the segment locations for PEBS similar to conventional double-buffering. The Linux driver for (newer) Intel's PT incorporates a similar single buffer design, but it is not used in the PEBS driver.

Second, PRORACE skips data processing irrelevant to data race detection during PEBS sample handling. Specifically, PRORACE does not add the metadata information (step ② in Figure 2).

Lastly, given a sampling period *P*, the sampling period is initially set to a random value between one and *P*. At the first event the sampling period is changed to *P*. This enables PRORACE to start sampling at a random location per thread on each run, increasing its sampling diversity to ultimately improve its race detection capability.

Experimental results in Section 7.2 show that the new driver reduces runtime overhead significantly, making it possible for applications to use a small sampling period.

4.2 PT-based Control-flow Tracing

PRORACE uses Intel's PT [21] to collect program control flows. PT is an extension to the PMU architecture for Intel's Broadwell and Skylake processors. At runtime, PT records the executed control-flow of the program in a highly-compressed format. Unlike event-based PEBS, PT keeps track of complete control-flow information including (indirect) branch target and call/return information without loss of precision. Nonetheless, PT incurs only a very small overhead because the tracking is done off the critical path and by hardware. This is significant improvement over previous (relatively) high overhead and limited tracking features such as Branch Trace Store (BTS) and Last Branch Record (LBR) in old processors.

ProRace's PT driver also implements the code-region based control-flow tracing feature. The PT hardware allows users to specify four memory regions of interest from which to collect the program control-flow. ProRace is configured to monitor only main executable memory regions because of its interests in detecting application data races (assuming no Just-In-Time compilation). Depending on use cases, dynamic library code regions may be included, or static library code regions may be excluded, by examining the symbol table.

The memory access trace collected by PEBS and the control flow trace collected by PT can be easily combined for offline processing because both traces include the per-core TSC value.

4.3 Synchronization Tracing

ProRace uses happens-before based data race detection [14] for precision (no false positives), but offloads the expensive vector-clock processing to the offline phase. At runtime, ProRace collects per-thread synchronization logs along with its type (e.g, lock/unlock), variable (e.g., lock variable address), and TSC value. The per-thread logs can be easily synchronized offline because recent processors support invariant TSC [18] that is synchronized among cores and runs at a constant rate.

For transparency, ProRace uses *LD_PRELOAD* to redirect standard *pthread* functions to ProRace instrumented functions. In addition, ProRace tracks dynamic memory allocation/deallocation. Suppose that one object is freed, and another object happens to be allocated to the same memory location. There can be no race condition between two different objects, but a data race detector may falsely report one as their memory addresses are the same. To avoid this kind of false positive, many data race detection tools keep track of malloc and free, and so does ProRace.

5. Recovering Unsampled Memory Accesses

ProRace leverages PMU-based instruction sampling to collect memory accesses. As with all the sampling-based race detectors, it might end up with false negatives due to unsampled memory accesses. To overcome the inherent limitation of sampling, ProRace reconstructs unsampled memory accesses offline by re-executing the program binary around each PEBS-sampled instruction with forward replay (Section 5.1) and backward replay (Section 5.2). In addition, ProRace leverages full control-flow information recorded by PT to guide which path to execute during both replays.

For each PEBS sample, ProRace alternates forward and backward replays following the observed program path as shown in Figure 4. Basically, the forward replay corresponds to the re-execution of the unsampled instructions between the current and the next samples, while the backward replay to that of those preceding the current sample for dealing with the instructions missed by the forward replay. ProRace repeats the replays until there is no more PEBS sample to be processed. The rest of this section details the path-guided binary re-execution and how it can reconstruct unsampled memory accesses.

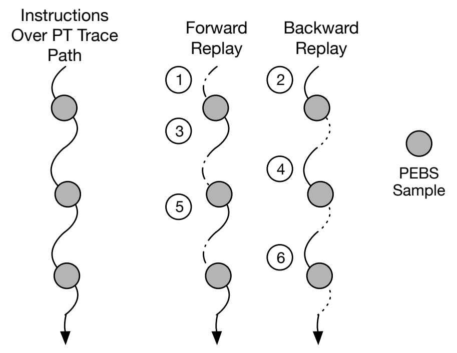

Figure 4: Forward and Backward Replays.

5.1 Forward Replay

When an event is sampled, PEBS not only offers precise instruction location of the event, but also provides the architectural states such as the entire register file contents at the sample time. By leveraging such execution contexts as inputs, ProRace re-executes the program binary from each PEBS sample point over the program path reconstructing the addresses of the memory operations. Such path-guided binary re-execution is called *forward replay*.

For each PEBS-sampled instruction, ProRace restores the register file contents, and attempts to execute every following instruction over the program path until the next PEBS-sample point is reached. For each instruction being executed, ProRace checks if the operands are available at the time of the instruction execution. For this purpose, ProRace keeps track of the architectural status by book-keeping all the register and memory values in a special hash table called *program map*.

