Leveraging the Short-Term Memory of Hardware to Diagnose Production-Run Software Failures

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

Abstract

Failures caused by software bugs are widespread in production runs, causing severe losses for end users. Unfortunately, diagnosing production-run failures is challenging. Existing work cannot satisfy privacy, run-time overhead, diagnosis capability, and diagnosis latency requirements all at once.

This paper designs a low overhead, low latency, privacy preserving production-run failure diagnosis system based on two observations. First, short-term memory of program execution is often sufficient for failure diagnosis, as many bugs have short propagation distances. Second, maintaining a short-term memory of execution is much cheaper than maintaining a record of the whole execution. Following these observations, we first identify an existing hardware unit, Last Branch Record (LBR), that records the last few taken branches to help diagnose sequential bugs. We then propose a simple hardware extension, Last Cache-coherence Record (LCR), to record the last few cache accesses with specified coherence states and hence help diagnose concurrency bugs. Finally, we design LBRA and LCRA to automatically locate failure root causes using LBR and LCR.

Our evaluation uses 31 real-world sequential and concurrency bug failures from 18 representative open-source software. The results show that with just 16 record entries, LBR and LCR enable our system to automatically locate failure root causes using LBR and LCR. Our evaluation uses 31 real-world sequential and concurrency bug failures from 18 representative open-source software. The results show that with just 16 record entries, LBR and LCR enable our system to automatically locate failure root causes using LBR and LCR. As our system does not rely on sampling, it also provides good diagnosis latency.

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

Keywords failure diagnosis; production runs; concurrency bugs; hardware performance monitoring unit

1. Introduction
1.1 Motivation

Software bugs are widespread. Although effective bug-detection tools have been proposed, many software bugs inevitably slip into production runs. They have led to many severe production-run failures, causing huge financial loss [8, 17, 30, 50] and threatening people’s lives [21]. Consequently, diagnosing failures that occur on production machines is a critical task.

Unfortunately, diagnosing production-run failures is challenging. Different from in-house bug detection and testing, production-run failure diagnosis has to preserve privacy and minimize run-time overhead, which often leads to sacrifices in diagnosis latency (i.e. how long it takes to diagnose a failure after its first occurrence) or diagnosis capability (i.e., what types of failures can be diagnosed).

Many tools have been proposed for production-run failure diagnosis. Since the occurrence of failures is difficult to predict, previous work either collects program states at the failure site, referred to as failure-site approach, or collects program states throughout the execution, referred to as whole-execution approach, as illustrated in Figure 1. These
two approaches make different tradeoffs among privacy, run-time overhead, diagnosis capability, and diagnosis latency.

The failure-site approach incurs negligible run-time overhead, but has difficulty in satisfying the other requirements. Fundamentally, inferring run-time information using the program states at failure sites is not only tedious but also often impossible. Making things even worse, sending a lot of program states, such as the whole coredump, back to developers could severely hurt end users’ privacy. Some of these problems are alleviated by recent work that uses static analysis to automate log-variable selection and backward inference [43, 45]. However, static analysis is not a panacea. It cannot help when software fails at unexpected locations (e.g., during a segmentation fault). Its help is limited for failures caused by concurrency bugs. Its analysis time also increases with the number of logging sites.

The whole-execution approach can achieve better diagnosis capability than the failure-site approach due to its access to the whole execution information. On the down side, it can easily lead to huge run-time overhead. Many dynamic bug detectors are suitable for in-house testing but not for production-run use, because they either lead to huge overhead or require complicated non-existing hardware support. Recent work [2, 18, 22, 23] uses random sampling to address the overhead problem. However, random sampling leads to long diagnosis latency. For example, with the default 1 out of 100 sampling rate used by previous work [22, 23], a failure often needs to occur for about 100 times before the developers obtain information useful for failure diagnosis. This is especially a concern for software that is not deployed on millions of machines and bugs that manifest infrequently, such as concurrency bugs.

In summary, more tools are needed to support production-run failure diagnosis.

### 1.2 Contribution

This paper presents a new approach to diagnosing a wide variety of production-run software failures with low run-time overhead and low diagnosis latency, while preserving end users’ privacy. This new approach is based on the following two observations:

First, short-term memory is valuable and often sufficient for failure diagnosis. Previous empirical studies [12, 34, 48] have shown that most bugs have short propagation distances and hence have root causes located shortly before failures. Even for bugs with long propagation distances, information useful for failure diagnosis is generally not distributed evenly throughout the execution. Intuitively, information collected closer to a failure is more likely to be useful for diagnosis.

Second, short-term memory can be maintained with extremely low cost. In fact, existing hardware already maintains such short-term memory of software execution through facilities like Last Branch Record (LBR). With only 4–16 record entries maintained by LBR, the hardware cost is low and the run-time overhead is negligible.

In short, maintaining a short-term memory of program execution can achieve a nice balance in the design space:

- Comparing with the failure-site approach, it has access to more run-time information and hence can achieve better diagnosis capability.
- Comparing with the whole-execution approach, it only keeps the most recent execution history and can achieve better performance without sacrificing diagnosis latency.

Following these observations, we propose a new production-run failure diagnosis approach that leverages the short-term memory of program execution maintained by hardware [1]. Several questions need to be answered to design such an approach.

First, what to remember in the short-term memory? A lot of hardware events occur during every machine cycle. We need to select hardware events that are most useful for diagnosing software failures.

Second, how large is short-term memory? Are 4 – 16 record entries, the settings in existing hardware LBR, sufficient for real-world failure diagnosis? Can this short-term memory provide information that cannot be inferred by the program states at the failure site? How often can this short-term memory contain failure root-cause information? These questions have to be answered by thorough evaluation with real-world failures.

Third, how to use the short-term memory? We need to design a software system that accesses this hardware short-term memory and integrates it into an automated failure diagnosis algorithm. The detailed implementation also needs to be careful not to pollute the precious short-term memory with irrelevant events, such as those from library or code used to access the short-term memory.

This paper answers the above questions and makes the following contributions:

- We propose a short-term memory approach to diagnosing production-run failures, with a good balance among privacy, run-time overhead, failure-diagnosis capability, and failure-diagnosis latency (illustrated in Figure [1]).
- We identify and design two hardware short-term memory facilities to support production-run failure diagnosis. Specifically, we identify an existing hardware performance monitoring unit, LBR, to help diagnose sequential-bug failures, and propose a simple hardware extension, Last Cache-coherence Record (LCR), to help diagnose concurrency-bug failures. The details are in Section [4].

