Diagnosing Production-Run Concurrency-Bug Failures

Shan Lu
University of Wisconsin, Madison
Outline

• Myself and my group
• Production-run failure diagnosis
  – What is this problem
  – What are our solutions
    • CCI [OOPSLA’10]
    • PBI [ASPLOS’13]
    • LXR [ASPLOS’14]
• Conclusions
A little bit about myself

Shan 山  Lu 卢
The most exciting thing ...
Software bugs

• How many of you have been bothered by bugs?
Fighting software bugs is crucial

• Software is everywhere

• Software bugs are widespread and costly
  – Lead to 40% system down time [Blueprints 2000]
  – Cost 312 Billion lost per year [Cambridge 2013]
Different aspects of fighting bugs

- In-house bug detection
- In-field failure recovery
- In-field failure diagnosis
- In-house bug fixing

Low overhead

High accuracy
Work from my group

In-house bug detection
- [ASPLOS06]; [SOSP07]; [ASPLOS09]; [ASPLOS10]; [ASPLOS11]; [OOPSLA13]

Concurrent bugs

In-field failure recovery
- [ASPLOS13.A]; [FSE14]

Performance bugs

In-field failure diagnosis
- [OOPSLA10]; [ASPLOS13.B]; [ASPLOS14]

In-field bug fixing
- [PLDI11]; [OSDI12]

Not yet

In-house bug fixing
- [PLDI12]; [ICSE13]

[OOPSLA14]

[CAV13]
Our high-level approach

[ASPLOS07];[ASPLOS11];[OOPSLA10];
[PLDI11];[PLDI12];[OSDI12];[ASPLOS13.A];
[CAV13]

fault

failure

error

[ASPLOS06];[SOSP07];[ASPLOS09];
[OOPSLA10];[ASPLOS10];[ASPLOS11];
[ASPLOS13.B]; [ICSE13]; [OOPSLA13]

[ASPLOS06];[MICRO06];
[ASPLOS13.B];[ASPLOS14]

[ASPLOS06];[SOSP07];[OOPSLA10];
[ASPLOS13.B]; [ASPLOS14];[OOPSLA14]
Focus of this talk

In-house:
- bug detection: [ASPLOS06]; [SOSP07]; [ASPLOS09]; [ASPLOS10]; [ASPLOS11]; [OOPSLA13]
- bug fixing: [PLDI11]; [OSDI12]

In-field:
- failure recovery: [ASPLOS13.A]; [FSE14]
- failure diagnosis: [OOPSLA10]; [ASPLOS13.B]; [ASPLOS14]
- Not yet

concurrency bugs

performance bugs
What are concurrency bugs?

- Untimely accesses among threads (buggy interleavings)

```
ptr = malloc(SIZE);
...
if (!ptr){
    ReportOutofMem();
    exit(1);
}
```

```
End = time();
print("%u", End);
print("%u", End-Start);
free(ptr);
ptr= NULL;
```

Mozilla

```
Thread 1
Thread 2
```

```
End = time();
print("%u", End);
print("%u", End-Start);
free(ptr);
ptr= NULL;
```

FFT
Con. bugs are common
These failures need to be diagnosed before they can be fixed!
Failure diagnosis is challenging

- Limited information
- Failures are difficult to repeat
- Root causes are difficult to reason about
Example

Thread 1

ptr = malloc(SIZE);
...
if (!ptr){
    ReportOutOfMem();
    exit(1);
}

Mozilla

Thread 2

free(ptr);
ptr=NULL;
Example

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

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

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

CALL STACK

- ReportOutOfMemory()
- InitState()
- ...
- main()

***.js
out of memory

***
Design space

Questions

Performance
Capability
Latency

Goals

What to collect
How to collect
How to use the collected
Previous work

- coredump
- bug detector
- replay

Capability vs. Performance

Star: Previous work

Diagram points:
- coredump
- bug detector
- replay
Our work

- coredump
- replay
- bug detector
- CCI
Our work
Our work

![3D scatter plot with axes labeled Diagnostic Latency, Performance, and Capability. Points marked for coredump, PBI, LXR, CCI, bug detector, and replay.]
• Myself & my group
• Production-run failure diagnosis
  – What is the problem
  – What are our solutions

• Conclusion
**How to do better than state-of-art?**

<table>
<thead>
<tr>
<th>What to collect</th>
<th>How to collect</th>
<th>How to use the collected</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Performance**
- **Capability**
- **Latency**
How to do better than state-of-art?

