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

Joy Arulraj, Guoliang Jin and Shan Lu
Production-Run Failure Diagnosis

• Goal
  – Figure out root cause of failure on client machines
  – Fix them quickly
Importance

• Social and financial impact
  – Toyota Prius software glitch
  – NASDAQ Facebook IPO glitch
Challenges

• Limited program execution information
  – Performance and privacy reasons

• Complicated root cause
  – Sequential bugs
  – Concurrency bugs

• Need to diagnose and fix quickly
Existing Tools

(1) During entire execution
(2) At failure-site
(3) Sampling
Limitations Of Existing Tools

- (1) Entire-execution approach
- (2) Failure-site approach

![Diagram showing limitations of existing tools with axes for Diagnosis Latency and Performance Overhead.](image)
Limitations Of Existing Tools

- (1) Entire-execution approach
- (2) Failure-site approach
- (3) Sampling

Performance
Overhead

Diagnosis Latency

1/100 sampling rate $\Rightarrow$ $\sim$100 failures required for diagnosis
Challenge

• Low performance overhead
  – Collect *little* execution information

• Low diagnosis latency
  – Collect *root-cause* related information

Which part of the program execution is most likely to contain root-cause information?
Our Solution: Last Execution Record

• Execution right before failure
  – Last Execution Record (LXR)

• **How** to collect this information efficiently?
  – Leverage simple hardware support
Last Execution Record (LXR)

- (1) Entire-execution approach
- (2) Failure-site approach
- (3) Sampling
Outline

• LXR Design
• Failure diagnosis using LXR
• Evaluation
LXR Design Questions & Principles

• **What** should we collect in LXR?
  – **Useful** for failure diagnosis

• **How** to collect LXR?
  – **Lightweight** to collect
LXR Design For Sequential Bugs

• **What** should we collect in LXR?
  – Recently taken branches

• **How** to collect LXR?
  – Use existing hardware support -- LBR
Last Branch Record (LBR)

• **Existing hardware feature**
  – Set of recently taken branches
  – Circular buffer with 16 entries (Intel Nehalem)
  – **Overhead: negligible**
Is LBR useful?

• Root-cause of many types of sequential bugs [PLDI.2005]
• Error-propagation distances tend to be short [DSN.2003]
Sequential Bug Example

- Coreutils: sort -m -o file1 file1

```c
// SORT.C

void merge (...) {
    ...
    open_input_files(...);
}

int open_input_files (...) {
    if (files[i].pid != 0) /* child process */
        table ➔ bucket = val;
    else ...
}
```

CALL STACK
- open_input_files()
- merge()
- ...
- main()
Is LBR useful?

- Coreutils: `sort -m -o file1 file1`

```c
// SORT.C

void merge (...) {
    avoid_trashing_input(...);
    ...
    open_input_files(...);
}

int avoid_trashing_input (...) {
    ...
    int num_merged = 0;
    while (i + num_merged < nfiles) {
        num_merged += mergefiles(...);
        memmove(&files[i], &files[i+num_merged],);
    }
}

int open_input_files (...) {
    if (files[i].pid != 0)
        table bucket = val;
    else ...
}
```

LBR

<table>
<thead>
<tr>
<th>if (files[i].pid != 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>while (i+num_merged &lt; nfiles)</td>
</tr>
</tbody>
</table>

failure
LXR Design For Concurrency Bugs

• **What** should we collect in LXR?
  – Recently executed cache-access instructions
  – Cache-coherence state observed (M/E/S/I)

• **How** to maintain and collect LXR?
  – Key hardware feature already exists
  – Propose a simple hardware extension to use that
Last Cache-coherence Record (LCR)

• **Existing hardware feature**
  – Configurable cache-coherence event counting

• **Extension:**
  – Buffer to collect this information
  – Set of recent L1 data cache access instructions

• **Overhead: not perceivable**

<table>
<thead>
<tr>
<th>Cache-access Instruction Pointer</th>
<th>Cache-coherence State (M/E/S/I)</th>
</tr>
</thead>
</table>

**Lightweight**
Is LCR Useful?

- Related to concurrency bug root-causes [ASPLOS.2013]
- Error-propagation distances tend to be short [ASPLOS.2011]
Is LCR useful?

- Mozilla JavaScript Engine

```javascript
InitState(...){
    table = New();

    if (table == NULL) {
        ReportOutOfMemory();
        return JS_FALSE;
    }
}

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

CALL STACK
- ReportOutOfMemory()
- InitState()
- ...
- main()

failure
Is LCR useful?

- Mozilla JavaScript Engine: Success Run

```javascript
InitState(...){
    table = New();
    if (table == NULL) {
        ReportOutOfMemory();
        return JS_FALSE;
    }
    ReportOutOfMemory(){
        error("out of memory");
    }
}
```

```javascript
FreeState(...){
    Destroy(table);
    table = NULL;
}
```

```javascript
Thread 1
Modified
Thread 2
Modified
```

```
W
R
```

```
R
W
```

```
Modified
```

```
✔
```

```
W
```

```
R
```

```
Modified
```
Is LCR useful?

- Mozilla JavaScript Engine: Failure Run

```javascript
InitState(...){
  table = New();
  if (table == NULL) {
    ReportOutOfMemory();
    return JS_FALSE;
  }
}