1 In this paper, we will refer to the hardware record of recent execution history as short-term memory. This short-term memory is composed of special machine registers, and has nothing to do with the main memory.
We design and implement two ways to use the hardware short-term memory for production-run failure diagnosis. The basic way, referred to as LBRLOG and LCRLOG, is to use LBR and LCR as a generic mechanism to enhance failure logging. It provides developers a straightforward and generic mechanism to obtain the execution history right before a failure, which often contains hints of failure root causes. The advanced way, referred to as LBRA and LCRA, uses a statistical model to automatically locate failure root causes from LBR/LCR records. The details are presented in Section 5.

A thorough evaluation based on 31 real-world failures from 18 open-source applications. Our evaluation based on 6945 failure-logging sites shows that more than 80% of LBR entries contain useful information that cannot be inferred by static control-flow analysis. LBRA can automatically locate branches that are closely related to failure root causes and bug patches for all the 20 evaluated sequential-bug failures, with less than 3% run-time overhead measured on commodity machines. In addition, LCRLOG and LCRA can help locate the root causes for 7 out of 11 tested concurrency-bug failures. Comparing with state-of-the-art systems that rely on sampling [2, 18, 22, 23], our failure-diagnosis system has tens to hundreds of times shorter diagnosis latency.

2. Background

2.1 Hardware branch-tracing facilities

There are two types of branch-tracing facilities in Intel processors. One is called Last Branch Record (LBR), which stores branch records in a circular ring of hardware registers. The other is called Branch Trace Store (BTS), which keeps branch records in cache or DRAM. BTS can store many more records than LBR. However, it incurs much larger overheads that is not suitable for production runs, ranging from 20% to 100% [31]. The following discussion will focus on LBR.

LBR is part of the hardware performance monitoring unit, originally designed for performance profiling. LBR branch recording uses special bus cycles on the system bus [14] and incurs negligible overhead. Its recording can be enabled and disabled through a special machine register, as shown in Table 1. Once enabled, LBR keeps recording newly retired branch instructions, with each new record evicting the oldest record. Each record contains the source and target addresses of a branch instruction. The total number of records in LBR varies in different microarchitectures, following an increasing trend over the years — it goes from 4 entries in Pentium 4 and Intel Xeon processors, to 8 in Pentium M processors, and to 16 in Nehalem processors [15]. All the experiments in this paper are conducted on an Intel Nehalem processor.

LBR can be configured to record different types of branch instructions, including conditional branches, unconditional jumps, calls, returns, and others, as shown in Table 1.

A subtle yet important issue in using LBR is that a conditional branch in source code does not simply map to a conditional branch in machine code. Figure 2 shows a simple example. The conditional branch in Figure 2(a) is translated into one conditional jump instruction on Line 2 and one unconditional jump instruction on Line 5 in Figure 2(b). The former will be taken when the original conditional branch is evaluated false, and the latter will be taken if the original branch is evaluated true.

Previous work proposes inserting harmless unconditional branches along the fall-through edges [40] to make the mapping between machine-code branches and source-code branches easier. We reuse this technique and skip the details.

In general, no matter the true edge or the false edge of a conditional branch in the source code is taken, some corresponding machine-level branch will get recorded in LBR. Developers will be able to locate the source-level branch and know its outcome based on the LBR record.

2.2 Hardware performance counters

Many modern processors equip each core with a few hardware performance-counter registers. These registers can be configured to monitor and count a wide variety of hardware performance events. Different from the branch tracing facil-

---

**Table 1: LBR related machine specific registers in Intel Nehalem (*: the masks used in this work)**

<table>
<thead>
<tr>
<th>IA32_DEBUGCTL</th>
<th>ID: 0x1d9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x801</td>
<td>Enable LBR</td>
</tr>
<tr>
<td>0x00</td>
<td>Disable LBR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LBR_SELECT</th>
<th>ID: 0x1c8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1</td>
<td>*Filter branches occurring in ring 0</td>
</tr>
<tr>
<td>0x2</td>
<td>Filter branches occurring in other levels</td>
</tr>
<tr>
<td>0x4</td>
<td>Filter conditional branches</td>
</tr>
<tr>
<td>0x8</td>
<td>*Filter near relative calls</td>
</tr>
<tr>
<td>0x10</td>
<td>*Filter near indirect calls</td>
</tr>
<tr>
<td>0x20</td>
<td>*Filter near returns</td>
</tr>
<tr>
<td>0x40</td>
<td>*Filter near unconditional indirect jumps</td>
</tr>
<tr>
<td>0x80</td>
<td>Filter near unconditional relative branches</td>
</tr>
<tr>
<td>0x100</td>
<td>*Filter far branches</td>
</tr>
</tbody>
</table>

**Figure 2: Conditional branches in source and machine code.**
Avoid_trashing_input function, which is not on the call stack at the moment of failure. Third, even if developers pay attention to avoid_trashing_input, they do not know which basic blocks have executed, given the complicated control flow, not to mention discover the root cause at A.

In short, to effectively diagnose this failure, developers need to know the execution path leading to the failure. Otherwise, it is difficult to locate the root-cause code region and the root-cause branch. This path information often cannot be inferred by core-dumps, call-stacks, or log variables.

### 3.2 Case 2: a concurrency-bug failure

Figure 4 shows a concurrency bug in Mozilla JavaScript Engine. This bug is caused by unsynchronized accesses of the shared variable st->table. At runtime, the variable is initialized by InitState at t1, and then checked at t2. In most cases, this check will pass, as long as New has succeeded at t1. Occasionally, st->table is set to NULL by another thread at t3 right before the check is conducted. As a result, the software will fail with an “out of memory” message issued by ReportOutOfMemory.

Developers will encounter two major challenges in diagnosing this failure. First, control-flow uncertainties. The failure location in the source code is difficult to identify, because “out of memory” can be emitted by any one of the 55 locations where ReportOutOfMemory is invoked. Second, interleaving uncertainties. Even if the failure location...
is resolved, developers will probably mistakenly attribute the failure to memory-consumption problems in $a_1$, based on the control flow of Investate. Traditional log-enhancing techniques provide little help here: developers cannot easily infer interleavings based on variable values logged at failure.

Interleavings are difficult to diagnose. To successfully diagnose this failure, developers need at least two pieces of information: (1) "out of memory" is reported at $F$; (2) $\text{st->table}$ is overwritten by another thread after the assignment at $a_1$ and before the checking at $a_2$. Both pieces can be collected from execution shortly before the failure. Unfortunately, existing production-run failure-diagnosis techniques cannot deterministically provide these two pieces of information with low overhead.

4. Maintaining Short-term Memory

In this section, we identify and design hardware facilities that maintain short-term memory of program execution to support production-run failure diagnosis.

Our main task is to identify the right types of information to keep in the short-term memory. There is a wide variety of runtime information accessible to the hardware, such as the program counter of every executed instruction and the value stored in every register. We cannot record all these hardware events due to hardware cost and performance concerns. Therefore, we need to identify events that are most useful for failure diagnosis to keep in the short-term memory.