<table>
<thead>
<tr>
<th>What to collect</th>
<th>How to collect</th>
<th>How to use the collected</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sampling</td>
<td></td>
</tr>
</tbody>
</table>

- **Performance**
- **Capability**
- **Latency**
How to do better than state-of-art?

<table>
<thead>
<tr>
<th>What to collect</th>
<th>How to collect</th>
<th>How to use the collected</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sampling</td>
<td>Cooperative statistical analysis</td>
</tr>
</tbody>
</table>

Performance | Capability | Latency
Cooperative Bug Isolation (CBI)

True in most failure runs, false in most correct runs

<table>
<thead>
<tr>
<th>Performance</th>
<th>Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>??</td>
</tr>
</tbody>
</table>

Program Source

Compiler
Predicates
Sampling

Predicate & 😊/😊

Branch
Return value
...

Failure Predictors

Statistical Debugging
Does it work for concurrency bugs?

Why does CBI not work?

```c
Thread 1

ptr = malloc(SIZE);
...
if (!ptr){ //b
    ReportOutofMem();
    exit(1);
}

Thread 2

free(ptr);
ptr=NULL;
```

Predicate

```
... taken b
!taken b
...
```
Cooperative Con-Bug Isolation (CCI)

Program Source → Compiler

Predicates → Sampling

Failure Predictors → Statistical Debugging

Performance | Capability
---|---
Mixed | Good

Instrumentation and Sampling Strategies for Cooperative Concurrency Bug Isolation, OOPSLA’10
What to collect? (predicate design)

[Capability] reflect the root causes of many concurrency bugs

[Performance] Simple properties that
Concurrency bug root cause patterns

Atomicity Violation | Order Violation

Learning from Mistakes --- A Comprehensive Study on Real World Concurrency Bug Characteristics, ASPLOS’08
Concurrence bug root cause patterns

**Atomicity Violation**

Thread 1
- access x
- access x
- access x

Thread 2
- access x
- access x
- access x

**Order Violation**

Thread 1
- access x
- access x
- access x

Thread 2
- access x
- access x
- access x
Whether two successive accesses to a memory location were by two distinct threads or one thread.
CCI-Prev can reflect root causes

### Atomicity Violation

- **Thread 1:**
  - Access X
  - Access X
  - Access X

- **Thread 2:**
  - Access X
  - Access X
  - Access X

### Order Violation

- **Thread 1:**
  - Access X
  - Access X
  - Access X

- **Thread 2:**
  - Access X
  - Access X
  - Access X
Is CCI-Prev useful? (Example)

Thread 1

```c
ptr = malloc(SIZE);
...
if (!ptr){
    ReportOutOfMem();
    exit(1);
}
```

Thread 2

```c
free(ptr);
ptr=NULL;
```

Mozilla
Example (correct runs)

thread 1

\[
\text{ptr} = \text{malloc} (\text{SIZE});
\]

... if (!ptr) {
    ReportOutOfMem();
    exit(1);
}

thread 2

\[
\text{free} (\text{ptr});
\text{ptr}=\text{NULL};
\]

Predicate

<table>
<thead>
<tr>
<th>Predicate</th>
<th>😊</th>
<th>😞</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remote_1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>local_1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example (failure run)

Thread 1

\[ \text{ptr} = \text{malloc (SIZE)}; \]

Thread 2

\[ \text{free (ptr);} \]

\[ \text{ptr} = \text{NULL}; \]

