Accurate, Efficient, and Adaptive Calling Context Profiling

Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, Jong-Deok Choi

Presented by
Brian Norris
Overview

Earliest of four “calling context” papers we've studied
  Bond and McKinley, “Probabilistic Calling Context” (2007)
  Sumner, Zheng, Weeratunge, and Zhang, “Precise Calling Context Encoding” (2010)

All reference this paper
  All have some criticism for this paper
Outline

- Introduction
- Existing Approaches
- Our Approach: Adaptive Bursting
- Results
- Related Work
- Conclusion and Future Work
Motivation

What is a calling context?
- Methods that are on the stack when an event happens

```java
java.lang.ArrayIndexOutOfBoundsException: 3 >= 3
at java.util.Vector.elementAt(Vector.java:427)
at junit.samples.VectorTest.testElementAt(Vector
```

Call Trace:
- [] __handle_sysrq+0x58/0xc6
- [] write_sysrq_trigger+0x23/0x29
- [] vfs_write+0xb6/0xe2
- [] sys_write+0x3c/0x62
- [] syscall_call+0x7/0xb

Applications of calling context information
- Optimizations based on profiling: inlining, devirtualization, etc..
- Program understanding
  - Large server applications have a complex method-level profile
- Debugging
Examples

- Call Tree (CT)
- Call Graph (CG)
- Calling-Context Tree (CCT)

- **Call Tree:** complete calling context info, but huge tree
- **Call Graph:** no context information
- **Calling Context Tree:** merges identical child nodes of the same parent node → *much* smaller than Call Tree
Collecting Calling Context Profile

Existing approaches incur high-overhead
- OO program: highly *interprocedural*
- Exhaustive: 50x overhead?

New approach
- Reduce overhead while maintaining high accuracy
- Use an adaptive scheme:
  - Bursty mode sampling
  - Disable bursts when similar contexts are found
  - Re-enable bursts when accuracy could be decreased
Contributions

- Improved:
  - Efficiency
  - Accuracy
  - Portability (i.e., doesn't rely on HW features)
- New metric (overlap vs. hot-edge coverage)
- Rigorous comparison of efficiency and accuracy
Outline

- Introduction
- Existing Approaches
- Our Approach: Adaptive Bursting
- Results
- Related Work
- Conclusion and Future Work
Building CCT: Exhaustive Approach

- Capture all calls and returns

- High instrumentation cost:
  - Authors' experiments indicate 50 times slowdown based on JVMPi
Building CCT: Sampled Stack Walking

At each sampling point, walk the full stack back
- What about long method calls?
- Stack-walking is quite efficient (at 10 ms interval)
  - But on some platforms, the interval cannot be smaller
  - Sacrifice accuracy
Building CCT: Bursting

At each sampling point, capture a burst of method calls and returns

Useful to build call graph profiles, not useful for CCT

BI: Burst Interval
SI: Sampling Interval
Building CCT: Static Bursting

Perform stack walking before each burst

- Gets expensive with longer burst intervals or shorter sampling intervals for a precise CCT

BI: Burst Interval
SI: Sampling Interval
Outline

- Introduction
- Existing Approaches
- Our Approach: Adaptive Bursting
- Results
- Related Work
- Conclusion and Future Work
Adaptive Bursting: Reduce Redundant Bursts

- Control flow is highly repetitive (e.g. loops) → bursts are redundant

- Selectively disable previously sampled calling contexts

- Call stack information can serve as a good signature → a hash of methods on the stack at the beginning of the burst

- Use a history table to record if similar burst has occurred earlier
Overview of Adaptive Bursting

sample point reached

Build stack signature and check in the history table

disable enable

reenable mechanism

N
skip the feedback?

Y

Weight compensation/feedback?

perform the burst with weight adjustment

perform the burst

feedback

history table

hash

CCT
Adaptive Bursting: Weight Compensation

Disabling redundant bursts loses CCT edge weights

Statistically reenable some of the disabled bursts, with a Reenable Ratio \((RR)\) between 0 and 1.

The probability a burst is reenabled is \(RR\). Every counter value added to the CCT is multiplied by \(1/RR\).

