Fast Instruction Cache Analysis via Static Cache Simulation

Frank Mueller
Humboldt-Universität zu Berlin
Fachbereich Informatik
Unter den Linden 6
10099 Berlin (Germany)

e-mail: whalley@cs.fsu.edu

David Whalley
Florida State University
Department of Computer Science
Tallahassee, FL 32304-4019
U.S.A.
Overview

- caches bridge bottleneck between CPU and MM speed
- traditional (trace-driven) methods slow (about 100x overhead)
- new, efficient method for instruction cache simulation:
  - provides faster instruction cache performance evaluation
  - determine number of hits and misses of a program execution
  - used to evaluate new cache designs
  - used to analyze new optimization techniques
Methods in Contrast

- Goal: faster instruction cache performance evaluation
- traditional approach: inline tracing
  - instrument program on complement of min. spanning tree
  - generate trace addresses
  - simulate caches based on trace
- our approach: on-the-fly analysis
  - analyze program statically (static cache simulation)
  - instrument program on “unique paths”
  - do NOT generate trace addresses
  - simulate remaining cache behavior within program execution
Static Cache Simulation

- address of instructions known statically
- predicts large portion of instruction cache references
- uses iterative analysis of call graph and control flow
- categorizes each instruction
- assumes:
  - direct-mapped caches
  - currently no recursion allowed
Overview of Static Cache Simulation

source files \rightarrow \text{compiler} \rightarrow \text{assembly files} \rightarrow \text{assembler} \rightarrow \text{object files} \rightarrow \text{linker} \rightarrow \text{execut. program}

- cache configuration
- control flow info
- static cache simulator
- cache state table instrumentation macros
- cache analysis library routines
Instruction Categorization

- transforms call graph into function-instance graph (FIG)
- performs analysis on FIG and control-flow graph
- uses data-flow analysis algorithms for prediction
- *abstract cache state*: potentially cached program lines
- *reaching state*: reachable program lines
- categories based on these states:
  - always hit
  - always miss
  - first miss: miss on first reference, hit on consecutive ones
  - conflict: either hit or miss (dynamic)
Algorithm to Calculate Cache States

\begin{align*}
\text{input\_state}(\text{main}) & := \text{all invalid lines}; \\
\text{WHILE any change DO} & \\
\quad \text{FOR each instance of a UP in the program DO} & \\
\quad \quad \text{input\_state}(\text{UP}) & := \phi; \\
\quad \quad \text{FOR each immediate predecessor P of UP DO} & \\
\quad \quad \quad \text{input\_state}(\text{UP}) & := \text{input\_state}(\text{UP}) \cup \text{output\_state}(P); \\
\quad \quad \quad \text{output\_state}(\text{UP}) & := \\
\quad \quad \quad \quad [\text{input\_state}(\text{UP}) \cup \text{prog\_lines}(\text{UP})] \setminus \text{conf\_lines}(\text{UP});
\end{align*}
Fast Instruction Cache Analysis via Static Cache Simulation
Fast Instruction Cache Analysis via Static Cache Simulation

- 4 cache lines
- 16 bytes per line (4 instructions)
- instances foo (a) block 8a and (b) block 8b
- 7(1): always hit, spatial locality
- 8b(1): always hit, temporal locality
- 3(3): first miss
- 5(1) and 6(1): group first miss
- 3(1): conflict with 8b(2) conditionally executed
"I" = invalid

cache 0 1 2 3 0 1 2 3 0 1 cache ln. 0 1 2 3 0 1 2 3 0 1
program I I I I 0 1 2 3 4 5 prog. ln. I I I I 0 1 2 3 4 5

PASS 1

<table>
<thead>
<tr>
<th>in(1)</th>
<th>I I I I</th>
<th>out(1)</th>
<th>I I I I 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>in(8a)</td>
<td>I I I 0</td>
<td>out(8a)</td>
<td>I I I 4 5</td>
</tr>
<tr>
<td>in(2)</td>
<td>I I 4 5</td>
<td>out(2)</td>
<td>I I 1 4</td>
</tr>
<tr>
<td>in(3)</td>
<td>I 1 4</td>
<td>out(3)</td>
<td>I 1 2 4</td>
</tr>
<tr>
<td>in(4)</td>
<td>1 2 4</td>
<td>out(4)</td>
<td>1 1 2 4</td>
</tr>
<tr>
<td>in(5)</td>
<td>1 2 3 4</td>
<td>out(5)</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>in(8b)</td>
<td>1 2 3 4</td>
<td>out(8b)</td>
<td>2 3 4 5</td>
</tr>
<tr>
<td>in(6)</td>
<td>I 1 2 3 4 5</td>
<td>out(6)</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>in(7)</td>
<td>1 2 3 4 5</td>
<td>out(7)</td>
<td>1 2 3 4 5</td>
</tr>
</tbody>
</table>