ProRace simply treats every memory location as *unavailable* in the first place. The destination register of load instructions becomes unavailable when they read from unavailable memory locations. If all the operands of an instruction being replayed are not *available*, ProRace simply skips the instruction setting all its outputs as unavailable. Otherwise, ProRace executes the instruction updating the resultant architectural status such as registers and memory locations in the program map. Note that the memory emulation requires a special care for correctness, and thus it is used in a limited fashion. By default, when any available register is written to a certain memory location, ProRace bookmarks the value for a later access during the replay in the program map and treats the location as *available*. However, when ProRace hits a system call or an unavailable instruction, it conservatively invalidates emulated memory

```
0: mov    %rax,0x18(%rsp)
1: movslq 0x0(%rbp,%rbx,4),%rdx
2: mov    (%r15,%rbx,8),%rsi
3: mov    0x8(%rsi),%rax
4: mov    %r10,%rdi
5: mov    0x8(%r14),%rax
6: add    %rax,%r13
7: xor    %eax,%eax
8: mov    %r13,0x8(%r14)
9: mov    0x18(%rsp),%rcx
10: mov    (%r15,%r12,8), %rsi
```

Figure 5: Example for Forward and Backward Replay

states. Moreover, the memory emulation might lead to incorrect memory address reconstruction after the racy access (i.e., conflicting write) from other threads. To address this problem, when a race is detected on the emulated memory location in a later phase, PRORACE invalidates the memory location and regenerates the trace from that racy point (i.e., conflicting read) with the unavailable register value. Thus, PRORACE is safe as it never uses racy memory location during the trace regeneration.

While the forward replay progresses further, more registers become unavailable by the load instructions reading from unavailable memory locations. Thus, at some point, PRORACE may end up with a situation where no register is available. One might think that the forward replay cannot proceed anymore because no more instruction can be executed due to the lack of available operands. However, continuing the replay even across the point where all registers become unavailable can capture some unsampled memory accesses that would otherwise be impossible to reconstruct. For example, if memory instructions leverage PC-relative instructions, e.g., `mov offset(%rip)` in x86-64, PRORACE can figure out the memory location by adding the offset to `%rip` which is always available as an instruction pointer (PC). By taking advantage of the full control-flow trace recorded by PT, PRORACE performs the forward replay across basic block boundaries until it reaches the very next PEBS-sampled instruction.

Figure 5 shows how PRORACE reconstructs many unsampled memory accesses using forward replay with a real-world example extracted from *Apache*. Suppose PRORACE sampled the `mov` at a line 0 and recorded the register states at the sample time. After restoring all the register values, PRORACE performs the forward replay for the following instructions. Here, the forward replay can successfully reconstruct the memory addresses of the instructions at line 1, 2, 5, 8, 9 and 10 since their registers used for the address calculation are all available.

However, the memory address of the instruction at line 3, i.e., `mov 0x8(%rsi),%rax`, cannot be reconstructed because `%rsi` reads from memory location that is currently unavailable by the instruction at line 2, i.e., `mov (%r15,%rbx,8),%rsi`. To solve this problem, PRORACE performs the backward replay right after the forward replay.

5.2 Backward Replay

Forward replay cannot reconstruct the address of memory operations if the register operand of memory instructions is unavailable, or if the address is not obtained by PC-relative addressing. This motivates PRORACE to leverage two forms of backward replay to reconstruct the memory addresses skipped by the forward replay: backward propagation and reverse execution.

5.2.1 Backward Propagation

The key observations is that many of unavailable registers can be recovered by consulting the next PEBS-provided execution contexts where all the register values are available. More precisely, the backward replay can reconstruct the memory access whose register operand became unavailable during the forward replay, provided the register has not been updated before the next PEBS-sampled instruction. Fortunately, according to empirical results, the registers used for memory address calculation often have a long live-range [41] after they become unavailable during the forward replay.

In light of this, PRORACE back-propagates all the register values restored at the very next PEBS sample to the instructions where each register has been most recently updated. For this purpose, the forward replay marks such instructions checkpointing the register file at the time the register is updated. In addition, the forward replay keeps track of the youngest one among the instructions as an entry point of the later backward replay. Once all the register back-propagation is performed, PRORACE simply jumps to the youngest instruction and resumes the re-execution there. In a sense, the backward replay can be considered as yet-another forward replay starting from a different location, i.e., the youngest instruction, not the current PEBS-sampled instruction.

Figure 5 also shows how the backward replay reconstructs an unsampled memory access that the forward replay cannot deal with. Suppose PRORACE sampled the instruction at line 10. This allows PRORACE's backward analysis to restore the value of `%rsi`, which is not possible for the forward replay to deal with. In this way, PRORACE can successfully reconstruct the memory address of the instruction at line 3 using the restored register.

5.2.2 Reverse Execution

The second type of PRORACE's backward replay is based on *reverse execution* [4, 8, 35]. In its simplest form such as register-to-register copy, the reverse execution can restore both register values based on the equality as long as at least one of them is known. It is also possible to restore the register used as an operand of arithmetic instructions provided the other operand (register) is known during the

	Thread	Workload
apache	14	ApacheBench. 100K requests, 8 clients, 128KB file size
cherokee	38	ApacheBench. 100K request, 8 clients, 128KB file size
mysql	20	SysBench. 10K requests, 32 clients, 10 million records
memcached	5	YCSB. 200K requests, all ABCDE workload
transmission	4	1.48GB file
pfscan	4	6.8GB file
pbzip2	4	1GB file
aget	4	2.1GB file

Table 1: Evaluation Setup

backward replay. For example, the reverse execution can restore the `%rdx` operand of an instruction (`%rax = %rdx + $offset`), if the other operand (`%rax`) is already available by subtracting the `$offset` from `%rax`. PRORACE's backward replay engine currently supports reverse execution of integer arithmetic instructions such as additions and subtractions.