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

```javascript
FreeState(...){
  Destroy(table);
  table = NULL;
}
```

Thread 1: **Invalid**
- `InitState(...){
  table = New();
  if (table == NULL) {
    ReportOutOfMemory();
    return JS_FALSE;
  }
}
- `ReportOutOfMemory(){
  error("out of memory");
}

Thread 2: **Modified**
- `FreeState(...){
  Destroy(table);
  table = NULL;
}
```

**LCR**
- `table == NULL` Invalid
- `table = New()` Invalid
- `...` ...

`failure`
Outline

• LXR Design
• Failure diagnosis using LXR
• Evaluation
Manual failure diagnosis

- Enhance logging by collecting LXR
  - Existing failure logging functions
  - Signal handler

```c
// failure logging function
define_error_wrapper(args){
    DISABLE_LXR();
    PROFILE_LXR();
    error (args);
}
```

```c
// signal handler
void handler(int signo){
    DISABLE_LXR();
    PROFILE_LXR();
    ...
}
```
Automated failure diagnosis

• Collect LXR in both failure and success runs
• Statistical analysis
  – Automatically identify failure predictors

<table>
<thead>
<tr>
<th>LCR</th>
<th>table == NULL</th>
<th>Invalid</th>
<th>table = New()</th>
<th>Invalid</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCR</td>
<td>table == NULL</td>
<td>Modified</td>
<td>table = New()</td>
<td>Invalid</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LCR AUTOMATED</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>table == NULL</td>
<td>0.91</td>
</tr>
<tr>
<td>table = New()</td>
<td>0.56</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Implementation details

• LBR exposed via Linux kernel module
  – Enable, configure and access using our interface
  – Reducing LBR pollution from irrelevant branches
  – Details in paper

• LCR simulated using PIN infrastructure
  – L1 data cache with MESI coherence protocol
  – Interface similar to LBR
  – Details in paper
Outline

• LXR Design
• Failure diagnosis using LXR
• Evaluation
Methodology

• 31 real-world failures
  – In open-source server, client, utility programs
  – 20 sequential and 11 concurrency bugs

• Compared against CBI/CCI
  – State-of-the-art software-based tools
  – Diagnose sequential and concurrency bugs
  – Perform sampling to lower overhead
Does LBR help locate root cause?

<table>
<thead>
<tr>
<th>APPLICATION</th>
<th>LBR MANUAL (N\textsuperscript{th} entry)</th>
<th>LBR AUTOMATED (N\textsuperscript{th} entry)</th>
<th>CBI (N\textsuperscript{th} entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Squid</td>
<td>2</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Coreutils</td>
<td>12</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Tar</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>PBZIP</td>
<td>4</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

- Root-cause branch mostly in recent 8 LBR entries
- So, even short-term LBR memory sufficient!
Cross-checking LBR with patches

<table>
<thead>
<tr>
<th>APPLICATION</th>
<th>FAILURE SITE TO PATCH</th>
<th>LBR TO PATCH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(LoC)</td>
<td>(LoC)</td>
</tr>
<tr>
<td>Apache</td>
<td>Another file</td>
<td>3</td>
</tr>
<tr>
<td>Squid</td>
<td>123</td>
<td>2</td>
</tr>
<tr>
<td>Coreutils</td>
<td>309</td>
<td>0</td>
</tr>
<tr>
<td>Tar</td>
<td>Another file</td>
<td>2</td>
</tr>
<tr>
<td>PBZIP</td>
<td>41</td>
<td>1</td>
</tr>
</tbody>
</table>

- LBR entries are much closer to patch for most bugs
## Diagnosis Latency

<table>
<thead>
<tr>
<th>TOOL</th>
<th>Diagnosis Latency</th>
<th>Sampling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual LBR</td>
<td>1 failure run</td>
<td>No</td>
</tr>
<tr>
<td>Automated LBR</td>
<td>10 failure runs</td>
<td>No</td>
</tr>
<tr>
<td>CBI</td>
<td>1000 failure runs</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- LBR tools need fewer failure runs for diagnosis
- CBI uses sampling which increases latency
Performance Overhead

![Performance Overhead Graph]

- Apache
- Squid
- Coreutils
- Tar
- PBZIP

Performance Overhead (%)

- Manual LBR
- Automated LBR
- CBI
Does LCR help locate root cause?

<table>
<thead>
<tr>
<th>APPLICATION</th>
<th>LCR MANUAL (N\textsuperscript{th} entry)</th>
<th>LCR AUTOMATED (N\textsuperscript{th} entry)</th>
<th>CCI (N\textsuperscript{th} entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Cherokee</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Mozilla</td>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MySQL</td>
<td>9</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>PBZIP</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Locates root-cause in 7 out of 11 failures
- So, short-term LCR memory sufficient
Summary

Performance Overhead

Low

Diagnosis Latency

High

Last Execution Record

(1) Entire-execution approach

(2) Failure-site approach

(3) Sampling