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

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 performance-monitoring unit, Last Branch Record (LBR), that records the last few taken branches to help diagnose sequential-bug failures. 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-bug failures. 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 the root causes for 27 out of 31 failures, with less than 3% run-time overhead. Comparing with existing production-run failure diagnosis systems that rely on sampling, our system does not require a failure to occur many times and significantly shortens the latency of failure diagnosis.

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 [9, 17, 29, 34] and threatening people's lives [22]. 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).

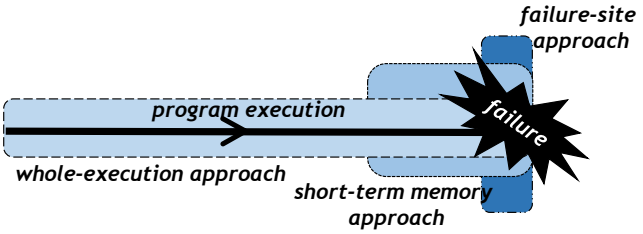

Figure 1: Different approaches of diagnosing production-run failures (The rectangles illustrate the program states directly collected by different approaches).

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 always 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 [41–43]. However, static analysis is not a panacea. It cannot help when software fails at unpredicted locations (e.g., during a segmentation fault). Its help is very 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 of 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 [3, 18, 23, 24] 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 [23, 24], 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, 32, 45] 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 diagnose likely does not distribute evenly throughout the execution. Intuitively, the closer to a failure, the more likely such information exists.

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 maintains a short-term execution history, and hence can easily 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? At every machine cycle, a lot of hardware events occur. 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 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 presented in Section 4.
- 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. Section 5 presents the details.
- 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. 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 production-run failure diagnosis systems that rely on sampling [3, 18, 23, 24], our failure-diagnosis system has tens to hundreds of times shorter diagnosis latency.

2. Background

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% [2]. The following discussion will focus on LBR.

LBR is part of the hardware performance monitoring unit, originally designed for performance profiling. LBR branch

[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.

IA32_DEBUGCTL	ID: 0x1d9
0x801	Enable LBR
0x0	Disable LBR
LBR_SELECT	**ID: 0x1c8**
0x1	*Filter branches occurring in ring 0.
0x2	Filter branches occurring in other levels.
0x4	Filter conditional branches.
0x8	*Filter near relative calls.
0x10	*Filter near indirect calls.
0x20	*Filter near returns.
0x40	*Filter near unconditional indirect jumps.
0x80	Filter near unconditional relative branches.
0x100	*Filter far branches.

Table 1: LBR related Machine Specific Registers in Intel Nehalem (*: the masks used in this work).

```
1  if(a!=0)
2
3
4   a++;
5
6
7  else
8   a--;
```

(a)

```
1  cmpl $0x0,-0x4(%rbp)
2  je label<else>
3
   ; jump of the false edge
4  addl $0x1,-0x4(%rbp)
5  jmp label<end>
6  ; jump of the true edge
7  label<else> :
8  subl $0x1,-0x4(%rbp)
9  label<end> :
```

(b)

Figure 2: Conditional branches in source and machine code.

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 [16]. All the experiments in this paper are conducted on Intel Nehalem processors.

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 [38] to make the

```
/* sort.c */
void merge (...){
  avoid_trashing_input (...);
  for (...){
    open_input_files (...);
  }
}
int open_input_files (...){
  if(files[i].pid != 0) // C
    open_temp(files[i].name, files[i].pid);
  else ...
}
/* lib/hash.c */
void * hash_lookup (Hash_table *table){
 struct hash_entry *bucket = table->bucket ; // F
} //called by wait_proc, which is called by open_temp
/* sort.c */
int avoid_trashing_input (...){
 for (; i < nfiles; i++){
  if(...)
    same = true;
  else if (...)
     break;
  ...
  if(same){
    int num_merged = 0;
    while (i + num_merged < nfiles){ // A
      num_merged += mergefiles(...);
      memmove(&files[i], &files[i+num_merged],..);//B
      ...
  }}}
  ...
}
```

Figure 3: A sequential bug in `sort` utility in Coreutils-7.2.

mapping between machine-code branches and source-code branches easier. We reuse this technique and skip the details due to space constraints.

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.

3. Motivating Examples

3.1. Case 1: a sequential-bug failure

Figure 3 shows a memory bug in `sort` utility from Coreutils. When a user tries to merge already sorted files such that the output file is one of the input files, the program crashes inside the `hash_lookup` function.