4.1 LBR for sequential-bug failure diagnosis

The outcome of conditional branches in software is among the most useful information for failure diagnosis. It can address the control-flow uncertainties discussed in Section explicitly presenting the execution path leading to the failure. In addition, previous work has shown that many sequential-bug failures are exactly caused by control-flow problems.

Fortunately, the hardware facility that maintains the short-term memory of this information already exists in the form of Last Branch Record (LBR). There are different types of branches that could be recorded in LBR. Our system conceptually LBR to record three types of branches that can help resolve the outcomes of conditional branches in user-level programs, as shown in Table.

4.2 LCR for concurrency-bug failure diagnosis

4.2.1 LCR design

Previous work has found coherence events maintained by existing hardware performance counters, which are discussed in Section useful in diagnosing concurrency-bug failures. Inspired by that, we propose a hardware extension that maintains the short-term memory of coherence events. We call it Last Cache-coherence Record, short as LCR.

Assuming a MESI cache-coherence protocol, LCR includes the following components on chip:

1. A special hardware register that configures which type of coherence events to record. The supported events are exactly those that can already be counted by existing hardware performance counters – load or store instructions that observe certain cache-coherence states right before the cache access, as discussed in Section. Detailed configuration options follow those that are provided by existing hardware for performance counter registers, as also discussed in Section. For example, the cache-coherence state can be any combination of modified state, exclusive state, shared state, and invalid state; this register can be configured to filter out kernel-level instructions or user-level instructions from LCR.

2. $K$ pairs of special hardware registers per core that record the latest $K$ LCR events. Each pair records the instruction counter and the specific cache-coherence state observed by that instruction. By default, we set $K$ to be 16, resembling the setting of LBR on Nehalem processors.

3. Extra circuits that keep updating LCR on each core. After the retirement of L1 data-cache access instructions, the program counter of the instruction and the cache-coherence state observed by this instruction will be recorded in LCR, if the state matches the configuration.

We expect LCR to be a simple extension on machines that already support hardware cache-coherence performance counters and LBR, such as Intel machines discussed in Section. LCR essentially requires extending these machines from being able to count cache-coherence events to being able to record while counting, just like building LBR in machines that can count branch-taken events.

It is difficult to accurately estimate the overhead incurred by LCR without building it in real hardware. We expect the overhead to be low for two reasons. (1) Counting cache-coherence events incurs no perceivable overhead on commodity machines. (2) Using LBR incurs negligible overhead on commodity machines, as shown in Section.

---

2 Memory addresses are not recorded.
failure thread
23]. A failure runs and is related to the failure root cause [2, 18, atomicity violations and order violations [25].

most common types of concurrency bugs: single-variable threads or not. In the following, we discuss this issue for two remaining question is whether such events occur in failure events exist for all common types of concurrency bugs. The work [2], which shows that failure predicting coherence access LCR at the failure site.

occur outside the failure thread cannot be obtained when we fault, violates an assertion, and so on. Coherence events that occurs, such as the thread that encounters a segmentation fault lead to failure, as shown in the example in Table 3. Our previous work [2] discovered that failure predicting events exist at a2 for all these atomicity violations[3]. Therefore, we will focus on discussing whether a2 is in the failure thread below.

a2 almost always exists in the failure thread for RWR and WWR atomicity violations, as the incorrect value read by a2 will soon lead to failure in the same thread as a2, such as a segmentation fault inside puts (the RWR example in Table 3) and the out-of-memory failure in Figure 4. Here, we leverage observations from previous empirical studies [47, 48], which show that concurrency-bug failures always occur in the thread that first reads an incorrect value from a shared variable.

Figure 5: A read-too-early order violation in FFT. Thread 2 should initialize Gend before thread 1 accesses it.

Single-variable atomicity violations occur when two consecutive memory accesses from one thread, denoted as a1 and a2, are unserializable interleaved by an access, denoted as a3, from another thread. All four types of single-variable atomicity violations are demonstrated in Table 3. Our previous work [2] discovered that failure predicting events exist at a2 for all these atomicity violations[3]. Therefore, we will focus on discussing whether a2 is in the failure thread below.

a2 almost always exists in the failure thread for RWR and WWR atomicity violations, as the incorrect value read by a2 will soon lead to failure in the same thread as a2, such as a segmentation fault inside puts (the RWR example in Table 3) and the out-of-memory failure in Figure 4. Here, we leverage observations from previous empirical studies [47, 48], which show that concurrency-bug failures always occur in the thread that first reads an incorrect value from a shared variable.

a2 often exists in the failure thread for RWW atomicity violations — after a2 writes an incorrect value, the thread performing a2 will very likely use this incorrect value which will lead to failure, as shown in the example in Table 3.

For RWR atomicity violations, failures usually occur in the thread containing a3, instead of a2, as shown in Table 3. Unfortunately, a3 is not always a failure predicting event, unless it is preceded by another access to the same variable.

3 The rationale is that a2 would encounter different cache-coherence states during failure runs and success runs, due to the impact of a3. For example, the invalid state of st->table encountered by if (!st->table) in Figure 4 is related to the failure root cause and can predict the failure.

Table 3: The failure predicting events (FPE) of concurrency bugs (Invalid and Exclusive refer to the cache-coherence state observed by a load or store right before it accesses L1 data-cache; F denotes the location of failure.)

<table>
<thead>
<tr>
<th>Bug Type</th>
<th>FPE</th>
<th>Does FPE exist in failure thread?</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>RWR Atomicity Violation</td>
<td>Invalid Read</td>
<td>Almost Always</td>
<td>/<em>Thread 1</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>if(ptr) / /a1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>puts(ptr); //a2,F</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>/<em>Thread 2</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>prt= NULL; //a3</td>
</tr>
<tr>
<td>RWW Atomicity Violation</td>
<td>Invalid Write</td>
<td>Often</td>
<td>/<em>Thread 1</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>tmp=cnt+deposit1; //a1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cnt=tmp; //a2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>printf(&quot;Balance=%d&quot;,cnt); //F</td>
</tr>
<tr>
<td>WWR Atomicity Violation</td>
<td>Invalid Read</td>
<td>Almost Always</td>
<td>/<em>Thread 1</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>log=CLOSE; //a1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>log=OPEN; //a2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>printf(&quot;Takes %f&quot;,Gend); //B2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>printf(&quot;End at %f&quot;, Gend); //B2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>{/output failure} //F</td>
</tr>
<tr>
<td>WRW Atomicity Violation</td>
<td>Invalid Read</td>
<td>Sometimes</td>
<td>/<em>Thread 2</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>if(log!=OPEN) //a3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>{/Gend uninitialized}</td>
</tr>
<tr>
<td>Read-too-early Order Violation</td>
<td>Exclusive Read</td>
<td>Often</td>
<td>Figure 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>printf(&quot;Takes %f&quot;,Gend = Init); //B2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>{/output failure} //F</td>
</tr>
<tr>
<td>Read-too-late Order Violation</td>
<td>Invalid Read</td>
<td>Often</td>
<td>Figure 5</td>
</tr>
</tbody>
</table>

Since our LCR design is built upon existing hardware cache-coherence performance events, it will share some constraints of existing hardware performance counters. For example, the LCRs on different cores are separately maintained and accessed, just like how existing performance counters work. When we profile LCR from a particular thread in a multi-threaded program, only the LCR maintained by the core that is currently running this thread will be accessed. Extending LCR to access multiple cores’ information simultaneously would require non-trivial change to existing hardware. As another example, on SMT machines, multiple hardware threads in one core currently share one LBR. We expect a similar design for LCR. This will shorten the execution history recorded for each thread.

