Branch Prediction

or

sometimes you just have to guess

Importance of Branch Prediction

- DLX/MIPS R2000 -- branch hazard of 1 cycle, 1 instruction issued per cycle
  - delayed branch
- next generation -- 1-2 cycle hazard, 1-2 instructions issued per cycle
  - cost of branch misprediction goes up
- Pentium Pro -- 12+ cycle misprediction penalty, 3 instructions issued per cycle
  - HUGE penalty for mispredicting a branch

2-bit branch prediction

- has 4 states instead of 2, allowing for more information about tendencies.

- Loops?
Two different 2-bit schemes

- Predict taken
- Predict not taken
- Taken
- Not taken

2-bit prediction accuracy

Branch History Table

- Has limited size
- 2 bits by N (e.g. 4K)
- Uses low bits of branch address to choose entry

Can We Do Better?

- Can we get more information dynamically than just the history of this branch?
- We can look at patterns (2-level predictor).
  - Last eight branches 00100100, then it is a good guess that the next one is "1" (taken)

Even/odd branch?
Can We Do Better?

- **Correlating Branch Predictors** also look at other branches for clues
  
  ```
  if (i == 0)
  ...
  if (i > 7)
  ...
  ```

- Typically use two indexes
  - Global history register --> history of last m branches (e.g., 0100011)
  - branch address

---

Correlating Branch Predictors

- The global history register is a shift register that records the last $n$ branches encountered by the processor.

---

Two-level correlating branch predictors

- Can use both the PC address and the GHR

---

2-level branch prediction

- If we concatenate the GHR and the PC, we get...

  - This is a (2,2) scheme (2 bits of global history, 2-bit predictors)
  - Could also XOR GHR and PC (gshare).
Performance of 2-level Correlating Branch Prediction

- 4096 entries: 2 bits per entry
- Unlimited entries: 2 bits per entry
- 1024 entries

- nasa7
- matrix300
- tomcatv
- doduc
- SPEC89 benchmarks
- spice
- fpppp
- gcc
- espresso
- eqntott
- li

Frequency of mispredictions:

0% 1%
0% 0%
0% 0%
1% 0%
5% 5%
9% 9%
5% 5%
12% 11%
18% 18%
10% 10%
6%
4%
18%
14%
16%
12%
5%
5%
14%

But...

- When do we need to do the prediction to avoid any control hazards on a correct prediction?
- A taken/not taken prediction only helps us if....?

Are we happy yet????

- Combining branch predictors use multiple schemes and a voter to decide which one typically does better for that branch.

Branch Target Buffers

- predict the location of branches in the instruction stream
- predict the destination of branches
BTB Operation

- Use PC (all bits) for lookup
  - Match implies this is a branch
- If match and predict bits => taken, set PC to predicted PC
- If branch predict wrong, must recover (same as branch hazards we’ve already seen)
  - But what about dynamically scheduled processor??
- If decode indicates branch when no BTB match, two choices:
  - Look up prediction now and act on it
  - Just predict not taken
- When branch resolved, update BTB (at least prediction bits, maybe more)

BTB Performance

- Two things that can go wrong
  - Didn’t predict the branch (misfetch)
  - Mispredicted a branch (mispredict)
- Suppose BTB hit rate of 85% and predict accuracy of 90%, misfetch penalty of 2 cycles and mispredict penalty of 5 cycles, what is average branch penalty?
- Can use both BTB with BPT
  - Have no prediction bits in BTB (why is that a good idea?)
  - Presence of PC in BTB indicates a lookup in BPT to predict whether the branch will go to destination address in BTB.

What about indirect jumps/returns?

- BPT does really well with conditional jumps
- BTB does really well with unconditional jumps (jump, jal, etc.)
- Indirect jumps often jump to different destinations, even from the same instruction. Indirect jumps most often used for return instructions.
- Return easily handled by a stack.
  - Jal -> push PC+4
  - Return -> predict jump to address on top of stack, pop stack

Real BP -- PowerPC 620

- 256-entry 2-way set-associative BTB
- 2048-entry BHT
- Return-address stack
Pentium Pro

- 512-entry BTB 4-way set-associative
- 2-level predictor (1st level in BTB, one per set, 4 bits)
- return stack

Compaq/Digital Alpha 21264

- next-cache-line field in I-cache replaces BTB
- return address stack

Branch Prediction Key Points

- The better we predict, the behinder we get.
- 2-bit predictors capture tendencies well.
- Correlating predictors improve accuracy, particularly when combined with 2-bit predictors.
- Accurate branch prediction does no good if we don’t know there was a branch to predict
- BTB identifies branches in (or before) IF stage.
- BTB combined with branch prediction table identifies branches to predict, and predicts them well.

CPI = 1.0 + BSPI + FPSPI + LdSPI

Now what?

