Production-Run Software Failure Diagnosis via Hardware Performance Counters

Joy Arulraj Po-Chun Chang Guoliang Jin Shan Lu
University of Wisconsin–Madison
{joy,pchang9,aliang,shanlu}@cs.wisc.edu

Abstract

Sequential and concurrency bugs are widespread in deployed software. They cause severe failures and huge financial loss during production runs. Tools that diagnose production-run failures with low overhead are needed. The state-of-the-art diagnosis techniques use software instrumentation to sample program properties at run time and use off-line statistical analysis to identify properties most correlated with failures. Although promising, these techniques suffer from high run-time overhead, which is sometimes over 100%, for concurrency-bug failure diagnosis and hence are not suitable for production-run usage.

We present PBI, a system that uses existing hardware performance counters to diagnose production-run failures caused by sequential and concurrency bugs with low overhead. PBI is designed based on several key observations. First, a few widely supported performance counter events can reflect a wide variety of common software bugs and can be monitored by hardware with almost no overhead. Second, the counter overflow interrupt supported by existing hardware and operating systems provides a natural and effective mechanism to conduct event sampling at user level. Third, the noise and non-determinism in interrupt delivery complements well with statistical processing.

We evaluate PBI using 13 real-world concurrency and sequential bugs from representative open-source server, client, and utility programs, and 10 bugs from a widely used software-testing benchmark. Quantitatively, PBI can effectively diagnose failures caused by these bugs with a small overhead that is never higher than 10 %. Qualitatively, PBI does not require any change to software and presents a novel use of existing hardware performance counters.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging; B.8.1 [*Performance and Reliability*]: Reliability, Testing, and Fault-Tolerance; D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Reliability, Languages, Measurement

Keywords failure diagnosis, production run, performance counters, concurrency bugs

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.

1. Introduction

1.1 Motivation

Software bugs are a major cause of production-run system failures. Software companies spend over 30% of development resources in testing [34], but still cannot eliminate all bugs from released software [15]. Things are getting worse in the multi-core era. The increasingly popular multi-threaded software has a huge state space that makes software testing even more difficult. Many bugs, including not only sequential bugs but also concurrency bugs, escape into the field, causing severe failures and huge financial loss [29, 35]. Diagnosing these failures is time consuming, because non-deterministic concurrency bugs are difficult to repeat and multi-threaded execution is difficult to reason about. Previous studies show that concurrency-bug failures often take developers weeks to months to diagnose [10, 24] and the released patches are 4 times more likely to be incorrect than those for sequential bugs [41]. To improve the quality of production-run software, effective failure-diagnosis tools are needed.

There are two major challenges for *production-run failure-diagnosis* tools: diagnostic capability and diagnostic performance.

Diagnostic capability Tools with good diagnostic capability should be able to accurately explain failures caused by a wide variety of bugs. Previous research [16, 21, 22, 25, 28] has found an effective approach for achieving good diagnostic capability. This approach collects program predicates, such as whether a branch is taken, during both success and failure runs of the program. It then uses statistical processing to identify predicates that are highly correlated with failure runs. The only concerns of this approach are designing predicates that can reflect common failure root causes and avoiding large run-time overhead in collecting predicates.

Diagnostic performance Tools with good diagnostic performance should incur small run-time *overhead* that is suitable for deployment. Although progress has been made, we are still far from addressing this challenge.

One potential approach is to use hardware support. Many techniques have been proposed that try to leverage hardware support to speed up bug detection [4, 23, 25, 31, 32, 40, 42, 45, 46]. Unfortunately, these solutions require hardware support that is not available in existing machines.

Another promising approach is *sampling* [16, 21, 22]. By randomly collecting a small portion of predicates in each run, the overhead can be significantly decreased. Unfortunately, this sampling approach does not work as well for diagnosing concurrency-bug failures as it does for sequential-bug failures. Diagnosing concurrency-bug failures requires interleaving-related predicates, such as whether two consecutive accesses to one variable come from the same thread. Collecting these predicates requires coordinating multiple threads and serializing global data-structure updates, which often incurs

more than 100% of overhead even with very sparse sampling [16]. In addition, sampling can hurt the evaluation accuracy of interleaving-related predicates (see details in Section 4.3).

Furthermore, existing software-based monitoring and sampling techniques rely on code instrumentation [16, 21, 28]. This approach requires changes to program source code or binary code. It significantly increases the size of executables (by 11–123X in our experiments) and burdens code caches [38].

1.2 Contribution

This paper presents PBI, a system that leverages existing hardware performance counters to effectively diagnose production-run failures caused by a wide variety of sequential and concurrency bugs with negligible overhead. PBI does *not* rely on code instrumentation and makes no changes to the program source code or binary code.

Performance counters are special registers supported by hardware to monitor hardware performance events. Although widely used for performance tuning and architectural-design verification [7, 30], they have only had limited use in automated software-reliability tools for several reasons:

First, **limited information**. Only a limited and fixed set of performance-counter events can be monitored. For most of these events, hardware only provides the occurrence-count information. Interrupt-based counter access provides information about the instruction that triggered a counter overflow/increment. Unfortunately, interrupt-based access poses extra challenges as follows.

Second, **expensive access**. Although maintaining event counts by hardware has almost no overhead, accessing a counter by user-level software through interrupts does incur overhead.

Third, **noise and randomness**. The information about which instruction triggered a counter overflow is inevitably imprecise, containing noise due to out-of-order execution and imprecise interrupt delivery. Further, it is difficult to command interrupts to be raised at specific program locations. The instructions that raise counter interrupts in a program are quite random.

As a result, no previous work has used hardware performance counters for automated failure diagnosis. Some recent work uses performance counters in an inspiring yet limited way, either to provide hints [11] or to provide part of the information needed [36], for detecting a specific type of bugs (i.e., data races).

In this paper, we address the above challenges and design PBI based on several observations:

First, although only limited hardware events are supported, a couple of widely supported events can already reflect the occurrence of a wide variety of sequential bugs (e.g., branch-related events) and concurrency bugs (e.g., L1 cache coherence events). They can be used as program predicates to help PBI achieve good diagnostic capability (see details in Section 3).

Second, although every single counter access is relatively expensive, interrupt-based performance-counter access is a natural fit for predicate sampling. Specifically, the predicate evaluation is handled by hardware with high accuracy and almost no overhead; the sampling overhead is oblivious to predicate types (e.g., concurrency-bug predicates vs. sequential-bug predicates) or program structure (e.g., branch intensity); the sampling overhead can be controlled easily and drops to almost 0% during sparse sampling. This allows PBI to achieve much better diagnostic performance than instrumentation-based sampling (see details in Section 4).

Third, the noise in the performance-event profile can be filtered by statistical processing. The randomness in interrupt-based counter access is exactly needed for sampling-based failure diagnosis (see details in Sections 4 and 5).

Fourth, as PBI does **not** require code instrumentation, we can monitor programs independent of the programming language in which they are developed, with no increase in the code size.

Overall, this work has made several contributions:

- A novel way of using existing hardware performance counters that leverages their unique characteristics.
- A failure-diagnosis system that handles a wide variety of failures while requiring no changes to software and incurs low overhead that is suitable for production-run deployment.
- An evaluation of 13 real-world concurrency and sequential bugs from open-source server/client/utility programs, as well as 10 bugs from a widely used software-testing benchmark. The evaluation shows that PBI can effectively diagnose failures caused by these bugs, including several concurrency-bug failures that cannot be effectively diagnosed by previous work [16]. PBI failure diagnosis incurs only a small run-time overhead that is never higher than 10% for all applications, much faster than traditional concurrency-bug failure diagnosis techniques. The evaluation also reveals challenges and opportunities in using performance counters.

2. Background

2.1 Hardware Performance Counters

Counters and counter events A hardware *performance counter* is a register dedicated to monitoring *performance events*, such as cache misses and taken branches. Modern architectures usually support a small number of performance-counter registers and a relatively large number of performance events. Each register can be configured to monitor any supported performance event at run time.

The numbers of performance counters and supported performance events have both increased in recent years. Among Intel processors, Pentium Pro has only two counters supporting 98 events in 68 categories, while Nehalem (Core i7) has four counters per core supporting 325 events in 97 categories [19]. Some events such as retired instruction count are supported by almost all architectures; some events like load/store latency are only supported by a few architectures.

Accessing performance counters To monitor a performance event, the user first configures a counter register for that event. The event count can then be obtained by polling the counter periodically or during the handling of counter-overflow interrupt.