Example (failure run)

```
if (!ptr) {
    ReportOutOfMem();
    exit(1);
}
```
How to evaluate?

thread 1

...  
ptr = malloc (SIZE);
lock(glock);
remote = test_and_insert(&ptr, curTid);
record(l, remote);
temp = ptr;
unlock(glock);
if (!temp) {
    ReportOutOfMem();
    exit(1);
}

thread 2

...  
free (ptr);
ptr=NULL;
...  
Predicate

<table>
<thead>
<tr>
<th></th>
<th>😊</th>
<th>😞</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remote_l</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>local_l</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a global hash table

<table>
<thead>
<tr>
<th>address</th>
<th>ThreadID</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>&amp; ptr</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
How to sample?
How to sample branch predicates?

A: if (!temp2) {
  if (sample())
  record (A, TRUE);
  ...
} else {
  if (sample())
  record (A, FALSE);
  ...
}

B: if (!temp3) {
  ... independent
  if (sample())
  record (C, TRUE);
  ...
} else {
  if (sample())
  record (C, FALSE);
  ... independent
  ...
}

B: if (!temp) {
  if (sample())
  record (B, TRUE);
  ...
} else {
  if (sample())
  record (B, FALSE);
  ...
}
thread 1

…

ptr = malloc (SIZE);

…

if (!ptr) {
    ReportOutOfMem();
    exit(1);
}

thread 2

…

free (ptr);

ptr = NULL;

…

Does traditional sampling work?
How to sample CCI-Prev?

Does traditional sampling work? **NO!**
### Thread-coordinated, bursty sampling

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>if (sample())</strong>&lt;br&gt;lock (...);&lt;br&gt;...&lt;br&gt;<strong>ptr = tmp1;</strong>&lt;br&gt;unlock(...);&lt;br&gt;<strong>else ...</strong></td>
<td><strong>if (sample())</strong>&lt;br&gt;lock (...);&lt;br&gt;...&lt;br&gt;<strong>tmp2 = ptr;</strong>&lt;br&gt;unlock(...);&lt;br&gt;<strong>else ...</strong></td>
</tr>
<tr>
<td><strong>if (sample())</strong>&lt;br&gt;lock (...);&lt;br&gt;...&lt;br&gt;<strong>tmp3 = ptr;</strong>&lt;br&gt;unlock(...);&lt;br&gt;<strong>else ...</strong></td>
<td><strong>if (sample())</strong>&lt;br&gt;lock (...);&lt;br&gt;...&lt;br&gt;<strong>ptr=NULL;</strong>&lt;br&gt;unlock(...);&lt;br&gt;<strong>else ...</strong></td>
</tr>
</tbody>
</table>
Other predicates

- Capability (manual effort)
- Performance (overhead)

- Prev
- Havoc
- FunRe
Evaluation methodology

<table>
<thead>
<tr>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache-1</td>
</tr>
<tr>
<td>Apache-2</td>
</tr>
<tr>
<td>Cherokee</td>
</tr>
<tr>
<td>FFT</td>
</tr>
<tr>
<td>LU</td>
</tr>
<tr>
<td>Mozilla-JS-1</td>
</tr>
<tr>
<td>Mozilla-JS-2</td>
</tr>
<tr>
<td>Mozilla-JS-3</td>
</tr>
<tr>
<td>PBZIP2</td>
</tr>
</tbody>
</table>

CIL-based static code instrumentor
1/100 sampling rate, ~3000 runs in total (failure:success~1:1)
## Diagnosis capability (w/ sampling)

<table>
<thead>
<tr>
<th>Program</th>
<th>CCI-Prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache-1</td>
<td>✓ top1</td>
</tr>
<tr>
<td>Apache-2</td>
<td>✓ top1</td>
</tr>
<tr>
<td>Cherokee</td>
<td>✗</td>
</tr>
<tr>
<td>FFT</td>
<td>✓ top1</td>
</tr>
<tr>
<td>LU</td>
<td>✓ top1</td>
</tr>
<tr>
<td>Mozilla-JS-1</td>
<td>✗</td>
</tr>
<tr>
<td>Mozilla-JS-2</td>
<td>✓ top1</td>
</tr>
<tr>
<td>Mozilla-JS-3</td>
<td>✓ top2</td>
</tr>
<tr>
<td>PBZIP2</td>
<td>✓ top1</td>
</tr>
</tbody>
</table>

