CS263
Program Profiling
Profiling

• “the formal summary or analysis of data representing distinctive features or characteristics” (American Heritage Dictionary)
  – Program profiling: analyzing the execution characteristics of code to extract and summarize its behavior

• Offline vs online
  – Offline – Collection time doesn’t matter
  – Online – Slow-down is an important factor

• Exhaustive vs sample-based
  – Sampling is estimation. The difference between a sampled profile and an exhaustive profile is the accuracy/error measure
  – Sampling is commonly used online where time matters
    • Can be used offline if inaccuracy can be tolerated
Why Profile?

• To characterize program behavior
  – Understand how programs behave
  – Guide tool, runtime, system (hw/sw) design
  – Program test generation

• To capture specific or unusual behavior
  – Security attacks, intrusion detection, bugs, test coverage
  – Logging

• Track performance regressions
Why Profile?

• To improve performance
  – Time different parts of the program to find out where time is being spent
    • 80/20 rule – identify the 20 and focus your optimization energy
    • By hand optimization
    • Automatic (compiler or runtime) feedback-directed optimization
      – Target hot code
      – Inlining and unrolling
      – Code scheduling and register allocation

  – Increasingly important for speculative optimization
    • Hardware trends → simplicity & multiple contexts
    • Less speculation in hardware, more in software
What to Profile

• Individual instructions
  – Memory accesses (allocations/deletions, loads/stores)
    • Lends insight into caching, paging, garbage collection, bugs & more
  – If individual instruction detail isn’t needed: capture basic blocks
    • Estimate bb’s by recording branches and their direction
    • Lends insight into branch miss overhead
What to Profile

• Individual instructions
  – Memory accesses (allocations/deletions, loads/stores)
    • Lends insight into caching, paging, GC, races, bugs & more
  – If individual instruction detail isn’t needed: capture basic blocks
    • Estimate bb’s by recording branches and their direction
    • E.g: lends insight into branch miss overhead

• Paths
• Function invocations and call sites
• Memory allocation, GC time
• Interfaces (ABI, APIs to other components, foreign function)
• Resource use
  – CPU, Network, disk, other I/O
  – Runtime services (compiler/interp, GC, runtime, OS)
• User interactivity
Instrumentation vs Event Monitoring

- Instrumentation: Insert code into the code of a program
  - The additional code executes interleaved with program code
  - To collect information about the program code activity

- Can perturb the behavior that it is trying to measure
Instrumentation vs Event Monitoring

• Instrumentation: Insert code into the code of a program
  – The additional code executes interleaved with program code
  – To collect information about the program code activity

• Event monitoring
  – Profiling external to the executing program
  – Output timestamps, upon OS or runtime activity, around program
  – Record of operations (timings, counts) in runtime that execute concurrently with the executing program, yet independent of it
    • Garbage collection activity
    • Accesses to the OS
    • Accesses to libraries (e.g. GUI, cloud SDK, web services)
  – Hardware performance counters/monitors (HPMs)

• Can perturb the behavior that it is trying to measure
Adaptive Optimization

• Sample the system (lightweight)
• Predict future behavior based on past behavior
  – Does the past predict the future?
• Determine if prediction can amortize the cost of applying more optimization overhead
Adaptive Optimization

• Sample the system (lightweight)
• Predict future behavior based on past behavior
  – Does the past predict the future?
• Determine if prediction can amortize the cost of applying more optimization overhead

• Sampling to find hotspots or problem methods
  – Periodically record the top N methods on the runtime stack
    • Finding the right period and a value for N is tricky!
  – **Use HPMs** to identify methods that are causing stalls in the hardware...
    Careful, calling HPM services increments counters
    • Branch mispredictions
    • Cache misses
  – Very low overhead (< 2%)
Hardware Performance Monitors/Counters

• Libraries provide access to hardware collected HPMs

• Other types of sampling
  – Random
  – Periodic
  – Phase
Time Varying Behavior of Programs

Program Behavior changes over time

- Different behavior during different parts of execution
- Many programs execute as a series of phases possibly with recurring patterns
- Capture via basic block profiles for fixed number of instructions=vector
  - Compare counts across vectors for similarity
Phase Aware Remote Profiling

- Extant approaches (random/periodic sampling)
  - Do not consider time-varying and repeating behavior
  - Collect redundant information
Phase Aware Remote Profiling

- Extant approaches (random/periodic sampling)
  - Do not consider time-varying and repeating behavior
  - Collect redundant information

- Our approach: Sample according to **phase** behavior
Results

- 50-75% reduction in overhead (over periodic and random sampling)
Sampling Interactive Sessions

- A period of user interaction: Each application has a specific pattern
Interactive Sessions

- A period of user interaction: Each application has a specific pattern
Exhaustive Path Profiling (Instrumentation)

Thanks to Mike Bond (Ohio State) for his presentations of PEP and Continuous Path/Edge Profiling for these slides [CGO/MICRO 2005] on path profiling and its optimization.