Note that once an unavailable register is restored by the reverse execution, PRORACE can restore others that have a dependence on that register. As PT provides the program path, PRORACE only needs to track the data dependencies, and triggers forward and backward replays iteratively until they reach the fixed point [41] where no further restoration is found. This simple yet effective technique allows the backward replay to go backward further possibly reconstructing more unsampled memory accesses.

6. Implementation

The online tools for PRORACE consists of two parts: kernel-level PMU drivers and user-land *perf* tool. The new PEBS driver is implemented based on the Linux kernel version 4.5.0. The four PT hardware filter is added to collect branch traces only from the regions of interest.

The offline tool is comprised of four parts: 1) the dynamic standard C library (*glibc* version 2.21) to intercept synchronization and memory allocation operations; 2) the modified *perf* tool to decode raw PT data; 3) the forward-and-backward replay engine that reconstructs memory traces, implemented using Intel's PIN [39] dynamic binary instrumentation tool; and 4) the FastTrack-based data race detector.

The PMU drivers and perf tool includes 4579 lines of C and assembly codes. The offline tools contain 7024 lines of C/C++ code, 793 lines of perl code, 105 lines of python code, and 623 lines of bash code. The implementation of PRORACE can be downloaded from `https://github.com/lzto/ProRace`.

7. Evaluation

This section evaluates PRORACE's runtime overhead, trace size, data race detection effectiveness, memory reconstruction ratio, and offline analysis overhead.

7.1 Methodology

We ran experiments on a 4.0GHz quad-core Intel Core^{TM}i7-6700K (Skylake) processor, with 16GB of RAM, running Gentoo Linux Kernel 4.5.0. PRORACE was evaluated using (1) PARSEC benchmark suite; and (2) seven real-world applications including *apache* web server, *mysql* database server, *cherokee* web server, *pbzip2* parallel compressor, *pfscan* parallel file scanner, *transmission* BitTorrent client, and *aget* parallel web downloader. We use *simlarge* input for all the applications in the PARSEC suite and set the thread number to be four (equal to the number of cores). The evaluation setup for the real-world applications is listed in Table 1. All network and database applications were tested using the local area network which has a gigabit connection.

For data race detection analysis, PRORACE was evaluated using 12 data race examples in real-world applications from previous study [60]. The 12 cases include three data races in *apache*, three races in *mysql*, two races in *cherokee*, two races in *pbzip2*, one race in *pfscan*, and the last one in *aget*. Some other cases in [60] are excluded because they do not include a data race, or are not well documented.

7.2 Performance Overhead

Figure 6 shows the performance overhead of PRORACE for PARSEC benchmarks, with the varying PEBS sampling period from 10 to 100K. As expected, a small sampling period results in more samples, leading to high overhead. The geometric mean of performance overhead over all 13 applications in the PARSEC suite goes up from 4%, 7%, 31%, 2.85x, to 7.52x for the decreasing sampling period of 100K, 10K, 1K, 100, and 10, respectively. There are four applications *bodytrack*, *canneal*, *dedup*, *streamcluster* that incurs small 5-9% runtime overhead for the sampling period of 1K. Setting the sampling period to 10K makes 12/13 applications' overhead less than 10%. The user of PRORACE can perform similar sensitivity analysis to find the lowest sampling period, given a performance overhead budget. Assuming the 10% budget, our experiment shows that the sampling period should be set between 1K and 10K for such CPU-intensive applications.

Figure 7 shows the performance overhead of PRORACE for real world applications, with the varying PEBS sampling period from 10 to 100K. Some applications including *mysql*, *transmission*, *pfscan*, *pbzip2* showed a similar trend of high overhead for a small sampling period. However, the other applications shows negligible ($<1\%$) overhead even with the very small sampling period of 10. The applications belonging to this second category are indeed network I/O dominant applications (with not much file I/O). The runtime over-

