Modeling Power

• Last time:
  • Architectural power estimation techniques
  • Analytical vs. Empirical Modeling methods
  • Flexibility vs. Accuracy tradeoffs

• Today: Measuring power
Measuring Power

• **Why Measure?**
  • Help validate models
  • Good for software/compiler studies with fixed architecture
  • Good for OS studies (relying on HW feedback)
    – Another challenge – how to measure power on a running machine?

• **Challenges:**
  • Difficult to get enough motherboard information to measure the power you want to.
  • Even harder to break down on-chip power into a pie chart of different contributors
  • Difficult to ascribe power peaks and valleys to particular software behavior or program constructs.
  • Difficult to get very fine-grained readings due to power-supply capacitors
Typical meter-based setups: Voltage-drops with transceivers

- Power = Vsupply * Vsense/Rsense
Instruction-level Power Analysis

One of the first efforts in the area of power-aware software:


<table>
<thead>
<tr>
<th>No.</th>
<th>Instruction</th>
<th>Current (mA)</th>
<th>Cycles</th>
<th>Energy ($10^{-8}$ J)</th>
<th>Instruction</th>
<th>Current (mA)</th>
<th>Cycles</th>
<th>Energy ($10^{-8}$ J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>nop</td>
<td>276</td>
<td>1</td>
<td>2.27</td>
<td>nop</td>
<td>198</td>
<td>1</td>
<td>3.26</td>
</tr>
<tr>
<td>2</td>
<td>mov dx,[bx]</td>
<td>428</td>
<td>1</td>
<td>3.53</td>
<td>ld [%10],%i0</td>
<td>213</td>
<td>1</td>
<td>3.51</td>
</tr>
<tr>
<td>3</td>
<td>mov dx, bx</td>
<td>302</td>
<td>1</td>
<td>2.49</td>
<td>or %g0,%i0,%10</td>
<td>198</td>
<td>1</td>
<td>3.26</td>
</tr>
<tr>
<td>4</td>
<td>mov [bx], dx</td>
<td>522</td>
<td>1</td>
<td>4.30</td>
<td>st %i0,[%10]</td>
<td>346</td>
<td>2</td>
<td>11.4</td>
</tr>
<tr>
<td>5</td>
<td>add dx,bx</td>
<td>314</td>
<td>1</td>
<td>2.59</td>
<td>add %i0,%e0,%10</td>
<td>199</td>
<td>1</td>
<td>3.28</td>
</tr>
<tr>
<td>6</td>
<td>add dx,[bx]</td>
<td>400</td>
<td>2</td>
<td>6.60</td>
<td>mul %g0,%r29,%r27</td>
<td>198</td>
<td>1</td>
<td>3.26</td>
</tr>
<tr>
<td>7</td>
<td>jmp</td>
<td>373</td>
<td>3</td>
<td>9.23</td>
<td>srl %i0,1,%10</td>
<td>197</td>
<td>1</td>
<td>3.25</td>
</tr>
</tbody>
</table>
Traumagen Microbenchmark

- Toolset developed to generate assembly language microbenchmarks with intentionally devious behavior.
- For x86 processors, can generate desired:
  - Instruction Cache Miss Rates
  - Data Cache Miss Rates
  - Branch Misprediction Rates
  - Data Activity
**Traumagen: Creating Controllable Cache Misses**

**Input parameters to Traumagen:**
- Desired miss rate
- Desired dependence structure
- Cache size and organization

**Controlled Misses**

Only two sets of data are accessed:
- Cache Hit: A single line in the cache which will always be present.
- Cache Miss: A circular linked list with intentionally conflicting cache lines. When traversed, it always produces a miss.
- All L1 misses are designed to hit in L2.
Power vs. Cache Hit Rate

- Power not monotonic with hit rate because L2 access costs are significant.
Traumagen: Creating Controllable Branch Mispredicts

Traumagen inputs:
- Branch predictor size
- Branch prediction accuracy
- Amount of computation between branches

Observations:
- Easy to generate predictable branches.
- Difficult to generate branches that are always mispredicted.

Solution:
- Mix ratio of predictable and unpredictable branches.
- Generate unpredictable branches with little overhead.
- Base branch conditions on data dependent rotations for semi-random outcomes.
Next time

• Itsy Pocket Computer