This segmentation fault is caused by a buffer overflow in `memmove` within `avoid_trashing_input` (marked as *B* in Figure 3). This buffer overflow corrupts `files[i].pid`, which causes the control flow to deviate from the intended path at *C*. This eventually leads to a segmentation fault at *F*.

This buffer overflow is caused by the wrong while-loop condition at *A*. As we can see, the loop condition (`i+num_merged<nfiles`) is intended to avoid buffer overflow. However, since the value of `num_merged` is increased

after this sanity check and before the access of `files` array, the buffer overflow occurs as long as this while-loop executes at least one iteration.

The *control-flow uncertainty* makes this failure very difficult to diagnose. First, `hash_lookup` has 9 different callers across 6 different files. Developers cannot even start their diagnosis without knowing the execution history. Second, even if the developers obtain the call-stack or even the core-dump from the end users' machines, they will likely ignore the `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 successfully 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 execution-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 a_1, and then checked at a_2. In most cases, this check will pass, as long as `New` has succeeded at a_1. Occasionally, `st->table` is set to `NULL` by another thread at a_3 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 `InitState`. Traditional log-enhancing techniques [43] *cannot* help as logging extra variables at failure-logging site F does not help diagnose this failure.

To successfully diagnose this failure, developers need at least two pieces of information: (1) "`out of memory`" is reported at F; (2) `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. Maintain Short-term Memory in Hardware

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

```
InitState(...){                    FreeState(...){
 // executed by Thread1              //executed by Thread2
 st->table=New(st); // a1           ...
 ...                                Destroy(st->table);
 if(!st->table) { // a2             st->table = NULL; // a3
   ReportOutOfMemory();//F          ...
   return JS_FALSE;                }
 }
}

ReportOutOfMemory(){
 error("out of memory");
}
```

Figure 4: A concurrency bug in Mozilla JavaScript Engine.

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 3, helping developers to reconstruct 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 [24, 33].

Fortunately, the hardware facility that maintains the short-term memory of such 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 configures LBR to record three types of branches that can help resolve the outcomes of conditional branches in user-level programs, as shown in Table 1.

4.2. LCR for concurrency-bug failure diagnosis

4.2.1. LCR design Previous work [3] has found that a specific set of hardware performance events are very useful in diagnosing concurrency-bug failures. This set of performance events are L1 data-cache cache-coherence events, which we will simply refer to as *coherence events*. Inspired by previous work, we propose a hardware extension that maintains the short-term memory of coherence events and hence help diagnose concurrency-bug failures. 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. Specifically, we can configure LCR to record load or store instructions that observe certain cache-coherence states right before the cache access. The cache-coherence state can be any combination of mutual

	Does FPE exist in failure thread?	FPE	Example

RWR Atom. Vio.	Yes	Invalid Read	/*Thread 1*/
			if(ptr) //a_1
			fputs(ptr); //a_2,F
			/*Thread 2*/
			ptr=NULL; //a_3

RWW Atom. Vio.	Mostly	Invalid Write	/*Thread 1*/
			tmp=cnt+deposit1; //a_1
			cnt=tmp; //a_2
			printf("Balance=%d",cnt); //F
			/*Thread 2*/
			tmp=cnt+deposit2;
			cnt=tmp; //a_3

WWR Atom. Vio.	Yes	Invalid Read	Figure 4

WRW Atom. Vio.	Sometimes	Invalid Read	/*Thread 1*/
			log=CLOSE;//a_1
			log=OPEN; //a_2
			/*Thread 2*/
			if(log!=OPEN) //a_3
			{//output failure} //F

Read-too-early Order Vio.	Mostly	Exclusive Read	Figure 5
Read-too-late Order Vio.	Mostly	Invalid Read	Figure 6

Table 2: The failure predicting events (FPE) of concurrency bugs (Invalid and Exclusive refer to the cache-coherence state observed by a load or store when it accesses L1 data cache; F denotes where failure occurs.)

state, exclusive state, shared state, and invalid state. This register can also 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 LCR configuration.

The LCRs on different cores are separately maintained and accessed. 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. Since existing hardware performance counters already support counting L1 data-cache cache-coherence events on each core, we expect LCR to be a simple hardware extension.

4.2.2. How useful is LCR? To some extent, LCR is always helpful in diagnosing concurrency-bug failures, because it can help developers get more understanding about thread interaction right before the failure.