4.2.2 How useful is LCR?

Is LCR helpful in diagnosing concurrency-bug failures? To some extent, the answer is always “yes”, because LCR can help developers understand the thread interaction right before failures.

Can LCR directly point out the failure root cause? It depends on whether LCR contains a coherence event that is failure predicting and is from the failure thread. An event is considered failure predicting, if it mostly occurs during failure runs and is related to the failure root cause [2, 18, 23]. A failure thread is the thread where the failure first occurs, such as the thread that encounters a segmentation fault, violates an assertion, and so on. Coherence events that occur outside the failure thread cannot be obtained when we access LCR at the failure site.

The above question is partly answered by our previous work [2], which shows that failure predicting coherence events exist for all common types of concurrency bugs. The remaining question is whether such events occur in failure threads or not. In the following, we discuss this issue for two most common types of concurrency bugs: single-variable atomicity violations and order violations [24].
Order violations occur when the expected order between two operations from two threads is flipped. The two most common types of order violations occur either when a read instruction executes too early and hence accesses an uninitialized value (Figure 5) or when a read instruction executes too late and hence accesses a stale value (Figure 6). Previous work [2] has shown that the coherence event at the read instruction is often a failure predicting event. For the example shown in Figure 5, B2 would encounter an exclusive state only during failure runs, when End is uninitialized. For the example shown in Figure 6, B3 rarely encounters an invalid state during success runs, but always encounters an invalid state during failure runs when B2 executes after the NULL-assignment from another thread.

The above failure predicting events do exist in failure threads, as the incorrect values returned by the read instructions quickly lead to failures, such as wrong outputs (Figure 5) and crashes (Figure 6).

In summary, LCR has a good chance of directly pointing out the root cause of almost all common types of concurrency-bug failures. Even if the root cause cannot be directly pointed out, the thread interaction information provided by LCR is still helpful.

**LCR configuration** For extensibility and generality, we have designed LCR to record a wide variety of coherence events. Following the discussion in Table 3, the following two configurations are most useful for diagnosing user-level concurrency-bug failures.

The first configuration records invalid loads, invalid stores, and exclusive loads. These events cover all the different types of failure-predicting events for common concurrency bugs, as shown by Table 3. However, this configuration may waste the LCR space with stack accesses that are often exclusive loads.

A more space-saving configuration is to replace exclusive loads with shared loads, using the latter to replace the former in diagnosing read-too-early order violations. For instance, consider the bug shown in Figure 5. During success runs, B2 will always encounter a shared state. Therefore, failures are highly correlated with B2 not encountering a shared state. This configuration is more LCR space-saving than the first one, but it may not be as straightforward as the first configuration for developers to reason about.

**4.3 Implementation details**

**Accessing LBR** This includes three steps. First, we configure which types of branches to record through the special register LBR_SELECT, as shown in Table 1. Second, we enable LBR through the IA32_DEBUGCTL register. Finally, the records in LBR can be accessed through registers BRANCH_0_FROM_IP through BRANCH_16_FROM_IP, where BRANCH_i_FROM_IP is the linear address of the ith branch instruction. Since the above registers cannot be accessed at user level, we implemented a Linux kernel module to support accesses from user level, mainly using kernel wrapper functions that execute rdmsr and wrmsr assembly instructions. The interface is shown in Figure 7.

**Minimizing LBR pollution** As mentioned earlier, LBR has a limited capacity — 16 branch entries in Nehalem processors. To minimize the LBR pollution by branches that are irrelevant to user-level software failures, we use the following methods.

First, as discussed in Section 4.1, we configure LBR to filter out kernel level branches.

Second, we always disable LBR right before we read LBR. Our LBR-disabling code does not contain any user-level branches. Consequently, LBR will not be polluted by code executed to access it.

Finally, we remove pollution from common library functions through a collection of wrapper functions. Each wrapper function disables LBR on entry, invokes the original library function, and finally enables LBR on exit. We will refer to this method as LBR toggling. We wrote wrappers for glibc functions and application-specific error-reporting functions. For example, for MySQL, we wrote wrappers for three MySQL-specific functions: `db_doprint`, `my_error`, and `sql_print.error` Each wrapper function contains fewer than 15 lines of code and follows the same routine.

**LCR simulation** We expect that LCR will be added into the hardware performance monitoring unit in the future and will be accessed in a similar way as we access LBR. We expect that the pollution issue will be similarly solved by toggling around common library functions, filtering out kernel-level instructions, and disabling LCR before profiling.

We implement a LCR simulator using PIN binary-instrumentation infrastructure [22]. Our LCR simulator includes two parts. The first part is a simulated L1-cache with MESI coherence protocol, implemented by instrumenting
every memory instruction in the user program and libraries. The second part simulates LCR configuration, disable, enable, and profiling functions. We implement LCR as a per-thread circular buffer with a configurable size. Once enabled, every thread’s circular buffer gets filled by program counters and coherence states of instructions that are executed by the specific thread and satisfy the LCR configuration. Once disabled, every thread’s circular buffer is frozen. When a thread executes the profiling function, that thread’s circular buffer content is retrieved. Finally, we simulate the pollution effect of these four functions by adding dummy entries into the corresponding circular buffer. Specifically, two user-level exclusive reads will be introduced by the ioctl call that enables LCR; two user-level exclusive reads and one user-level shared read will be introduced by the ioctl call that disables LCR.

Our simulation does not simulate OS events, such as exceptions and context switches. However, we believe it is accurate enough to provide a solid evaluation of LCR.

5. Using Short-term Memory

5.1 Log enhancement

The basic way of using LBR and LCR is to enhance failure logging, which we will refer to as LBRLOG and LCRLOG respectively. Developers can use the LBR/LCR record collected at a failure site to reconstruct the control flow and interleaving right before the failure. They may also find failure predicting events from the LBR/LCR record.