PASS 2

<table>
<thead>
<tr>
<th>in(1)</th>
<th>I I I I</th>
<th>out(1)</th>
<th>I I I I 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>in(8a)</td>
<td>I I I 0</td>
<td>out(8a)</td>
<td>I I I 4 5</td>
</tr>
<tr>
<td>in(2)</td>
<td>I I 4 5</td>
<td>out(2)</td>
<td>I I 1 4</td>
</tr>
<tr>
<td>in(3)</td>
<td>I 1 2 3 4 5</td>
<td>out(3)</td>
<td>I 1 2 3 4</td>
</tr>
<tr>
<td>in(4)</td>
<td>1 2 3 4</td>
<td>out(4)</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>in(5)</td>
<td>1 2 3 4</td>
<td>out(5)</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>in(8b)</td>
<td>1 2 3 4</td>
<td>out(8b)</td>
<td>2 3 4 5</td>
</tr>
<tr>
<td>in(6)</td>
<td>I 1 2 3 4 5</td>
<td>out(6)</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>in(7)</td>
<td>1 2 3 4 5</td>
<td>out(7)</td>
<td>1 2 3 4 5</td>
</tr>
</tbody>
</table>
Code Instrumentation

- merging states: local path state, shared path state (SPS)
- states provide DFA to simulate conflicts locally
- frequency counters
- macros for calls
- macros for paths
- first miss table
- calculate hits and misses from frequencies and states
Fast Instruction Cache Analysis via Static Cache Simulation
Measurements

- modified back-end of opt. compiler VPO
- performed static cache simulation
- instrumented programs for instruction cache simulation
- direct-mapped cache simulated
- uniform instruction size of 4 bytes simulated
- cache line size was 4 words (16 bytes)
- results verified by comparison against trace-driven simulation
Performance Evaluation

- UPPAs and function instances vs. basic block partitioning
  - static savings: 24% fewer measurement points
  - dynamic savings: 31% fewer measurement points
- predictability of instructions
  - static: 16% conflicts, other 84% predictable
  - dynamic: 26% conflicts, other 74% predictable
- efficient in-line code instrumentation accounts for remaining savings
- trace-driven overhead 18x, our method only 2x
### Static Measurements for 1kB Direct-Mapped Cache

<table>
<thead>
<tr>
<th>Name</th>
<th>Hit</th>
<th>Miss</th>
<th>Firstmiss</th>
<th>Conflict</th>
<th>Measure Pts.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cachesim</td>
<td>70.83%</td>
<td>6.99%</td>
<td>0.70%</td>
<td>21.48%</td>
<td>73.38%</td>
</tr>
<tr>
<td>cb</td>
<td>79.03%</td>
<td>2.35%</td>
<td>0.00%</td>
<td>18.63%</td>
<td>89.62%</td>
</tr>
<tr>
<td>compact</td>
<td>70.12%</td>
<td>4.96%</td>
<td>0.12%</td>
<td>24.80%</td>
<td>68.89%</td>
</tr>
<tr>
<td>copt</td>
<td>70.89%</td>
<td>7.41%</td>
<td>7.03%</td>
<td>14.67%</td>
<td>84.19%</td>
</tr>
<tr>
<td>dhrystone</td>
<td>70.03%</td>
<td>10.71%</td>
<td>7.30%</td>
<td>11.96%</td>
<td>81.61%</td>
</tr>
<tr>
<td>fft</td>
<td>74.07%</td>
<td>4.85%</td>
<td>16.42%</td>
<td>4.66%</td>
<td>78.43%</td>
</tr>
<tr>
<td>genreport</td>
<td>70.61%</td>
<td>9.95%</td>
<td>5.61%</td>
<td>13.84%</td>
<td>71.58%</td>
</tr>
<tr>
<td>mincost</td>
<td>72.79%</td>
<td>9.96%</td>
<td>1.14%</td>
<td>16.11%</td>
<td>83.19%</td>
</tr>
<tr>
<td>sched</td>
<td>67.65%</td>
<td>5.06%</td>
<td>0.09%</td>
<td>27.19%</td>
<td>73.16%</td>
</tr>
<tr>
<td>sdiff</td>
<td>68.94%</td>
<td>12.06%</td>
<td>0.89%</td>
<td>18.11%</td>
<td>72.13%</td>
</tr>
<tr>
<td>tsp</td>
<td>72.61%</td>
<td>13.50%</td>
<td>3.88%</td>
<td>10.01%</td>
<td>64.08%</td>
</tr>
<tr>
<td>whetstone</td>
<td>75.70%</td>
<td>12.84%</td>
<td>0.24%</td>
<td>11.22%</td>
<td>70.49%</td>
</tr>
<tr>
<td>average</td>
<td>71.94%</td>
<td>8.39%</td>
<td>3.62%</td>
<td>16.06%</td>
<td>75.90%</td>
</tr>
</tbody>
</table>
## Dynamic Measurements for 1kB Direct-Mapped Cache

