Production-Run Software Failure Diagnosis via Hardware Performance Counters

Joy Arulraj, Po-Chun Chang, Guoliang Jin and Shan Lu

THE UNIVERSITY of WISCONSIN MADISON

Motivation

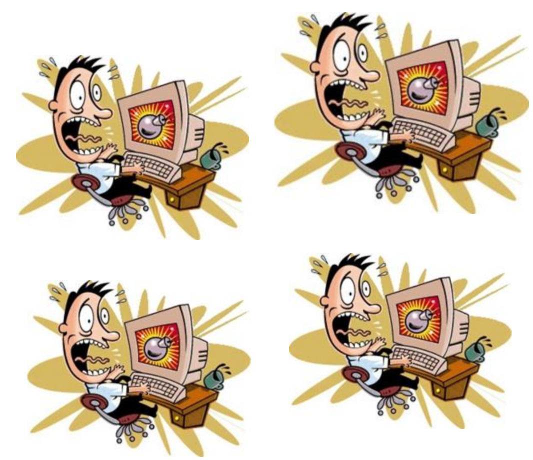
- Software inevitably fails on production machines

- These failures are widespread and expensive
  - Internet Explorer zero-day bug [2013]
  - Toyota Prius software glitch [2010]

These failures need to be diagnosed before they can be fixed !

Production-run failure diagnosis

- Diagnosing failures on client machines
  - Limited info from each client machine
  - One bug can affect many clients
  - Need to figure out root cause & patch quickly



Executive Summary

Use existing hardware support to diagnose widespread production-run failures with low monitoring overhead

Diagnosing a real world bug

- Sequential bug in print_tokens

```
int is_token_end(char ch){
if(ch == ‘\n’)
  return (TRUE);
else if(ch == ‘ ’)
// Bug: should return FALSE
  return (TRUE);
else
  return (FALSE);
}
```

Input:
Abc Def

Expected Output:
{Abc}, {Def}

Actual Output:
{Abc Def}

Diagnosing concurrency bugs

- Concurrency bug in Apache server

THREAD 1

```
decrement_refcnt(...)
{
        atomic_dec(           2 --> 1
        &obj->refcnt);

                              0
  ✗     if(!obj->refcnt)
         cleanup(obj);
}
```

THREAD 2

```
decrement_refcnt(...)
{

        atomic_dec(
        &obj->refcnt);        1 --> 0

                              0
        if(!obj->refcnt)
          cleanup(obj);
}
```

Requirements for failure diagnosis

- Performance
  - Low runtime overhead for monitoring apps
  - Suitable for production-run deployment

- Diagnostic Capability
  - Ability to accurately explain failures
  - Diagnose wide variety of bugs