To ease the adoption and evaluation of LBRLOG and LCRLOG, we implemented a source-to-source code transformer to automatically enhance a program’s failure logging. The transformation includes several steps:

1. Changing the program compilation configuration to use our wrappers for common library functions (Section 4.2).
2. Inserting LBR/LCR configuration and enabling code at the entry of main function, as shown in Figure 7.
3. Inserting LBR/LCR profiling code right before every existing failure-logging function in the program, as shown in Figure 7. Currently, our implementation takes a developer configurable list of application-specific failure-logging functions, such as ap_log_error in Httpd and error in GNU core utilities software.
4. Registering a custom segmentation-fault signal handler to profile LBR/LCR.

5.2 Automatic failure diagnosis

A more sophisticated way of using LBR/LCR is to automatically locate failure predicting events based on the LBR/LCR content collected from production runs. To achieve this goal, we have designed LBRA and LCRA.

LBRA/LCRA follows the existing statistical fault localization approach 22, 23. It compares information collected from failure runs (i.e., failure-run profiles) with information

```
if (tmp)
  { LBR_LCR_PROFILE(); // success logging site
    tmp = expr;
    error(...);
  }
  { LBR_LCR_PROFILE(); // failure logging site
    tmp = expr;
    error(...);
  }
```

(a) Original code

(b) Transformed code

Figure 8: The logging sites for success and failure profiles.

collected from success runs (i.e., success-run profiles) to figure out events that are most correlated with failure runs. We will go through the design of these three components below:

1. What is the failure-run profile and how to collect it?
2. What is the success-run profile and how to collect it?
3. How to make the comparison?

Failure-run profile A good failure-run profile should have a high chance to contain failure-predicting events. In our system, we use the content of LBR and LCR collected by LBRLOG and LCRLOG as the failure-run profile. Clearly, the profile would contain exactly one LBR/LCR record for each run where a fail-stop failure occurs. For example, when the sort bug shown in Figure 3 manifests, a segmentation fault would occur at location F. The LBR/LCR record collected inside that segmentation fault handler would become the failure run profile.

Success-run profile LBRLOG and LCRLOG do not profile LBR and LCR at all during success runs. Therefore, we need to design success-run profiling separately from LBRLOG and LCRLOG.

Intuitively, we expect success-run profiles to contain LBR/LCR record collected nearby the failure sites, so that success-run profiles and failure-run profiles are comparable.

Guided by this intuition, we define the following program locations as success logging sites: (1) If a segmentation fault can be triggered by instruction i, the program location right after i is a success logging site; (2) If a failure logging function is located at F, a success logging site is right before where the program jumps to the basic block containing F, as shown in Figure 8.

Collecting LBR/LCR record at the success logging sites defined above will naturally provide success-run profiles that are comparable with failure-run profiles. In addition, it also naturally excludes irrelevant success runs from the failure diagnosis process — LBR/LCR will not be profiled during runs that do not execute the code around the failure site.

We have implemented two schemes to collect LBR/LCR at the success logging sites. The proactive scheme inserts LBR/LCR profiling code at every success logging site corresponding to every failure logging site before software release. The reactive scheme waits until a failure occurs at a particular location F, and then inserts LBR/LCR-profiling
code at the success logging site corresponding to $F$. This change can be conducted either on the end users’ machines through dynamic binary transformation [5] or at the development site. In the latter case, the changes will be propagated to the end users in the form of code patches.

These two schemes each have their own strengths. The proactive scheme does not require code redistribution, but incurs higher run-time overhead due to more frequent LBR/LCR profiling. In addition, it cannot help diagnose failures that manifest at unexpected locations, which is always the case for segmentation faults. The reactive scheme has better performance. However, it needs software updates to collect success-run profiles, which may take time. Of course, since failure runs are much rarer than success runs, delays in collecting success-run profiles rarely lead to longer diagnosis latency.

**How to compare?** Each success/failure run profile is a set of events recorded in LBR and LCR. We want to identify the event whose occurrence can best predict the failure.

Similar to previous work on statistical debugging [2] [23], we identify the best failure-predicting event based on the expected prediction precision and recall of the events. Specifically, in our context, prediction precision measures how many runs indeed fail among those that are predicted to fail by the event. It can be calculated by $\frac{|F \cap e|}{|e|}$, where $|e|$ denotes the number of runs where $e$ is recorded in the profile and $|F \cap e|$ denotes the number of failure runs that contain $e$ in their profiles. Prediction recall measures how many runs are predicted to fail by the event among those that indeed fail. It can be calculated by $\frac{|F \cap e|}{|F|}$, where $|F|$ denotes the number of failure runs. We rank all events based on the harmonic mean of the expected prediction precision and recall, and identify the highest ranked event as the best failure-predicting event.

### 5.3 Discussion

**Failure sites** In our current implementation, LBRA/LCRA has much shorter diagnosis latency than the CBI approach. Suppose $e$ needs to occur in a couple of failure-run profiles to be identified as a high-confidence failure predictor. To identify $e$, LBRA/LCRA needs a failure to occur for a couple of times. The CBI approach needs a failure to occur for hundreds of times under their default
sampling rate (1 out of 100). This difference, hundreds of failure occurrences, could mean a long time in practice, because bugs that slip into fields often manifest rarely.

In terms of run-time performance, LBRA/LCRA is better than CBI/CCI and is comparable with PBI. The advantage of LBRA/LCRA mainly comes from two sources. First, LBRA/LCRA collects failure-run profile only at one location, the failure-logging site. Instead, the CBI approach periodically evaluates predicates throughout the execution. Second, CBI and CCI pay extra cost to enable their random sampling. This overhead is often more than 30% for CPU-intensive applications for CBI [3] and more than 800% for CCI [2].

In terms of diagnosis capability, LBRA/LCRA is comparable with PBI and CCI when failures have reasonably short propagation distances, which is true for most failures [12, 29, 34, 48]. CBI is better than LBR-tools for sequential-bug failures whose root causes are not related to branches.

Finally, LBRA and LCRA have a much smaller impact on the executable-file size than CBI and CCI, leaving a much smaller footprint on cache and memory.

**Limitations** The accuracy of our LBR/LCR based tools could be slightly affected by certain hardware issues. Hardware tracks the cache-coherence states at cache-line granularity, instead of variable granularity. This could lead to false sharing problems. Invalid cache states could be caused by both cache eviction and remote write accesses. This could cause one coherence event to appear in both success runs and failure runs. Of course, since the ranking model discussed in Section 5.2 naturally filters out random noises, we expect the diagnosis results to be rarely affected by these issues.

6. Methodology

We conduct all the experiments on an Intel Core i7 machine with 4 physical cores running Linux 3.5. kernel. We separately evaluate LBR and LCR related tools. LBR-related experiments are conducted directly on the real machine; LCR-related experiments are conducted on our PIN-based simulator.

