Reducing Branch Penalties

• Last chapter – static schemes
  – Move branch calculation earlier in pipeline
  – Static branch prediction
    • Always taken, not taken
  – Delayed branch
• This chapter – dynamic schemes
  – Loop unrolling
    • Good, but limited to loops
• A more general dynamic scheme that can be used with all branches
  – Dynamic branch prediction
Dynamic Branch Prediction

- Becomes **crucial** to any processor that tries to issue more than one instruction per cycle
  - Scoreboard, Tomasulo’s we’ve seen so far operate on a basic block (no branches)
    - Possible to extend Tomasulo’s algorithm to include branches
  - Just not enough instructions in a basic block to get the superscalar performance
- Result
  - Control dependencies can become the limiting factor
    - Hard for compilers to deal with, so may be ignored, resulting in a higher frequency of branches
  - Amdahl’s Law too
    - As CPI decreases the impact of control stalls increases

Branch Prediction Buffer

- Simplest Scheme – one bit **Branch Prediction Buffer** (BPB) aka **Branch History Table** (BHT)
- Idea
  - Take low order bits of the address of branch instruction, and store a branch prediction in the BHT
  - Can be implemented in a fashion very similar to cache

Instruction Stream

<table>
<thead>
<tr>
<th>10F00: LD R1, 1000(R0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10F04: BEQZ L1</td>
</tr>
</tbody>
</table>

BHT

<table>
<thead>
<tr>
<th>00</th>
<th>Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>04</td>
<td>Not Taken</td>
</tr>
<tr>
<td>FF</td>
<td>Taken</td>
</tr>
</tbody>
</table>

Set bit to actual result of the branch
Simple and Effective, But…

- **Aliasing Problem**
  - branches with same lower order bits will reference the same entry if we get unlucky, causing mutual prediction
  - Counter-argument: there’s no guarantee that a prediction is right so it might not matter
  - **Avoidance**
    - Same ideas as in caching
    - Make the table bigger
      - Not much of a problem since it’s only a single bit we are storing
    - Can try other cache strategies as well, like set-associative mapping

- **Shortcomings with loops**
  - always mispredict twice for every loop
    - Mispredict upon exiting a loop, since this is a surprise
    - If we repeat the loop, we’ll miss again since we’ll predict to not take the branch
  - Book example: branch taken 90% of time predicted 80% accuracy

---

Solution to Loops – N bit Prediction

- Use a finite state automata with \(2^n\) states
- Called an **n-bit** prediction
- Most common is to use 2 bits, giving 4 states
  - Example below will only miss on exit

```
T 00 Predict taken 01 Predict taken
|   | T | NT |
T 10 Predict not taken 11 Predict not taken
|   | T | NT |
NT 00 Predict taken 01 Predict taken
|   | T | NT |
```

Note: book fig 4.13 wrong
Implementation

- Separate Branch History Cache Buffer
  - associated with the IF stage (using the PC) but we don’t know if this is a branch until the ID stage
  - but IF stage knows the target address and hence the index
  - at ID if it’s a branch then the prediction goes into effect
  - This is still useful for most pipelines
  - Not useful for the improved DLX
    - branch resolution happens in ID stage for improved DLX (in EX for original DLX!)
    - so this model doesn’t improve anything for the DLX, we would need the predicted branch by the ID stage so we could be fetching it while decoding!
- Instruction cache extension
  - hold the bits with the instruction in the Instruction Cache as an early branch indicator
  - Increased cost since the bits will take up space for non-branch instructions as well

Does It Work?

SPEC89

Prediction accuracy for 4K two-bit prediction buffer
Somewhat misleading for the scientific programs (top three)
Increased Table Size

Increasing the table size helps with caching, does it help here?

We can simulate branch prediction quite easily. The answer is NO compared to an unlimited size table!

Performance bottleneck is the actual goodness of our prediction algorithm.