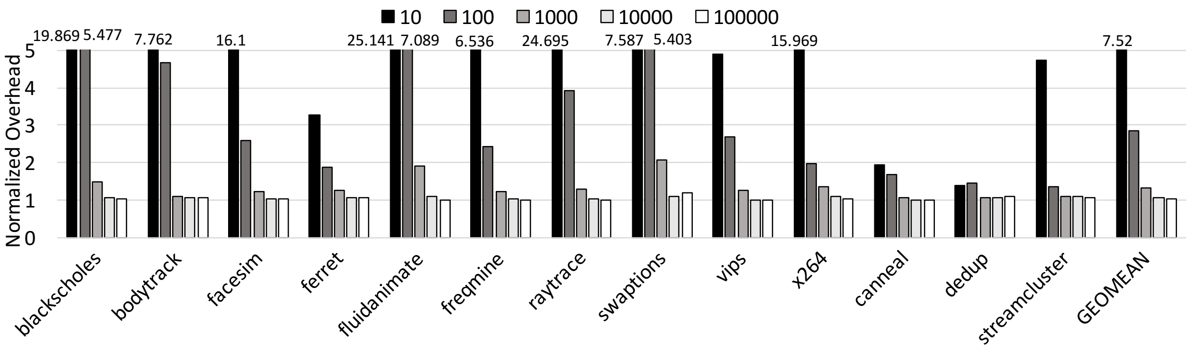

Figure 6: Performance overhead for PARSEC benchmarks

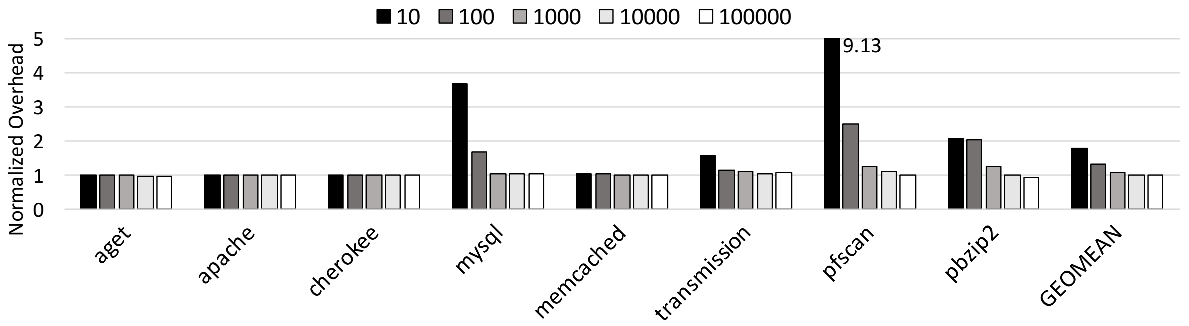

Figure 7: Performance overhead for real applications

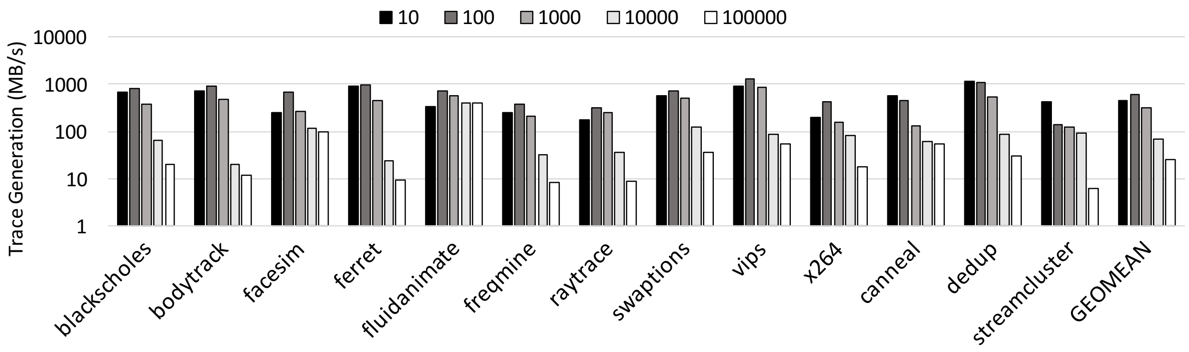

Figure 8: Space overhead for PARSEC benchmarks

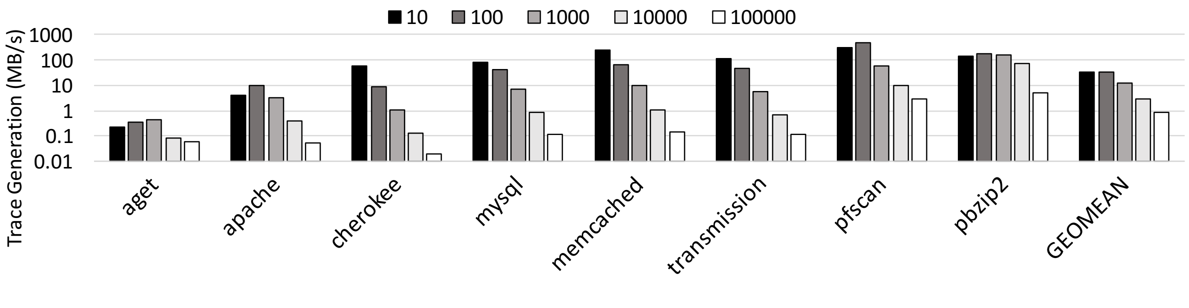

Figure 9: Space overhead for real applications

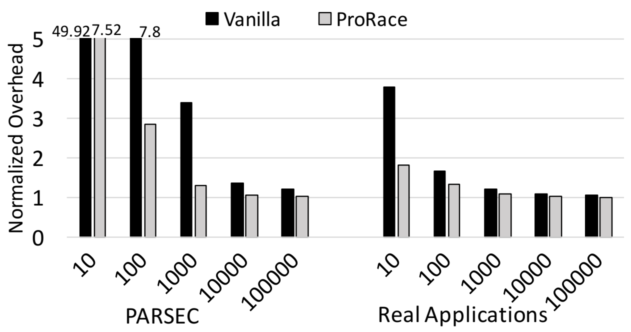

Figure 10: Performance overhead comparison

head of PRORACE can be hidden by network I/O. However, PRORACE apparently cannot hide its overhead with file I/O well because it has to perform many writes to a file during tracing. On geometric average, the runtime overhead goes up from 0.8%, 2.6%, 8%, 34%, to 80% for the decreasing sampling period of 100K, 10K, 1K, 100, and 10, respectively. Assuming the 10% budget, our experiment shows that the sampling period may be set to smaller than 1K (even 10) for real (I/O-bound) applications.