For LCR to directly point out the failure root cause, a failure predicting coherence event has to exist in the failure thread. An event is considered *failure predicting*, if it mostly occurs during failure runs and is related to the failure root cause [3, 18, 24]. 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.

Previous work [3] shows that failure predicting coherence events exist for all common types of concurrency bugs. It is unclear whether such events occur in failure threads. In the following, we discuss this issue for two most common types of concurrency bugs: single-variable atomicity violations and order violations [25].

Single-variable atomicity violations occur when two consecutive memory accesses from one thread, denoted as a_1 and a_2, are unserializably interleaved by an access, denoted as a_3, from another thread. All four types of single-variable atomicity violations are demonstrated in Table 2.

As discovered by previous work, failure predicting events exist at a_2 for all these atomicity violations. The rationale is that a_2 would encounter different cache-coherence states during failure runs and success runs, due to the impact of a_3. 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.

To know whether a_2 is in the failure thread, we use observations from previous empirical studies [45, 46], which show that concurrency-bug failures almost always occur in the thread that first reads an incorrect value from a shared variable.

a_2 exists in the failure thread for RWR and WWR atomicity violations, as the incorrect value read by a_2 will soon lead to failure in the same thread as a_2, such as a segmentation fault inside `fputs` (the RWR example in Table 2) and the out-of-memory failure in Figure 4.

a_2 often exists in the failure thread for RWW atomicity violations — after a_2 writes an incorrect value, the thread performing a_2 will very likely use this incorrect value which will lead to failure, as shown in the example in Table 2.

For WRW atomicity violations, failures usually occur in the thread of a_3, instead of a_2, as shown in Table 2. Unfortunately, a_3 is not always a failure predicting event, unless it is preceded by another access to the same memory location.

Order violations occur when the expected order between two operations from two threads are flipped. The two most

```
/* Thread1 (failure thread)*/    /*Thread2*/
printf("End at %f", Gend);//B1   //Gend uninitialized
printf("Take%f",Gend-init);//B2  //until here
                                 Gend=time();//A
```

Figure 5: An order violation in FFT. Thread 2 should initialize `Gend` before thread 1 accesses it.

```
/*Thread 1*/          /*Thread 2 (failure thread)*/
                      pthread_mutex_lock(mutex);//B1
mutex = NULL;//A      ...
                      pthread_mutex_unlock(mutex);//B2
                      ...
                      pthread_mutex_lock(mutex); //B3
```

Figure 6: A read-too-late order violation in PBZIP2. Thread 2 should use `mutex` before thread 1 destroys it.

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 [3] has shown that the coherence event at the read instruction is often a failure predicting event. For the example shown in Figure 5, B_2 would encounter an exclusive state only during failure runs, when `Gend` is uninitialized. For the example shown in Figure 6, B_3 will encounter an invalid state only during failure runs, when it 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 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 2, 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 2. 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, B_2 will always encounter a shared state. Therefore, failures are highly correlated with B_2 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.

```
fd = open ("/dev/lbrdriver", O_RDWR);
ioctl(fd, DRIVER_CLEAN_LBR); // Reset LBR entries
ioctl(fd, DRIVER_CONFIG_LBR); // Configure filtering
ioctl(fd, DRIVER_ENABLE_LBR); // Enable LBR recording
...
...
ioctl(fd, DRIVER_DISABLE_LBR);// Disable LBR recording
ioctl(fd, DRIVER_PROFILE_LBR);// Profile LBR
error();
```

Figure 7: Interface exported by our LBR kernel module.

4.3. Implementation details

Accessing LBR Using LBR includes three steps. First, configure which types of branches to record through the special register LBR_SELECT, as shown in Table 1. Second, 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_n_FROM_IP is the linear address of the n^{th} 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 written 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*.

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 [28]. 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 a 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 our simulator is accurate enough to provide a solid evaluation of LCR.

5. Use short-term memory for failure diagnosis

5.1. Log enhancement

The basic way of using LBR or LCR is to enhance failure logging, which we will refer to as LBRLOG or LCRLOG. 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 4.3).

(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 signal handler to profile LBR/LCR whenever the program encounters a segmentation fault.

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 both failure runs and success runs. These failure predicting events are highly related to failure root causes and can help developers design patches [23, 24]. We will refer to this system as LBRA and LCRA. Like previous work [23, 24], the design of LBRA/LCRA includes three components:

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 compare the success-run profiles with the failure-run profiles?