Improving Branch Prediction

- Let’s look at an example of the types of branches that our scheme performed poorly on
  - if (aa==2) aa=0;
  - if (bb==2) bb=0;
  - if (aa!=bb) { .... }

- If the first two branches are not taken, then we will always take the third
  - But our branch prediction scheme for the third branch is based on the prior history of the third branch only, not on the behavior of other branches!
  - Will never capture this behavior with the existing model

- Solution
  - Use a correlating predictor, or what happened on the previous (in a dynamic sense, not static sense) branch
  - Not necessarily a good predictor (consider spaghetti code)
Example – Correlating Predictor

• Consider the following code fragment
  – if (d==0) d=1;
  – if (d==1) { … }

• Typical code generated by this fragment
  – BNEZ R1, L1 ; check if d==0 B1
  – ADDI R1, R0, #1 ; d ← 1
  – L1: SEQI R3, R1, #1 ; Set R3 if R1==1
  – BNEZ R3, L2 ; Skip if d!=1 B2
  – L2: …

• Let’s look at all the possible outcomes based on the value of d going into this code

<table>
<thead>
<tr>
<th>Initial value of d</th>
<th>d==0?</th>
<th>Value of d before b2</th>
<th>d==1?</th>
<th>b2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Yes</td>
<td>Not taken</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>1</td>
<td>No</td>
<td>Taken</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>No</td>
<td>Taken</td>
<td>2</td>
<td>No</td>
</tr>
</tbody>
</table>

Any value not 0 or 1

If b1 not taken, then b2 not taken, all the time!

Worst-case sequence: all predictions fail!
Solution – Use correlator

• Use a predictor with one bit of correlation
  – i.e. we remember what happened with the last branch to predict the current branch
  – Think of this as the last branch has two separate prediction bits
    • Assuming the last branch was taken
    • Assuming the last branch was not taken

• Leads to 4 possibilities: which way the last one went chooses the prediction
• (Last-taken, last-not-taken) X (predict-taken, predict-not-taken)

<table>
<thead>
<tr>
<th>Prediction bits</th>
<th>Prediction if last branch not taken</th>
<th>Prediction if last branch taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>NT/NT</td>
<td>Not taken</td>
<td>Not taken</td>
</tr>
<tr>
<td>NT/T</td>
<td>Not taken</td>
<td>Taken</td>
</tr>
<tr>
<td>T/NT</td>
<td>Taken</td>
<td>Not taken</td>
</tr>
<tr>
<td>T/T</td>
<td>Taken</td>
<td>Taken</td>
</tr>
</tbody>
</table>

Notation a bit confusing since we have two interpretations taken/not-taken: for what really happened last branch, and prediction

Behavior using one-bit of correlation: get every branch correct!
Start in NT/NT state for both branches

<table>
<thead>
<tr>
<th>d=?</th>
<th>b1 prediction</th>
<th>b1 action</th>
<th>New b1 prediction</th>
<th>b2 prediction</th>
<th>b2 action</th>
<th>New b2 prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>NT/NT</td>
<td>T</td>
<td>T/NT</td>
<td>NT/NT</td>
<td>T</td>
<td>NT/T</td>
</tr>
<tr>
<td>0</td>
<td>T/NT</td>
<td>NT</td>
<td>T/NT</td>
<td>NT/NT</td>
<td>NT</td>
<td>NT/T</td>
</tr>
<tr>
<td>2</td>
<td>T/NT</td>
<td>T</td>
<td>T/NT</td>
<td>NT/NT</td>
<td>T</td>
<td>NT/T</td>
</tr>
<tr>
<td>0</td>
<td>T/NT</td>
<td>NT</td>
<td>T/NT</td>
<td>NT/NT</td>
<td>NT</td>
<td>NT/T</td>
</tr>
</tbody>
</table>
Predictors in General

- Previous example a (1,1) predictor
  - Used last 1 branches, with a 1 bit predictor
- Can generalize to a (m, n) predictor
  - M = number of previous last branches to consider
    - Results in $2^m$ branch predictors, each using n bits
  - Total number of bits needed
    - $2^m \times n \times \text{Number of entries selected in table}$
    - E.g. (2, 2) with 16 entries = 128 bits
      - Shown on next slide
  - Can implement in hardware without too much difficulty
    - Some shifters needed to select entries
- (2,2) and (0,2) the most interesting/common selections

(2,2) buffer with 2 bit history

![Diagram of (2,2) buffer with 2 bit history]
Performance of Predictors?

- SPEC 89 Benchmark
- Improves performance, but of course with the extra cost
- Note no improvement in first few cases, but no decrease in performance either

Big win here!