- Processors need long instruction sequences
- Programs have branches
Why path profiling?

- Compiler identifies hot paths across multiple basic blocks
Why path profiling?

- Compiler identifies hot paths across multiple basic blocks
  - Forms and optimizes “traces” -- like superinstructions!
Why path profiling?

- Compiler identifies hot paths across multiple basic blocks
  - Speculative in that control may exit the trace early
  - Requires special case handling/overhead
Ball-Larus path profiling

- 4 paths $\rightarrow [0, 3]$
Ball-Larus path profiling

- 4 paths $\rightarrow [0, 3]$
- Each path sums to a unique integer
Ball-Larus path profiling

- 4 paths $\rightarrow [0, 3]$ 
- Each path sums to unique integer

Path 0
Ball-Larus path profiling

• 4 paths $\rightarrow [0, 3]$
• Each path sums to unique integer

Path 0
Path 1
Ball-Larus path profiling

- 4 paths → [0, 3]
- Each path sums to unique integer

Path 0
Path 1
Path 2
Ball-Larus path profiling

- 4 paths $\rightarrow [0, 3]$
- Each path sums to unique integer

Path 0
Path 1
Path 2
Path 3
Ball-Larus path profiling

- $r$: path register
  - Computes path number
- `count`: Stores path frequencies

```
\text{count}[r]++
```
Ball-Larus path profiling

- $r$: path register
  - Computes path number

- **count:**
  - Stores path frequencies
  - Array by default
  - Too many paths?
    - Hash table
    - High overhead

\[
\begin{align*}
  r &= 0 \\
  r &= r + 2 \\
  r &= r + 1 \\
  \text{count}[r] &= ++
\end{align*}
\]
Optimizing Path Profiling

Computes path

Updates path profile

\[ r = r + 2 \]

\[ r = 0 \]

\[ r = r + 1 \]

\[ \text{count}[r]++ \]
Optimizing Path Profiling

* Where have all the cycles gone?
Optimizing Path Profiling

All-the-time instrumentation

Sampling (piggybacks on existing mechanism)
Optimizing Path Profiling

All-the-time instrumentation

Overhead: 30% → 2%
[Bond et al. 2005]

Sampling (piggybacks on existing mechanism)
Profile-guided profiling

- Existing edge profile informs path profiling
  - Profile some initially
    - Quite fast to profile edges
    - Can be sample based
    - Just need to determine which branch edges are taken more frequently

\[\text{freq} = 30 \quad \text{freq} = 70 \]
\[\text{freq} = 90 \quad \text{freq} = 10\]
Profile-guided profiling

- Existing edge profile informs path profiling
- Assign zero to hotter edges
  - No instrumentation
Calling Context Profiling

A calls B calls C calls D calls E
Versus
A calls D calls E

Two different contexts for E
May provide insight into errors/bugs in E
Calling Context Profiling

• Provide more detail for debugging and analysis
  – Race conditions, locking problems
  – Memory management (with and without GC)
  – General bugs
  – Improve static analysis
    • Better information
    • Better precision

• Critical in modern software:
  – Intensive code reuse (e.g., frameworks)
  – Many small methods
  – Highly dynamic behavior

• Identify new behavior (new contexts)
  – Testing vs production
## Context Is Nontrivial

<table>
<thead>
<tr>
<th>Program</th>
<th>API calls</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Call sites</td>
<td>Distinct contexts</td>
<td></td>
</tr>
<tr>
<td>antlr</td>
<td>4,184</td>
<td>128,627</td>
<td></td>
</tr>
<tr>
<td>bloat</td>
<td>3,306</td>
<td>600,947</td>
<td></td>
</tr>
<tr>
<td>chart</td>
<td>2,335</td>
<td>202,603</td>
<td></td>
</tr>
<tr>
<td>eclipse</td>
<td>9,611</td>
<td>226,020</td>
<td></td>
</tr>
<tr>
<td>fop</td>
<td>2,225</td>
<td>37,710</td>
<td></td>
</tr>
<tr>
<td>hsqldb</td>
<td>947</td>
<td>16,050</td>
<td></td>
</tr>
<tr>
<td>jython</td>
<td>1,830</td>
<td>628,048</td>
<td></td>
</tr>
<tr>
<td>luindex</td>
<td>654</td>
<td>102,556</td>
<td></td>
</tr>
<tr>
<td>lusearch</td>
<td>507</td>
<td>905</td>
<td></td>
</tr>
<tr>
<td>pmd</td>
<td>1,890</td>
<td>847,108</td>
<td></td>
</tr>
<tr>
<td>xalan</td>
<td>1,530</td>
<td>17,905</td>
<td></td>
</tr>
</tbody>
</table>
Example: Dynamic data race detector

Thread A
write x
unlock m
read x

Thread B
lock m
write x
Example: dynamic data race detector

Thread A
- write x
- unlock m
- read x

Thread B
- lock m
- write x

race!
Reporting a race

Thread A
- write x
- unlock m
- read x

Thread B
- lock m
- write x

race!