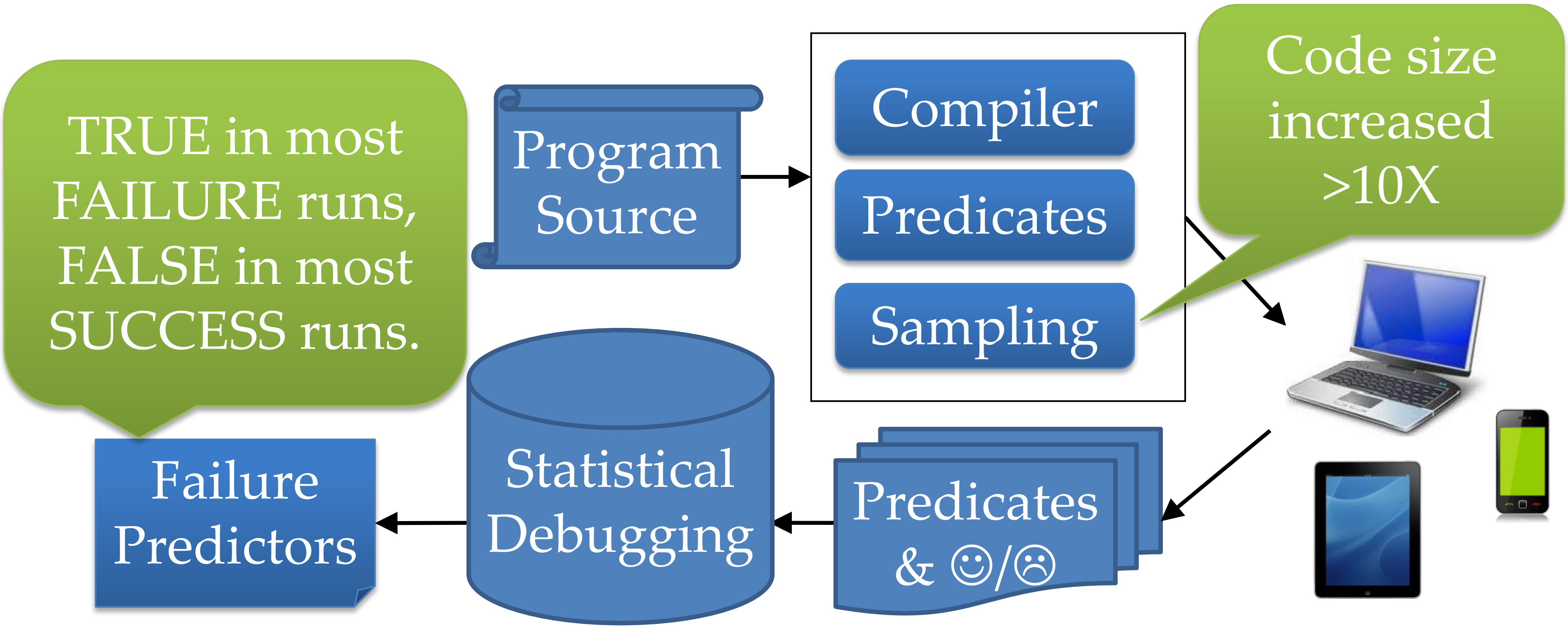
Existing work

Approach	Performance	Diagnostic Capability
FAILURE REPLAY	High runtime overhead OR	Manually locate root cause
BUG DETECTION	Non-existent hardware support	Many false positives

Cooperative Bug Isolation

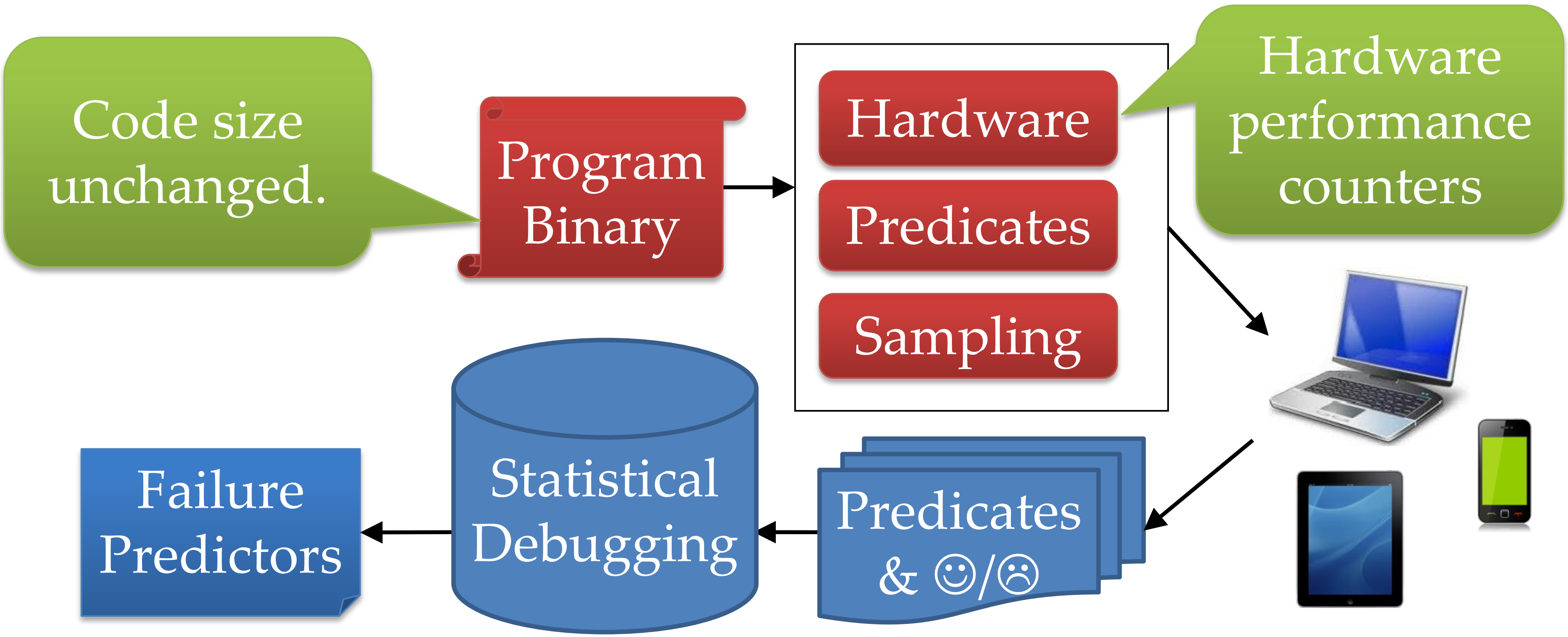
- Cooperatively diagnose production-run failures
  - Targets widely deployed software
  - Each client machine sends back information
- Uses sampling
  - Collects only a subset of information
  - Reduces monitoring overhead
  - Fits well with cooperative debugging approach