1/1000 sampling rate, ~3000 runs in total (failure:success~1:1)
## Diagnosis performance (overhead)

<table>
<thead>
<tr>
<th></th>
<th>Prev</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Sampling</td>
</tr>
<tr>
<td>Apache-1</td>
<td>62.6%</td>
</tr>
<tr>
<td>Apache-2</td>
<td>8.4%</td>
</tr>
<tr>
<td>Cherokee</td>
<td>19.1%</td>
</tr>
<tr>
<td>FFT</td>
<td>169 %</td>
</tr>
<tr>
<td>LU</td>
<td>57857 %</td>
</tr>
<tr>
<td>Mozilla-JS</td>
<td>11311 %</td>
</tr>
<tr>
<td>PBZIP2</td>
<td>0.2%</td>
</tr>
</tbody>
</table>
Are we done?
### How to do better than CCI?

<table>
<thead>
<tr>
<th>What to collect</th>
<th>How to collect</th>
<th>How to use the collected</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCI-Prev</td>
<td>Sampling</td>
<td>Cooperative statistical analysis</td>
</tr>
</tbody>
</table>

- Performance
- Capability
- Latency
# How to do better than CCI?

## What to collect | How to collect | How to use the collected
--- | --- | ---
Sampling

**Slow sampling infrastructure**

- **Performance**
- **Capability**
- **Latency**
### How to do better than CCI?

<table>
<thead>
<tr>
<th>What to collect</th>
<th>How to collect</th>
<th>How to use the collected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sampling</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Slow sampling infrastructure
- Inaccurate evaluation

---

**Performance** | **Capability** | **Latency**
How to do better than CCI?

<table>
<thead>
<tr>
<th>What to collect</th>
<th>How to collect</th>
<th>How to use the collected</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hardware-based evaluation &amp; sampling</td>
<td>Slow sampling infrastructure</td>
</tr>
</tbody>
</table>

Inaccurate evaluation

<table>
<thead>
<tr>
<th>Performance</th>
<th>Capability</th>
<th>Latency</th>
</tr>
</thead>
</table>
PerfCnt-based Bug Isolation (PBI)

- Program Binary
- Statistical Debugging
- Hardware Perf. Events
- Counter Overflow Interrupt
- Predicates & 😊/😊

<table>
<thead>
<tr>
<th>Performance</th>
<th>Capability</th>
<th>Code Size</th>
<th>Change Hardware?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good (&lt;5% overhead)</td>
<td>Good</td>
<td>No Change</td>
<td>NO!</td>
</tr>
</tbody>
</table>

Production-Run Software Failure Diagnosis via Hardware Performance Counters, ASPLOS’13
Hardware Performance Counters

• Registers monitor hardware performance events
  – 1—8 registers per core
  – Each register can contain an event count
  – Large collection of hardware events
    • Instructions retired, TLB misses, cache misses, etc.

• Traditional usage
  – Hardware testing/profiling

How can this help diagnose software failures?
What to collect?

- Capability
- Reflect the root causes of many concurrency bugs
- Performance
- An existing hardware performance event
Which event can reflect root causes?

- L1 data cache cache-coherence events

It tracks which cache-coherence state (M/E/S/I) an instruction observes:

- Modified
- Exclusive
- Shared
- Invalid

Local read
Local write
Remote read
Remote write
Is cache-coherence event useful?