T@A loc1
T’@B loc2
T’’@A loc3
Reporting a race

Thread A
- write x
- read x
- AbstractDataTreeNode.indexOfChild():426
- race!
- T’’@A loc3

Thread B
- AbstractDataTreeNode.storeStrings():536
- write x
- race!
- T’@B loc2

lock m
Full stack traces

AbstractDataTreeNode.indexOfChild():426
AbstractDataTreeNode.childAtOrNull():212
DeltaDataTree.lookup():666
ElementTree.includes():528
Workspace.getResourceInfo():1135
Resource.getResourceInfo():973
Project.hasNature():479
JavaProject.hasJavaNature():224
JavaProject.computeExpandedClasspath():430
JavaProject.getExpandedClasspath():1444
...
EclipseStarter.run():376
...

AbstractDataTreeNode.storeStrings():536
DataTreeNode.storeStrings():343
AbstractDataTreeNode.storeStrings():541
DataTreeNode.storeStrings():343
...
ElementTree.shareStrings():706
SaveManager.shareStrings():1154
...
StringPoolJob.shareStrings():124
...
Worker.run():76
...

read x
lock m
write x
T'@B loc2

T'@A loc3
How hard is this?

Race discovered here

EASY

AbstractDataTreeNode.indexOfChild():426
AbstractDataTreeNode.childAtOrNull():212
DeltaDataTree.lookup():666
ElementTree.includes():528
Workspace.getResourceInfo():1135
Resource.getResourceInfo():973
Project.hasNature():479
JavaProject.hasJavaNature():224
JavaProject.computeExpandedClasspath():430
JavaProject.getExpandedClasspath():1444
...
EclipseStarter.run():376
...

AbstractDataTreeNode.storeStrings():536
DataTreeNode.storeStrings():343
AbstractDataTreeNode.storeStrings():541
DataTreeNode.storeStrings():343
...
ElementTree.shareStrings():706
SaveManager.shareStrings():1154
...
StringPoolJob.shareStrings():124
...
Worker.run():76
...

write x

lock m

T''@A loc3

T''@B loc2
How hard is this?

AbstractDataTreeNode.indexOfChild():426
AbstractDataTreeNode.childAtOrNull():212
DeltaDataTree.lookup():666
ElementTree.includes():528
Workspace.getResourceInfo():1135
Resource.getResourceInfo():973
Project.hasNature():479
JavaProject.hasJavaNature():224
JavaProject.computeExpandedClasspath():430
JavaProject.getExpandedClasspath():1444
...
EclipseStarter.run():376
...

Previously recorded information

T’@A loc3

T’@B loc2

write x

lock m

read x
Challenge

• Many events *might* need context information *e.g.*, *race detector: every read and write (!)*

• Existing approaches
  – Walk the stack: up to 100X slowdown
  – Build calling context tree: 2-3X, plus space
Goal

Efficient context sensitivity for dynamic bug detectors

- Compact representation of calling contexts
- Fast correct execution
- Print out stack trace when bug detected
Calling Context Profiling

Probabilistic Calling Context
Bond and McKinley OOPSLA 07

- Represent a calling context in 1 word – **PCC value**
- Computed online, low overhead – **<5%**

Breadcrumbs: Efficient Context Sensitivity for Dynamic Bug Detection Analyses
Bond, Baker, and Guyer PLDI 10

- Efficiently decode a PCC value when needed - **10-20%**
  --- make any dynamic bug detector context sensitive
  --- threshold/sample based (focus on cold contexts)
Sample-based Instrumentation

• Turn on and off instrumentation dynamically
  • Challenge: when to turn instrumentation on and off
Sample-based Instrumentation

• Turn on and off instrumentation dynamically
  – Challenge: when to turn instrumentation on and off

  – Why is this important to do?

  – How: **Switching between (un-)instrumented versions**
    • Via **code patching**: Ephemeral Instrumentation, DynInst, IBM Java Developer Kit
      – Have two versions of the methods (or code blocks) you want to instrument
      – In the uninstrumented version, put a patch point at entry
        » Dummy instruction large enough to hold a jump
      – Overwrite (patch) the entry point to instrument
      – “Undo” patch to turn off instrumentation
    • Via **recompilation and on-stack replacement**
      – Via **code copying** (today’s paper)
Profiling Tools

• Of binaries (independent of language)
  – Pin, Dynamo
  – Valgrind
  – gprof (call graph and function timings)

• Of programs (language specific)
  – Java – JVMP/TI, JProfiler, many others, GCSpy
  – Ruby – ruby-prof
  – Python – cprofile

• HPMs
  – Library support: PAPI
  – OS Integration: PerfMon, OProfile, XenOProf