Predicting Program Phases and Defending against Side-Channel Attacks using Hardware Performance Counters

Junaid Nomani and Jakub Szefer

Computer Architecture and Security Laboratory
Yale University

junaid.nomani@yale.edu
jakub.szefer@yale.edu

4\textsuperscript{th} Workshop on Hardware and Architectural Support for Security and Privacy
14\textsuperscript{th} June 2014
Outline

- Motivation & Approach
- Side-Channel Example
- Program Behavior and Phases
- Hardware Performance Counters
  - Estimating Interference
- New Scheduler Architecture
  - Machine Learning Module
- Overhead
- Results
Motivation

- Side channel attacks require interference between programs
  
  *Two programs must share hardware functional units to interfere with another*

- Program behavior, or program phases, correlate with hardware functional units used

- Can reduce interference by scheduling interfering programs away from each other so they do not share hardware functional units
Cache Side Channel Example

• Sharing of cache by attacker (A) and victim (V) leads to potential side-channel attacks

• Scheduling the attacker and victim, when they are doing memory accesses, on separate cores means they don’t share caches

• Non-sharing of cache mitigates side-channels
Cache Side Channel Example

CPU 1

Cache

No collisions between A and V

CPU 2

Cache

Cache collisions between A and V

CPU 1

Cache

A

V

Our Approach

Schedule programs based on predicted program behavior in order to prevent the interference required for side channel attacks
Background: Program Behavior

- Programs tend to exhibit repeating patterns of behavior
  - Program Phases
- Thus by determining past behavior can predict future behavior

[Diagram showing program phases with time on the x-axis and categories like Integer, FP, Logical, Memory, Branches on the y-axis]
Background: Hardware Performance Counters

- Also known as Hardware Performance Monitors
- Can determine current behavior of programs by counting events
- Usually 2 to 4 counters per CPU
- Many events can be counted, e.g. from Intel:

<table>
<thead>
<tr>
<th>Event Num.</th>
<th>Event Mask Mnemonic</th>
<th>Umask Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3CH</td>
<td>UnHalted Core Cycles</td>
<td>00H</td>
<td>Unhalted core cycles</td>
</tr>
<tr>
<td>3CH</td>
<td>UnHalted Reference Cycles</td>
<td>01H</td>
<td>Unhalted reference cycles</td>
</tr>
<tr>
<td>C0H</td>
<td>Instruction Retired</td>
<td>00H</td>
<td>Instruction retired</td>
</tr>
<tr>
<td>2EH</td>
<td>LLC Reference</td>
<td>4FH</td>
<td>Longest latency cache references</td>
</tr>
<tr>
<td>2EH</td>
<td>LLC Misses</td>
<td>41H</td>
<td>Longest latency cache misses</td>
</tr>
<tr>
<td>C4H</td>
<td>Branch Instruction Retired</td>
<td>00H</td>
<td>Branch instruction at retirement</td>
</tr>
<tr>
<td>C5H</td>
<td>Branch Misses Retired</td>
<td>00H</td>
<td>Mispredicted Branch Instruction at retirement</td>
</tr>
</tbody>
</table>
Performance & Interference

- Observed performance changes as programs interfere

- Scheduling of programs affects interference, e.g. mem-mem vs. mem only

- Preliminary tests to correlate performance counter data with interference

Benchmark Performance

(Higher time means more interference)

<table>
<thead>
<tr>
<th>P1-P2</th>
<th>P1 Time (s)</th>
<th>P2 Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int-int</td>
<td>44</td>
<td>44</td>
</tr>
<tr>
<td>int-mem</td>
<td>44</td>
<td>44</td>
</tr>
<tr>
<td>int-fp</td>
<td>44</td>
<td>46</td>
</tr>
<tr>
<td>mem-mem</td>
<td>60</td>
<td>61</td>
</tr>
<tr>
<td>mem-fp</td>
<td>74</td>
<td>44</td>
</tr>
<tr>
<td>fp-fp</td>
<td>44</td>
<td>44</td>
</tr>
<tr>
<td>int</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>mem</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>fp</td>
<td>44</td>
<td></td>
</tr>
</tbody>
</table>
Outline

