Branch statistics

- Branches occur every 4-6 instructions (16-25%) in integer programs; somewhat less frequently in scientific ones
- Unconditional branches : 20% (of branches)
- Conditional (80%)
  - 66% forward (i.e., slightly over 50% of total branches). Evenly split between Taken and Not-Taken
  - 33% backward. Almost all Taken
- Probability that a branch is taken
  - p = 0.2 + 0.8 (0.66 * 0.5 + 0.33) ≈ 0.7
  - In addition call-return are always Taken

Control hazards (branches)

- When do you know you have a branch?
  - During ID cycle
- When do you know if the branch is Taken or Not-Taken
  - During EXE cycle (e.g., for the MIPS)
- Easiest solution
  - Wait till outcome of the branch is known
- Need for more sophisticated solutions because
  - Modern pipelines are deep (several stages between ID and EXE)
  - Several instructions issued/cycle (compounds the "number of issue instruction slots" being lost)

Penalty for easiest solution

- Simple single pipeline machine with 5 stages
  - Stall is 2 cycles hence
  - Contribution to CPI due to branches

    2 x Branch freq. ≈ 2 * 0.20 = 0.4
- Modern machine with about 20 stages and 4 instructions issued/cycle
  - Stall would be, say, 12 cycles
  - Loss in "instruction issue slots" = 12 * 4 = 48 … and this would happen every 4-6 instructions!!!!!!!!

Simple schemes to handle branches

- Techniques that could work for CPU's with a single pipeline with few stages:
  - Comparison could be done during ID stage: cost 1 cycle only
    - Need more extensive forwarding plus fast comparison
    - Still might have to stall an extra cycle (like for loads)
  - Branch delay slots filled by compiler
    - Not practical for deep pipelines
- Predictions are required
  - Static schemes (only software)
  - Dynamic schemes: hardware assists

Simple static predictive schemes

- Predict branch Not -Taken
  - If prediction correct no problem
  - If prediction incorrect, and this is known during EXE cycle, zero out (flush) the pipeline registers of the already fetched instructions following the branch (the number of fetched inst.

    delay = number of stages between ID and EXE
  - With this technique, contribution to CPI due to cond. branches:

    0.20 * (0.7 * *delay* + 0.3 * 0)

    (e.g., if delay =2 (10), yields 0.28 (1.40))
  - The problem is that we are optimizing for the less frequent case; but it will be the "default" for dynamic branch prediction since it is so easy to implement.

Static schemes (c'ed)

- Predict branch Taken
  - Interesting only if target address can be computed **before** decision is known
  - With this technique, contribution to CPI due to cond. branches:

    0.20 * (0.7 * 1 + 0.3 * *delay*)

    (e.g., if delay =2 (10), yields 0.26 (0.74))
  - The 1 is there because you need to compute the branch address
  - Better than Predict Not-taken and increasingly relatively better as *delay* increases

Static schemes (c'ed)

- Prediction depends on the direction of branch
  - Backward-Taken-Forward-Not-Taken (BTFNT)
    - Rationale: Backward branches at end of loops: mostly taken; Forward branches : we can assume 50-50 or maybe better 45%T and 55% NT because branches forward for exceptions are seldom taken.
  - Contribution to CPI due to cond. branches more complex
    - Need to reconcile % of B vs F, and T vs NT. Assume 33% of B all T; This leaves 66% of T with approx. 45% T and 55% NT.
      But we need to now the accuracy of the prediction. When pred is correct we pay 1 for T and 0 for NT; when incorrect we pay *delay*
      0.20 (0.33 * 1 + 0.66 * [0.45 *acc* + (1-*acc*) * *delay*])(the first term corresponds to BT and the next two to FT and FNT resp.)
      (e.g., if *acc* = 0.8 and *delay* =2 (10), yields 0.18 (0.39))

Dynamic branch prediction

- Execution of a branch requires knowledge of:
  - There is a branch but one can surmise that every instruction is a branch for the purpose of guessing whether it will be taken or not taken (i.e., prediction can be done at IF stage)
  - Whether the branch is Taken/Not-Taken (hence a branch prediction mechanism)
  - If the branch is taken what is the target address (can be computed but can also be "precomputed", i.e., stored in some table)
  - If the branch is taken what is the instruction at the branch target address (saves the fetch cycle for that instruction)

Basic idea

- Use a Branch Prediction Buffer (BPB)
  - Also called Branch Prediction Table (BPT), Branch History Table (BHT)
  - Records previous outcomes of the branch instruction
  - How it will be indexed, updated etc. see later
- A prediction using BPB is attempted when the branch instruction is fetched (IF stage or equivalent)
- It is acted upon during ID stage (when we know we have a branch)

Prediction Outcomes

- Has a prediction been made (Y/N)
  - If not use default "Not Taken"
- Is it correct or incorrect
- Four cases:
  - Case 1: Yes and the prediction was correct (known at EXE stage)
  - Case 2: Yes and the prediction was incorrect
  - Case 3: No but the default prediction (NT) was correct
  - Case 4: No and the default condition (NT) was incorrect

Penalties (Predict/Actual)

In what's below the number of stall cycles (*delay*) would be 2 for a simple pipe. It would be larger for deeper pipes.

