TheME:
A System for Testing by Hardware Monitoring Events

Kristen R. Walcott-Justice
kwalcott@uccs.edu
University of Colorado - Colorado Springs

Jason Mars
jom5x@cs.virginia.edu
University of California - San Diego

Mary Lou Soffa
soffa@cs.virginia.edu
University of Virginia
• Measuring test quality:
  • Recompilation
  • High run time overheads
  • Large code growth

Software lifecycle

- Requirement
- Design
- Implementation
- Testing
- Bug fix
- Release
- Maintenance and Patching
Expense of Traditional Test Coverage Analysis

- Instrumentation
- Probe
- Payload
- Branch analysis overheads:
  - Time: 10% - 30%
  - Code growth: 60% - 90%

<table>
<thead>
<tr>
<th>Branch</th>
<th>Executed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>✓</td>
</tr>
<tr>
<td>B2</td>
<td></td>
</tr>
</tbody>
</table>

Will B1 execute?  Will B2 execute?
Efficient Program Monitoring

- Profiling
- Optimization
- Race Detection

Software-Level Monitoring
What is a Hardware Mechanism?

System State Sample

Sample read()

Interrupt from counter

User space

Operating System

L1 cache
L2 cache

Core Core

Core Core

Core

Shared L3 Cache

Performance Monitoring Unit (counters and mechanisms)
Using Hardware Mechanisms

- Developed for operating system performance analysis
- Widely available on nearly all processors
- Low overhead
  - Short setup time (318μs)
  - Quick read time (3.5μs)
- Use of samples
  - Estimate profiles
  - Reveal program execution behavior
- Removes need for instrumentation
Hardware Mechanisms in Testing: Goals and Challenges

- Structural testing requires more exact data
- Can we capture ALL events with which we are concerned?
- Can we capture ONLY the events with which we are concerned?
- Tradeoff:
  - Amount of information collected
  - Overhead of sampling
THEME: TESTING BY HARDWARE MONITORING EVENTS

- Program modification
- Hardware Sampling/Monitoring
- Coverage Calculation
THEME: Testing by Hardware
Monitoring Events

Program modification

Hardware Sampling/Monitoring

Coverage Calculation

Original Program
Assembly Instrumentation
Modified Program
Static Analysis
Modified Branch Table

Branch Sampler
High Level API Access
Low Level Access

Static Compiler Analysis
Sampled branches
Coverage Analysis
THEME: Testing by Hardware Monitoring Events

Program modification

Hardware Sampling/Monitoring

Coverage Calculation
THEME: Testing by Hardware Monitoring Events

Program modification

Hardware Sampling/Monitoring

Coverage Calculation

Original Program → Assembly Instrumentation → Modified Program → Static Analysis → Modified Branch Table → Branch Sampler → High Level API Access, Low Level Access → Sampled branches → Static Compiler Analysis → Coverage Analysis
Branch Vector Recording: Last Branch Record (LBR)

- Mechanism for partial branch profiling
- Intended for OS performance and debugging
- Tracks set of executed branches
- Branch source
- Branch destination
- Sample == Set of branches “Branch Vector”
THEME: Testing by Hardware Monitoring Events

Program modification

Hardware Sampling/Monitoring

Coverage Calculation
Enabling Fall-through Visibility

Challenge:
Hardware branch-based monitors can only see 1 of 2 branch edges

Methods
- Supplement with more samples
- Use static analysis to infer branches
- Minor program modification

Our Solution:
Insert innocuous unconditional branches
Enabling Fall-through Visibility

Challenge:
Hardware branch-based monitors can only see 1 of 2 branch edges

Methods
- Supplement with more samples
- Use static analysis to infer branches
- Minor program modification

Our Solution:
Insert innocuous unconditional branches

Figure 2: The THeME System

Figure 3: The LBR is incapable of detecting the fall-through branch edge from 1 to 2.

Example in Figure 2: Monitoring should be able to detect both the execution of the fall-through path from 2 to 3 and the target path from 2 to 4. While this is obvious when looking at a flow graph in the binary code, a branch is made up of some kind of jump to a target followed by another instruction. The xnR will report the jump from 2 to 4 but not the fall-through from 2 to 3. Therefore, the xnR by itself is only capable of monitoring some of the source level branches.