Branch Target Buffers

- How can we use branch prediction on DLX?
  - As indicated earlier, we need the branch target during the IF stage
  - But we don’t know it’s a branch until ID … stuck?
  - Solution
    - Use the address of the current instruction to determine if it is a branch! Recall we have the Branch History Buffer

Instruction Stream
10F04: BEQZ L1

Get from PC

BHT

| 00 | Taken |
| 04 | Not Taken |
| FF | Taken |

Will change and rename to Branch Target Buffer/Cache
Branch Target Buffer

- Need to change a bit from Branch History Table
  - Store the actual Predicted PC with the branch prediction for each entry
  - If an entry exists in the BTB, this lets us look up the Predicted PC during the IF stage, and then use this Predicted PC for the IF of the next instruction during the ID of the branch instruction

BTB Diagram

Only need to store Taken branches in the table
Penalties with DLX

- Branch penalties for incorrect predictions are shown below
  - No delay if prediction correct
  - Two cycle penalty if incorrect
    - One to discover the wrong branch was taken
    - One to update the buffer (might be able to overlap this with other stages)

<table>
<thead>
<tr>
<th>Instruction in buffer</th>
<th>Prediction</th>
<th>Actual branch</th>
<th>Penalty cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Taken</td>
<td>Taken</td>
<td>0</td>
</tr>
<tr>
<td>Yes</td>
<td>Taken</td>
<td>Not taken</td>
<td>2</td>
</tr>
<tr>
<td>No</td>
<td>Taken</td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

- Penalty not too bad here, but much worse for many other machines (e.g. with longer pipelines)
Other Improvements

- **Branch Folding**
  - Store the target instruction in the cache, not the address
  - We can then start executing this instruction directly instead of having to fetch it!
    - Branch “disappears” for unconditional branches
    - Will then update the PC during the EX stage
    - Used with some VLIW architectures (e.g. CRISP)
    - Increases buffer size, but removes an IF stage
    - Complicates hardware
- **Predict Indirect Jumps**
  - Dynamic Jumps, target changes
  - Used most frequently for Returns
  - Implies a stack, so we could try a stack-based prediction scheme, caching the recent return addresses
  - Can combine with folding to get Indirect Jump Folding
- **Predication**
  - Do the If and Else at the same time, select correct one later

Branch Prediction Summary

- **Hot Research Topic**
  - Tons of papers and ideas coming out on branch prediction
    - Easy to simulate
    - Easy to forget about costs
  - Motivation is real though
    - Mispredicted branches are a large bottleneck and a small percent improvement in prediction can lead to larger overall speedup
    - Intel has claimed that up to 30% of processor performance gets tossed out the window because of those 5-10% wrong branches
- **Basic concepts presented here used in one form or another in most systems today**
Example – Intel Pentium Branch Prediction

- Two level branch prediction scheme
  - 1. Four bit shift register, indicates last 4 branches
  - 2. Sixteen 2-bit counters (the FSA we saw earlier)
  - The shift register selects which of the 16 counters to use

Advantage: remembers history and can learn patterns of branches

Consider 1010 (T/NT)

History shifts:
- 1010 → 0101 → 1010 → 0101
- Update 5th and 10th counters

ILP Summary

<table>
<thead>
<tr>
<th></th>
<th>Software Solution</th>
<th>Hardware Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Hazards</strong></td>
<td>Pipeline Scheduling</td>
<td>Scoreboarding</td>
</tr>
<tr>
<td></td>
<td>Register Renaming</td>
<td>Tomasulo’s Algo</td>
</tr>
<tr>
<td><strong>Structural Hazards</strong></td>
<td>Pipeline Scheduling</td>
<td>More Functional Units</td>
</tr>
<tr>
<td><strong>Control Hazards</strong></td>
<td>Static Branch Prediction</td>
<td>Dynamic Branch Prediction /</td>
</tr>
<tr>
<td></td>
<td>Pipeline Scheduling –</td>
<td>Correlation</td>
</tr>
<tr>
<td></td>
<td>Delayed Branch</td>
<td>Branch Folding</td>
</tr>
<tr>
<td></td>
<td>Loop Unrolling</td>
<td>Predication</td>
</tr>
</tbody>
</table>