Branch prediction attacks this
Out-of-order execution, instruction scheduling attack these
Multiple Instruction Issue

or

*The insts go marching two by two, hurrah, hurrah...*

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- **Superscalar**
  - variable number of instructions issued each cycle
  - parallelism detected in hardware

- **Very Long Instruction Words (VLIW)**
  - fixed number of instructions issued each cycle
  - parallelism scheduled by the compiler
  - IA-64 (Merced, Itanium)

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- **Superscalar DLX**: 2 instructions, 1 FP & 1 anything else
  - Fetch 64-bits/clock cycle; Int on left, FP on right
  - Can only issue 2nd instruction if 1st instruction issues

<table>
<thead>
<tr>
<th>Type</th>
<th>Pipe Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int. instruction</td>
<td>IF ID  EX  MEM  WB</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF ID  EX  MEM  WB</td>
</tr>
<tr>
<td>Int. instruction</td>
<td>IF ID  EX  MEM  WB</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF ID  EX  MEM  WB</td>
</tr>
<tr>
<td>Int. instruction</td>
<td>IF ID  EX  MEM  WB</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF ID  EX  MEM  WB</td>
</tr>
</tbody>
</table>

Superscalar DLX Implications

- More ports for FP registers to do FP load & FP op in a pair
- 1 cycle load delay expands to 3 instructions in SS
  - instruction in right half can’t use it, nor instructions in next slot
- Branch delay also expands to 3
Unrolled Loop that Minimizes Stalls for Scalar

1 Loop: LD F0,0(R1)  
2 LD F6,-8(R1)  
3 LD F10,-16(R1)  
4 LD F14,-24(R1)  
5 ADDD F4,F0,F2  
6 ADDD F8,F6,F2  
7 ADDD F12,F10,F2  
8 ADDD F16,F14,F2  
9 SD 0(R1),F4  
10 SD -8(R1),F8  
11 SD -16(R1),F12  
12 SUBI R1,R1,#32  
13 BNEZ R1,LOOP  
14 SD 8(R1),F16 ; 8-32 = -24

14 clock cycles, or 3.5 per iteration

Loop Unrolling in Superscalar

1 Loop: LD F0,0(R1)  
2 LD F6,-8(R1)  
3 LD F10,-16(R1)  
4 LD F14,-24(R1)  
5 ADDD F4,F0,F2  
6 ADDD F8,F6,F2  
7 ADDD F12,F10,F2  
8 ADDD F16,F14,F2  
9 SD 0(R1),F4  
10 SD -8(R1),F8  
11 SD -16(R1),F12  
12 SUBI R1,R1,#32  
13 BNEZ R1,LOOP  
14 SD 8(R1),F16 ; 8-32 = -24

Unrolled 5 times to avoid delays  
12 clocks, or 2.4 clocks per iteration

Dynamic Scheduling in Superscalar

● Dependencies stop instruction issue in In-order SS  
● Code compiled for scalar version will run poorly on SS  
  – May want code to vary depending on how superscalar  
● Simple approach: Combine Tomasulo with the ability to fetch  
  and issue an integer and FP instruction simultaneously  
  – simplified if we don’t issue instructions that read the same register file in  
    the same cycle.  
  – requires multiple CDBs  
  – can complicate updating of register bookkeeping if instructions are  
    dependent, but can be done

Performance of Dynamic SS

\[
\begin{array}{cccc}
\text{Iteration Instructions} & \text{Issues} & \text{Executes} & \text{Writes result} \\
\text{no.} & \text{clock-cycle number} & & \\
1 & LD F0,0(R1) & 1 & 2 & 4 \\
1 & ADDD F4,F0,F2 & 1 & 5 & 8 \\
1 & SD 0(R1),F4 & 2 & 9 & \\
1 & SUBI R1,R1,#8 & 3 & 4 & 5 \\
1 & BNEZ R1,LOOP & 4 & 5 & \\
2 & LD F0,0(R1) & 5 & 6 & 8 \\
2 & ADDD F4,F0,F2 & 5 & 9 & 12 \\
2 & SD 0(R1),F4 & 6 & 13 & \\
2 & SUBI R1,R1,#8 & 7 & 8 & 9 \\
2 & BNEZ R1,LOOP & 8 & 9 & \\
\end{array}
\]

\(\approx 4 \text{ clocks per iteration}\)

Branches, Decrements still take 1 clock cycle
Limits of Superscalar

- While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  - Exactly 50% FP operations
  - No hazards
- If more instructions issue at same time, greater difficulty of decode and issue
  - Even 2-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue

Superscalar Key Points

- Only way to get CPI < 1 is multiple instruction issue
- SS requires duplicated hardware, more dependence checking
- Without duplication of functional units, will see limited improvement
- SS combined with dynamic scheduling can be powerful