Performance counters can be maintained and accessed in different modes, including per-thread mode, per-core mode, and system-wide mode with operating system support. The OS can save performance-counter values for a thread during a context switch. This way, we can easily obtain per-thread statistics when multiple threads are sharing the same set of hardware counter registers.

In PBI, we access hardware performance counters through PERF [17], a profiling utility available in Linux kernel 2.6.31 and later versions. PERF provides a simple command-line interface. When a program is launched under PERF to monitor a specific event, PERF handles the performance counter-overflow interrupt and generates a log file that contains information related to every counter overflow, such as the program counter of the instruction associated with counter overflow and the event being counted.

2.2 Cooperative Bug Isolation

PBI shares a common program-sampling and statistical-processing philosophy with CBI [16, 21, 22], a state-of-the-art production-run failure-diagnosis framework. We present an overview of CBI below.

CBI performs source-to-source transformation of C programs and randomly samples the values of *predicates* at particular program points, called *instrumentation sites* or *sites*, across many success and failure runs of the software. It then uses statistical techniques to identify predicates that correlate with failures, which are referred to as *failure predictors*. The key components of CBI are:

L1 Data Cache LOAD (STORE)	Event Code : 0x40 (41)
Unit mask	Description
0x01	L1 data cache access in I state
0x02	L1 data cache access in S state
0x04	L1 data cache access in E state
0x08	L1 data cache access in M state
perf record –event=r0140:u <program monitored>	

Figure 1. L1 data cache load and store events in Intel Nehalem along with a PERF command that monitors user-level (specified by ":u") load events in I state

Predicate design and evaluation Predicates are carefully designed to reflect common root causes of software bugs. To diagnose sequential-bug failures, CBI monitors three types of predicates. *Branch predicates* evaluate whether a branch is taken; *Return predicates* evaluate whether a function return value is zero, negative, or positive; *Scalar-pair predicates* evaluate whether a variable is smaller than, larger than, or equal to other in-scope variables.

CBI can hardly diagnose concurrency-bug failures. To address this problem, an extension of CBI called CCI [16] is proposed. CCI monitors three types of predicates related to thread interleaving. *CCI-Prev* predicates evaluate whether two successive accesses to a given location were by two distinct threads; *CCI-Havoc* predicates evaluate whether the value of a given location changes between two consecutive accesses by one thread; *CCI-FunRe* predicates evaluate whether two threads execute a function F in parallel.

Previous work has demonstrated that CBI and CCI predicates can effectively diagnose a wide variety of software failures. Among CCI predicates, CCI-FunRe has significantly worse diagnosis capability than the other two. Therefore, we will restrict the comparison of PBI to only CCI-Prev and CCI-Havoc predicates.

Predicate sampling To lower performance overhead, CBI uses an instrumentation-based sampling technique that randomly collects predicates from a small number of sites instead of all sites during every run. CCI has extended this scheme to support interleaving-related predicates. This extension sometimes incurs high overhead, causing more than 100% of overhead in some memory-intensive programs even with very sparse sampling [16].

At the end of each run, CBI generates a profile. The profile includes information for every predicate at all program sites indicating whether the predicate was ever observed in that particular run, and if observed, whether it was true or false. CBI assumes that the runs will be classified as success or failure runs by an automated tool or by the users themselves. PBI continues to use this assumption.

Statistical processing After collecting predicate samples from many success and failure runs, CBI framework uses a statistical model to measure each predicate's correlation with software failures. The correlation is quantitatively evaluated using two scores: *frequency* and *Increase*. *Frequency* measures the frequency of a predicate being observed true in a failure run. *Increase* measures how much more likely it is for a program to fail when a predicate is observed as *true* than when it is simply observed. The statistical model in CBI finally ranks the predicates based on the harmonic mean of *frequency* and *increase* scores. The top ranked predicates are reported as good failure predictors.

In the following sections, we will describe how PBI uses hardware performance counters to diagnose a wide variety of production-run software failures and provide a detailed comparison of PBI with CBI and CCI.

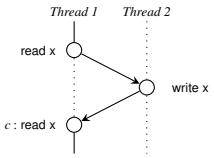
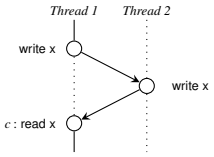
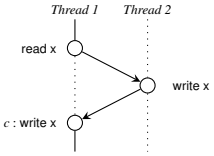
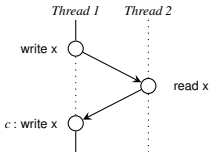
ID	Failure-Inducing Interleaving	Value of Related Predicates	
		Failure run	Success run
RWR		$P_I(c)$=TRUE	$P_I(c)$=FALSE
WWR		$P_I(c)$=TRUE	$P_I(c)$=FALSE
RWW		$P_I(c)$=TRUE	$P_I(c)$=FALSE
WRW		$P_S(c)$=TRUE	$P_S(c)$=FALSE

Table 1. Diagnostic potential of MESI predicates for different atomicity violations ("⟶" shows execution order. "x" is a shared memory location. In success runs, the two accesses to x from Thread 1 are *not* interleaved by the depicted access to x from Thread 2).

3. PBI Predicate Design

3.1 Design for Concurrency-bug Failures

3.1.1 MESI predicate design

Previous work has proposed extending cache-coherence protocols and L1 caches to help detect concurrency bugs [23, 31, 32]. Inspired by these studies, PBI utilizes cache-coherence related performance counter events to diagnose concurrency-bug failures.

Cache-coherence events are supported by performance counters in many modern microarchitectures, such as Intel Core i7 [19] and IBM Power 7 [1]. PBI uses eight cache-coherence events shown in Figure 1. When a hardware performance counter is configured with event code 0x40 (or 0x41) and unit mask 0x01, the counter will count every cache load (or store) that observes an Invalid cache-coherence state at its corresponding cache line just *before* the access is performed. We will refer to this performance-counter event as *I Event*. Similarly, we will refer to performance-counter events that count cache accesses that observe Modified, Exclusive, and Shared states, as *M Event*, *E Event*, and *S Event* respectively.

PBI defines 4 MESI-predicates to diagnose concurrency-bug failures: an *M*-predicate, an *E*-predicate, an *S*-predicate and an *I*-predicate. Every memory-access instruction i in the application is a *site* for MESI-predicates. The *I*-predicate at i is denoted as $P_I(i)$. $P_I(i)$ is true when I Event is observed to occur at i at run time. The other MESI-predicates are similarly denoted and defined.

3.1.2 Diagnostic Potential of MESI predicates

Previous study [24] of real-world concurrency bugs has shown that atomicity violations and order violations are the two most

```
// Thread 1                                  // Thread 2
printf ("End at %f", Gend); //B1             // Gend is uninitialized
...                                          // until here
printf ("Take %f", Gend−init); //B2          Gend = time(); //A
```

Figure 2. An order violation in FFT. Thread 2 should initialize Gend before thread 1 accesses it.

```
//Master Thread                              //Slave Thread
while(...){                                  while(...){
  pos = my_pos; //W_good                       pos+=4; //W_bad
  ...                                          ...
}//many iterations                             log_read(pos); //R
                                             }// many iterations
```

Figure 3. A real-world bug in MySQL that is neither an atomicity violation nor an order violation [44]: during each iteration, *R* should read the value of pos defined by the Master thread (success run), instead of its own thread (failure run).

common types of concurrency bugs, with the former contributing to almost 70% of non-deadlock concurrency bugs. We now assess the potential of MESI-predicates for diagnosing atomicity-violation failures, order-violation failures, and other concurrency-bug failures.

Atomicity violation Atomicity violation happens when a sequence of memory accesses in one thread is unserializably interleaved by memory accesses from another thread(s). Earlier studies [23, 24, 39] show that the majority of atomicity violations only involve one variable and can be categorized into 4 types as listed in Table 1.

We observe that the *I*-predicate and *S*-predicate can differentiate failure runs from success runs for all 4 types of atomicity violations. Therefore, we expect good diagnostic potential for failures caused by atomicity violations.

Order violation Order violation happens when the expected order between two groups of memory accesses from two threads are flipped. Order violation bugs are usually accompanied by abnormal values of MESI-predicates.