• Motivation & Approach
• Side-Channel Example
• Program Behavior and Phases
• Hardware Performance Counters
  • Estimating Interference
• New Scheduler Architecture
  • Machine Learning Module
• Overhead
• Results
New Scheduler Architecture

Kernel Scheduler

Scheduler

Kernel Module

PMC Mod

User Module

ML Mod

Kernel Space

User Space
New Scheduler Architecture

- Modified scheduler
  - Collect performance counter data
  - Uses prediction of upcoming program phase to separate memory programs
  - Attempt to minimized side-channels

- PMC Module
  - Interface between kernel data structures and ML Module

- ML Module
  - Machine learning module responsible for predicting upcoming program phase for each program
ML Module

- Predict upcoming program phase using ML module
- Uses neural network
  - 7 Layers (5 Hidden)
  - Input layer receives counter data from last 15 context switches
  - Counter data and output clustered using K-Means into 5 categories
  - Outputs which category the next context switch will be in
Asynchronous ML Module Execution

CPU0

T2

T9

CPU1

T8

T3

P: T3, T6, T5, ...

CPU2

T1

T6

P: T7, T9, T1, ...

CPU3

T4

T7
Asynchronous ML Module Execution

CPU0

CPU1

T2

T8

P: T3, T6, T5, ...

T3

CPU2

CPU3

T1

P: T7, T9, T1, ...

T6

T7

T9
Evaluation – Scheduler Overhead

- Counter recording only takes 50 instructions per context switch. Negligible.

- ML module prediction takes about 210us with about 10us of communication overhead. Context switches occur every ~2500us. Have enough time to predict future behavior of ~10 threads.

- ML module training done off-line. Similar to updating a user application when a new version is released.
Evaluation – Prediction Error Rates

- PE-M: Our predictor leverages the machine learning algorithms
- LE-M: Base predictor using last phase to predict next phase
- Memory phase prediction error rates:
  PE-M ~30% avg vs. LE-M: ~50% avg

<table>
<thead>
<tr>
<th>Prog</th>
<th>PE-M</th>
<th>LE-M</th>
</tr>
</thead>
<tbody>
<tr>
<td>astar</td>
<td>14</td>
<td>22</td>
</tr>
<tr>
<td>bzip2</td>
<td>33</td>
<td>50</td>
</tr>
<tr>
<td>dealII</td>
<td>12</td>
<td>33</td>
</tr>
<tr>
<td>gobmk</td>
<td>73</td>
<td>63</td>
</tr>
<tr>
<td>hmmer</td>
<td>12</td>
<td>67</td>
</tr>
<tr>
<td>lbm</td>
<td>32</td>
<td>79</td>
</tr>
<tr>
<td>libquantum</td>
<td>68</td>
<td>61</td>
</tr>
<tr>
<td>mcf</td>
<td>32</td>
<td>41</td>
</tr>
<tr>
<td>milc</td>
<td>29</td>
<td>55</td>
</tr>
<tr>
<td>namd</td>
<td>22</td>
<td>46</td>
</tr>
<tr>
<td>perlbench</td>
<td>27</td>
<td>43</td>
</tr>
<tr>
<td>povray</td>
<td>14</td>
<td>50</td>
</tr>
</tbody>
</table>

Prediction Error Rates (Less is better)
Summary and Ongoing Work

- Can use prediction of program behavior to schedule tasks on different cores to eliminate interference and minimize side channels

- Ongoing Work: Develop scheduler to utilize this prediction directly into the Linux scheduler with minimal overhead:

  **SOFT: Soft, low-Overhead, Fair Transfer scheduler**
Thank you!

Questions?

Junaid Nomani and Jakub Szefer

Computer Architecture and Security Laboratory
Yale University

junaid.nomani@yale.edu
jakub.szefer@yale.edu