Cooperative Bug Isolation



Approach	Performance	Diagnostic Capability
CBI / CCI	>100% overhead for many apps (CCI)	Accurate & Automatic

Performance-counter based Bug Isolation



- Requires no non-existent hardware support
- Requires no software instrumentation

PBI Contributions

Approach	Performance	Diagnostic Capability
PBI	<2% overhead for most apps evaluated	Accurate & Automatic

- Suitable for production-run deployment
- Can diagnose a wide variety of failures
- Design addresses privacy concerns

Outline

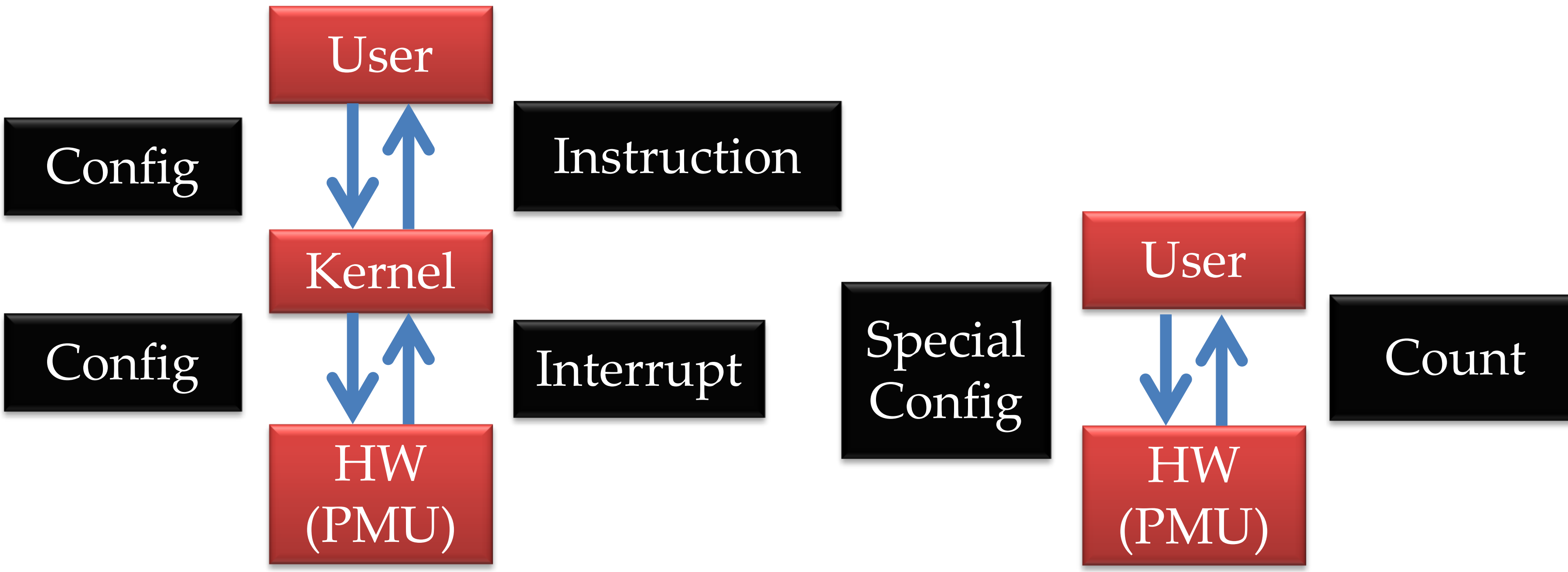
- Motivation
- Overview
- PBI
  - Hardware performance counters
  - Predicate design
  - Sampling design
- Evaluation
- Conclusion

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, L1 cache misses, etc.