A real-world order violation in FFT is illustrated in Figure 2. Instruction A from thread 2 is expected to initialize the global variable Gend before B1 and B2 access Gend from thread 1. Unfortunately, without proper synchronization, A could execute after B1 and B2, causing wrong outputs. Failures caused by this bug can be diagnosed by the MESI-predicates at the site B2. Specifically, during success runs, B2 would observe a Shared state before its access of Gend. During most failure runs, with A executed after B1 and B2, B2 will observe an Exclusive state. During rare failure runs, when A executes between B1 and B2, B2 will observe an Invalid state. Clearly, the MESI-predicates can differentiate failure runs from success runs.

However, there is no guarantee that all order-violation failures can be diagnosed by MESI-predicates. In the example shown in Figure 2, if the program did not contain B2, MESI-predicates would fail to diagnose the failure. In that made-up scenario, A and B1 are the only two instructions involved in failures. They would both observe Invalid states during both success and failure runs.

Other concurrency bugs Recent bug-detection work [44] and previous characteristics study have shown that some real-world concurrency bugs are caused by neither atomicity violation nor order violation, such as the bug shown in Figure 3. MESI-predicates are still effective for this rare bug as the *M*-predicate at site R will be a good failure predictor.

To summarize, we expect MESI-predicates to diagnose a wide variety of concurrency-bug failures by leveraging the difference in cache-coherence states observed between success and failure runs.

3.1.3 Predicate Evaluation

Evaluating MESI-predicates involves two steps: (1) run-time monitoring using PERF, and (2) off-line log analysis.

We configure PERF to collect user-level MESI-related performance counter events when the program is executed as shown in Figure 1. At run time, PERF handles every counter-overflow interrupt and logs down information like the instruction that caused the counter overflow and the performance event that was counted.

The log analysis is straightforward. While processing the log obtained from each run, we treat every instruction *i* that appears in the log as a *site*. We evaluate $P_e(i)$ to be true if *i* appears in the log with event type *e* (*e* is *M*, *E*, *S*, or *I*). Otherwise, we evaluate $P_e(i)$ to be false.

Ideally, we want to monitor all 8 MESI events in each run. In practice, the number of performance events that can be accurately monitored at the same time depends on hardware support. As most existing machines contain 1 – 8 hardware performance-counter registers in each core, only one event can be accurately monitored at a time in the worst case.

To ensure that PBI runs correctly on a wide range of machines, our current implementation monitors only one performance-counter event at a time. In each run, we randomly select one out of the 8 *MESI* events listed in Figure 1 to be monitored by PERF, and use the trace-analysis algorithm discussed above. Note that the failure-diagnosis accuracy will not be affected by this implementation decision, because each event is given an equal chance to be monitored in the long term. Our implementation can also be easily extended to collect more events in each run given hardware/OS support on some existing or future platforms.

3.1.4 Statistical Processing of MESI Predicates

PBI uses the statistical model developed in CBI to identify predicates that are highly correlated with failures. As discussed in Section 2.2, this statistical model assigns *frequency* and *increase* scores to each predicate. It then uses the harmonic mean of these two scores to rank the predicates. The top ranked predicates are considered as good failure predictors.

We compute these scores during the off-line analysis of PERF logs. The formulas used to calculate *frequency* and *increase* scores of a predicate $P_e(i)$ are provided in eqs. (1) and (2). $|expr|$ denotes the number of runs that satisfy the condition *expr*. *F* denotes the condition that a run has failed; $P_e(i)$ denotes the condition that a run has evaluated $P_e(i)$ to be true (*e* could be *M*, *E*, *S*, or *I* here); *i* denotes the condition that a run has observed site *i*.

$$\text{Frequency} = \frac{|\mathrm{P_e(i)}|}{|\mathrm{F}|} \tag{1}$$

$$\text{Increase} = \frac{|\mathrm{F\&P_e(i)}|}{|\mathrm{P_e(i)}|} - \frac{|\mathrm{F\&i}|}{|\mathrm{i}|} \tag{2}$$

3.1.5 Comparison with Software-based Predicates

CCI-Prev and CCI-Havoc are two predicates used by CCI to diagnose concurrency-bug failures. They treat every instruction *i* that accesses a heap/global variable as a *site*. When *i* is executed by a thread *t* to access a memory location *m*, CCI-Prev evaluates whether the last access to *m* is from thread *t* or not, and CCI-Havoc evaluates whether the value of *m* has changed between *i* and the last access of *m* in *t*. We briefly compare the diagnostic potential and overhead of these two predicates with PBI-MESI predicates.

//Thread 1	//Thread 2	//Thread 1	//Thread 2

tmp=Gcnt;
...
Gcnt=tmp+1;//s
	printf("%d",Gcnt);	tmp=Gcnt;
		...
		Gcnt=tmp+1;//s
			tmp=Gcnt
			...
			Gcnt=tmp−1;

(a) A success run ($P_{\text{CCIPrev}}(s)$=true,$P_I(s)$=false)		**(b)** A failure run ($P_{\text{CCIPrev}}(s)$=true,$P_I(s)$=true)	

Figure 4. Difference in diagnosing atomicity violations between PBI-MESI and CCI-Prev

Diagnostic Potential CCI-Havoc can diagnose three out of four types of atomicity violation failures listed in Table 1. It cannot diagnose order-violation failures. When two writes from a thread are interleaved by a read from another thread (WRW violation), the failure-inducing interleaving does not cause any value change in the variable involved, and hence cannot be diagnosed by CCI-Havoc.

Similar to PBI-MESI, CCI-Prev can diagnose atomicity violations and some order violations. A key difference between CCI-Prev and PBI-MESI is that CCI-Prev does not differentiate read accesses from write accesses. For instance, the predicates of CCI-Prev cannot distinguish success runs from failure runs in the bug shown in Figure 4, while PBI-MESI can. This prevents CCI-Prev from diagnosing some atomicity-violation failures [16].

Diagnostic overhead To evaluate CCI-Prev and CCI-Havoc predicates, CCI maintains a table that records memory-access information for *every* heap/global variable touched by the program. The overhead is especially large for CCI-Prev where all threads share one table that records which thread has issued the most recent access to a variable. Synchronizing accesses to this table from different threads incurs large overhead.

In contrast, hardware completely takes care of monitoring MESI events for PBI. We do not need to maintain an auxiliary data structure like the memory-access table in CCI. Therefore, PBI can scale well to large multi-threaded software with large heap/global memory footprint and many heap/global accesses.

The main overhead of PBI-MESI is due to the counter-overflow interrupt. If we want to evaluate MESI predicates for *every* memory access, the interrupt overhead would be large.

Code-size overhead CCI needs to insert a predicate-evaluation function, which queries and updates the memory-access table, before every instruction that accesses heap/global variables. In contrast, PBI does not change the program at all.

3.2 Design for Sequential-bug Failures

3.2.1 Predicate Design

State-of-the-art failure-diagnosis systems [22, 33] have found that branch-related predicates are effective in diagnosing sequential-bug failures. Inspired by previous work, PBI uses branch-related performance counter events to diagnose sequential bugs.

Branch-related events are widely supported in modern microarchitectures [19]. PBI uses the "branches retired" events shown in Figure 5. When a counter is configured with event code 0xC4 and unit mask 0x20, it counts every retired conditional branch that is *taken*. We will denote this performance-counter event as *BT Event*. We apply the mask 0x10 to count the retired branches that are *not taken*, and denote that event as *BNT Event*.

PBI defines BT and BNT predicates to diagnose sequential-bug failures. Every branch instruction b is a *site* of BT and BNT predicates, denoted as $P_{BT}(b)$ and $P_{BNT}(b)$. $P_{BT}(b)$ is true when BT Event is observed to occur at b at run time. Similarly, $P_{BNT}(b)$ is true when BNT Event is observed to occur at b at run time.

BR instructions retired	Code: 0xC4
Unit mask	Description
0x20	Retired branches that are taken
0x10	Retired branches that are not-taken
perf record –event=r20*C*4:u <program monitored>	
perf record –event=r10*C*4:u <program monitored>	

Figure 5. Branch-retired events in Intel Nehalem along with PERF commands that monitor user-level branch taken and non-taken events

3.2.2 Diagnostic Potential of BT Predicate

Previous work [22, 33] has shown that branch-related predicates are effective in diagnosing sequential-bug failures, because many sequential bugs are associated with abnormal control flows. For example, many semantic bugs are related to corner cases and wrong control flows [20]; many buffer-overflow bugs occur under incorrect boundary-condition checking; many dangling-pointer and double-free bugs occur due to incorrect conditions for memory deallocation. Overall, we expect PBI-BT and PBI-BNT predicates to possess good diagnostic capability for sequential-bug failures.

