CSE 533
Advanced Computer Architectures

Lecture 9
Branch Prediction
Outline

• Reducing the impact of control hazards
  – Branch prediction buffers
  – Branch target buffers
  – Return address stacks

[Hennessy/Patterson CA:AQA (3rd Edition): Appendix A, Chapter 3]
(Review) Reducing Branch Stalls

- **Baseline**: Stall pipeline on a branch instruction
  - 3 cycle stall on 5-stage RISC pipeline (2 if not taken)

- **Faster** stall pipeline
  - Move branch direction/target determination to ID stage
  - 1 cycle stall

- **Predict branch not taken**, squash instructions as necessary
  - 0 cycle stall on not-taken branches, 1 cycle stall for taken

- **Predict branch taken**, squash instructions as necessary
  - Need target address, so 1-cycle stall
  - Can reduce to 0 with additional hardware: Branch Target Buffer

- **Branch delay slots**
  - Instruction after branch always executes
  - 0 cycle stall if useful instruction can be found (from above/below branch)
  - Else, 1-cycle stall (effectively)
Evaluating Branch Alternatives

Pipeline speedup = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}

- **Assumptions**
  - 14% of instructions are branches
  - 30% of branches are not taken
  - 50% of delay slots can be filled with useful instructions

<table>
<thead>
<tr>
<th>Scheduling scheme</th>
<th>Branch penalty</th>
<th>CPI</th>
<th>speedup v. unpipelined</th>
<th>speedup v. stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow stall pipeline</td>
<td>3</td>
<td>1.42</td>
<td>3.5</td>
<td>1.0</td>
</tr>
<tr>
<td>Fast stall pipeline</td>
<td>1</td>
<td>1.14</td>
<td>4.4</td>
<td>1.26</td>
</tr>
<tr>
<td>Predict taken</td>
<td>1</td>
<td>1.14</td>
<td>4.4</td>
<td>1.26</td>
</tr>
<tr>
<td>Predict not taken</td>
<td>0.7</td>
<td>1.10</td>
<td>4.5</td>
<td>1.29</td>
</tr>
<tr>
<td>Delayed branch</td>
<td>0.5</td>
<td>1.07</td>
<td>4.7</td>
<td>1.34</td>
</tr>
</tbody>
</table>

- A compiler can reorder instructions to further improve speedup
Importance of Avoiding Branch Stalls

• Crucial in modern microprocessors, which issue/execute multiple instructions every cycle
  – Need to have a steady stream of instructions to keep the hardware busy
  – Stalls due to control hazards dominate

• So far, we have looked at static schemes for reducing branch penalties
  – Same scheme applies to every branch instruction

• Potential for increased benefits from dynamic schemes
  – Can choose most appropriate scheme separately for each instruction
    • Branches to top of loop have different behavior (Taken) than “if (x == 0) return;” (Not Taken)
  – Can “learn” appropriate scheme based on observed behavior
    • Dynamic (hardware) branch prediction schemes
      • For both direction (T or NT) and target prediction
      • Key element of all modern microprocessors
Dynamic Branch Prediction (1):
Branch Prediction (History) Buffer

- Small memory indexed by the low-order bits of the branch instruction
  - Stores a single bit of information: T or NT
    - Starts off as T, flips whenever a branch behaves opposite to prediction
  - For now, assume supported in the ID stage
    - Benefits for larger pipelines, more complex branches

- Problems with this simple scheme
  - Prediction value may not correspond to branch being considered
    - Cannot avoid this: Branch Prediction Buffer serves as a cache without tags
  - Does not do a good job of predicting “mostly-taken branches”
    - E.g., a loop: `for (i=0; i<10; i++) { ... }`
    - Repeated executions of the loop will result in 2 mispredictions
      - Last iteration flips T to NT
      - First iteration flips NT to T
    - So, prediction accuracy of 80%

- Can we do better?
Dynamic Branch Prediction (2): 2-bit Prediction Schemes

- Store 2 bits of information in branch history buffer
- How does this do on our loop example?
  - 1 misprediction per iteration if we start off in the (11) state
  - 1 misprediction per iteration (plus 2 mispredictions the first time) if we start in (00) state

- Generalization: n-bit saturating counters
  - Increment if taken, decrement if not
  - Predict T if value more than half, else NT
  - Not too much of a win over 2-bit counters
Prediction Accuracy of 2-bit Prediction Schemes

- SPEC89 benchmarks using a 4096-entry 2-bit prediction buffer (Pan, So, and Rameh [1992])

- Is hit-rate in the cache an issue?
Dynamic Branch Prediction (3): Correlating Branch Predictors

- 2-bit predictor uses only the recent behavior of a single branch to predict its future behavior

- Is branch direction affected by more “global” properties?

```assembly
if (aa == 2)
    aa = 0;
if (bb == 2)
    bb = 0;
if (aa != bb)
    { ... }

DSUBUI R3, R1, #2
BNEZ R3, L1 ; branch b1
DADD R1, R0, R0
L1: DSUBUI R3, R2, #2
BNEZ R3, L2 ; branch b2
DADD R2, R0, R0
L2: DSUBU R3, R1, R2
BEQZ R3, L3 ; branch b3