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.

```
if (expr)
{
  error(...);
}
```

(a) Original code

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

(b) Transformed code

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

Success-run profile A good success-run profile should contain content that is comparable with the failure-run profile. For example, if the success-run profile contains cache-coherence events and the failure-run profile contains the outcomes of conditional branches, it is meaningless to compare them. Furthermore, even if both success-run and failure-run profiles contain outcomes of conditional branches, the comparison is still meaningless if the branches contained in these two profiles do not overlap.

We aim to collect LBR/LCR at locations that are close to the failure site and are likely to be executed during success runs. We will refer to these locations as *success logging sites*. For a segmentation fault triggered by instruction i, the ideal success logging site is right after i. For a failure reported by logging function at F, we choose the success logging site to be right before the program jumps to the basic block that contains F, as shown in Figure 8.

The above design naturally provides 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 [6] 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 larger run-time overhead due to more frequent LBR/LCR profiling. In addition, it cannot help diagnose failures that manifest at unpredicted locations, which is always the case for crashes. 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 [3, 24], 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; prediction *recall* measures how many runs are predicted to fail by the event among those that indeed fail. The formulae to calculate these two metrics for an event e are shown below, where $|F|$ denotes the number of failure runs, $|e|$ denotes the number of runs where e is recorded in the profile; $|F\&e|$ denotes the number of failure runs that contain e in their profiles.

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.

$$\text{Precision} = \frac{|\text{F\&e}|}{|\text{e}|} \quad \text{Recall} = \frac{|\text{F\&e}|}{|\text{F}|} \tag{1}$$

5.3. Discussion and comparison

Failure sites In our current implementation, LBRLOG and LCRLOG treat existing failure-logging functions and segmentation-fault handler as failure sites. Consequently, if software fails by silently corrupting data, LBR and LCR will not be collected in a timely manner and hence may not help the diagnosis. Fortunately, the problem of identifying the right places to insert log functions has been largely addressed by recent work [42] and is outside the scope of this work.

Multiple failures It is very natural for large software to encounter failures caused by different bugs during production runs. The existence of multiple failures will not affect our system. From each failure-run profile, we can identify the location of the failure site and hence separately handle failures that occur at different program locations. Very rarely, different root causes may lead to failures at the same location, such as the example shown in Figure 4. In this case, we will see that even the best failure predicting event does not appear in every failure run. Fortunately, this rarely affects the relative ranking among events, and our system can still identify the best failure predicting events.

Log enhancement Traditional failure logging either dumps core images or call stacks, or records the values of selected program variables. LBR/LCR has its unique advantages and can well complement these traditional approaches.

In terms of preserving privacy, LBR/LCR is among the best.

In terms of failure diagnosis capability, LCR provides interleaving related information that is difficult to obtain from traditional approaches. LBR can resolve uncertain control flows that cannot be inferred by traditional approaches. Note that, even the recently proposed LogEnhancer [43] can have difficulty in resolving control flows in sibling functions (i.e. functions that are not in the call-stack at the failure site). Of course, LBR and LCR content is limited to the execution shortly before the failure. In addition, coredumps and logging variables can provide concrete variable value information that may be unavailable from LBR/LCR.

In terms of logging latency, logging LBR/LCR is much faster than dumping cores and logging call-stacks. In our evaluation, logging LBR/LCR takes less than 20 μs; dumping core can easily take more than 200 ms; and recording the call stack takes about 200 μs. If developers want to conduct logging at locations beyond failure sites, logging LBR/LCR is more suitable than dumping cores or call-stacks.

In terms of run-time performance, LBR/LCR profiling incurs negligible overhead if toggling is disabled. Enabling toggling around library functions could cause perceivable overhead for applications that frequently invoke library functions, which will be evaluated in Section 7.

Finally, logging LBR/LCR is more generic than logging selected variables. At different program locations, different variables need to be selected to represent the most important program states for failure diagnosis. When software fails at an unpredicted location, such as during a segmentation fault, variable logging usually cannot be applied.

Automated failure diagnosis The cooperative bug isolation (CBI) approach, including CBI [23, 24], CCI [18], and PBI [3], are the state of the art production-run failure diagnosis techniques. This approach first evaluates certain predicates, such as whether a branch is taken, at randomly sampled locations. It then performs statistical analysis on the data collected from many success and failure runs to identify predicates that strongly correlate with failures. These predicates are referred to as *failure predictors*. To conduct random sampling, CBI and CCI use source-code instrumentation, while PBI leverages hardware performance counters.