<table>
<thead>
<tr>
<th>Name</th>
<th>Measure Pts.</th>
<th>Hit Ratio</th>
<th>Trace</th>
<th>SSim</th>
<th>Conflict</th>
</tr>
</thead>
<tbody>
<tr>
<td>cachesim</td>
<td>60.56%</td>
<td>77.19%</td>
<td>8.41</td>
<td>1.53</td>
<td>34.12%</td>
</tr>
<tr>
<td>cb</td>
<td>65.61%</td>
<td>93.84%</td>
<td>33.56</td>
<td>3.51</td>
<td>30.67%</td>
</tr>
<tr>
<td>compact</td>
<td>56.56%</td>
<td>92.90%</td>
<td>22.29</td>
<td>2.31</td>
<td>21.34%</td>
</tr>
<tr>
<td>copt</td>
<td>74.88%</td>
<td>93.64%</td>
<td>16.43</td>
<td>1.58</td>
<td>30.00%</td>
</tr>
<tr>
<td>dhrystone</td>
<td>72.73%</td>
<td>83.73%</td>
<td>19.89</td>
<td>1.31</td>
<td>16.01%</td>
</tr>
<tr>
<td>fft</td>
<td>74.08%</td>
<td>99.95%</td>
<td>5.79</td>
<td>0.95</td>
<td>8.80%</td>
</tr>
<tr>
<td>genreport</td>
<td>81.31%</td>
<td>97.45%</td>
<td>13.57</td>
<td>1.91</td>
<td>28.92%</td>
</tr>
<tr>
<td>mincost</td>
<td>76.27%</td>
<td>89.08%</td>
<td>23.47</td>
<td>2.23</td>
<td>30.67%</td>
</tr>
<tr>
<td>sched</td>
<td>58.29%</td>
<td>96.41%</td>
<td>25.90</td>
<td>3.62</td>
<td>42.01%</td>
</tr>
<tr>
<td>sdiff</td>
<td>77.82%</td>
<td>97.61%</td>
<td>32.10</td>
<td>3.99</td>
<td>28.40%</td>
</tr>
<tr>
<td>tsp</td>
<td>58.67%</td>
<td>86.98%</td>
<td>5.70</td>
<td>1.19</td>
<td>17.63%</td>
</tr>
<tr>
<td>whetstone</td>
<td>68.25%</td>
<td>100.00%</td>
<td>13.44</td>
<td>1.36</td>
<td>23.56%</td>
</tr>
<tr>
<td>average</td>
<td>68.75%</td>
<td>92.40%</td>
<td>18.38</td>
<td>2.12</td>
<td>26.01%</td>
</tr>
</tbody>
</table>
Average Simulation Overhead

Execution Overhead vs. Cache Size [Bytes]

- SSsim
- Trace

Fast Instruction Cache Analysis via Static Cache Simulation
Future Work

- recursion
- set-associative caches
- data caching
- integrate with timing tool to tightly predict WET/BET
- other applications
Summary

- uses efficient on-the-fly analysis
- performs static instruction cache simulation
- instruments program
- provides accurate cache performance measurements
- instrumented program has only about 2x execution overhead
- faster than any other cache analysis method published so far