Ex. \(RR = 0.25\), enable 1 per 4 disabled bursts, multiply each counter by 4.
Weight Compensation Example

Assume sig1=sig2=sig3=sig4=sig5

Static Bursting

Adaptive Bursting w/o reenable

Adaptive Bursting w/ reenable ratio=0.5
Outline

- Introduction
- Existing Approaches
- Our Approach: Adaptive Bursting
- Results
- Related Work
- Conclusion and Future Work
Benchmarks & Setup

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESCRIPTION</th>
<th>PLAT FORM</th>
<th>Call Graph</th>
<th>CCT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td># nodes</td>
<td># edges</td>
</tr>
<tr>
<td>checkit</td>
<td>jvm98 - check program</td>
<td>x86</td>
<td>988</td>
<td>1827</td>
</tr>
<tr>
<td>compress</td>
<td>jvm98 - Modified Lempel -Ziv method</td>
<td>x86</td>
<td>721</td>
<td>1227</td>
</tr>
<tr>
<td>db</td>
<td>jvm98 - database simulation</td>
<td>x86</td>
<td>744</td>
<td>1310</td>
</tr>
<tr>
<td>ipsixql</td>
<td>Persistent XML -database</td>
<td>x86</td>
<td>802</td>
<td>1330</td>
</tr>
<tr>
<td>jack</td>
<td>jvm98 - Java Parser Generator</td>
<td>x86</td>
<td>987</td>
<td>1996</td>
</tr>
<tr>
<td>javac</td>
<td>jvm98 - java compiler</td>
<td>x86</td>
<td>1505</td>
<td>4144</td>
</tr>
<tr>
<td>specjbb</td>
<td>Java business application</td>
<td>x86</td>
<td>2467</td>
<td>5368</td>
</tr>
<tr>
<td>jess</td>
<td>jvm98 - Expert Shell System</td>
<td>x86</td>
<td>1101</td>
<td>2106</td>
</tr>
<tr>
<td>kawa</td>
<td>Java -based Scheme system</td>
<td>x86</td>
<td>2454</td>
<td>5496</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>jvm98 - decompress audio files</td>
<td>x86</td>
<td>898</td>
<td>1516</td>
</tr>
<tr>
<td>JAS</td>
<td>SpecJAppServer2004:3 tier java server</td>
<td>AIX</td>
<td>6918</td>
<td>14597</td>
</tr>
</tbody>
</table>

- **Two configurations:** Windows/Sun JVM, AIX/J9-3tier
- **Sampling Interval=10ms, Burst Interval=0.2ms.**
- **Re-enable Ratio=0.05, History Table 2048 entries.**
Measuring the Accuracy of Calling Context

- **Degree of Overlap**
  - Focus on measuring the completeness of a CCT against the complete CCT

- **Hot-edge Coverage**
  - Focus on the coverage of hot edges (above a threshold)

- Formal definitions explained in the paper.
Results—Degree of Overlap

Average: stack walk (49.8%), adaptive (68.8%), adaptive with re-enable (85.2%), static burst (91.4%).
Results—Hot-edge Coverage

- Average: stack walk (52.9%), adaptive (79.1%), adaptive with re-enable (88.2%), static burst (88.1%).
**Results—Slowdown**

- **Average:** stack walk (<1%), adaptive (14.8%), adaptive w/ reenable (18.8%), static burst (117%)

- **JVMPi is inefficient**
Results—Percentage of Disabled Bursts

Both approaches disabled most bursts
Reenablement only adds small % of bursts ($RR = 5\%$)
Summary of Results

- JVMPI-based adaptive bursting
  - A modest slowdown
  - 85% degree of overlap
  - 88% hot-edge coverage

- Sampled stack walking
  - Negligible slowdown
  - Around 50% degree of overlap and hot-edge coverage
  - Bad for large server benchmark JAS (0% coverage)

- Static bursting
  - Accuracy is close to adaptive bursting (<6%)
  - Slowdown 6 times higher
Outline

- Introduction
- Existing Approaches
- Our Approach: Adaptive Bursting
- Results
- Related Work
- Conclusion and Future Work
Related Work

- **Exhaustive approach:** Ammons et al. [PLDI-97], Spivey [SPE-04],

- **Sampling-based approach:** Arnold & Sweeney [IBM TR-00], Froyd et al. [ICS-05], Whaley [Java Grande-00]

- **Context Sensitive Inlining:** Hazelwood & Grove [CGO-03]
Outline

- Introduction
- Existing Approaches
- Our Approach: Adaptive Bursting
- Results
- Related Work
- Conclusion, Future Work, Criticism/Discussion
Conclusion

- Novel, efficient construction of accurate CCT
  - Accuracy: 80% to 90%.
  - Moderate overhead with JVMPI
  - ~6% overhead observed with JVM-based instrumentation.

- Formal definitions of two metrics for evaluating CCT accuracy
  - Degree of overlap
  - Hot-edge coverage

- Extensive measurements using a large number of benchmark programs, including a very large commercial J2EE Java application
Future Work

- Further reduce the overhead
  - Better instrumentation (alternatives to JVMPI)
  - M. Bond suggested using PCC to identify history

- Call site information

- Applications: context sensitive optimizations
  - Lock contention analysis?
  - Object allocation analysis
  - Method inlining
Criticism/Discussion

- Cold path coverage?
  - Insufficient cold path coverage
  - Rare bugs can't be discovered

- Overlap vs. Hot Edge Coverage: which is better?