3.2.3 Predicate Evaluation and Statistical Processing

Similar to PBI-MESI predicates, evaluation of PBI-BT and PBI-BNT predicates are done using PERF. For each run, we configure PERF to monitor branch taken or branch not-taken event that occurs at user level using the commands shown in Figure 5. We process the log generated by PERF for each run, and evaluate $P_e(i)$ to be true if and only if i occurs in the log with event e (e is BT or BNT here).

The statistical model used to identify sequential-bug failure predictors is the same one used to identify concurrency-bug failure predictors, as described in Section 3.1.4.

3.2.4 Comparison with Software-based Predicates

Techniques designed to diagnose sequential-bug failures are much more mature than those designed for diagnosing concurrency-bug failures. We briefly compare the diagnostic potential and run-time overhead between CBI [21, 22] and PBI.

In general, traditional software-based techniques have better diagnostic potential for sequential-bug failures than PBI. The flexibility of software instrumentation allows tools like CBI to collect not only branch-related predicates, but also *Scalar* and *Return* predicates, as discussed in Section 2. In contrast, it is very difficult for PBI to support these two types of predicates that are not related to any performance event. Of course, PBI still has good diagnostic potential, as branch taken or not-taken information is among the most effective predicates for diagnosing sequential-bug failures [33].

To evaluate branch-related predicates, CBI instruments every branch. This overhead is not small for branch-intensive programs. However, the evaluation of predicates designed for diagnosing sequential-bug failures is much more lightweight than that of concurrency-related predicates.

In PBI, branch events are monitored by hardware with almost no overhead. However, handling counter-overflow interrupts incurs overhead. Modern microarchitectures provide several alternatives to monitor branch events. Special hardware registers like Last Branch Record [18] could be utilized to further improve the performance of PBI in future.

4. Predicate Sampling

4.1 PBI Sampling Scheme

Sampling is widely used to amortize the overhead of run-time monitoring [3, 16, 21, 22, 26]. It is traditionally achieved through software instrumentation. PBI also uses sampling to improve performance. Sampling in PBI is performed through performance counter-overflow interrupts rather than software instrumentation.

Two parameters in PERF affect the sampling rate of PBI. One is PERF's kernel parameter *perf_event_max_sampling_rate*. This parameter specifies the maximum number of samples allowed during a time slice before context switch. It is used to avoid too frequent overflow interrupts. PBI currently uses the default PERF setting of this parameter, 10^5. The other is PERF's command-line parameter "-c". When "$-c$ *C*" is specified, PERF configures the underlying performance counter to trigger an interrupt approximately once every C occurrences of event instructions. In our evaluation, PBI changes this command-line parameter to adjust the sampling rate: larger value of *C* corresponds to sparser sampling. Note that when *C* is small, as we will see in Section 7, the effective sampling rate is much sparser than one out of C due to the throttling effect of *perf_event_max_sampling_rate*.

4.2 Diagnostic Potential of PBI Sampling

A poorly designed sampling scheme can hurt diagnostic potential. For instance, a naïve scheme that deterministically collects the first K predicates in every execution cannot diagnose failures whose root causes are in the later part of the execution, no matter how well the predicates are designed. To support effective failure diagnosis, every site should have a fair chance to be sampled. For example, CBI framework uses uniformly random sampling, where each site independently has an equal probability of being sampled.

At first glance, PBI sampling may not appear to be random enough for effective failure diagnosis. With a "-c C" configuration, in a deterministic computer system, PERF will be expected to collect a deterministic sequence of predicates for a program under a given input: the C-th predicate, the 2C-th predicate, and so on.

Interestingly, PBI sampling has inherent randomness in practice benefiting from non-determinism in modern hardware and operating systems, and the throttling effect of *perf_event_max_sampling_rate*. We evaluated this randomness using a microbenchmark as discussed below. The effectiveness of PBI in diagnosing different types of failures in real-world applications (Section 7) also reflects the randomness of PBI sampling to some extent. We could also introduce more randomness in PBI sampling, by inserting a random set of dummy memory-access/branch instructions at the beginning of a program. However, based on our experience, there is no need for extra randomness in PBI sampling.

Evaluating the randomness of* PBI *sampling We designed a microbenchmark to evaluate the randomness of sampling used in PBI. The microbenchmark contains 10^5 static branches. It is designed in a way that every branch will be executed and taken exactly once at run time. We configure PERF to sample the branch-taken event with a configuration of $-c$ 100.

The microbenchmark is executed 10^5 times under the same configuration. If PERF deterministically collects samples every 100 instructions, as suggested by its command-line option, we would see a *wide* variance in the sampling frequency — some branches are sampled with 100% frequency and others are sampled with 0% frequency. Furthermore, we would see only 1% of the branches in the microbenchmark get sampled after 10^5 runs. The real results obtained from PERF contradict with these and address several important questions.

Is the sampling frequency uniform across sites? We observe in Figure 6 that the sampling frequency at different sites actually exhibits low variance in practice. Therefore, PBI sampling provides every program site with a fair chance for getting sampled.

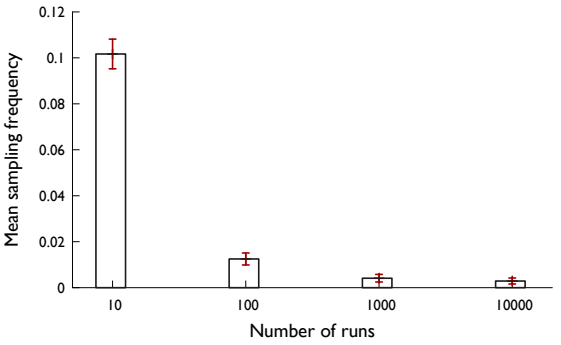


Figure 6. Variation of sampling frequency among all program sites. The sampling frequency of site *b* after N runs is the number of times a predicate is observed at *b* divided by *N*. The error bars represent standard deviation; shorter bars indicate more uniform sampling.

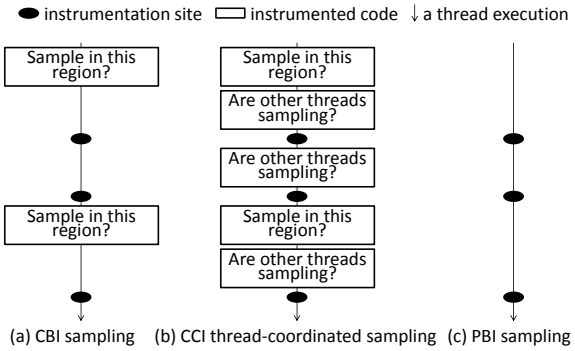


Figure 7. Different sampling schemes

Can PERF *sampling eventually cover all sites?* The results from the microbenchmark study confirm that the answer is yes! With just 2.7% branch coverage in the microbenchmark after 10 runs, the coverage quickly reaches 81.8% after 10^3 runs. We obtain an almost ideal coverage of 97.5% after 10^4 runs.

We can also observe the throttling effect of *perf_event_max_sampling_rate* in Figure 6. With the $-c$ 100 configuration, ideally the sampling rate should be around one out of 100. However, in practice, the observed mean sampling frequency is only about one out of 500.

We also designed microbenchmarks for other performance-counter events. The overall trend is the same as what we presented above for branch-taken event.

The above observations indicate that PERF sampling can support our sampling-based failure diagnosis framework well. We will revisit and confirm these observations in Section 7.

4.3 Comparison with Instrumentation-based Sampling

We compare PBI sampling with traditional instrumentation-based sampling with respect to overhead, accuracy and code-size increase.

Performance overhead Instrumentation-based sampling does not come for free as it needs to frequently check whether a predicate should be evaluated or skipped, as shown in Figure 7. The frequency and overhead of this checking varies across different sampling schemes. CBI performs this type of checking at the beginning

of every acyclic code region. In addition to that, CCI needs an extra checking before every global/heap-memory access for thread-coordinated sampling, which is required by CCI-Prev predicates. CCI also needs bursty sampling [14] to effectively collect CCI-Prev and CCI-Havoc predicates, which adds further overhead.

