Toward Measuring Memory Hierarchy Effects by Region

Bryan Buck

Jeffrey K. Hollingsworth

© Copyright 1998, Jeffrey K. Hollingsworth, All Rights Reserved.
Measuring Cache Effects by Region

- Simple base/bound register
  - Duplicate cache related performance counters
  - Each counter set collects info in own base/bounds
  - Difficult to convince chip makers to include

(assuming counters can use virtual, not physical addresses)
Approximating in Software

- Use a software cache simulator
  - Instrument applications to keep statistics

- Useful without hardware if fast enough
  - Need to instrument all load/stores, could be slow

- Demonstrates potential benefits
  - Even if slow, tells us if hardware would be useful
Cache Simulator

- Used existing simple cache simulator
  - Instruments application using ATOM
    - Calls cache simulator at loads/stores
    - Cycle count updated at each basic block
  - No need to change application source code
    - Optionally make a call to start measurement

- Added multiple performance counters
  - Each counter set has base/bounds
Memory Hotspot Search

- Goal: identify region causing most misses
- Use $n$-way search
  - Start with all memory split $n$ ways and narrow down
  - Sample counters at regular intervals and readjust
  - Question: how does $n$ affect the results?

- Tested on SPEC95 benchmark applications
## Search Results

<table>
<thead>
<tr>
<th>application</th>
<th>variable</th>
<th>% of misses</th>
<th>2-way</th>
<th>10-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>RX</td>
<td>23.59</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tomcatv</td>
<td>RY</td>
<td>23.57</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DD</td>
<td>9.71</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>9.60</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>H</td>
<td>7.72</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>UOLD</td>
<td>7.70</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VOLD</td>
<td>7.70</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>swim</td>
<td>UNEW</td>
<td>7.69</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PNEW</td>
<td>7.69</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>POLD</td>
<td>7.69</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>7.64</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>
## Search Results Continued

<table>
<thead>
<tr>
<th>application</th>
<th>variable</th>
<th>% of misses</th>
<th>2-way</th>
<th>10-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>su2cor</td>
<td>U</td>
<td>43.35</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>W1</td>
<td>11.47</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>9.82</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>W2</td>
<td>9.04</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>mgrid</td>
<td>U</td>
<td>41.76</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>40.88</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>21.23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>applu</td>
<td>A</td>
<td>21.22</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>21.22</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>compress</td>
<td>htab</td>
<td>66.49</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>codetab</td>
<td>25.81</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>jpeg</td>
<td>jpeg_com…</td>
<td>3.92</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
Arrays Crossing Search Boundaries

- An array may span two or more regions
  - Not enough misses in single region for detection
  - This is the problem with su2cor
Search Time

The bar chart shows the search time in billion cycles for several applications. The applications are: tomcatv, swim, su2cor, mgrid, applu, compress, and jpeg. The chart compares the search time between 2-way and 10-way configurations.
Misses vs. Time: Compress

Sample interval (25 million cycles each)
Instrumentation Costs

- cycle count
- load/store
- 2-way search
- 10-way search

(normalized execution time vs. application cost)

Applications:
- tomcat
- swim
- su2cor
- mgrid
- applu
- compress
- ijpeg
Implementing with the Dyninst API

● **What we need**
  – Determine instruction type (load/store)
  – Get effective address of load/store target
  – Basic block information

● **What Dyninst will let us do**
  – Allow application to initialize at normal speed
  – More selective instrumentation
  – Ability to add other axes (like Paradyn)
    • Show which function is causing cache problem
Conclusion

- Region miss information is useful
  - Automatic search can efficiently find arrays
- Simple algorithm has problems with...
  - Phases
  - Arrays spanning search regions
- More counters are more useful
  - 10-way search gets better results than 2-way
  - More counters doesn’t mean faster solution
- Cost of software instrumentation is high
  - Due to executing cache simulator every load/store
  - Much less instrumentation needed with hardware
Future Work

- Port simulator and search to Dyninst API
  - Need additional Dyninst features
- Assess value of counters in hardware
  - Simulate slowdown
- More sophisticated algorithms
  - Deal with phases
  - Better handling of dynamically allocated memory
    • Rearrange allocation for measurement
- More sophisticated instrumentation
  - Filter load/store instructions
  - Use hardware counters