The next study focuses on evaluating the efficiency of PRORACE's new PEBS driver over the vanilla Linux driver. Figure 10 shows side-by-side runtime overhead comparing for each sampling period from 10 to 100K. For clarity, the figure only presents the geometric mean of PARSEC and real applications, respectively. As can be seen in the figure, PRORACE's new driver outperforms the vanilla Linux driver. For an extreme case of the period of 10, the vanilla driver incurs 50x slowdown, but PRORACE shows 7.5x slowdown for PARSEC benchmarks. As another data point, with relatively large period of 100K, the vanilla driver incurs 20%, whereas the PRORACE reports only 4% slowdown for the same benchmarks.

The overhead of RaceZ can be estimated to be around the same because it depends on the stock Linux driver. RaceZ also reports similar performance figures: 2.8% for the sampling period of 200K and 30% for 20K. The experimental result shows that PRORACE has much less overhead than RaceZ. For example with the period 1K, RaceZ results in a 3.4x slowdown, whereas PRORACE only incurs 31% overhead for the PARSEC suite.

As the last experiment for performance evaluation, we study a breakdown of runtime overhead among PEBS overhead, PT overhead, synchronization tracing overhead. We find that the PT overhead is very small contributing only 3% slowdown at most, respectively. The results show the benefits of PT's hardware supports for trace compression and memory range based filtering. Similarly, the synchronization tracing overhead also has a very small impact on performance (<1%). Finally, the PEBS overhead dominates the overall PRORACE performance ranging from 97% to 99%. The result makes sense because PEBS events (memory operations) are much more frequent than PT records (branches), and PEBS events require rich information collection such as register states.

7.3 Trace Size

PRORACE uses PEBS and PT to collect memory access samples and control-flow information at runtime. Figure 8 shows the trace size generated per a second during program execution of PARSEC benchmarks, with the varying PEBS sampling period from 10 to 100K. The PT trace size remains constant across different PEBS configurations, and its size is measured before decompression. As PT records are highly compressed by hardware, the PEBS trace dominates the overall trace size (∼99%). As expected, a small sampling period results in more samples, leading to large trace size. Note that the y-axis is logarithmic. On geometric average, the trace size per second (in MB/s) goes up from 26, 69, 321, 597, to 463 for the decreasing sampling period of 100K, 10K, 1K, 100, and 10, respectively. One outlier is that the trace size for the sampling period of 10 turns out to be less than that of 100 (though it incurs higher overhead as shown in the above experiment). Further investigations show that with a very low sampling period, though the hardware may sample more, these samples may be dropped if the kernel finds that too much time has been spent on the interrupt handling. This implies that there is no benefit of setting the sampling period smaller than a certain (application-specific) threshold.

Figure 9 shows the trace size per second (in MB/s) for real-world applications, with the varying PEBS sampling period from 10 to 100K. The result shows the similar trend but much less space overhead compared to PARSEC benchmarks. On geometric average, the trace size per second (in MB/s) goes up from 0.2, 1.2, 7.9, 40.8, to 99.5 for the decreasing sampling period of 100K, 10K, 1K, 100, and 10, respectively.

7.4 Race Detection

To evaluate the PRORACE's effectiveness in data race detection, we used 12 real-world data race bugs [60]. For each race bug, we fed a buggy input as documented in the previous study [60], and did not control the thread schedules. We collected 100 traces for each PEBS sampling period: 100, 1K and 10K; and counted how many times PRORACE can report the data race among the 100 traces. In effect, the resulting number can be regarded as an approximate detection probability. For comparison, we also measured the number of data races detected by RaceZ. Note that RaceZ enables memory trace reconstruction within one basic block, and for backward replay, it only supports a trivial form of backward propagation within that single basic block. On the other hand, PRORACE includes PT-based full forward-and-backward replay across basic blocks; and supports backward propagation and reverse execution based backward replay.

	Bug manifestation	Access Type	RaceZ			ProRace		
			Period:100	Period:1000	Period:10000	Period:100	Period:1000	Period:10000
apache-21287	double free	memory indirect	6	0	0	50	3	0
apache-25520	corrupted log	register indirect	14	3	0	57	52	15
apache-45605	assertion	register indirect	0	0	0	60	11	1
mysql-3596	crash	memory indirect	0	0	0	5	1	0
mysql-644	crash	memory indirect	20	1	0	21	6	1
mysql-791	missing output	memory indirect	12	0	0	59	2	0
cherokee-0.9.2	corrupted log	register indirect	43	11	2	63	29	8
cherokee-bug1	corrupted log	register indirect	7	3	0	57	19	5
pbzip2-0.9.4-crash	crash	memory indirect	0	0	0	0	0	0
pbzip2-0.9.4-benign	-	pc relative	2	0	0	100	100	100
pfscan	infinite loop	pc relative	0	0	0	100	100	100
aget-bug2	wrong record in log	pc relative	0	0	0	100	100	100
		(average)	8.7	1.5	0.2	56	35.3	27.5

Table 2: Data Race Detection

Table 2 shows the summary of PRORACE's data race detection effectiveness. The first column corresponds to the application name and its bug-tracking number, if exists, while the second refers to how the bug manifests during a program execution. The third column describes its characteristics that we analyzed manually. The next six columns show the number of traces where RaceZ and PRORACE detect data races out of 100 traces (i.e., representing the detection probability) for each sampling period of 100, 1K and 10K, respectively.