We evaluate the failure-diagnosis capability and runtime performance of LBRLLOG and LBRA using 20 real-world sequential-bug failures. We include in our benchmarks all the 10 failures from LogEnhancer [44] that we can reproduce. We also randomly pick 5 failures from 13 reproducible crash failures from Errolg [45]. Finally, since the above failures are all from C applications, we randomly selected 5 reproducible bugs from the bug database of open-source C++ applications Cppcheck and PBZIP.

To measure performance, we use workloads designed by the software developers that represent the common scenarios in production runs and do not lead to failures. The performance overheads reported are the mean of 10 measurements. To evaluate failure-diagnosis capability, we use the bug triggering inputs used by LogEnhancer, Errolg, and the original bug reports. For all these benchmarks, we conduct a head-to-head quantitative comparison between LBRA and CBI. Further, we evaluate how LBRLLOG can help resolve control-flow uncertainties using all the 6945 logging points in 13 open-source applications. Detailed information about these benchmarks along with the main logging functions instrumented by LBRLLOG is shown in Tables 4 and 5.

We evaluate the failure-diagnosis capability of our LCR proposal using all the 11 real-world concurrency-bug failures used in PBI [2] and CCI [13], following their experiment settings. Our LCR simulator simulates each core’s L1 data-cache as a 2-way associative cache with a block size of 64 Bytes and a total size of 64 KB.

We evaluate different configurations of our LBR/LCR tools. By default, we enable toggling in all tools.

<table>
<thead>
<tr>
<th>Program</th>
<th>Version</th>
<th>KLOC</th>
<th>Root Cause</th>
<th>Failure Symptom</th>
<th>Log Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential-Bug Failures</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache 1</td>
<td>2.0.43</td>
<td>273</td>
<td>config.</td>
<td>error message</td>
<td>2534</td>
</tr>
<tr>
<td>Apache 2</td>
<td>2.2.3</td>
<td>311</td>
<td>semantic</td>
<td>error message</td>
<td>2511</td>
</tr>
<tr>
<td>Apache 3</td>
<td>2.2.9</td>
<td>333</td>
<td>semantic</td>
<td>error message</td>
<td>2515</td>
</tr>
<tr>
<td>cp</td>
<td>4.5.8</td>
<td>1.2</td>
<td>semantic</td>
<td>error message</td>
<td>108</td>
</tr>
<tr>
<td>Cppcheck 1</td>
<td>1.58</td>
<td>138</td>
<td>memory</td>
<td>crash</td>
<td>304</td>
</tr>
<tr>
<td>Cppcheck 2</td>
<td>1.56</td>
<td>131</td>
<td>memory</td>
<td>crash</td>
<td>284</td>
</tr>
<tr>
<td>Cppcheck 3</td>
<td>1.52</td>
<td>118</td>
<td>memory</td>
<td>crash</td>
<td>225</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>1.4.16</td>
<td>55</td>
<td>config.</td>
<td>error message</td>
<td>857</td>
</tr>
<tr>
<td>ln</td>
<td>4.5.1</td>
<td>0.7</td>
<td>semantic</td>
<td>error message</td>
<td>29</td>
</tr>
<tr>
<td>mv</td>
<td>6.8</td>
<td>4.1</td>
<td>semantic</td>
<td>error message</td>
<td>46</td>
</tr>
<tr>
<td>paste</td>
<td>6.10</td>
<td>0.5</td>
<td>memory</td>
<td>hang</td>
<td>23</td>
</tr>
<tr>
<td>PBZIP1</td>
<td>1.1.5</td>
<td>5.7</td>
<td>semantic</td>
<td>error message</td>
<td>305</td>
</tr>
<tr>
<td>PBZIP2</td>
<td>1.1.0</td>
<td>4.6</td>
<td>memory</td>
<td>crash</td>
<td>269</td>
</tr>
<tr>
<td>rm</td>
<td>4.5.4</td>
<td>1.3</td>
<td>semantic</td>
<td>error message</td>
<td>31</td>
</tr>
<tr>
<td>sort</td>
<td>7.2</td>
<td>3.6</td>
<td>memory</td>
<td>crash</td>
<td>36</td>
</tr>
<tr>
<td>Squid1</td>
<td>2.5.5</td>
<td>120</td>
<td>semantic</td>
<td>error message</td>
<td>2427</td>
</tr>
<tr>
<td>Squid2</td>
<td>2.3.54</td>
<td>102</td>
<td>memory</td>
<td>crash</td>
<td>2096</td>
</tr>
<tr>
<td>tac</td>
<td>6.11</td>
<td>0.7</td>
<td>memory</td>
<td>crash</td>
<td>21</td>
</tr>
<tr>
<td>tar 1</td>
<td>1.22</td>
<td>82</td>
<td>semantic</td>
<td>error message</td>
<td>243</td>
</tr>
<tr>
<td>tar 2</td>
<td>1.19</td>
<td>76</td>
<td>semantic</td>
<td>error message</td>
<td>188</td>
</tr>
</tbody>
</table>

Table 4: Features of real-world failures evaluated.

7. Experimental Results

7.1 LBRLLOG evaluation

Our evaluation aims to answer the following questions:

1. Is the LBR record profiled by LBRLLOG useful in resolving control-flow uncertainties?
Table 5: Resolution of control-flow uncertainties by LBR-LOG.

<table>
<thead>
<tr>
<th>Application</th>
<th>Useful br. ratio</th>
<th>#LogSites</th>
<th>Main Log Fun.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>0.86</td>
<td>2515</td>
<td>ap_log_error</td>
</tr>
<tr>
<td>cp</td>
<td>0.77</td>
<td>108</td>
<td>error</td>
</tr>
<tr>
<td>cppcheck</td>
<td>0.98</td>
<td>304</td>
<td>reportError</td>
</tr>
<tr>
<td>lighttpd</td>
<td>0.84</td>
<td>857</td>
<td>log_error_write</td>
</tr>
<tr>
<td>ln</td>
<td>0.81</td>
<td>29</td>
<td>error</td>
</tr>
<tr>
<td>mv</td>
<td>0.74</td>
<td>46</td>
<td>error</td>
</tr>
<tr>
<td>paste</td>
<td>0.86</td>
<td>23</td>
<td>error</td>
</tr>
<tr>
<td>pbzip</td>
<td>0.81</td>
<td>305</td>
<td>fprintf</td>
</tr>
<tr>
<td>rm</td>
<td>0.79</td>
<td>31</td>
<td>error</td>
</tr>
<tr>
<td>sort</td>
<td>0.91</td>
<td>36</td>
<td>error</td>
</tr>
<tr>
<td>Squid</td>
<td>0.88</td>
<td>2427</td>
<td>debug</td>
</tr>
<tr>
<td>tac</td>
<td>0.89</td>
<td>21</td>
<td>error</td>
</tr>
<tr>
<td>tar</td>
<td>0.84</td>
<td>243</td>
<td>open_fatal</td>
</tr>
</tbody>
</table>

As shown in Table 6, LBRLOG is very helpful for diagnosing 16 out of 20 failures. These 16 failures are caused by different types of software bugs: 8 by semantic bugs, 6 by memory bugs, and 2 by configuration errors. As an example, the simplified patch for the sort bug from Section 3.1 is shown in Figure 9a. The root cause branch \( A \) is recorded as the 3rd latest entry in LBR collected by LBRLOG.

