Plan for Today

• Try to finish HW2 early!
• Questions
• Dynamic execution II discussion
• Assignment
Topics

• Memory disambiguation

• Branch predictors

• Multithreading Primer
Memory Disambiguation

• Loads and stores to different addresses can be reordered w/out hazards

• Not so for memory operations using the same address
  – RAW: Load X moved above Store X
  – WAR: Store X moved above Load X
  – WAW: Store X moved above Store X

• Thus, to be safe:
  – A load to X must check for prior stores to X
  – A store to X must check for prior loads and stores to X

• Two-step implementation
  – Effective address computations for Load/store operations are carried out in program order
  – Then operations are inserted into load/store buffers and executed after checking for the hazards above
Branch Prediction

• Dynamic scheduling can keep a pipeline busy, given either:
  – Long basic blocks
  – Accurate branch prediction

• When branches are predicted, we must be able to handle mispredicts (i.e., roll back). Two methods:
  – Instruction cannot Execute until all prior branches are resolved (creates no false exceptions)
  – Instruction cannot Write Result until all prior branches are resolved (w/ delayed, program-order handling of exceptions)
Branch Predictor Taxonomy

• Static approaches via the compiler
  – Always predict taken
  – Backwards taken, forward not taken
  – Branch delay slot (OK for single-issue pipes)
  – Profile-based
    • provide individual branch ‘hints’ based on expected inputs

• Dynamic approaches
Dynamic Branch Prediction

• Parameters to predict
  – Is this instruction a branch?
  – Will it be taken?
  – What is the target address?
  – What is the instruction at the target address?

• Here, we first consider how to predict, then we consider when to predict
Basic Idea

BPT Lookup

Branch

Predict. Fetch target if taken, else fall through.

- Right: Yes. Update history.
- Wrong: Kill instruction in progress, reload pipeline, update history.

Not a Branch

Fall through

- Right: Do nothing.
- Wrong: Kill instruction in progress, reload pipeline, update history.
Simplest Design

- Branch history is a table (BPT) addressed by lower bits of the PC
  - Or store predictions at icache lines (more later)

- One bit prediction stored at each entry
  - The prediction is the direction taken previously
  - This will mispredict the first and last loop iterations

- We can avoid mispredicting the first iteration in steady-state by using 2-bits at each entry
2-bit Saturating Counter
2-bit Performance

- Improvements
  - More counter bits
  - More entries
  - Better prediction approach

- Counters rely on the recent behavior of the branch in question
Correlating Branch Predictors

• Branches interact

• Example
  – If b1 and b2 are taken, b3 will not be
  – Counter schemes cannot capture this

• Use history of past N branches as index
  – This vector indexes a pattern history table (PHT)

```plaintext
If (x==2)  //b1
  x = 0;
If (y==2)  //b2
  y = 0;
If (x != y)  //b3
  ...
```
Implementation: Basic Idea

Pattern History Table (PHT)

History register of last $k$ branches (org. as shift register)

$2^k$ entries of e.g. 2-bit counters
Performance

- Correlation helps

- Improvements
  - Tournament predictors
    - Choose between a global and a local predictor on a per-branch basis
Tournament Predictors

Legend: x/y

- x- 0 means P1 was wrong, 1 means P1 was right
- y- 0 means P2 was wrong, 1 means P2 was right
Tournament Predictor Performance

![Graph showing conditional branch misprediction rate vs total predictor size for different predictor types: Local 2-bit predictors, Correlating predictors, Tournament predictors. The graph illustrates the performance improvement as predictor size increases.]
Predicting Branch Targets

• What good is a prediction if we do not have it at IF?
  – Instructions are not known to be branches until the end of ID (one cycle too late)

• Idea: index a table, called the branch target buffer (BTB), with PCs of taken branches, store target address there

• Steps
  – When we fetch a PC, do this table lookup in parallel
  – On a predicted taken branch, the target PC is known at the end of IF
  – Think of this as a BPT that stores the target along with the prediction
Sample BTB Organization

- PC of instruction to fetch
- Look up
- Predicted PC
- Number of entries in branch-target buffer

- Yes: instruction is branch and predicted PC should be used as the next PC
- No: instruction is not predicted to be branch; proceed normally

Branch predicted taken or untaken

© 2003 Elsevier Science (USA). All rights reserved.
BTB Process

IF

Send PC to memory and branch-target buffer

Entry found in branch-target buffer?

No

Yes

ID

Is instruction a taken branch?

No

Normal instruction execution

Yes

Send out predicted PC

EX

Enter branch instruction address and next PC into branch-target buffer

Mispredicted branch; kill fetched instruction; restart fetch at other target; delete entry from target buffer

Branch correctly predicted; continue execution with no stalls

Taken branch?

No

Yes
Multithreading Primer

• Idea: Exploit parallelism available between threads

• Each thread needs HW support
  – Register File
  – PC, other special registers
  – Memory system bookkeeping (e.g., page table)
  – Fast context switches
Thread Granularity

• Fine-grained
  – Context switches every cycle, round-robin schedule that skips stalled threads
  – Pro: Hides short and long latencies
  – Con: Each thread issues a max of 1 instruction per issue “rotation” (i.e., many threads can provide good throughput but poor individual latency)

• Coarse-grained
  – Context switches only on long-latency stalls (e.g., cache misses)
  – Pro: Individual threads can have better latency
  – Con: Cannot hide short latencies due to pipeline flushes
Simultaneous Multithreading

• Idea: mix fine-grain multithreading with a wide-issue speculative superscalar
  – HW can already handle dependencies, just add thread id bits
  – Use multiple issue from multiple threads since single threads rarely have enough ILP to fill issue width
Visual Comparison

<table>
<thead>
<tr>
<th>Issue slots</th>
<th>Time (proc cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Superscalar</td>
</tr>
<tr>
<td></td>
<td>Multiprocessing</td>
</tr>
<tr>
<td></td>
<td>SMT</td>
</tr>
</tbody>
</table>

From: Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading, LO et al.
Assignment

• Readings
  – For Wednesday
    • H&P: sections 2.3, 2.6-2.7
    • Skim V&L: Ch. 6
  – For Monday
    • H&P: 2.8-2.10

HW1 Statistics

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total</strong></td>
<td>15</td>
</tr>
<tr>
<td><strong>Mean</strong></td>
<td>13.9</td>
</tr>
<tr>
<td><strong>Max</strong></td>
<td>15</td>
</tr>
<tr>
<td><strong>Min</strong></td>
<td>12</td>
</tr>
<tr>
<td><strong>Std Dev</strong></td>
<td>0.9</td>
</tr>
</tbody>
</table>