Accessing performance counters

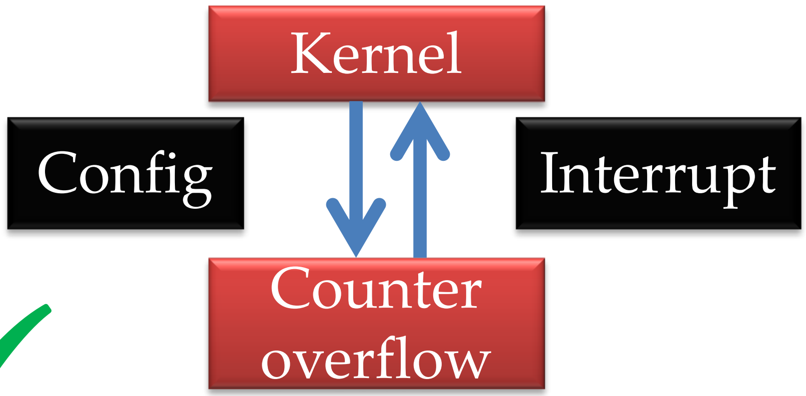
INTERRUPT-BASED	POLLING-BASED



How do we monitor which event occurs at which instruction using performance counters ?

Predicate evaluation schemes

INTERRUPT-BASED	POLLING-BASED




```
old = readCounter()
< Instruction C >
new = readCounter()
if(new > old)
    Event occurred at C
```

```
Interrupt at Instruction C
=> Event occurred at C
```

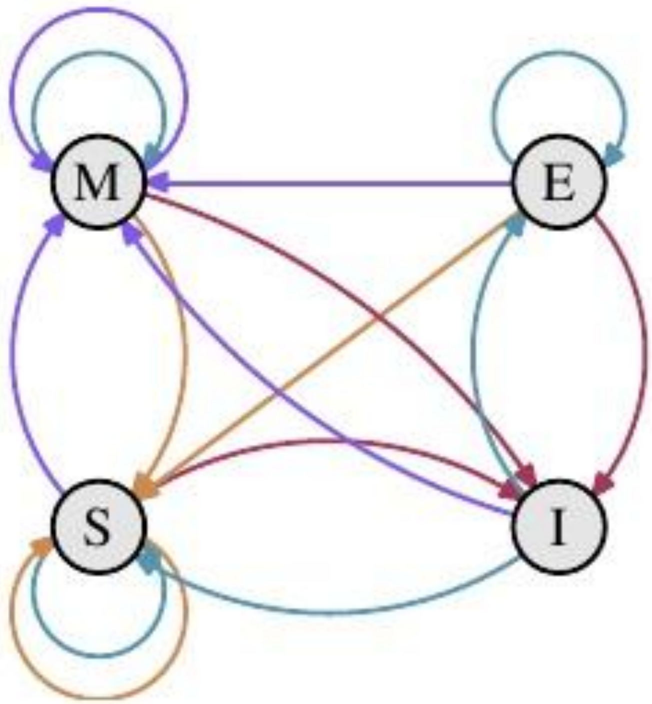
Natural fit for sampling	Requires instrumentation
More precise	Imprecise due to OO execution

Concurrency bug failures

How do we use performance counters to diagnose concurrency bug failures ?

- L1 data cache cache-coherence events

Modified
Exclusive
Shared
Invalid



Local read
Local write
Remote read
Remote write

Atomicity Violation Example

CORE 1 – THD 1

```
decrement_refcnt(...)
{
  apr_atomic_dec(
  &obj->refcnt);
```

Local Write

Modified

```
C:if(!obj->refcnt)
    cleanup_cache(obj);
}
```

Atomicity Violation Example

CORE 1 – THD 1

```
decrement_refcnt(...)
{
  apr_atomic_dec(
  &obj->refcnt);

C:if(!obj->refcnt)
    cleanup_cache(obj);
}
```

Invalid

CORE 2 - THD 2

```
decrement_refcnt(...)
{

  apr_atomic_dec(
  &obj->refcnt);

  if(!obj->refcnt)
   cleanup_cache(obj);

}
```