It is important to note that PRORACE does detect a data race. As expected, in general, the detection probability increases as the sampling period decreases. On the other hand, some race bugs in *pbzip2-0.9.4*, *pfscan*, and *aget-bug2* are detected every time (100%). Examining the results, we see that the address of the racy variable uses PC-relative addressing in the program. Thus, reproducing the address of such racy memory accesses is easy because the `%rip` register is always available as PC, i.e., an instruction pointer. Here, to detect such race bugs, PRORACE only needs to know what basic blocks contain the racy memory accesses, which is obtained by PT's control-flow trace, without understanding the PEBS-provided execution contexts.

As can be seen, for a given sampling period, PRORACE detects many more data races than RaceZ. For example, PRORACE improves the detection probability from 0.2% to 27.5% on average (arithmetic mean) for the sampling period of 10K, which only incurs 2.6% runtime overhead (Figure 7). For the low sampling period of 100, PRORACE can detect almost all cases (11/12), but RaceZ misses many. It also turns out that RaceZ cannot effectively detect races on simple PC-relative addressing cases because RaceZ requires sampling at the exact basic block containing the racy access. Overall, the results show that PRORACE's PT-guided forward-and-backward replays are very helpful in detecting data races.

7.5 Memory Operation Reconstruction

PRORACE leverages the forward and backward replays to reconstruct unsampled memory operations. RaceZ also tries to recover other memory accesses, but its scope is limited to one basic block that the sampled instruction belongs to. This section shows the benefit of using PRORACE's forward and backward replays in terms of the memory reconstruction ratio.

Figure 11 shows the memory instruction recovery ratio (i.e., the number of recovered and sampled memory operations normalized to the number of original PEBS-sampled instructions) for the six buggy applications with the sampling period of 10K.

The first left-most bar shows how many more memory operations can get reconstructed within one basic block (equivalent to RaceZ's approach). The results show that the basic-block granularity recovery scheme can reconstructs only 1.3x-11.9x memory operations, with the average (arithmetic mean) of 5.4x ratio. Upon further investigation, we found out that *apache*, which shows a good 9.53x recovery ratio, has a lot of simple memory instructions that use PC-relative addressing in a basic block. However, that was not the case for other applications like *mysql*, which shows only a 1.6x recovery ratio.

The next two bars show the benefit of forward replay only and forward+backward replays in PRORACE. On average, the forward replay recovers 134x more memory accesses compared to the baseline (PEBS samples). The backward replay provides additional benefits, and when the backward replay is combined with the forward replay, they achieve a higher recovery ratio of 164x on average. The results shows that PRORACE's race detection coverage (which is approximately proportional to the number of recovered and sampled memory operations) is more than 30 times better than RaceZ's limited basic-block level reconstruction.

7.6 Offline Analysis Overhead

Lastly, Figure 12 shows the offline analysis overhead when traced with the sampling period of 10K. The results shows that to analyze one second of program execution, offline analysis takes 54.5 seconds for *apache* and 35.3 seconds for *mysql*. *Pfscan* shows the worst analysis overhead as it generates a very large trace for a short amount of program execution time.

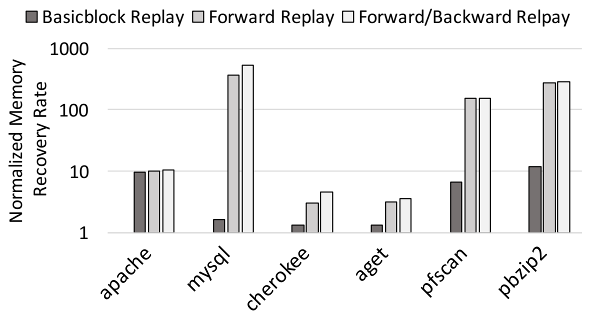

Figure 11: Memory Recovery Ratio

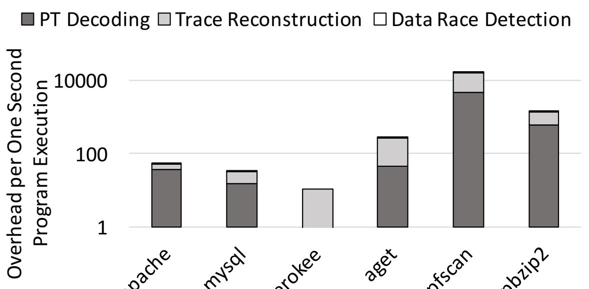

Figure 12: Offline analysis overhead

We also present the breakdown of offline analysis overhead, including 1) PT trace decoding; 2) memory trace reconstruction; and 3) data race detection, each of which takes 33.7%, 64.7%, and 1.6% of the total offline analysis cost, respectively (note the logarithmic y-axis). Note that we conducted this experiment using a single machine. However, the PT trace decoding and trace reconstruction parts, which contributes >98% of the total cost, can be easily parallelized. PT records are independent each other, and the forward-and-backward replay can be also performed region by region, making it suitable for using multiple analysis machines. Moreover, the result includes the overhead of PIN-based dynamic binary instrumentation. The same features can be implemented using static instrumentation tools [30, 62] for better performance.

