Branch Prediction

or

sometimes you just have to guess

Looking for Instruction Level Parallelism (ILP)

- We want to identify and exploit ILP – instructions that can potentially be executed at the same time.
- Branches are 15-20% of instructions
  - Implications?
    - 
    - 
- Can only keep the pipeline full if we can consistently keep fetching well past unresolved branches.
- Can only exploit high levels of parallelism if we consistently have multiple basic blocks in the processor at once.

Importance of Branch Prediction

- DLX/MIPS R2000 -- branch hazard of 1 cycle, 1 instruction issued per cycle
  - delayed branch
- next generation – 2-3 cycle hazard, 1-2 instructions issued per cycle
  - cost of branch misprediction goes up
- Pentium 4

Branch Prediction

- Easiest (static prediction)
  - always not taken, always taken
  - forward not taken, backward always taken
  - compiler predicted (branch likely, branch not likely)
- Next easiest (1-bit dynamic)
  - remember last taken/not taken per branch (1 bit)
  - per branch approximated
    - per I cache line
    - use part of address
  - what happens on a loop?

```c
for (l=0;l<5;l++) {
  ...
  ...
  beeq r1, loop;
  }
```
2-bit branch prediction

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

Loops?

Two different 2-bit schemes

Strongly Taken 11
Weakly Taken 10
Weakly Not Taken 01
Strongly Not Taken 00

Decrement when not taken
Increment when taken

Branch History Table

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

- what happens when table too small?
- what about even/odd branch?

Can We Do Better?

- Can we get more information dynamically than just the general history of this branch?
- We can look at patterns (local predictor) for a particular branch.
  - 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
  
  \[
  \begin{align*}
  \text{if} \ (i == 0) \\
  \quad \ldots \\
  \text{if} \ (i > 7) \\
  \quad \ldots 
  \end{align*}
  \]

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

Two-level correlating branch predictors

- Can use both the PC address and the GHR

Performance of 2-level Correlating Branch Prediction

- The global history register is a shift register that records the last n branches (of any address) encountered by the processor.
Are we happy yet???

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

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....?

Branch Target Buffers

- predict the **location** of branches in the instruction stream
- predict the **destination** of branches

Branch Target Buffers

- 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 10 cycles, what is average branch penalty?
- Can use both BTB and branch predictor
  - have no prediction bits in BTB (why is that a good idea?)
  - presence of PC in BTB indicates a lookup in branch predictor to predict whether the branch will go to destination address in BTB.

---

**What about indirect jumps/returns?**

- Branch predictor 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
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.