Due to different sampling mechanisms, the *best-case* performance, which occurs under very sparse sampling, is different between PBI and traditional failure-diagnosis systems. PBI can achieve almost zero best-case overhead, when the sampling rate drops to zero. In contrast, CBI could have as large as 20% best-case overhead, depending on the control-flow structure of a program [21]. CCI could cause over 100% best-case overhead, depending on the memory-access density of a program [16].

PBI incurs substantially lower overhead at the same sampling rate when compared with CBI and CCI as it exploits hardware support. Further, it requires *fewer* runs to isolate failure predictors under the same overhead constraint, as sampling can be done much more intensively.

Predicate-evaluation accuracy Sampling can affect the accuracy of predicate evaluation. In fact, this is a concern for many interleaving-related predicates. For example, when a memory location m is accessed for the first time in a sampling period, we cannot possibly know whether the preceding access to m is from the same thread or not. This dilemma is shared by all CCI predicates.

PBI sampling suffers much less from this problem. Hardware keeps monitoring the specified performance event without any gap, unless a cache-line is evicted while cache-related events are monitored. Normal cache-line eviction cycles are much longer than the sampling period in CCI, which is only 100 instructions by default for CCI performance concerns.

Code-size increase PBI sampling does not require any change to a program. In contrast, a significant amount of instrumentation is needed to support sampling in traditional systems. CBI framework prepares one fast version and one slow version for every acyclic region in a program to support its sampling scheme, which at least doubles the code size.

5. Detailed Issues

This section discusses several potential sources of inaccuracy in PBI failure diagnosis that we have not discussed.

Skid in counter-overflow interrupts Interrupt-based sampling is generally *imprecise*. Specifically, the last instruction retired prior to the service of a performance-event overflow interrupt may not be the instruction that actually triggered the interrupt. This gap between the instruction that actually triggered the interrupt and the instruction whose program counter gets collected is termed as *skid*. According to PERF documentation [17], the amount of skid for a particular event could vary, and can be up to dozens of instructions.

To better understand the impact of skid in failure diagnosis, we designed 3 microbenchmarks, each comprised of two threads. Each thread executes a loop that contains a store followed by a load from a shared variable. In each benchmark, different mixes of assembly instructions that do not access memory are inserted before and after the two shared-variable access instructions in the loop. The microbenchmarks contain 100, 200, and 600 static padding instructions respectively. We execute these benchmarks under PERF to monitor I Event. In theory, the padding instructions could never be associated with cache-access events. However, in practice, they appear in the PERF logs, which allows us to calculate the amount of skid.

After running each benchmark 1000 times, we have several observations. First, skid exists in all three benchmarks.

Second, the *variance* of skid for each instruction is very small. In fact, there is always a dominant amount of skid for an instruction. That is, more than 99% of PERF logs for a microbenchmark have a specific amount of skid. This dominant amount of skid is 8 in the first benchmark, and 1 in the second and third benchmarks. Given the existence of dominant skids, performance events that actually occurred at instruction i would mostly be reported at instruction s, and rarely be reported at other instructions s_1, s_2, etc. If i is a good failure predictor, PBI would still be able to identify s as a good failure predictor and filter out noise generated by s_1, s_2, and others, benefiting from the statistical processing of PBI.

Third, skids are usually not long, mostly within 10 assembly instructions. Therefore, the failure predictor detected by PBI can still guide developers to the actual bug location in the program.

We will revisit and further confirm these observations in our experiments on real world bugs presented in Section 7.

Hardware issues False sharing is a common concern for hardware-based concurrency-bug detection and diagnosis [23, 31, 32]. Since multiple shared variables could be placed in the same cache line, cache coherence states may not indicate how a shared variable is accessed by multiple threads. False sharing could affect the quality of PBI failure diagnosis. In our experiments, it has never affected the topmost failure predictor. More details will be presented in Section 7.3.4.

Thread migration and scheduling can also be sources of inaccuracy for hardware-based techniques. Since a thread's performance-counter values are saved by OS during context switches, they have relatively little impact on PBI. Failure diagnosis could be affected when multiple threads scheduled in the same core are executing simultaneously in SMT mode. In this case, shared-variable access interleavings may not cause any cache coherence traffic. If the multiple threads are scheduled to run one at a time, there would be no impact.

In summary, the accuracy of PBI could be affected by hardware and system issues in theory. In practice, although false sharing and skid effects are observed in our evaluation, we have never observed PBI fail in diagnosing a failure due to these two issues or other issues discussed above, as we will see in Section 7.

6. Methodology

We have evaluated PBI on 13 real-world software failures caused by concurrency and sequential bugs, and 10 failures caused by sequential bugs in a widely used software-testing benchmark print_tokens2[1]. These bugs all come from C/C++ programs, and represent different types of common root causes and failure symptoms, as shown in Table 2. All the experiments were performed on an Intel Core i7 (Nehalem) machine running Linux 3.2 kernel.

The experiments focus on evaluating the two design goals for PBI: diagnostic capability and diagnostic performance.

For applications with concurrency-bug failures, we perform a head-to-head comparison between PBI and the state-of-the-art instrumentation-based failure diagnosis tool CCI [16]. We have evaluated *all* the benchmarks examined in CCI paper (Table 2) under the same environment used for CCI. Our evaluation uses the same bug-triggering inputs for PBI and CCI. Further, we use two MySQL benchmarks that are not used in CCI work to evaluate PBI's capability of supporting C++ programs.

Following the evaluation methodology used in CCI, we added randomly executed thread yield calls in the source code to make

[1] Print_tokens2 is a tokenizer program in Siemens benchmark suite [13]. It includes a correct version and 10 buggy versions. Each buggy version contains a bug injected by benchmark designers to emulate common real-world sequential bugs, especially semantic bugs.

Program	KLOC	Root Cause	Failure Symptom	Runs Total	Runs Failed
Concurrency-Bug Failures					
Apache1	333	A.V.	corrupted log	1000	464
Apache2	333	A.V.	crash	1000	504
Cherokee	83	A.V.	corrupted log	1000	480
FFT	1.3	O.V.	wrong output	1000	416
LU	1.2	O.V.	wrong output	1000	480
Mozilla-JS1	107	A.V.	crash	1000	520
Mozilla-JS2	107	A.V.	wrong output	1000	511
Mozilla-JS3	107	A.V.	crash	1000	467
MySQL1	478	A.V.	crash	1000	576
MySQL2	478	A.V.	wrong output	1000	488
PBZIP2	2.1	A.V.	crash	1000	592
Sequential-Bug Failures					
BC	14.4	mem.	crash	1000	444
CUT	1.2	mem.	crash	1000	505
print_tokens2*	0.5	sem.	wrong output	1000	500

Table 2. General characteristics of buggy applications evaluated. "KLOC": program size in thousands of lines of code; "A.V.": atomicity violations; "O.V.": order violations; "mem.": memory bugs; "sem.": semantic bugs; Mozilla-JS: Mozilla JavaScript engine; *: print_tokens2 contains 10 versions, with 1 bug in each version.

the program fail more frequently. This is applied in the same way to our PBI and CCI experiments. The failure rate for each application is shown in Table 2. These random yields do *not* affect the quality of our evaluation, as they only increase the failure occurrence frequency. In real-world software deployment, the failure-occurrence frequency is usually low for concurrency bugs. In these scenarios, PBI users simply need to collect enough failure runs and combine them with a random subset of successful runs to obtain a mix of program runs similar to the mix we are evaluating.

For applications with sequential-bug failures, we have primarily used the input sets designed by the original developers. The developers of print_tokens2 have designed nearly 2000 inputs for it. For each version of print_tokens2, we executed each of these inputs once, and randomly took PERF logs generated by 500 success runs and 500 failure runs. We used the GNU Coreutils program dd to generate random bug-triggering inputs for BC and used bug-triggering inputs provided in bug reports for CUT. We attempted to compare PBI and CBI with respect to sequential-bug failure diagnosis. Unfortunately, we were unable to set up software libraries required by CBI on our machines due to configuration problems. As a result, we will only present a qualitative comparison between CBI and PBI.

PBI's experiments use the default PERF setting of *perf_event_max_sampling_rate* and a command-line "-c 3" configuration to control the sampling rate. This configuration provides a comparable sampling rate as that of CCI, which by default starts sampling at one out of 10,000 rate and each period lasts for 100 instructions. The impact of different "-c" configurations is evaluated and presented in Section 7.3.