Fall-through branch observation is possible in several ways. One technique is to supplement the information from branch-based monitoring with other event data. For example, the INST_RETIRED event could be polled in addition to the xnR to look for fall-through instruction execution. Another technique to detect fall-through branches includes a static post-mortem analysis of the program and observed information. These techniques would require no code modification, recompilation, or code growth. However, because we want to evaluate the capabilities of using the hardware mechanisms for monitoring, we instead give the branch-based mechanism the potential to observe the fall-through path by inserting harmless unconditional branches along every fall-through edge in the binary, as pictured in Figure 3. This is different from instrumentation, which is heavy weight and includes both probe and payload code. Our fall-through enabling technique adds only a single instruction along fall-through branch edges. Using this technique, negligible code growth is incurred.

3.2 User-level Branch Vector Access
Once the program has been modified and analyzed, it is executed, as shown in Figure 3. xnR monitoring begins when the test program enters its main method and branch recording continues until the last instruction before the program ends. This prevents observation of the setup and teardown instructions executed as the program is loaded into and taken out of memory. Samples are taken based on the number of o—cycles observed during execution. When the sample rate of cycles is reached, the branches in the xnR are read and compared against the items in the branch table, and observed branches are marked as taken.

There are a number of ways to access branch vector data contained in the xnR. Many techniques in profiling, debugging, and other software tasks use some form of user-level performance monitoring. Alternatively, a lower level approach using interrupts can be used.

3.2.1 Access via Polling
The simplest technique to access and read the xnR is through a performance monitoring module and Linux's poll event. The test program is first spawned and executed using ptrace. Once the program has started execution successfully, xnR reading is enabled through a high level call to the operating system, as is the hardware counter that is to be used to trigger sampling. The monitoring program then repeatedly calls poll, which waits for the file descriptor associated with the performance counter to contain data that can be read, as shown below:

```c
for(;;) {
    ret = poll(pollfds, 1, -1);
    if (ret < 0 && errno == EINTR)
        break;
    process_smpl_buf(file_descriptor);
}
```

While poll is an effective technique to report sets of xnR and performance counter data, repeatedly calling poll when no data is available causes unnecessary overhead. Thus, we created an alternative technique that takes advantage of interrupts.

3.2.2 Interrupt Driven Access
In our second technique, we replace the repetitious call to poll with a lower level, more efficient hardware access approach. The hardware counters and xnR are enabled in the same way as described in Section "WZW. The poll calls are replaced by an uX–signal handler associated with our desired hardware mechanisms. The signal handler is immediately triggered upon the associated performance counter's overflow. After performing several checks, the signal handler reads the xnR branch vector, and each branch is processed. The associated hardware counter then is reset and the program is resumed. Any handling the performance counter notification and refreshing the counter directly from within the
THEME: Testing by Hardware Monitoring Events

Program modification
- Original Program
  - Assembly Instrumentation
  - Modified Program
    - Static Analysis
      - Modified Branch Table

Hardware Sampling/Monitoring
- Branch Sampler
  - High Level API Access
  - Low Level Access

Coverage Calculation
- Static Compiler Analysis
  - Sampled branches
    - Coverage Analysis
THEME: Testing by Hardware Monitoring Events

Program modification

Hardware Sampling/Monitoring

Coverage Calculation
THEME: Testing by Hardware Monitoring Events

- Program modification
- Hardware Sampling/Monitoring
- Coverage Calculation
Improving Branch Coverage

- Sampling → Some missed data
- Goal: Improve coverage using static analysis
- Dominator analysis
  - Associate seen branches with control flow graph
  - Branch $b$ executed → branch $c$ also executed
Experiment and System Design

- Intel Core i7 860 quad-core processor
- LBR size of 16 branches
- Linux 2.6.34
- Hardware access tools: libpfm4 (user-level), perf (kernel-level)

- SPEC2006 C Benchmarks
- Metrics:
  - Efficiency - time
  - Code growth size
  - Effectiveness - branch coverage
  - Instrumented vs Hardware Monitoring
The Intel x86 processor was selected because it is effective in comparison to using instrumentation. The primary goal of this paper's empirical study is to evaluate the trade-offs between the use of a hardware approach for structural testing and monitoring using hardware mechanisms on a single core and the benefit of incorporating static analysis into the design of test cases.