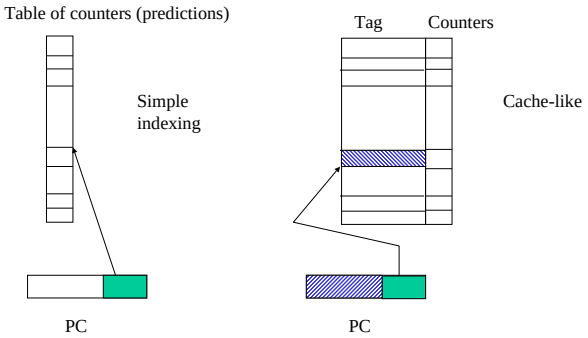
- Case 1:
  - NT/NT no penalty
  - T/T need to compute address: 1 bubble (but can be 0; see later)
- Case 2
  - NT/T *delay*
  - T/NT *delay*
- Case 3:
  - NT/NT 0 penalty
- Case 4:
  - NT/T *delay*

Note: This assumes that the target (or next sequential) address is always computed and available in case of a mispredict

Branch Prediction Buffers

- Branch Prediction Buffer (BPB)
  - How addressed (low-order bits of PC, hashing, cache-like)
  - How much history in the prediction (1-bit, 2-bits, n-bits)
  - Where is it stored (in a separate table, associated with the I-cache)
- Correlated branch prediction
  - 2-level prediction (keeps track of other branches)
- Branch Target Buffers (BTB)
  - BPB + address of target instruction (+ target instruction -- not implemented in current micros as far as I know--)
- Hybrid predictors
  - Choose dynamically the best among 2 predictors

Variations on BPB design

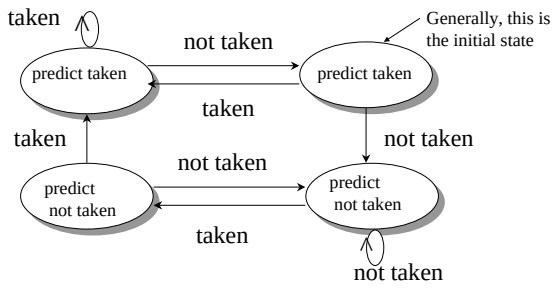


Simplest design

- BPB addressed by lower bits of the PC
- One bit prediction
  - Prediction = direction of the last time the branch was executed
  - Will mispredict at first and last iterations of a loop
- Known implementation
  - Alpha 21064. The 1-bit table is associated with an I-cache line, one bit per line (4 instructions)

Improve prediction accuracy (2-bit saturating counter scheme)

Property: takes two wrong predictions before it changes T to NT (and vice-versa)



Two bit saturating counters

- 2 bits scheme used in:
  - Alpha 21164, UltraSparc, Pentium, Power PC 604 and 620 with variations, MIPS R10000 etc...
- PA-8000 uses a variation
  - Majority of the last 3 outcomes (no need to update; just a shift register)
- Why not 3 bit (8 states) saturating counters?
  - Performance studies show it's not that worthwhile although it is present in the Alpha 21264

Where to put the BPB

- Associated with I-cache lines
  - 1 counter/instruction: Alpha 21164
  - 2 counters/cache line (1 for every 2 instructions) : UltraSparc
  - 1 counter/cache line (AMD K5)
- Separate table with cache-like tags
  - direct mapped : 512 entries (MIPS R10000), 1K entries (Sparc 64), 2K + BTB (PowerPC 620)
  - 4-way set-associative: 256 entries BTB (Pentium)
  - 4-way set-associative: 512 entries BTB + "2-level"(Pentium Pro)

Performance of BPB's

- Prediction accuracy is only one of several metrics
- Others metrics:
  - Need to take into account branch frequencies
  - Need to take into account penalties for
    - Misfetch (correct prediction but time to compute the address; e.g. for unconditional branches or T/T if no Branch Target Buffer (BTB); see in a few slides)
    - Mispredict (incorrect branch prediction)
  - These penalties might need to be multiplied by the number of instructions that could have been issued

Prediction accuracy

- 2-bit vs. 1-bit
  - Significant gain: approx. 92% vs. 85% for f-p in Spec benchmarks, 90% vs. 80% in *gcc* but about 88% for both in *compress*
- Table size and organization
  - The larger the table, the better (in general) but seems to max out at about 1K entries
  - Larger associativity also improves accuracy (in general)