7. Evaluation Results

Overall, PBI has successfully diagnosed all the 11 concurrency-bug failures and 10 out of 12 sequential-bug failures. PBI incurs low overhead in the range of 0.4–8.6% for all applications evaluated. In the following paragraphs, we will discuss about diagnostic capability, run-time overhead, parameter sensitivity, and other results in detail.

```
//Thread 1: start a new log          //Thread 2: writing to log
1.1 log_status = OPEN ;
...                                  2.1 if(log_status!=OPEN)
...                                  2.2 // skip logging
1.2 log_status = CLOSED ;
```

Figure 8. A WRW atomicity-violation bug in MySQL (skipping logging leads to a wrong output failure)

7.1 Diagnostic Capability Results

7.1.1 Concurrency bugs

For all the 11 concurrency-bug failures evaluated, PBI has identified predicates directly related to the failure root cause as its **topmost** failure predictor, as shown in the "diagnostic capability" column in Table 3. PBI identifies I-predicates as top predictors for 8 failures caused by atomicity violations. PBI also identifies E-predicates as top predictors for 2 failures caused by order violations (FFT and LU). Finally, PBI identifies an S-predicate as the top predictor for the MySQL2 failure caused by a WRW atomicity violation.

In contrast, there are 6 failures that CCI-Prev or CCI-Havoc fails to diagnose. This can be attributed to four reasons.

First, the current implementation of CCI cannot handle C++ programs, such as MySQL, because its underlying static analysis infrastructure [27] is designed for C programs. One of the MySQL failures is depicted in Figure 8. PBI successfully identifies the S-predicate on line 1.2 as the top failure predictor. We believe that CCI-Prev can also diagnose this failure, if it could use a C++-friendly static-analysis infrastructure.

Second, CCI-Havoc cannot diagnose failures caused by WRW atomicity violations, such as MySQL2 shown in Figure 8, or order violations, such as FFT and LU. The root cause of FFT and LU failures are similar to the one depicted in Figure 2. Since order violations usually do not cause a variable to change its value between two consecutive accesses in one thread, CCI-Havoc cannot diagnose FFT and LU failures. PBI E-predicates are usually good predictors for order-violation failures, as discussed in Section 3.1.2.

Third, sampling prevents CCI from diagnosing the Mozilla-JS1 failure whose root-cause code regions cannot fit into one sampling period, as discussed in Section 4.3. PBI successfully diagnosed this failure, because hardware has accurately tracked the cache-access history related to this failure.

Fourth, CCI-Prev fails to diagnose the Cherokee failure because it does not differentiate read accesses from write accesses. The Cherokee failure is similar to the one depicted in Figure 4.

In summary, PBI is effective in diagnosing a wide variety of concurrency-bug failures. It is also much more language independent than software instrumentation-based techniques.

7.1.2 Sequential bugs

For 10 out of 12 sequential-bug failures, PBI has successfully identified predicates that are directly related to the failure root causes within the top two failure predictors, as shown in Table 4.

An example of a successfully diagnosed sequential-bug failure is shown in Figure 9. This piece of code should return FALSE on line 6. However, it incorrectly returns TRUE there. As a result, whenever there is white-space character (' ') in the input, print_tokens2 will output an incorrect token in its text-parsing output. The topmost failure predictor reported by PBI is a BT predicate at line 4. As we can see, the predicate can help developers to quickly locate the root cause of the wrong output.

In CUT, the buggy code is enclosed by two nested if-conditions. Clearly, the inner-if is closer to the buggy code than the outer-if. However, in our experiments, the BT predicate at the outer-if is ranked first and the BT predicate at the inner-if is ranked second.

Program	Diagnosis Capability			Skid	Diagnosis Overhead			Avg. # samples in each run		
	PBI	CCI-P	CCI-H		PBI	CCI-P	CCI-H	PBI	CCI-P	CCI-H
Apache1	✓ 1(I)	✓ 1	✓ 1	1	0.4%	1.9%	1.2%	305051	127299	129907
Apache2	✓ 1(I)	✓ 1	✓ 1	0	0.4%	0.4%	0.1%	284936	192196	198902
Cherokee	✓ 1(I)	-	✓ 2	1	0.5%	0.0%	0.0%	29795	15966	5182
FFT*	✓ 1(E)	✓ 1	-	0	1.0%	121%	118%	7615	8875	8869
LU*	✓ 1(E)	✓ 1	-	1	0.8%	285%	119%	9541	51893	23356
Mozilla-JS1	✓ 1(I)	-	✓ 2	1	1.5%	800%	418%	111847	74707	38831
Mozilla-JS2	✓ 1(I)	✓ 1	✓ 1	1	1.2%	432%	229%	37851	77919	37162
Mozilla-JS3	✓ 1(I)	✓ 2	✓ 1	0	0.6%	969%	837%	11439	5501	1290
MySQL1	✓ 1(I)	-	-	0	3.8%	-	-	583597	-	-
MySQL2	✓ 1(S)	-	-	1	1.2%	-	-	299389	-	-
PBZIP2	✓ 1(I)	✓ 1	✓ 1	2	8.4%	1.4%	3.0%	312808	125668	124785

Table 3. Concurrency-bug failure diagnosis. "✓ n" indicates that the n^{th} highest ranked predictor captures the root cause, while "-" indicates that neither of the top two predictors is useful. The E/S/I following "✓ n" indicates which MESI-predicate is the top predictor. *: For fair comparison, we use the same program configuration for CCI and PBI, which is different from that in the CCI work [16] for FFT and LU.

Program	Diagnosis Capability	Skid	Diagnosis Overhead
BC	✓ 1(BT)	1	6.1%
CUT	✓ 2(BT)	2	7.6%
print_tokens2 v1	-	-	8.6%
print_tokens2 v2	-	-	8.6%
print_tokens2 v3	✓ 1(BNT)	1	8.6%
print_tokens2 v4	✓ 1(BT)	2	8.6%
print_tokens2 v5	✓ 1(BT)	1	8.6%
print_tokens2 v6	✓ 1(BT)	2	8.6%
print_tokens2 v7	✓ 1(BT)	3	8.6%
print_tokens2 v8	✓ 1(BNT)	2	8.6%
print_tokens2 v9	✓ 1(BNT)	2	8.6%
print_tokens2 v10	✓ 1(BT)	3	8.6%

Table 4. Sequential-bug failure diagnosis. "✓ n" indicates that the n^{th} highest ranked predictor captures the root cause, while "-" indicates that neither of the top two predictors is useful. The BT/BNT following "✓ n" indicates the predictor type.

```
1 // Print_tokens2 v7
2 if(ch == '\n')
3   return (TRUE);
4 else if(ch == ' ')
5 // Bug: should return FALSE
6   return (TRUE);
7 else
8   return (FALSE);
```

Figure 9. A semantic bug in print_tokens2 v7

Our investigation shows that most test inputs that take the outer-if branch end up also taking the inner-if branch. In other words, once the BT predicate is observed at the inner-if site, it is mostly evaluated to be true. This causes a low *increase* score for the BT predicate at the inner-if site.

PBI fails to diagnose two print_tokens2 failures for different reasons. In v1, the buggy program does not contain any branch that is related to the software failure. In a function that is executed by every input, the program misses both the code that handles a special character in the input and the branch that checks whether the input contains that special character. As a result, BT and BNT predicates cannot help in diagnosing this program's failure at all. In v2, the buggy code is indeed enclosed by a branch. The BT predicate at this branch suffers a low *increase* score for a similar reason as that in CUT discussed above. So, the top ranked failure predictor is an *effect*, rather than the *cause* of the failure. We have also checked whether the above two failures can be diagnosed by CBI. Based on our understanding of CBI, it should be able to diagnose the failure of v2 through scalar predicates.

Summary The above results show that PBI can effectively diagnose a wide variety of concurrency and sequential bugs. Comparing with the state-of-the-art failure-diagnosis system, PBI has better diagnostic capability for the evaluated concurrency-bug failures and comparable capability for sequential-bug failures.

7.2 Diagnostic Performance Results

PBI has incurred only 0.4–8.6% overhead during its failure diagnosis for *all* the benchmarks. This performance is comparable with and in some cases (e.g., in BC) slightly better than CBI for sequential programs[2, 21], and is much better than CCI for multi-threaded programs. In fact, CCI incurs over 100% overhead for 5 out of the 9 concurrency-bug benchmarks it is applied to, as shown in the "diagnosis overhead" columns of Table 3.