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 most bugs that slip into production runs 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 [4] and more than 800% for CCI [3].

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, 32, 45]. 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.

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, with the LCR-related experiments conducted on our PIN-based LCR simulator.

We evaluate the failure-diagnosis capability and runtime performance of LBRLOG and LBRA using 20 real-world sequential-bug failures. We include in our benchmarks all the 10 failures from LogEnhancer [43] that we can reproduce. We also randomly pick 5 failures from 13 reproducible crash failures from Errlog [42]. 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, Errlog, 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 LBRLOG can help resolve control-flow uncertainties using all the 6945 logging points in 13 open-source applications. The detailed information about these benchmarks along with the main logging functions instrumented by LBRLOG are shown in Table 3 and Table 4.

We evaluate the failure-diagnosis capability of our LCR proposal using all the 11 real-world concurrency-bug failures used in PBI [3] and CCI [19], 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 will evaluate different configurations of our LBR/LCR tools. By default, we enable toggling in all tools.

7. Experimental Results

7.1. LBRLOG evaluation

Our evaluation aims to answer the following questions:

1. Is the LBR record profiled by LBRLOG useful in resolving control-flow uncertainties?
2. Is the LBR record profiled by LBRLOG useful in locating failure root causes?
3. Is the runtime performance of LBRLOG suitable for production-run deployment?

7.1.1. Resolving uncertain control flows For each logging site l, a branch record in LBR is considered *useful* if the taken-ness of this branch cannot be inferred based on the execution of l through static control flow analysis. We compute the ratio of useful branches in LBR record entries collected at every failure-logging site in an application. We refer to this ratio as *useful branch ratio*. Since it is impractical to design inputs to exercise all the logging sites, we implement an LLVM-based

Program	Version	KLOC	Root Cause	Failure Symptom	Log Points
Sequential-Bug Failures					
Apache 1	2.0.43	273	config.	error message	2534
Apache 2	2.2.3	311	semantic	error message	2511
Apache 3	2.2.9	333	semantic	error message	2515
cp	4.5.8	1.2	semantic	error message	108
Cppcheck 1	1.58	138	memory	crash	304
Cppcheck 2	1.56	131	memory	crash	284
Cppcheck 3	1.52	118	memory	crash	225
Lighttpd	1.4.16	55	config.	error message	857
ln	4.5.1	0.7	semantic	error message	29
mv	6.8	4.1	semantic	error message	46
paste	6.10	0.5	memory	hang	23
PBZIP1	1.1.5	5.7	semantic	error message	305
PBZIP2	1.1.0	4.6	memory	crash	269
rm	4.5.4	1.3	semantic	error message	31
sort	7.2	3.6	memory	crash	36
Squid1	2.5.S5	120	semantic	error message	2427
Squid2	2.3.S4	102	memory	crash	2096
tac	6.11	0.7	memory	crash	21
tar 1	1.22	82	semantic	error message	243
tar 2	1.19	76	semantic	error message	188
Concurrency-Bug Failures					
Apache 4	2.0.50	263	A.V.	crash	2412
Apache 5	2.2.9	333	A.V.	corrupted log	2515
Cherokee	0.98.0	85	A.V.	corrupted log	184
FFT	2.0	1.3	O.V.	wrong output	59
LU	2.0	1.2	O.V.	wrong output	45
Mozilla-JS1	1.5	107	A.V.	crash	343
Mozilla-JS2	1.5	107	A.V.	wrong output	343
Mozilla-JS3	1.5	107	A.V.	error message	343
MySQL1	4.0.18	658	A.V.	crash	1585
MySQL2	4.0.12	639	A.V.	wrong output	1523
PBZIP3	0.9.4	2.1	O.V.	crash	163

Table 3: Features of real-world failures evaluated.

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

Table 4: How LBRLOG resolves control-flow uncertainties.