Thread 1

```c
ptr = malloc(SIZE);
...
if (!ptr){
    ReportOutOfMem();
    exit(1);
}
```

Mozilla

Thread 2

```c
free(ptr);
ptr=NULL;
```
Example (correct runs)

```
thread 1 (core 1)
  Modified
  ... ptr = malloc (SIZE);
  ...
  l: if (!ptr) {
    ReportOutofMem();
    exit(1);
  }
  ...
thread 2 (core 2)
  Invalid
  ...
  free (ptr);
  ptr=NULL;
  ...
```

Concurrency Bug from Apache HTTP Server
Example (failure run)

thread 1 (core 1)

Invalid

```c
ptr = malloc (SIZE);
...
```

if (!ptr) {
    ReportOutofMem();
    exit(1);
}

thread 2 (core 2)

Modified

```c
...
free (ptr);
ptr=NULL;
...
```

Predicate Table:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>😊</th>
<th>😞</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M_i</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>E_i</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S_i</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>I_i</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Concurrency Bug from Apache HTTP Server
Useful for Atomicity Violations

<table>
<thead>
<tr>
<th>Bug Type</th>
<th>FAILURE PREDICTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>WWR Violation</td>
<td>INVALID</td>
</tr>
<tr>
<td>RWR Violation</td>
<td>INVALID</td>
</tr>
<tr>
<td>RWWW Violation</td>
<td>INVALID</td>
</tr>
<tr>
<td>WRW Violation</td>
<td>SHARED</td>
</tr>
</tbody>
</table>
Useful for order violations

<table>
<thead>
<tr>
<th>Bug Type</th>
<th>FAILURE PREDICTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-too-early</td>
<td>EXCLUSIVE (!INVALID)</td>
</tr>
<tr>
<td>Read-too-late</td>
<td>INVALID</td>
</tr>
</tbody>
</table>
How to evaluate & sample?

Which performance events occur at a specific instruction?
Accessing performance counters

**INTERRUPT-BASED**

1. **User**
2. **Kernel**
3. **HW (PMU)**

**POLLING-BASED**

1. **User**
2. **Kernel**
3. **HW (PMU)**

- **Config**
- **PC, e**
- **Read**
- **Count**
### More details of counter access

```bash
perf record -event=<code> -c <sampling_rate><program_monitored>
```

<table>
<thead>
<tr>
<th>Log Id</th>
<th>APP</th>
<th>Core</th>
<th>Performance Event</th>
<th>Instruction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Httpd</td>
<td>2</td>
<td>0x140 (Invalid)</td>
<td>401c3b</td>
<td>decrement__refcnt</td>
</tr>
</tbody>
</table>
Beyond concurrency bugs

• Which event?
  – Branch taken/non-taken event

• How to evaluate & sample?
  – Performance counter overflow interrupt
PBI vs. CBI/CCI (Qualitative)

• Performance

  Sample in this region?

  CBI

  Are other threads sampling?

  CCI

  Are other threads sampling?

  PBI

• Diagnostic capability
  – Discontinuous monitoring (CCI/CBI)
  – Continuous monitoring (PBI)
  – PBI differentiates interleaving reads from writes
### Evaluation methodology

<table>
<thead>
<tr>
<th>Program</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cherokee</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mozilla-JS-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mozilla-JS-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mozilla-JS-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MySQL-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MySQL-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PBZIP2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1/100 sampling rate, ~1000 runs in total (failure:success~1:1)
## Diagnosis capability (w/ sampling)

<table>
<thead>
<tr>
<th>Program</th>
<th>CCI-Prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache-1</td>
<td>✓ top1</td>
</tr>
<tr>
<td>Apache-2</td>
<td>✓ top1</td>
</tr>
<tr>
<td>Cherokee</td>
<td>✗</td>
</tr>
<tr>
<td>FFT</td>
<td>✓ top1</td>
</tr>
<tr>
<td>LU</td>
<td>✓ top1</td>
</tr>
<tr>
<td>Mozilla-JS-1</td>
<td>✗</td>
</tr>
<tr>
<td>Mozilla-JS-2</td>
<td>✓ top1</td>
</tr>
<tr>
<td>Mozilla-JS-3</td>
<td>✓ top2</td>
</tr>
<tr>
<td>MySQL-1</td>
<td>-</td>
</tr>
<tr>
<td>MySQL-2</td>
<td>-</td>
</tr>
<tr>
<td>PBZIP2</td>
<td>✓ top1</td>
</tr>
</tbody>
</table>
## Diagnosis capability (w/ sampling)