```

- Behavior of b3 is **correlated** with that of b1 and b2
  - if both b1 and b2 are NT, b3 will be T

- Can (how do) we predict such branches?
A (1,1) Correlating Predictor

if (d == 0)
  d = 1;
if (d == 1)
  { ... }

L1:
  DADDUI R1, R0, #1
L2:
  DADDUI R3, R1, #-1

Behavior of a 1-bit predictor for repeated executions of above with values of d=2, 0, 2, 0,…

1-bit predictor that uses 1 bit of correlation
- \( X/Y \): X if last branch was NT, Y if last branch was T

These predictions would be correct, irrespective of value of d
(m,n) Correlating Predictors

- Use behavior of the last m branches to choose from among $2^m$ n-bit predictors (for a single branch)
  - Makes the predictor context-sensitive

- Yields improved prediction accuracy for small hardware cost
  - History of last m branches can be kept as a shift register
    - Each bit records whether corresponding branch was T/NT
  - Branch prediction buffer can then be indexed by concatenating the lower-order bits of address with the m-bit history

- Variant: gshare
  - History and branch address bits are xor-ed
Prediction Accuracy of Correlating Predictors

Is this a fair comparison?

**Metric:** Number of bits

- 4096 entries (2-bit): 8K
- 1024 entries (2,2): $2^2 \times 2 \times 1024 = 8K$
Tournament Predictors

- **Adaptively** combine local and global predictors
  - Make more effective use of large numbers of prediction bits

- **Generalization**: Multilevel branch predictors
  - Several levels of branch prediction tables
    - Combination of local (with different bits/branch) and global (different m,n)
  - **Selector** algorithm that chooses which predictor to apply

- A selector should be adaptive (“learn” from its mistakes)

- **Saturating counters** as in local predictors
  - Increment/decrement based on which predictor is correct (when the other is incorrect)

0|1: Predictor 1 was wrong, Predictor 2 was correct
Performance of Tournament Predictors

- Misprediction rate on SPEC89 as total number of bits is increased
  - “optimal” correlating predictor chosen at each point
Examples of Branch Predictors in Use

- **Pentium**
  - 2-bit local predictor
  - direct jump from (00) to (11) state

- **Pentium MMX, Pentium Pro, Celeron, Pentium II**
  - (4,2) correlating predictor

- **Pentium III**
  - 2-level adaptive tournament predictor
    - details are sketchy
  - 512-entry Branch Target Buffer (BTB) (next)

- **Pentium IV**
  - 4096-entry BTB
  - Execution trace cache (later)

- **Alpha 21264** used a tournament predictor

- **Selector**
  - Uses 4K 2-bit counters indexed by the local branch address to select between local and global predictor

- **Global predictor**
  - 4K entries, indexed by history of last 12 branches
  - Each entry is a 2-bit predictor
  - (12,2) correlating predictor

- **Local predictor is itself 2-level**
  - Top-level: 1K 10-bit history of (local) branch outcomes
    - Detects patterns
  - History used to index a 1K 3-bit saturating counter table
Branch Target Buffer (BTB)

- In the classic 5-stage pipeline, an instruction is identified as a branch only in the ID stage
  - Branch prediction buffer can help decide whether to fetch from target address (fast pipeline), or fall-through
  - However, IF still ends up fetching a (possibly) useless instruction
    - So, even with perfect branch prediction, cannot achieve 0-cycle branch latency

- Solution: Branch Target Buffer (BTB)
  - Accessed during the IF stage
  - A cache that stores predicted address for the next instruction after a branch

- Variants
  - Store only predicted taken branches
    - Works for 1-bit local predictors: store entry when changing prediction to T
  - Use separate target and prediction buffers
Combining Target and Prediction Buffers

- Assuming IF only has BTB, and ID is responsible for prediction

- Branch penalties
  - Assuming new target is written into PC only at the end of EX

<table>
<thead>
<tr>
<th>BTB hit</th>
<th>T/NT?</th>
<th>Actual</th>
<th>Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>T</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>Yes</td>
<td>Mispredict or predict NT</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>T</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>NT</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

- Current-day processors combine target and prediction logic into a separate Instruction Fetch Unit
  - operates independently of the pipeline
  - Variant: Store (decoded) instructions in BTB (as opposed to just the target)
    - WHY?
Role of Branch Prediction

• Branch prediction techniques achieve 80 – 95% accuracy
  – Exact benefit varies based on program type, size of buffer
  – Crucial for current-day microprocessors
    • Need to supply multiple instructions per cycle to subsequent stages

• Can also reduce branch penalties by reducing misprediction penalties
  – Fetch from both predicted/unpredicted paths
  – Store/buffer instructions from both paths in the BTB
    • Extension of this idea: Execution Trace Cache (later)

  – Possible to do the above because of increasing transistor budgets