App.	Locate Root Cause				Patch distance (LoC)		Overhead (%)				
	LBRLOG w/ tog.	LBRLOG w/o tog.	LBRA	CBI	failure site	LBR	LBRLOG w/ tog.	LBRLOG w/o tog.	LBRA reactive	LBRA proactive	CBI
Apache1	✓ 3	✓ 3	✓ 1	✓ 2	∞	**3**	0.31	0.11	0.39	3.87	3.01
Apache2	✓ 2*	✓ 2*	✓ 2*	-	∞	475	0.42	0.09	0.43	4.61	5.48
Apache3	✓ 2	✓ 2	✓ 1	✓ 1	**1**	**1**	0.33	0.17	0.52	3.43	2.70
cp	✓ 2	-	✓ 1	✓ 1	17	15	1.77	0.23	2.13	3.61	25.90
Cppcheck1	✓ 5*	✓ 5*	✓ 1*	N/A	∞	∞	2.04	0.04	2.73	5.61	N/A
Cppcheck2	✓ 3	✓ 3	✓ 1	N/A	∞	**2**	0.24	0.02	0.29	2.09	N/A
Cppcheck3	✓ 6	✓ 6	✓ 1	N/A	∞	10	1.16	0.06	1.39	4.68	N/A
Lighttpd	✓ 4	✓ 4	✓ 1	-	**0**	**1**	0.65	0.11	0.73	2.33	6.34
ln	✓ 13*	-	✓ 1*	✓ 1	254	33	1.88	0.18	1.95	4.69	22.48
mv	✓ 12	✓ 14	✓ 1	✓ 2	309	**0**	1.79	0.11	2.84	5.70	15.55
paste	✓ 6	-	✓ 1	✓ 1	35	**3**	1.31	0.08	1.78	2.50	14.32
PBZIP1	✓ 4	-	✓ 1	N/A	41	**1**	0.29	0.07	0.34	5.73	N/A
PBZIP2	✓ 1	✓ 1	✓ 1	N/A	12	**1**	0.79	0.04	0.91	4.62	N/A
rm	✓ 5	✓ 5	✓ 1	✓ 2	31	**0**	2.28	0.21	2.38	6.29	24.77
sort	✓ 3	✓ 5	✓ 1	✓ 1	∞	**4**	0.44	0.19	0.74	4.16	43.45
Squid1	✓ 2	✓ 2	✓ 1	-	123	**2**	1.26	0.05	1.45	2.79	6.29
Squid2	✓ 10	✓ 10	✓ 1	✓ 1	59	**1**	2.19	0.03	2.42	3.62	7.49
tac	✓ 3*	✓ 3*	✓ 1*	✓ 3*	∞	∞	2.13	0.06	2.57	2.82	26.43
tar1	✓ 4	✓ 4	✓ 1	✓ 1	∞	**2**	0.52	0.09	0.73	3.10	14.30
tar2	✓ 2	-	✓ 1	✓ 2	24	**0**	0.40	0.11	0.45	2.63	9.91

Table 5: Results of LBRLOG and LBRA. (The number n after ✓ indicates the n-th latest LBR entry returned by LBRLOG or the n-th top predictor identified by LBRA/CBI; *: root-cause branch is missed, but a branch related to the root cause is provided; "-": no branches related to the root cause are identified; N/A: CBI does not work for C++ applications; ∞: in different files.)

analyzer to calculate this ratio. Specifically, given a logging site, the analyzer explores backwards along all possible paths until each path contains 16 branches that could fill LBR and checks which branches are useful. The numbers presented in Table 4 are the averaged ratio among all logging sites.

As shown by Table 4, the average useful branch ratio ranges from 0.74 to 0.98 for all the 6945 logging points across 13 applications. This shows that LBR provides a generic and useful mechanism to resolve control-flow uncertainties.

7.1.2. Failure diagnostic capability To measure the failure-diagnosis capability of LBRLOG, we compare the branches captured by LBRLOG with the patches. We consider LBRLOG to be very helpful in locating the failure root cause, if the patch mainly changes one of the branches recorded by LBRLOG, denoted as ✓in Table 5. We will refer to this branch changed by the patch as *root-cause branch*. We consider LBRLOG to be helpful, if the patch mainly changes the computation or usage of a condition variable that involves in one of the branches recorded by LBRLOG, denoted as ✓* in Table 5.

As shown in Table 5, 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 3^{rd} 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 *A* that is related to the root cause as shown in Figure 9b.

```
avoid_trashing_input (...)
- while (i + num_merged < nfiles) // A
+ do{
          ...
+ }while (i < nfiles);
```

(a) sort patch

```
int main (int argc, char **argv)
- if (n_files == 1)
+ if (!target_directory_specified && n_files == 1)
  ...
  if (target_directory_specified) // A
```

(b) ln patch

Figure 9: Branches captured by LBRLOG and patches.