To better understand the performance difference between PBI and CCI, we first confirmed that the number of samples collected by PBI and CCI are mostly comparable to each other, as shown in Table 3. We then evaluated the overhead of CCI under *very* sparse sampling mode (i.e., sampling rate is 0). As shown in Table 5, even with *no* predicate collected, the instrumentation-based sampling framework of CCI still incurs around or more than 100% overhead for 5 out of the 9 benchmarks it is applied to. This shows that counter-overflow based PBI sampling has a significant performance advantage over traditional instrumentation-based sampling techniques.

Program	CCI-Prev	CCI-Havoc
Apache1	0.4%	0.5%
Apache2	0.0%	0.0%
Cherokee	0.0%	0.0%
FFT	96%	102%
LU	112%	95%
Mozilla-JS1	608%	371%
Mozilla-JS2	254%	191%
Mozilla-JS3	884%	830%
PBZIP2	0.8%	2.3%

Table 5. Overhead of CCI under *very* sparse sampling mode

PBZIP2 is the only application where PBI has slightly larger overhead than CCI. The reason is that CCI does not instrument

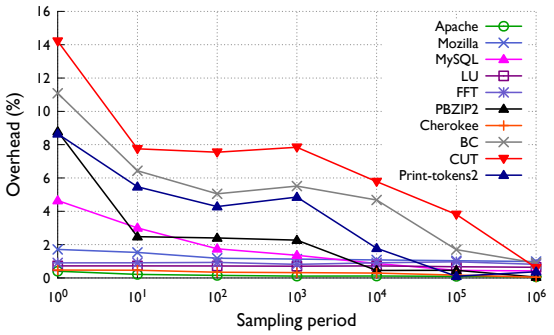


Figure 10. Overhead of PBI under different "-c" configurations

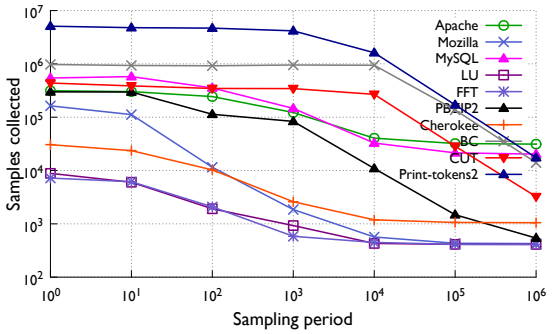


Figure 11. Samples collected by PBI under different "-c" configurations

the memory-intensive BZIP2 compression library used by PBZIP2. PBI monitors *all* user-level event occurrences, including those from library code. This causes extra overhead for PBI. In fact, when CCI is configured to monitor the compression library as PBI does, it incurs over 100% overhead even under very sparse sampling mode.

Unlike the CBI/CCI frameworks, PBI does not incur larger overhead during concurrency-bug failure diagnosis than during sequential-bug failure diagnosis. The reason is that PBI essentially uses the same mechanism to evaluate and sample its concurrency-related predicates and sequential-related predicates.

Note that the overhead of FFT and LU are different from what was originally reported in CCI work [16]. In the original CCI work, accesses to floating-point variables are not instrumented for FFT and LU. For a fair head-to-head comparison, we have reconfigured CCI to monitor floating-point variable accesses, similar to the monitoring done by PBI. In fact, even when CCI does not instrument floating-point instructions, CCI still has much worse performance than PBI. For example, CCI-Prev would still incur around 100% overhead for FFT and LU.

Summary PBI aims to leverage hardware performance counters to achieve low-overhead failure diagnosis that is suitable for production-run deployment. As we can see, PBI has successfully achieved this goal and made significant improvement over previous work.

7.3 Other Detailed Results

7.3.1 Parameter sensitivity

We have evaluated the variation of performance overhead and the number of samples collected by different "-c" configurations.

As we can see in Figure 10, the overhead of PBI reduces substantially as the sampling sparsity increases. In fact, the overhead drops significantly to around 1% for *all* benchmarks under the "-c 10^6" configuration. Of course, diagnostic capability would have to be sacrificed for reduced overhead. As shown in Figure 11, the number of collected samples also significantly drops as the sampling sparsity increases. As a result, many more runs will be needed to collect sufficient samples for PBI failure diagnosis.

Overall, hardware performance counters provide a naturally effective mechanism for PBI users to control the tradeoff between diagnostic performance and diagnostic capability. Furthermore, future work can adjust the sampling sparsity of PBI at run time to achieve more efficient and effective failure diagnosis.

7.3.2 Code size increase

Program	PBI	CCI-Prev	CCI-Havoc
Apache1	1X	65X	40X
Apache2	1X	123X	60X
Cherokee	1X	18X	11X
FFT	1X	64X	37X
LU	1X	69X	39X
Mozilla-JS1	1X	42X	28X
Mozilla-JS2	1X	36X	24X
Mozilla-JS3	1X	43X	29X
PBZIP2	1X	38X	37X

Table 6. Code size with respect to original binary

As shown in Table 6, CCI would increase the binary-code size by 11 to 123 times. Instead, PBI does not change program binary code at all, because it does not rely on code instrumentation for predicate evaluation and sampling.

7.3.3 Skid in real world bugs

Imprecise interrupt handling often causes *skid* between an instruction that triggered the performance-counter overflow and the instruction that is collected by PERF, as discussed in Section 5. Occasionally, the amount of skid could be up to dozens of instructions.

Fortunately, the statistical model used in PBI is capable of pruning noise caused by random skids. As shown in Table 3 and Table 4, all failure predictors have a skid of 0–2 binary instructions, except for two versions of print_tokens2 that have a skid of 3 instructions. These binary-instruction skids do *not* cause any skid at the source-code level. Further, in all cases where skid is observed, the instruction corresponding to the skid is not a memory-access or branch instruction. Therefore, we believe it would be easy for developers to identify the "actual" instruction where the failure predictor is located.

We validated the true location of failure predictor in all benchmarks using the padding-insertion technique described in Section 5.

7.3.4 Effect of false-sharing on PBI failure diagnosis

None of the topmost PBI failure predictors is affected by the cache-line false sharing problem discussed in Section 5. Further checking shows that the 2^{nd} and 3^{rd} ranked failure predictors in FFT and LU are actually due to false sharing. That is, success and failure runs have different patterns in accesses to two/more variables that are located in the same cache line. To some extent, they are also valid failure predictors, as long as the developers are aware of false sharing, which might be challenging. We verified that after changing the source code to avoid false sharing, these failure predictors are pruned out from PBI diagnosis results.

7.4 Limitations of PBI

PBI is not a panacea for production-run failure diagnosis. Although PBI has successfully diagnosed all the concurrency-bug failures in our experiments, it could fail to diagnose some concurrency-bug failures in practice, such as some order-violation failures discussed in Section 3.1.2. Its failure diagnosis could also be affected by hardware constraints, as discussed in Section 5. In addition, PBI is not as flexible as instrumentation-based tools. For example, given the limited types of performance events supported by hardware, PBI cannot support all types of predicates supported by CBI. As another example, instrumentation-based tools can easily skip certain instructions during its program monitoring, such as stack-variable accesses, floating-point instructions, or instructions from library code. It is difficult to achieve the same effect using hardware performance counters alone.

Like the CBI/CCI framework, PBI intentionally collects less information from each run to achieve better run-time performance. As a result, only failures that are observed for multiple times, which is a common pattern in production-run systems, can be diagnosed effectively by PBI. We believe that this tradeoff is worthwhile and suitable for production-run use, as also demonstrated by previous work [3, 16, 21, 22, 26]. Without sampling, program monitoring would incur too large an overhead to be deployed in the field. The production-run usage scenario also allows failures to easily repeat for many times for widely deployed software. In fact, for popular software [9], only failures that have bothered many users (or important users) are processed by developers.

Overall, PBI complements existing techniques and is useful for diagnosing real-world software failures. In practice, users can instruct PBI to collect and analyze MESI, BT, and BNT predicates to diagnose potential failures in single- and multi-threaded software.

8. Related Work

Due to space constraints, this section discusses some closely related work that has not been discussed in the earlier sections.

Using performance counters for software reliability Performance counters are useful for diagnosing system performance bottlenecks [5]. Recently, several interesting techniques have been proposed to help bug detection and software testing using performance counters.

One system [11] uses a performance-counter overflow as a signal to enable and disable a software-based race detector at run time. Specifically, it uses OTHER_CORE_L2_HITM event to count the number of times a cache line is loaded when it is stored in modified state within another core's private L2 cache. The count can imply the existence of conflicting accesses to shared variables.