Remote Write

Atomicity Violation Bugs

THREAD INTERLEAVING	FAILURE PREDICTOR
WWR Interleaving	INVALID
RWR Interleaving	INVALID
RWW Interleaving	INVALID
WRW Interleaving	SHARED

Order violation

CORE 1 – MASTER THD

CORE 2 – SLAVE THD

Local Read

```
print("End",Gend)
C:print("Run",Gend-init)
```

Shared

```
Gend = time()
```

Remote Write

Order violation

CORE 1 – MASTER THD

CORE 2 – SLAVE THD

Local Read

```
print("End",Gend)
```

Exclusive

```
C:print("Run",Gend-init)
```

```
Gend = time()
```

PBI Predicate Sampling

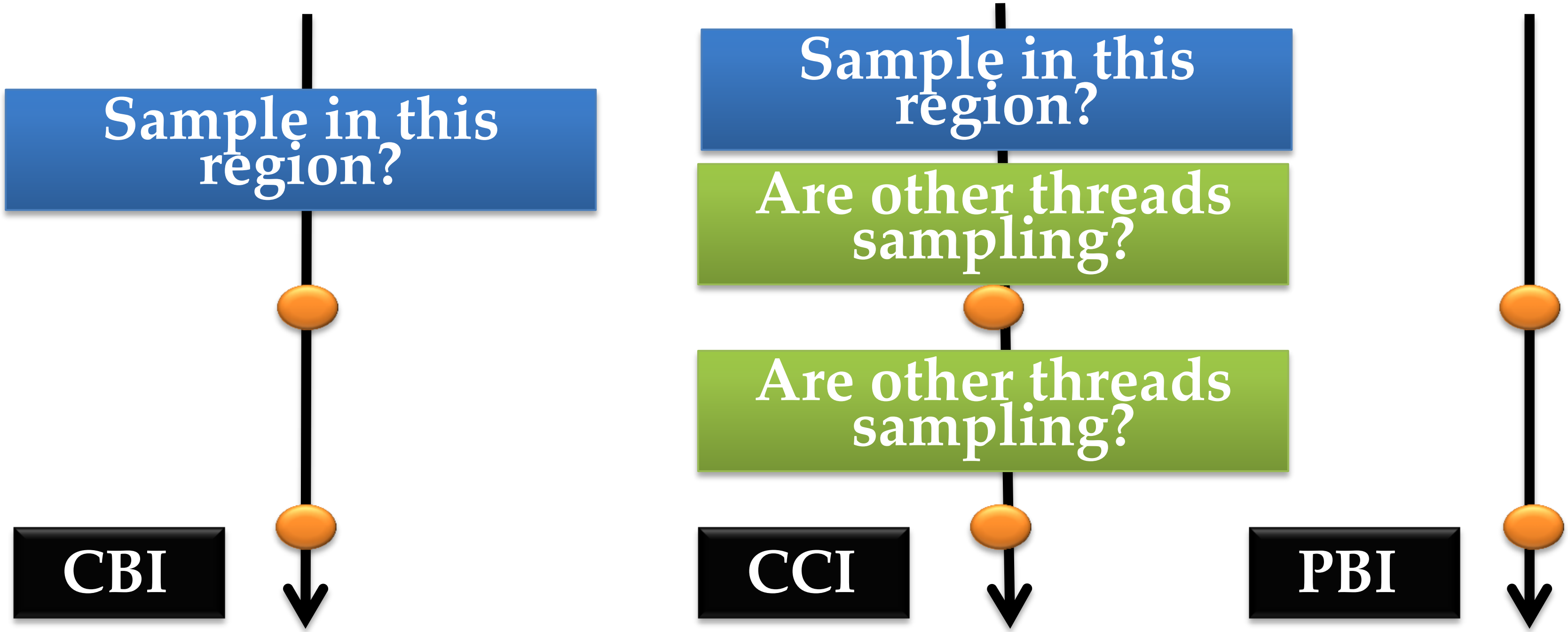
- We use Perf (provided by Linux kernel 2.6.31+)

```
perf record -event=<code> -c <sampling_rate>
<program monitored>
```

Log Id	APP	Core	Performance Event	Instruction	Function
1	Apache	2	0x140 (Invalid)	401c3b	decrement _refcnt