<table>
<thead>
<tr>
<th>Program</th>
<th>CCI-Prev</th>
<th>PBI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache-1</td>
<td>✓ top1</td>
<td>✓ top1</td>
</tr>
<tr>
<td>Apache-2</td>
<td>✓ top1</td>
<td>✓ top1</td>
</tr>
<tr>
<td>Cherokee</td>
<td>✗</td>
<td>✓ top1</td>
</tr>
<tr>
<td>FFT</td>
<td>✓ top1</td>
<td>✓ top1</td>
</tr>
<tr>
<td>LU</td>
<td>✓ top1</td>
<td>✓ top1</td>
</tr>
<tr>
<td>Mozilla-JS-1</td>
<td>✗</td>
<td>✓ top1</td>
</tr>
<tr>
<td>Mozilla-JS-2</td>
<td>✓ top1</td>
<td>✓ top1</td>
</tr>
<tr>
<td>Mozilla-JS-3</td>
<td>✓ top2</td>
<td>✓ top1</td>
</tr>
<tr>
<td>MySQL-1</td>
<td>-</td>
<td>✓ top1</td>
</tr>
<tr>
<td>MySQL-2</td>
<td>-</td>
<td>✓ top1</td>
</tr>
<tr>
<td>PBZIP2</td>
<td>✓ top1</td>
<td>✓ top1</td>
</tr>
</tbody>
</table>
## Diagnosis capability (w/ sampling)

<table>
<thead>
<tr>
<th>Program</th>
<th>CCI-Prev</th>
<th>PBI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache-1</td>
<td>✓ top1</td>
<td>✓ top1-l</td>
</tr>
<tr>
<td>Apache-2</td>
<td>✓ top1</td>
<td>✓ top1-l</td>
</tr>
<tr>
<td>Cherokee</td>
<td>✗</td>
<td>✓ top1-l</td>
</tr>
<tr>
<td>FFT</td>
<td>✓ top1</td>
<td>✓ top1-E</td>
</tr>
<tr>
<td>LU</td>
<td>✓ top1</td>
<td>✓ top1-E</td>
</tr>
<tr>
<td>Mozilla-JS-1</td>
<td>✗</td>
<td>✓ top1-l</td>
</tr>
<tr>
<td>Mozilla-JS-2</td>
<td>✓ top1</td>
<td>✓ top1-l</td>
</tr>
<tr>
<td>Mozilla-JS-3</td>
<td>✓ top2</td>
<td>✓ top1-l</td>
</tr>
<tr>
<td>MySQL-1</td>
<td>-</td>
<td>✓ top1-S</td>
</tr>
<tr>
<td>MySQL-2</td>
<td>-</td>
<td>✓ top1-S</td>
</tr>
<tr>
<td>PBZIP2</td>
<td>✓ top1</td>
<td>✓ top1-l</td>
</tr>
</tbody>
</table>
## Diagnosis performance (overhead)

<table>
<thead>
<tr>
<th>Program</th>
<th>CCI-Prev</th>
<th>PBI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache-1</td>
<td>1.90%</td>
<td>0.40%</td>
</tr>
<tr>
<td>Apache-2</td>
<td>0.40%</td>
<td>0.40%</td>
</tr>
<tr>
<td>Cherokee</td>
<td>0.00%</td>
<td>0.50%</td>
</tr>
<tr>
<td>FFT</td>
<td>121%</td>
<td>1.00%</td>
</tr>
<tr>
<td>LU</td>
<td>285%</td>
<td>0.80%</td>
</tr>
<tr>
<td>Mozilla-JS-1</td>
<td>800%</td>
<td>1.50%</td>
</tr>
<tr>
<td>Mozilla-JS-2</td>
<td>432%</td>
<td>1.20%</td>
</tr>
<tr>
<td>Mozilla-JS-3</td>
<td>969%</td>
<td>0.60%</td>
</tr>
<tr>
<td>MySQL-1</td>
<td>-</td>
<td>3.80%</td>
</tr>
<tr>
<td>MySQL-2</td>
<td>-</td>
<td>1.20%</td>
</tr>
<tr>
<td>PBZIP2</td>
<td>1.40%</td>
<td>8.40%</td>
</tr>
</tbody>
</table>