### 4.2 Experiments and Results

#### Time overhead

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Branch Cov.</th>
<th>Time (s)</th>
<th>Mod. Time (s)</th>
<th>Instr. Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>64.20%</td>
<td>1499</td>
<td>1514</td>
<td>1599</td>
</tr>
<tr>
<td>h264ref</td>
<td>35.72%</td>
<td>1753</td>
<td>1786</td>
<td>1890</td>
</tr>
<tr>
<td>libquantum</td>
<td>39.07%</td>
<td>1056</td>
<td>1178</td>
<td>1236</td>
</tr>
<tr>
<td>mcf</td>
<td>74.01%</td>
<td>529</td>
<td>539</td>
<td>575</td>
</tr>
<tr>
<td>sjeng</td>
<td>48.87%</td>
<td>1028</td>
<td>1162</td>
<td>1312</td>
</tr>
</tbody>
</table>

**Results:**

- **Avg:** Increase by 5%
- **Instr.:** Increase by 14%

Increases code growth.
Results: Enabling Fall-Through Visibility

- Impact:
  - Increases time overhead
  - Increases code growth
  - How compared to instrumentation?

Code Growth

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Native Size (kB)</th>
<th>Mod. % Increase</th>
<th>Instr. % Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>260 kB</td>
<td>1.52</td>
<td>32.65</td>
</tr>
<tr>
<td>h264ref</td>
<td>2892 kB</td>
<td>0.69</td>
<td>18.39</td>
</tr>
<tr>
<td>libquantum</td>
<td>208 kB</td>
<td>0</td>
<td>20.00</td>
</tr>
<tr>
<td>mcf</td>
<td>128 kB</td>
<td>0</td>
<td>17.95</td>
</tr>
<tr>
<td>sjeng</td>
<td>592 kB</td>
<td>0.67</td>
<td>30.05</td>
</tr>
</tbody>
</table>

Avg: 0.5%  Avg: 24%
Results: Testing on a Single Core - Effectiveness

Coverage Comparison: Full Instrumentation vs Sampling on Ref Inputs

- bzip2
- h264ref
- libquantum
- mcf
- sjeng

Inst 500K 1M 5M 10M 50M
Sample periods per benchmark

Branch coverage

With Dom Analysis
LBR Alone
Results: Testing on a Single Core - Effectiveness

Coverage Comparison: Full Instrumentation vs Sampling on Ref Inputs

Branch coverage

- bzip2
- h264ref
- libquantum
- mcf
- sjeng

With Dom Analysis
LBR Alone
Results: Testing on a Single Core - Efficiency

Percent Time Overhead Using Interrupt Driven Approach on Ref Inputs

Sample periods per benchmark

Percent time overhead

-10% 0% 10% 20% 30% 40% 50% 60%

500K 1M 5M 10M 50M

bzip h264ref libquantum mcf sjeng
Results: Better Coverage at High Sample Rates

Coverage Comparison: Full Instrumentation vs Sampling on Ref Inputs

- bzip2
- h264ref
- libquantum
- mcf
- sjeng

Branch coverage

- With Dom Analysis
- LBR Alone
Results: Better Coverage at High Sample Rates

Coverage Comparison: Full Instrumentation vs Sampling on Ref Inputs

- bzip2
- h264ref
- libquantum
- mcf
- sjeng

Branch coverage
Results: Better Coverage at High Sample Rates

Coverage Comparison: Full Instrumentation vs Sampling on Ref Inputs

Branch coverage

With Dom Analysis
LBR Alone

71%
Results: Better Coverage at High Sample Rates

Coverage Comparison: Full Instrumentation vs Sampling on Ref Inputs

- bzip2
- h264ref
- libquantum
- mcf
- sjeng

With Dom Analysis
LBR Alone

90% 72%
Results: Testing on a Multiple Cores - Efficiency

Percent Time Overhead Splitting Inputs Across Cores

Sample periods per benchmark

bzip2

h264ef

Percent time overhead

-20%
-10%
0%
10%
20%
30%
40%

500K
1M
5M
10M
50M
Hardware Monitoring

Benefits

• Low overhead, effective branch testing technique
• Up to 90% of branch coverage
• 2% time improvement
• 0.5% code growth (compared to 60% to 90%)
• Test coverage approximation
• Testing on resource constrained devices
• “Imprecise” tasks (e.g. regression test prioritization)
• Partial program monitoring
• Significant benefits
• Enable testing on resource constrained devices
• Generates full picture of program execution
Conclusions and Future Work

- Extensible, portable system for single or multiple cores
- Up to 11.13% improvement in time overhead
- Up to 90% of the coverage reported by instrumentation
  - Reduced time overhead (~2%)
- Negligible code growth
- Future work:
  - Combine hardware monitoring with limited instrumentation
  - Implement on resource constrained device
  - Extend system to other coverage metrics
Thank You!

Website:
http://www.cs.virginia.edu/walcott

Questions?