Table 5 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 result further shows that

LBRLOG can help diagnose failures and design patches.

7.1.3. Performance As shown in Table 5, 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 5, 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. LBRA evaluation

Our evaluation of LBRA targets the following questions:

1. Is LBRA able to automatically locate root-cause branches?
2. Is the performance of LBRA (reactive and proactive schemes) suitable for production-run deployment?
3. Can LBRA 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 LBRA only use 10 success runs and 10 failure runs.

LBRA 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 for too many times.

The above diagnosis results are achieved with LBRA 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 5, the run-time overhead of LBRA (reactive mode) is always less than 3%, well suitable for production-run deployment. The overhead of LBRA 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 LBRA, mainly due to the instrumentation done by CBI to performs sampling.

The evaluation shows that LBRA well complements CBI.

7.3. LCR evaluation

Our evaluation of LCR 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 6, LCRLOG directly locates the root cause for 7 out of 11 concurrency-bug failures, which cover different types of root causes and symptoms.

LCRLOG 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 demonstrated in Table 2, since the failure-predicting event does not exist in the failure thread, it is not profiled by LCRLOG.

Further, as shown in Table 6, 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.

ID	LCRLOG (Conf1)	LCRLOG (Conf2)	LCRA
Apache4	✓ 3	✓ 5	✓ 1
Apache5	-	-	-
Cherokee	-	-	-
FFT	✓ 4	✓ 6	✓ 1
LU	✓ 4	✓ 6	✓ 1
Mozilla-JS1	✓ 3	✓ 8	✓ 1
Mozilla-JS2	-	-	-
Mozilla-JS3	✓ 3	✓ 11	✓ 1
MySQL1	-	-	-
MySQL2	✓ 3	✓ 9	✓ 1
PBZIP3	✓ 3	✓ 7	✓ 1

Table 6: Failure diagnosis capability of LCR. (A number n after the ✓ indicates the n-th latest entry returned by LCRLOG 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 LCRLOG. 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 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 [3, 19]. 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. THeME uses LBR for testing coverage analysis [38]. Recent work conducts vulnerability or malware analysis on branch traces generated by the branch tracing facility [39, 44]. Intel GNU* GDB tool [15] 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 [15, 44] intentionally use the large-overhead BTS.

General hardware performance counters have been used to identify malware [10] and detect data races [11, 35]. These tools all monitor and analyze the whole execution, instead of focusing on the execution leading to a failure. Race detectors [11, 35] focus on one specific type of software bugs, and cannot help diagnose general software failures. In addition, without guided by a specific failure, race detectors would report a large number of false positives [18].

Production-run failure diagnosis Record-and-replay techniques [1, 13, 20, 21] 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. Overall, record-and-replay techniques and our system can complement each other in failure diagnosis.

Triage [36] diagnoses production-run failures by applying automated bug detection during on-site replay. Triage takes checkpoints throughout the execution, and relies on OS modification to support low-overhead checkpoint and replay. Triage and LBR/LCR-system well complement each other, as they use completely different diagnosis techniques — Triage relies on bug detectors; LBRA/LCRA uses statistical debugging; and LBRLOG/LCRLOG aids developers' manual investigation by recording key events right before the failure.

An adaptive version of CBI was proposed based on dynamic binary rewriting [5]. This adaptive CBI 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 [37, 47]. Different from LBRA and LBRLOG, most of these proposals rely on non-existing hardware.

A lot of work has proposed detecting concurrency bugs through hardware support [7, 8, 26, 27, 30, 31, 40, 48]. 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 [27] 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 with 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.

9. Conclusion

We design and implement a novel mechanism that leverages hardware's short-term memory to support production-run failure diagnosis. We identify 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 less than 3% run-time overhead. We believe 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 failure diagnosis.

References

[1] Gautam Altekar and Ion Stoica. Odr: output-deterministic replay for multicore debugging. In *SOSP*, 2009.

[2] Intel Technical Articles. Intel code execution trace resources. http://noggin.intel.com/content/intel-code-execution-trace-resources.

[3] Joy Arulraj, Po-Chun Chang, Guoliang Jin, and Shan Lu. Production-run software failure diagnosis via hardware performance counters. In *ASPLOS*, 2013.