Sequential-bug failure diagnosis results are also good!
Are we done?

1/100 sampling rate $\Rightarrow$ $\sim$100 failures required for diagnosis
How to do better than PBI?

<table>
<thead>
<tr>
<th>What to collect</th>
<th>How to collect</th>
<th>How to use the collected</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sampling</td>
<td></td>
</tr>
</tbody>
</table>

How to collect sufficient root-cause information in 1 run w/ small overhead?
How to do better than PBI?

What to collect | How to collect | How to use the collected
---|---|---
Biased sampling

Collect information @ likely root-cause locations
LXR – Last eXecution Record

• What to collect?
  – Last few branches right before failure
  – Last few cache-coherence events right before failures

• How to collect/maintain LXR?
  – Existing* hardware support!

<table>
<thead>
<tr>
<th>Performance</th>
<th>Capability</th>
<th>Code Size</th>
<th>Change Hardware?</th>
<th>Diagnosis Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good (&lt;5% overhead)</td>
<td>Good</td>
<td>Little Change</td>
<td>Simple Extension*</td>
<td>Short</td>
</tr>
</tbody>
</table>

Leveraging the Short-Term Memory of Hardware to Diagnose Production-Run Software Failures, ASPLOS’14
Last Branch Record (LBR)

• **Existing** hardware feature
  – Store recently taken branches
  – Circular buffer with 16 entries (Intel Nehalem)
  – **Negligible** overhead

<table>
<thead>
<tr>
<th>Branch Source Instruction Pointer</th>
<th>Branch Target Instruction Pointer</th>
</tr>
</thead>
</table>

Good performance
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
- **Negligible overhead (estimated)**

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

Good performance
Is LXR useful?

```
ptr = malloc(SIZE);
...
if (!ptr){
    ReportOutOfMem();
    exit(1);
}
Apache

free(ptr);
ptr=NULL;
```

Thread 1        Thread 2

```
print("%u", End);
print("%u", End-Start);
End=time();
FFT
```

```
Good diagnosis capability

Bugs have short error-propagation distance

LXR is sufficient for failure diagnosis

ConSeq: Detecting Concurrency Bugs through Sequential Errors, ASPLOS’11
```
## LXR vs PBI vs CBI/CCI

<table>
<thead>
<tr>
<th></th>
<th>Performance</th>
<th>Capability</th>
<th>Diagnosis Latency (#-failure-runs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LXR</td>
<td>&lt;5%</td>
<td>23/31</td>
<td>1~10 failures</td>
</tr>
<tr>
<td>PBI</td>
<td>&lt;5%</td>
<td>25/31</td>
<td>1000 failures</td>
</tr>
<tr>
<td>CBI/C CI</td>
<td>3% ~ 969%</td>
<td>18/31</td>
<td>1000 failures</td>
</tr>
</tbody>
</table>
Outline

Performance Latency

LXR PBI CCI

Capability

Performance
Conclusions & Future Work

Constraints/Requirements

Bugs

Techniques
Thanks!
Questions?

My collaborators
• Prof. Tom Reps
• Prof. Ben Liblit
• Prof. Michael Swift
• Prof. Karthikeyan Sankaralingam
• Prof. Darko Marinov

My students
• Wei Zhang (IBM Research)
• Guoliang Jin (N. Carolina State Univ.)
• Linhai Song
• Joy Arulraj
• Po-chun Chang