RACEZ [36] uses Instruction_Retired event to collect samples of executed instructions and related register images. It also instruments the software to collect a trace of executed synchronization operations. During off-line analysis, it uses the the first trace to reconstruct memory-access samples and then combines the two traces to detect data races.

The above two systems have different goals from PBI. They are both designed to detect data races, not to diagnose failures. In fact, they would fail to diagnose many failures, because many concurrency bugs are not caused by data races. They will also incur many false positives in failure diagnosis, because many data races do not cause externally visible failures. Previous work [16] has shown that a race detector would cause 9–81 false positives for the benchmarks evaluated in Section 7. In contrast, PBI can diagnose failures caused by a wide variety of concurrency bugs. In addition, PBI has very few false positives (0 in our evaluation of concurrency-bug failure diagnosis), leveraging its statistical model and failure information. Furthermore, these two systems use performance counters differently from PBI. The first system only uses performance counter outputs as a hint to start traditional race detection; the second system still relies on software instrumentation to collect synchronization information. In contrast, PBI completely relies on performance counters to collect all the needed program-property information for failure diagnosis.

Eunomia [43] uses branch-related events and itlb_misses event to detect certain types of security attacks. It collects instruction and branch execution traces through performance counters. It then checks the trace to discover invalid instructions or infeasible control-flow paths caused by code-injection attacks. This tool does not target general software bugs, and cannot be used to diagnose general software failures.

Other recent works have proposed using branch-related performance counter events to measure code coverage during software testing [37] and using cache-hit/miss performance-counter events to help improve performance [30]. Similar to PBI, these tools all use performance counters to improve traditional techniques that are based on software instrumentation. But, these tools also have completely different goals and hence have very different designs from PBI.

Hardware support for bug detection Researchers have made several proposals [4, 23, 25, 31, 32, 40, 42, 45, 46] to speed up the detection of concurrency bugs and memory bugs by extending hardware. PBI has drawn inspiration from these works. At the same time, PBI is different from them because it uses existing hardware support.

Bug detection Sampling is a widely used technique to lower the overhead of run-time monitoring. Some previous bug-detection tools [3, 26] use software-based sampling techniques to give hot code paths smaller probability to be sampled than cold paths, which helps them detect data races with small overhead. PBI is different from these previous techniques, because its sampling does not require code instrumentation. In addition, as a failure diagnosis tool, PBI has different design goals and strengths from these race-detection tools as discussed above. PBI uses CBI's statistical model to measure each predicate's correlation with software failures. Statistical approaches have also been used before to detect general software bugs during in-house development and testing [6, 8, 12].

9. Conclusion

This paper presents PBI, a production-run failure-diagnosis system. PBI samples hardware performance-counter events at run time and uses statistical processing to discover event instructions closely related to failures. We evaluate PBI using 13 real-world bugs from representative open-source software and 10 bugs from a widely used software-testing benchmark. Our evaluation demonstrates that PBI can effectively diagnose software failures caused by a wide variety of concurrency and sequential bugs, including some failures that cannot be effectively diagnosed by previous work. Benefiting from hardware performance-counter sampling, PBI incurs less than 10% overhead for all the benchmarks, much faster than previous work in concurrency-bug failure diagnosis. PBI does not require any change to the monitored software, operating systems, or hardware. Future work can further improve PBI by exploring more types of performance events and combining instrumentation-based failure-diagnosis techniques with performance-counter based techniques.

Acknowledgments

We thank the anonymous reviewers for their insightful feedback which has substantially improved the content and presentation of this paper. We thank Mark Hill, Ben Liblit and Piramanayagam Nainar for their invaluable comments on PBI. We are grateful to

Stephane Eranian for useful discussions on PERF. We also thank Subhadip Ghosh and Dibakar Gope for their assistance in evaluating skid-related microbenchmarks. This work is supported in part by NSF grants CCF-1018180, CCF-1054616, and CCF-1217582; and a Claire Boothe Luce faculty fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] V. R. I. Alex Mericas, Brad Elkin. Comprehensive pmu event reference - POWER7. https://www.power.org/events/Power7/.

[2] P. Arumuga Nainar. *Applications of Static Analysis and Program Structure in Statistical Debugging*. PhD thesis, University of Wisconsin – Madison, Aug. 2012.

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

[4] L. Ceze, P. Montesinos, C. von Praun, and J. Torrellas. Colorama: Architectural support for data-centric synchronization. In *HPCA*, 2007.

[5] J. Demme and S. Sethumadhavan. Rapid identification of architectural bottlenecks via precise event counting. In *ISCA*, 2011.

[6] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.

[7] S. Eranian. Perfmon2, 2010. http://perfmon2.sourceforge.net.

[8] M. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE*, 2000.

[9] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. C. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *SOSP*, 2009.

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

[11] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and T. M. Austin. Demand-driven software race detection using hardware performance counters. In *ISCA*, 2011.

[12] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, 2002.

[13] M. J. Harrold and G. Rothermel. Siemens Programs, HR Variants. http://www.cc.gatech.edu/aristotle/Tools/subjects/.

[14] M. Hirzel and T. M. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2001.

[15] J. L. Lions et. al. ARIANE 5 Flight 501 Failure – report by the inquiry board. http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html.

[16] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for Cooperative Concurrency Bug Isolation. In *OOPSLA*, 2010.

[17] L. kernel developers. Perf profiling framework, 2012. https://perf.wiki.kernel.org/index.php/Main_Page.

[18] D. Levinthal. Performance analysis guide for intel processors. Intel manual, Feb. 2009. http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

[19] D. Levinthal. Ia-32 architectures software developers manual volume 3b: System programming guide, part 2. Intel manual, June 2009.

[20] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *ASID*, 2006.

[21] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[22] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

[23] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *ASPLOS*, 2006.

[24] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – a comprehensive study of real world concurrency bug characteristics. In *ASPLOS*, 2008.

[25] B. Lucia and L. Ceze. Finding concurrency bugs with context-aware communication graphs. In *MICRO*, 2009.

[26] D. Marino, M. Musuvathi, and S. Narayanasamy. Effective sampling for lightweight data-race detection. In *PLDI*, 2009.

[27] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *International Conference on Compiler Construction*, 2002.

[28] S. Park, R. W. Vuduc, and M. J. Harrold. Falcon: fault localization in concurrent programs. In *ICSE*, 2010.

[29] PCWorld. Nasdaq's Facebook Glitch Came From Race Conditions. http://www.pcworld.com/businesscenter/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html.

[30] A. Pesterev, N. Zeldovich, and R. T. Morris. Locating cache performance bottlenecks using data profiling. In *EuroSys*, 2010.

[31] M. Prvulovic. Cord:cost-effective (and nearly overhead-free) order-reordering and data race detection. In *HPCA*, 2006.

[32] M. Prvulovic and J. Torrellas. Reenact: using thread-level speculation mechanisms to debug data races in multithreaded codes. In *ISCA*, 2003.

[33] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *ICSE*, 2009.

[34] SDTimes. Testers spend too much time testing. http://www.sdtimes.com/SearchResult/31134.

[35] SecurityFocus. Software bug contributed to blackout. http://www.securityfocus.com/news/8016.

[36] T. Sheng, N. Vachharajani, S. Eranian, R. Hundt, W. Chen, and W. Zheng. Racez: a lightweight and non-invasive race detection tool for production applications. In *ICSE*, 2011.

[37] M. L. Soffa, K. R. Walcott, and J. Mars. Exploiting hardware advances for software testing and debugging (nier track). In *ICSE*, 2011.

[38] Uh, Gang-ryung, Cohn, Robert, Ayyagari, and Ravi. Analyzing dynamic binary instrumentation overhead. *WBIA at ASPLOS*, 2006.

[39] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.

[40] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *HPCA*, 2007.

[41] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. N. Bairavasundaram. How do fixes become bugs? In *FSE*, 2011.

[42] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, 2009.

[43] L. Yuan, W. Xing, H. Chen, and B. Zang. Security breaches as pmu deviation: detecting and identifying security attacks using performance counters. In *APSys*, 2011.

[44] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: detecting concurrency bugs through sequential errors. In *ASPLOS*, 2011.

[45] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architecture Support for Software Debugging. In *ISCA*, 2004.

[46] P. Zhou, R. Teodorescu, and Y. Zhou. Hard: Hardware-assisted lockset-based race detection. In *HPCA*, 2007.