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

[5] Piramanayagam Arumuga Nainar and Ben Liblit. Adaptive bug isolation. In *ICSE*, 2010.

[6] Bryan Buck and Jeffrey K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 2000.

[7] Luis Ceze, Pablo Montesinos, Christoph von Praun, and Josep Torrellas. Colorama: Architectural support for data-centric synchronization. In *HPCA*, 2007.

[8] Lee Chew and David Lie. Kivati: Fast detection and prevention of atomicity violations. In *EuroSys*, 2010.

[9] CNNMoneyTech. Is knight's $440 million glitch the costliest computer bug ever? http://money.cnn.com/2012/08/09/technology/knight-expensive-computer-bug/index.html.

[10] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. In *ISCA '13*, 2013.

[11] Joseph L. Greathouse, Zhiqiang Ma, Matthew I. Frank, Ramesh Peri, and Todd M. Austin. Demand-driven software race detection using hardware performance counters. In *ISCA*, 2011.

[12] Weining Gu, Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Zhenyu Yang. Characterization of linux kernel behavior under errors. In *DSN*, 2003.

[13] Derek R. Hower, Pablo Montesinos, Luis Ceze, Mark D. Hill, and Josep Torrellas. Two hardware-based approaches for deterministic multiprocessor replay. *Commun. ACM*, 52(6), June 2009.

[14] Intel. Nehalem performance monitoring unit programming guide. http://software.intel.com/sites/default/files/m/5/2/c/f/1/30320-Nehalem-PMU-Programming-Guide-Core.pdf, 2010.

[15] Intel. Gdb - the gnu* project debugger for intel architecture. http://software.intel.com/en-us/articles/intel-system-studio-gdb#Trace, March 2013.

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

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

[18] Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. Instrumentation and sampling strategies for Cooperative Concurrency Bug Isolation. In *OOPSLA*, 2010.

[19] Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *OOPSLA*, 2010.

[20] Samuel T. King, George W. Dunlap, and Peter M. Chen. Operating systems with time-traveling virtual machines. In *Usenix Annual Technical Conference*, 2005.

[21] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. In *ASPLOS*, 2010.

[22] Nancy Leveson and Clark S. Turner. An investigation of the therac-25 accidents. In *IEEE Computer*, 1993.

[23] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[24] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

[25] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes – a comprehensive study of real world concurrency bug characteristics. In *ASPLOS*, 2008.

[26] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *ASPLOS*, 2006.

[27] Brandon Lucia and Luis Ceze. Finding concurrency bugs with context-aware communication graphs. In *MICRO*, 2009.

[28] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

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

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

[32] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies c a safe method to survive software failures. In *SOSP*, 2005.

[33] Raul Santelices, James A. Jones, Yanbing Yu, and Mary Jean Harrold. Lightweight fault-localization using multiple coverage types. In *ICSE*, 2009.

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

[35] Tianwei Sheng, Neil Vachharajani, Stéphane Eranian, Robert Hundt, Wenguang Chen, and Weimin Zheng. Racez: a lightweight and non-invasive race detection tool for production applications. In *ICSE*, 2011.

[36] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Triage: Diagnosing production run failures at the user's site. In *SOSP*, 2007.

[37] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *HPCA*, 2007.

[38] Kristen Walcott-Justice, Jason Mars, and Mary Lou Soffa. Theme: a system for testing by hardware monitoring events. In *ISSTA*, 2012.

[39] Carsten Willems, Ralf Hund, Andreas Fobian, Dennis Felsch, Thorsten Holz, and Amit Vasudevan. Down to the bare metal: using processor features for binary analysis. In *ACSAC '12*, 2012.

[40] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, 2009.

[41] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *ASPLOS*, 2010.

[42] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, M. Michael Lee, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *OSDI*, 2012.

[43] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. In *ASPLOS*, 2011.

[44] Liwei Yuan, Weichao Xing, Haibo Chen, and Binyu Zang. Security breaches as pmu deviation: detecting and identifying security attacks using performance counters. In *APSys*, 2011.

[45] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. ConSeq: detecting concurrency bugs through sequential errors. In *ASPLOS*, 2011.

[46] Wei Zhang, Chong Sun, and Shan Lu. ConMem: Detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS*, 2010.

[47] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iWatcher: Efficient Architecture Support for Software Debugging. In *ISCA*, 2004.

[48] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. Hard: Hardware-assisted lockset-based race detection. In *HPCA*, 2007.