PBI vs. CBI/CCI (Qualitative)

- Performance



- Diagnostic capability
  - Discontinuous monitoring (CCI/CBI)
  - Continuous monitoring (PBI)

Outline

- Motivation
- Overview
- PBI
  - Hardware performance counters
  - Predicate design
  - Sampling design
- Evaluation
- Conclusion

Methodology

- 23 real-world failures
  - In open-source server, client, utility programs
  - All CCI benchmarks evaluated for comparison
- Each app executed 1000 runs (400-600 failure runs)
  - Success inputs from standard test suites
  - Failure inputs from bug reports
  - Emulate production-run scenarios
- Same sampling settings for all apps

Evaluation

Program	Diagnostic Capability		
	PBI	CCI-P	CCI-H
Apache1	✓	✓	✓
Apache2	✓	✓	✓
Cherokee	✓	X	✓
FFT	✓	✓	X
LU	✓	✓	X
Mozilla-JS1	✓	X	✓
Mozilla-JS2	✓	✓	✓
Mozilla-JS3	✓	✓	✓
MySQL1	✓	-	-
MySQL2	✓	-	-
PBZIP2	✓	✓	✓

Diagnostic Capability

Program	Diagnostic Capability		
	PBI	CCI-P	CCI-H
Apache1	✓(Invalid)	✓	✓
Apache2	✓ (Invalid)	✓	✓
Cherokee	✓ (Invalid)	X	✓
FFT	✓ (Exclusive)	✓	X
LU	✓ (Exclusive)	✓	X
Mozilla-JS1	✓ (Invalid)	X	✓
Mozilla-JS2	✓ (Invalid)	✓	✓
Mozilla-JS3	✓ (Invalid)	✓	✓
MySQL1	✓ (Invalid)	-	-
MySQL2	✓(Shared)	-	-
PBZIP2	✓(Invalid)	✓	✓

Diagnostic Capability

Program	Diagnostic Capability		
	PBI	CCI-P	CCI-H
Apache1	✓	✓	✓
Apache2	✓	✓	✓
Cherokee	✓	X	✓
FFT	✓	✓	X
LU	✓	✓	X
Mozilla-JS1	✓	X	✓
Mozilla-JS2	✓	✓	✓
Mozilla-JS3	✓	✓	✓
MySQL1	✓	-	-
MySQL2	✓	-	-
PBZIP2	✓	✓	✓

Diagnostic Capability

Program	Diagnostic Capability		
	PBI	CCI-P	CCI-H
Apache1	✓	✓	✓
Apache2	✓	✓	✓
Cherokee	✓	X	✓
FFT	✓	✓	X
LU	✓	✓	X
Mozilla-JS1	✓	X	✓
Mozilla-JS2	✓	✓	✓
Mozilla-JS3	✓	✓	✓
MySQL1	✓	-	-
MySQL2	✓	-	-
PBZIP2	✓	✓	✓

Diagnostic Overhead

Program	Diagnostic Overhead		
	PBI	**CCI-P**	**CCI-H**
Apache1	0.40%	1.90%	1.20%
Apache2	0.40%	0.40%	0.10%
Cherokee	0.50%	0.00%	0.00%
FFT	1.00%	121%	118%
LU	0.80%	285%	119%
Mozilla-JS1	1.50%	800%	418%
Mozilla-JS2	1.20%	432%	229%
Mozilla-JS3	0.60%	969%	837%
MySQL1	3.80%	-	-
MySQL2	1.20%	-	-
PBZIP2	8.40%	1.40%	3.00%

Diagnostic Overhead

Program	Diagnostic Overhead		
	PBI	CCI-P	CCI-H
Apache1	0.40%	1.90%	1.20%
Apache2	0.40%	0.40%	0.10%
Cherokee	0.50%	0.00%	0.00%
FFT	1.00%	121%	118%
LU	0.80%	285%	119%
Mozilla-JS1	1.50%	800%	418%
Mozilla-JS2	1.20%	432%	229%
Mozilla-JS3	0.60%	969%	837%
MySQL1	3.80%	-	-
MySQL2	1.20%	-	-
PBZIP2	8.40%	1.40%	3.00%

Conclusion

- Low monitoring overhead
- Good diagnostic capability
- No changes in apps
- Novel use of performance counters

PBI will help developers diagnose production-run software failures with low overhead

Thanks !