8. Related Work

The commodity data race detectors such as FastTrack [14] and ThreadSanitizer [54] do not only incur high runtime overhead but also require instrumenting original program. Thus, they would be more appropriate for early testing phase as long as there are many effective test cases. When a program gets mature and used in production, PRORACE would be more useful as it minimally perturbs the program execution thereby observing the real execution characteristics which lead to data races. This is particularly important because data races are highly sensitive to execution environments [2, 63], that testing cannot completely cover, as with other bugs [25, 26, 34]. Section 2 discusses the limitation of other sampling-based dynamic race detectors that cannot be used for production runs.

Several strategies other than sampling have been explored to reduce the overhead of dynamic data race detection. Overlap-based data race detectors [2, 11, 38] focus on detecting races only when racy instructions or code regions overlap at runtime. Wester et al. [57] parallelizes data race detection. Frost [55] compares multiple replicas of the program running in different schedules. Greathouse et al. [16] monitor cache miss events using PEBS and uses them to turn on race detection in an on-demand manner. TxRace [63] uses hardware transactional memory support. Custom hardware [10, 42, 46, 47, 64] has been proposed as well.

The idea of using separate low-cost tracing and high-cost (offline) analysis has been used for program runtime monitoring [5, 7], especially in deterministic replay domain [1, 17, 31, 32, 43]. To the best of our knowledge, there is no software-based record-and-replay solution that achieves (configurable) low overhead equivalent to PRORACE. In addition to input logging, record-and-replay solutions typically require checkpointing support (even fork-based solution) to monitor a long-lived execution. The lowest-overhead solution would be recording only synchronizations and program-input as in RecPlay [49], which guarantees detecting the first race. With additional offline analysis, ProRace can provide higher detection capability beyond the first race. In the context of deterministic replay, PRORACE addresses an important but unanswered question of how much program states can be reconstructed when a PEBS trace is used to start a replay from; and when a PT trace is used to guide the replay. Moreover, PRORACE does not require program input logging.

There are a large body of works that leverage PMU to reduce the runtime overhead of program monitoring for various purposes. For debugging, Gist [28] uses Intel's PT to track the program execution paths for root cause diagnosis of failures, while CCI [23] uses Intel's Last Branch Record (LBR) to collect the branch trace and the return values for cooperative concurrency bug isolation. For security, FlowGuard [36] uses Intel's PT to achieve transparent and efficient Control Flow Integrity (CFI), while CFIMon [58] uses Intel's Branch Trace Store (BTS) for the same goal. For performance, Brainy [27] leverages Intel's PEBS to understand the effect of the underlying hardware for effective selection of data structures, while Jung et al. [24, 33] use the PEBS to characterize the cache behavior of OpenMP [9] program for dynamic parallelism adaptation.

9. Conclusion

PRORACE presents a novel PMU sampling-based data race detector that can be deployed in production settings. Its new kernel driver, that eliminates unnecessary copying and data processing, significantly lowers the runtime overhead of using PEBS to sample memory accesses. Furthermore, PRORACE introduces a novel technique to reconstruct unsampled memory operations with the PT-guided forward and backward replays, thereby enhancing the data race detection coverage. The experimental results highlight PRORACE's high data race detection capability using the 12 real-world data race bugs.

10. Acknowledgments

The authors would like to thank the anonymous referees for their valuable comments. This work was in part supported by the National Science Foundation under the grant CCF-1527463 and Google Faculty Research Awards.

References

[1] Gautam Altekar and Ion Stoica. Odr: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 193–206. ACM, 2009.

[2] Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. Efficient, software-only data race exceptions. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '15, 2015.

[3] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. Pacer: Proportional detection of data races. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 255–268, 2010.

[4] Christopher D. Carothers, Kalyan S. Perumalla, and Richard M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Trans. Model. Comput. Simul.*, 9(3):224–253, July 1999.

[5] Shimin Chen, Babak Falsafi, Phillip B. Gibbons, Michael Kozuch, Todd C. Mowry, Radu Teodorescu, Anastassia Ailamaki, Limor Fix, Gregory R. Ganger, Bin Lin, and Steven W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 63–65, New York, NY, USA, 2006. ACM.

[6] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 258–269, 2002.

[7] Jim Chow, Tal Garfinkel, and Peter M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[8] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P. Kemerlis. Retracer: Triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 820–831, New York, NY, USA, 2016. ACM.

[9] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computer Science and Engineering*, 5(1):46–55, 1998.

[10] Joseph Devietti, Benjamin P. Wood, Karin Strauss, Luis Ceze, Dan Grossman, and Shaz Qadeer. Radish: Always-on sound and complete ra detection in software and hardware. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 201–212, 2012.

[11] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. Ifrit: Interference-free regions for dynamic data-race detection. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 467–484, 2012.

[12] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: A race and transaction-aware java runtime. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 245–255, 2007.

[13] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *In Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI '10, 2010.

[14] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, 2009.

[15] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A study of the internal and external effects of concurrency bugs. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2010, Chicago, IL, USA, June 28 - July 1 2010*, pages 221–230, 2010.

[16] Joseph L. Greathouse, Zhiqiang Ma, Matthew I. Frank, Ramesh Peri, and Todd Austin. Demand-driven software race detection using hardware performance counters. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 165–176, 2011.

[17] Jeff Huang, Charles Zhang, and Julian Dolby. Clap: Recording local executions to reproduce concurrency failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 141–152, 2013.

[18] Intel. Intel 64 and ia-32 architectures software developers manual, 2013. http://download.intel.com/products/processor/manual/325462.pdf.