LBRLOG fails to contain the root-cause branch, but is still helpful in diagnosing the remaining 4 failures. For example, the ln bug has a long error propagation distance. The root-cause branch would have been captured, if LBR had 4 more entries. LBRLOG captures the branch \( B \) that is related to the root cause as shown in Figure 9b.

Table 6 also shows that most root-cause branches are located within the top 8 entries in LBR. This result validates the heuristic that most software bugs have short error propagation distances, and indicates that even on machines with smaller LBR, LBRLOG is still very useful.

Finally, we measure the distance between the LBR branches and the patch, comparing it with the distance between the failure site and the patch. In general, the former is much shorter than the latter. The patches are within 5 lines of code from some LBR branches in 14 out of 20 cases, while only 2 failure sites are within 5 lines of code from the patches. For 13 failures, some LBR branches are more than 30 lines of code closer to the patches than the failure sites, and all these branches are useful LBR records that cannot be inferred by static control-flow analysis. This further shows that LBRLOG can help diagnose failures and design patches.

7.1.3 Performance

As shown in Table 6, LBRLOG incurs at most 2.28% runtime overhead for all the benchmarks, which is suitable for production-run deployment.

The overhead mainly comes from toggling around library functions. Without toggling, the overhead is at most 0.23% across all benchmarks. This performance improvement comes at the expense of diagnosis capability. As shown
in Table 6 without toggling, LBRLOG will fail to locate any branch that is related to the patch in 5 cases.

Overall, LBRLOG incurs small overheads across a wide range of applications and is suitable for production-run deployment. We can turn off toggling to satisfy even higher performance requirements.

### 7.2 LBR evaluation

Our evaluation of LBR targets the following questions:

1. Is LBR able to automatically locate root-cause branches?
2. Is the performance of LBR (reactive and proactive schemes) suitable for production-run deployment?
3. Can LBR complement CBI, the state-of-the-art production-run failure diagnosis system?

Our experiments configure CBI using its default settings: 1/100 sampling rate; 1000 success runs and 1000 failure runs; with only branch predicates enabled. Our experiments for LBR only use 10 success runs and 10 failure runs.

LBR successfully and automatically locates all the 16 root-cause branches contained in LBR as the top 1 failure predictors. It identifies root-cause related branches as top predictors for all the 20 failures. In comparison, CBI identifies root-cause branches as top predictors for 11 out of 15 C-program failures. CBI fails to report any root-cause related branches in 3 cases, where its random sampling missed the relevant predicates too many times.

The above diagnosis results are achieved with LBR analyzing much fewer failure runs than CBI (10 vs. 1000). When we applied CBI to 500, instead of 1000, failure-run profiles, CBI failed to identify any useful failure predictors for 10 out of 15 C-program failures. This difference would be crucial for software that is not deployed on millions of machines or failures that do not occur very frequently.

As shown in Table 6 the run-time overhead of LBR (reactive mode) is always less than 3%, well suitable for production-run deployment. The overhead of LBR in proactive mode is slightly larger, ranging between 2.09% and 6.29%. It is a good choice for software where updates are infrequent or very expensive. CBI incurs an average overhead of 15.23%, much larger than LBR, mainly due to the instrumentation done by CBI to perform sampling.

The results show that LBR well complements CBI.

### 7.3 LCR evaluation

Our evaluation tries to answer the following questions:

1. Can LCRLOG help diagnose concurrency-bug failures?
2. Can LCRA automatically locate the root causes of concurrency-bug failures?
3. Can LCR complement PBI and CCI, the state-of-the-art production-run concurrency-bug failure diagnosis tools?

**LCRLOG** We consider LCRLOG to directly locate the failure root cause, if the LCR profiled by it contains the failure-predicting coherence event (defined in Section 4.2.2).
As shown in Table 7, LCRL\textsubscript{LOG} directly locates the root cause for 7 out of 11 concurrency-bug failures, which cover different types of root causes and symptoms.

LCRL\textsubscript{LOG} does not directly locate the root cause for Apache5, Cherokee, and Mozilla-JS2 failures, because these bugs cause silent data corruption with no failure logging near the root cause. The MySQL1 failure is caused by a WRW atomicity violation. As shown in Table 3 since the failure-predicting event does not exist in the failure thread, it is not profiled by LCRL\textsubscript{LOG}.

Further, as shown in Table 7, the capacity of LCR is not a problem for the failures we evaluated. Using the more space-saving configuration, the failure-predicting events are always contained in top 4 LCR entries. Even with the more space-consuming configuration, the failure-predicting events are still located within top 12 LCR entries.

<table>
<thead>
<tr>
<th>ID</th>
<th>LCRL\textsubscript{LOG} (Conf1)</th>
<th>LCRL\textsubscript{LOG} (Conf2)</th>
<th>LCRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache4</td>
<td>✓ 3</td>
<td>✓ 5</td>
<td>✓ 1</td>
</tr>
<tr>
<td>Apache5</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Cherokee</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FFT</td>
<td>✓ 4</td>
<td>✓ 6</td>
<td>✓ 1</td>
</tr>
<tr>
<td>LU</td>
<td>✓ 4</td>
<td>✓ 6</td>
<td>✓ 1</td>
</tr>
<tr>
<td>Mozilla-JS1</td>
<td>✓ 3</td>
<td>✓ 8</td>
<td>✓ 1</td>
</tr>
<tr>
<td>Mozilla-JS2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Mozilla-JS3</td>
<td>✓ 3</td>
<td>✓ 11</td>
<td>✓ 1</td>
</tr>
<tr>
<td>MySQL1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MySQL2</td>
<td>✓ 3</td>
<td>✓ 9</td>
<td>✓ 1</td>
</tr>
<tr>
<td>PBZIP3</td>
<td>✓ 3</td>
<td>✓ 7</td>
<td>✓ 1</td>
</tr>
</tbody>
</table>

Table 7: Failure diagnosis capability of LCR. (A number \(n\) after the ✓ indicates the \(n\)-th latest entry returned by LCRL\textsubscript{LOG} or the \(n\)-th best failure predictor returned by LCRA is the root-cause failure-predicting event; Conf1 is the space-saving configuration of LCR; Conf2 is the space-consuming configuration of LCR; LCRA uses Conf2.)