[19] Intel. Intel inspector xe, 2015. http://software.intel.com/en-us/intel-inspector-xe.

[20] Intel Corporation. *Intel®Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide*, 2010.

[21] Intel Corporation. *6th Generation Intel®Processor Datasheet for S-Platforms*, 2015.

[22] Intel Corporation, Santa Clara, CA. *Intel®64 and IA-32 Architectures Software Developers' Manual*, 2016.

[23] Guoliang Jin, Aditya V. Thakur, Ben Liblit, and Shan Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *OOPSLA*, 2010.

[24] C. Jung, D. Lim, J. Lee, and S. Han. Adaptive execution techniques for SMT multiprocessor architectures. pages 236–246, 2005.

[25] Changhee Jung. *Effective techniques for understanding and improving data structure usage*. Ph.D. Dissertation, Georgia Institute of Technology, Atlanta, GA, 2013.

[26] Changhee Jung, Sangho Lee, Easwaran Raman, and Santosh Pande. Automated memory leak detection for production use. In *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[27] Changhee Jung, Silvius Rus, Brian P. Railing, Nathan Clark, and Santosh Pande. Brainy: effective selection of data structures. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 86–97, New York, NY, USA, 2011. ACM.

[28] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 344–360, New York, NY, USA, 2015. ACM.

[29] Baris Kasikci, Cristian Zamfir, and George Candea. Racemob: Crowdsourced data race detection. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 406–422, 2013.

[30] Michael A. Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. Pebil: Efficient static binary instrumentation for linux. In *International Symposium on the Performance Analysis of Systems and Software*, 2010.

[31] Dongyoon Lee, Mahmoud Said, Satish Narayanasamy, and Zijiang Yang. Offline symbolic analysis to infer total store order. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 357–358, 2011.

[32] Dongyoon Lee, Mahmoud Said, Satish Narayanasamy, Zijiang Yang, and Cristiano Pereira. Offline symbolic analysis for multi-processor execution replay. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 564–575, 2009.

[33] Jaejin Lee, Jung-Ho Park, Honggyu Kim, Changhee Jung, Daeseob Lim, and SangYong Han. Adaptive execution techniques of parallel programs for multiprocessors. *J. Parallel Distrib. Comput.*, 70(5):467–480, May 2010.

[34] Sangho Lee, Changhee Jung, and Santosh Pande. Detecting memory leaks through introspective dynamic behavior modelling using machine learning. In *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[35] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2016.

[36] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. Transparent and efficient cfi enforcement with intel processor trace. In *Proceedings of the 2017 IEEE 23rd International Symposium on High Performance Computer Architecture*, HPCA '17, 2017.

[37] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, 2008.

[38] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 210–221, 2010.

[39] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, 2005.

[40] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. Literace: Effective sampling for lightweight data-race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 134–143, 2009.

[41] S.S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[42] Abdullah Muzahid, Dario Suárez, Shanxiang Qi, and Josep Torrellas. Sigrace: Signature-based data race detection. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 337–348, 2009.

[43] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H Lee, and Shan Lu. Pres: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 177–192. ACM, 2009.

[44] PCWorld. Nasdaq's facebook glitch came from race conditions, May 2012. http://www.pcworld.com/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html.

[45] Eli Pozniansky and Assaf Schuster. Multirace: Efficient on-the-fly data race detection in multithreaded c++ programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):327–340, March 2007.

[46] Milos Prvulovic. Cord: Cost-effective (and nearly overhead-free) order-recording and data race detection. In *Proceedings of the 2006 IEEE 12th International Symposium on High Performance Computer Architecture*, HPCA '06, 2006.

[47] Milos Prvulovic and Josep Torrellas. 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, pages 110–121, 2003.

[48] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, 30(4).

[49] Michiel Ronsse and Koen De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, May 1999.

[50] Paul Sack, Brian E. Bliss, Zhiqiang Ma, Paul Petersen, and Josep Torrellas. Accurate and efficient filtering for the intel thread checker race detector. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 34–41, 2006.

[51] SecurityFocus. Software bug contributed to blackout, Feb. 2004. http://www.securityfocus.com/news/8016.

[52] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, 2009.

[53] Tianwei Sheng, Neil Vachharajani, Stephane Eranian, Robert Hundt, Wenguang Chen, and Weimin Zheng. Racez: A lightweight and non-invasive race detection tool for production applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 401–410, 2011.

[54] The Clang Team. Clang 3.8 threadsanitizer, 2015. http://clang.llvm.org/docs/ThreadSanitizer.html.

[55] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 15–26, 2011.

[56] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: Static race detection on millions of lines of code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 205–214, 2007.

[57] Benjamin Wester, David Devecsery, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Parallelizing data race detection. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 27–38, 2013.

[58] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[59] Junfeng Yang, Ang Cui, Sal Stolfo, and Simha Sethumadhavan. Concurrency attacks. In *The 4th USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, 2012. USENIX.

[60] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 325–336, 2009.

[61] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 221–234, 2005.

[62] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar. A platform for secure static binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 129–140, New York, NY, USA, 2014. ACM.

[63] Tong Zhang, Dongyoon Lee, and Changhee Jung. Txrace: Efficient data race detection using commodity hardware transactional memory. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 159–173, 2016.

[64] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. Hard: Hardware-assisted lockset-based race detection. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 121–132, 2007.