**LCRA** We evaluate whether LCRA can automatically locate the failure-predicting event by applying LCRA to 10 failure runs and 10 success runs in each case.

LCRA successfully ranks the failure-predicting event at the top for all the 7 failures where the failure-predicting event is captured by LCRL\textsubscript{LOG}. For example, for the failure discussed in Section 3.2 (Mozilla-JS3), LCRA automatically locates the invalid state observed by \(a_2\) as the top failure predictor.

We expect LCRA to well complement PBI and CCI. In terms of performance, CCI incurs up to 10 times slow down, due to its software based sampling schemes. We expect LCRA to have similar performance as LBRA, which would be comparable to or slightly better than PBI. In terms of failure-diagnosis capability, LCRA is slightly worse than PBI, which can successfully diagnose all the 11 failures, and comparable with CCI, which can successfully diagnose 7 out of the 11 failures.

The biggest advantage of LCRA is its short failure-diagnosis latency. LCRA achieves the above diagnosis results using only 10 failure-run profiles, while PBI and CCI need the failures to occur hundreds to thousands of times \([2, 18]\). This is especially a problem for concurrency-bug failures that often occur non-deterministically and rarely.

## 8. Related Work

We briefly discuss related work that has not been discussed.

### Hardware performance monitoring unit

The branch tracing facility has been used in several recent work, but has never been used for production-run failure diagnosis. TH\textsuperscript{HE}ME uses LBR for testing coverage analysis \([40]\). Recent work conducts vulnerability or malware analysis on branch traces generated by the branch tracing facility \([41, 46]\). Intel GNU* GDB tool \([16]\) uses BTS to store all executed branches in an OS-provided ring buffer.

Our work uses the branch tracing facilities for different purposes from previous work, which leads to different designs. For example, all the above work collects the branch trace of the whole execution, while we focus on the LBR collected at the failure site. Our system aims to achieve very small run-time overhead for production-run deployment, while the above work does not share the same goal. Some of them \([16, 46]\) intentionally use BTS which has higher overhead. TH\textsuperscript{HE}ME \([40]\) uses LBR, but still incurs much larger overhead than our tools. The reason is that TH\textsuperscript{HE}ME’s design goal, computing testing coverage, demands periodic LBR profiling throughout program execution. Instead, LBRL\textsubscript{LOG} only profiles LBR when software fails.

General hardware performance counters have been used to identify malware \([10]\) and detect data races \([11, 37]\). These tools all monitor and analyze the whole execution, instead of focusing on the execution leading to a failure. Race detectors \([11, 37]\) focus on one specific type of software bugs, and cannot help diagnose general software failures. In addition, not being guided by a specific failure, race detectors would report a large number of false positives \([18]\).

### Production-run failure diagnosis

Record-and-replay techniques \([11, 13, 19, 20]\) can help diagnose production-run failures. However, they could hurt the end users’ privacy and incur large overhead for deterministic replay of multi-threaded software. Triage \([38]\) diagnoses production-run failures by applying automated bug detection during on-site replay, which is supported by OS modifications. Overall, record-and-replay techniques and our system can complement each other in failure diagnosis.

An adaptive version of CBI was proposed based on dynamic binary rewriting \([4]\). CBI-adaptive iteratively changes sampling locations based on the failure location and the diagnosis results from earlier iterations. Without knowing the exact control-flow leading to failures, CBI-adaptive needs hundreds of iterations and evaluates about 40% of all program predicates before it finishes failure diagnosis.
Hardware support for bug detection  A lot of work has been done to speed up sequential-bug detection through hardware support\cite{6, 7, 24, 26, 32, 33, 42, 50}. Different from LBRA and LBR-LOG, most of these proposals rely on non-existing hardware.

A lot of work has proposed detecting concurrency bugs through hardware support\cite{6, 7, 24, 26, 32, 33, 42, 50}. LCR has drawn inspiration from these work. However, since previous work focuses on bug detection, it requires the hardware and software system to contiguously monitor and analyze program execution, while maintaining a long execution history. Many bug detectors need to report suspicious execution patterns even if they do no lead to failures. LCR leverages the unique need of failure diagnosis and designs a very simple hardware extension to maintain a short-term execution history.

Bugaboo\cite{26} detects a wide variety of concurrency bugs by identifying rare communication patterns. The communication graph in Bugaboo associates with every memory instruction \(m\) from thread \(t\) a context, the sequence of communication events observed by \(t\) immediately prior to \(m\). A LCR record is similar to a context, as they both contain a short-term history of thread interaction. However, Bugaboo and our system have very different designs, because they have different goals — Bugaboo detects concurrency bugs even without failure information; our system helps diagnose production-run failures. Bugaboo maintains and checks the context of every memory instruction in every thread throughout the execution. Our system leverages the unique need of failure diagnosis and only uses LCR collected in the failure thread right before the failure. In addition, Bugaboo extends existing cache-coherence protocol to collect context events, while each LCR event is already supported by existing hardware performance monitoring unit.

ECMon\cite{28} proposes a hardware extension that allows custom handlers to execute whenever certain type of cache (coherence) events happen. ECMon and LCR aim for different usage scenarios, and hence have different designs. Software uses ECMon to process cache events throughout the program execution; our system uses LCR to access the last few cache coherence events at the moment of failure. Also different from ECMon, the design of LCR is built upon existing hardware performance monitoring unit.

9. Conclusion

We design and implement a novel mechanism that leverages hardware’s short-term memory to support production-run failure diagnosis. We identify an existing hardware performance monitoring unit, LBR, and design a simple hardware extension, LCR, to maintain a short-term memory of hardware events that are useful for failure diagnosis. Our evaluation of 31 sequential-bug and concurrency-bug failures from 18 open-source software shows that our LBR/LCR based tools can effectively enhance failure logging and automatically locate failure root causes with small run-time overhead. We believe that our LBR/LCR system provides a good balance between run-time performance, diagnosis latency, and diagnosis capability. Our experience demonstrates that short-term memory is sufficient for diagnosing a wide variety of real-world failures. It also shows that a very simple hardware-extension can provide significant help for production-run failure diagnosis.

Acknowledgments

We thank the anonymous reviewers for their insightful feedback which has substantially improved the content and presentation of this paper; Ding Yuan and the Opera Group for their tremendous help in sharing the benchmarks; Ben Liblit, Piramanayagam Nainar and Peter Ohmann for their valuable guidance on using the CBI framework; Jason Mars and Asim Kadav for their insightful comments on designing the kernel interface to LBR. This work is supported in part by NSF grants CCF-1018180, CCF-1054616, and CCF-1217582; and a Clare Boothe Luce faculty fellowship.

References


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


